

# **AN INTRODUCTION TO R STUDIO FOR ANALYZING DATA**

**Episode 1: What is this all about?**

**Episode 2: Programming 101**

**Episode 3: R Studio Open House**

**Episode 4: Data In, Data Out**

**Episode 5: Plot twists**

**Episode 6: Significantly yours!**



# Episode 1 - What is this all about?

In this tutorial, we will look at how to analyze data for research purposes using R Studio. There will be code, analogies, bad jokes, and (too) many cat pictures. If you are ok with that, welcome aboard ! Fasten your seat belt because we are going for a ride. :)

## *Why should we use R?*

Several reasons motivated the use of R in this course.

### **Applying concepts on real data**

- ➔ Applying the theoretical research and statistical concepts to real data should help you: 1. to have a better understanding of them, 2. to learn when to use them and how they are relevant. The idea is trying to make research and statistics more concrete to you. Imagine you want to learn how to paint. Talking about painting techniques for many lectures is relevant for sure, but nothing like taking a brush and some paint and trying to do it yourself!
- ➔ Having a more practical view on how things are done in real research also shows some problems that are sometimes not mentioned in statistics textbooks, but play a major role in actual research such as missing data, outlier values...

### **Advantages of programming and R**

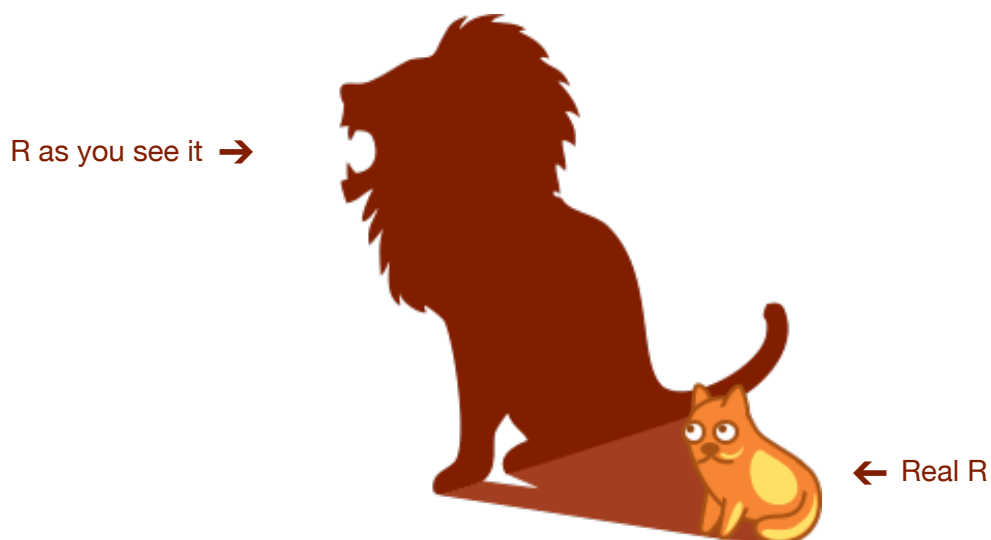
- ➔ Why are we annoying you with R when there are softwares like SPSS? Well, it is a good thing to have a quick introduction to programming as it is widely used in many fields. We are living in a period where data are central and having a rough idea of how to analyze them will be advantageous for you.
- ➔ R and R Studio are free and open source. Everyone can download it and contribute to it. On the contrary, SPSS is expensive and has to be bought if you are not a McGill student anymore. While SPSS is still currently used in research, more and more people move towards R or Python (another programming language) to run their analyses.
- ➔ Statistical softwares like SPSS are easier at first sight, but they are much less flexible. Indeed, R has several packages (add-ons) adapted to diverse analyses or research methodologies. Moreover, R is more than just statistics, you can do many things that SPSS or Excel would only dream of doing regarding data collection, types of graphs or new statistical analyses etc.
- ➔ In the long run R is more efficient. When you run the same analyses repeatedly, in SPSS or Excel you would have to click many times in different menus for each one of your analyses. A script makes it more straightforward. Moreover, you can share scripts with other researchers which improves research transparency and replicability while you cannot really share where you have clicked and all the parameters you have entered in SPSS in a direct and easy way.

## It's easier than it looks and it will be made at your reach

- ➔ We are aware that the R Studio interface and looking at code can be a bit frightening at the beginning, especially if you have no past programming experience. We also know that you are not future programmers, or full-time researchers, so the goal will not be to show you every little detail about R. We just want you to learn the basics of real data analyses and we will prepare scripts for you that you will just have to use and adapt in a very direct way. We will mask the complexity of technical details. It is more scary than difficult.

## *What are we gonna do during our R journey?*

1. Grasp the very basic concepts of programming
2. Discover the R Studio Interface
3. Learn how to set things up when you start R Studio
4. Import data into R from a file on your computer
5. Save data created in R to a file on your computer
6. Compute basic values such as the mean of a variable contained in a dataset
7. Create subsets of data
8. Create various types of plots to look at your data
9. Run various statistical tests




# Episode 2 - Programming 101

*“Programming is so awful, it’s really hard to understand,  
it’s really not for me and I should just give up...”*



We know that your brain might be telling you that. But your brain is not always right! Remember the first time you rode a bike!

Let’s go for a little overview of the very basic concepts of programming. 

## *Commands, Scripts & Comments*

When you do anything on a computer, there is a command behind it:

**A command is just some code to execute a simple action.**

Commands can be typed directly in a console<sup>1</sup> or it can be written in scripts.

A **script** is exactly like a recipe or a play: it’s a bunch of successive actions written together and that can be used multiple times. In R, it will be a succession of commands and comments to read files, make plots, run statistical tests... Writing scripts has the advantage of keeping track of the commands you used and reuse them later.

In a script, **comments** are like special considerations or tips in a recipe, or like stage directions in a play. When actors play, they do not say them. Well, for comments it’s the same : our favourite actor R will just ignore our comments starting by a hash #. In a more programming style, we would say that R will not execute anything that comes after an hash #. But for us, the authors, comments are very important. They help to organize our code but also to understand quickly what each command does, which is especially useful when you work with colleagues or when you look at your code several weeks later, as sometimes commands can be long and hard to read.

```
# Adding the numbers 4 and 5  
4+5  
4+5 # Adding the numbers 4 and 5
```

>> *On the first line the comment covers the whole line, while on the third line it follows a command.*

---

<sup>1</sup> A console is just like a messaging app, where you would talk directly with the computer: you ask something and it answers. We will see later where the R console is.

## Variable

A variable is used to store some simple or complex data in a human readable label : it has a name and it has a value (a bit like a box containing some object with a specific label on it). We use variables to store different types of objects: single values (e.g. 2 or 'Montreal'), collections of values (e.g. lists), or datasets that we can then use multiple times by just calling their name.

Let's say you want to call your Mom to share your enthusiasm about R. 🦄 You pick up your phone, go to your contact list, and just click on Mom to start the call. Here **Mom** is a variable with the label '**Mom**' and the value '**514-123-4567**'. You could then see your contact list as a list of variables. If your Mom gets a new number, then you would assign a new value to the variable **Mom** (a.k.a. as updating her number).

### Some conventions on a variable's name :

- ▶ Usually, the name of a variable does not begin with a capital letter.
- ▶ It's important to give a precise name to your variables<sup>2</sup>. You know exactly what they mean now but imagine in 6 months, 2 years..., imagine your colleagues. They have to be precise but not too long either. The challenge is to find the right balance between calling your data myData or dataFromProjectOnBilingualInfantsOct2018...
- ▶ It needs to be read easily. If it's made of multiple words, you should separate different words with an underline \_ (NOT a dash - !!!), or a dot or start each word with a capital letter, such as in: **multiple\_words\_variable**, or **multiple.words.variable**, or **multipleWordsVariable**.

## How to assign a value to a variable in R

In R, the arrow `<—` is used to store a value in a variable (it is the R version of '=')

### >> Assigning a value to a variable

Let's imagine we have the **variable** `celsius_temperature`. We will assign it the **value** 200. The following R command does just that.

```
celsius_temperature <- 200
```

>>> *It's like storing the value 200 in our box labeled 'celsius\_temperature'.*

**We can store a single value, but we can store much more as a dataframe (as table of data) or the result of a function. Let's look at a few more things.**

<sup>2</sup> You can find the reference to many 'Software Horror Stories' at this link <https://www.cs.tau.ac.il/~nachumd/horror.html>. A bunch of them involve variables that have been poorly named...

## 1 >> Updating a variable with a new single value

```
celsius_temperature <- 220
```

> The new value (220) is assigned to the variable *celsius\_temperature* and replaces the previous value (200)

## 2 >> Adding two numbers and storing the result

```
celsius_temperature <- 200 + 20
```

> Summing 200 and 20 with the sign '+' and assigning the result to the variable *celsius\_temperature*

## 3 >> Updating a variable's value using its previous value

```
celsius_temperature <- celsius_temperature + 20
```

> 20 is added to the previous value of *celsius\_temperature* (200) and stored as the new value of *celsius\_temperature*

## 4 >> Storing the result of a function in a variable

```
celsius_temperature <- sum(200,20)
```

> Here it's similar to example 2 but using the function *sum* and assigning the result to *celsius\_temperature*

You might wonder what functions are. All I can say is that it's the topic of our next section :)

## *Function & argument ~ food processor & ingredients*

To execute our simple actions, we will use functions. Functions are like a kitchen appliance. It has a name (that you have not chosen) and you can use it by feeding it with something - in programming we call that **an argument**. Using **a function with some arguments** usually produces some **output**. Let's consider some examples.

- ➔ Let's say you open the file 'MySecretStatisticalDream.txt' on your computer (because we all know you have such a file on your computer). When you double-click on the icon, behind the magical stage of your graphical interface, your computer runs **the function open** with **the argument 'MySecretStatisticalDream.txt'**. Your **output** is a new window displaying your file. Written with a programming style, it gives something like: `open('MySecretStatisticalDream.txt')`<sup>3</sup>
- ➔ When you deleted the file 'EmbarrassingPicture.jpg', your computer called the function **delete** and applied it to your file's name : `delete('EmbarrassingPicture.jpg')`

---

<sup>3</sup> In reality it's obviously more complex than that, but who cares about reality these days...

- ➔ When you push the power button of your computer, it calls the command **start()**

## >>> how to use functions in R

The basic notation for a function is:

```
functionName(argument1, argument2, argument3)
```

- ▶ Note that there is no space between functionName and the parenthesis, it's important!
- ▶ Various arguments are separated by commas
- ▶ If the function produces an output it can be stored in a variable, as below, otherwise the output will be displayed in the console<sup>4</sup>

```
output <- function(argument1, argument2, argument3)
```

### Some analogies to grasp concepts of variable, function and argument

- ▶ how to assign the **value** 'pizza' (between quotes) to the **variable** meal\_to\_be\_cooked (no quotes)

```
meal_to_be_cooked <- 'pizza'
```

- ▶ the function **oven** has 2 arguments and 1 output.

- ▶ We can use our variable **meal\_to\_be\_cooked** (if we have executed the line above) and put it as an argument in the function. Note that there is no quotes for a variable's name.

```
warm_meal <- oven(celsius_temperature, meal_to_be_cooked)
```

- ▶ We can also directly give our function oven the raw value 'frozen\_pizza' (between quotes). Here the result would be exactly the same than before. In both cases, the value of the variable **warm\_meal** could be something like 'warm\_pizza'.

```
warm_meal <- oven(celsius_temperature, 'frozen_pizza')
```

- ▶ the function **mixer** has 4 arguments and 1 output

```
smoothie <- mixer('strawberry', 'banana', 'almond_milk', 'orange_juice')
```

- ▶ the function **feel\_confident\_to\_cook** has 3 arguments and no output

```
is_able_to_cook(has_skills, has_energy, has_time)
```

<sup>4</sup> Again, we will see in the next episode where the console is!



## >>> some real R functions

We have already seen the function **sum** earlier (on page 7). Here it has 2 arguments and the result (220) is stored in the variable `celsius_temperature`.

```
celsius_temperature <- sum(200,20)
```

We will introduce several functions later on. Let's just see now the function **print** which takes one argument and will display it in the console. It is useful to display some messages when using very long scripts.

```
print("Hello you! You look so pretty today :)")
```

### A little summary

Scripts are lists of commands and comments. A command is an action, this action being most of the time represented by a function applied to some arguments. These arguments can be raw values or variables. If a function produces an output, it can be stored in a variable or displayed in the console.

It's a lot of concepts and vocabulary that you do not need to master immediately. You will certainly end up coming back several times to this chapter.


Before starting our next section, take a (long) moment to relax



### In the next episode ...

A little overview of what it would look like if we typed in the R console some commands we have mentioned in this episode.



```
Console ~/     
> celsius_temperature <- 200  
> celsius_temperature  
[1] 200  
> celsius_temperature <- sum(200,20)  
> celsius_temperature  
[1] 220  
> print("Hello you! You look so pretty today :) ")  
[1] "Hello you! You look so pretty today :) "  
> |
```

# Episode 3 - R Studio Open House

"Ok, I think I get the basic concepts of programming. But how do I use them? What are the different panels in R Studio? Where to click? What do I have to set up before I start using R?"




In this chapter, we will first install R and R Studio. Then we will have a quick tour of R Studio so that you can feel at home. Just follow the guide :)

## Install R and R Studio

R is like the motor of our statistical spaceship while R studio would be the cockpit with many little buttons and monitors to properly pilot our statistical analyses.

### 1. Install R: go to <https://cran.r-project.org/>

- Click on *Download R for Linux / MacOS X / Windows* depending on your operating system
  - For **MacOSX** : Download the file 'R-3.5.1.pkg' in the section **latest release**
  - For **Windows**: Click on 'install R for the first time' and then 'Download R 3.5.1 for Windows'
  - For **Linux** : select your Linux distribution and follow the associated instructions



**CRAN**  
[Mirrors](#)  
[What's new?](#)  
[Task Views](#)  
[Search](#)

**About R**  
[R Homepage](#)  
[The R Journal](#)

**Software**  
[R Sources](#)  
[R Binaries](#)  
[Packages](#)  
[Other](#)

**Documentation**  
[Manuals](#)  
[FAQs](#)  
[Contributed](#)

**The Comprehensive R Archive Network**

---

**Download and Install R**

Precompiled binary distributions of the base system and contributed packages, **Windows and Mac** users most likely want one of these versions of R:

- [Download R for Linux](#)
- [Download R for \(Mac\) OS X](#)
- [Download R for Windows](#)

R is part of many Linux distributions, you should check with your Linux package management system in addition to the link above.

---

**Source Code for all Platforms**

Windows and Mac users most likely want to download the precompiled binaries listed in the upper box, not the source code. The sources have to be compiled before you can use them. If you do not know what this means, you probably do not want to do it!

- The latest release (2018-07-02, Feather Spray) [R-3.5.1.tar.gz](#), read [what's new](#) in the latest version.
- Sources of [R alpha and beta releases](#) (daily snapshots, created only in time periods before a planned release).
- Daily snapshots of current patched and development versions are [available here](#). Please read about [new features and bug fixes](#) before filing corresponding feature requests or bug reports.
- Source code of older versions of R is [available here](#).

**2. Install R Studio:** go to <https://www.rstudio.com/products/rstudio/download/>

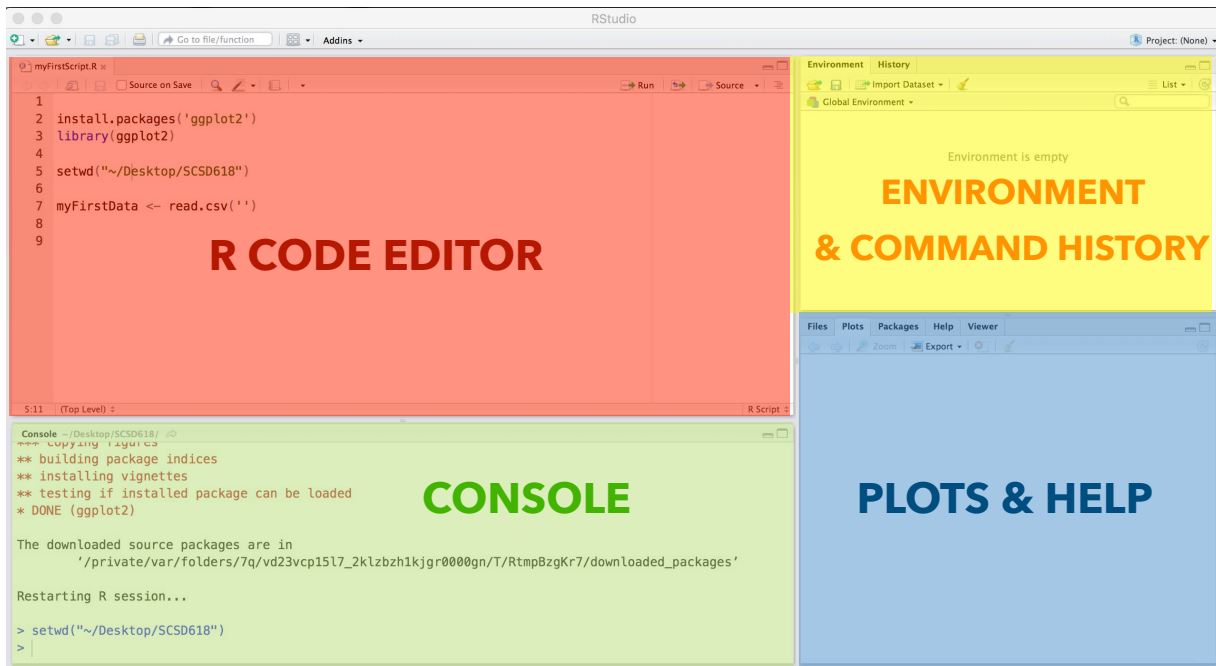
- Click on Download in the leftmost column FREE version of RStudio Desktop
- Then download the installer adapted to your operating system

The screenshot shows the RStudio website's 'Installers for Supported Platforms' page. It features a navigation bar with links for Products, Resources, Pricing, About Us, and Blogs. Below the navigation bar is a table listing various installers for different operating systems and architectures, including Windows, Mac OS X, Ubuntu, and Fedora. Each row in the table provides the installer name, its size, the release date, and its MD5 hash.

Installers	Size	Date	MD5
RStudio 1.1.456 - Windows Vista/7/8/10	85.8 MB	2018-07-19	24ca3fe0dad8187aab4bfb9dc2b5ad
RStudio 1.1.456 - Mac OS X 10.6+ (64-bit)	74.5 MB	2018-07-19	4fc4f4f70845b142bf96dc1a5b1dc556
RStudio 1.1.456 - Ubuntu 12.04-15.10/Debian 8 (32-bit)	89.3 MB	2018-07-19	3493f9d5839e3a3d697f40b7bb1ce961
RStudio 1.1.456 - Ubuntu 12.04-15.10/Debian 8 (64-bit)	97.4 MB	2018-07-19	863ae806120358fa0146e4d14cd75be4
RStudio 1.1.456 - Ubuntu 16.04+/Debian 9+ (64-bit)	64.9 MB	2018-07-19	d96e63548c2add890bac633bdb883f32
RStudio 1.1.456 - Fedora 19+/RedHat 7+/openSUSE 13.1+ (32-bit)	88.1 MB	2018-07-19	1df56c7cd80e2634f8a9fdd11ca1fb2d
RStudio 1.1.456 - Fedora 19+/RedHat 7+/openSUSE 13.1+ (64-bit)	90.6 MB	2018-07-19	5e77094a88fdbdddb0d35708752462

*A little tour of R Studio*

Let's look at our new cockpit. We have 4 main panels.



Let's have a closer look at each window !

## >>> Console

If we were cooking, it would be the countertop, where the action happens! The console is the R window where commands are executed. It is like a dialogue window: if you type a command and press **Enter**, R executes your command, and can return an output in the console.

If we type the commands directly in the console it will look like that:



```

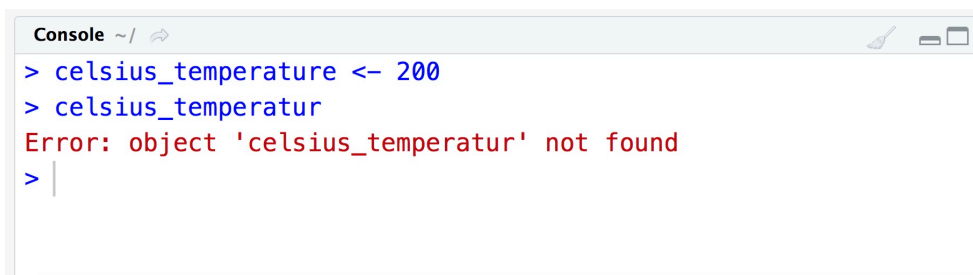
Console ~/
> celsius_temperature <- 200
> celsius_temperature
[1] 200
> |

```

- Note that presence of the > on line 1, 2 and 4. It is called a prompt. It indicates a new line waiting for a command. I will usually never show it in the code examples.
- On the first line, we typed a command to assign the value 200 to the variable **celsius\_temperature**. We can also say that we declare the variable **celsius\_temperature** with the value 200.
- R executes our command (it creates a variable **celsius\_temperature** and assigns the value 200) but does not tell us anything in return. R can work quietly.
- On line 2, we just typed the name of our variable **celsius\_temperature**, it's a way to ask R "Hey R, what is the value of **celsius\_temperature** pleaaaase?".
- And R kindly answers on line 3: it gives us one variable ([1]) with the value 200. It is its own way of saying "Hey you, the value of **celsius\_temperature** is 200. Cheers"
- Thank you R!
- On the last line, R give us again a prompt to enter a new command

## Errors & Warnings

If something goes wrong, R will indicate it to us using a red colour. There are two types of problems : **warnings** means that R was able to execute your command, but it noticed that some things might be wrong, so it just gives you a heads-up. On the contrary, **errors** are when R was not able to execute your command. As Amy Winehouse would say, let's go back to black (font). For example, below, I made a mistake and typed **celsius\_temperatur** instead of **celsius\_temperature**. R reports an error: it cannot tell me what is the value of the variable **celsius\_temperatur** because it did not find any object named **celsius\_temperatur**.



```

Console ~/
> celsius_temperature <- 200
> celsius_temperatur
Error: object 'celsius_temperatur' not found
> |

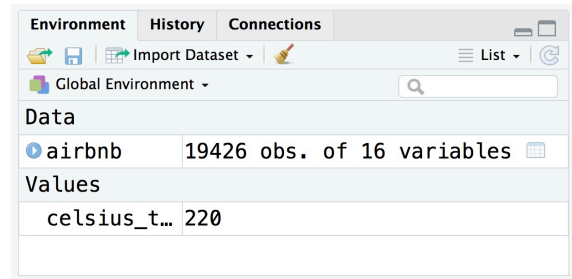
```

## >>> Environment

Each time we declare a variable or a dataframe, it is stored in the “environment”. The environment is a bit like an active memory. Unless you save it, it will be lost when you close R Studio.

The environment window enables us to have a list of all the relevant elements that are currently active in the R memory.

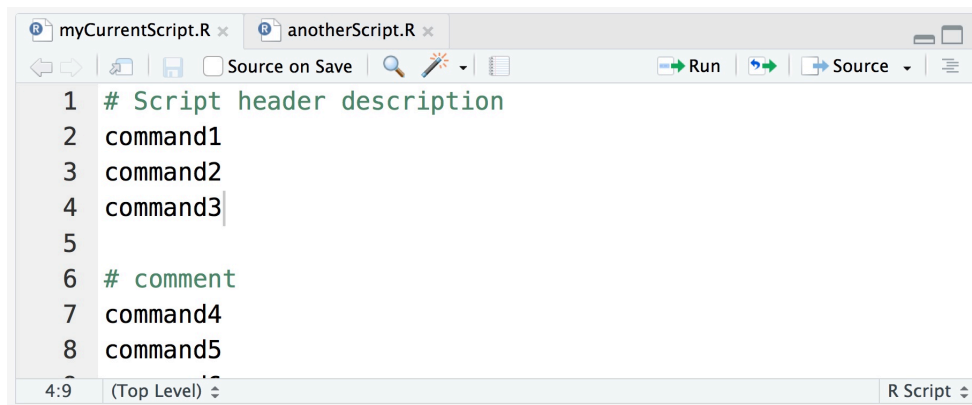
It is divided in two sections: **Data** for the dataframes (tables), and **Values** for the variables with other formats (single values, vectors, lists...). In the screenshot above, there is one dataframe (**airbnb**) and one value (**celsius\_temperature**).



## >>> History

We can browse all the commands we have entered in the console since we have started our current R session by clicking on the History tab.

## >>> Editor



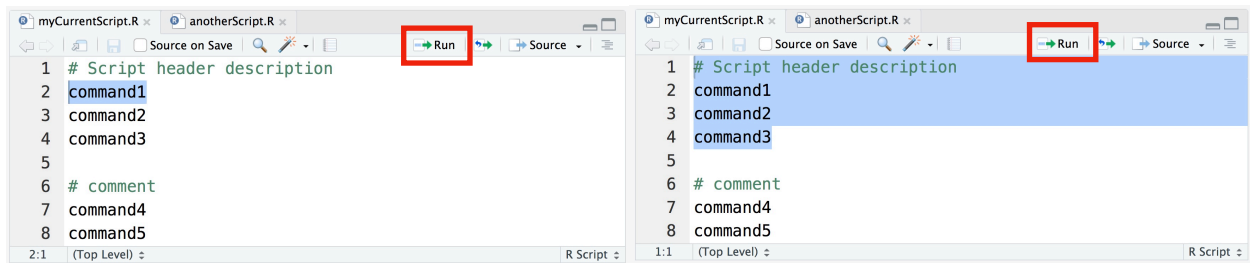
You can think of the editor as a notepad document. It is where you can write scripts and save them. You can have different tabs for each of your current open scripts (here we have a script named myCurrentScript.R and a second script named anotherScript.R). Line numbers are indicated on the left of the panel. The bottom-left corner of the panel indicates the position of the cursor (here line 4 column 9). R Studio will by default colour the comments in green.


## Running a script, or a section of a script, from the editor

There are several ways to execute commands in R:

1. you type it directly in the console window and then press Enter (as we have seen earlier)
2. you can select some commands that are written in a script, press '**cmd+Enter**' ('ctrl+Enter for Windows) and it will be executed in the console. Instead of pressing '**cmd+Enter**', you can also press the button 'Run' on the top-right of the Editor panel.

The example on the left screenshot would execute only command1 while the one on the right screenshot would execute command1, command2 and command3. The first line would be recognized as a comment by R and therefore it would not be executed.

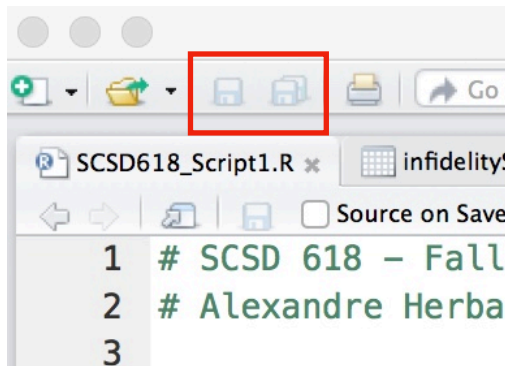


3. you can execute all the commands that are in a script (= running the whole script) by clicking on the button  on the top right of the editor. In our previous example, it would execute all commands (from command1 to command5)

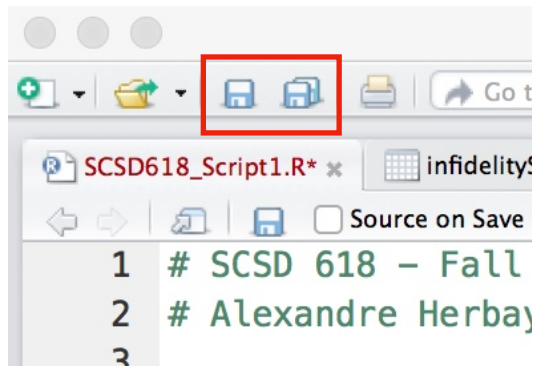
## Saving a script

A script is like a Word document, if you want your changes to be saved, don't forget to save it! An unsaved script will appear in red followed by an asterisk. A saved script will be in black.

> Saved script



> Unsaved script



## >>> Plots

In the bottom-right panel of the R Studio window (see page 12), in the **Plots** tab, you will be able to look at the plots you have created. R Studio will only display the last created plot, but you can access previous plots that are still in memory using the little arrows.

## >>> Help

In the same window, in the tab **Help**, you also have access to a manual for each function. If you want to learn more about a function just type `? followed by the name of the function.`

```
?sum or help("sum")
```

## Starting the R spaceship

*"Ok, nice tour, thank you. But now what should I do to start to do anything useful?"*

Well, thanks for asking. There are actually a few things to set up before leaving.

### >>> Set up your working directory

**Ok, but what is a working directory ??**

It's like our home folder. R can read a file anywhere on your computer if you give its exact location (~the file address, that we call a *path* in computer science). However, if R only receives a file's name, it will only look for this file in the current working directory. Same thing when writing a file: by default it will be created in the working directory. Defining a working directory makes it easier to read and write files in an organized manner.

**There are 2 methods to define the working directory:**

#### >> By clicking in R Studio menus

- Click on the menu **Session** then **Set Working Directory** then **Choose Directory**
- then select **the folder** that will become your working directory

#### >> With a command

##### >> function `setwd`

*(setwd = set working directory)*

```
setwd(working_directory_path)
```

##### **Mandatory Input :**

- ▶ the variable `working_directory_path`. Its value is the path to your working directory

##### **Output :**

- ▶ there is no output for this function



- ▶ for Mac OS users :

```
setwd("~/Desktop/SCSD618")
```

- ▶ for Windows users :

```
setwd("C:/Users/YOUR WINDOWS USER NAME/Desktop/SCSD618")
```

>>> change YOUR WINDOWS USER NAME above with your actual windows user name

> **The working directory should be checked (and potentially set up) each time you start R.**

The lazy cat tip 

*You can first set up your working directory using the menus and you will get the command in your console once it's done! You just have to copy this command for your future scripts.*

### >>> Get the current working directory

```
getwd()
```

> Just type the function `getwd()` in the console, and R will indicate what is your current working directory,

**Note that the working directory is also indicated at the top of the console window.**



### ANNND ACTION !



- For this class, create a folder on your desktop named 'SCSD618'
- Set up this folder as our working directory

## Let's go to the (code) library!

When we cook, we are not inventing all the recipes ourselves, we use cook books. The same way when programming, we will use packages (~ cookbooks) that are a collection of useful functions already prepared for us.

We need to : 1. install a package on our computer (you just have to do it once!)  
2. load it in our current environment (each time you restart R)

### >>> Install a package

*When you order a cookbook on the bookstore's website, they deliver it to you in a nice package.*

When you use the function **install.packages**, R goes online on the R website (named the CRAN) and looks for a package with the name you gave it. If R finds your package, it will be downloaded and installed on your computer. See packages as softwares, you just have to install it once, not every time you want to use it.

- ▶ The syntax to install a package is : `install.packages('package_name')`

Note that the package name has to be between quotes

In this example we install **ggplot2**, the most popular R package/library to create graphs.

```
install.packages('ggplot2')
```

### >>> Load a library

Once a software is installed on your computer, you still have to double-click to launch it when you want to use it. Here, if we want to use a function from a library that is already installed, we still need to load this library in our environment.

- ▶ The syntax to load a library in memory is : `library(library_name)`

Note that the name of the library IS NOT between quotes

```
library(ggplot2)
```

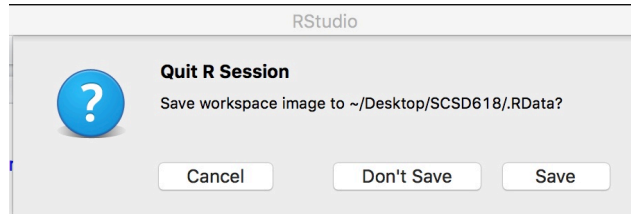
### ANNND ACTION !



- Install the package **ggplot2** on your computer
- Load the library **ggplot2** in your environment

## *Shutting down the R spaceship*

When you quit R, you will often have the pop-up below. Usually, you should choose **Save** ! It will save your environment (all the variables that appeared in the top-right window of R Studio) in a file name `.RData` located in your working directory (it's a hidden file, so usually you won't see it directly when browsing your folder).



# Episode 4 - Data In, Data Out

“Data are to R what food is to cats: the cornerstone of their existence”



## Importing Data into R

We will never enter all our data manually into R. We will rather use datasets that have been stored in an excel or a text file, and import them in our R environment.

### >>> Import data from CSV files

A CSV<sup>5</sup> file is like an excel file without any fancy formatting or formula. It's a table of raw values. To import our CSV file, we will use the function **read.csv**

#### >> function read.csv

```
dataframe_in_R <- read.csv(file_to_read, sep = ',', header = TRUE)
```

**Input :** (the star indicates that the argument is not mandatory)

- ▶ the variable **file\_to\_read** containing the name of the CSV file to read. This name will be indicated between quotes and include the extension .csv. Moreover this file should be in your working directory.
- ▶ \*a value for the argument **sep** (that stands for separator) : how the file separates values that are in different columns : here it uses commas, but it can also be semicolon (;) or a tabulation ('\t').
- ▶ \*a value for the argument **header** : if the first line of the CSV file contains column's name, you would put the value TRUE. Otherwise, if it is data, you would put FALSE.

#### **Output :**

- a dataframe in R named **dataframe\_in\_R** including all data from **file\_to\_read**

---

<sup>5</sup> CSV stands for Comma Separated Value : it's a table with raw values (formatting). Usually, the way to encode that 2 values are in different columns is to put a comma between them.

Let's say we want to import the file **'secretData.csv'**, that is saved in my working directory, and to store the data it contains in a dataframe named **secretData**:

```
secretData <- read.csv('secretData.csv', sep = ',', header = TRUE)
```

## Look at your Data in R

Click on **secretData** in the section Data in the Environment window (top-right panel)  
Or you can also enter the following command

```
View(secretData)
```

If data are stored in a dataframe, each column is a **variable**, and each line is an **observation**.

	variable1	variable2	...	variableN
observation1	value11	value12	...	value1N
observation2	value21	value22	....	value2N
...	...	....	...	...
observationM	valueM1	valueM2	...	valueMN

>> Getting a **variable** (column) in a **dataframe**

```
dataframeName$variableName6
```

>> Getting a **observation** (row) in a **dataframe**

```
dataframeName[observationNumber, ]
```

>> Getting a specific **variable** (column) of a specific **observation** (row) in a **dataframe**

```
dataframeName[observationNumber, 'variableName']
```

### ANNND ACTION !



- Make sure that **airbnb\_mtl.csv** is placed in the working directory
- Import in R the data contained in the file **airbnb\_mtl.csv**
- Look at the table of data in the R Studio interface

<sup>6</sup> or less direct: `dataframeName[ , 'variableName']`

## Subsetting Data

Sometimes, we will want to operate on only a subset of our observations. Let's say that our data contains observations of participants from many countries, and we want to study only participants from a given set of countries.

### >> function subset

```
subsetDataFrame <- subset(originalDataFrame, condition_to_be_included_in_subset)
```

#### Input :

- ▶ the variable `originalDataFrame` containing the original data you want to subset
- ▶ the `condition_to_be_included_in_subset` that observations need to match to be included in the new data frame

#### Output :

- a dataframe named `subsetDataFrame` including only observations from `originalDataFrame` matching the given `condition_to_create_subset`

### >>> How to define conditions for subsetting

We need to talk (*#dramatic pause*<sup>7</sup>) ... about how to write the conditions (*#substantial relief*)

Usually you will want to select observations :

1. that equals one specific value (e.g. only the French participants)
2. that are different from one specific value (e.g. all participants who are not French)
3. that have a value included in a list of possible values (e.g. French and Spanish participants)
4. that are different than a list of possible values (e.g. all but French or Spanish participants)

The R syntax for each type of condition is specified in the box below:

1. `originalDataFrame$variableA == value Y`
2. `originalDataFrame$variableA != valueY`
3. `originalDataFrame$variableA %in% c(valueB,valueC,valueD)`
4. `!(originalDataFrame$variableA %in% c(valueB,valueC,valueD))`

<sup>7</sup> There is an hash because it's a comment, R-style ;-)

Here is a more concrete example, subsetting from our dataframe **secretData** previously defined:

```
# select only French participants
frenchParticipants <- subset(secretData, secretData$nationality == 'French')

# select all but French participants
nonFrenchParticipants <- subset(secretData, secretData$nationality != 'French')

# select French and Spanish participants
frenchAndSpanishParticipants <- subset(secretData, secretData$nationality %in%
c('French', 'Spanish'))

# select all but French and Spanish participants
nonFrenchAndSpanishParticipants <- subset(secretData, !(secretData$nationality
%in% c('French', 'Spanish')))
```

## >>> More than one condition

Sometimes, you might have more than one condition to select your observations. Let's say :

1. you want only the female participants from France,
2. you want participants that have French either as their native language or their second language

Situation	R notation
Condition A <b>AND</b> Condition B	A & B
Condition A <b>OR</b> Condition B	A   B

And here is an example of code in R:

```
# A AND B : select only female French participants
femaleFrenchPpts <- subset(secretData, secretData$nationality == 'French' &
secretData$gender == 'Female')

# A OR B: select all participants with as first or second language
frenchSpeakingPpts <- subset(secretData, secretData$nativeLanguage == 'French' |
secretData$secondLanguage == 'French')
```

## Export data into a file

```
>> function write.csv
```

```
write.csv(nameOfYourDataframeInR, nameYouWantForYourFile)
```

### Mandatory Input :

- ▶ the variable `nameOfYourDataframeInR` containing the data you want to export
- ▶ the variable `nameYouWantForYourFile` containing the name of the CSV file to write. This name will be indicated between quotes and include the extension `.csv`, such as in `'myNewFancyData.csv'`

### Output :

- there is no output here in R. The real output is the CSV file in your working directory

The following command will export the data included in the dataframe `frenchParticipants` into the file `participants_fr.csv` in my working directory

```
write.csv(frenchParticipants, 'participants_fr.csv')
```

## ANNND ACTION !



- Create a subset of the dataframe `airbnb_mtl` named `affordableListings` that include only listings with a price below \$2000
- Create a subset of the dataframe `affordableListings` named `touristyListings` that include only listings in the neighbourhood 'Ville-Marie' or in 'Le Plateau-Mont-Royal'
- Save the dataframe `touristyListings` in the file `'touristyListings.csv'`

## Importing data with the Graphic User Interface (GUI)

If you really love to click buttons, it is possible to import data by clicking on menus (yay menus!). Follow the menu 'Import dataset' at the top of the **Environment** window. All subsequent details will be for a future version of this tutorial.



## Episode 5 - Basic operations

### *Compute basic values of a variable in a dataframe*

Let's say we are interested in the variable **age** in the dataframe **secretData**.

>>> **Minimum value: function** `min`

```
min(secretData$age)
```

>>> **Maximum value: function** `max`

```
max(secretData$age)
```

>>> **Mean value: function** `mean`

```
mean(secretData$age)
```

>>> **Median value: function** `median`

```
median(secretData$age)
```

>>> **Standard deviation: function** `sd`

```
sd(secretData$age)
```

>>> **Function** `summary`

```
summary(secretData$age)
```

► The output of the function **summary** would look like that:

```
   Min.  1st Qu.  Median    Mean  3rd Qu.   Max.
 70.00  76.00   79.00   79.14  82.75   88.00
```