A Brief Introduction to R

Professor Carlos Carvalho, PhD
Teaching Assistant: Pedro Santos

Introduction

This file is a brief introduction on how to start using R/RStudio. In this tutorial no previous programming knowledge is assumed, which means that if reader already has some familiarity with R, then this tutorial might not be so useful. All of the examples should run easily in any machine that had R/RStudio properly installed (which is also covered at the beginning of this file).

This tutorial is a short version of a tutorial created by Pedro Santos, Marina Muradian, Professor Maria Venezuela, and Professor Tatiana Melhado in 2019 as support material for the students of the Economics Program of the Institute of Education and Research (INSPER).

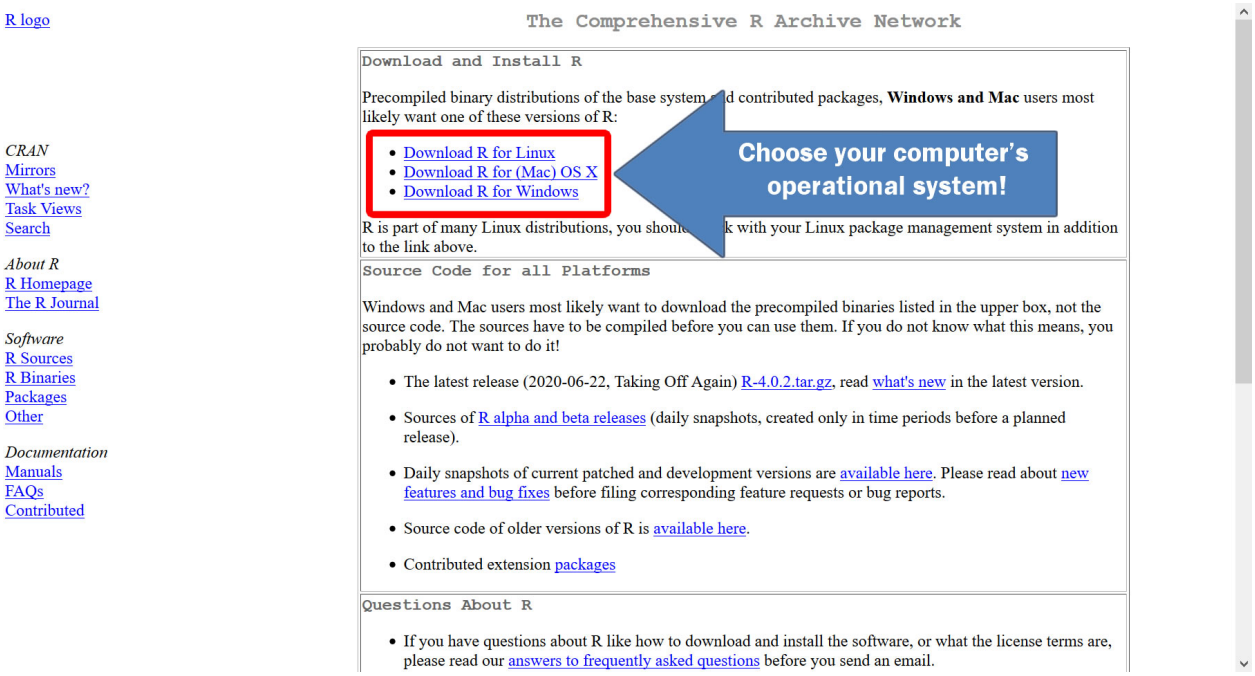
Software

For this class the following softwares are recommended:

- **R** (development envinronment);
- **RStudio** (the "friendly" IDE for R).

Installing R

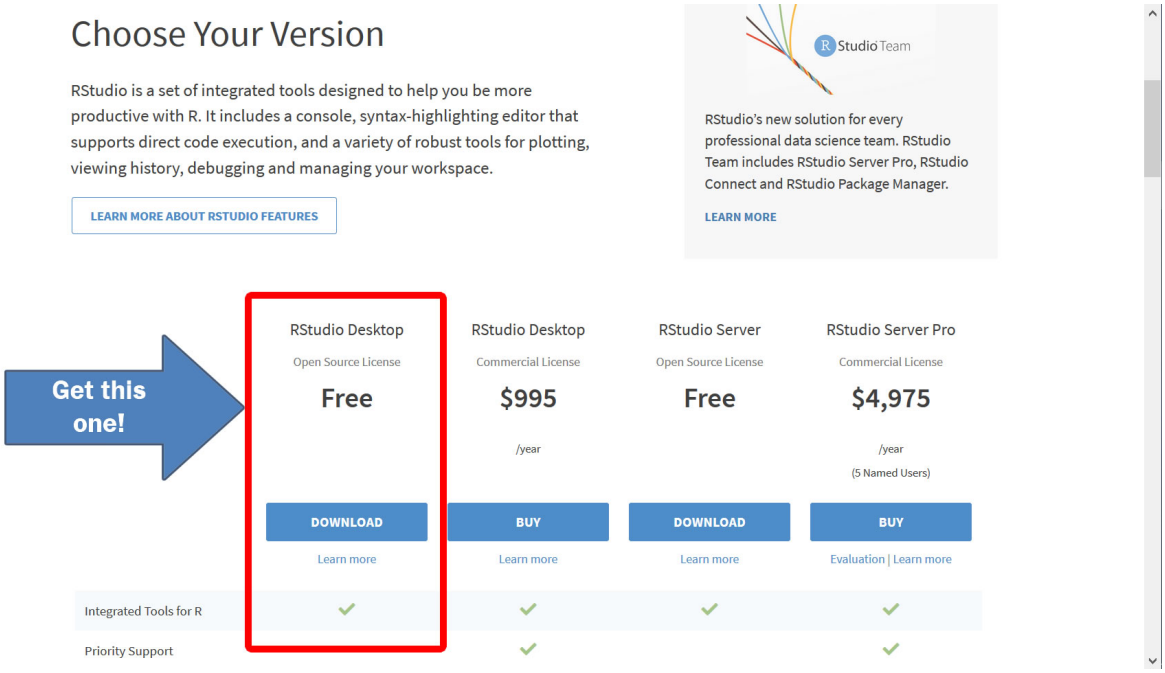
You can acquire the software for Windows, Linux and Mac at http://archive.linux.duke.edu/cran/.



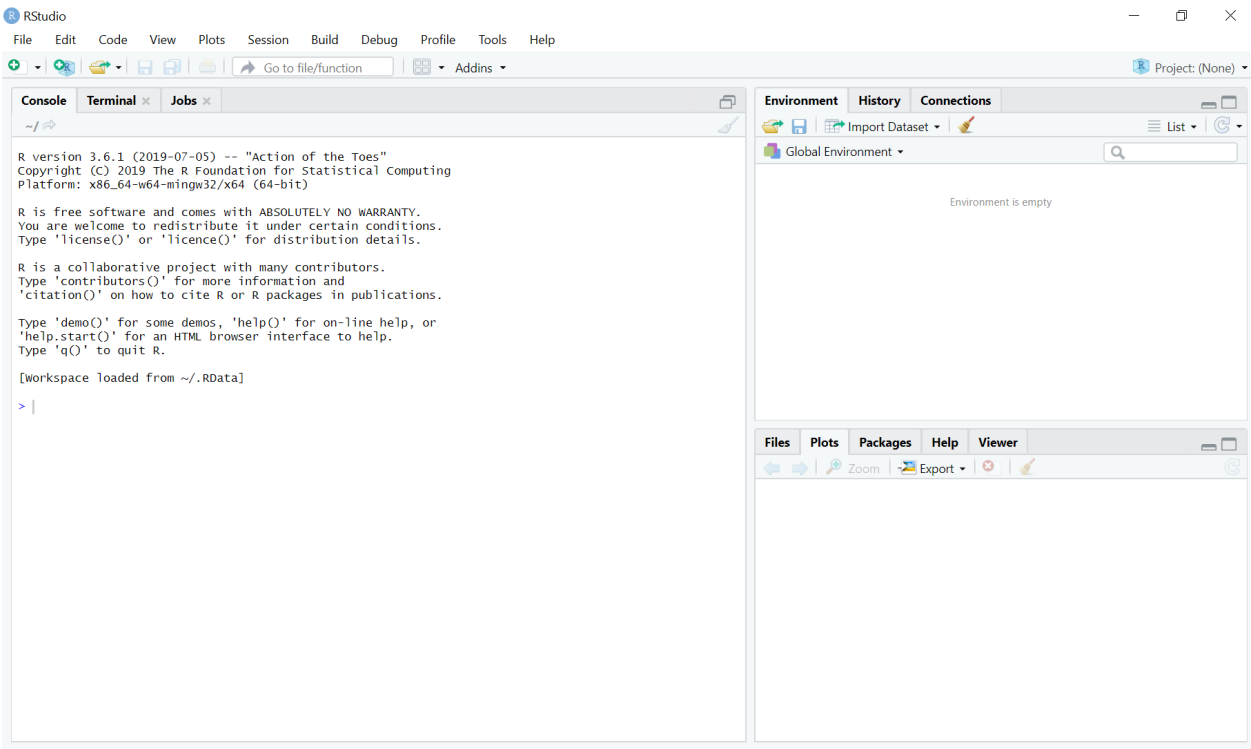
Installing RStudio

After installing R, you can download RStudio using this link: https://www.rstudio.com/products/RStudio/download/.

> **REMINDER**: RStudio is only the graphical IDE, you still **NEED** to have previously installed R for it to do anything.



After concluding the installation, open RStudio and a screen like this will pop up:

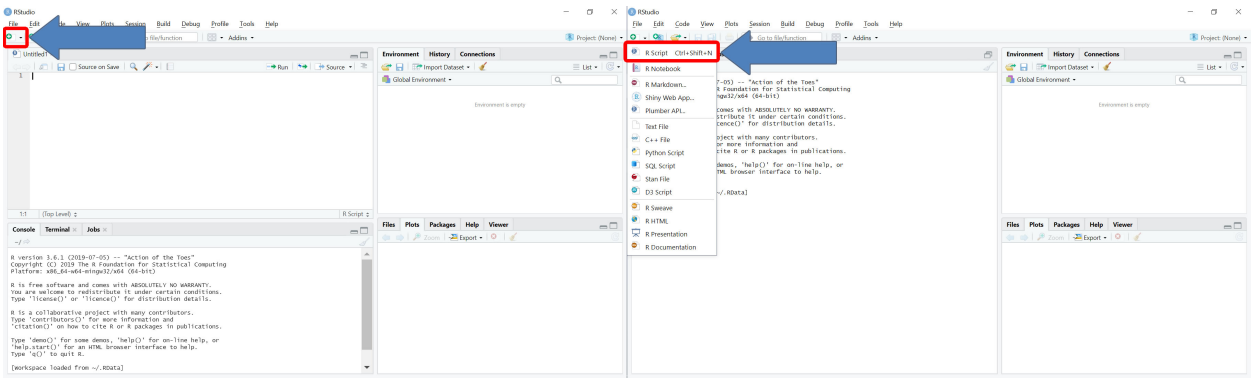


Introduction to RStudio

After opening RStudio, we will create a new script – the place where we input our code:

Knowing the Environment

All you need to do to create a new script is click on the icon for "new" in the upper left corner of RStudio and then select the item "R Script" (or use the shortcut **Ctrl+Shift+N** on Windows). The following image shows how it is done:



RStudio has four main environments:

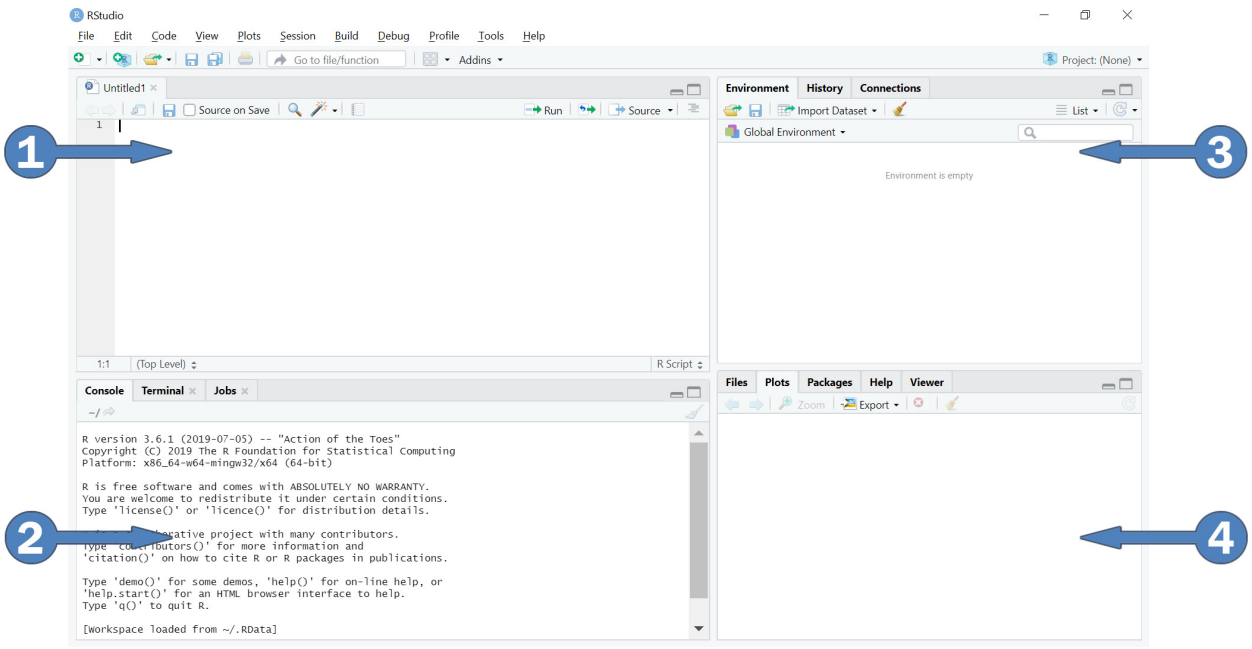
- **Script** (1st field):

  - You can save the code you wrote into a script. This makes it easier to spot any programming mistakes and also allows you to easily run it again as needed.

- **Console** (2nd field):
  - The interface where your program "runs". It is also the quickest way to insert code. Every executed code will display in the Console.

- **Environment/History** (3rd field):
  - The environment window displays the objects created by your code, along with a brief description of their characteristics. History shows previously executed lines. It is also possible to save those objects or *upload* objects and scripts from online repositories – and many other things – but for now we will only use this tab as a way to keep watch on created objects.

- **Files/Plots/Packages/Help/Viewer** (4th field):
  - And lastly, the multiuse window:
    * Shows the current work directory and allows navigation through your computer's files.
    * Displays plots and images;
    * Shows your installed packages;
    * Allows you to consult detailed documentation with examples of every function in R;
    * Shows a preview of presentations, apps, text files, pdf, html and other filetypes compatible with R.



DISCLAIMER: Unlike in a script, you **CANNOT** save code typed on R console. That is why you should use a script instead! It is the most important element when coding.

On Best Coding Practices

Now we will start using the more complex features and create objects in R. As such, it is important to learn some good practices when developing a script:

- **Commentaries**:

  - Commentaries are made using the "#" symbol (without the quotes). The system will ignore everything that comes after a "#" character.
  - Commentary is made in the script.
  - Comment your script as much as possible. It might seem unnecessary at first, but it is really important to give others – or even yourself in the future – a chance to understand what you were thinking at the time.

- **Object Names (Variables)**:
  - Prefer short and simple, but meaningful names that remind you of that object's content.
  - Avoid using single letter names, they can be confusing.
  - Special characters are not allowed in object names.

My First Line of Code

In the script, after typing a command and clicking on the "Run" button (or the shortcut "Ctrl/Command + Enter") the console will display the result. Type and run the following code, which uses the *print* function:

```
print("Hello World!")
```

```
## [1] "Hello World!"
```

R is also a calculator. We will start with something simple for our first examples, using basic operations.

```
1 + 3
```

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

> **REMINDER**: The command "Ctrl/Command + Enter" or "Run" only executes what is in the selected lines.

If you want to execute more than one line of code, use the command "Ctrl/Command + Shift + Enter" to execute the whole code. Alternatively, you may select the desired lines and press "Run".

Functions

We may also use *functions* (as the *print* function in the previous section).

For example, if we want to realize the arithmetical operation 1+2+3+4, we can instead use the *sum* function:

```
sum(1, 2, 3, 4)
```

```
## [1] 10
```

To get the square root of a number, you can use the function *sqrt*:

```
sqrt(9)
```

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

We can also use trigonometric functions like cosine:

```r
cos(3.1415)
```

```
## [1] -1
```

Variables

Variables are used to "store" information or operations:

The operators "<-" or "=" (without quotes) are used to assign a value to a variable. You may also use the shortcut **Alt + -**.

For example, if we want to save the result of 1+2+3+4, we can create the variable "sumRes":

```r
sumRes <- sum(1, 2, 3, 4)
```

Similarly, we can save the value of $\sqrt{9}$ in the variable "root":

```r
root <- sqrt(9)
```

You can freely use that variable in future operations after its creation:

For example, if we want to know the result of $2 + \sqrt{(9)}$, we can just reuse the same "root" variable we created previously in our code:

```r
2+root
```

```
## [1] 5
```

To see the current value of a variable, just call it:

```r
root
```

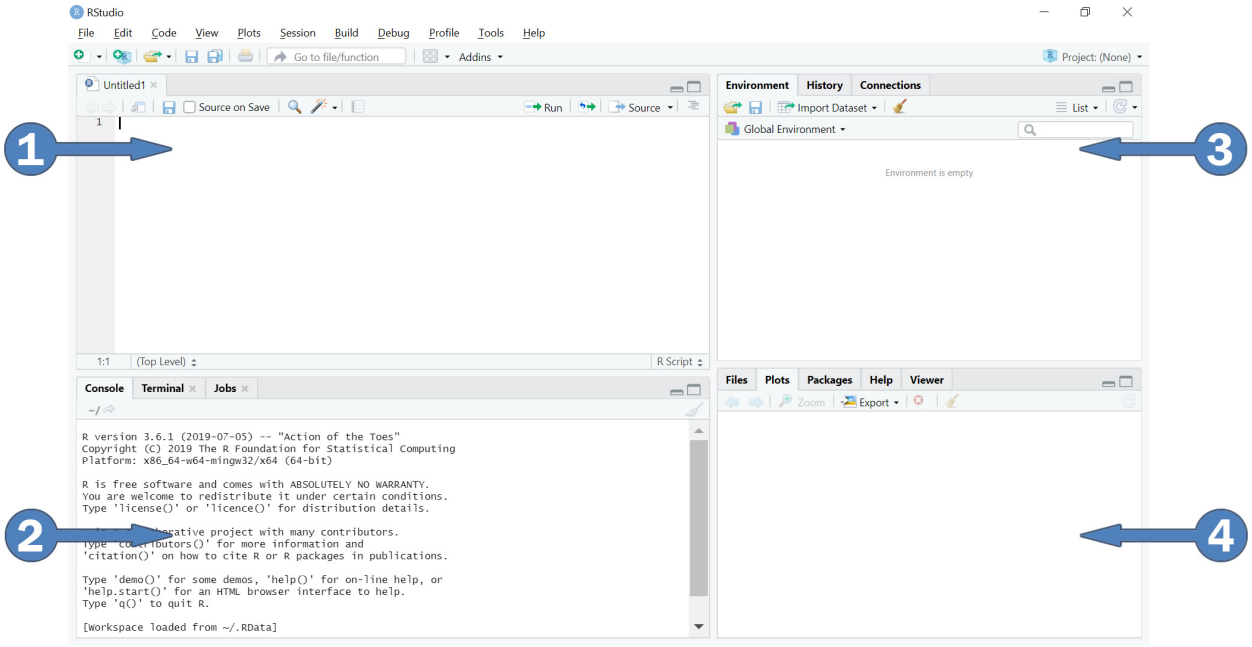
```
## [1] 3
```

This is equivalent to using the *print* function:

```r
print(root)
```

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

After its creation, you can confirm that a variable exists in the indicated area of screen by the third field (Environment), which shows all currently loaded objects:



To remove a variable from the Environment, just use the function **rm()**:

```
rm(sumRes) #Removes only the variable "sumRes"
rm(sumRes, root) #Removes the variables "sumRes" and "root" both
rm(list = ls()) # Removes ALL variables!
```

You may also completely wipe all elements off the environment clicking in the little broom icon on the top of the third field.

> **REMINDER**: When you want to write a commentary, do it after the symbol "#" and R will ignore everything that comes after it. This works even for code, so you can "comment out" lines you do not want to execute instead of deleting them, which is very useful when testing.

Arrays

Using arrays is very important in R. The most basic function to create an array is **c()** (the c stands for "concatenate"/"combine"):

For instance, if we want to create an array with the numbers 1, 2, 3 we do:

```
c(1, 2, 3)
```

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

You may also create an array with consecutive values by using the colon (":") symbol:

```
1:3
```

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

If we want to generate an array that contains all consecutive values between 100 and 200 we also use ":":

```
100:200
```

```
##   [1] 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116
##  [18] 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131 132 133
##  [35] 134 135 136 137 138 139 140 141 142 143 144 145 146 147 148 149 150
##  [52] 151 152 153 154 155 156 157 158 159 160 161 162 163 164 165 166 167
##  [69] 168 169 170 171 172 173 174 175 176 177 178 179 180 181 182 183 184
##  [86] 185 186 187 188 189 190 191 192 193 194 195 196 197 198 199 200
```

> **DISCLAIMER**: Notice that the first element in every line of the console is always a number in square brackets "[ ]". These numbers represent the position of the first element on each line in the current object displayed in the console. Even when only one number is displayed in the console, the square brackets "[1]" will be displayed.

Many functions take an array as their argument. For instance, the function **mean()**: to calculate the mean value of some numbers, you need to previously store them in an array.

For example, say we want to calculate the mean of 1, 3, 4, 7 and 10.

First, we create a variable "numbers", which will be assigned with an array of the desired numbers:

```
numbers <- c(1, 3, 4, 7, 10)
```

Now we will store in the variable "meanRes" the result of the mean of the "numbers" array using the function "mean":

```
meanRes <- mean(numbers)
```

And we will call the variable "meanRes" to display the result:

```
meanRes
```

```
## [1] 5
```

> **REMINDER**: Do note that the result of the mean was not shown on console when the object "meanRes" was created. This happens because the information returned by the function was "used up" to directly store it to the "meanRes" object. As such, there was nothing to be printed to console.

We may also create a variable "sumRes" to receive the value of the sum 1+2+3+4+7+10:

```
sumRes <- 1 + 3 + 4 + 7 + 10
```

And create a variable "mean2" to receive the result of the variable "sumRes" divided by 5 (the number of the added up numbers):

```
mean2 <- sumRes / 5
```

Similarly, we call the variable "mean2" to display its result:

```
mean2
```

```
## [1] 5
```

Matrices

On top of one dimensional arrays, you might need matrices. Something very important you should remember about matrixes is that **all** of its elements HAVE to be of the same type (be it numeric, character, etc).

1. To create a matrix, the numbers that make it up are lined up as an array (that is why you previously make the array "numbers");

```
numbers <- c(1, 3, 4, 7, 10, 14)
```

2. The function **matrix()** is used to create a matrix. The function needs a couple additional arguments:
    - nrow (the number of rows) and
    - ncol (the number of columns).

An example: we will distribute the numbers that are in "numbers" into a matrix of 3 rows and 2 columns and save it to the variable "matA":

```
matA <- matrix(numbers, nrow = 3, ncol = 2)
```

We then call the variable "matA" to display the created matrix:

```
matA
```

```
##      [,1] [,2]
## [1,]    1    7
## [2,]    3   10
## [3,]    4   14
```

Data Frames

The Data Frame is a fundamental object for data analysis in R. It is similar to a matrix. Usually, each row represents a variable and each column represents an observation.

What makes it differ from a matrix is that it can hold more than one type of variable at once (for instance, column X can hold numeric data, while colune Y holds text data).

R has some preset databases already in the data frame format. You may call those with the function **data()**:

```
data(mtcars)
```

Note that after executing this command, the object "mtcars" appears in the "Environment", on the upper right corner of RStudio.

If you directly call the data frame, all of its observations will be printed to Console, which may be troublesome if the database is too big.

To view the database on a separated screen, click on its name (mtcars, in this case) in the Envronment screen. This is the same as running the command **View(mtcars)**:

```
View(mtcars)
```

The functions **head()** and **tail()** are also useful when dealing with databases. They display the six first observations – in the case of the former – or six last, in the case of the latter, for a given database.

```
head(mtcars)
```

```
##                   mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
## Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
```

```
tail(mtcars)
```

```
##                 mpg cyl  disp  hp drat    wt qsec vs am gear carb
## Porsche 914-2  26.0   4 120.3  91 4.43 2.140 16.7  0  1    5    2
## Lotus Europa   30.4   4  95.1 113 3.77 1.513 16.9  1  1    5    2
## Ford Pantera L 15.8   8 351.0 264 4.22 3.170 14.5  0  1    5    4
## Ferrari Dino   19.7   6 145.0 175 3.62 2.770 15.5  0  1    5    6
## Maserati Bora  15.0   8 301.0 335 3.54 3.570 14.6  0  1    5    8
## Volvo 142E     21.4   4 121.0 109 4.11 2.780 18.6  1  1    4    2
```

Exploring the Data Frame

The function **dim** returns a vector of two values, where the first is the number of rows of a data frame, and the second is the number of columns of a data frame.

In general: **dim(name_of_data_frame)**:

```
dim(mtcars)
```

```
## [1] 32 11
```

Usually every column represent a different variable.

To get the names of the columns of a data frame, it is possible to use the function **names**.

In general: **names(name_of_data_frame)**:

```
names(mtcars)
```

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

It is also possible to use the function **str** to see the structure of a data frame, which includes the types of variables that are stored inside the current object.

In general: **str(name_of_data_frame)**

```
str(mtcars)
```

```
## 'data.frame':    32 obs. of  11 variables:
##  $ mpg : num  21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...
##  $ cyl : num  6 6 4 6 8 6 8 4 4 6 ...
##  $ disp: num  160 160 108 258 360 ...
##  $ hp  : num  110 110 93 110 175 105 245 62 95 123 ...
##  $ drat: num  3.9 3.9 3.85 3.08 3.15 2.76 3.21 3.69 3.92 3.92 ...
##  $ wt  : num  2.62 2.88 2.32 3.21 3.44 ...
##  $ qsec: num  16.5 17 18.6 19.4 17 ...
##  $ vs  : num  0 0 1 1 0 1 0 1 1 1 ...
##  $ am  : num  1 1 1 0 0 0 0 0 0 0 ...
##  $ gear: num  4 4 4 3 3 3 3 4 4 4 ...
##  $ carb: num  4 4 1 1 2 1 4 2 2 4 ...
```

Subsets

An important part of coding in R is referencing subsets. In this case, the used operators are the square brackets "[ ]".

Say we want to select the fourth element of a sequence between 1 and 10.

First, we generate the consecutive numbers from 20 to 30 using ":", like we saw previously and store it to a variable called "numbers".

```
numbers <- 20:30
```

Now we ask R to return us the fourth element of this sequence:

```
numbers[4]
```

```
## [1] 23
```

We may also ask R to return us the elements in the 1st, 3rd and 7th positions in this sequence. For this, we create an array with those positions of interest using "c()" and place them in the middle of square brackets:

```
numbers[c(1, 3, 7)]
```

```
## [1] 20 22 26
```

If we then want R to return us those elements, but we do not want the element in the second position, we use "[-2]":

```
numbers[-2]
```

```
##  [1] 20 22 23 24 25 26 27 28 29 30
```

Similarly, if we want R to return us all numbers, except those in the 2nd, 5th and 9th position, we combine the logic used in the last two examples:

```
numbers[-c(2, 5, 9)]
```

```
## [1] 20 22 23 25 26 27 29 30
```

For *matrices* and *data frames*, the procedure is somewhat similar. You just need to pay attention that we are now dealing with two dimensions, one for rows and one for columns.

Say we want R to return us an element in the second line and third column of the data frame "mtcars". We use the command:

```
mtcars[2, 3]
```

```
## [1] 160
```

Now if we want to view all elements of the first column (its lines), we use:

```
mtcars[, 1]
```

```
##  [1] 21.0 21.0 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 17.8 16.4 17.3 15.2
## [15] 10.4 10.4 14.7 32.4 30.4 33.9 21.5 15.5 15.2 13.3 19.2 27.3 26.0 30.4
## [29] 15.8 19.7 15.0 21.4
```

Similarly, if we want to view all elements of the first line (its columns):

```
mtcars[1, ]
```

```
##           mpg cyl disp  hp drat   wt  qsec vs am gear carb
## Mazda RX4  21   6  160 110  3.9 2.62 16.46  0  1    4    4
```

Columns of a Data Frame

If we want to do operations with specific columns of a given data frame we are using, we may use the symbol **$**. This way, you just need to know the name of that variable instead of knowing the exact position of that variable.

As such, if we want to select all elements of the column "cyl" of the database "mtcars", we use:

```
mtcars$cyl
```

```
##  [1] 6 6 4 6 8 6 8 4 4 6 6 8 8 8 8 8 8 4 4 4 4 8 8 8 8 4 4 4 8 6 8 4
```

Filtering specific observations in a Data Frame

Let us say that I want to find which observations of *mtcars* have the variable **cyl=4**. Again, we can use the square brackets, but this time with logical operators. Logical operators perform a test according to the rule defined in the logical operator, and if the test is positive, **TRUE** is returned, while if the test is negative, **FALSE** is returned.

Inside the square brackets we can add any desirable set of logical operations. In our example, since we want observations with **cyl=4**, it is possible to use the "==" operator.

In the case of using a logical operator in a vector, the test is performed for each individual and a logical vector filled with **TRUE** and **FALSE** elements is returned.

```
mtcars$cyl == 4
```

```
##  [1] FALSE FALSE  TRUE FALSE FALSE FALSE FALSE  TRUE  TRUE FALSE FALSE
## [12] FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE FALSE
## [23] FALSE FALSE FALSE  TRUE  TRUE  TRUE FALSE FALSE FALSE  TRUE
```

By inserting this logical vector inside the square brackets, only the elements at the positions where **TRUE** is located will be returned.

```
mtcars$mpg[mtcars$cyl == 4]
```

```
##  [1] 22.8 24.4 22.8 32.4 30.4 33.9 21.5 27.3 26.0 30.4 21.4
```

> **DISCLAIMER**: The logical operator "==" performs a logic test, while the operator "=" is used to store elements in objects.

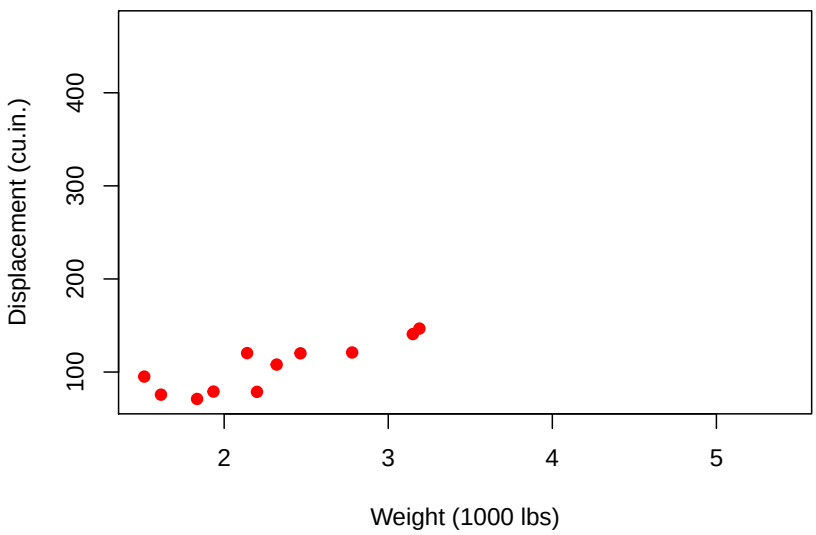
Data Analysis

Scatter Plots segmented by a Qualitative Variable

Let us analyse the scatter plot of two variables from *mtcars* (x = wt, y = disp), but I want to see the number of cylinders ("cyl") of each car.

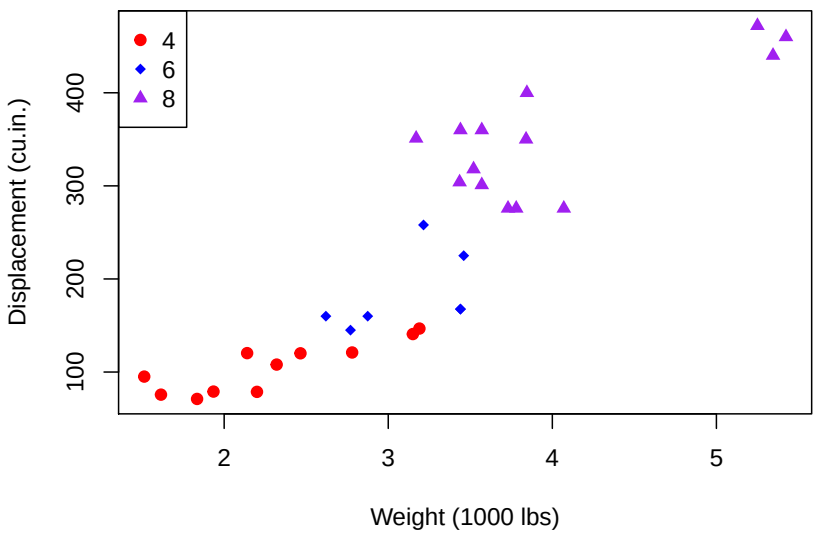
Initially, you can create a scatter plot with only one of the categories using the function **plot()**. Please note that to set the arguments **xlim** and **ylim** we use the function **range()**, which returns an array with the minimum value and the maximum value of the array given as argument.

```
plot(x = mtcars$wt[mtcars$cyl == 4],
     y = mtcars$disp[mtcars$cyl == 4],
     xlab = "Weight (1000 lbs)",
     ylab = "Displacement (cu.in.)",
     pch = 19,      # Format: circles
     col = "red",   # Color: red
     xlim = range(mtcars$wt), #limits for the x axis
     ylim = range(mtcars$disp)        #limits for the y axis
     )
```



To add points in an already existing plot we simply add the function *points* after the plot's code:

```
plot(x = mtcars$wt[mtcars$cyl == 4],
     y = mtcars$disp[mtcars$cyl == 4],
     xlab = "Weight (1000 lbs)",
     ylab = "Displacement (cu.in.)",
     pch = 19,      # Format: circles
     col = "red",   # Color: red
     xlim = range(mtcars$wt), #limits for the x axis
     ylim = range(mtcars$disp)        #limits for the y axis
     )

# Adds points for 6 cylinders:
points(x = mtcars$wt[mtcars$cyl == 6],
       y = mtcars$disp[mtcars$cyl == 6],
       pch = 18,      # Format: diamond
       col = "blue"  # Color: blue
       )

# Adds points for 8 cylinders:
points(x = mtcars$wt[mtcars$cyl == 8],
       y = mtcars$disp[mtcars$cyl == 8],
       pch = 17,      # Format: triangles
       col = "purple"  # Color: blue
       )

# Adds caption to the plot:
legend("topleft",                    # Plot's position (topleft)
       c("4", "6", "8"),   # Variable names
       pch = c(19, 18, 17),              # Point's formats
       col = c("red", "blue", "purple")        # Point's colors
       )
```



Getting the Linear Fit Line ($\hat{y} = ax + b$) segmented by a Qualitative Variable

To get the linear fit equation we use the function **lm()**.

```
# For cyl = 4:
eq_4 <- lm(mtcars$disp[mtcars$cyl == 4] ~ mtcars$wt[mtcars$cyl == 4])
eq_4
```

```
##
## Call:
## lm(formula = mtcars$disp[mtcars$cyl == 4] ~ mtcars$wt[mtcars$cyl ==
##     4])
##
## Coefficients:
##                 (Intercept)  mtcars$wt[mtcars$cyl == 4]
##                       12.77                       40.41
```

```
# For cyl = 6:
eq_6 <- lm(mtcars$disp[mtcars$cyl == 6] ~ mtcars$wt[mtcars$cyl == 6])
eq_6
```

```
##
## Call:
## lm(formula = mtcars$disp[mtcars$cyl == 6] ~ mtcars$wt[mtcars$cyl ==
##     6])
##
## Coefficients:
##                 (Intercept)  mtcars$wt[mtcars$cyl == 6]
##                       11.52                       55.11
```

```r
# For cyl = 8:
eq_8 <- lm(mtcars$disp[mtcars$cyl == 8] ~ mtcars$wt[mtcars$cyl == 8])
eq_8
```

```
##
## Call:
## lm(formula = mtcars$disp[mtcars$cyl == 8] ~ mtcars$wt[mtcars$cyl ==
##     8])
##
## Coefficients:
##                 (Intercept)  mtcars$wt[mtcars$cyl == 8]
##                       83.54                       67.40
```

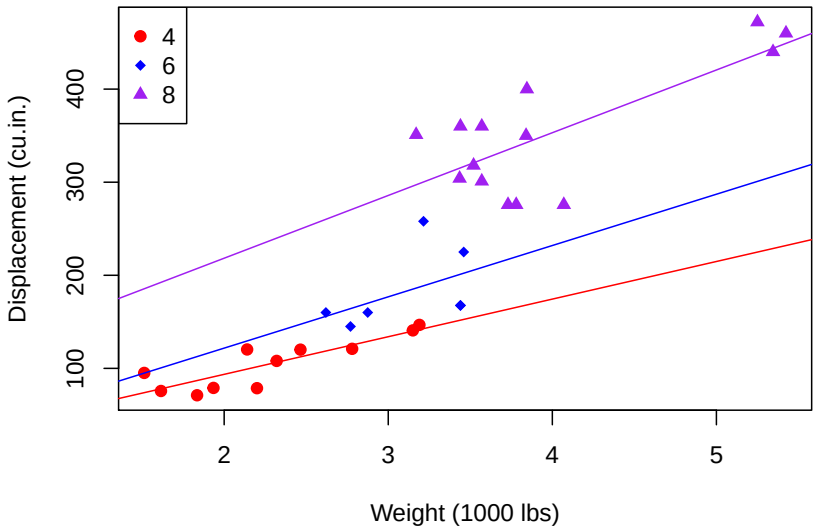
With that, we have those linear fit equations:

- **cyl = 4**: $\hat{y} = 40.41x - 12.77$
- **cyl = 6**: $\hat{y} = 55.11x - 11.52$
- **cyl = 8**: $\hat{y} = 67.40x - 83.54$

As before, we may also include those trend lines to the scatter plot from before using the function **abline()**:

```r
plot(x = mtcars$wt[mtcars$cyl == 4],
     y = mtcars$disp[mtcars$cyl == 4],
     xlab = "Weight (1000 lbs)",
     ylab = "Displacement (cu.in.)",
     pch = 19,      # Format: circles
     col = "red",   # Color: red
     xlim = range(mtcars$wt), #limits for the x axis
     ylim = range(mtcars$disp)        #limits for the y axis
     )

# Adds points for 6 cylinders:
points(x = mtcars$wt[mtcars$cyl == 6],
       y = mtcars$disp[mtcars$cyl == 6],
       pch = 18,      # Format: diamond
       col = "blue"  # Color: blue
       )

# Adds points for 8 cylinders:
points(x = mtcars$wt[mtcars$cyl == 8],
       y = mtcars$disp[mtcars$cyl == 8],
       pch = 17,      # Format: triangles
       col = "purple"  # Color: blue
       )

# Adds a OLS line for cyl = 4:
abline(lm(mtcars$disp[mtcars$cyl == 4] ~ mtcars$wt[mtcars$cyl == 4]),
       col = "red"
       )
```

```r
# Adds a OLS line for cyl = 6:
abline(lm(mtcars$disp[mtcars$cyl == 6] ~ mtcars$wt[mtcars$cyl == 6]),
       col = "blue"
       )

# Adds a OLS line for cyl = 8:
abline(lm(mtcars$disp[mtcars$cyl == 8] ~ mtcars$wt[mtcars$cyl == 8]),
       col = "purple"
       )

# Adds caption to the plot:
legend("topleft",                    # Plot's position (topleft)
       c("4", "6", "8"),   # Variable names
       pch = c(19, 18, 17),              # Point's formats
       col = c("red", "blue", "purple")        # Point's colors
       )
```



Libraries

Until now the functions used are available with the **base R** package. The **base** has a wide range of functions that are often enough for beginners.

However, when dealing with specfic problems, users can create their own functions, which may also be included in custom libraries/packages. Then the users can share their work through online repositories, such as **CRAN**.

Let us use the library **lubridate** as an example. To install the library we must run the following code:

```r
install.packages("lubridate")
```

In general: **install.packages("name_of_the_library")**.

REMINDER: The name of the library must always be between quotation marks when installing because the function install.packages needs to receive a character vector as an argument.

Once installed, the library is always available to the user, but must be turned on every session by using the function **library**:

```
library(lubridate)
```

In general: **library(name_of_the_library)**.

Now all functions of the library are available to the user.

DISCLAIMER: Only turn on the libraries that are going to be used at the moment. Turning many libraries on might cause some conflits, like multiple functions with the same name and unnecessary use of the computer memory.

Common Issue: Special Characters

A common problem with RStudio when opening a user's R script is encountering characters not present in the English alphabet.

Any unrecognized character like accents will be replaced by a question mark (?) to indicate that RStudio failed to identify that character.

With the script open, do the following to solve this issue:

1. Click on the tab "File", on the upper left corner of RStudio.
2. Select the item "Reopen with Encoding...".
3. Select the Encoding option. Usually the options "UTF-8" or "WINDOWS-1252" will solve the issue. Which one works depends on the script.
4. Click on "OK".

Accessing Function Documentation

To get help about a function and its arguments, use the command "**?**". Try it for yourself on the functions **abs()** and **sort()**.

```
?abs
?sort
```

Changing the Working Directory

When you are using R, the software uses a working directory. This directory is where saved files are be stored, and usually where you want any external datasets to be located for easy access. You can check your current working directory by using the function **getwd()**:

```
getwd()
```

If you want to change your current directory by any reason, you can use the function **setwd()**:

```r
setwd("~/Documents/files/my_directory")
```

You can also set your working directory manually by:

- Clicking on: Session > Set Working Directory > Choose Directory...; or
- Pressing: Ctrl + Shift + H (on Windows).

Keep Learning!

Since R is an open software, a community collaboratively works on it. Thanks to that, there are many books, websites, blogs and discussion groups made by users that wish to solve some of the problems encountered by users accross the globe. Here are some examples available on the Internet:

- For a better understanding of R and its many tools, the book **R for Data Science** by the authors Hadley Wickham and Garrett Grolemund is highly recommended. The book is available, free of charge, here: http://r4ds.had.co.nz/
- Found some odd issue? Chances are someone ran on that before you and asked about it on **Stack Overflow**. This site is a forum where users bring their problems and other users solve them. This site is for many languages other than R, too. You may find it at: https://stackoverflow.com/
- **R-bloggers** is a blog full of tutorials that is updated frequently. There, experienced users explain useful resources of R, from very basic concepts to advanced coding techniques, often with examples. You may find it at: https://www.r-bloggers.com/
- On top of the references found here, other tutorials about the main topics of how R is structured may prove useful. A good example may be found at: https://www.rstudio.com/wp-content/uploads/2016/05/base-r.pdf