Workshop: Data science with R

ZEW - Session #1

Obryan Poyser
2019/02/19

What's R?

- R is a programming language and free software environment for statistical computing and graphics supported by the R Foundation for Statistical Computing.
- Is an implementation of the S programming language combined with lexical scoping semantics, inspired by Scheme.
- Created by Ross Ihaka and Robert Gentleman at the University of Auckland, New Zealand, and currently developed by the R Development Core Team. [1]
- The most common integrated development environment (IDE) is RStudio. The one we are going to use

[1] Wikipedia

Installing R and RStudio IDE

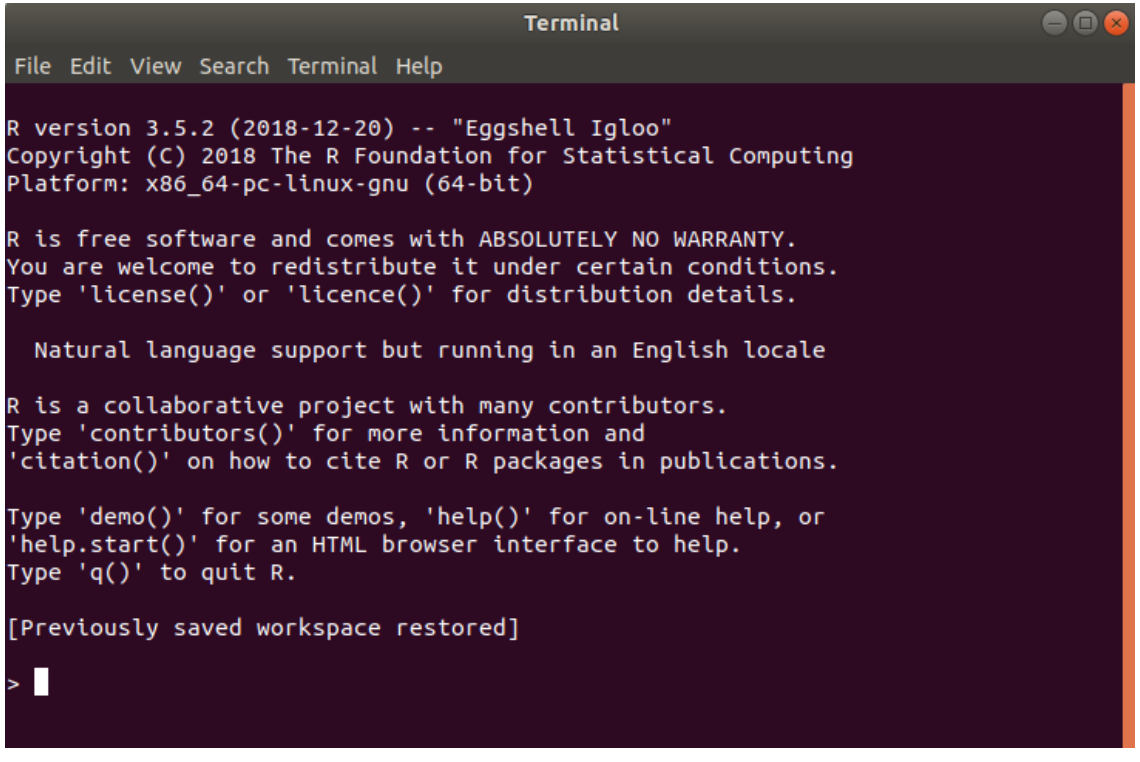
Install R base

1. Go to The R Project for Statistical Computing official website
2. Search for the **download R** link
3. Select the closest Comprehensive R Archive Network (CRAN) mirror (it won't make much difference if you choose another one)
4. Select the file according to your OS

Install IDE

1. Go to RStudio
2. Choose the RStudio Desktop Open Source License
3. Then again, the option for your specific OS
4. Execute both

R Console



Here it is possible to execute all kind of build-in functions. Give it a try

```
1+1
```

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

```
log(0)
```

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

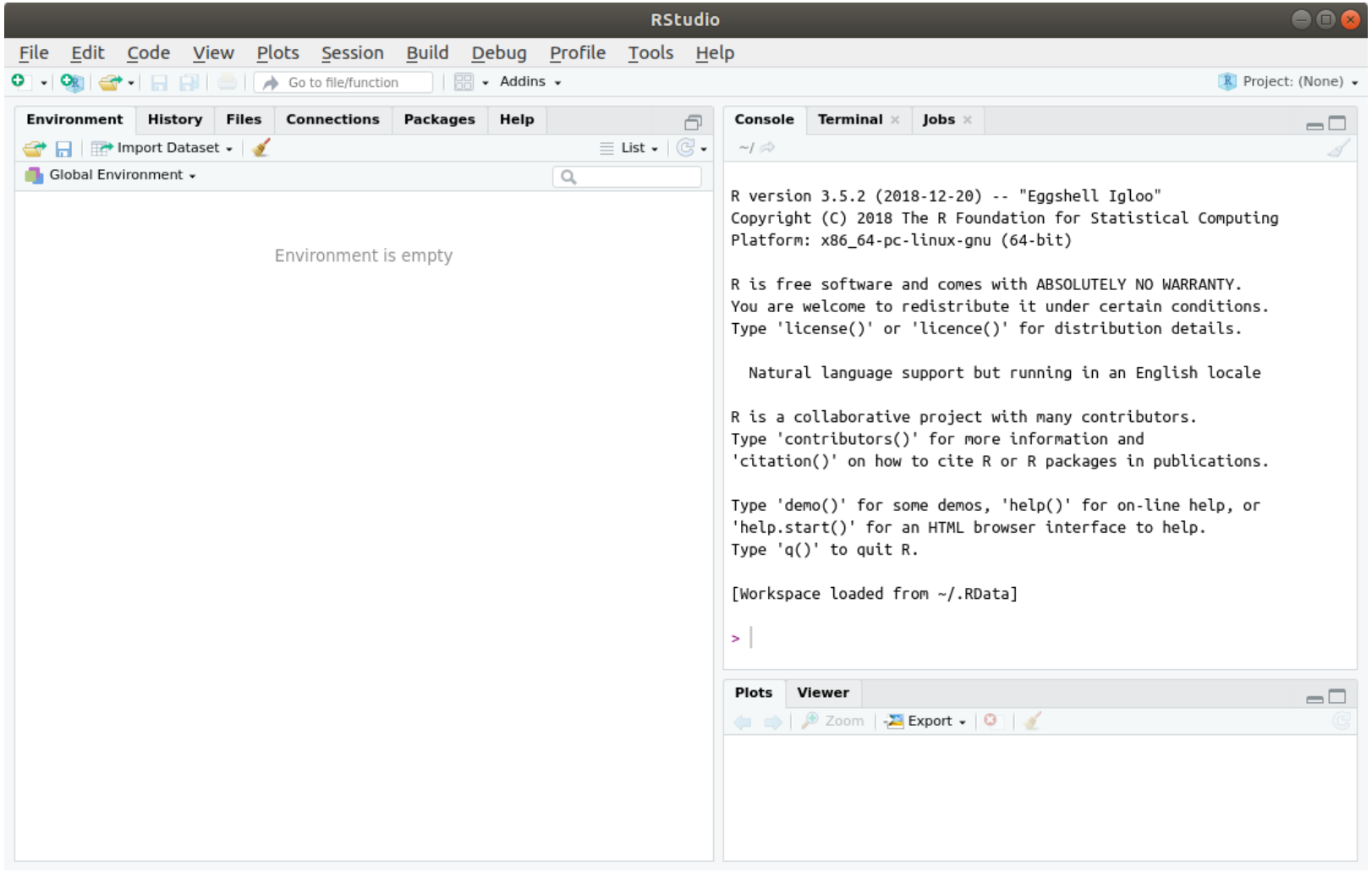
```
"Hello World"
```

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

```
TRUE*TRUE
```

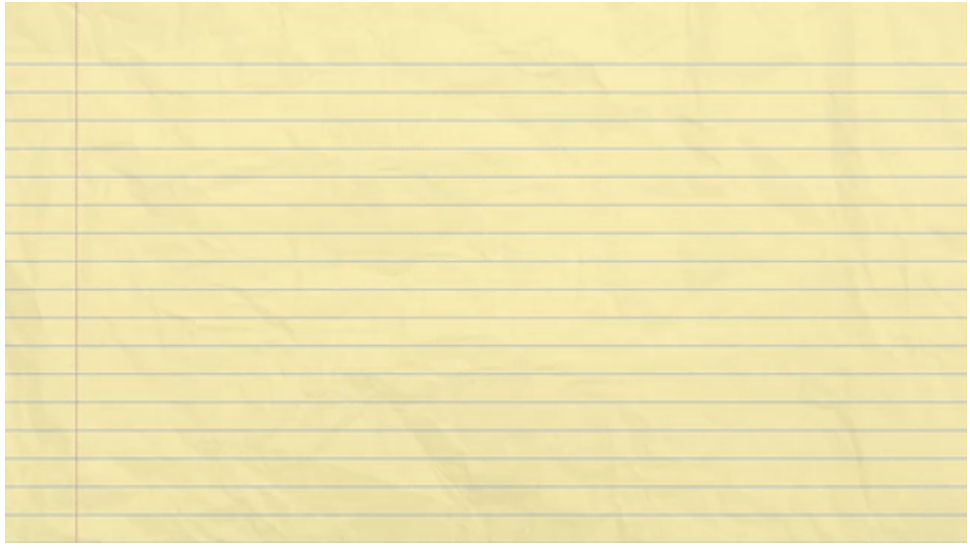
```
## [1] 1
```

RStudio interface



One can change the layout by following (among other things): `Tools-> Global options -> Panel layout`

Creating a project



RStudio projects make it straightforward to divide your work into multiple contexts, each with their own working directory, workspace, history, and source documents.[1]

[1] Rstudio

Basic commands

Getting help

```
?glm
```

Install a package

```
install.packages("plm")
```

Load a package

```
# Into the session
require("plm")
library("plm")
# An specific function
plm::vcovHC()
```

Load a build-in dataset

```
data(mtcars)
```

Current working directory

```
getwd()
```

Change current WD

```
setwd()
```

Assigning values

- R is an object oriented programming language. That is, simple things such a number `5` can be a object when is assigned to `x`. `x` can also be an image, model, dataset, function, etc.
- There are three types of assiggment operadors
  - `<-` which is the same as `=`
  - `<<-` and `->>` mostly used inside functions to assign values to objects in the environment.

```
x_0 <- 5
x_1 = 5
```

It is possible to visualize the object either with the function `print()` or only naming it.

```
print(x_0)
```

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

```
x_1
```

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

Classes

- Each object is identified with a class. The basic classes are:
  - Logical (True or False)
  - Numeric
  - Character or string
  - Factor
  - Complex

```
obj_1 <- 5 # Double precision number
obj_1a <- 5L # Integer
obj_2 <- "5" # Note that character elements are in between double quotes
obj_3 <- TRUE # Also T in upper case is equal to TRUE, conversely F for FALSE
obj_4 <- gl(1, k = 1) # Factors are special types of characters
obj_5 <- 1+1i # Complex numbers are followed by a i (I have never used :/)
```

Then using `class()` on every object above:

```
## [1] "numeric"   "integer"   "character" "logical"   "factor"    "complex"
```

Note: There are rules for naming, for example, one can not create an object with the name `8`. Try it!

```
8 <- 9
```

Classes: Missing values

Missing values are denote by `NA` for general use, or `NaN` for undefined numerical operations.

```
na <- NA
nan <- NaN
```

Testing could be performed with the following functions

```
is.na(na)
```

```
## [1] TRUE
```

```
is.na(nan)
```

```
## [1] TRUE
```

```
is.nan(na)
```

```
## [1] FALSE
```

```
is.nan(nan)
```

```
## [1] TRUE
```

Classes: Coercion

Objects such as `obj_1` are vectors of length 1. Vectors, as we will see further can only be composed by one type of class. Explicit coercion is achieved as:

```
as.integer(234.4); as.logical(1); as.logical(0); as.character(5); as.factor(5); as.numeric("string"); as.complex(9)
```

```
## [1] 234

## [1] TRUE

## [1] FALSE

## [1] "5"

## [1] 5
## Levels: 5

## Warning: NAs introduced by coercion

## [1] NA

## [1] 9+0i
```

Functions

Functions have named arguments, some can have default values. Other unnamed arguments such as `...` can be passed to the function too. Arguments can be obtained with `args()`

```
args(glm)
```

```
## function (formula, family = gaussian, data, weights, subset,
##     na.action, start = NULL, etastart, mustart, offset, control = list(...),
##     model = TRUE, method = "glm.fit", x = FALSE, y = TRUE, singular.ok = TRUE,
##     contrasts = NULL, ...)
## NULL
```

Creating a function is easy, one has to create arguments then pass them to the statement.

```
divide_function <- function(x, y=10){
  z <- x/y # See how a variable inside a function is created. Calling z outside won't work
  return(z)
}
```

If `y` is not specified explicitly, then the default value is passed to the function

```
divide_function(x = 100)
```

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

Functions

Note we can change the position of the arguments as long as they are specified.

```
divide_function(y=50, x=100)
```

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

However, if they are not denoted explicitly, R uses the value according to the order of the arguments

```
divide_function(50, 100)
```

```
## [1] 0.5
```

Scoping rules

In R functions is possible to define local variables. For instance, `z` does not exist outside the function `divide_function`.

```
print(z)
```

```
## Error in print(z): object 'z' not found
```

- If a variable at a given run-time:
  - Avoid renaming of the variables in the environment
  - All potentical confussions
- Scoping rules treats about how R manage *unknown* variables, that is, variables that are not defined inside the function.

```
sum_function <- function(x){
  x+y # This is a simple result, since is a single numeric value. But be careful
}

sum_function(10)
```

```
## Error in sum_function(10): object 'y' not found
```

The `sum_function` does not find `y` inside, looks outside, and neither is there. Hence, if we intentionally create a variable `y <- 5`, then:

```
y <- 5
sum_function(10)
```

```
## [1] 15
```

Scoping rules

Pro-tip 😎

Using the special assigning symbol `<<-` inside a function makes any variable scale to the environment. For example, `log_x` does not exist in the current environment.

But if we purposely try a second `<` into the following function:

```
some_function <- function(x){
  log_x <<- log(x)}
```

```
some_function(10)
```

```
## [1] 2.302585
```

```
log_x
```

```
## [1] 2.302585
```

Vectors

Creating vectors is easy with `vector()`. Also, the function `c()` stands for concatenate, and is used for aggregating elements into a single object.

```
vec_1 <- c(1, 1, 2, 3, 5)
vec_2 <- 1:5
class(vec_1)
```

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

Vectors **can only have one type of class**. If a given vector is initialy composed by a mixture of elements, R automatically apply coercion rules to homogenize the vector.

```
vec_3 <- c(TRUE, F, 1)
vec_4 <- c("A", T, 2)
class(vec_3)
```

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

```
class(vec_4)
```

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

Mathematical operations

```
vec_1/vec_2
```

```
## [1] 1.0000000 0.5000000 0.6666667 0.7500000 1.0000000
```

```
vec_1*vec_2
```

```
## [1]  1  2  6 12 25
```

```
vec_1+vec_2
```

```
## [1]  2  3  5  7 10
```

```
vec_1-1
```

```
## [1] 0 0 1 2 4
```

```
vec_1^2
```

```
## [1]  1  1  4  9 25
```

Selecting elements (subsetting) vectors

By position

```
vec_1
```

```
## [1] 1 1 2 3 5
```

```
vec_2[2] # 2nd element of vector vec_2
```

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

```
vec_2[-1] # Every element of vector vec_2 except the 1st
```

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

```
vec_2[c(2:4)] # Elements between 2nd and 4th (inclusive)
```

```
## [1] 2 3 4
```

By coincidence

For this, we have to take into account logical conditions

- `a == b`
- `a != b`
- `a > b`
- `a < b`
- `a >= b`
- `a <= b`

```
vec_1==1
```

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

```
vec_1[vec_1==1]
```

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

```
vec_1[vec_1 %in% c(2, 3)]
```

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

Matrices

Vectors are one dimensional array, while matrices are two-dimensional. A matrix can be created as:

```
mat_1 <- matrix(c(1:7, "A", NA), nrow = 3, ncol = 3
                , dimnames = list(c("row_1", "row_2", "row_3")
                                  , c("col_1", "col_2", "col_3
mat_1
```

```
##       col_1 col_2 col_3
## row_1 "1"   "4"   "7"
## row_2 "2"   "5"   "A"
## row_3 "3"   "6"   NA
```

As any matrix, in R a matrix has a size $m \times n$, where m is the number of rows and n the number of columns. Subsetting a matrix is fairly simple, one can only has to index a second element of interest divided by a comma inside the squared bracket.

```
mat_1[1,3] # 1st row, 3rd column
```

```
## [1] "7"
```

```
mat_1[1:2, 3] # Rows 1:2, 3rd column
```

```
## row_1 row_2
##   "7"   "A"
```

```
mat_1[1:3, 2:3] # Rows 1:3, columns 2:3
```

```
##       col_2 col_3
## row_1 "4"   "7"
## row_2 "5"   "A"
## row_3 "6"   NA
```

Lists

A list is a set of elements which can be of different types.

Empty list

```
(list_1 <- list())
```

```
## list()
```

List of one element

```
(list_2 <- list(5))
```

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

List of several elements (unnamed)

```
(list_3 <- list(F, "string", factor("factor"), 8))
```

```
## [[1]]
## [1] FALSE
##
## [[2]]
## [1] "string"
##
## [[3]]
## [1] factor
## Levels: factor
##
## [[4]]
## [1] 8
```

Lists

Since lists admit various types of types or even objects, it is possible to name them.

```
(list_4 <- list(logical=F
                , string="string"
                , factor=factor("factor"), numeric=8))
```

```
## $logical
## [1] FALSE
##
## $string
## [1] "string"
##
## $factor
## [1] factor
## Levels: factor
##
## $numeric
## [1] 8
```

```
names(list_4)
```

```
## [1] "logical" "string"  "factor"  "numeric"
```

Subsetting of list can be performed by several ways. It will depend on the structure, for instance, list can be nested into other lists. The function `str()` shows the structure of the objects.

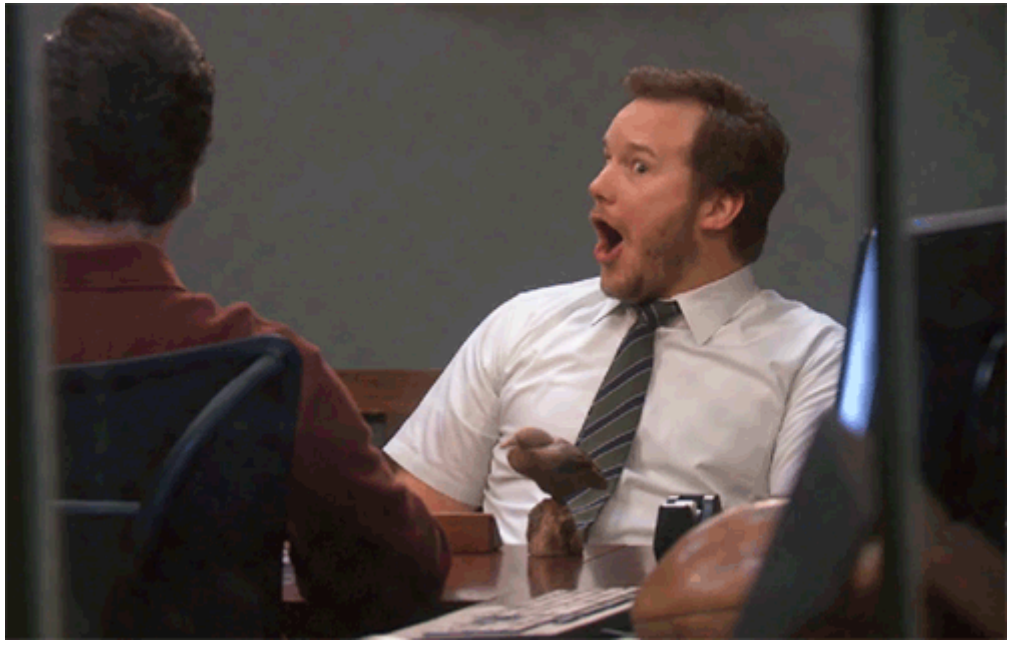
```
str(list_4)
```

```
## List of 4
##  $ logical: logi FALSE
##  $ string : chr "string"
##  $ factor : Factor w/ 1 level "factor": 1
##  $ numeric: num 8
```

Lists

Understanding lists can be a pain in the arse. Mainly because complex nested configurations and their first order internal structure.

For instance, a list can allocate a `list_4` from the past slide and a dataset



```
nested_list <- list(list=list_4, dataframe=mtcars[1:10, 1:5])
str(nested_list)
```

```
## List of 2
##  $ list     :List of 4
##   ..$ logical: logi FALSE
##   ..$ string : chr "string"
##   ..$ factor : Factor w/ 1 level "factor": 1
##   ..$ numeric: num 8
##  $ dataframe:'data.frame':    10 obs. of  5 variables:
##   ..$ mpg : num [1:10] 21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2
##   ..$ cyl : num [1:10] 6 6 4 6 8 6 8 4 4 6
##   ..$ disp: num [1:10] 160 160 108 258 360 ...
##   ..$ hp  : num [1:10] 110 110 93 110 175 105 245 62 95 123
##   ..$ drat: num [1:10] 3.9 3.9 3.85 3.08 3.15 2.76 3.21 3.69 3.92 3.92
```

Subsetting lists

Option #1 with `[[]]`

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

```
## $logical
## [1] FALSE
##
## $string
## [1] "string"
##
## $factor
## [1] factor
## Levels: factor
##
## $numeric
## [1] 8
```

Option #2 with a `$` (💡 lists need to be named in order to use this alternative)

```
nested_list$list
```

```
## $logical
## [1] FALSE
##
## $string
## [1] "string"
##
## $factor
## [1] factor
## Levels: factor
##
## $numeric
## [1] 8
```

Subsetting lists

Option #3

```
nested_list["dataframe"]
```

```
## $dataframe
##                   mpg cyl  disp  hp drat
## Mazda RX4         21.0   6 160.0 110 3.90
## Mazda RX4 Wag     21.0   6 160.0 110 3.90
## Datsun 710        22.8   4 108.0  93 3.85
## Hornet 4 Drive    21.4   6 258.0 110 3.08
## Hornet Sportabout 18.7   8 360.0 175 3.15
## Valiant           18.1   6 225.0 105 2.76
## Duster 360        14.3   8 360.0 245 3.21
## Merc 240D         24.4   4 146.7  62 3.69
## Merc 230          22.8   4 140.8  95 3.92
## Merc 280          19.2   6 167.6 123 3.92
```

For example, in order to access to the first column of the dataset `mtcars` one could write

```
nested_list["dataframe"]$dataframe$mpg
```

```
##  [1] 21.0 21.0 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2
```

It is possible to subset a list partially

```
nested_list[["d"]]
```

```
## NULL
```

```
nested_list[["da", exact=F]]
```

```
##                   mpg cyl  disp  hp drat
## Mazda RX4         21.0   6 160.0 110 3.90
## Mazda RX4 Wag     21.0   6 160.0 110 3.90
## Datsun 710        22.8   4 108.0  93 3.85
## Hornet 4 Drive    21.4   6 258.0 110 3.08
## Hornet Sportabout 18.7   8 360.0 175 3.15
## Valiant           18.1   6 225.0 105 2.76
## Duster 360        14.3   8 360.0 245 3.21
## Merc 240D         24.4   4 146.7  62 3.69
## Merc 230          22.8   4 140.8  95 3.92
## Merc 280          19.2   6 167.6 123 3.92
```

Dataframes

Dataframes are a special type of list in which each column represents a vector of elements associated with a row, they resemble the typical dataset that we use in Stata.

```
data("iris") # build-in dataframe
DT::datatable(iris, fillContainer = FALSE, options = list(page
```

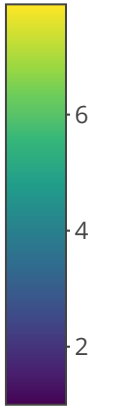
Show 6 entries Search:

	Sepal.Length	Sepal.Width	Petal.Length	Petal.Wid
1	5.1	3.5	1.4	
2	4.9	3	1.4	
3	4.7	3.2	1.3	
4	4.6	3.1	1.5	
5	5	3.6	1.4	
6	5.4	3.9	1.7	

Showing 1 to 6 of 150 entries

Previous 1 2 3 4 5 … 25 Next

Dataframe and matrix share several features. Depending on the task (graphics or regression) we have to covert one to the other.



Useful functions

Environment

```
summary(x) # Main function of R, shows object results. Varies
rm() # Removes a given object
ls() # List of every object in the environment
list.files() # List of files into the working directory
rm(list = ls()) # Eliminates every object inside the environme
View() # Visualize dataframes
dim() # Dimensions
nrow() # Number of rows in a matrix/dataframe
ncol() # Number of cols in a matrix/dataframe
length() # Number of elements in a vector
head() # Top elements in a matrix/dataframe
tail() # Bottom elements in a matrix/dataframe
names() # Get the name of dataframes, lists, matrices
dimnames() # Get the rownames and colnames
```

Character or factor

```
unique() # Show unique values
sort() # Sort the elements inside a vector, work alphabeticall
table() # Counts of elements
levels() # Shows the unique values in a factor variable
```

Numeric

```
log(x) # Natural logarithm
exp(x) # Exponential
max(x) # Maximum value
min(x) # Minimum value
mean(x) # Mean
var(x) # Variance
sd(x) # Standard deviation
quantile(x) # Quantiles
median(x) # Median value
round(x, digits) # Round numeric values
cor(x, y) # Correlation
rank(x) # Creates a ranking given the elements in a vectors
```

Next session

1. Read and write data frames 1. CSV
    1. Excel
    2. Stata, SPSS, etc.
2. Control structures and loops
    1. If statements
    2. While loop
    3. For loop