Lab 1 - Introduction to R

PROBABILITY AND STATISTICS - LABORATORY

BACHELOR'S DEGREE IN COMPUTER SCIENCE

A.Y. 2022/2023

DOTT. MATTEO CALGARO
matteo.calgaro_01@univr.it

Aknowledgements

- The current slides are a re-visited version of those created by **Prof. Silvia Francesca Storti**.
- The program of the course for the Laboratory part is very similar to the course kept by Prof. Storti during the previous years.
- This year **R** has been adopted, instead of **Matlab**, as programming language for the laboratory.

Introduction to 

- **R** was started by professors Ross Ihaka and Robert Gentleman as a programming language to teach introductory statistics at the University of Auckland in 1993.
- The language took heavy inspiration from the **S** programming language.
- The **Comprehensive R Archive Network** (CRAN) was founded in 1997 by Kurt Hornik and Fritz Leisch to host R's source code, executable files, documentation, and user-created packages.
- CRAN originally had three mirrors and 12 contributed packages. As of December 2022, it has 103 mirrors and 18,976 contributed packages.
- The **R Core Team** was formed in 1997 to further develop the language.

Curiosity: The name of the language comes from being an S language successor and the shared first letter of the authors, Ross and Robert.

Strengths and Weaknesses

Strengths

- Wide range of functions and libraries that allow for a variety of statistical analyses.
- Open source
- It has a large and active community of users and developers, providing support and resources
- Highly customizable, allowing for the creation of tailored analysis and visualization tools.

Weaknesses

- The user interface can be unintuitive and less user-friendly compared to other data analysis software such as SAS or SPSS.
- It is an interpreted language, slower in executing some operations compared to other programming languages.
- Its memory management can be inefficient for large datasets.
- The documentation for some R packages can be incomplete or unclear.

Typical Uses

- Statistical analysis (regression, hypothesis testing)
- Data visualization
- Machine learning (clustering, classification, prediction)
- Data manipulation (cleaning data and pre-processing)
- Time series analysis (useful in economics and finance)
- Web scraping (data mining from web pages)
- Simulation and optimization
- Scientific research (bioinformatics, genomics, medical research)

Packages

An R package is a collection of functions, data sets, and documentation that can be used to extend the functionality of the R language.

Some of the most important and popular R packages include:

- **ggplot2**: for data visualization
- **dplyr**: for data manipulation, including filtering, grouping, and summarizing data.
- **tidyr**: for cleaning and reshaping data.
- **caret**: for machine learning.
- **lubridate**: for working with dates and times.
- **stringr**: for working with strings, including functions for pattern matching, string manipulation, and regular expressions.
- **reshape2**: for transforming data between wide and long formats.
- **data.table**: for data manipulation and aggregation, particularly useful for working with large datasets.

To install and use a package:

```
> install.packages("package_name") # install the package
> library("package_name") # load the package
```

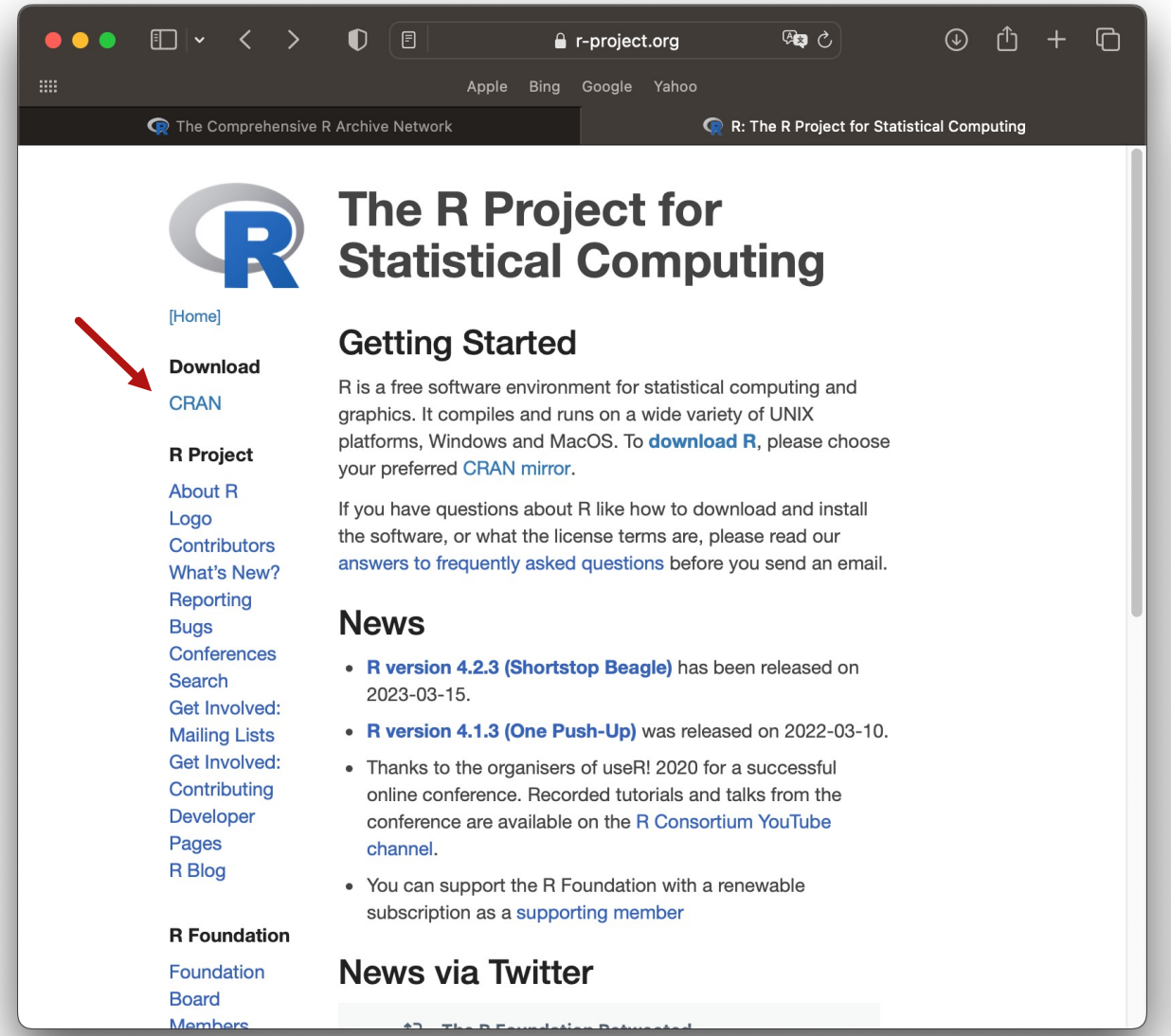
Package repositories

- **CRAN** is the primary repository for R packages and tools. The packages on CRAN are contributed by R users and developers from around the world, and undergo a rigorous review process before being accepted for inclusion in the repository. This helps ensure that the packages are of high quality and are useful for a wide range of users.
- **Bioconductor** is a collection of R packages and tools specifically designed for the analysis and interpretation of genomic data. It was created to address the unique challenges of working with large-scale genomic data sets, such as gene expression data, DNA sequencing data, and microarray data. The packages on Bioconductor undergo a rigorous review process before being accepted for inclusion in the repository.
- **GitHub**, while not specifically designed for R packages, GitHub is a popular platform for sharing and collaborating on code, including R packages.

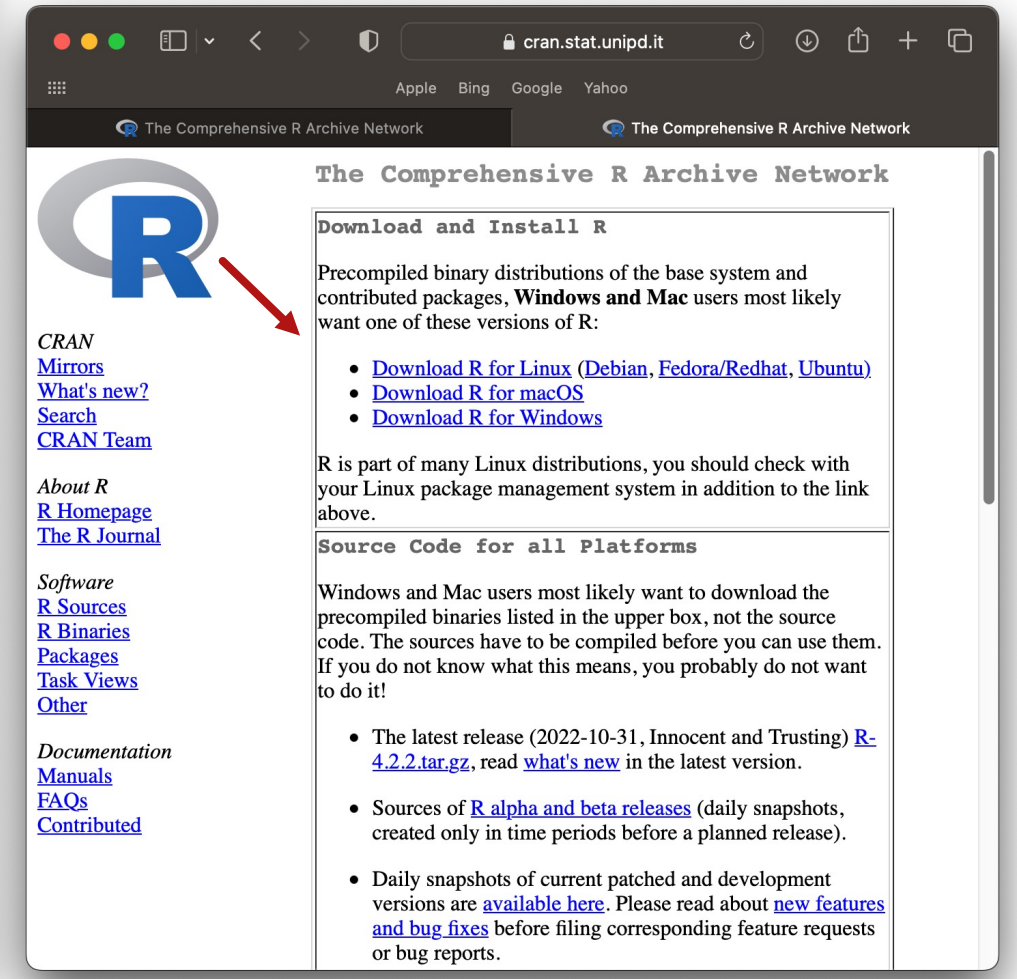
Download R

To download R, you can follow these steps:

- Go to the official R website at https://www.r-project.org/
- Click on the "CRAN" link on the left side of the page.
- Choose a CRAN mirror location near you from the list of mirror sites (*e.g.*, Padova).
- Click on the link to download the appropriate version of R for your operating system (*e.g.*, Windows, Mac, Linux).
- Follow the installation instructions for your operating system to install R on your computer.
- Once installed, you can launch R by double-clicking the R icon on your desktop (if you chose to create one during installation) or by typing "R" into your terminal or command prompt.

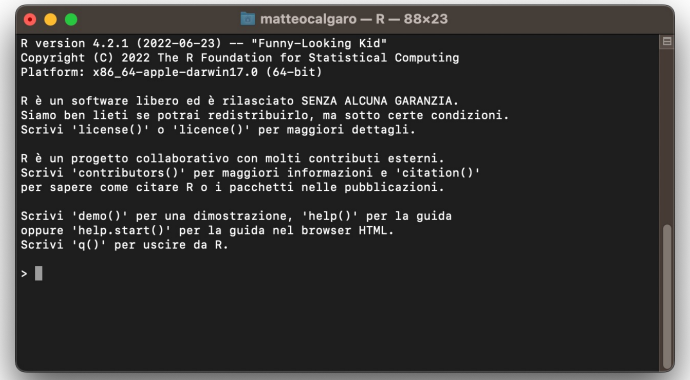


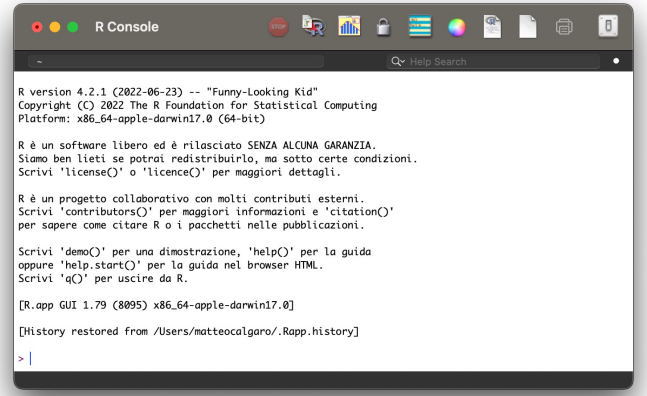
www.r-project.org

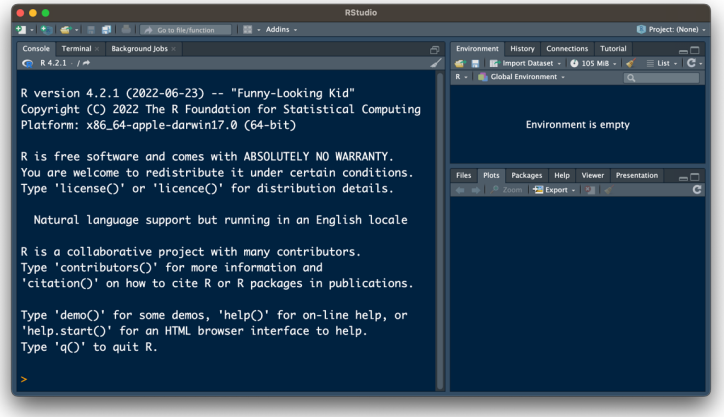


cran.stat.unipd.it

Using R







From Terminal:

- Just type R in your terminal

From R Graphical User Interface (GUI):

- Search the R app among the installed programs

From Rstudio (suggested):

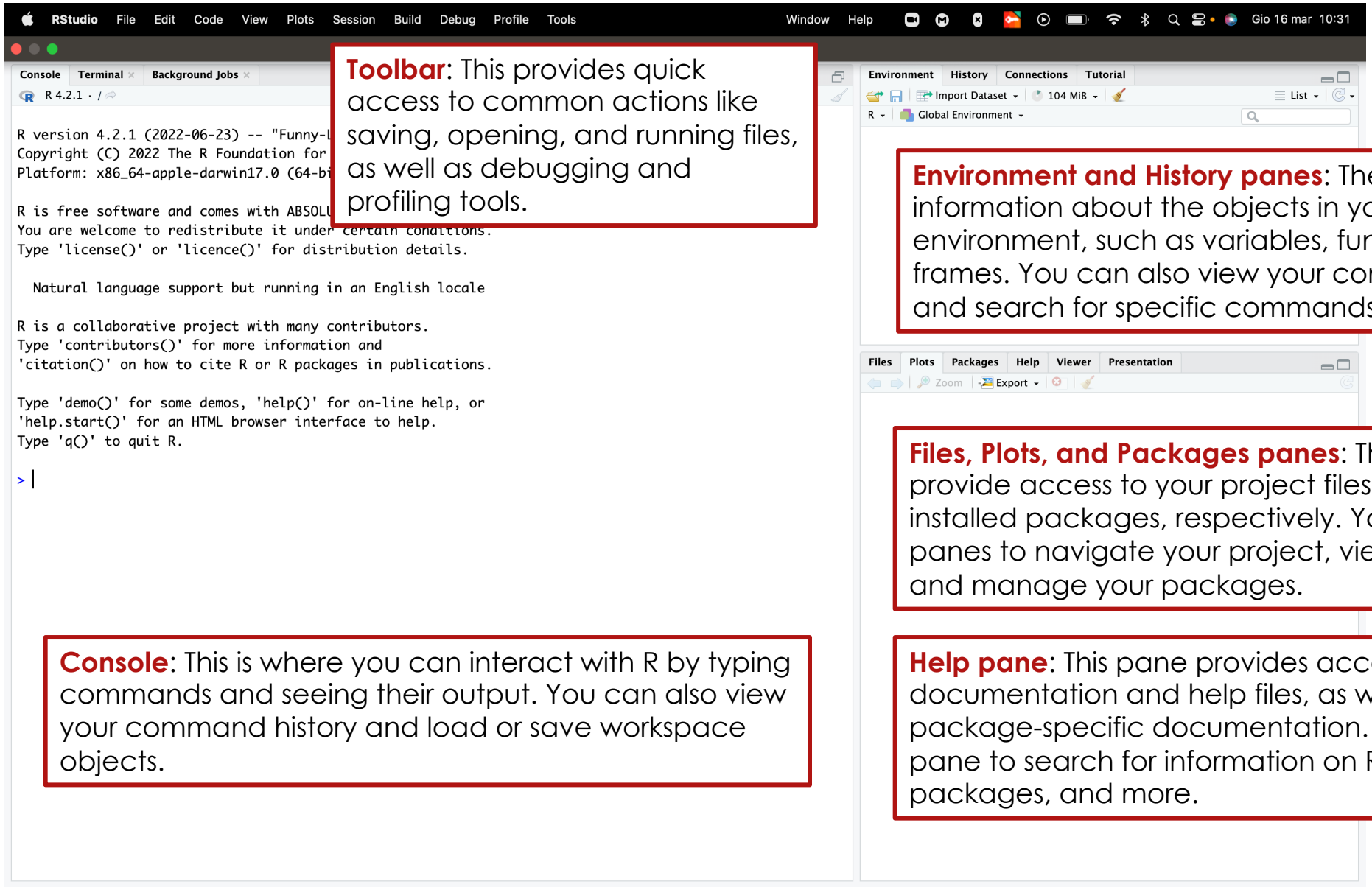
- Launch RStudio after the installation of R and RStudio.

RStudio is an Integrated Development Environment (IDE) for R and Python. It includes a console, syntax-highlighting editor that supports direct code execution, and tools for plotting, history, debugging, and workspace management. RStudio is available in open source and commercial editions and runs on the desktop (Windows, Mac, and Linux).

Once R has been installed, to obtain RStudio:

- Go to the official RStudio website at posit.co/download/rstudio-desktop/.
- Click on the link to download the appropriate version of RStudio for your operating system (*e.g.*, Windows, Mac, Linux).
- Follow the installation instructions for your operating system to install RStudio on your computer.
- Once installed, you can launch RStudio by double-clicking the RStudio icon.

R Studio - Interface



Toolbar: This provides quick access to common actions like saving, opening, and running files, as well as debugging and profiling tools.

Environment and History panes: These panes display information about the objects in your R environment, such as variables, functions, and data frames. You can also view your command history and search for specific commands.

Files, Plots, and Packages panes: These panes provide access to your project files, plots, and installed packages, respectively. You can use these panes to navigate your project, view your plots, and manage your packages.

Console: This is where you can interact with R by typing commands and seeing their output. You can also view your command history and load or save workspace objects.

Help pane: This pane provides access to the R documentation and help files, as well as any package-specific documentation. You can use this pane to search for information on R functions, packages, and more.

Your first commands

You can execute commands in the R console, for example to calculate a mathematical expression:

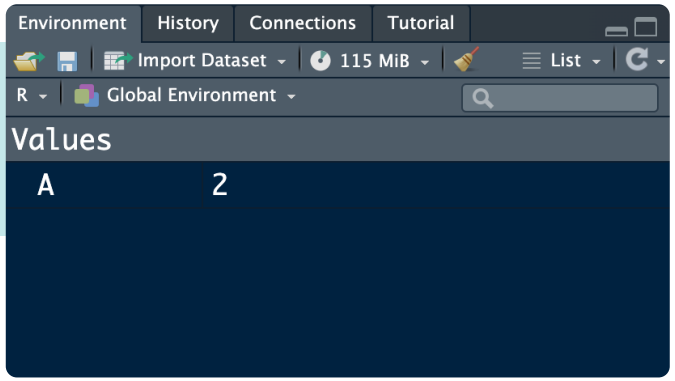
```
> 1+1
[1] 2
```

Or write the same expression in a .R script file and then press <Ctrl> + Enter, to send the selected line to the console.

You can also declare variables by using the <- or = operators:

```
> A <- 1+1
> A
[1] 2
```

You can find the declared variables in the Environment Pane on the top right corner.

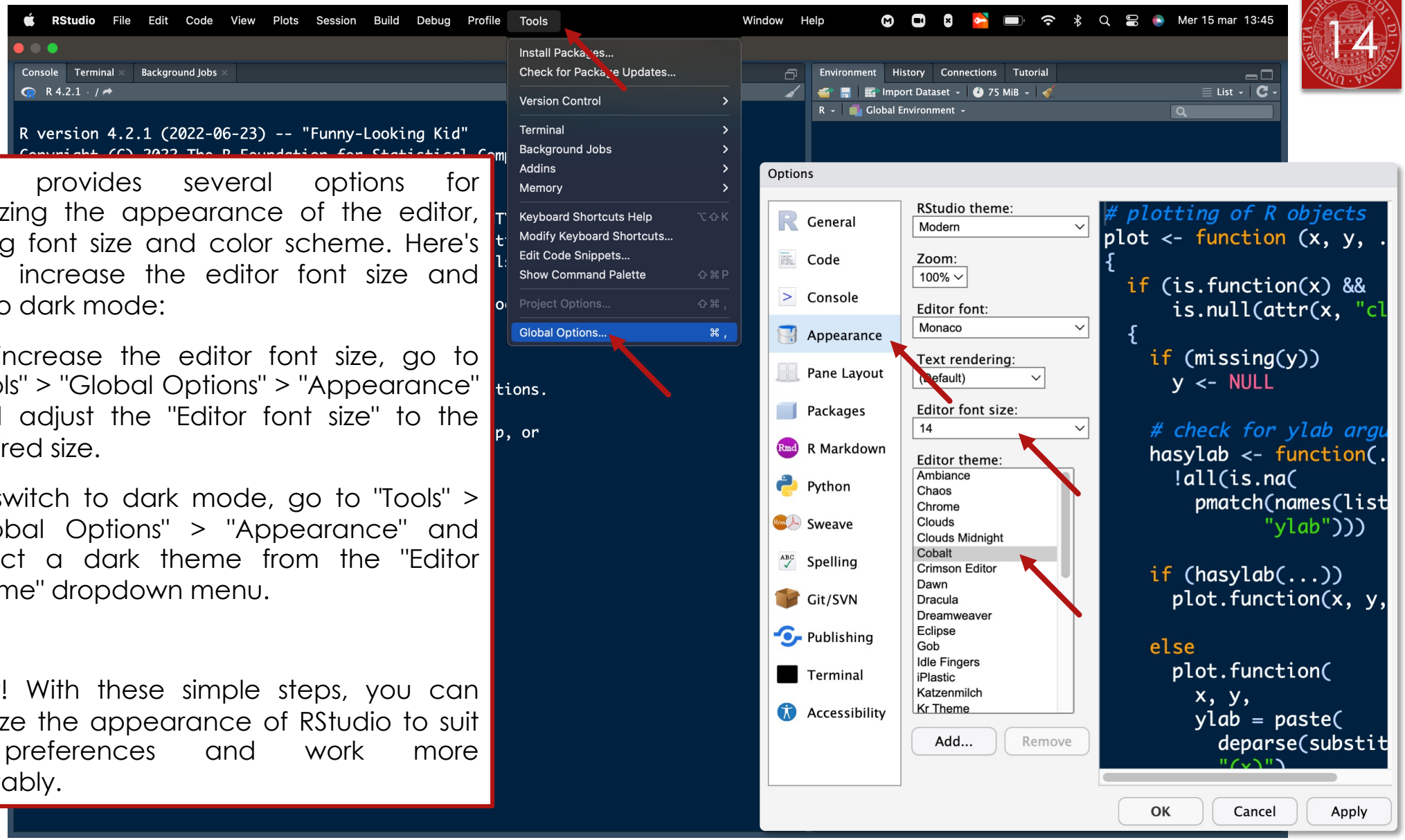


R Studio - Customization

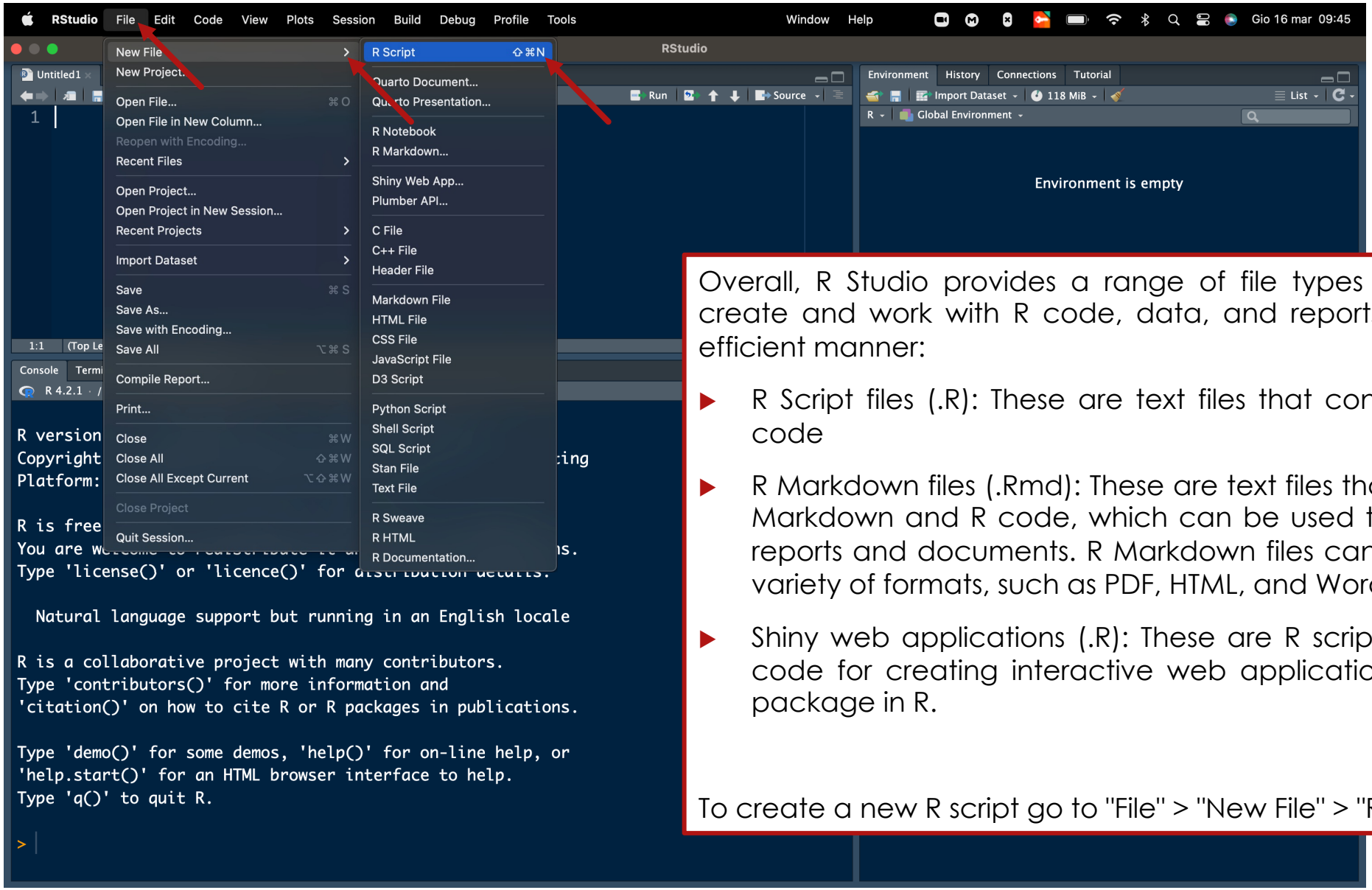
RStudio provides several options for customizing the appearance of the editor, including font size and color scheme. Here's how to increase the editor font size and switch to dark mode:

- To increase the editor font size, go to "Tools" > "Global Options" > "Appearance" and adjust the "Editor font size" to the desired size.
- To switch to dark mode, go to "Tools" > "Global Options" > "Appearance" and select a dark theme from the "Editor Theme" dropdown menu.

That's it! With these simple steps, you can customize the appearance of RStudio to suit your preferences and work more comfortably.



R Studio – Create a .R file



Overall, R Studio provides a range of file types that allow you to create and work with R code, data, and reports in a flexible and efficient manner:

- R Script files (.R): These are text files that contain executable R code
- R Markdown files (.Rmd): These are text files that contain a mix of Markdown and R code, which can be used to create dynamic reports and documents. R Markdown files can be exported to a variety of formats, such as PDF, HTML, and Word.
- Shiny web applications (.R): These are R script files that contain code for creating interactive web applications using the Shiny package in R.

To create a new R script go to "File" > "New File" > "R Script"

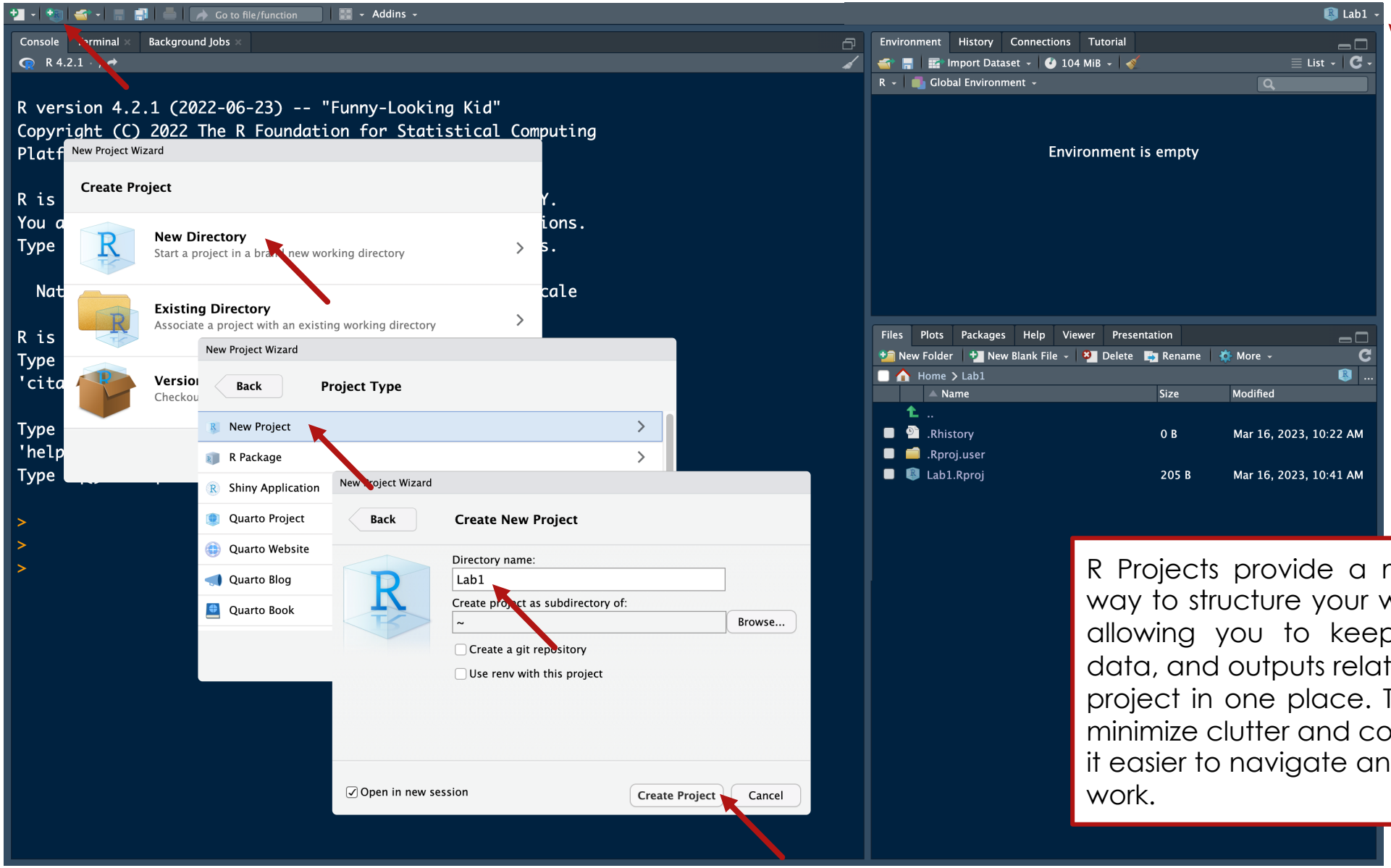
R Studio – Decide what to use

The creation of a .R or a report .Rmd file is not mandatory, commands can be directly written in the console. The decision depends on the task you're trying to accomplish and your personal preferences.

Using the console can be useful for quick and simple tasks, such as exploring data, trying out a new function, or performing basic calculations. It allows you to quickly experiment with code and see the results in real-time. However, it can be difficult to keep track of what you've done in the console, and it can be harder to reproduce your work if you need to do it again later.

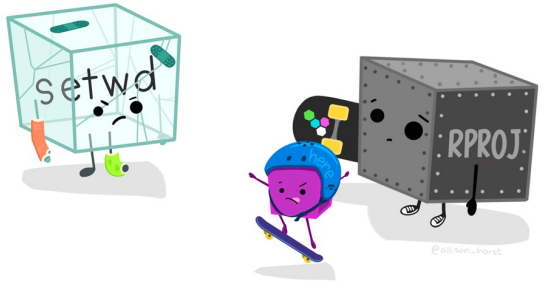
On the other hand, using an R script can be useful for more complex tasks, such as data cleaning, data analysis, and statistical modeling. It allows you to write, test, and organize your code in a more structured way, making it easier to understand, reproduce, and share your work. It also provides an audit trail of your work, which can be useful for documentation and collaboration (especially when using .Rmd files).

R Studio – Work inside a project



R Projects provide a more organized way to structure your work in R Studio, allowing you to keep all your files, data, and outputs related to a specific project in one place. This can help to minimize clutter and confusion, making it easier to navigate and manage your work.

Exercise 1– Create the Lab1



Artwork by @allison_horst

To organize the scripts and the other content of this first laboratory lesson create an R Project called **Lab1**. From the RStudio IDE, "Files" pane, create 3 new folders into the Project's directory, named **scripts**, **plots**, and **data.**

Hints:

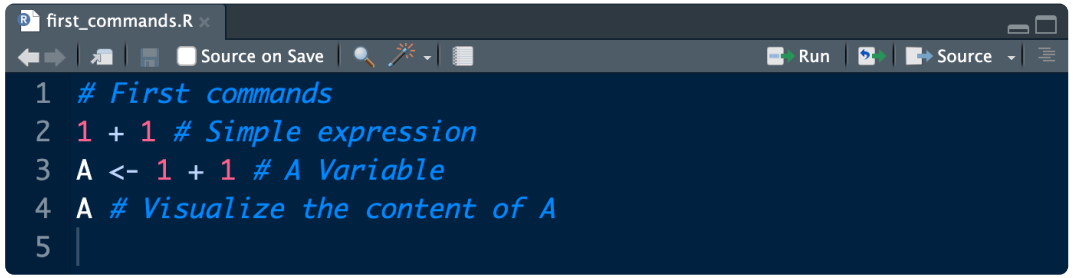
- Follow the steps of the previous slide to create the R Project.
- Use the "New folder" button into the "Files" pane to create a new directory.

Internal documentation

When you write your code in the .R script files you can add comments to help you interpret the code. These are lines of code which are not executed. These can help to:

- Improve readability (understand what your code is doing).
- Provide context (background information that is not immediately apparent from the code itself).
- Document code (making it easier to maintain and update in the future)
- Debug (identify and isolate errors or bugs)
- Collaboration (making it easier for multiple developers to understand and contribute)

Use the # character before the text to comment it.



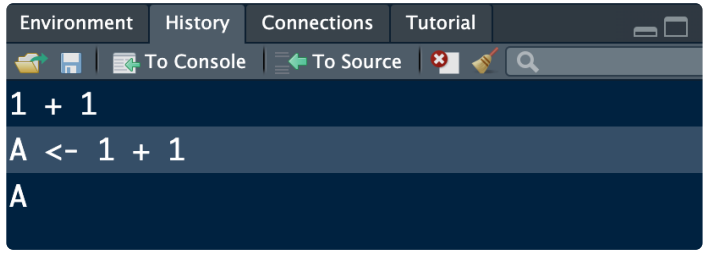
Usually, comments are at the beginning of the script or just after the declaration of a variable or a function.

Command history

In R, you can see previous commands by using the up arrow key in the console. Each time you press the up arrow, the previous command you typed will appear in the console. You can then modify or execute that command as needed.

Alternatively, by using the `history()` function in the console or by navigating to the History pane, you can access the history of the commands you've executed during the current session.

You can send one or more selected lines to the console or active .R script



You can also import/export the history by pressing the first two icons of the History pane.

To get help

To access the help documentation for a specific function or package, simply type `?function_name` or `?package_name` in the console and press Enter.

```
> # Get help for the mean() function
> ?mean
```

By using the `??"search term"` syntax a list of all help pages that contain the search term is returned.

You can also use the `help(function_name)` or `help(package_name)`

```
> # This time get help using the help() function
> help(mean)
```

In the .R script, the combination of <ctrl> + F1 keys when a function name is selected will open its documentation.

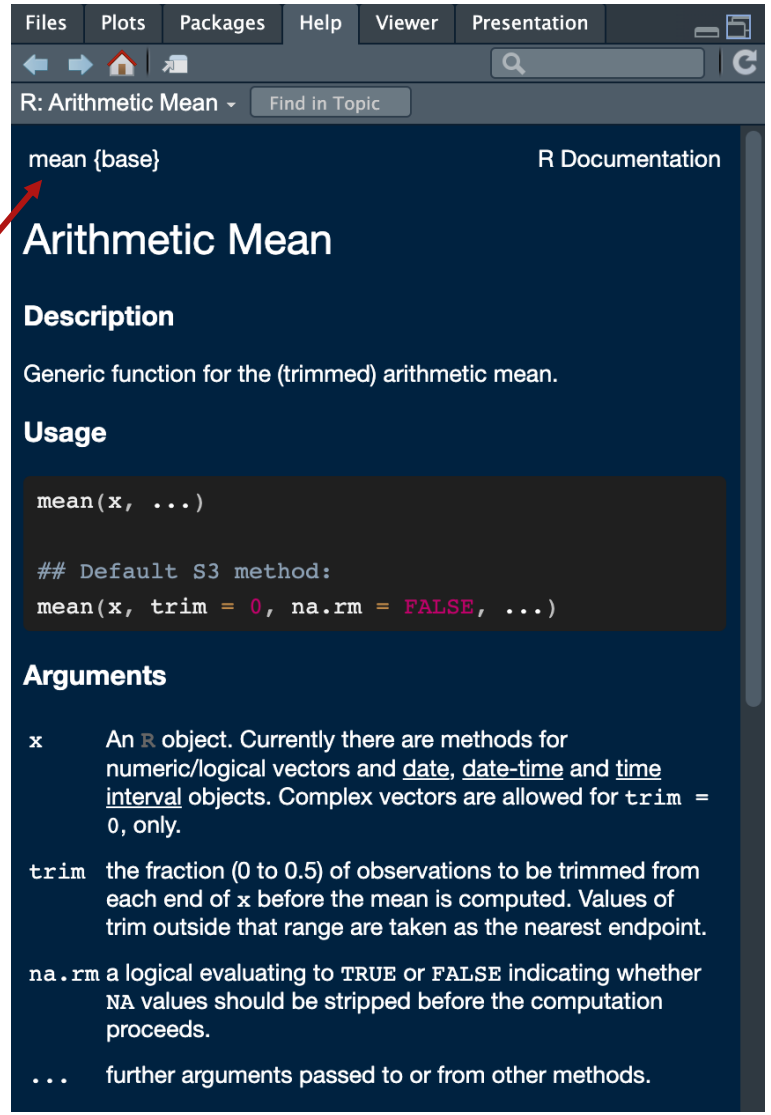
The Help pane is the place where all the helping information is shown. If this is not enough, there are many online resources available for learning R and getting help with specific problems (*e.g.*, R documentation website, RStudio community and Stack overflow)

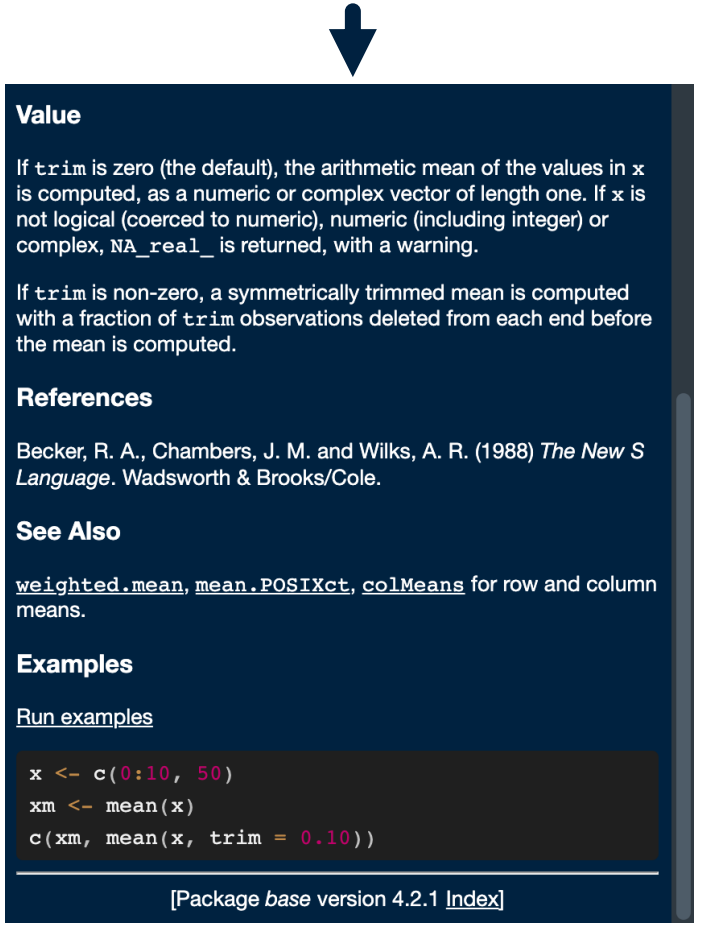
R Studio – The Help pane

It is usually located in the bottom right corner of RStudio interface

The name of the function and its package. Omonym functions exist, but they belong to distinct packages

Structure: the documentation page for a generic function is usually made of a short description, type of usages, a list of the function arguments, its output (called Value), some references, related functions, and working examples



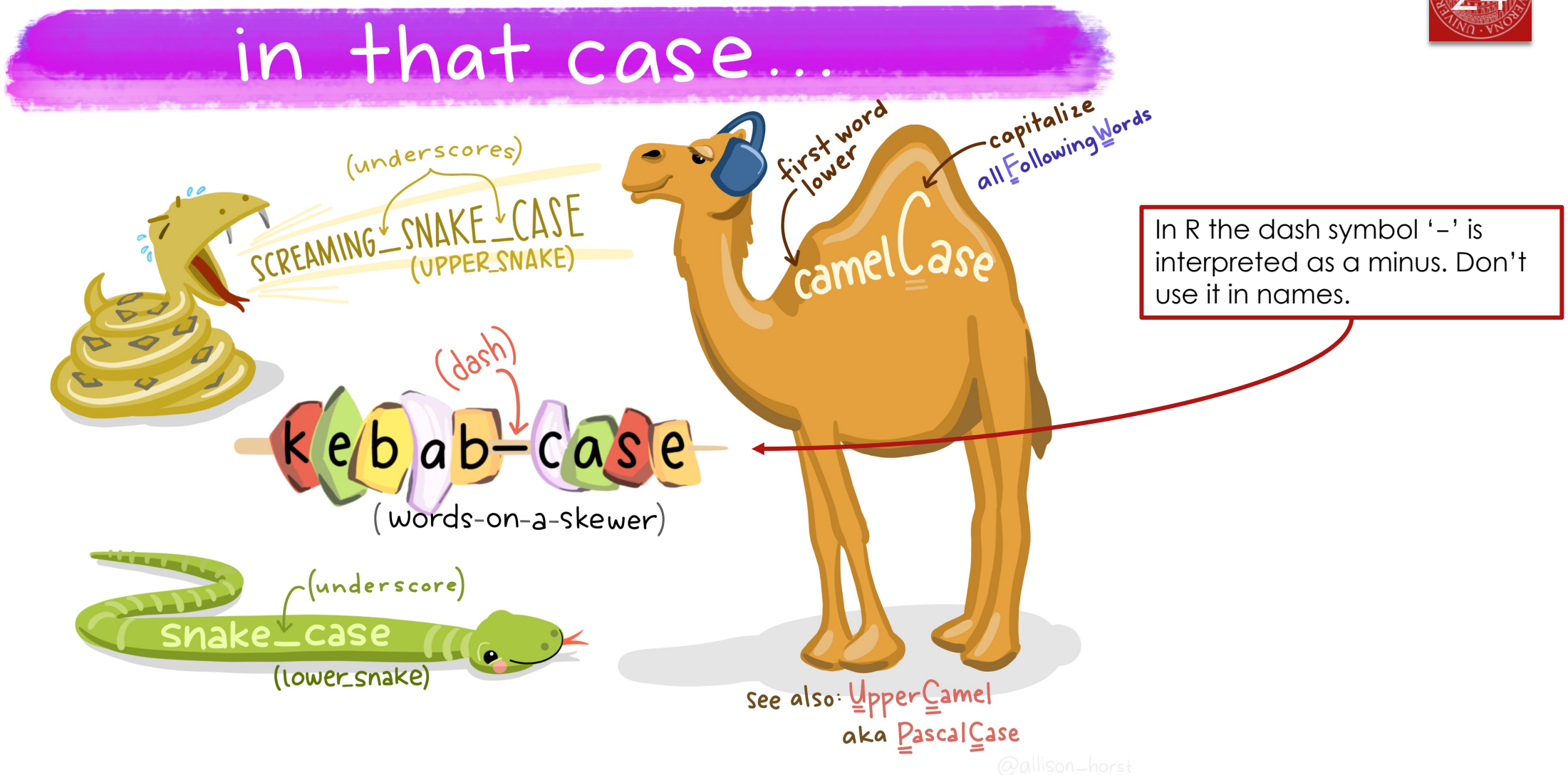


Variables

When you want to store the result of a computation for reuse, or to give it a sensible name and make your code more readable, define a variable:

- Variable names must begin with a letter.
- Names can include any combinations of letters, numbers, and underscores.
- R is case sensitive (**A** ≠ **a**).
- Avoid the following names: **i**, **j**, **pi**, **true**, **false**, and all built-in R function names such as **length**, **size**, **plot**, **cos**, **log**, …
- It is good programming practice to name your variables to reflect their function in a program rather than using generic **x**, **y**, **z** variables
- No need to make any type declarations or dimension statements (useful preallocation)

Some best practices for variables and function names



In R the dash symbol '–' is interpreted as a minus. Don't use it in names.

Artwork by @allison_horst

Variables – Generic examples

At the prompt, type the following code and press enter:

```
> students <- 100
```

A variable, **students**, of type numeric is created. The value **100** is stored in that memory location called students.

To replace the value 100, just overwrite the content of the variable:

```
> students <- 102
```

You can check its content both by typing the variable name in the console or by looking at the Environment pane.

To remove variables from the environment:

```
> remove(students) # Removes a single variable
> remove(list = ls()) # Removes all the variables listed by the
ls() function
```

Or use the broom icon in the Environment Pane

Variables – Vectors

A vector in R is a list of elements (see the **`vector()`** function documentation for more details). A numeric vector, for example:

```
> v1 <- c(2, 5, 1)
```

A vector of 3 elements.

To access the elements of a vector use squared brackets:

```
> v1[1]
[1] 2
```

The indexing starts from 1. R is not 0-based index!

Vectors can also be created by appending one vector to another:

```
> r <- c(2, 4, 10)
> w <- c(9, -6, 4)
> u <- c(r, w)
[1] 2 4 10 9 -6 4
```

Variables – Vector sequence

The colon operator '**:**' generates a vector of regularly spaced elements:

```
> x1 <- 1:10
> x1
[1] 1 2 3 4 5 6 7 8 9 10
```

A vector of ten elements.

To create specific sequences the **`seq()`** function does the job:

```
> x2 <- seq(from = 0, to = 0.5, by = 0.1)
> x2
[1] 0.0 0.1 0.2 0.3 0.4 0.5
```

`seq_along()` and **`seq_len()`** functions are related to **`seq()`**. The first creates a vector with as many elements as the vector inside the parenteses, the second create a list of as many element as indicated by an integer argument.

Variables – Vector repetitions

To repeat a vector use the **`rep()`** function:

```
> r1 <- rep(c(1, 2, 3), times = 3)
> r1
[1] 1 2 3 1 2 3 1 2 3
```

A vector of 1,2,3 repeated 3 times.

To repeat each element a specific number of times:

```
> r2 <- rep(c(1, 2, 3), each = 3)
> r3 <- rep(c(1, 2, 3), times = c(1, 2, 3))
> r2
[1] 1 1 1 2 2 2 3 3 3
> r3
[1] 1 2 2 3 3 3
```

Functions for vectors

Some useful functions to use with vectors are summarized in the following table:

Function	Description
`length()`	Gives the length of a vector
`max(), min()`	Compute the maximum or the minumum value
`sum()`	Sum all the elements
`mean()`	Compute the mean
`cumsum()`	Compute the cumulative sum for each element

Variables – Matrices

To create a matrix:

- Vectors can be concatenated as columns using **`cbind()`**.
- Vectors can be concatenated as rows using **`rbind()`**.
- The function matrix() can be used. The number of rows and the number of columns are specified.

```
> mat1 <- cbind(c(1, 2, 3), c(4, 5, 6))
> mat2 <- rbind(c(1, 2, 3), c(4, 5, 6))
> mat3 <- matrix(data = c(1, 2, 3, 4, 5, 6), nrow = 3, ncol = 2)
```

```
> mat1
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
```

```
> mat2
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
```

```
> mat3
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
```

Variables – Matrices

To access the elements of a matrix, two indexes are necessary, one for the rows and the other for the columns:

```
> mat1 <- cbind(c(1, 2, 3), c(4, 5, 6))
> mat1[2,2] # element in second row, second column
[1] 5
```

```
> mat1
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
```

When all the elements of a row/column are requested, the opposite index should remain empty (indicating that all elements are of interest):

```
> mat1[,2] # elements of the second column
[1] 4 5 6
> mat1[2,] # elements of the second row
[1] 2 5
```

Variables – Matrices

To obtain a subset of a matrix:

```
> mat1 <- cbind(c(1, 2, 3), c(4, 5, 6))
> mat1[2:3,1:2] # second and third rows, both columns
     [,1][,2]
[1,]    2    5
[2,]    3    6
```

```
> mat1
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
```

To delete a row or a column of a matrix:

```
> mat1[-1,] # remove the first row
     [,1][,2]
[1,]    2    5
[2,]    3    6
```

The same approach can be used for vectors. Indexes of the elements to be removed are preceeded by a minus symbol.

Variables - Matrices

Special matrices:

```
> mat_zero <- matrix(0, 3, 3) # Matrix 3x3 of zeroes
> mat_ones <- matrix(1, 3, 3) # Matrix 3x3 of ones
> mat_id <- diag(3) # Identity matrix 3x3
```

```
> mat_zero
     [,1] [,2] [,3]
[1,]    0    0    0
[2,]    0    0    0
[3,]    0    0    0
```

```
> mat_ones
     [,1] [,2] [,3]
[1,]    1    1    1
[2,]    1    1    1
[3,]    1    1    1
```

```
> mat_id
     [,1] [,2] [,3]
[1,]    1    0    0
[2,]    0    1    0
[3,]    0    0    1
```

Operations with vectors and matrices

Arithmetic operations such as addition, subtraction, multiplication, and division are performed element-wise:

```
> A <- matrix(data = 1:9, nrow = 3)
> B <- A + 1
```

```
> A
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
```

```
> B
     [,1] [,2] [,3]
[1,]    2    5    8
[2,]    3    6    9
[3,]    4    7   10
```

Other element-wise operations are the exponentiation, and the square root:

```
> C <- A^2
> D <- sqrt(A)
```

```
> C
     [,1] [,2] [,3]
[1,]    1   16   49
[2,]    4   25   64
[3,]    9   36   81
```

```
> D
         [,1]     [,2]     [,3]
[1,] 1.000000 2.000000 2.645751
[2,] 1.414214 2.236068 2.828427
[3,] 1.732051 2.449490 3.000000
```

To obtain summary statistics, **`sum()`**, **`mean()`**, **`min()`**, and **`max()`** functions can be used:

```
> sum(A)   # 45        > min(A)   # 1
> mean(A)  # 5         > max(A)   # 9
```

Operations with vectors and matrices

Element-wise operation can be performed even between matrices (as long as they are comformable):

```
> A <- matrix(data = 1:9, nrow = 3) # 3x3
> E <- matrix(data = 1:4, nrow = 2) # 2x2
> A + E
Error in A + E : non-conformable arrays
```

Matrix operations follow the rules of linear algebra (*i.e.*, the required size and shape of the inputs in relation to one another depends on the operation). The **`%*%`** operator performs the matrix multiplication, **`t()`** function the transpose:

```
> v1 <- c(1, 2, 3)
> v2 <- c(4, 5, 6)
> v1 * v2
[1] 4 10 18
# Still element-wise
```

```
> t(v1) %*% v2
[1] 32
```

```
> v1 %*% t(v2)
     [,1] [,2] [,3]
[1,]    4    5    6
[2,]    8   10   12
[3,]   12   15   18
```

Other matrix functions

Other useful functions to use for matrices are summarized in the following table:

Function	Description
`det()`	Gives the determinant of a given matrix
`qr()`	Compute the QR decomposition
`solve()`	Inverse of a matrix
`nrow(), ncol()`	Gives the number of rows and the number of columns
`rowSums(), colSums()`	Gives the sum of all the elements by row or by columns
`rowMeans(), colMeans()`	Gives the mean for each row or for each column
`dim()`	Returns the number of rows and the number of columns

Character – Text variables

At the command prompt type:

```
> month <- "March" # The single quote ` can be also used
```

A variable, **month**, of type character is created. The value **March** is stored in that memory location called month.

Function	Description
nchar()	Gives the length of a string
paste(), paste0()	Concatenate two strings
toupper(), tolower()	Change case

As for numeric vectors, also vectors of characters can be created:

```
> months <- c("January", "February", "March")
```

Lists, arrays, and data frames

Lists, arrays, and data frames are among the most flexible data types in R.

A List is a collection of similar or different types of data. Use the **`list()`** function to create a list. For example:

```
> l1 <- list("Everest", "Nepal", 8848, TRUE)
> l2 <- list(81, 82, 90, 56)
```

l1 and **l2** are two lists. **l1** contains character, numeric and boolean values. **l2** contains numeric only elements.

Access its element like a vector :

```
> l1[1]
[[1]]
[1] "Everest"
```

Lists, arrays, and data frames

An Array is a data structure which can store data of the same type in more than two dimensions. Vectors are uni-dimensional, matrices are bi-dimensional, arrays are multi-dimensional:

```
> a1 <- array(c(1:12), dim = c(2,3,2))
> a1
, , 1
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6

, , 2
     [,1] [,2] [,3]
[1,]    7    9   11
[2,]    8   10   12
```

Accessing the elements: **`a1[i, j, k]`** is the element in position i-th of the first dimension, j-th of the second dimension, and k-th of the third dimension.

Lists, arrays, and data frames

A data frame is a two-dimensional data structure which can store data in tabular format. Data frames have rows and columns and each column can be a different vector. And different vectors can be of different data types:

```
> df1 <- data.frame(
      mount = c("Everest", "K2", "Fuji"),
      height = c(8848, 8611, 3776),
      todo = c(TRUE, TRUE, FALSE))
> df1
    mount height  todo
1 Everest   8848  TRUE
2      K2   8611  TRUE
3    Fuji   3776 FALSE
> df1$mount # Access one column
[1] "Everest" "K2"      "Fuji"
```

Concatenate: use **`cbind()`** and **`rbind()`** to concatenate vertically or horizontally two data frames.

Access: the **`df1$mount`**, **`df1[, "mount"]`**, and the **`df1[,1]`** are equivalent ways to access the first column.

Factors

A Factor is a data structure that is used to work with categorizable data. The **`factor()`** function can e used to create a factor. Once a factor is created, it can only contain predefined set values called levels.

```
> f1 <- marital_status <- factor(c("married", "single", "single",
"divorced", "married"))
> f1
[1] married  single   single   divorced married
Levels: divorced married single
> levels(f1)
[1] "divorced" "married"  "single"
> f1[1] <- "hey"
Warning message:
In `[<-.factor`(`*tmp*`, 1, value = "hey") :
  invalid factor level, NA generated
```

if, ifelse, for, while

```
if (condition) {
    # statements
} else if (condition) {
    # statements
} else {
    # statements
}
```

```
ifelse(condition, yes = …, no = …)
```

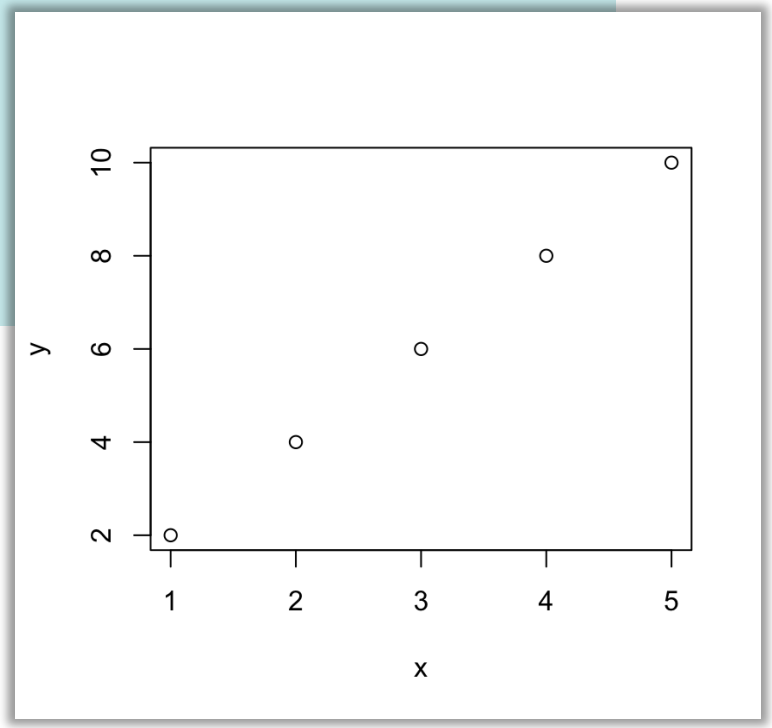
```
for (variable in vector) {
    # statements
}
```

```
while (condition) {
    # statements
}
```

Graphical representation – plot()

To create a plot you can use the built-in **`plot()`** function. Here's an example of how to create a basic scatterplot:

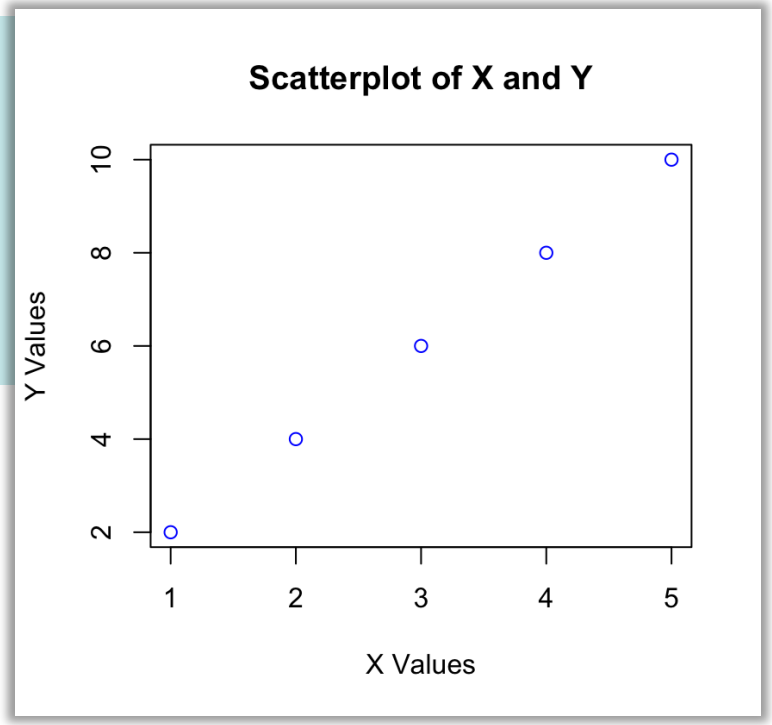
```
# Create some sample data
x <- c(1, 2, 3, 4, 5)
y <- c(2, 4, 6, 8, 10)

# Create a scatterplot
plot(x, y)
```



Graphical representation – Customization

To customize the appearance of the plot optional arguments can be passed to the **`plot()`** function. For example, labels to the axes, color of the points, and a title to the plot:

```
# Customize the scatterplot
plot(x, y,
     xlab = "X Values",
     ylab = "Y Values",
     main = "Scatterplot of X and Y",
     col = "blue")
```



Graphical representation – types of plot

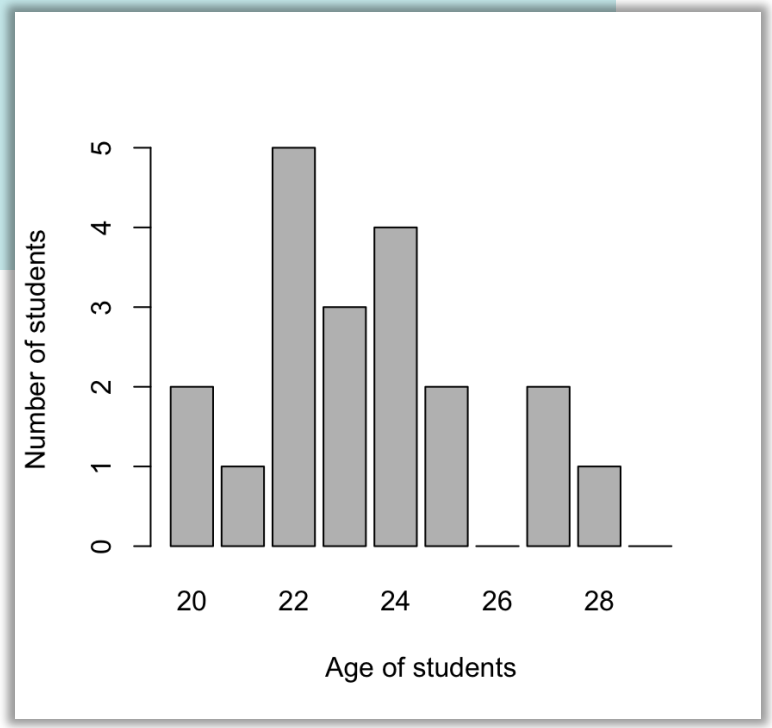
There are many types of plots that can be created in R. Here's a brief overview of some of the most commonly used types:

- **Bar plots (`barplot()`)**: Used to display categorical data, where each bar represents a category and its height represents the frequency or proportion of that category.
- **Line plots (`plot(…,type = "l")`)**: Used to display trends or changes over time, where each data point is connected by a line.
- **Histograms (`hist()`)**: Used to display the distribution of a continuous variable, where the x-axis represents the range of values and the y-axis represents the frequency or proportion of values in each range.
- **Box plots (`boxplot()`)**: Used to display the distribution of a continuous variable, where the box represents the range of the middle 50% of values, the line within the box represents the median, and the whiskers extend to the most extreme values within 1.5 times the interquartile range.
- **Scatter plots (`plot()`)**: Used to display the relationship between two continuous variables, where each data point is represented by a point on a two-dimensional graph.
- **Heatmaps (`pheatmap()`)**: Used to display the relationship between two categorical variables, where the frequency or proportion of each combination of categories is represented by a color.
- **Density plots (`plot(density(…))`)**: Used to display the distribution of a continuous variable using a smooth curve.

Graphical representation - Bar plot

Using the command **`barplot()`**:

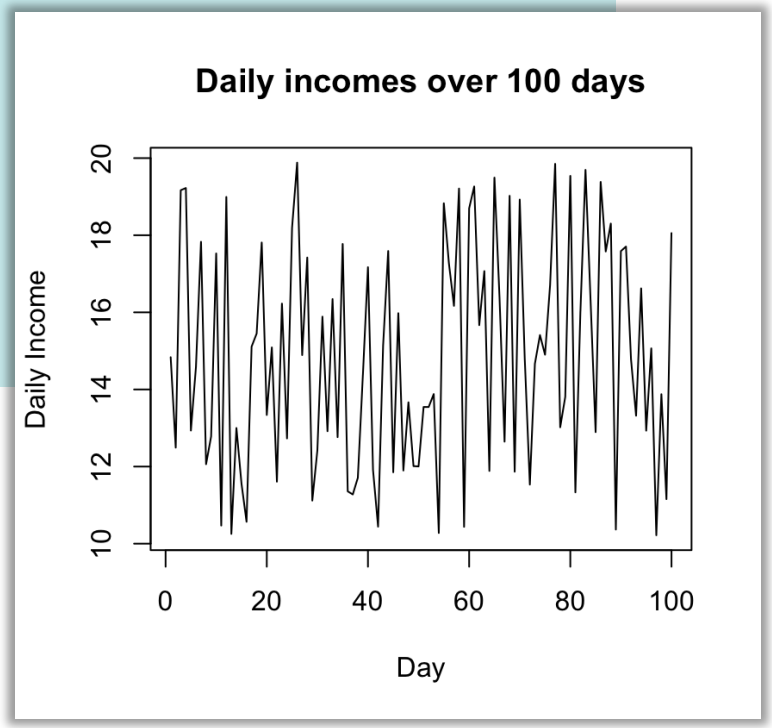
```
ages = 20:29
students = c(2,1,5,3,4,2,0,2,1,0)
barplot(students,
        xlab = "Age of students",
        ylab = "Number of students",
        names.arg = ages)
```



Graphical representation – Line plot

Using the command **`plot(…, type = "l")`**:

```
# Generate 100 random values between 10 and 20
daily_income <- runif(100, min = 10, max = 20)

plot(daily_income,
     type = "l",
     xlab = "Day",
     ylab = "Daily Income",
     main = "Daily incomes over 100 days")
```



Graphical representation - Histogram

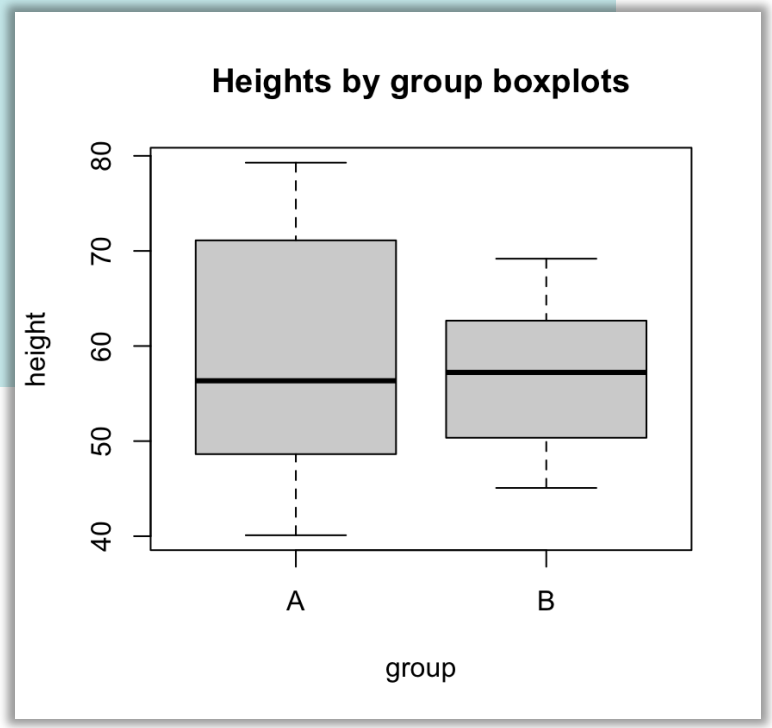
Using the command **`hist()`**:

```
# Generate 100 random values between 40 and 80
weights <- runif(100, min = 40, max = 80)

# To see the first elements
head(weights)

hist(weights,
     breaks = c(40,50,60,70,80))
```

Graphical representation - Boxplot

Using the command **`hist()`**:

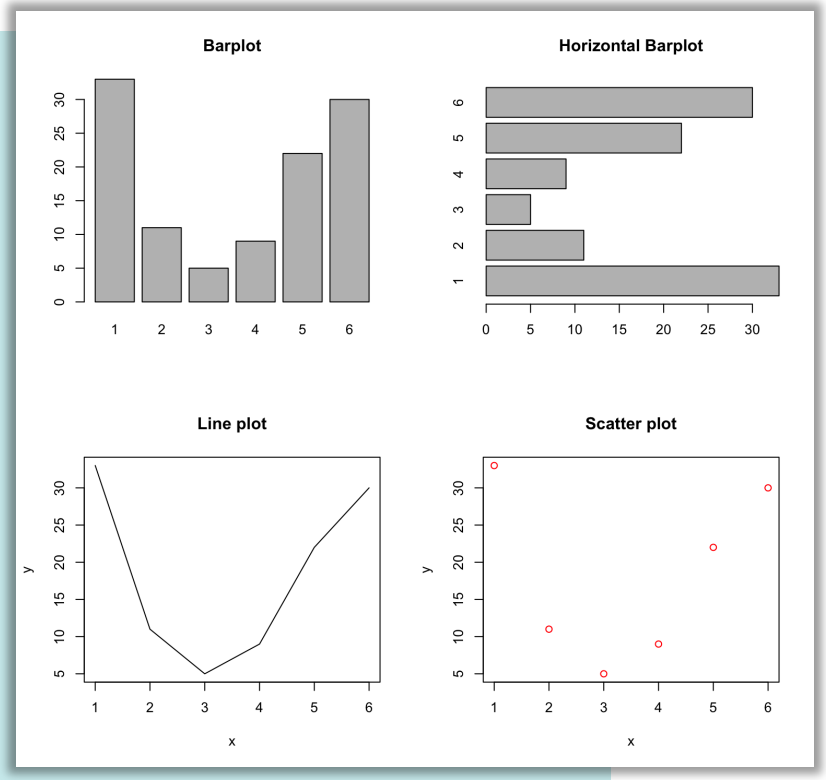
```
# Generate 50 random values between 40 and 80
# Generate 50 random values between 45 and 70
height <- c(runif(50, min = 40, max = 80),
            runif(50, min = 45, max = 70))
group <- rep(c("A", "B"), each = 50)

boxplot(height ~ group,
        main = "Heights by group boxplots")
```



Show many plot types simoultaneously

Using the command **`par(mfrow = c(nrows, ncols))`** it is possible to set the layout for multiple plots:

```
# Define some data
x = 1:6
y = c(33, 11, 5, 9, 22, 30)
# Set the layout, 2 rows and 2 columns
par(mfrow = c(2,2))
# Position [1,1]
barplot(y, main = "Barplot", names.arg = x)
# Position [1,2]
barplot(y, main = "Horizontal Barplot",
        names.arg = x, horiz = TRUE)
# Position [2,1]
plot(x, y, type = "l", main = "Line plot")
# Position [2,2]
plot(x, y, main = "Scatter plot", col = "red")
```



Save and export graphics

To save a plot you can use the **`pdf()`**, **`jpeg()`**, **`png()`**, or **`svg()`** functions to create a new graphics device, and then use the **`plot()`** function to create your plot. Once your plot is created, you can use the **`dev.off()`** function to close the graphics device and save the plot to a file. Here's an example:

```
# Create a new PDF graphics device
pdf("myplot.pdf")

# Create a plot
x <- 1:10
y <- x^2
plot(x, y)

# Close the graphics device and save the plot to a file
dev.off()
```

See **`?pdf`** to see saving options. You can also export from the RStudio interface.

Save and load variables

R variables can be saved using two main formats: .RData or .RDS.

- .RData is a binary file format used to save one or more R objects in a single file. You can save an .RData file using the **`save()`** function in R, and load it back into R using the **`load()`** function. .RData files are convenient for saving and loading multiple objects at once and for preserving the workspace across R sessions.
- .RDS is a binary file format used to save a single R object in a file. It is similar to the .RData format, but it only stores a single object. You can save an .RDS file using the **`saveRDS()`** function in R, and load it back into R using the **`readRDS()`** function. .RDS files are useful when you need to save a single object without including other objects in the file.

Practical considerations: .RDS files are more flexible because you can choose the name of the object when saving it, whereas with .RData, the object name is determined by the name of the object in the R environment at the time of saving.

Save and load tabular data

Often, data to input/export are organised in a tabular format.

In order to read/write them many options are available, depending on the available/desired formats:

- **`read.csv()/write.csv()`** and **`read.table()/write.table()`**: These functions are used to import/export data from/to a comma-separated or tab-separated text file, respectively.
- **`read_excel()/write_excel()`** from the readxl package: This function is used to import/export data from/to an Excel file.
- **`readr::read_delim()/write_delim()`** and **`readr::read_csv()/write_csv()`**: These functions from the readr package are used to import/export data from/to various delimited text files, including CSV, TSV, and others.
- Use the RStudio wizard tool for a guided import procedure.

.R files – scripts and functions

Scripts and functions are two important components that allow you to write and reuse code.

- A script is a text file containing a sequence of R commands that you want to execute. To execute them use the **`source()`** function.
- A function is a block of code that performs a specific task and can be called from other parts of your code. Functions in R are defined using the **`function()`** keyword, followed by a set of arguments and the code to be executed.

Practical considerations: An .R script containing a collection of functions can be sourced in any project in order to make those functions available. The evolution of this concept is represented by a package.

Functions

Here is an example of a simple function that takes two arguments and returns their sum:

```
my_sum <- function(x, y) {
  z <- x + y
  return(z)
}
```

Internal variables are local to the function. Once the function or block finishes executing, the local variable **z** is destroyed and cannot be accessed from outside

You can call this function by passing two values as arguments:

```
my_sum(2,3) # returns 5
```

Local and global variables

A global variable is a variable that is defined outside of any function or block of code and can be accessed from any part of the program. Global variables are created and stored in the global environment, which is a special environment that is created when R starts up. Here's and example

```
x <- 10
my_function <- function(){
    y <- x+1
    return(y)
}
my_function() #returns 11
```

Practical considerations: If you define a variable with the same name inside a function as a global variable, the local variable will take precedence and the global variable will be masked within the function. it's a good practice to avoid using global variables as much as possible and instead pass data between functions using arguments and return values. This makes your code more modular and easier to test and debug.

Debug

Debugging is an important part of writing code in R, as it helps to identify and correct errors in your programs. Here are some ways to debug code in R:

- Print statements to check the values of variables and identify where the program might be going wrong using **`print()`.**
- Step through your code line by line using the **`debug(function_to_debug)`** function.
- Reading the traceback message that shows the sequence of function calls that led up to the error.
- Use the **`browser()`** function, which allows you to pause the program at a specific point and interactively inspect the values of the variables.

```
my_function <- function(x) {
  y <- x + 1
  browser()
  return(y*2)
}
```

Exercise 2 – Basic matrices operations

A. Create the following vectors twice: the first using the colon operator and the second using the **`seq()`** command:
   - 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
   - 2, 7, 12

B. Create a 4x2 matrix of all zeros and store it in a variable (**`mymat`**). Then, replace the second row in the matrix with a vector consisting of a 3 and a 6.

C. Create a vector **`x`** which consists of 20 equally spaced points in the range from –**π** to +**π**. Create a **`y`** vector which is **`sin(x)`**.

D. Create a 4x6 matrix of random integers, each in the range from -5 to 5; store it in a variable (**`mat`**). Create another matrix that stores for each element the absolute value of the corresponding element in the original matrix (**`mat_pos`**).

E. Plot **`exp(x)`** for values of **`x`** ranging from -2 to 2 in steps of 0.1. Put an appropriate title on the plot, and label the axes.

F. Create a vector **`x`** with values ranging from 1 to 100 in steps of 5. Create a vector **`y`** which is the square root of each value in **`x`**. Plot these points. Now, use the **`barplot()`** function instead of **`plot()`** to get a bar chart. Keep both plots together.

Exercise 3 – Graphical representation

A. Load the **`sunspot.year`** dataset from the **`datasets`** package. Use **`data("sunspot.year")`** and then **`sunspot.year`** to load it in the workspace.

B. See the documentation to obtain information about the dataset and create a sequence vector corresponding to the years. Call this variable **`year`**.

C. Create a variable called **`sunspot`**, containing the values from the dataset.

D. Put together the variables into a **`data.frame`** object.

E. Make a line plot of sunspots vs. year.

F. Superimpose data points as red asterisks. Add a second layer to the plot by using the **`points()`** function. Use **`pch = "*"`** and **`col = "red"`** in the **`points()`** arguments.

G. Create a title **`'Sunspots by year'`**.

H. Make a column with 3 panels for the plot created in G., a barplot of sunspots (you can use the **`as.vector()`** function to convert a a data type to a vector data type), and a histogram of sunspots.

I. Save the plot in the **`./plots`** directory of the project as a .png file.

J. Save the data frame as a .csv file in the **`./data`** directory of the project.

Exercise 4 – Axis range and function

A. The first line is the hours of a day, and the second line is the recorded temperature at each of those times. The first value of 0 for the hours represents midnight.

```
  0    3    6    9   12   15   18   21
55.5 52.4 52.6 55.7 75.6 77.7 70.3 66.6
```

B. Plots the data using black '+' symbols.

C. Note that it is difficult to see the point at time 0 since it falls on the y-axis. Change the axis range using **`xlim`** arguments to improve visualization.

D. Create a function which takes as input a measure in cm and returns it in inches (2.54cm = 1inch).

E. Save the plot as a .pdf file in the **`./plots`** directory of the project. Plot it 10cm x 10cm (see the arguments of the **`pdf()`** function).

Solutions

They will be published during the week, before the next lab.