Day 1 Practical 2: Getting staRted

2023-06-05

In a previous practical you learnt how to set up your R and RStudio environment and create an initial workflow with an R project, folder structure and a first R script.

To start this practical:

- Go to your EpiRcourse project, open it
- Open the R script you created

During this practical, you will be prompted to write several lines of code, you can either copy and paste some of the code snippets in this worksheet, but try and code it from scratch (to get the feel of the programming flow!)

Data structures

These are the most basic structures in R. First, explore the code below. Create the same objects in your own script.

```
vector_double <- c(1, 2, 3, 4, 5, 6)

vector_logic <- c(TRUE, FALSE, FALSE, TRUE)

vector_character<- c("A", "B", "C", "D")

vector_integer <- c(1L, 2L, 3L, 4L)
```

Now your script contains new information, necessary to create four new objects, each one containing a different class of data.
But the script is just a log of your work. To actually create these objects you need to execute the code.

How to execute R code

There are different ways in which you can execute (run) the code in an R script:

1. Select the lines of code you wish to run and press ctrl + enter (or click the icon "Run" at the top right of the script)
2. If you wish to run all the contents of the script, click the icon "source" at the top right of your script.

Try running the code above. What happens when you run those lines?. Cn you find the new objects created?

From the lecture notes, what do you do to output the new vector's content in the console window? (You should be able to see results as below)

```
## [1] 1 2 3 4 5 6

## [1]  TRUE FALSE FALSE  TRUE

## [1] "A" "B" "C" "D"

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

Matrices

Matrices are slightly more complex structures than vectors, with two main characteristics:

- A matrix is composed of only one type of data
- A matrix has two dimensions

A command to construct a matrix uses three arguments:

Syntax: matrix_name <- matrix(*data*, *nrow* , *ncol*)

- *data* corresponds to the list of vectors that we want to use in the matrix
- *nrow* is the number of rows where the data will be divided (first dimension)
- *ncol* is the number of columns where the data will be divided (second dimension)

Note: By default, the matrix is filled by columns, unless we specify otherwise using *byrow = TRUE*

The argument *data* can also be a single value with which you want to fill your matrix, like 0 or 10, for example.

Try the following, create two new objects, one will be a 2 X 2 matrix containing the elements of *vector_double* that you have already created. The other object will be an empty matrix filled with the value 2 of size 2 X 3. Name your matrices mat1 and mat2. Check the contents of your matrix, do you get something like this?

```
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6

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

Matrix multiplication is an often required operation when coding. But one condition to do element-wise matrix multiplications, is that the two initial matrices have the same dimensions.

Imagine you need to multiply matrices *mat1* and *mat2*. What can you do?

- Use the internet to search for a method that allows you to "flip" one the matrices so they can be multiplied.
- Create a third matrix (*mat3*) that is the product of *mat1* and the transposed matrix *mat2*.

HINT:the operation you are looking for is called transposition !

The result should look like this:

```
##      [,1] [,2]
## [1,]    2    8
## [2,]    4   10
## [3,]    6   12
```

Data frames

A data frame is a heterogeneous, two-dimensional structure, similar but not exactly the same as a matrix. Unlike a matrix, multiple types of data can be part of a single data frame.

The arguments for the *data.frame* command are simply the columns in the data frame. (**Important**: Each column must have the same number of rows to fit in a data frame).

Data frames do not allow vectors with different lengths. When the length of the vector is less than the length of the data frame, the data frame forces the vector to its length.

Check the following code:

```
data_example <- data.frame(vector_character, vector_double, vector_logic, vector_integer)
```

What happens? Does it execute?

Now try creating the same data frame but only including the first four elements in vector double:

- Following the lecture notes, look how you access and subset a vector
- Create a data frame with all the vectors and the first four elements of *vector_double*

Should look like this:

```
##   vector_character vector_double.1.4. vector_logic vector_integer
## 1                A                   1         TRUE              1
## 2                B                   2        FALSE              2
## 3                C                   3        FALSE              3
## 4                D                   4         TRUE              4
```

Now, can you access the element in the second row, third column and assign it to the new object *new_vec*.

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

Lists

A list is the most complex structure in R. A list can be composed of any type of objects of any dimension!

```
list_example <- list(vector_character,
                     vector_double,
                     vector_logic,
                     data_example,
                     new_vec)
```

As you can see, a list can hold any type of data, disregarding of its dimension or class.

- Can you access the element *vector_logic* in the list above, calling it by the slot name?

Functions

There are several types of functions:

- Basic or primitive functions: these are the default functions in R under the base package. For example, they can include basic arithmetic operations, but also more complex operations such as extracting median values, means or summary of a variable.
- Package functions: these are functions created within a package. For example, the glm function in the *stats* package.
- User-defined functions: These are functions that any user can create for a custom routine.

The basic components of a function are:

a) name: is the name given to the function (remember that naming meaningfully is important!)

b) arguments: are input valu (or values) that control the oprtaiioon of the function.

c) body: these are the operations operations or modifications to the arguments.

d) output: these are the results after modifying the arguments. If this output corresponds to a data series, we can extract it using the return command.

e) internal function environment: these are the specific rules and objects that exist within a function. Those rules and objects will not work outside of the function.

User-defined functions

Creating your own functions is one of the most powerful tools in R. This can save you time and increase your productivity when coding.

First, explore the following function, look at its components and play with different argument values. Also, try passing the output to a new object. The *power_function* is absolutely unnecessary as a tool, but a good example for now!

```
power_function <- function(base,power){ # function definition and arguments

  #body
  x<-base^power

  #output
  return(x)
}
```

Now you have a better idea of how to write an R function. Follow the the next steps:

a) Create a function to calculate Body Mass Index (BMI)

b) Calculate the BMI for these individuals:

- Patient 1: 1.83m and 96kg

- Patient 2: 1.65m and 78kg
- Patient 3: 1.98m and 99kg

Do you get these resuts?

```
## [1] 28.66613 28.65014 25.25253
```

Packages

On top of all the great basic functions that come buit-in as default in R, you can access thousands of packages that other R users have created for specific purposes. Anyone can create a package as long as it meets some quality criteria. A good package should be transparent and come with enough documentation for all the functions that it contain.

Approved R packages can be found in the Comprehensive R Archive Network (CRAN) https://cran.r-project.org , but you can install it directly from Rstudio using the following comands:

- Install a package: install.packages("*package-name*"). This step only needs to be done once.
- Load a package in your session: library("*package-name*"). This line of code susually goes at the top of your script. So everytime you run it, it will load the packages that are relevant to your session.

Try installing a couple of useful packages that we will use during the course.

```
install.packages("deSolve")    # For solving differential equations
install.packages("reshape2")   # For handling datasets
install.packages("tidyverse") # collection of packages for data managing and plotting
install.packages("here")       # Allows you to get the root path to your folder
```

Google the name of these packages and you will find the documentation for each one. You can explore the sets of functions they provide and how they work.