Introduction to R

Zoltan Kekecs, Marton Kovacs

November 02, 2020

Contents

1 Introduction

1.1 Abstrakt

This is an intorduction to the use of the R software. This exercise demonstrates the basic functionalities of R, and the use of the consol and the script fields. It will also inform about different different types of data in R, such as vectors, matrices, and data frames, and shows some basic functions used for data management.

2 Console

During R programming we use "sentences", similar to if we were talking with humans. Commands (sentences) directed to R can be entered directly to the console (usually this is in the bottom left panel of the RStudio interface, labelled "console").

Let's try this now, try typing a mathematical operation into the console, and press ENTER (e.g. 12/3, the = sign is not needed).

R will run this task, and will display the results in the console under the command. Not that all lines containing a command have ">" at the beginning of the line, while there is a number in brackets in the beginning of each result line "[1]".

3 Intorduction to R programming

R is a programming language. We need to use the words and phrases of this language to interact with the program.

Some of the language is similar to that used in mathematics. For example writing these to the consol and pressing enter will return valid results:

```
21 + 16
```

```
## [1] 37
```

```
3 * 5
```

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

```
10 * 2 / 5 + 2
```

```
## [1] 6
```

4 Script

If we wanted, we could type every one of our commands directly in the console, and read the results subsequently. But this would make it very hard to see it clearly what we did, partly because the code would be riddled with the results (which sometimes can be very long outputs), and partly because the console has a capacity (number of rows) beyond which old commands and results are no longer kept, and are deleted.

Thus, instead of using the console to type our code, we usually use a text editor to plan, and write code, and to keep our code organized. This text containing our code is called the script (because it is acin to the script of a screenplay which will later be "enacted" by the console), and we can use the built-in script editor in R and RStudio to write it. So we write our script first, and when we are satisfied with it, we run the code in the script later in the console.

The script editor is usually in the top left panel in R studio. When you first open R and RStudio you sometimes only see the console takine up all the left side, and you will have to open a new script window in *File > New file > R script*.

Now open a script window and type some commands in it. E.g..:

2 * 4

10 / 5

When we hit ENTER in the script editor, you don't get the results/output. ENTER only means "line break" here. This means that we can type, plan, and re-edit our code without having to worry about running it before we are ready.

Now that we are happy with our code, we can run it in the console. There are many ways to doing this. Maybe the easiest is to highlight the code segement we want to run using the mouse, and hit *Ctrl + ENTER* (in Windows) to run that specific code segment. This way all of the code highlighted will be copied into the console and run by R.

An even faster apporach is to simply click anywhere in the line of code we want to run and hit *Ctrl + ENTER*, which will run the code in that line until the command is finished (this might include several lines of code.) By doing this, R puts the cursor to the beginning of the row right after the command we just run, so this can be used to run multiple commands successively by pressing *Ctrl + ENTER* multiple times.

Now try the following: type 3 mathematical operations in the script editor in separate lines. Now click anywhere within the first line, and press *Ctrl + ENTER* multiple times. You should see that the commands are run in the console one after the other as they follow each other in the script.

5 Finished and unfinished commands

Now try to run the following script:

```
9 / 3 + 6 *
```

Note that this is a command-fragment. This is an unfinished mathematical operation, since we did not write anything after the "*". Now if you look in the console you should see that instead of the familiar">" sign in the end of the console lines, you see a "+" sign. This means that R is waiting for the commmand to be finished. It needs some more information to be able to run the command.

It is common, especially in the early learning stage of using R to accidentally run command fragments without realizing it. This will manifest in a way that you will not get the desired results/output, rather, you will just see the command line with this "+" sign growing and growing as you try to run new code, without ever returning a result. This is likely due to an unclosed parenthesis somewhere in your code. Usually the best way to stop this cycle is to just put a ")" in the console and running it with ENTER, which will end the command, and likely produce an error, but at least you can now resume running code again and get results.

If you are not sure what was the result of your script that was run and you want to revert it, you can start over. There are multiple ways for doing this, one way is to clear the work environment (but note that this will not affect the packages you loaded), or to close R without sawing the workspace image and start it again.

6 Functions

In R, most commands involve the use of functions. You can think of these as the "verbs" in the R programming language.

For example the function to get the logarithm of a number with a base 10 is "log()". You need to enter the number you want the to get the logarithm of into the parenthesis.

```r
log(2)
```

```
## [1] 0.6931472
```

We will meet a lot of functions druing this course.

We can even create our own functions (you can find great videos about this on the web), but most functions required for this course are built in in either base R or some of it's official packages.

7 Objects

If the functions are the verbs, objects are the nouns in R language. Objects are usually data that we designate by a name so that we can easily refer to it in our commands later.

For example we can assign the text "Tesla model S" to the object "dreamcar".

```r
dreamcar <- "Tesla model S"
```

From now on, everytime we write dreamcar, R knows that we refer to "Tesla model S" during this R session. This is erased once you close the R session or once you clear the workspace, unless you save the workspace. (I generally do not recommend saving the worksapce at quitting R to avoid earlier code creating issue in later code, unless you really know what you are doing.)

By running the object in the console (we can also call this "pringting the contents of the object"), R will list all of it's contents.

```r
dreamcar
```

```
## [1] "Tesla model S"
```

You can assign things to objects using either the "<-" or "=". They are interchangable. Some programmers like to use these in different situations, for example assigning things with "<-" in case of a new object, and using "=" when an already existing object is changed, but this is up to you to decide.

So these mean and do the same thing:

```r
dreamcar <- "Tesla model S"
dreamcar = "Tesla model S"
```

We can use object as a shorthand for their contents, so we can run the same operations and commands on them just liek we would with their raw contents. E.g.:

```r
number1 <- 5
number2 <- 2

sum_total = number1 + number2

sum_total
```

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

_____*Practice*_____

1. Assign 4 to a new object called *number*.
2. Subtract this object from the sum of the objects number1 and number2 seen in the example above, and assign the result of this equation to a new object called *result*.
3. Print the *result* object to check whether you did things right.

---

8 Workspace, environment

In RStudio we can find the contents of our work environment listed in the top right panel. This contains our currently used workspace (and for the current course we will only work with a single worksapce). In the workspace you will find all the currently active objects and data listed.

By using the brush icon you can clear the contents of the wokspace. This will erase all of your previously assigned objects and data that you loaded into memory. This is a great tool if you run into some errors that you cannot figure out, and want to start running your code from the beginning.

9 Packages

One of the advantages of R is that it is developing so rapidly. This is because the R experts are constantly developing new functions for R that keep in pace with the developments on the field. Most commonly these functions will be collected in so called "packages". By installing and loading a package you gain access to the functions that are inlcuded in that package.

For our purposes the most important package is "tidyverse". This is a package collection, comprised of many packages that are written with the same governing principle. They are all dedicated to make data management easier while keeping the associated code clean and "human interpretable".

You can find the "packages" tab in the bottom right panel in RStudio. This tab lists all installed packages, and marks those that are currently loaded in memory with a checkmark.

9.1 Installing package

The next command let's you read data from a google spreadsheet, and assigns it to an object called mydata.

```
mydata = gsheet2tbl("https://docs.google.com/spreadsheets/d/1RSGUhjNpDH4HQHIyqTHH-yHXx3X4YfFH4tzN51Q_hR
```

When you run this command, you will get an error message. The reason for this is that the gsheet2tbl() function is not in the base R package which is loaded every time you start an R session.

You need to load the pakcage that contains this function to be able to access it. The gsheet2tbl() function is in the "gsheet" package. If you don't already have this package installed, you need to install it with the install.packages("gsheet") command.

```
install.packages("gsheet")
```

9.1.1 Load package

It is important that installing the package is only the first step. This downloads the package and installs it on the system, but in itself it will not make the functions accesible. You need to also load the pacakge into memory each time you want to use that pakcage. You can use the library() function to do this. E.g.:

```
library(gsheet)
```

Now you that the package is installed and loaded, the function can be used:

```
mydata = gsheet2tbl("https://docs.google.com/spreadsheets/d/1RSGUhjNpDH4HQHIyqTHH-yHXx3X4YfFH4tzN51Q_hR
```

```
## No encoding supplied: defaulting to UTF-8.
```

You can use the View() command to check that the data file is really assigned to the mydata object.

```r
View(data)
```

10 Types of objects

10.1 Data types

10.1.1 Atomic vectors:

Atmoic vectors are vectors with a single element.

- **character:** a string of characters enclosed with ""
- **numeric:** 2 or 13.5. Numeric vectors can be either integers or high precision numeric vectors called "double" (rational numbers). You can designate a vector to be integer by putting a capital L after the number.
- **complex:** 1+5i, complex numbers
- **logical:** can only take values TRUE or FALSE (note that capitalization is important!). NA (missing) is also a logical.

The **class()** and the **typeof()** functions can be used to get information about the data class and type.

```r
class("I love R") # character
```

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

```r
class(2.34) # numeric (type: duble)
```

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

```r
typeof(2.34)
```

```
## [1] "double"
```

```r
class(2L) # integer
```

```
## [1] "integer"
```

```r
class(1+4i) # complex
```

```
## [1] "complex"
```

```r
class(TRUE) # logical
```

```
## [1] "logical"
```

We can also directly ask about any specific class by using the **is.. . .() functions** for example:

```r
is.numeric("I love R")
```

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

```r
is.character("I love R")
```

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

```r
is.double(2.34)
```

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

```r
is.logical(FALSE)
```

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

10.1.2 Coercion of vectors

Vectors can be coerced to become other class or type.

```
as.character("I love R")
```

```
## [1] "I love R"
```

```
as.double(2L)
```

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

```
as.logical(0)
```

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

```
as.numeric("I love R")
```

```
## Warning: NAs introduced by coercion
```

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

10.1.3 Vectors with multiple elements

We can use the c() "combine" function to combine multiple elements into one vector. If all of the vectors that it was combined of had the same data type, the new vector will take the type of the old vector.

```
number <- c(3, 4)

number
```

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

```
class(number)
```

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

```
typeof(number)
```

```
## [1] "double"
```

```
is.numeric(number)
```

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

```
is.integer(number)
```

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

```
is.vector(number)
```

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

When combining vectors with different types into one vector, the type is determined by a hierarchy of types:

```
numbers <- c(3, 4)
letter <- c("letter1", "abcd")

new_vector <- c(numbers, letter)

new_vector
```

```
## [1] "3"       "4"       "letter1" "abcd"
```

```
class(new_vector)
```

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

```
typeof(new_vector)
```

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

```
is.numeric(new_vector)
```

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

```
is.integer(new_vector)
```

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

```
is.vector(new_vector)
```

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

```
is.character(new_vector)
```

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

10.1.4 Complex data structures

There are also more complex data structures capable of holding multiple variables, sometimes with different data types.

- **matrix:** A combination of vectors, arranged in a matrix. In a matrix all data must have the same type.

```
more_numbers <- c(1, 5, 2, 7)
my_matrix <- matrix(more_numbers, nrow = 2)

my_matrix
```

```
##      [,1] [,2]
## [1,]    1    2
## [2,]    5    7
```

- **data.frame:** Similar to a matrix with the main difference that columns can be of different data class.

```
my_dataframe <- data.frame(my_matrix)

my_dataframe
```

```
##   X1 X2
## 1  1  2
## 2  5  7
```

- **list:** A vector, elements of which can be other data structures, types and class. (There can be a list containing two dataframes, one atomic vector, and trhee matrices, etc. all within the same list)

```
x_character <- c("some", "characters")

my_list <- list(my_dataframe, numbers, x_character)

my_list
```

```
## [[1]]
##   X1 X2
```

```
## 1  1  2
## 2  5  7
##
## [[2]]
## [1] 3 4
##
## [[3]]
## [1] "some"       "characters"
```

A következő funckiókkal különböző információkat tudhatunk meg az objektumokról: class() - what kind of object is it (high-level)? typeof() - what is the object's data type (low-level)? length() - how long is it? What about two dimensional objects? attributes() - does it have any metadata?

11 A few useful abbreviations in R

If we put a ":" between two numbers, R will aotumatically interpret that we mean all numbers between the two designated numbers. So this qill result in a sequence of numbers growing by one at each step. Another longer way to write the same thing displayed below using the seq() function.

```
1:5
```

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

```
seq(1, 5, by = 1)
```

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

the "letters" object is a built in vector within base R containing the letters of the English alphabet.

```
letters
```

```
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s"
## [20] "t" "u" "v" "w" "x" "y" "z"
```

If you only want to use certain elements of a vector, you can use subsetting. We will talk about subsetting more in the next exercises, but basically what you can do is put the number of the position of the elements that you want to refer to in a bracket after the object name like this:

```
letters[3] # selects the third element of the object "letters"
```

```
## [1] "c"
```

```
letters[1:21] # selects the first to the 22nd element of the object  "letters"
```

```
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s"
## [20] "t" "u"
```

_____*Practice*_____

4. create an object called "my_first_vector" containing the numbers from 1:120 (Using the abbreviation shown in the end of the exercise makes this much easier.)
5. What is the class of my_first_vector?
6. Assign the first 20 elements of my_first_vector to a new vector object called my_second_vector.
7. Install the "tidyverse" package and load it into memory

_____