Loops and Functions in R

Refresher on Recommended Previous Knowledge

Instructor: Alfredo Ascanio

E-mail: ascaniaa@miamioh.edu

Assistant instructors: Tereza Jezkova, Meelyn Pandit

1 RECOMMENDED PREVIOUS KNOWLEDGE

This refresher will serve you to remember some R foundations that you may or may not have explored before. Upon these foundations we can start learning how to make our own loops and functions.

- Data types in R: numeric, character, factors, logical values, etc.
- Base R objects:
  - Vectors, Matrices, Lists, and Data Frames
  - Subsetting
- Using functions[1]
- Conditionals and Logical Operators[1]

[1] **WE WILL REVIEW SOME OF THESE CONTENTS IN THE COURSE**

2 DATA TYPES IN R

We use R with the intention to handle data. This handling can mean different things:

- Cleaning Data/Data management
- Change Data Formats
- Data Visualization
- Statistical Analysis
- Mathematical Analysis; among many others.

If you have used a spreadsheet software before, such as Excel or Google Sheets, you may have noticed that you can select the *type of data* that is stored in a given cell. These types of data or cells can be: numbers, currency, dates, text/character strings, etc. **R has the same functionality**. Whenever you load or read any data into R, the language will try to guess what type of data it is dealing with. There are several reasons why all software and programming languages do this, for me, the most important is that: each type of data occupies a different amount of space in the computer memory. I won't delve too deep into that, mainly because I'm not that knowledgeable about the topic. As an example: storing integer numbers require less memory than real numbers

(which can contain decimals), so saving a “42” as character, a 42 as an integer, or a 42.00 as a real number will have different impacts on memory, and also on how we can deal and manage that information, even if for us “the three values mean the same thing”.

Understanding this is important, because all software and functions (in R, python and another languages) have specific requirements on the type of data they need to execute their functions properly. Always check the type of data you are dealing with, and also check how any given software/language is interpreting them. As the user, you’ll always have the chance to change between data types, but there is a proper way and time to do it before or during an analysis or process.

2.1 SOME OF THE MAIN DATA TYPES THAT I USE IN R ARE:

- **integer:** Integer numbers, sometimes represented in R as a number followed by an uppercase L (2L, 4L, 7L, …). They allow for negative numbers also.
    - **CODE:** Print the number 37L in console. Also ask for the class of that value: class(37L)
- **numeric:** real numbers in general. Allow for decimal places.
    - **CODE:** Print the number 37.00 in console. Also ask for the class of that value: class(37.00)
- **character:** character values can be distinguished from others because they always appear surrounded by single ('x') or double quotes ("x"). That’s why having 42 is different from having “42”, as the computer will interpret those values as different data types.
    - **CODE:** Print the *string* “37”. Also ask for the class of that value: class(“37”)
- **date:** very useful when dealing with time series. They can be converted to an integer value, which will be equal to the number of days that have passed since 01/01/1970. A Date value will show the date in the format “YYYY-MM-DD”.
    - **CODE:** Try executing the function Sys.Date() in R. Also execute the function class, class(Sys.Date())
- **factor:** Factors are a unique element within the R programming language. It’s not part of other languages, and it’s a very useful data type for building and using statistical analysis. They can be viewed as a mixture of an integer value (factor levels) and a character value (factor labels).
    - **CODE:** Try executing the function
        - factor(c(0, 1, 0, 1, 1), levels = c(0, 1), labels = c("Male", "Female"))
    - Try changing the labels order
        - factor(c(0, 1, 0, 1, 1), levels = c(0, 1), labels = c("Female", "Male"))
- **logical:** A really important data type, containing two possible values: TRUE or FALSE (T or F can be used instead). These values can be interpreted numerically as 1 or 0, so we can sum logical values.

There are many other data types, and you could even create your own class of data. Several packages do that and create functions to deal with those specific data types. You can always check the data type of a value by using the function class() on it.

Many errors in R are the result of having the wrong data type as input of functions. Whenever you look at the help for a certain function, in the arguments section it will detail the type of data they require.

3 BASE R OBJECTS: VECTORS

Once we know that R, and any other software, makes some sort of interpretation and assignment of the type of data, the second step is to learn how to **save or store that data for R to handle it**. I think this is one of the things I struggled the most with, when I began working on R. This is because we usually store data in an Excel/Google sheet, make a sum or some simple operation by selecting the cells we need, and that's it. **But R does not have one standard object in which it stores data; it has many more.** And even making simple operations relies on the fact that we must have the right objects and data types to work with.

We will review 3 of the 4 basic objects, starting with **vectors**. I call them basic not because they are simple, but because they are the foundation of everything we do in R.

To save any type of data in any type of object, we use the term **assignment, and the assignment operator "<-"**. **We assign a value (or set of values) to a name:**

z <- 5

3.1 VECTORS

A **Vector** is the most basic object in R. Whenever we print or store a single value, we are dealing with a vector of length 1 (equivalent to a single cell in a spreadsheet). The length of the vector refers to the amount of values that are being stored in it. We can check for the length of a vector by using the function *length()*.

length(z)

An important characteristic of vectors is that **all elements contained in a vector will be considered of the same type**. So, we can have *character*, *integer, numeric, logical, factor, date, etc.* vectors, never in combination. Some ways to make a vector are:

x <- c(1, 3, 7) *# c() function concatenate values into a vector*

x <- 5:20 # ":" generates an integer sequence between two numbers

seq(from = 1, to = 10, by = 0.1) *# seq() generates a numeric sequence between two numbers*

rep(x = 3, times = 5) *# rep() generates a repeated sequence of the inputted vector x*

3.2 POSITIONS AND SUBSETTING IN A VECTOR

All objects in R are defined by a set of fixed **positions**; these positions are the places that values can occupy within an object. More complex objects can have multiple position coordinates for each value. **Vectors only have one set of position coordinates, ranging from 1 to the length of the vector** (in python and other languages, the first position is counted from 0 instead of 1)**.** This is important because **we can subset values from a vector using its positions.**

If we create a vector *x* of length 3:

x <- c(1, 3, 7)

This vector can be seen in the following way

x	Values	1	3	7
	Positions	1	2	3

We can extract information from it using the positions we wish to subset within **square brackets** in the following way:

x[2]

That statement can be read as: *I want the value of the SECOND position in my vector x*; which will result in the number 3. Try it out in your R.

x	Values	1	3	7
	Positions	1	2	3

We can extract multiple values by putting a vector of the desired values' positions within the square brackets:

x[c(1, 3)]

x	Values	1	3	7
	Positions	1	2	3

That statement can be read as: *I want the values of the FIRST and THIRD position in my vector x*; which will result in the numbers 1 and 7. Try it out in your R.

We can delete information from a vector by putting a negative value or vector in the square brackets

x[-c(1, 2)]

That statement can be read as: *I want the values of my vector x; except for the FIRST and SECOND positions;* which will result in the number 7. Try it out in your R.

And last, we can use this principle to reorganize data, given that the values we request are going to be returned in the order of the positions we input in the square brackets:

x[c(3, 1, 2)]

3.3 LOGICAL VECTORS AND SUBSETTING OTHER VECTORS

We will review this knowledge on logical vectors during the course, but it would be better if you have previously deal with them in some extent. **I view logical vectors as the result of making a question to the computer, that can be answered with one or more TRUE or FALSE values.** For example:

Logical Operators

> x <- 5; y <- 8
> x > y *#is x greater than y? Results to FALSE*
> x < y *#is x less than y? Results to TRUE*
> x >= y *#is x equal to or greater than y? Results to FALSE*
> x <= y *#is x equal to or less than y? Results to TRUE*
> x == y *#is x equal to y? Results to FALSE*
> x != y *#is x different to y? Results to TRUE*
> X %in% y *#is the value of x contained in the values of y? Results FALSE*

We can use this type of questions on longer vectors.

> x <- c(1, 3, 7)

> x > 5 *#is x greater than 5? Results to a logical vector c(FALSE, FALSE, TRUE)*

> x == 3 *#is x equal to 3? Results to a logical vector c(FALSE, TRUE, FALSE)*

> x %in% c(1, 7) *#are the values of x contained in the values of c(1, 7)? Results to a logical vector c(TRUE, FALSE, TRUE)*

We can use a logical vector to extract information from another vector. The condition is that both vectors must be of the same length. We use the square brackets. The logic behind this process is, **R will subset the positions of our vector that are TRUE** (or delete the positions of our vector that are FALSE).

> x <- c(1, 3, 7)

> x[x > 5] *# extract the values of x that are greater than 5*

x	**Values**	1	3	7
	Positions	1	2	3
	x>5 ?	FALSE	FALSE	TRUE

> x[x == 3] *# extract the values of x that are equal to 3?*

> x[x %in% c(1, 7)] *# extract the values of x that are contained in c(1, 7)*

x	**Values**	1	3	7
	Positions	1	2	3
	x in c(1, 7)?	TRUE	FALSE	TRUE

This may seem silly with only three values. But imagine that you are dealing with a database that contains thousands, hundreds of thousands, or even millions of records. Being able to ask a short question and extract information based on it simplifies subsetting and other operations, once you gain practice with it. What we want is to avoid reading a spreadsheet for hours looking for values, errors, missing cells, which will make you prone to mistakes once you get tired.

4 BASE R OBJECTS: MATRICES

Similar to vectors, **matrices can only contain elements of the same data type**. So, we can have a character matrix, numeric matrix, integer matrix, etc. One big difference from vectors is that **matrices have two position coordinates** for each element they contain; **this will also be the case in data frames.**

Matrices and data frames are bidimensional arrays of elements (two position coordinates) while vectors are unidimensional arrays of elements (one position coordinates). Given that matrices can only store one type of data, you may think they are not as useful as data frames. However, several functions for general statistics, ecological, and genetic analyses rely on different sets of matrices as input to run. We can create a matrix in R as follows:

A <- matrix(1:9, ncol = 3)

If you print the object *A*, it will look like this:

A	1	4	7
	2	5	8
	3	6	9

Remember that we have now two position coordinates, which we will call **rows** (horizontal) and **columns** (vertical). All columns and rows are vectors, always of the same type. Row vectors must have the same length. Column vectors must have the same length. The length of rows and columns vectors may differ. This means that we can have more rows than columns or vice versa.

	Row	Column	1	2	3
A	1		1	4	7
	2		2	5	8
	3		3	6	9

To extract any value by position, we can use the square brackets using the form

Object[rows, columns]:

A[3, 3]

	Row	Column	1	2	3
A	1		1	4	7
	2		2	5	8
	3		3	6	9

To extract multiple values by position, we can use the square brackets as follows:

A[c(2, 3), c(1, 2)]

	Row	Column	1	2	3
A	1		1	4	7
	2		2	5	8
	3		3	6	9

We can select rows leaving all columns, or vice versa, by leaving the place before or after the comma empty:

A[, 1:2]

	Row	Column	1	2	3
A	1		1	4	7
	2		2	5	8
	3		3	6	9

And last, we can use logical vectors to subset rows and columns:

A[, colSums(A) > 10]

This statement will return a subset matrix with all the rows but only the columns which sum was greater than 10:

Question			colSums(A) > 10		
Logical Result			FALSE	TRUE	TRUE
	Row	Column	1	2	3
A	1		1	4	7
	2		2	5	8
	3		3	6	9

We can erase rows and columns in the same way we delete vectors' positions (adding a - sign before the positions' number or vector)

All of these ways to subset matrices also apply for Data Frames

5 BASE R OBJECTS: DATA FRAMES

Data frames are similar to matrices in that they are bidimensional arrays of information. They differ from matrices in that the **column vectors can be of different data types**. In this sense, we can have a character column, a factor column, a numeric/integer/complex column, a date column, etc. All within one Data Frame. **A data frame is a special case of a list (the 4th basic R object)**, but as we won't use lists in the course, and they can certainly be more complex to understand I won't talk too much about them. In summary, **a list is an object that can contain other objects**. So, lists are not constrained by the length of the vectors, dimensions of other objects, or the data types they contain.

A data frame, however, is constrained in the following way: All column vectors must be of the same length. This means that all column vectors must have the same number of rows, i.e. observations. We will use the *iris* data set, always preloaded in R.

DF <- iris

We can subset values from a data frame in a similar way as with matrices: using the positions we want for rows and columns, or a logical result for rows and/or columns.

Column1 <- DF[, 1]

Another difference between Matrices and Data Frames is that the latter have row and column names. We can use these column names to subset specific columns we want, given that we now their names.

We can use the dollar sign to extract a column vector by name

Column1 <- DF$Sepal.Length

We can also use a character vector containing the names of the columns we want, and use it in the columns position within square brackets

Column1 <- DF[, "Sepal.Length"]

Columns2_5 <- DF[, c("Sepal.Width", "Species")]

An example of subseting rows with logical vectors could be as follows: let's retain all columns and subset all the rows of the iris data frame (DF) that have Petal.Length < 4

subDF <- DF[DF$Petal.Length < 4,]

As a final consideration, remember that we can combine logical expressions using **AND (&)** and **OR (|)** logical operators. For example, in the *iris* dataset we could ask for all the columns but only the rows that contain Petal.Length < 4 **OR** Sepal.Length > 5; we could also ask for the rows that contain Petal.Length < 4 **AND** Sepal.Length > 5. Try running those codes and analyze how they differ:

DF[DF$Pepal.Length < 4 | DF$Sepal.Length > 5,]

DF[DF$Pepal.Length < 4 & DF$Sepal.Length > 5,]

Combining logical expressions like this also works for subsetting the other R objects.

6 FINAL CONSIDERATIONS

I hope this short guide is useful to you for our Loops & Functions in R course, and for the future. I feel like being able to visualize what R objects are and how they can be rearranged, moved, and subsetted, makes things easier. You may have already encountered and learned the knowledge I present here; but if you don't, welcome to the foundations of R!

Again, this is not saying that these things are simple; it takes time to build enough experience to remember the details. That's why, if you choose that you want to become proficient in R or any other programming language, I encourage you to use it in a daily basis. Even for tasks that you may find silly, like adding two numbers together, reviewing a data frame in R instead of excel, correcting a mistake in a dataset. Through these, not always easy tasks, you'll gain experience and confidence in your abilities to use the language. And eventually you'll be ready to step up and start correcting/modifying old code from others, or even building your own code, functions, and algorithms.

In this Loops & Functions in R course, our objective is for you to learn the structures and basics of the creation of loops and functions. We don't intend for you to master in two days stuff that others have learned in months or years. But it's good to learn what is possible and how to do it, and understand that **you are capable!**

You'll be able to use these resources as much as you want, come back to them, reread them, and rewrite the codes to practice with new values and datasets. **Coding in the real world** is not as in the movies; and this is especially true as biologists with no background computing skills. I also came from that path. **Forums and courses are the way** in which we learn and relearn how to build stuff. If you have a question (let's say that 95% of the time) I bet that someone else already had it and posted it on an internet forum. **Coding is half knowing how to write code and half knowing how to look for what you don't know (but you know you need)**. This is an ability that also takes time to build.

Nobody knows R, mathematics, statistics, or anything from birth. It takes time and effort. So just begin, practice, search, read, learn. Whenever you get tired, leave it be. Whenever you are excited to work/learn, get back again, open R and start exploring how to solve a problem.

Good luck, and see you in the course,

Alfredo