Lab 2: Adavanced R

Instructions

Dr Mercedes Torres Torres

Contents

1 Introduction

In last week's lab session, we introduced R and some of its basic functionality (arithmetic operations, vectors, matrices and simple plotting functions). We also introduced R Notebooks and how to use them.

However, R is a much more sophisticated environment, capable of:

- Effective data handling and storage
- Efficient and extensive operations on arrays and matrices
- Robust data analysis using a large, coherent, integrated collection of intermediate tools
- Extensive graphical analysis and display either directly at the computer or on hardcopy
- Executing a well developed, simple and effective programming language which includes conditionals, loops, user defined recursive functions and input and output facilities.

This week, we will cover more advanced topics, such as reading and writing from different types of files, creating your own functions, and using control structures to create more sophisticated functions.

After this lab session, you will be able to use R to:

- Use control structures
- Learn how to write functions in R
- Learn how to write to file
- Learn how to read from file
- Learn how to create and use *lists* and *data frames*
- Learn how to access columns and rows of read data by name
- Learn how to use the *apply()* functions

2 Functions

A function allows you to repeat a series of commands that you or someone else has created without having to type all the instructions again. This is particularly useful for conducting experiments where you want to repeat tests with only minor changes. Almost all R commands are actually function files.

For a list of all of the functions already implemented in base R, type the follwing:

```r
library(help = "base")
#OR
builtins()
```

Instead of covering all functions at once, in this module we will introduce relevant functions as we cover different topics in the area of Data Analysis and Mining.

2.1 Your Functions

You may need to create you own functions to carry out calculations on your data. In order to write a new function, create a new source file by choosing *File -> New -> R* script from the menu.

The syntax of a function is:

```r
function_name = function(parameter_1, ...., parameter_n) {
  #Instructions
  ...
  #if you function returnso me data
  #R only allows you to return one variable
  return(data)
}
```

Let's see a very simple example. Let's create a function that, given a vector v, calculates and returns how many of the values in the vector are divisible by 7 and returns the calculation. Save this as *div7.r*.

```
div7 <-function(v){
  d = sum(v%%7==0)

  return(d)
  # a more condensed solution would be: return(sum(v%%7==0))
}

# Now let's test our function
v1 = c(1, 2, 5, 6, 7, 3)
v2 = c(7, 14, 49, -3)

div7(v1)
```

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

```
div7(v2)
```

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

```
#Note: once functions are read and sourced by R, you can use them at any point,
# even in separate chunks and documents.
```

You can add this file to the project through the interface or by using the *source()* command. This will add *div7()* to the library of functions you can call. You will now be able to call *div7(v)* in the command line or in code chunks, and see their effect (note that this file must be in the correct directory).

Of course, this is a very simple example. As the problems you try to solve become more and more complex, so will your functions.

Reading: For more information about writing your own functions, read Chapter 10 of An Introduction to R by Venables et al.

3 Control structures

Control structures allow you to design and control the flow of execution of a script or a chunk of code. Common ones include:

3.1 If/else statement

Used to make decisions depending on different conditions.

```
# if-else statement
x=10
if(x>1){
  print("x is greater than 1")
  }else{
    print("x is less than 1")
  }
```

```
## [1] "x is greater than 1"
```

```
#if and else-if statement
x=10
if(x>1 & x<7){
  print("x is between 1 and 7")
 }else if(x>8 & x< 15){
   print("x is bewtween 8 and 15")
 }else{
   print("x is smaller than 1 or larger than 15")
 }
```

```
## [1] "x is bewtween 8 and 15"
```

3.2 For/while loop

Used for iterating through items.

```
#for - using indeces
x = c(1,2,3,4,5)
for(i in 1:5){
  print(x[i])
  }
```

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

```
#for - using numerical vector elements
y = seq(1,10,3)
for (i in y){
  print(i)
}
```

```
## [1] 1
## [1] 4
## [1] 7
## [1] 10
```

```
#for - using char/string vectors
z = c("cats", "dogs","horses")
for (i in z){
  print(i)
}
```

```
## [1] "cats"
## [1] "dogs"
## [1] "horses"
```

```
#while
x = 2.987
while(x <= 4.987) {
  x = x + 0.987
  print(c(x,x-2,x-1))
  }
```

```
## [1] 3.974 1.974 2.974
## [1] 4.961 2.961 3.961
## [1] 5.948 3.948 4.948
```

3.3 Repeat loop

An infinite loop used in association with a break statement. It is usually accompanied of a *break* statement, which is used in a loop to stop the iterations and move the control outside of the loop. *Break* statements can also be used in *for* or *while* loops.

```r
#repeat loop:
a = 1
repeat {
  print(a)
  a = a+1
  if(a > 4)
    break }
```

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

```r
#break statement
x = 1:10
for (i in x){
  if (i == 2){
    break
    }
  print(i)
}
```

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

The *next* statement can also be added to the loops to skip the current iteration of a loop without terminating it.

```r
#next statement
x = 1: 4
for (i in x) {
  if (i == 2){
    next
  }
  print(i)}
```

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

You can find more information and examples typing *??Control* in the command line in RStudio.

Reading: For more information about Control Structures, read *Chapter 9* of An Introduction to R by Venables et al.

4 Adavanced Data Structures: Lists and Data Frames

Until now, we have seen very simple data types (numbers, vectors and matrices). However, very often you might want to store linked information of different types into one single variable. For example, you might want to store the name, surname, height, weight, age and nationality of a group of 20 people. For this, a matrix would be insufficient, as all elements of a matrix must be of the same type. To help with this, R offers lists and data frames.

4.1 Lists

Lists contain an ordered collection of objects. You can create them with the *list()* function.

```
#a list with four components
w = list(name="John", surname= "Silver", alias="Long John", age=30, alive="yes")
z = list(name="James", surname= "McGraw", alias= "James Flint", age= 45, alive ="unknown" )

#You can concatenate lists into lists:
v = c(w,z)

w
```

```
## $name
## [1] "John"
##
## $surname
## [1] "Silver"
##
## $alias
## [1] "Long John"
##
## $age
## [1] 30
##
## $alive
## [1] "yes"
```

```
z
```

```
## $name
## [1] "James"
##
## $surname
## [1] "McGraw"
##
## $alias
## [1] "James Flint"
##
## $age
## [1] 45
##
## $alive
## [1] "unknown"
```

```
v
```

```
## $name
## [1] "John"
##
## $surname
## [1] "Silver"
##
## $alias
## [1] "Long John"
##
## $age
## [1] 30
##
## $alive
## [1] "yes"
##
## $name
## [1] "James"
##
## $surname
## [1] "McGraw"
##
## $alias
## [1] "James Flint"
##
## $age
## [1] 45
##
## $alive
## [1] "unknown"
```

You can identify elements of a list using the *[[ ]]* operation

```
w[[2]] # 2nd component of the list
```

```
## [1] "Silver"
```

```
w[["surname"]] # component named "surname" in list
```

```
## [1] "Silver"
```

4.2 Data Frames

Data frames are some of the most powerful data structures in R. In a data frame, different columns can have different types. You can create a data frame with the *data.frame()* function.

```
#Remember that c() is the _concatenate_ function:
name<- c("jane", "elizabeth", "lydia", "kitty", "fitzwilliam", "charles", "georgiana")
age <- c(21, 20, 18, 17, 27, 25, 15)
status = c(TRUE,TRUE,TRUE,FALSE, FALSE, TRUE, FALSE)
savings = c(500, 300, 200, 100, 10000, 20000, 20000)

#Create data frame
df = data.frame(name, age, status, savings)
names(df) = c("ID","Age","Alive", "Funds") # variable names

#Let's take a look
df
```

```
##            ID Age Alive Funds
## 1        jane  21  TRUE   500
## 2   elizabeth  20  TRUE   300
## 3       lydia  18  TRUE   200
## 4       kitty  17 FALSE   100
## 5 fitzwilliam  27 FALSE 10000
## 6     charles  25  TRUE 20000
## 7   georgiana  15 FALSE 20000
```

There are a variety of ways to access the elements (or subsets of elements) of a data frame.

```
df[,2] # 2nd column
```

```
## [1] 21 20 18 17 27 25 15
```

```
df[2:4,] # 2nd to 4th row
```

```
##          ID Age Alive Funds
## 2 elizabeth  20  TRUE   300
## 3     lydia  18  TRUE   200
## 4     kitty  17 FALSE   100
```

```
df[c("ID","Funds")] # columns ID and Funds from data frame
```

```
##            ID Funds
## 1        jane   500
## 2   elizabeth   300
## 3       lydia   200
## 4       kitty   100
## 5 fitzwilliam 10000
## 6     charles 20000
## 7   georgiana 20000
```

```
df$ID # variable ID in the data frame
```

```
## [1] jane        elizabeth   lydia       kitty       fitzwilliam charles
## [7] georgiana
## Levels: charles elizabeth fitzwilliam georgiana jane kitty lydia
```

```
#You can even check for conditions, too
df[df$Alive==TRUE,]
```

```
##          ID Age Alive Funds
## 1      jane  21  TRUE   500
## 2 elizabeth  20  TRUE   300
## 3     lydia  18  TRUE   200
## 6   charles  25  TRUE 20000
```

```
# Check carefully the use of the conditional operations & and the blank space:
df[df$Age>20 & df$Funds>5000,]
```

```
##            ID Age Alive Funds
## 5 fitzwilliam  27 FALSE 10000
## 6     charles  25  TRUE 20000
```

Not only you can access different splits of data in a data frame, you can also carry our operations with them:

```
mean_age = mean(df$Age) # mean age across all
min = min(df$Age) # min age across all

sum_funds = summary(df$Funds) #summary of funds
```

And you can even modify the data frame:

```
df$Age = df$Age + 1 #adds 1 to all ages
#sets to 0 the funds of those over 20 years who are alive
```

```
df[df$Age>20 & df$Alive==FALSE,]$Funds = 0
df
```

```
##            ID Age Alive Funds
## 1        jane  22  TRUE   500
## 2   elizabeth  21  TRUE   300
## 3       lydia  19  TRUE   200
## 4       kitty  18 FALSE   100
## 5 fitzwilliam  28 FALSE     0
## 6     charles  26  TRUE 20000
## 7   georgiana  16 FALSE 20000
```

```
# You can also access the same  values by writing the name of the field in quotes
# inside the [ ] operator:
df[df$Age>20 & df$Alive==FALSE,"Funds"] = Inf #Infinite funds!
df
```

```
##            ID Age Alive Funds
## 1        jane  22  TRUE   500
## 2   elizabeth  21  TRUE   300
## 3       lydia  19  TRUE   200
## 4       kitty  18 FALSE   100
## 5 fitzwilliam  28 FALSE   Inf
## 6     charles  26  TRUE 20000
## 7   georgiana  16 FALSE 20000
```

Adding new columns (fields) in data frames is also very easy. Just name them:

```
#Let's create the field logical field "Married":
df$Married = c(TRUE, FALSE, TRUE, TRUE, FALSE, TRUE, FALSE)

#Ta-da! df now includes the field Married
df
```

```
##            ID Age Alive Funds Married
## 1        jane  22  TRUE   500    TRUE
## 2   elizabeth  21  TRUE   300   FALSE
## 3       lydia  19  TRUE   200    TRUE
## 4       kitty  18 FALSE   100    TRUE
## 5 fitzwilliam  28 FALSE   Inf   FALSE
## 6     charles  26  TRUE 20000    TRUE
## 7   georgiana  16 FALSE 20000   FALSE
```

And adding rows (instances or samples) is also very simple

```
nu_row = data.frame(ID = "mary", Age = 16, Alive = FALSE, Funds = 100, Married = TRUE)
df = rbind(df,nu_row)
```

Similarly to vectors, matrices, arrays and data frames, you can carry out arithmetic, logical and statististical operations in the elements of a list, as long as they are the right type.

4.3 Lists VS Data frames

At this point, you might be unsure about the differences between data frames and lists. Lists, like data frames, allow elements to be of different types. However, unlike data frames, they can have different dimensions. Let's see an example:

```r
js=  list(name="John", surname= "Silver", alias="Long John", age=30, alive="yes")
jm= list(name="James", surname= "McGraw", alias= "James Flint", age= 45 )
eg = list(name="Eleanor", surname = "Gurthrie")
extra = list(1,2,3,4,5,6,7) #a random list
#These three lists all have different lengths:
length (js)
```

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

```r
length(jm)
```

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

```r
length(eg)
```

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

```r
length(extra)
```

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

```r
#However, I'm able to concatenate them into one list without a problem:
people = c(js,jm,eg)
#I am even able to concatenate unrelated information:
conc = c(js,jm,eg,extra)
```

Now, let's try the same with data frames (*rbind()* concatenates data frames by rows):

```r
john=data.frame(name="John", surname= "Silver", alias="Long John", age=30, alive="yes")
james=data.frame(name="James", surname= "McGraw", alias= "James Flint", age= 45,
                 dead ="unknown")
eleanor=data.frame(name="Eleanor", surname = "Gurthrie")

#If dimensions are different, you will get an error:
rbind(john, eleanor)
```

```
## Error in rbind(deparse.level, ...): numbers of columns of arguments do not match
```

```r
#OR if the name of the fields is different, you will get an error:
#alive is not present in "james" and dead is not present in "john"
#so this will raise an error
rbind(john,james)
```

```
## Error in match.names(clabs, names(xi)): names do not match previous names
```

Reading: For more information about *Data Frames*, read *Chapter 6* of *An Introduction to R* by Venables et al.

5 File management

During the course of your work, you will need to interact with a wide variety of external files. You will create, write, import, and save or export different types of files. R has built-in functions that can help you store data into different file formats and read it into different types of variables.

5.1 Loading, Reading and Writing TXT files

Let's work with an example. First, let's define the matrix A:

```
# Define the matrix A as:
A = rbind(c(1,2), c(3,4), c(3,1), c(8,7))
```

The matrix A is not a big matrix, but it might be once we start using real data, and we might want to save it to a file. Make sure you have chosen a working directory where you can save files first.

Now, the matrix A can be written to a file with the *write.table(data, file, sep, col.names, row.names)* function, where *data* is the data to be saved, *file* is the path and name of the file, and *sep* is an optional parameter with the separator:

```
write.table(A, file="data.txt")
```

Open this file to see what you have got. You may have noticed that each row and column has been assigned a generic name ("V1" to "V7" and "1" to "2"). The column and row names are controlled by the attributes *col.names* and *row.names*.

Execute the following commands:

```
write.table(A, file="data2.txt", col.names = FALSE)
write.table(A, file="data3.txt", col.names = FALSE, row.names=FALSE)
write.table(A, file="data4.txt", col.names = c("T1", "T2"), row.names=FALSE)

# What are the differences in the files that are produced?
```

We can read or load files with the *read.table(file, header)*, but be careful with the quote marks: They are essential!

```
Z1= read.table("data4.txt")
Z2= read.table("data4.txt", header = TRUE)
# What are the differences between these two commands?
```

There are also commands to read and write delimited text:

```
L= read.table("data.txt", header=FALSE, sep=":")
```

Colon-delimited files are used on some Unix systems, but it is more usual to use commas or tabs on Windows systems.

5.2 Loading, Reading and Writing CSV files

A comma-separated-values (CSV) file is a file that stores tabular data (numbers and text) separated by a delimiter, such as a comma. They are frequently used in data science because they allow for data of different types to be stored, and they can easily be read by many platforms, such as R, MATLAB, JAVA, C, and even EXCEL.

Each line (or row) of the file is a data record (or an instance) and each instance consists of one or more fields (columns), separated by the delimiter. Since each column might be of a different type, data frames are specially suited to store data from CSV files.

To read data from a CSV to a data frame, use the *read.csv(file, header)* function:

```
# Read CSV into data frame
data = read.csv(file="Path/to/file/data.csv", header=TRUE, sep=",")
```

Remember that the first folder in which R will look for files (or will store files on) is your working directory. Therefore, if the file you want to read is stored there, you can just write:

```
data = read.csv(file="data.csv", header=TRUE, sep=",")
```

Where *header* is a boolean value indicating whether the file contains the names of the variables as its first line, *sep* is the parameter that establishes the delimiter (or separator) character. It can be changed to " " (a space), tabs, newlines, etc. You can find more information by using *help("read.csv")*.

To write your data into a CSV file, you can use the *write.csv(data, file)* function. For example:

```
# Modifying files Write data variable to CSV in R
write.csv(data, file = "myData.csv")
```

Remember: since you have not specified the path of *myData.csv*, this will be stored in your working directory.

Reading: For more information about Reading and Writing, read *Chapter 7* of *An Introduction to R* by Venables et al.

6 Apply Functions

Loops can be computationally expensive, especially when nested together. For this reason, R offers a family of functions, the *apply()* functions, that allow you to manipulate slices of data from matrices, arrays, lists and data frames in a repetitive way, without having to use loops.

The family is made of the following functions: *apply()*, *lapply()*, *sapply()*, *vapply()*, *mapply()*, *rapply()*, and *tapply()*. In this module, we will cover the first three.

6.1 *apply()*

The prototype of the function is: *apply(X, MARGIN, FUN, . . .)*, where:

- *X* is an array or a matrix (if the dimension of the array is 2).
- *MARGIN* is a variable defining how the function is applied. When *MARGIN=1*, it applies over rows. When *MARGIN=2*, it works over column (when you use *MARGIN=c(1,2)*, it applies to rows AND columns).
- *FUN* is the function that you want to apply to the data. It can be any R function, including a User Defined Function (UDF).
- . . . denotes additional arguments to *FUN* that you can add to your call. For more information, type ??base::apply in the console.

Let's see an example. First, define m, a 10x2 matrix such as:

```
m = matrix(c(1:10, 11:20), nrow = 10, ncol = 2)

# How can we use the apply() function to calculate the mean of each row?
apply(m,1,mean)
```

```
##  [1]  6  7  8  9 10 11 12 13 14 15
```

```
# How can you modify the apply() function so it returns the mean of the columns?
apply(m,2,mean)
```

```
## [1]  5.5 15.5
```

6.2 *lapply()*

You will need to use *lapply()* when you want to apply a function to every element of a list. The output will be a list of the same dimensions as *X*. The prototype of the function is: *lapply(X, FUN, . . .)*, where:

- *X* is an input vector, atomic, list, or data frame.
- *FUN* is the function to be applied to each element of *X*

- ... denotes optional arguments to *FUN*.

```
#Imagine that you have a list of matrices, called matrices_list
M1<-matrix(1:9, 3,3)
M2<-matrix(4:15, 4,3)
M3<-matrix(8:10, 3,2)
matrices_list<-list(M1, M2, M3)

# Use lapply() to perform the same operation on all the matrices in matrices_list.
# For example, you can calculate the mean of the three matrices:
mean_list = lapply(matrices_list, mean)

mean_list
```

```
## [[1]]
## [1] 5
##
## [[2]]
## [1] 9.5
##
## [[3]]
## [1] 9
```

```
# Or the minimum value in each matrix:
min_list = lapply(matrices_list, min)

min_list
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] 4
##
## [[3]]
## [1] 8
```

You may have noticed that both *mean_list* and *min_list* can be difficult to deal with when carrying out further operations. For example, imagine that you want to access the minimum value of the second matrix and multiply it by two.

```
min_list[2] # returns a list [4]
```

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

```
min_list[2] * 2 # returns an error
```

```
## Error in min_list[2] * 2: non-numeric argument to binary operator
```

```
# But,
min_list[[2]] # returns 4
```

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

```
min_list[[2]] * 2 # returns 8
```

```
## [1] 8
```

We can simplify the format of the value returned by *lapply()*. For this, we have *sapply()*.

6.3 *sapply()*

The prototype of the function is: *sapply(X, FUN, . . . , simplify = TRUE)*, where:

- *X* is an input vector, atomic, list, or data frame.
- *FUN* is the function to be applied to each element of *X*.
- . . . denotes optional arguments to *FUN*.
- *simplify* is a Boolean value that denotes whether or not we are simplifying the output. If it is false, it will return the same value as *lapply()*.

```r
#Imagine that you are still working with matrices_list as previously defined:
M1<-matrix(1:9, 3,3)
M2<-matrix(4:15, 4,3)
M3<-matrix(8:10, 3,2)
matrices_list<-list(M1, M2, M3)

# Can you guess what the following will return?
# We are not simplifying the results, so it will return the exact same
# values and in the same format as mean_list.
means1 = sapply(matrices_list, mean, simplify=FALSE)
means1
```

```
## [[1]]
## [1] 5
##
## [[2]]
## [1] 9.5
##
## [[3]]
## [1] 9
```

```r
#Now, you can use the returned values more easily.
means2 = sapply(matrices_list, mean, simplify=TRUE)
means2
```

```
## [1] 5.0 9.5 9.0
```