G54DMA - Lab 2: Advanced R

Instructions

In last week's lab session, we introduced R and some of its basic functionality (arithmetic operations, vectors, matrices and simple plotting functions).

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.

For a comprehensive reference manual on all things R, please refer to *An Introduction to R* by Venables et al., which can be found here:
https://cran.r-project.org/doc/manuals/r-release/R-intro.pdf

Thorough the labs, we will be using *An Introduction to R* as a reference manual. Chapters or sections of interest will be signalled with "***Reading:***" in the Instruction sheets.

Learning outcomes

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

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

1. Writing Your Functions

A function file allows you to repeat a series of commands that you have created without having to type them all in 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.

Create a new source file by choosing “File -> New -> R script” from the menu enter the following:

```
myplot <-function(){
        pdf("g1.pdf");
        a = seq(0,1,0.1);
        c = a * (a^2);
        plot(a, c, col="blue", pch="*");
        dev.off()
     }
```

Save this as *myplot.r* . You can add this file to the project through the interface or by using the *source()* command. This will add *myplot()* to the library of functions you can call. You will now be able to call *myplot()* at the command line and see the effect (note that this file must be in the correct directory).

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

2. Control Structures

Control structures allow you to control the flow of execution of a script, typically inside of a function. Common ones include:

1. **If/else** statement: Used to make decision depending on different conditions**.**

```
# See the code syntax below for if-else statement
x=10

if(x>1){
 print("x is greater than 1")
 }else{
  print("x is less than 1")
  }

#See the code below for 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 between 8 and 15")
 }else{
      print("x is smaller than 1 or larger than 15")
}

[1] "x is between 8 and 15"
```

2. **For** loop: Used for iterating through items

```
 #Below code shows for  loop implementation
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
```

3. **While** loop :

```
 #Below code shows while loop in R
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
```

4. **Repeat** loop**:** an infinite loop used in association with a break statement.

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

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

5. **Break** statement: used in a loop to stop the iterations and flow the control outside of the loop.

```
  #Below code shows break statement:
  x = 1:10
   for (i in x){
       if (i == 2){
           break
       }
       print(i)
   }
  [1] 1
```

6. **Next** statement: skips the current iteration of a loop without terminating it.

```
  #Below code shows 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.

3. Reading and Writing from a File

During the course of your work, you may need to save certain variables or results. R has built-in functions that can help you store data into different file formats and read it into different types of variables.

Let's work with an example: define the matrix *A* as:

```
A = rbind(c(1,2,4,5,23,1,1), c(3,4,9,1,2,1,0))
A = t(A)
```

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. The first thing to do is to ensure that the current directory is one we can write into and read from.

We can find which directory is the current working directory with *getwd()* and change it with *setwd()*, or we can use the File menu. Note that the forward slash '/' is used within R as the directory separator, so we might do something like:

```
setwd("H:/Temp")
```

Now, the matrix A can be written to a file with:

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

Look at this file to see what you have got.

You may have noticed that each row and column has been assigned generic names ("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=FAL
SE)
write.table(A, file="data4.txt", col.names = c("T1", "T2"), row.n
ames=FALSE)
```

What are the differences in the files that are produced?

We can load files as well – 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 which read and write delimited text if you want delimiters other than space delimiters

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

Colon delimited files are used on some unix systems – it is more usual to use commas or tabs on windows systems.

3.1 Reading and writing from 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. Each line (or row) of the file is a data record (or an instance) and each instance consists of one or more fields, separated by the delimiter.

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, or even EXCEL.

To read data from a CSV to a data frame (a type of variable which we will introduce in the next section), use the *read.csv* function:

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

Remember that the first folder in which R will look for files 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=",")
```

“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 (something that will be extremely useful for your lab submission and coursework), you can use the write.csv function:

```
# 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.

4. Lists and Dataframes

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 type into one single variable. For example, you might want to store the name, gender, age and nationality of a group of 20 people. For this, R offers **lists** and **data frames**.

In R, *lists* are an ordered collection of objects, known as components. Components can be of different type.

An example of a list is:

```
person = list(name="John", surname = "Silver", age = c(30),
profession = "pirate of the high seas", likes = c("treasure",
"cooking", "pirating", "parrots"))
```

Note that each component is numbered and that each of them has lengths. You can refer to each component by using the [] operator. Additionally, you can access each component using its name and the $ operator.

```
person[2]
person$surname
```

person$likes
person$likes[3]

A *data frame* is a list with class “data.frame”. For many purposes, a data frame can be regarded as a matrix with columns of different types. It is displayed and indexed following matrix conventions.

Execute the following two commands:

```
people_list <- list(name=c("John","Billy"), surname = c("Silver",
"Bones"), profession = c("captain", "pirate of the high seas", "t
hief"))

people_frame <- data.frame(name=c("John","Billy"), surname = c("S
ilver","Bones"), profession = c("cook", "pirate of the high seas
"))
```

Can you pinpoint the difference? In the list, there are no relationships between the components (that is why, each component can have a different length). On the other hand, in the data frame, components are linked through columns and rows (therefore, lengths between “components” must match).

Try the following:

```
people_frame_err<- data.frame(name=c("John","Billy"), surname =
c("Silver","Bones"), profession = c("cook", "pirate of the high
seas", "thief")) #this returns an error because the size of
profession is different from name and surname.
```

Now, let’s try to access different fields:

```
people_list[2]
people_frame[2]

people_list[2,]  #this returns an error – why?
people_frame [2,]
```

Because lists have no relationships between components, you can add values to only one field in a list. For example:

```
people_list <- list(name=c("John","Billy","Jack"), surname =
c("Silver","Bones"), profession = c("cook", "pirate of the high
seas"))

people_list # What does this return?
```

Now, try to do the same with the dataframe *people_frame*:

```
people_frame <- data.frame(name=c("John","Billy", "Jack"),
surname = c("Silver","Bones"), profession = c("cook", "pirate of
the high seas"))

people_frame # this returns an error
```

As we saw before, this returns an error due to the different lengths of each field or component.

To add a new value to a dataframe, you will have to add an input a value for all fields within the dataframe. For example:

```
people_frame <- data.frame(name=c("John","Billy","Jack"), surname
= c("Silver","Bones","Rackham" ), profession = c("cook", "pirate
of the high seas","captain"))

people_frame
```

As a final note, let's combine points 3 (Reading and Writing) and 4 (Dataframes). You can easily write dataframes as CSV files:

```
write.csv(people_frame, "crew.csv")
```

If you open *crew.csv* in Excel, you will see that each field in the dataframe (i.e. name, surname, and profession) is now a named column, and its corresponding values, a row.

You can quickly read back the file into a dataframe using:

```
people = read.csv("crew.csv") #if omitted, header is true and sep
is “,”
```

You can see that you have an additional filed which has the number (or position) of each instance.

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

5. 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 dataframes 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.

5.1 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 columns.
    - Note that 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) #returns [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) # returns [1]  5.5 15.5
```

5.2 lapply():

The prototype of the function is:

lapply(X, FUN, …)

where:

- *X* an input vector, atomic, list, or dataframe.
- *FUN* the function to be applied to each element of *X*
- ... denote optional arguments to *FUN*.

You 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*.

Let's see an example. 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(A,B,C)
```

You can now 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
```

Or the minimum value in each matrix:

```
min_list = lapply(matrices_list, min)

min_list
```

You may notice 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]

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

However,

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

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

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

5.3 sapply():

The prototype of the function is:

sapply(X, FUN, ..., simplify = TRUE)

where:

- *X* an input vector, atomic, list, or dataframe.
- *FUN* the function to be applied to each element of *X*
- ... denote 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().*

Let's see an example. 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(A,B,C)
```

If we do:

```
means = sapply(matrices_list, mean, simplify=FALSE)
```

Can you guess what it will return? Because we are not simplifying the results, it will return the exact same values and in the same format as *mean_list*.

To compare, execute:

```
means = sapply(matrices_list, mean, simplify=TRUE) # instead of a
lists, sapply returns a vector. Now, you can use the returned
values more easily.
```