Conditional statements, Loops and Function creation

In this lesson, you will learn how to save and load the workspace from and into R. Furthermore, you will learn the conceptual as well as the syntax of "conditional statement", "loop" and "writing own functions".

Lesson Plan

Saving and loading objects in R files

- Save the **workspace**
- Save **one or more objects**
- Load **.RData** files

Conditional statement

- **Conceptual** principles
- **Syntax** principles

Loop

- **Conceptual** principles
- **Syntax** principles

Write functions

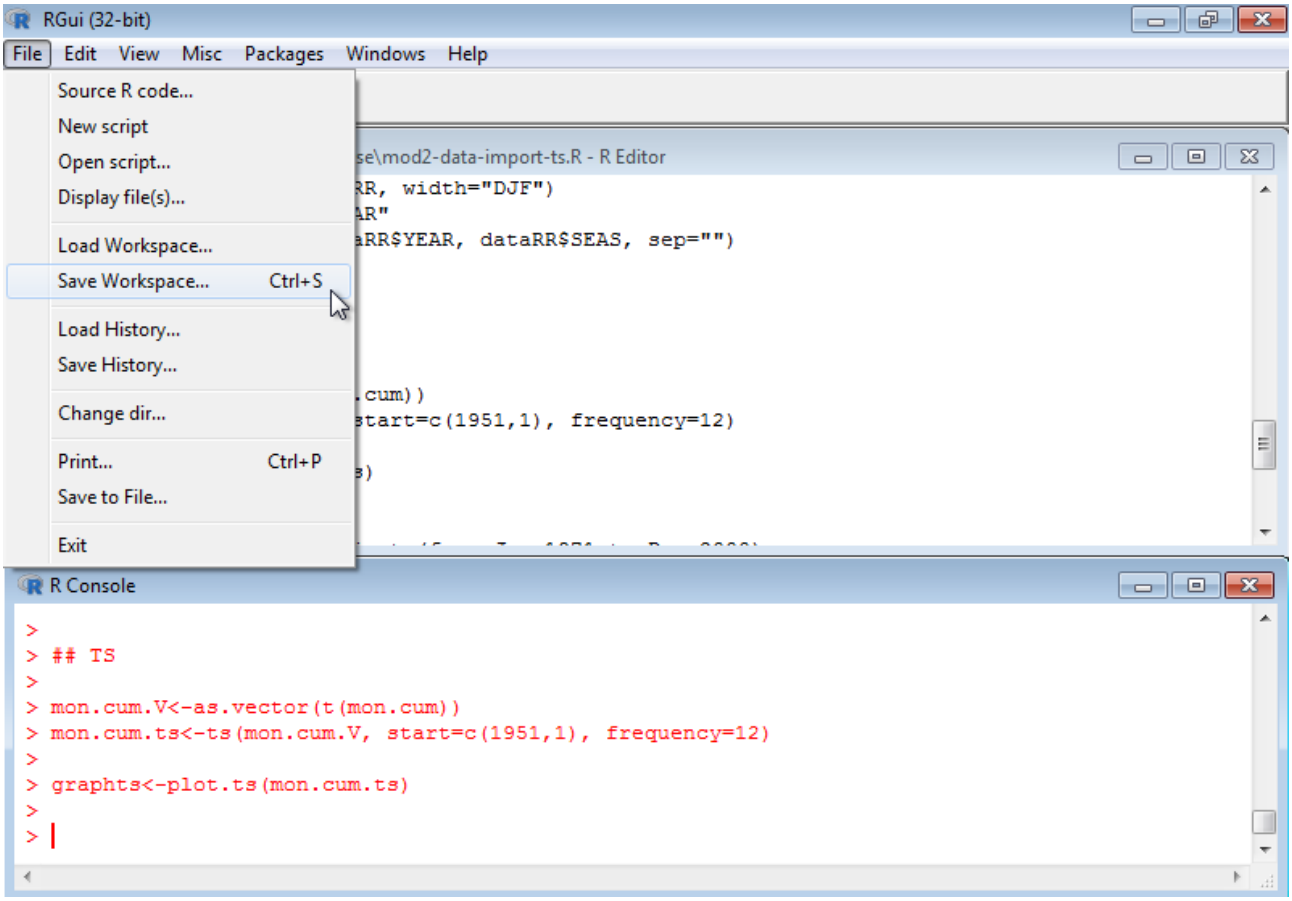
- **Why** and **How** to

Saving and loading objects in R files

Save workspace image

You can save the workspace, i.e. **the whole objects created** during your working session in a R file (.RData):

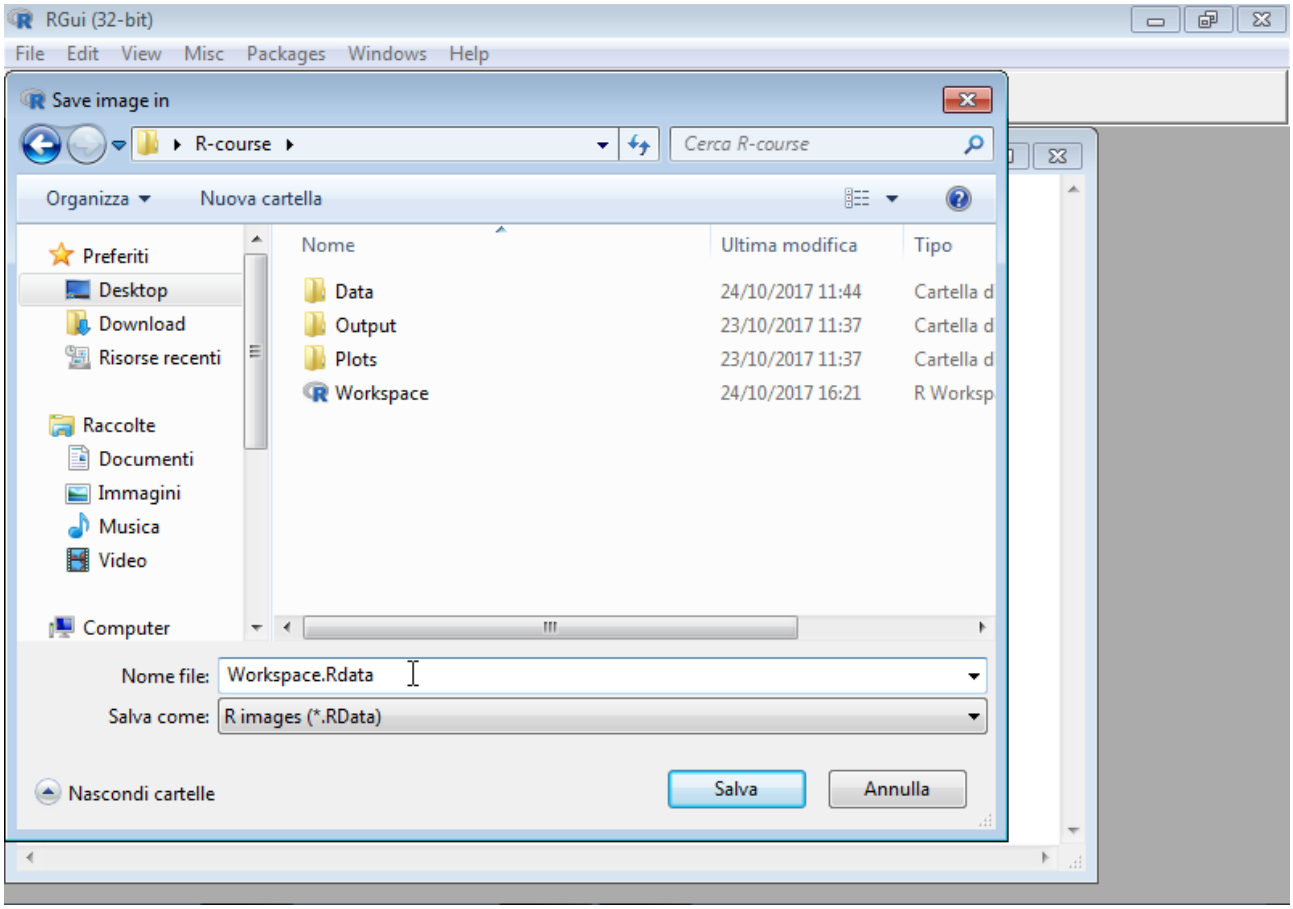
1) using the Graphical user interface (GUI)



2) from command line, let's say "***Workspace***" the name of this file

```
save.image("~/Desktop/R-course/Workspace.RData")
```

In both cases, a new file is created in the **working directory** (you may want to find out which is it by typing **getwd()**):



It might be useful to have objects ready to work on, especially if you are in the middle of a debugging phase but

BE CAREFUL WHEN USING BIG DATASET SINCE THE ".RDATA" FILE MAY BECOME MEMORY SPACE CONSUMING

Notice that you are requested to save workspace or not whenever you close R software.

Save one or more objects

You might want to overcome the memory issue by saving a single object (recall the objects of Module 2):

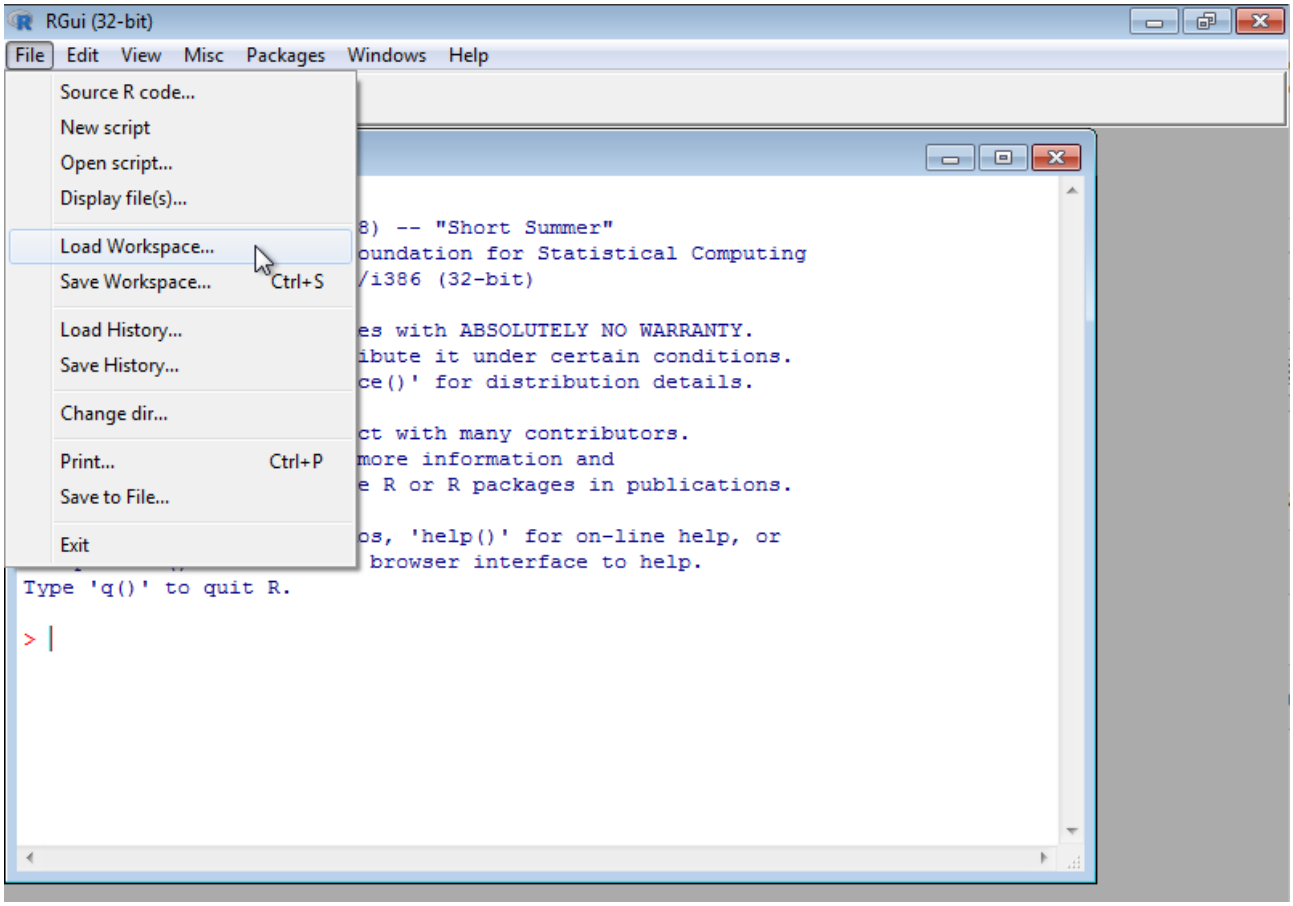
```
save(dataRR,file="dataRR.RData")
```

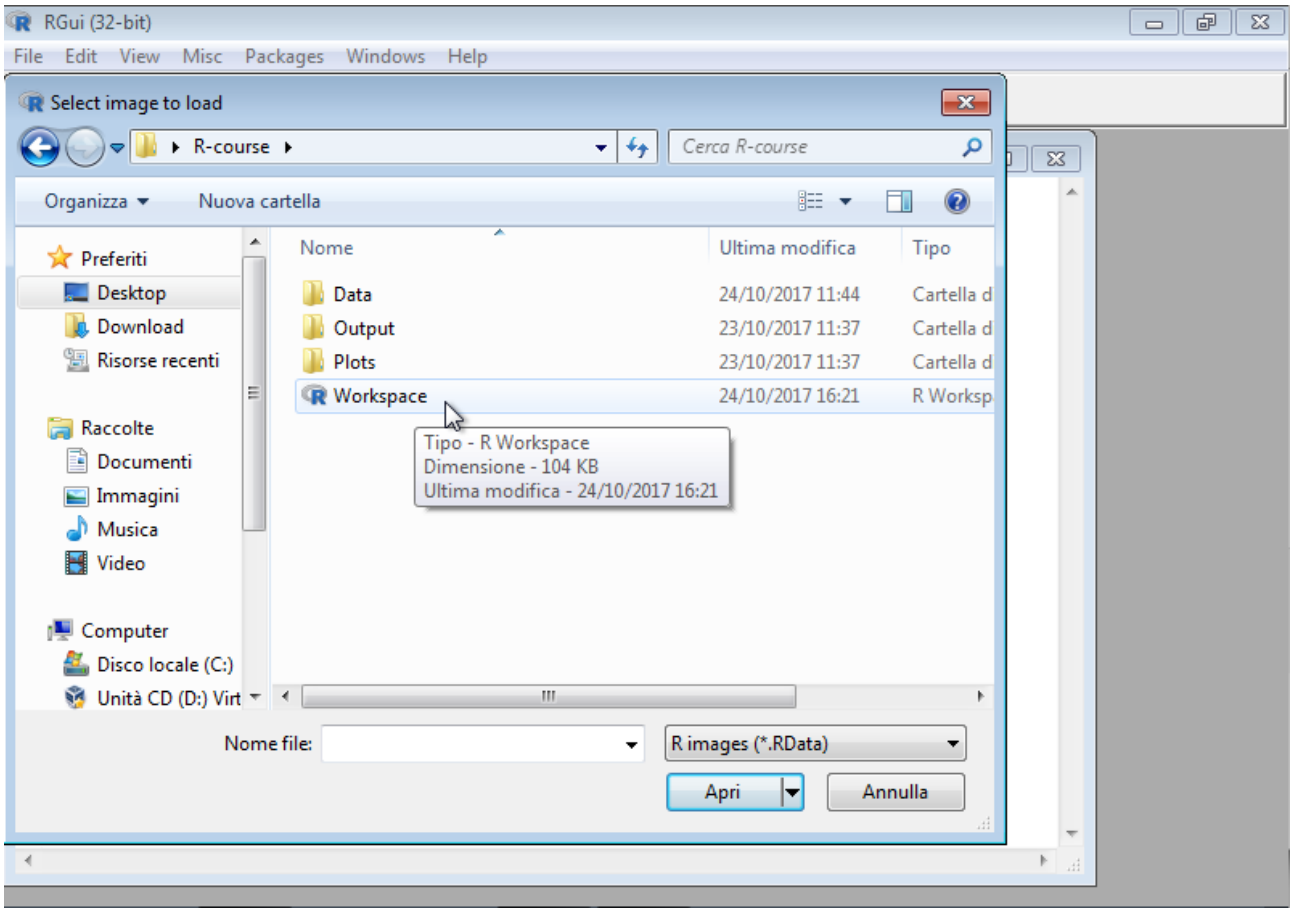
or a list of selected objects

```
save(list=c("dataRR","mon.cum"),file="TwoObjects.RData")
```

Load workspace or objects

Now, close and re-launch R. **Load the workspace** from the GUI pointing at the **Workspace.RData** file in the **R-course folder**:


RGui (32-bit)
File Edit View Misc Packages Windows Help
Source R code...
New script
Open script...
Display file(s)...
Load Workspace...
Save Workspace... Ctrl+S
Load History...
Save History...
Change dir...
Print... Ctrl+P
Save to File...
Exit
8) -- "Short Summer"
oundation for Statistical Computing
/i386 (32-bit)
es with ABSOLUTELY NO WARRANTY.
ibute it under certain conditions.
ce()' for distribution details.
ct with many contributors.
more information and
e R or R packages in publications.
os, 'help()' for on-line help, or
browser interface to help.
Type 'q()' to quit R.
>


RGui (32-bit)
File Edit View Misc Packages Windows Help
Select image to load
R-course
Cerca R-course
Organizza
Nuova cartella
Preferiti
Desktop
Download
Risorse recenti
Raccolte
Documenti
Immagini
Musica
Video
Computer
Disco locale (C:)
Unità CD (D:) Virt
Nome
Ultima modifica
Tipo
Data 24/10/2017 11:44 Cartella d
Output 23/10/2017 11:37 Cartella d
Plots 23/10/2017 11:37 Cartella d
Workspace 24/10/2017 16:21 R Worksp
Tipo - R Workspace
Dimensione - 104 KB
Ultima modifica - 24/10/2017 16:21
Nome file:
R images (*.RData)
Apri
Annulla

Question-List of loaded objects

Which command do we use to list the objects loaded in the R engine?

str()

ls()

summary()

Conditional statements/1

to DO: create a new script file in the R-course project and name it "*mod3-if-for-ownfunction.R*"

Conceptual framework and Syntax

An important subject of programming is being able to determine **if a condition is true or false** and then **perform specific actions depending on the outcome of the condition test**.

For instance, let us set the variable **x** to **1:**

```
x = 1    # the symbols "=" and "<-" are interchangeable
```

This variable is an object of type "**vector**":

```
class(x)
[1] "numeric"
```

Then, we can **test a general condition** on the **value** of the object **x** and **define a consequent action to do.**

For example, let us ask the system **if the value of x is greater than *0*** and, if the response is positive, to **print the value of object x.** The translation of the previous sentence from lexical to R language is:

```
if(x > 0) x
```

We can also use the function **print()** for more complex printing options, such as the implementation of **paste()** function, that is used to combine numeric and string elements:

```
if(x > 0) print(paste("x =",x))
```

We can make also an explicit request of equality using the symbol "**==**":

```
if(x == 1) x:(x+10)   # the ":" symbol is used to build a sequence

if(x == 1) (x+3)/2      # the "/
" symbol is used for mathematical division
```

The latter sequence of commands can be also written in a more compact way using the symbol "**;**", as follows:

```
if(x == 1) x:(x+10); if(x == 1) (x+3)/2
```

Question-Conditional statement

Select the correct answer in each box.

Correct answer.

Incorrect answer.

```
if(x == 1) x:(x+10)
```

```
if(x == 1) (x+10):x
```

```
if(x = 1) x+10
```

```
if(x == 1) x<-x+10; if(x <= 10) "yes" else "no"
```

Conditional statement/2

The subsequent actions of a conditional test can be divided into:

- **action1** to do **if** test response is **TRUE**;
- **action2** to do **if** test response is **FALSE**.

The command to divide the two types of action is

if(response is **TRUE)**

action1

else

action2

A very compact way to code such a conditional test is

```
if(x > 2) "yes" else "no"       # or alternatively
if(x > 2) print("yes") else print("no")
```

When ***actions*** are more elaborated, the use of curly brackets is needed to separate **actions1** from **actions2**. The following example shows how to use them:

```
x <- c(4,10, NA, NA)
    # x is a vector with missing data in position 3 and 4
```

recall the usage of the function **is.na()**. This function returns a logical vector that contains **TRUE** in the **place of the vector** where NA is located:

```
is.na(x)
> FALSE FALSE TRUE TRUE
```

and the function **which()**returns their position:

```
which(is.na(x))
[1] 3 4
```

Now, the conditional test is:

1. is there any missing data in **x**?
2. if yes then substitute missing with the minimum of non-missing data;
3. if not then let the **x** vector unchanged

and the correspondent code is:

```
if(length(x[is.na(x)]) > 0 ){    #we use length() to count missing data
     position.NA<-which(is.na(x))
     x[position.NA] <- min(x, na.rm=TRUE)
  }else{
     x <- x
  }
```

Loops

Conceptual framework and Syntax

Whenever we need to repeat the same **action** for some objects (vectors, matrixes, arrays, data frames, lists etc) or part of them (elements of a vector, columns/rows of a matrix, array, data frame), we use a framework of **repeating code** that is called "**Loop**".

There are **three different looping frameworks** in R:

1. for
2. **repeat**
3. while

To stop iterating through a loop we use the **break** statement.

To skip to the next iteration without evaluating the remaining expressionsin the loop body, we use the command **next**.

For

The **for** loop iterates through each element in, say, a vector and the index (counter) "**i**" is incremented of 1 at each iteration. Its syntax in R is:

for(i in *sequence of elements*)

{

action

i=i+1

}

We do not need to write explicitly i=i+1 in the coding since it is implicit in the **for** call.

As an example, the following R code prints out multiples of 2 up to 10:

```
for(i in 1:5) {i <- i*2; print(i)}
[1] 2
[1] 4
[1] 6
[1] 8
[1] 10
```

Repeat

The **repeat** loop just repeats the same *action* under a condition:

repeat *action* break *action*

Let's rewrite the example above using a **repeat** loop:

```
i <- 1
repeat {if(i <= 5) {i <- i*2; print(i)} else break}
[1] 2
[1] 4
[1] 8
```

Notice that if you don to include a **break** command in **repeat** construct, the R code will be an infinite loop.

While

Again, let's rewrite the example above using a **while** loop:

```
i <- 1
while(i <= 5){i <- i*2; print(i)}
[1] 2
[1] 4
[1] 8
```

Loops: for

Something more on “for” loop

We use now “i” as a pointer to the values of an object, i.e. **x[i] is the i-th elements of x.**

Then, since “i” increments of 1 at each iteration, we access iteratively the 1st element x[1], 2nd element x[2],….., n-element x[n].

Example1

```
x<-c(2,5,9,12,15)
y<-c()     #set an empty vector "y"
for (i in 1:length(x)) {
  x2 <- (x[i]+1)^2
  y <- c(y,x2)       #populate the i-th element of y at each iteration
}
xy <- cbind(x,y)        #cbind() builds a two column data frame
```

Example1bis

```
x<-c(2,5,9,12,15)
y<-numeric(length = length(x))
#set a vector "y" of 5 elements (as long as "x"), each of value equals to
 0
for (i in 1:length(x)) {
  y[i] <- (x[i]+1)^2
}
xy <- cbind(x,y)
```

TIP: you should know that the good programmer **avoids as much as possible the use of loops**. Then, the compact solution for **Example1** and **Example1bis** is simply:

```
x<-c(2,5,9,12,15)
xy<-cbind(x,(x+1)^2)
```

Question-Loop

Let us write some code to transform values of a vector from "**centimeters**" to "**inch**".

```
x_cm <- seq(10,100,by=20)   # height in cm
```

Which of the following code line is correct?

```
for(i in x_cm){x_in = i/2.54; print(paste(i,"cm is equivalent to",round(x
_in,1),"inches"))
```

```
for(i in x_cm){x_in = i/2.54; print(paste(i,round(x_in,1)))}
```

```
for(i in x_cm){x_in= x/2.54}
x_in
```

Write your own functions

Every statement in R—setting variables, doing arithmetic, repeating code in a loop—can be written as a function.

A very simple function

Write the following code in console and press return:

```
From_cm_to_inches = function(x){x/2.54}  #x=height_in_centimeters
```

(The *name* given to the function is up to you).

Nothing happens apparently, nevertheless you have included the function From_cm_to_inches() in the **R environment** and you can use it **exactly as you do with other functions**. The function works assigning whatever value to the x variable and pressing return…….

```
From_cm_to_inches(x=2)
```

or even a vector

```
> From_cm_to_inches(x=c(2,5,7))
[1] 0.7874016 1.9685039 2.7559055
```

ESSAY: write the function that transforms inches to centimeters and pastes the code below.

Re-writing a loop construct in a function form

Now, recall the Example1bis done in the loop card:

Example1bis

```
x<-c(2,5,9,12,15)
y<-numeric(length = length(x))
for (i in 1:length(x)) {
  y[i] <- (x[i]+1)^2
}
xy <- cbind(x,y)
```

and let us re-write it in a **functional form** as follows:

```
Simple_math<-function(x){
    y <- (x+1)^2
    xy <- cbind(x,y)
    return(xy)
    }
```

Again, execute function code to load it into the R Environment and assign values to x to use it. For instance,

```
MyVector<-c(2:6,NA,6:2)
Simple_math(MyVector)
```

Notice that the **return** command is used inside functions in order **to define which object has to be the output**.

ESSAY: write a new function starting from **Simple_math()**, where the third column with **exp((x[i]+1)^2)** is added to the output and paste the code below.

Final exercise Module 3

Make the following exercise and paste below the correspondent code.

Block A: using loop

1. assign to **x** the output of **rnorm()** function that generates random number from a Gaussian distribution (for example, set mean=10 and variance=2);
2. find outliers, let us say when value is greater than "**mean + 1.96*standard deviation**";
3. print a data frame that contains two columns: one with the original vector and the other with the string "outlier" when needed.

Block B: writing function

1. repeat the same exercise using function construct instead of loop