A minimalist introduction to R

Timothée Bonnet & al.

February 4, 2018

Contents

There are many ways to achieve the same goal in `R`*, and we do not claim to teach you the most efficient way to use* `R`*. You may find more elegant ways!*

Do try to understand exactly what the code and the functions we use do. The best way to learn how functions work is by either using the `R`*-manual (type* `?functionname` *or use the RStudio Help tab by clicking on it or pressing F1) or by creating dummy data (just make up a small amount of data yourself, using* `R` *if possible!) and analyse what the function does to this data.*

How this document works

R code and output is generally contained within boxes with a gray background. Comments within the R code start with a # symbol; lines with R-outputs start with ##.

All the files necessary to go through the workshop are (or should be!) in the folder of a github repository. We recommend you copy these files, or fork the repository if you are a git user.

Now, let's the fun begin.

1 Trash your calculator

1.1 Operators

`R` can be used as a calculator, and a far more powerful one that any physical calculator. If you use your calculator to enter numbers in `R`, you are being inefficient.

Below we demonstrate the use of some basic mathematical operators:

```
1+3 #addition

## [1] 4

5-2 #substraction

## [1] 3

6*4 #multiplication

## [1] 24

14/2 #division

## [1] 7

2^3 #exponent

## [1] 8

2**3 #or equivalently

## [1] 8
```

There are many mathematical functions already present in R:

```r
exp(3) #exponential

## [1] 20.08554

log(2.71) #logarithm

## [1] 0.9969486

sqrt(9) #square root, which of course you can also write as:

## [1] 3

9 ^ (1/2)

## [1] 3

sin(pi/2); cos(1); tan(pi/3) #trigonometric functions

## [1] 1
## [1] 0.5403023
## [1] 1.732051
```

Small exercise
Use R to compute

$$y = \frac{1}{2\sqrt{2\pi}} e^{\frac{-1}{2}\left(\frac{3-\pi}{2}\right)^2}$$

Logical operators are very important for programming and scripting. You can test whether two things are equal with double = signs:

```r
3 == 6/2 #is 3 equal to 6/2? TRUE!

## [1] TRUE

3 == pi  # FALSE!

## [1] FALSE
```

You can also test if they are NOT equal with the operator `!=`:

```r
2 != 3

## [1] TRUE

2 != 2

## [1] FALSE
```

The AND operator is `&`

```
2 == 2 & 3==3

## [1] TRUE

2 ==2 & 3==2

## [1] FALSE
```

The OR operator is |

```
2 == 2 | 3==2

## [1] TRUE

2 == 4 | 3==2

## [1] FALSE
```

Small exercise
Try and guess the result of these logical tests before running them:

```
! 1==2
(1!=2 | 3==4) & (2==4/2)
"abc" != "bc"
```

1.2 Assignment

Values can be assigned to objects to store them and make your code flexible. You assign a value to an object using the operator `<-` (or `=`, but be careful not to confuse this with the `==` used in tests).

```
#You can use objects in calculation
a <- 12
a + 2

## [1] 14

# you can assign an object value to another object
b <- a
c <- a*b
```

```r
# you can re-assign an object
a <- "c"
b <- "c"
a == b

## [1] TRUE

c <- a == "b"
c

## [1] FALSE
```

2 Containers

A container is some kind of object that can contain several values.

2.1 Vectors

The simplest container is a vector. A flexible way to create a vector by *concatenating* several values with the syntax `c(x,y,...)`.

```r
a <- c(3,9,3,5)
a

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

You can now do calculations on your vector:

```r
a * 2

## [1]  6 18  6 10
```

You can access one or several elements in the vector using squared brackets

```r
#access one value
a[1]

## [1] 3

a[2]

## [1] 9
```

```
#access multiple values by concatenating locations
a[c(1,3)]

## [1] 3 3

#access mutiple successive values
a[2:4]#the syntax x:y means "all integers between x and y"

## [1] 9 3 5

#modify a value
a[3] <- -5
a

## [1]  3  9 -5  5

#modify mutiple values
a[1:2] <- 1
a

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

2.2 Matrix

A matrix is similar to a vector, but in two dimensions. You can create one with the function `matrix()`.

For instance:

```
a <- matrix(data = c(1,2,3,4), nrow = 2)
a

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

You can access the cell in the row i of the column j using squared brackets like for vectors, but since there are two dimensions rather than one, you may give two numbers: `a[i,j]`.

```
#extracting the element in the first row of the second column:
a[1,2]

## [1] 3
```

```
# extracting all of the second row:
a[2,]

## [1] 2 4

# changing all of the first column:
a[,1] <- 29
a

##      [,1] [,2]
## [1,]   29    3
## [2,]   29    4
```

That is all for now. If you want to learn more, check the help for this function, using:

```
?matrix
```

2.3 Data-frame

Data-frames are similar to matrices, but are much more flexible: they can store different data types and their elements can be accessed in more efficient ways.

R is probably most efficient and user-friendly when analyses rely on data-frames.

```
plant_data <- data.frame(plant = c("potatoes", "water hemlock", "carrot"),
                         number= c(3,5,39), danger= c(FALSE,TRUE,FALSE))

plant_data

##           plant number danger
## 1      potatoes      3  FALSE
## 2 water hemlock      5   TRUE
## 3        carrot     39  FALSE
```

You can access and modify elements in the same way as for matrices:

```
plant_data[3,2]

## [1] 39

plant_data[3,]

##    plant number danger
## 3 carrot     39  FALSE
```

But you can also use column names, which are more human-friendly than numbers:

```
plant_data[1,"plant"]

## [1] potatoes
## Levels: carrot potatoes water hemlock

plant_data[,"danger"]

## [1] FALSE  TRUE FALSE
```

In some case it is easier to work with a different syntax using the dollar sign. Below, we access the same elements using this alternative syntax:

```
plant_data$plant[1]

## [1] potatoes
## Levels: carrot potatoes water hemlock

plant_data$danger

## [1] FALSE  TRUE FALSE
```

3 Loops and if statements

Loops allow you to

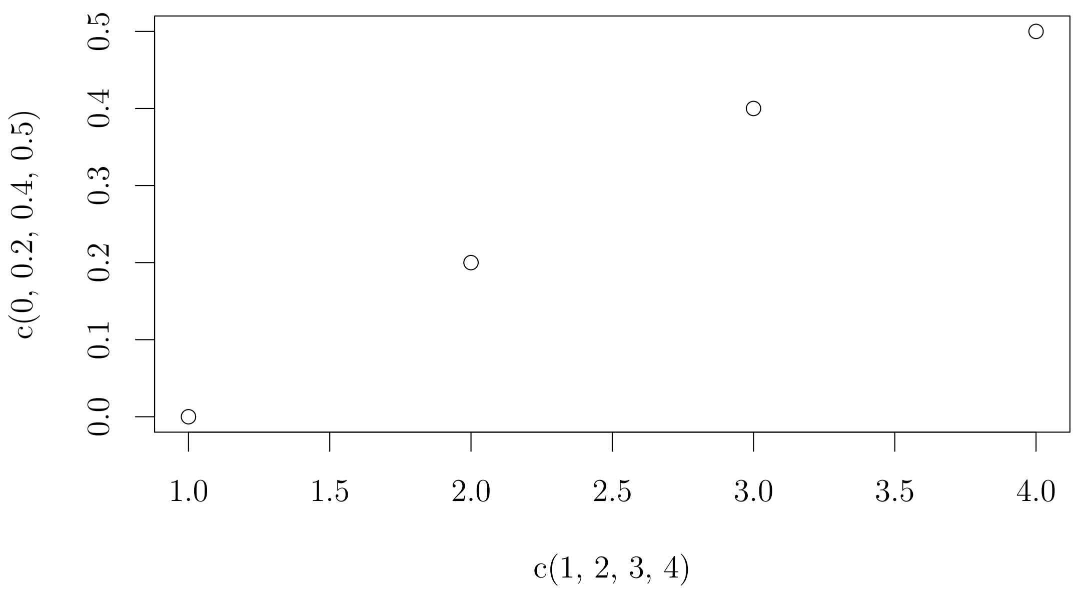
3.1 for loops

3.2 if statements

4 Simple Graphics

You can create graphics with various functions, the most fundamental one being `plot()`. For instance:

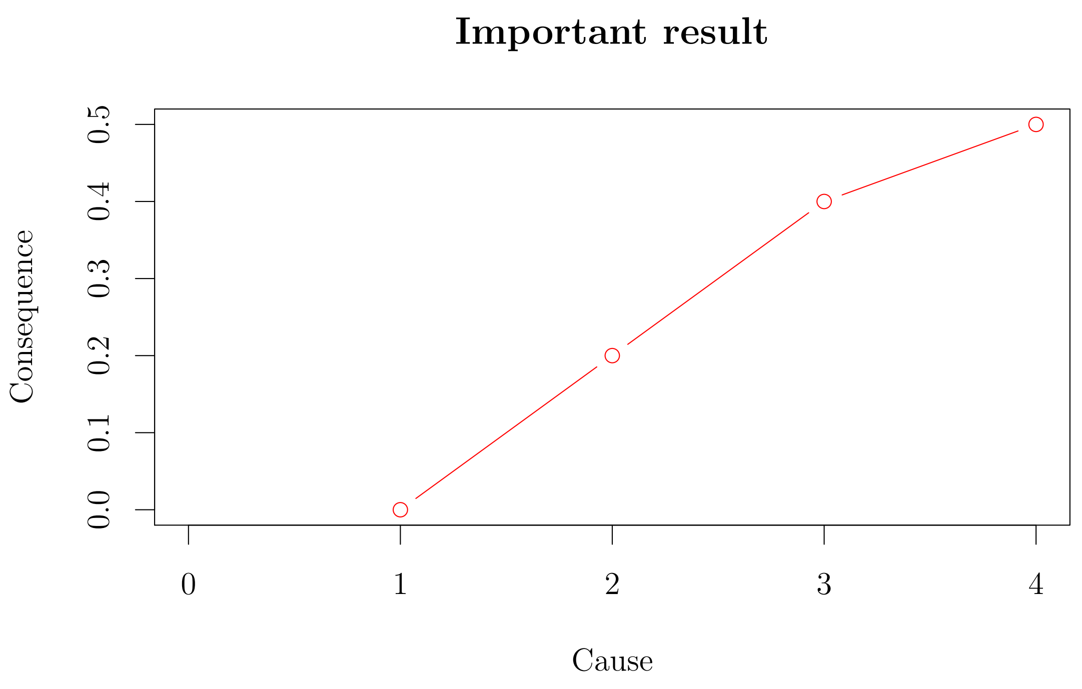
```
plot(x = c(1,2,3,4), y=c(0,0.2,0.4,0.5))
```



(the graph should appear when you run this line)

That's rather ugly, and we will learn how to make beautiful graphes in efficient ways. For now, let's just tweak a few things to demonstrate the use of options within the plot function:

```
plot(x = c(1,2,3,4), y=c(0,0.2,0.4,0.5), type = "b",
     main = "Important result", xlab = "Cause",
     ylab = "Consequence", xlim = c(0,4), col="red")
```



Small exercise
Modify the code above to obtain a graph with a y-axis that goes up to 1 (maybe what we are measuring on the y-axis is a proportion, so it seem fair to show the axis from 0 to 1), with the data being represented by a line only (without the dots), plotted in blue instead of red.