# A minimalist introduction to R

Timothée Bonnet
(thanks to Koen van Benthem and Ashley Latimer for help)

February 6, 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 observe what the function does with these data.*

## How this document works

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

There are a few small exercises in each section. If you encounter any difficulty with them, ask me for help before the workshop (for instance just before, I will be in the room 30 minutes early).

Now, let 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 #other version for exponent

## [1] 8
```

There are many mathematical functions already present in R:

```
exp(3) #exponential

## [1] 20.08554

log(2.71) #logarithm

## [1] 0.9969486

sqrt(9) #square root

## [1] 3

9 ^ (1/2) # other version for square root

## [1] 3

#trigonometric functions :
#(use ";" for several expressions on the same line)
sin(pi/2); cos(1); tan(pi/3)

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

**Small exercise**
Use R to compute:

$$y = (\cos(0.1\pi))^3$$

You should get 0.8602387

And:

$$y = \log(3 - 2) + 5$$

You should get 5

And if you dare:

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

You should get 0.1989719

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

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

## [1] TRUE
```

```r
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 `!` symbol means “not” in general, so you can use it to get the opposite result:

```r
!TRUE

## [1] FALSE

!FALSE

## [1] TRUE
```

The AND operator is `&`

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

## [1] TRUE

2 ==2 & 3==2

## [1] FALSE
```

The OR operator is |

```r
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

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

## [1] TRUE

c <- a == "b"
c

## [1] FALSE
```

# 2 No need to write it on paper: Containers

A container is some kind of object that can store values, like numbers, characters, booleans (TRUE/FALSE)... We will present vectors, matrices and data-frames.

## 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,...)`.

```
a <- c(3,9,3,5) # c is for concatenate
a

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

You can now do calculations on your vector:

```
a * 2

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

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

```
#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) #fills columns before rows
a

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

#if you want to fill rows before columns:
matrix(data = c(1,2,3,4), nrow = 2, byrow = TRUE)

##      [,1] [,2]
## [1,]    1    2
## [2,]    3    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", "hemlock", "carrot"),
                         number= c(3,5,39), danger= c(FALSE,TRUE,FALSE),
                         stringsAsFactors = FALSE)

plant_data

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

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

```
plant_data[3,2] #first number is for row, second for column

## [1] 39

plant_data[3,] #get all columns for the third row

##    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"] # first row of column named "plant"

## [1] "potatoes"

plant_data[,"danger"] # all of "danger" column

## [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"

plant_data$danger

## [1] FALSE  TRUE FALSE
```

It is very easy to create new columns, with one or the other syntax:

```
plant_data[,"tasty"] <- c(TRUE, NA, TRUE) #NA (Not Applicable) indicates missing val

plant_data$color <- c("variable", "green", "orange")

plant_data

##      plant number danger tasty    color
## 1 potatoes      3  FALSE  TRUE variable
## 2  hemlock      5   TRUE    NA    green
## 3   carrot     39  FALSE  TRUE   orange
```

You can also add new entries (rows):

```
plant_data[4,] <- c("eucalyptus", 24, NA, FALSE, "green")
plant_data

##        plant number danger tasty    color
## 1   potatoes      3  FALSE  TRUE variable
## 2    hemlock      5   TRUE  <NA>    green
## 3     carrot     39  FALSE  TRUE   orange
## 4 eucalyptus     24   <NA> FALSE    green
```

**Small exercise**
Imagine you are a koala. Change the information about eucalyptus tastiness in the data-frame `plant_data` (yes, you will need to create this data-frame using the code above if you haven't done that already.)

## 2.4 What is this object again?

If you don't remember what your object is, or what it contains, you can use the function `str`:

```
str(plant_data)# a data.frame

## 'data.frame': 4 obs. of  5 variables:
##  $ plant : chr  "potatoes" "hemlock" "carrot" "eucalyptus"
##  $ number: chr  "3" "5" "39" "24"
##  $ danger: chr  "FALSE" "TRUE" "FALSE" NA
##  $ tasty : chr  "TRUE" NA "TRUE" "FALSE"
##  $ color : chr  "variable" "green" "orange" "green"

str(a) # a num(eric vector)

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

That works not only for containers, but for any R object:

```
str(1) #num(eric)

##  num 1

str(TRUE) # logi(cal)

##  logi TRUE

str("banana") #chr (character string)

##  chr "banana"

str(exp) # a function taking one parameter...

## function (x)
```

## 3 For loops

Loops are a way to automatize repetitive tasks.
To demonstrate this, let's load some data that are buit-in R:

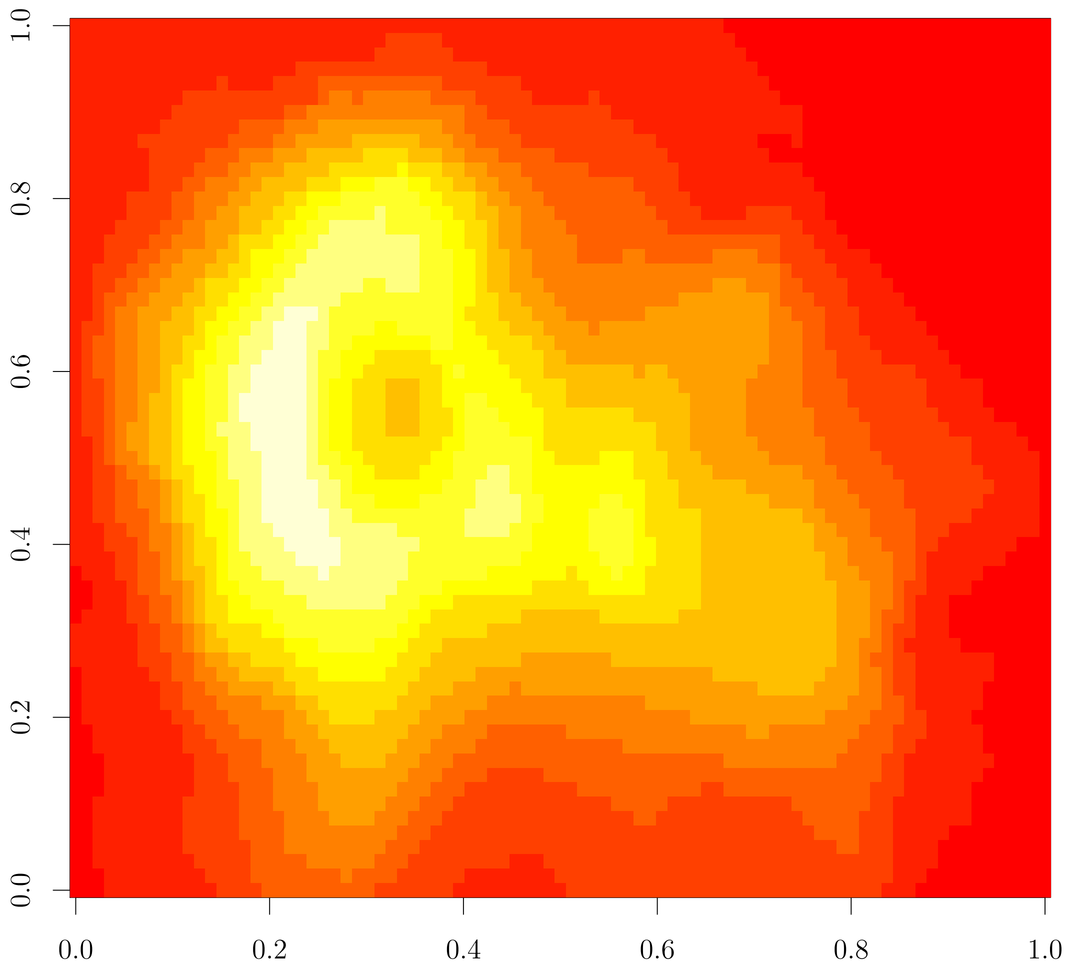
```
data(volcano)
str(volcano)

##  num [1:87, 1:61] 100 101 102 103 104 105 105 106 107 108 ...
```

The function `str` tells us that the object volcano is a numeric object, with two dimensions, a.k.a. a matrix, and that this matrix has 87 rows and 61 columns.
We can have a look at the first rows of the matrix with the function `head`:

```r
head(volcano)#shows the first rows of the matrix
## 	   [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
## [1,]  100  100  101  101  101  101  101  100  100   100
## [2,]  101  101  102  102  102  102  102  101  101   101
## [3,]  102  102  103  103  103  103  103  102  102   102
## [4,]  103  103  104  104  104  104  104  103  103   103
## [5,]  104  104  105  105  105  105  105  104  104   103
## [6,]  105  105  105  106  106  106  106  105  105   104
##      [,11] [,12] [,13] [,14] [,15] [,16] [,17] [,18] [,19]
## [1,]   101   101   102   102   102   102   103   104   103
## [2,]   102   102   103   103   103   103   104   105   104
## [3,]   103   103   104   104   104   104   105   106   105
## [4,]   103   104   104   104   105   105   106   107   106
## [5,]   104   104   105   105   105   106   107   108   108
## [6,]   104   105   105   106   106   107   109   110   110
##      [,20] [,21] [,22] [,23] [,24] [,25] [,26] [,27] [,28]
## [1,]   102   101   101   102   103   104   104   105   107
## [2,]   103   102   102   103   105   106   106   107   109
## [3,]   104   104   105   106   107   108   110   111   113
## [4,]   106   106   107   108   110   111   114   117   118
## [5,]   108   109   110   112   114   115   118   121   122
## [6,]   112   113   115   116   118   119   121   124   126
##      [,29] [,30] [,31] [,32] [,33] [,34] [,35] [,36] [,37]
## [1,]   107   107   108   108   110   110   110   110   110
## [2,]   110   110   110   110   111   112   113   114   116
## [3,]   114   115   114   115   116   118   119   119   121
## [4,]   117   119   120   121   122   124   125   126   127
## [5,]   121   123   128   131   129   130   131   131   132
## [6,]   126   129   134   137   137   136   136   135   136
##      [,38] [,39] [,40] [,41] [,42] [,43] [,44] [,45] [,46]
## [1,]   110   110   110   108   108   108   107   107   108
## [2,]   115   114   112   110   110   110   109   108   109
## [3,]   121   120   118   116   114   112   111   110   110
## [4,]   127   126   124   122   120   117   116   113   111
## [5,]   132   131   130   128   126   122   119   115   114
## [6,]   136   136   135   133   129   126   122   118   116
##      [,47] [,48] [,49] [,50] [,51] [,52] [,53] [,54] [,55]
## [1,]   108   108   108   108   107   107   107   107   106
## [2,]   109   109   109   108   108   108   108   107   107
## [3,]   110   110   109   109   109   109   108   108   107
## [4,]   110   110   110   109   109   109   109   108   108
## [5,]   112   110   110   110   110   110   109   109   108
## [6,]   115   113   111   110   110   110   110   109   108
##        [,56] [,57] [,58] [,59] [,60] [,61]
## [1,]    106   105   105   104   104   103
## [2,]    106   106   105   105   104   104
## [3,]    107   106   106   105   105   104
## [4,]    107   107   106   106   105   105
## [5,]    107   107   107   106   106   105
## [6,]    108   108   107   107   106   106
```

`volcano` contains topographic information for the Maunga Whau volcano. You can visualize it with:

```r
image(volcano) #a volcano indeed!
```



Now, let's pretend we want the average elevation for every column.

We can use the function `mean()` to calculate the average of the first column, remembering how to access a column in a matrix:

```
mean( volcano[,1] )

## [1] 110.5862
```

We could change 1 to 2, then 3, then 4... until 61, and run 61 R-commands, but that is horribly inefficient. That is where a for-loop may be useful. Instead of writing code for the column "1" or "2"..."61", we will write code for column "i", where "i" varies between 1 and 61: `mean( volcano[,i] )`.

Now, R doesn't know yet what "i" is, and would return an error message if you run this:

```
mean( volcano[,i] )

## Error in mean(volcano[, i]):  object 'i' not found
```

We need to include our code within a for-loop defining "i":

```
for (i in 1:61)
{
  ...
}
```

The above code can be read as: "Define a variable i that will take integer values between 1 and 61, and do whatever is withing curly brackets for each value of i. "

If we run this:

```
for (i in 1:61)
{
  mean( volcano[,i] )
}
```

it looks like nothing happened. Actually, R did compute all the averages, but we didn't ask R to print the results or store them somewhere, so the loop was useless.

We can print the results using the function `print()`:

```
for (i in 1:61)
{
  print(mean( volcano[,i] ))
}
```

```
## [1] 110.5862
## [1] 111.8276
## [1] 112.954
## [1] 114.1149
## [1] 115.1264
## [1] 116.1034
## [1] 117.1494
## [1] 118.069
## [1] 119.4483
## [1] 121.3218
## [1] 123.3448
## [1] 125.4368
## [1] 127.6207
## [1] 130.023
## [1] 132.6667
## [1] 134.9655
## [1] 137.1379
## [1] 139.1034
## [1] 141.1149
## [1] 143.2414
## [1] 145.2414
## [1] 147.1494
## [1] 148.8736
## [1] 150.023
## [1] 150.8391
## [1] 151.2529
## [1] 151.1034
## [1] 150.4253
## [1] 149.4023
## [1] 148.4023
## [1] 147.5402
## [1] 146.3908
## [1] 145.2644
## [1] 144.3103
## [1] 143.5517
## [1] 142.977
## [1] 142.4943
## [1] 141.7701
## [1] 141.2069
## [1] 140.4483
## [1] 139.4368
## [1] 138.092
## [1] 136.4483
## [1] 134.7126
## [1] 132.6092
## [1] 130.3218
## [1] 128.3793
## [1] 126.1724
## [1] 124.3103
## [1] 122.023
## [1] 119.4828
## [1] 116.9655
## [1] 114.8736
## [1] 112.8161
## [1] 110.9885
## [1] 109.0115
## [1] 107.3678
## [1] 105.8276
## [1] 104.6322
## [1] 103.8046
## [1] 103.1609
```

Even better, we can store the results in a vector that we create before the loop:

```
averages <- vector(length = 61)

for (i in 1:61)
{
  averages[i] <- mean( volcano[,i] )
}

averages

##  [1] 110.5862 111.8276 112.9540 114.1149 115.1264 116.1034
##  [7] 117.1494 118.0690 119.4483 121.3218 123.3448 125.4368
## [13] 127.6207 130.0230 132.6667 134.9655 137.1379 139.1034
## [19] 141.1149 143.2414 145.2414 147.1494 148.8736 150.0230
## [25] 150.8391 151.2529 151.1034 150.4253 149.4023 148.4023
## [31] 147.5402 146.3908 145.2644 144.3103 143.5517 142.9770
## [37] 142.4943 141.7701 141.2069 140.4483 139.4368 138.0920
## [43] 136.4483 134.7126 132.6092 130.3218 128.3793 126.1724
## [49] 124.3103 122.0230 119.4828 116.9655 114.8736 112.8161
## [55] 110.9885 109.0115 107.3678 105.8276 104.6322 103.8046
## [61] 103.1609
```

Finally, we can make the code more robust and reproducible by calculating the number of columns in the data instead of assuming it is always going to be 61. For that, we use the function `ncol()` that simply returns the number of columns in a data-frame or a matrix:

```
averages <- vector(length = ncol(volcano))

for (i in 1:ncol(volcano))
{
  averages[i] <- mean( volcano[,i] )
}
```

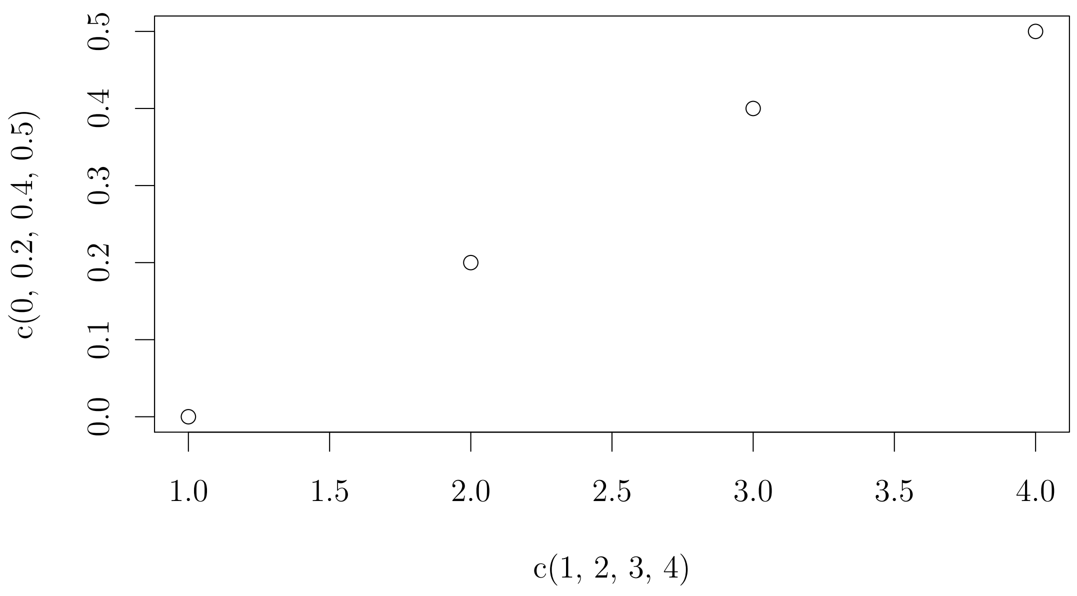
**Small exercise**
Make a for-loop that calculate the variance for each row of volcano. You will probably want to use the function `var()` and the function `nrow()`.

NB: here you do not really need a for loop. R has shorter, more efficient alternatives (we will talk about them later), but I believe it is crucial to master for loops which are much more flexible and transparent than alternatives.

## 4 Simple Graphics

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

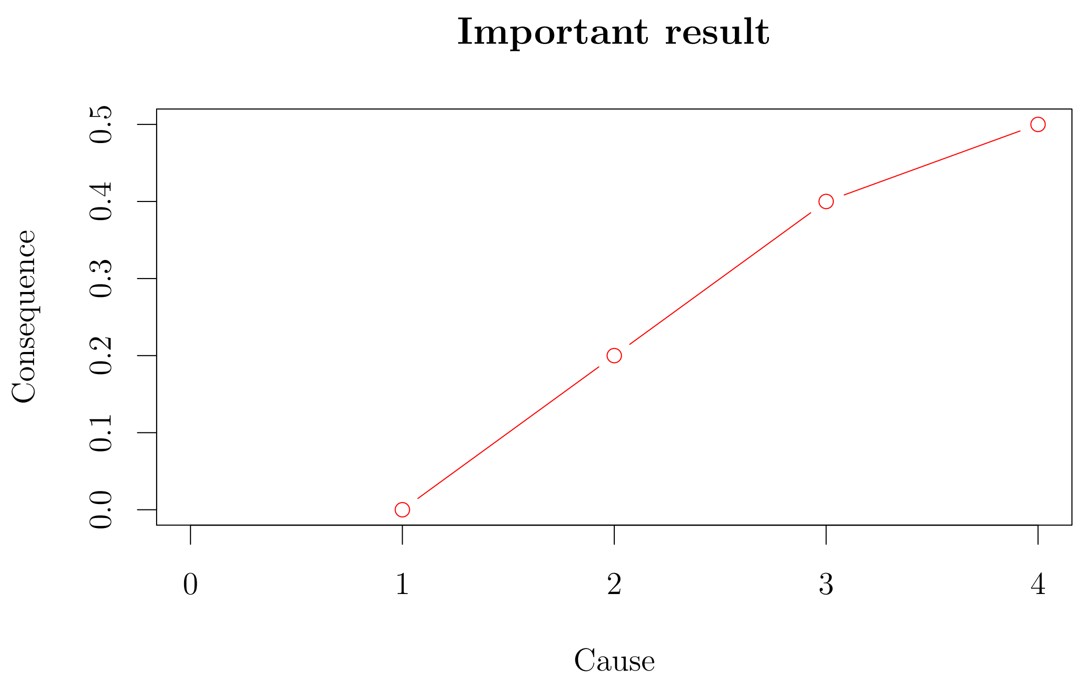
```
plot(x = c(1,2,3,4), y=c(0,0.2,0.4,0.5))
```



(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. You may want to check the help page for plot, type `?plot` (or search plot in the RStudio Help tab).