R - Basics

Gildas Taliah

Contents

Preliminaries

In this project we present the basics of R. We will have a look at the syntax, some basic objects and functions of R. The great advantage of R is that it is free, easy to understand/use and very powerful (on the other hand it is slower than some other prog lang like C++, Matlab, python etc.). Therefore it is used by many people especially in academia all over the world and its user base is still growing.

Its huge user base is a great advantage, should not surprise that for many problems we might think of or encounter later on, a solution already exists somewhere (Google, GitHub, GPT etc is our friend!).

To use the programs of others we will need to install certain so-called packages. For this purpose R offers the function **install.packages("???")** [Note the quotes!]. To be able to use these packages in our session we have to use the command **library(???)**. The installation of a certain package is only needed once, then we call to the library every time we restart R!

To make things work we have to first install the four packages and activate them. To do so press the small green run button in the grey box below:

```r
# I put a '#' cause I already have
#install.packages("rmarkdown")
#install.packages("formatR")
#install.packages("caTools")
#install.packages("knitr")
library(rmarkdown)
library(formatR)
library(caTools)
library(knitr)
```

Especially in the beginning a very useful function is the help. We can open it in R by typing **help(???)**.

```r
# Call the help page of the command help
help(help)
```

Note that the hash tag (#) can be used in R scripts to make comments. Comments are just for the reader of the code and will not be executed by R. A good program should include comments where the code is explained.

Numbers, vectors and basic functions

Of course R could be used as an (advanced) pocket calculator by using the operator **"+"**, **"-"**, **"·"** or **"/"**. We can store the results in variables. The name of the variable is always on the left, the statement you would like to save on the right. We use **"="** or **"<-"** to assign something to a variable name. Note that there are certain rules for variable names: It has to begin with a letter, special characters are (mostly) not allowed and certain names as *TRUE* or *FALSE* are reserved for R commands or functions. If we use the same name for a variable twice we overwrite the content of the former.

Calculate the square of 5 and save the result in the variable x. What is x? Now assign the value 7 to x. What is x now?

```r
x = 5^2
x
```

```
## [1] 25
```

```r
x = 7
x
```

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

See that!

We can have a look at the content of any variable by calling the name - as we did for x. Very often in practice we will deal with (large) data sets. The most basic R object to save data - which is called data structure - is a vector. A vector is easy to construct via the function **c(???)**.

```r
vec = c(3, 5, 7, 9, 11, 13, 15)
vec
```

```
## [1]  3  5  7  9 11 13 15
```

Sometimes this takes time to type out therefore another useful function to create a vector based on a certain pattern is **seq(???)**. Any function in R has a name - e.g. **"seq"** - and round brackets **"()"**, pretty much like in maths and stats. The *???* are called arguments. If you have a look at the help page of the function **seq()**

```
help(seq)
```

We notice that this is structured in different parts which are very typical for (most) help pages of functions. First a description what the function is intended to do, the explanation of the syntax (Usage) followed by a list of potential arguments and at the very end you find very often some examples to illustrate the usage.

Try to create the same vector as in the previous part using the function **seq()**.

```
vec2 = seq(3, 15, 2)
vec2
```

```
## [1]  3  5  7  9 11 13 15
```

Create a sequence of integers starting at 1 ending at 100 by specifying all arguments with names.

```
s = seq(from = 1, to = 100, length.out = 10)
s
```

```
##  [1]   1  12  23  34  45  56  67  78  89 100
```

```
s = seq(from = 1, to = 100, by = 1)
s
```

```
##   [1]   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18
##  [19]  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36
##  [37]  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54
##  [55]  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72
##  [73]  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90
##  [91]  91  92  93  94  95  96  97  98  99 100
```

A very important difference to some other programming languages is that R will manipulate (numeric) vectors element wise. For example the scalar product of two vectors a and b of the same length $<a,b>$ is typically ab but not in R!

Construct two numeric vectors a and b with the same number of elements (n = 4) and multiply them

```
a = c(1,2,3,4, 4.5)
b = c(5,6,7,8, 8.5)
a*b           # elementwise
```

```
## [1]  5.00 12.00 21.00 32.00 38.25
```

```
a%*%b         # scalar product
```

```
##        [,1]
## [1,] 108.25
```

In some applications this is very useful in other applications you really have to think hard how to translate element wise manipulations of a vector to your desired outcome. If we multiply (divide, . . .) vectors of different length the shorter vector will be recycled automatically. Note that if the length of the shorter is an exact multiple of the larger vector you will not even get an error message, otherwise only a warning.

Sometimes you may be interested in certain subsets of a given vector. In R this is possible by specifying the name and the index number of the elements you are interested in, e.g. *vec[1:3]* which extracts the first three elements of the vector vec.

How can you obtain the elements 2 to 5 and 7 of the vector vec?

```
vec[1:3]
```

```
## [1] 3 5 7
```

```
vec[c(2:5,7)]
```

```
## [1]  5  7  9 11 15
```

Notice that if you would like to obtain elements of a vector (or later any data structure) you have to use rectangular brackets **"[]"**, if you would like to call a function always round brackets **"()"**. You can exclude certain elements of a vector with a minus, e.g. *vec[-(2:5)]*.

We proceed to solve related exercises.

1.1 Create a vector "prob" with 50 elements which has the following form $prob = \begin{pmatrix} 1 & 4 & 9 & 16 & \cdots \end{pmatrix}$.

```
prob = seq(from=1, to=50, by=1)
prob = prob^2
print(prob)
```

```
##  [1]    1    4    9   16   25   36   49   64   81  100  121  144  169  196  225
## [16]  256  289  324  361  400  441  484  529  576  625  676  729  784  841  900
## [31]  961 1024 1089 1156 1225 1296 1369 1444 1521 1600 1681 1764 1849 1936 2025
## [46] 2116 2209 2304 2401 2500
```

1.2 Calculate the scalar product for the vector prob with itself.

```
prob_scalar = prob%*%prob
print(prob_scalar)
```

```
##          [,1]
## [1,] 65666665
```

1.3 Take the square root of the vector prob and add this to this vector. How can you obtain the same result? Test whether the difference is indeed 0.

```
prob_sqrt = sqrt(prob)
prob_transform1 = prob + prob_sqrt
prob_transform2 = prob + seq(from=1, to=50, by=1)
identical(prob_transform1, prob_transform2)
```

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

1.4 Construct another vector "alphabet" consisting of characters. To this end use the build in variable letters. Can you add the vectors alphabet and prob? Why or why not? Set up a logical vector "logi" and use it instead of alphabet. What can you conclude?

```r
alphabet = letters
print(alphabet)
```

```
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s"
## [20] "t" "u" "v" "w" "x" "y" "z"
```

```r
letters
```

```
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s"
## [20] "t" "u" "v" "w" "x" "y" "z"
```

```r
# Test prob + alphabet
# prob + alphabet # Afterwards, delete this line or turn it into a comment to continue
logi = rep(c(TRUE, FALSE), 25)
prob + logi
```

```
##  [1]    2    4   10   16   26   36   50   64   82  100  122  144  170  196  226
## [16]  256  290  324  362  400  442  484  530  576  626  676  730  784  842  900
## [31]  962 1024 1090 1156 1226 1296 1370 1444 1522 1600 1682 1764 1850 1936 2026
## [46] 2116 2210 2304 2402 2500
```

We cannot add a vector of whose elements are character type ones to a numeric vector. Additionally, the dimensions would not match. Assuming that logi has the same length as prob, TRUE is interpreted as 1 while FALSE is treated as 0.

1.5 Find the minimal and maximal value as well as the sum and the range of the vector zeta. In addition use the function *summary()*.

```r
zeta = rnorm(1000)
zeta_min = min(zeta)
zeta_max = max(zeta)
zeta_sum = sum(zeta)
zeta_mean = mean(zeta)
zeta_range = range(zeta)
print(paste("zeta_min: ", zeta_min))
```

```
## [1] "zeta_min:  -3.47656210521688"
```

```r
print(paste("zeta_max: ", zeta_max))
```

```
## [1] "zeta_max:  3.1545786964925"
```

```r
print(paste("zeta_sum: ", zeta_sum))
```

```
## [1] "zeta_sum:  -26.3023864469525"
```

```r
print(paste("zeta_mean: ", zeta_mean))
```

```
## [1] "zeta_mean:  -0.0263023864469525"
```

```r
print(paste("zeta_range: min =", zeta_range[1], ", max =", zeta_range[2], ", difference = ", zeta_range
```

```
## [1] "zeta_range: min = -3.47656210521688 , max = 3.1545786964925 , difference =  6.63114080170938"
```

```r
summary(zeta)
```

```
##      Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
## -3.476562 -0.690958 -0.004754 -0.026302  0.635087  3.154579
```

1.6 Now extract the minimal and maximal values of the vector zeta and save these in the variables min and max. Additionally count how many times you obtained a value smaller than 0. How many data points are in the interval from -1 to 1?

```r
min = zeta_min
max = zeta_max
smaller_zero = sum(zeta < 0)
minus_one_to_one = sum(zeta >= -1 & zeta <= 1)
print(paste("No. of elements < 0: ", smaller_zero))
```

```
## [1] "No. of elements < 0:  502"
```

```r
print(paste("No. of elements -1 < zeta < 1: ", minus_one_to_one))
```

```
## [1] "No. of elements -1 < zeta < 1:  683"
```

1.7 Construct a numeric variable "threshold" with any number you like. Exclude all elements of the vector zeta that are greater than your threshold value. Repeat this procedure with different threshold values.

```r
zeta_sort = sort(zeta)
zeta_threshold = zeta[zeta >=  -1 & zeta <= 1]
range(zeta_threshold)
```

```
## [1] -0.9975327  0.9964679
```

```r
length(zeta_threshold)
```

```
## [1] 683
```

1.8 Use the vectors a1 and b1 to verify both DeMorgan's laws. Do not forget to test the result!

```r
a1 = rbinom(n = 10000, size = 1, prob = 0.5)
b1 = rbinom(n = 10000, size = 1, prob = 0.5)
all(!(a1|b1)==((!a1)&(!b1)))
```

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

```r
all(!(a1&b1)==((!a1)|(!b1)))
```

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

It checks if the statement "not (a1 or b1)" is logically equivalent to "not a1 and not b1" for all observations. Second line checks if the statement "not (a1 and b1)" is logically equivalent to "not a1 or not b1" for all observations.

Loops

Whenever something has to be repetitively executed the most basic procedure to solve such a task is a so called *loop*. Loops are just designed to repeatedly execute (similar) expressions. In R two basic types of loops exist: 1. the so called *"for"* loop and 2. the *"while"* loop (to be more precise there exist a third option: repeat loop, but this behaves pretty much like the while loop). The for loop is easier to handle, therefore we begin with that. You might have a look at the help page. The syntax for a for loop is quiet similar to that for 'if' statements.

Suppose that we want to calculate the sum of the variable s from earlier without using the *sum()* function.

```r
sum = 0
for(i in 1:length(s)){
  sum=sum+s[i]
}
sum
```

```
## [1] 5050
```

```r
sum(s)
```

```
## [1] 5050
```

Having a look at the list of variables, the variable sum has the value 5050 which is indeed the sum of all elements from 1 to 100 (we used use *sum()* to verify this result).

The second type of loops are the so called while loops. The syntax is similar to the for loop with a main difference. The statement within the **{}**-brackets is executed as long as the condition in the round brackets **()** is true! This yields a major problem because the condition may be always true, therefore the while loop will never stop (infinite loop)!!!

Again calculate the sum of s:

```r
i = 1
sum = 0
while(i<=length(s)){
  sum = sum + s[i]
  i = i+1
}
sum
```

```
## [1] 5050
```

To ensure that the while loop will stop we have to increase the index i within the loop.

Loops are relatively easy to use, unfortunately they are relatively slow. If we are running for example larger simulations it might pay off to use *vectorized commands or predefined functions* instead of loops to save a lot of computation time.

Writing functions

R is a functional language which means almost everything is a function (besides data of course). We are recommended to use this mind-set, too. Thus if we have to implement a new procedure for a given task, analyzing data or running a simulation we define our own function that does exactly this. Typically this is not one big function but a function that makes use of other functions (which we have also written for this purpose). The goal for almost any larger project is to disaggregate this project into smaller parts that can be handled separately (maybe with some dependency between those) and write a function for each sub-part.

Now let's write our first function. We need several things to define a function. A name, some arguments or variables that the function may use and of course a procedure. The syntax is as follows:

```r
name = function(argument1, argument2, argument3 = a,...){
  procedure
  return(result)
}
```

Note that the third argument has a default value, i.e. a value which is used whenever nothing is specified for this argument by the user.

Below we ttry writing a function that calculates the mean of any numerical vector. We use the functions **sum()** and **length()** for this purpose. Test this function with the vector a you defined earlier.

```r
our_mean = function(x){
  result = sum(x)/length(x)
  return(result)
}

our_mean(s)
```

```
## [1] 50.5
```

```r
identical(our_mean(s), mean(s))
```

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

Now let us write some (custom/predefined) functions.

2.1 Write a procedure to obtain the smallest value in the vector zeta. Use a for- or while-loop. Compare the procedure to the function *min()*.

```r
# Test vector
zeta = rnorm(1000)

# Create a function
min_vec = function(x){
  # Inputs
  #   x: some vector
  # Outputs
  #   res: Minimum of vector x

  n = length(x)
  res = x[1]
  for (i in 2:n){
    if (x[i] < res){
      res = x[i]
    }
  }
  return(res)
}
# Test the function and compare the result to min(zeta)
min_vec(zeta)
```

```
## [1] -2.99847
```

```
identical(min_vec(zeta), min(zeta))
```

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

2.2 Write a function that creates an $n \times n$ identity matrix using for- or while-loops. Let n be the only argument of this function.

```
identity = function(n){
  # Inputs:
  #    n: Dimension of the square identity matrix
  # Outputs:
  #    nxn identity matrix

  iden = matrix(0, nrow=n, ncol=n)
  for (i in 1:n){
    iden[i, i]=1
  }
  return(iden)
}
identity(3)
```

```
##      [,1] [,2] [,3]
## [1,]    1    0    0
## [2,]    0    1    0
## [3,]    0    0    1
```

```
identity(10)
```

```
##       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
##  [1,]    1    0    0    0    0    0    0    0    0     0
##  [2,]    0    1    0    0    0    0    0    0    0     0
##  [3,]    0    0    1    0    0    0    0    0    0     0
##  [4,]    0    0    0    1    0    0    0    0    0     0
##  [5,]    0    0    0    0    1    0    0    0    0     0
##  [6,]    0    0    0    0    0    1    0    0    0     0
##  [7,]    0    0    0    0    0    0    1    0    0     0
##  [8,]    0    0    0    0    0    0    0    1    0     0
##  [9,]    0    0    0    0    0    0    0    0    1     0
## [10,]    0    0    0    0    0    0    0    0    0     1
```

2.3 Write a function that creates an $n \times n$ matrix of the following form $\begin{pmatrix} 1 & 2 & 3 & \cdots & n \\ 2 & 2 & 3 & \cdots & n \\ 3 & 3 & 3 & \cdots & n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ n & n & n & \cdots & n \end{pmatrix}$. Again let n be the only argument of this function.

```r
mat = function(n){
  # Inputs:
  #    n: Dimension of the matrix
  # Output
  #    mat_res: nxn matrix

  mat_res = matrix(0, nrow=n, ncol=n)
  for (i in 1:n){
    for (j in 1:n){
      if (i >= j){
        mat_res[i, j] = i
      }
      else{
        mat_res[i, j] = j
      }
    }
  }
  return(mat_res)
}

# mat()
mat(5)
```

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

```r
mat(10)
```

```
##       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
##  [1,]    1    2    3    4    5    6    7    8    9    10
##  [2,]    2    2    3    4    5    6    7    8    9    10
##  [3,]    3    3    3    4    5    6    7    8    9    10
##  [4,]    4    4    4    4    5    6    7    8    9    10
##  [5,]    5    5    5    5    5    6    7    8    9    10
##  [6,]    6    6    6    6    6    6    7    8    9    10
##  [7,]    7    7    7    7    7    7    7    8    9    10
##  [8,]    8    8    8    8    8    8    8    8    9    10
##  [9,]    9    9    9    9    9    9    9    9    9    10
## [10,]   10   10   10   10   10   10   10   10   10    10
```

Working with a data set

Task: Load the package **'MASS'** and the dataset named **'Boston'**. Save this set in the variable 'Bost'.

```r
library(MASS)
Bost = Boston
```

Hint: In the case we want to work with the data set we already had in your own PC, we look at the 'Environment' tab from the top-right window, then click to 'Import Dataset' and follow the instructions.

Task: Have a look at the Boston dataset

```r
View(Bost)
```

Task: Get the variable names of all variables in the Boston dataset

```r
names(Bost)
```

```
##  [1] "crim"    "zn"      "indus"   "chas"    "nox"     "rm"      "age"
##  [8] "dis"     "rad"     "tax"     "ptratio" "black"   "lstat"   "medv"
```

Task: Compute descriptive statistics and frequencies of the Boston dataset (meaning compute the minimum and maximum values, mode, median, etc for each variable in the data set)

```r
summary(Bost)
```

```
##       crim                zn             indus            chas
##  Min.   : 0.00632   Min.   :  0.00   Min.   : 0.46   Min.   :0.00000
##  1st Qu.: 0.08205   1st Qu.:  0.00   1st Qu.: 5.19   1st Qu.:0.00000
##  Median : 0.25651   Median :  0.00   Median : 9.69   Median :0.00000
##  Mean   : 3.61352   Mean   : 11.36   Mean   :11.14   Mean   :0.06917
##  3rd Qu.: 3.67708   3rd Qu.: 12.50   3rd Qu.:18.10   3rd Qu.:0.00000
##  Max.   :88.97620   Max.   :100.00   Max.   :27.74   Max.   :1.00000
##       nox               rm             age              dis
##  Min.   :0.3850   Min.   :3.561   Min.   :  2.90   Min.   : 1.130
##  1st Qu.:0.4490   1st Qu.:5.886   1st Qu.: 45.02   1st Qu.: 2.100
##  Median :0.5380   Median :6.208   Median : 77.50   Median : 3.207
##  Mean   :0.5547   Mean   :6.285   Mean   : 68.57   Mean   : 3.795
##  3rd Qu.:0.6240   3rd Qu.:6.623   3rd Qu.: 94.08   3rd Qu.: 5.188
##  Max.   :0.8710   Max.   :8.780   Max.   :100.00   Max.   :12.127
##       rad              tax           ptratio          black
##  Min.   : 1.000   Min.   :187.0   Min.   :12.60   Min.   :  0.32
##  1st Qu.: 4.000   1st Qu.:279.0   1st Qu.:17.40   1st Qu.:375.38
##  Median : 5.000   Median :330.0   Median :19.05   Median :391.44
##  Mean   : 9.549   Mean   :408.2   Mean   :18.46   Mean   :356.67
##  3rd Qu.:24.000   3rd Qu.:666.0   3rd Qu.:20.20   3rd Qu.:396.23
##  Max.   :24.000   Max.   :711.0   Max.   :22.00   Max.   :396.90
##      lstat            medv
##  Min.   : 1.73   Min.   : 5.00
##  1st Qu.: 6.95   1st Qu.:17.02
##  Median :11.36   Median :21.20
##  Mean   :12.65   Mean   :22.53
##  3rd Qu.:16.95   3rd Qu.:25.00
##  Max.   :37.97   Max.   :50.00
```

We have printed the statistical properties of each variable in the data. Displaying Lowest (min) to highest(max) possible values possible from each variable. 1st (3rd) quantile represent the value greater than or equal to 25% (75%) of the variablle. When the mean is greater than (less than) the median the data is positively (negatively) skewed and the underlying distribution is not normally distributed.

Task: Extract the third variable in the Boston data set.

```
Bost[,3]
```

```
##   [1]  2.31  7.07  7.07  2.18  2.18  2.18  7.87  7.87  7.87  7.87  7.87  7.87
##  [13]  7.87  8.14  8.14  8.14  8.14  8.14  8.14  8.14  8.14  8.14  8.14  8.14
##  [25]  8.14  8.14  8.14  8.14  8.14  8.14  8.14  8.14  8.14  8.14  8.14  5.96
##  [37]  5.96  5.96  5.96  2.95  2.95  6.91  6.91  6.91  6.91  6.91  6.91  6.91
##  [49]  6.91  6.91  5.64  5.64  5.64  5.64  4.00  1.22  0.74  1.32  5.13  5.13
##  [61]  5.13  5.13  5.13  5.13  1.38  3.37  3.37  6.07  6.07  6.07 10.81 10.81
##  [73] 10.81 10.81 12.83 12.83 12.83 12.83 12.83 12.83  4.86  4.86  4.86  4.86
##  [85]  4.49  4.49  4.49  4.49  3.41  3.41  3.41  3.41 15.04 15.04 15.04  2.89
##  [97]  2.89  2.89  2.89  2.89  8.56  8.56  8.56  8.56  8.56  8.56  8.56  8.56
## [109]  8.56  8.56  8.56 10.01 10.01 10.01 10.01 10.01 10.01 10.01 10.01 10.01
## [121] 25.65 25.65 25.65 25.65 25.65 25.65 25.65 21.89 21.89 21.89 21.89 21.89
## [133] 21.89 21.89 21.89 21.89 21.89 21.89 21.89 21.89 21.89 21.89 19.58 19.58
## [145] 19.58 19.58 19.58 19.58 19.58 19.58 19.58 19.58 19.58 19.58 19.58 19.58
## [157] 19.58 19.58 19.58 19.58 19.58 19.58 19.58 19.58 19.58 19.58 19.58 19.58
## [169] 19.58 19.58 19.58 19.58  4.05  4.05  4.05  4.05  4.05  4.05  4.05  2.46
## [181]  2.46  2.46  2.46  2.46  2.46  2.46  2.46  3.44  3.44  3.44  3.44  3.44
## [193]  3.44  2.93  2.93  0.46  1.52  1.52  1.52  1.47  1.47  2.03  2.03  2.68
## [205]  2.68 10.59 10.59 10.59 10.59 10.59 10.59 10.59 10.59 10.59 10.59 10.59
## [217] 13.89 13.89 13.89 13.89  6.20  6.20  6.20  6.20  6.20  6.20  6.20  6.20
## [229]  6.20  6.20  6.20  6.20  6.20  6.20  6.20  6.20  6.20  6.20  4.93  4.93
## [241]  4.93  4.93  4.93  4.93  5.86  5.86  5.86  5.86  5.86  5.86  5.86  5.86
## [253]  5.86  5.86  3.64  3.64  3.75  3.97  3.97  3.97  3.97  3.97  3.97  3.97
## [265]  3.97  3.97  3.97  3.97  3.97  6.96  6.96  6.96  6.96  6.96  6.41  6.41
## [277]  6.41  6.41  6.41  3.33  3.33  3.33  3.33  1.21  2.97  2.25  1.76  5.32
## [289]  5.32  5.32  4.95  4.95  4.95 13.92 13.92 13.92 13.92 13.92  2.24  2.24
## [301]  2.24  6.09  6.09  6.09  2.18  2.18  2.18  2.18  9.90  9.90  9.90  9.90
## [313]  9.90  9.90  9.90  9.90  9.90  9.90  9.90  9.90  7.38  7.38  7.38  7.38
## [325]  7.38  7.38  7.38  7.38  3.24  3.24  3.24  6.06  6.06  5.19  5.19  5.19
## [337]  5.19  5.19  5.19  5.19  5.19  1.52  1.89  3.78  3.78  4.39  4.39  4.15
## [349]  2.01  1.25  1.25  1.69  1.69  2.02  1.91  1.91 18.10 18.10 18.10 18.10
## [361] 18.10 18.10 18.10 18.10 18.10 18.10 18.10 18.10 18.10 18.10 18.10 18.10
## [373] 18.10 18.10 18.10 18.10 18.10 18.10 18.10 18.10 18.10 18.10 18.10 18.10
## [385] 18.10 18.10 18.10 18.10 18.10 18.10 18.10 18.10 18.10 18.10 18.10 18.10
## [397] 18.10 18.10 18.10 18.10 18.10 18.10 18.10 18.10 18.10 18.10 18.10 18.10
## [409] 18.10 18.10 18.10 18.10 18.10 18.10 18.10 18.10 18.10 18.10 18.10 18.10
## [421] 18.10 18.10 18.10 18.10 18.10 18.10 18.10 18.10 18.10 18.10 18.10 18.10
## [433] 18.10 18.10 18.10 18.10 18.10 18.10 18.10 18.10 18.10 18.10 18.10 18.10
## [445] 18.10 18.10 18.10 18.10 18.10 18.10 18.10 18.10 18.10 18.10 18.10 18.10
## [457] 18.10 18.10 18.10 18.10 18.10 18.10 18.10 18.10 18.10 18.10 18.10 18.10
## [469] 18.10 18.10 18.10 18.10 18.10 18.10 18.10 18.10 18.10 18.10 18.10 18.10
## [481] 18.10 18.10 18.10 18.10 18.10 18.10 18.10 18.10 27.74 27.74 27.74 27.74
## [493] 27.74  9.69  9.69  9.69  9.69  9.69  9.69  9.69  9.69 11.93 11.93 11.93
## [505] 11.93 11.93
```

Task: Take all values from the first to the 90th observations of the 7th variable in the dataset

```
Bost[1:90,7]
```

```
##  [1]  65.2  78.9  61.1  45.8  54.2  58.7  66.6  96.1 100.0  85.9  94.3  82.9
## [13]  39.0  61.8  84.5  56.5  29.3  81.7  36.6  69.5  98.1  89.2  91.7 100.0
## [25]  94.1  85.7  90.3  88.8  94.4  87.3  94.1 100.0  82.0  95.0  96.9  68.2
## [37]  61.4  41.5  30.2  21.8  15.8   2.9   6.6   6.5  40.0  33.8  33.3  85.5
## [49]  95.3  62.0  45.7  63.0  21.1  21.4  47.6  21.9  35.7  40.5  29.2  47.2
## [61]  66.2  93.4  67.8  43.4  59.5  17.8  31.1  21.4  36.8  33.0   6.6  17.5
## [73]   7.8   6.2   6.0  45.0  74.5  45.8  53.7  36.6  33.5  70.4  32.2  46.7
## [85]  48.0  56.1  45.1  56.8  86.3  63.1
```

Task: Create a new dataset named 'sub_Bost_1' including variables 1 to 4 and the last three variables

```
sub_Bost_1 = Bost[,c(1:4,12:14)]
```

Hint: The task asks us to extract the data set into the sub data set including variables number 1, 2, 3, 4, 12, 13 and 14. So, we have to indicate these indices to R.

Task: Create a new data set name 'sub_Bost_2' including all variables of the data set Boston except the third variable

```
sub_Bost_2 = Bost[,-3]
```

Task: Now let's find the mean, variance, range, 25% and 75% quantile and draw the box plot of the first variable in the Boston dataset.

```
mean(Bost[,1])
```

```
## [1] 3.613524
```

```
var(Bost[,1])
```

```
## [1] 73.98658
```

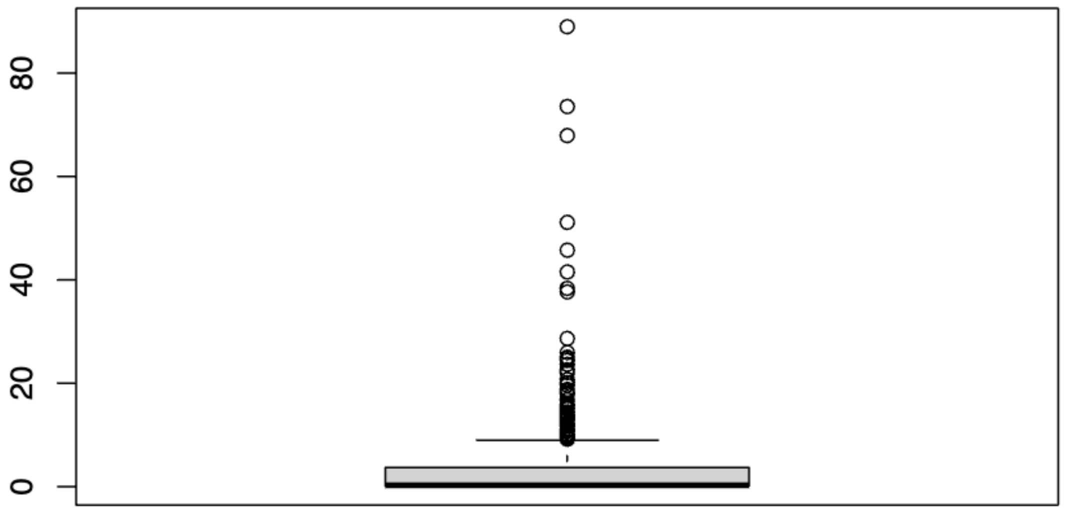
```
range(Bost[,1])
```

```
## [1]  0.00632 88.97620
```

```
quantile(Bost[,1])
```

```
##        0%       25%       50%       75%      100%
##  0.006320  0.082045  0.256510  3.677083 88.976200
```

```
boxplot(Bost[,1])
```



Boxplot depicts that the data is positively skewed. Task: Repeat the procedure by using the name of the variable 'crim' instead.

```r
mean(Bost$crim)
```

```
## [1] 3.613524
```

```r
var(Bost$crim)
```

```
## [1] 73.98658
```

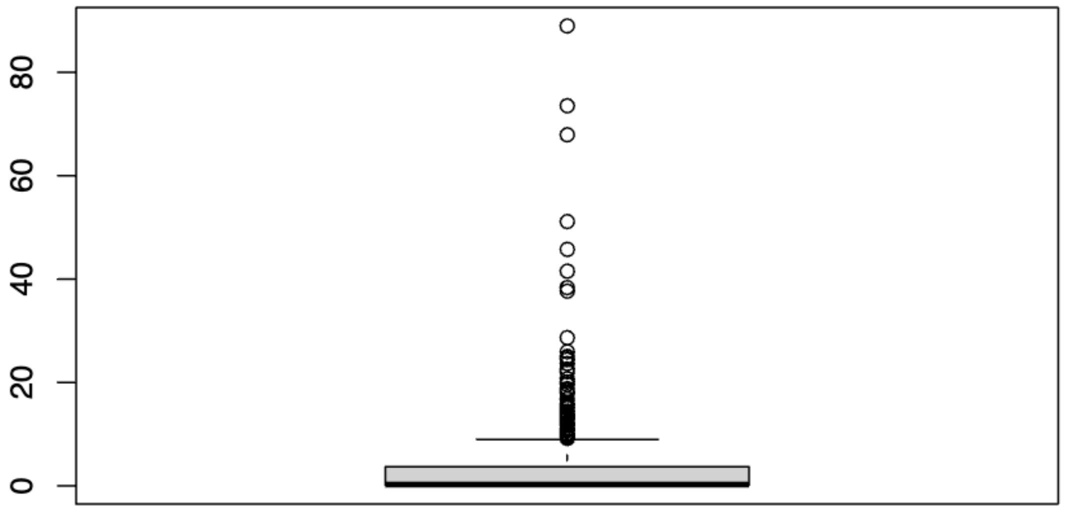
```r
range(Bost$crim)
```

```
## [1]  0.00632 88.97620
```

```r
quantile(Bost$crim)
```

```
##        0%       25%       50%       75%      100% 
##  0.006320  0.082045  0.256510  3.677083 88.976200
```

```r
boxplot(Bost$crim)
```



Task: We suspect that the variables 'tax' and 'indus' are correlated. We display the correlation coefficient matrix to find out whether this may be the case.

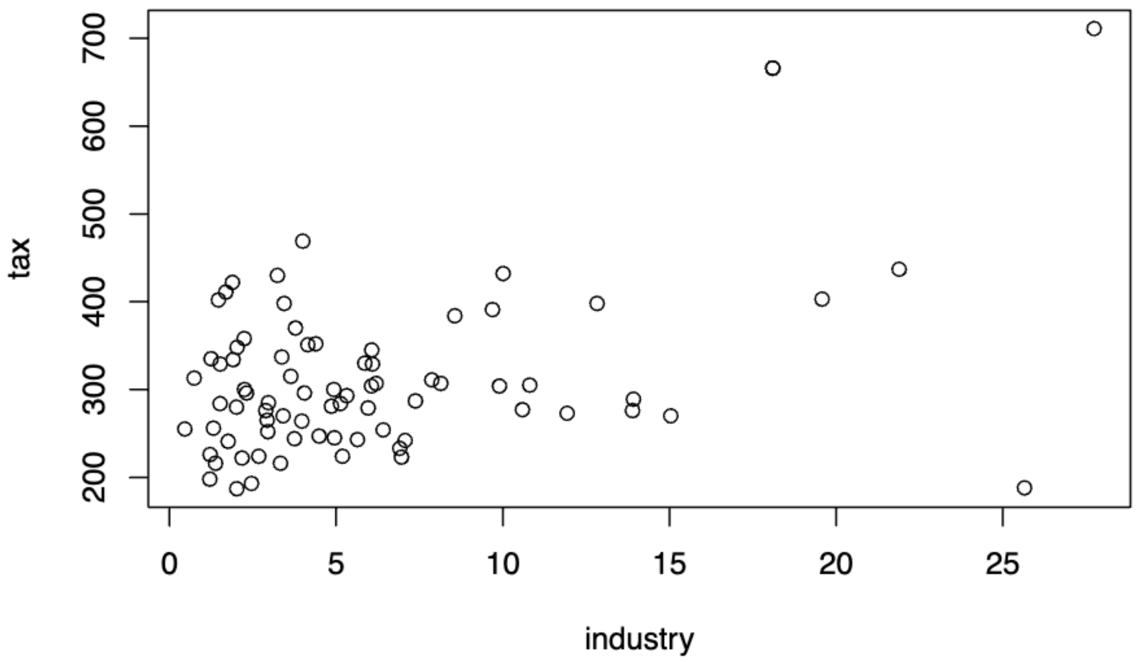
```
cor(Bost$tax, Bost$indus)
```

```
## [1] 0.7207602
```

Correlation as we all know ranges from -1 to 1. The result 0.72 would indicate a strong positive relationship between 'tax' and 'indus'.

Task: Let us draw a scatter plot to show graphically this relationship, label the x-axis by 'industry', the y-axis by 'tax'.

```
plot(Bost$indus, Bost$tax, xlab = 'industry', ylab = 'tax')
```



Task: Let us regress 'tax' on 'indus' by OLS with **lm(???)**. Assign the result to a new variable named 'result'.

```
result = lm(tax ~ indus, data = Bost)
```

Task: Summarize the result of the above regression

```
summary(result)
```

```
##
## Call:
## lm(formula = tax ~ indus, data = Bost)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -477.22  -67.91  -11.76  134.47  187.13
##
```

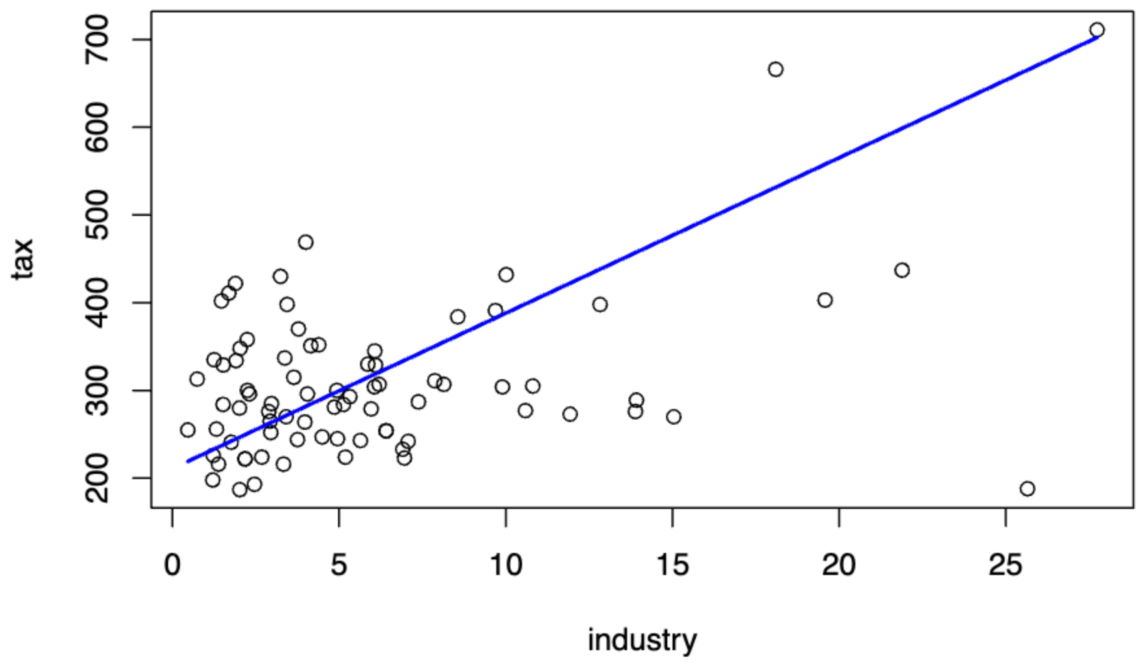
```
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)    
## (Intercept) 211.0405     9.9193   21.28   <2e-16 ***
## indus        17.7068     0.7585   23.34   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 116.9 on 504 degrees of freedom
## Multiple R-squared:  0.5195, Adjusted R-squared:  0.5185 
## F-statistic: 544.9 on 1 and 504 DF,  p-value: < 2.2e-16
```

The 'indus' coefficient (17.71) is quantitatively relevant and is statistically different from zero (p-value ~ 0). R-Squared value of 0.5195, means that approximately 51.95% of the variation in the dependent variable tax is explained by the independent variable indus, higher values indicate better fit.

Task: Plot the estimated line and the scatter plot in the same figure

Instruction: 1. Create x and y values of the regression curve 2. Draw the scatter plot between 'indus' and 'tax' 3. Draw a line plot between x and y in the same figure, use command 'lines'

```
x = seq(min(Bost[,"indus"]), max(Bost[,"indus"]), by = 0.05)
x = Bost[,"indus"]
y = result$coefficients[1] + result$coefficients[2]*x
plot(Bost[,'indus'], Bost[,'tax'], xlim = range(Bost[,'indus']), ylim = range(Bost[,'tax']), xlab = 'in
lines(x, y, lwd = 2, col = "blue")
```



Simulations

Simulations the process of creating models or algorithms that mimic real-world phenomena, involving generating synthetic data based on certain assumptions to study the behavior of a system or to make predictions about its future outcomes.

In Monte Carlo simulations, random variables are generated according to specified probability distributions, and these variables are used to model the behavior of the system being studied.

To carry out simulation we have to draw a sample of some random numbers. We will start with 10 values of the standard normal distribution. (By setting a **seed** we'll always get the same random numbers. This is useful for comparing results.)

```
set.seed(8)
x = rnorm(10, mean = 0, sd = 1)
x
```

```
##  [1] -0.08458607  0.84040013 -0.46348277 -0.55083500  0.73604043 -0.10788140
##  [7] -0.17028915 -1.08833171 -3.01105168 -0.59317433
```

Assume we are interested in the distribution of the t-test statistic. Set up a model - with 100 observations - which we can estimate via OLS. Define x as a uniform and u as a normal distributed random variable.

```
x = runif(100)
u = rnorm(100, sd = 2)
y = 1 + 2*x + u
```

Now we estimate this model via OLS:

```
estim = lm(y~x)
coeff = estim$coefficients
print(coeff)
```

```
## (Intercept)           x
##    1.192502    1.246928
```

```
covariance = vcov(estim)
```

Construct the t-test statistic for testing the second parameter in our model whether it is 2:

```
t = (coeff[2]-2)/sqrt(covariance[2,2])
t
```

```
##         x
## -0.926045
```

Unfortunately there is only one value for the test statistic. If we would like to obtain the empirical distribution, we have to repeat this procedure a couple of times. To do this we write a function that simulates and estimates the model and returns the test statistic (for the second parameter):

```
statistic = function(){
  x = runif(1000)
  u = rnorm(1000, sd = 2)
  y = 1 + 2*x + u
  estim = lm(y~x)
  coeff = estim$coefficients
  covariance = vcov(estim)
  t = (coeff[2]-2)/sqrt(covariance[2,2])
  return(t)
}
```

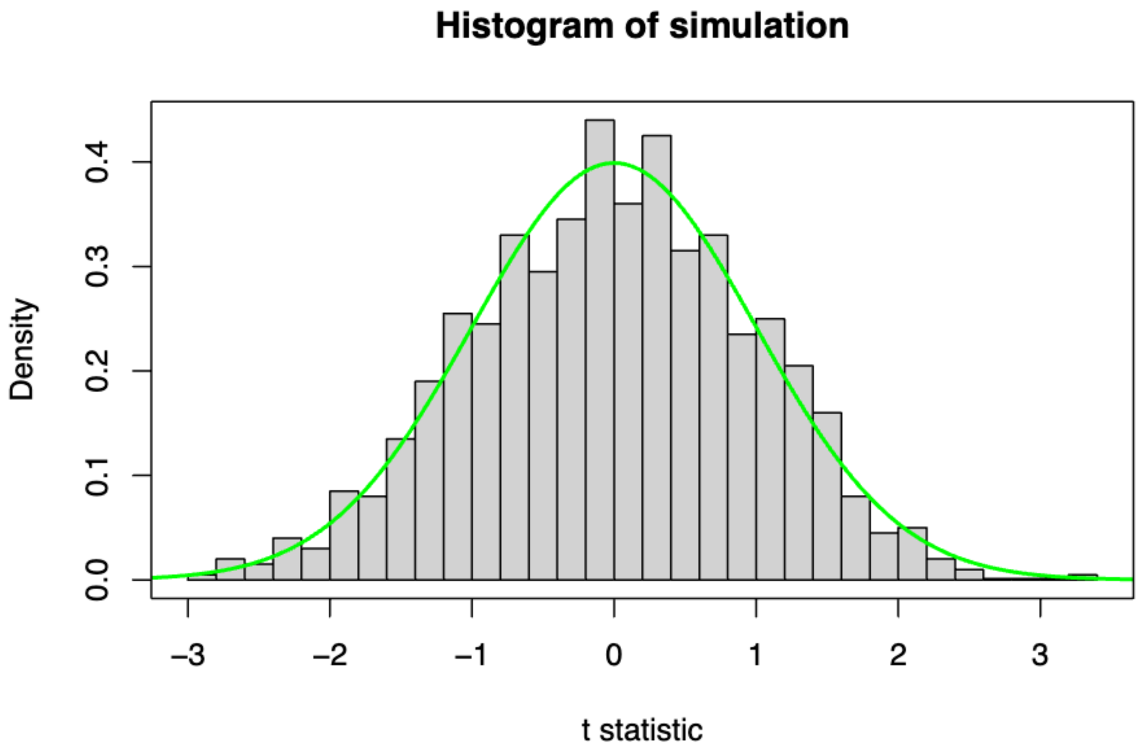
Now we can use this function for the construction of a sequence of test statistics

```r
set.seed(8)
N_MC = 1000 # number of replications in the simulation
simulation = replicate(N_MC, statistic())
```

We can use this sequence to plot the empirical distribution and compare this against the theoretical distribution:

```r
# empirical:
hist(simulation, breaks = 40, probability = T, xlab = "t statistic")

# theoretical:
x = seq(-5, 5, length.out = 10000)
y = dnorm(x, sd = 1)
lines(x, y, col = "green", lwd = 2)
box()
```



```r
summary(simulation)
```

```
##      Min.   1st Qu.    Median      Mean   3rd Qu.      Max. 
## -2.997518 -0.713101 -0.002318 -0.013372  0.694085  3.257010
```

```r
print('The End!')
```

```
## [1] "The End!"
```