Introduction to R

Gabriel S. Ferreira

2024-10-20

Table of contents

In this document you will find the necessary basis to start using R. This can be used as you would use a dictionary: it will help you with the meaning of "words" you are unfamiliar with or need to remember, and with the syntax necessary to understand R code. Remember: R is a language and, as such, you need to be able to read it in order to understand how to use it. This is what this document is for.

Getting help

There are many ways of getting help within R and outside of it. Because R is a free, open source, and very flexible environment, with applications in many different areas, it has a large active and engaged community of users. Getting help with R can be as simple as typing on your favorite search engine "*How to run a linear regression analysis in R?*". You will find many pages, blog posts, and forums dedicated to answering such questions. One of those, Stack Overflow gathers many questions made and answered by users, and can be very helpful. The official R page, www.r-project.org, has a very good documentation section, with manuals and frequently asked questions which are definitely worth checking and a question arises. Manuals and FAQs can also be found on the Comprehensive R Archive Network or simply CRAN.

In R, you can also find help for functions whose name you know. This is as simple as typing a question mark before the functions name:

```
?read.table
```

If you cannot remember the function's name, but do remember the subject, use the *help.search()* function:

```
help.search("data input")
help.search("csv")
```

We will see that packages are the way the base of R can be expanded by its users. If you know a function's name, but cannot remember its package, the function *find()* can help you

```
find("read.csv")
```

```
[1] "package:utils"
```

```
find("max")
```

```
[1] "package:base"
```

```
# apropos returns a character vector giving the names of all objects in the
## search list that match your enquiry
apropos("max")
```

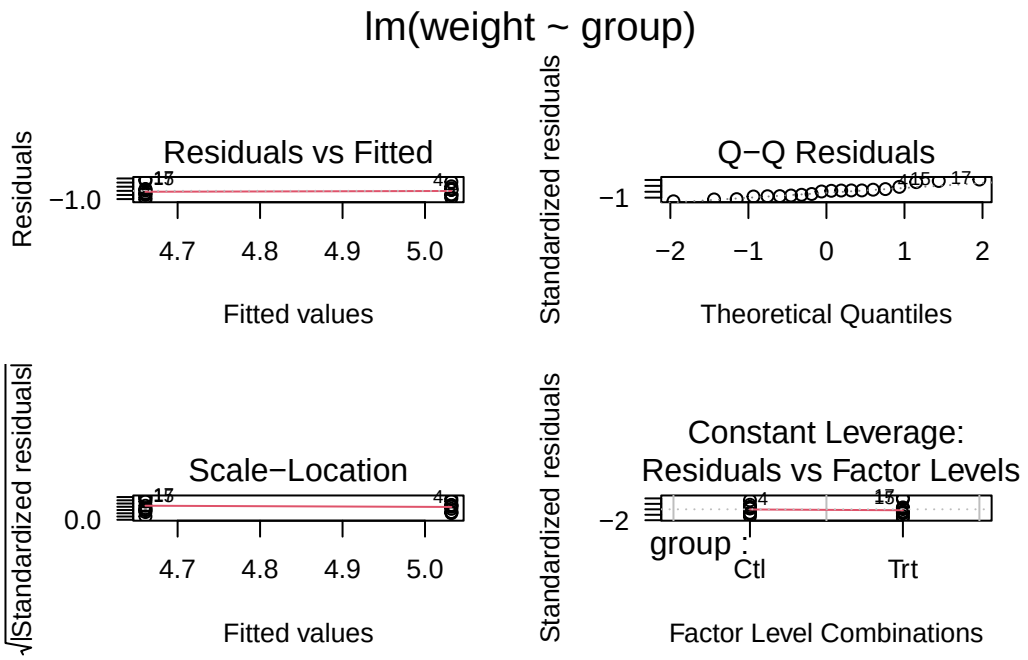
```
 [1] "cummax"       "max"          "max.col"      "mem.maxNSize" "mem.maxVSize"
 [6] "pmax"         "pmax.int"     "promax"       "varimax"      "which.max"
```

Two other functions, *example()* and *demo()*, and can show you how to use functions

```
example(lm)
```

```
lm> require(graphics)

lm> ## Annette Dobson (1990) "An Introduction to Generalized Linear Models".
lm> ## Page 9: Plant Weight Data.
lm> ctl <- c(4.17,5.58,5.18,6.11,4.50,4.61,5.17,4.53,5.33,5.14)

lm> trt <- c(4.81,4.17,4.41,3.59,5.87,3.83,6.03,4.89,4.32,4.69)
```

```
lm> group <- gl(2, 10, 20, labels = c("Ctl","Trt"))

lm> weight <- c(ctl, trt)

lm> lm.D9 <- lm(weight ~ group)

lm> lm.D90 <- lm(weight ~ group - 1) # omitting intercept

lm> ## No test:
lm> ##D anova(lm.D9)
lm> ##D summary(lm.D90)
lm> ## End(No test)
lm> opar <- par(mfrow = c(2,2), oma = c(0, 0, 1.1, 0))

lm> plot(lm.D9, las = 1)      # Residuals, Fitted, ...
```




```
lm> par(opar)

lm> ## Don't show:
lm> ## model frame :
```

```
lm> stopifnot(identical(lm(weight ~ group, method = "model.frame"),
lm+                     model.frame(lm.D9)))

lm> ## End(Don't show)
lm> ### less simple examples in "See Also" above
lm>
lm>
lm>
```

```
#demo(graphics)
```

And, of course, citing the correct version of R is important. You can check it using:

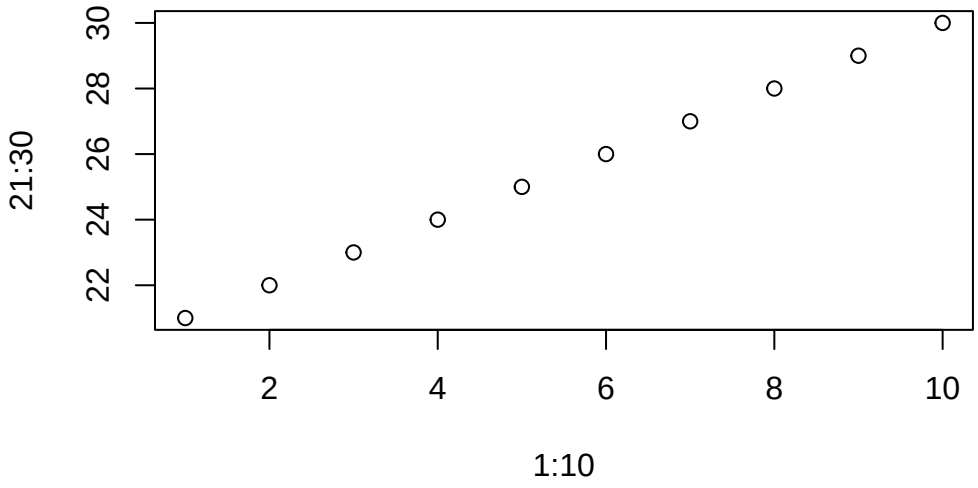
```
citation()
```

```
To cite R in publications use:

  R Core Team (2023). _R: A Language and Environment for Statistical
  Computing_. R Foundation for Statistical Computing, Vienna, Austria.
  <https://www.R-project.org/>.

A BibTeX entry for LaTeX users is

  @Manual{,
    title = {R: A Language and Environment for Statistical Computing},
    author = {{R Core Team}},
    organization = {R Foundation for Statistical Computing},
    address = {Vienna, Austria},
    year = {2023},
    url = {https://www.R-project.org/},
  }

We have invested a lot of time and effort in creating R, please cite it
when using it for data analysis. See also 'citation("pkgname")' for
citing R packages.
```

Basic syntax

Above you saw examples of how code and comments are written in R. Everything that follows a hash-tag symbol (#) is considered a comment and will not be read. This is very useful (and important!) to make your code self-explanatory and reproducible by other people as you will not be there next to that user every time.

```r
plot(1:10, 21:30)  ## on the left you have code, this is a comment
```



Making your code easier to read is a **must**! To do so, be consistent in the way you write, try to keep your code within a defined margin (RStudio can help you with that), use section breakers (see in the next chunk), and document all the steps in your code. Not only other users, but *your future-self* will be very grateful to those simple good manners.

R can do all the basic arithmetic operations using the correct characters:

```r
#############################################################################
# R as a calculator  ## above you see an example of a section breaker in code
# Basic arithmetic  operators

2+3  ## sum
```

```
[1] 5
```

```r
2-3  ## subtraction
```

```
[1] -1
```

```r
2*3  ## multiplication
```

```
[1] 6
```

```r
2/3  ## division
```

```
[1] 0.6666667
```

```r
3^2  ## exponentiation
```

```
[1] 9
```

```r
################################################################################
# R can also perform multiple operations

2+3+5+10+25-2  ## multiple operations
```

```
[1] 43
```

```r
2+3*4  ## multiplications and divisions are done first
```

```
[1] 14
```

```r
2+10/5
```

```
[1] 4
```

```r
3^2/2  ## but power comes first
```

```
[1] 4.5
```

```r
# Parenthesis are useful to determine the order of operations
(2+10)/5
```

```
[1] 2.4
```

```
# multiple but independent operations can also be done using semicolon
2+3; 2*3; 1-10
```

```
[1] 5
```

```
[1] 6
```

```
[1] -9
```

```
# for large numbers R uses the following schemes
1.2e3 ## 1200 e3 means "move the decimal point 3 places to the right"
```

```
[1] 1200
```

```
1.2e-2 ## 0.012 e-2 means "move the decimal point 2 places to the left
```

```
[1] 0.012
```

An important part of using a programming language is being able to perform *logical operations*. In R, you can do so using:

```
# Logical operators
1 == 1  ## with two equal signs you are stating "1 equals 1"
```

```
[1] TRUE
```

```
1 == 2
```

```
[1] FALSE
```

```
1 != 2  ## with '!=' you are stating "1 differs from 2"
```

```
[1] TRUE
```

```
1 != 1
```

```
[1] FALSE
```

```r
1 > 2  ## 1 greater than 2
```

```
[1] FALSE
```

```r
1 < 2  ## 1 smaller than 2
```

```
[1] TRUE
```

```r
1 > 1
```

```
[1] FALSE
```

```r
1 >= 1 ## 1 greater than or equal to 1
```

```
[1] TRUE
```

```r
# TRUE and FALSE can be stated as:
TRUE == T
```

```
[1] TRUE
```

```r
FALSE != T
```

```
[1] TRUE
```

```r
F == FALSE
```

```
[1] TRUE
```

```r
# other logical operators
## ! & |  --> not, and, or

# Modulo and integer quotients
# Suppose we want to know the integer part of a division, in other words,
## how many 13s are in 119:
119 %/% 13  ## nine 13s in 119
```

```
[1] 9
```

```
# Now suppose we want to know the remainder, or what is left:
119 %% 13
```

```
[1] 2
```

```
# You can use the modulo to test whether a number is odd or even
2 %% 2
```

```
[1] 0
```

```
3 %% 2
```

```
[1] 1
```

```
# Likewise, you can use modulo to test if a number is an exact multiple
## of another
15421 %% 7 == 0  ## if 15321 is an exact multiple of 7, than modulo = 0
```

```
[1] TRUE
```

Storing results of your operations is needed if you want to use such results later for more complex intents. In R, there are two ways to store a result into an object:

```
x <- 1+2  ## press alt + minus sign in RStudio
x
```

```
[1] 3
```

```
y = 1-x
y
```

```
[1] -2
```

Some programmers prefer to leave the equal sign for operations, but since it works similarly to the "<-" sign in R, it is ok to use it as such. Just **be consistent** in the way you use them!

A variable or object does not need to be a single letter. It can be a name, or two names (if connected by a . or _, do not use spaces). Attention: ***R is case sensitive***. The information to be stored does not need to be a number. It can a character string, an equation, a function or another variable.

```
# storing the answer of an equation
answer = y - x == 3
answer
```

```
[1] FALSE
```

```
# R is case sensitive! So Y is not the same as y
y
```

```
[1] -2
```

```
Y = 3
y + Y
```

```
[1] 1
```

```
# use " " for character strings
written.answer <- "y - x is not 3"
written.answer
```

```
[1] "y - x is not 3"
```

Objects in R can store multiple types of information. For example, lists or vectors can be created as such:

```
sequence = 1:10
sequence
```

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

```
# you can also combine values into a vector or list using:
sequence = c(1, 2, 3, 4, 5)  ## use the combine function
sequence
```

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

Matrices can also be created by combining vectors:

```r
# Matrices by combining vectors
vec.1 = c(2, 4, 6)
vec.2 = c(3, 5, 7)

matrix.1 = rbind(vec.1, vec.2)  ## binds them by row
matrix.1
```

```
      [,1] [,2] [,3]
vec.1    2    4    6
vec.2    3    5    7
```

```r
matrix.2 = cbind(vec.1, vec.2)  ## binds them by column
matrix.2
```

```
     vec.1 vec.2
[1,]     2     3
[2,]     4     5
[3,]     6     7
```

Arithmetic and logical operations can be conducted on vectors and matrices:

```r
vec.1 + vec.2  ## sums the compatible positions
```

```
[1]  5  9 13
```

```r
sequence + vec.1  ## smaller lists will be repeated
```

```
Warning in sequence + vec.1: longer object length is not a multiple of shorter
object length
```

```
[1] 3 6 9 6 9
```

```r
vec.1 < 4  ## logical operations with a vector
```

```
[1]  TRUE FALSE FALSE
```

```r
sequence == vec.1  ## logical operators also work; positions are repeated
```

```
Warning in sequence == vec.1: longer object length is not a multiple of shorter
object length
```

```
[1] FALSE FALSE FALSE FALSE FALSE
```

Functions

An important aspect of R is the possibility of using and writing functions. Functions are always used in the format *function()* with the arguments of that function being provided between the parenthesis. For example, instead of defining a sequence of numbers, as we did before, we can use a function to do so:

```r
sequence.1 = 1:20
sequence.2 = seq(from = 1, to = 20)
sequence.1 == sequence.2  ## compares each position
```

```
 [1] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
[16] TRUE TRUE TRUE TRUE TRUE
```

This is because functions usually enable more complex results to be achieved. For example:

```r
seq(from = 1, to = 20, by =2)  ## increment by 2
```

```
 [1]  1  3  5  7  9 11 13 15 17 19
```

```r
seq(from = 1, to = 20, length.out = 5)  ## five numbers will be spread along
```

```
[1]  1.00  5.75 10.50 15.25 20.00
```

```r
                                        ## this sequence

seq(0, 10)  ## if you do not use the argument's name, the order is kept
```

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

```r
# Repetitions:
rep(9, 5)  ## repeat 9 five times
```

```
[1] 9 9 9 9 9
```

```r
rep(1:4, 2)
```

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

```r
rep(1:4, each = 2)  ## repeat 1:4, each number twice
```

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

```r
rep(1:4, each = 2, times = 3)
```

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

```r
# it also works with character strings
rep("cat", 5)
```

```
[1] "cat" "cat" "cat" "cat" "cat"
```

```r
rep(c("cat", "dog", "mouse"), times = 1:3)
```

```
[1] "cat"   "dog"   "dog"   "mouse" "mouse" "mouse"
```

```r
# remember:
vec.1 < 4
```

```
[1]  TRUE FALSE FALSE
```

```r
# we can check this with more detail
all(vec.1 < 4)  ## all() tests if ALL values in the vector are less than 4
```

```
[1] FALSE
```

```r
any(vec.1 < 4)  ## is there ANY values in the vector less than 4?
```

```
[1] TRUE
```

R base has many functions implemented for running common mathematical operations, such as logarithms, square roots, means, etc.

```r
# logarithms
log(42)  ## natural log
```

```
[1] 3.73767
```

```r
log10(42)  ## base 10 logs
```

```
[1] 1.623249
```

```r
log10(42*54)  ## operations can be fitted as arguments
```

```
[1] 3.355643
```

```r
log10(seq(2, 10, by = 2))  ## functions can also be fitted as arguments
```

```
[1] 0.3010300 0.6020600 0.7781513 0.9030900 1.0000000
```

```r
log(19, 3)  ## the second argument of log allows determining the base
```

```
[1] 2.680144
```

```r
# antilog function
exp(1)
```

```
[1] 2.718282
```

```r
log(10)
```

```
[1] 2.302585
```

```r
exp(log(10))
```

```
[1] 10
```

```r
# Square root
X = sqrt(4)
X*X == X^2
```

```
[1] TRUE
```

```r
# mean, median, sum, max, min, range
mean(1:10)
```

```
[1] 5.5
```

```r
sum(1:3)
```

```
[1] 6
```

```r
max(1:52)
```

```
[1] 52
```

```r
min(1:52)
```

```
[1] 1
```

```r
sum.rep = rep(1:3, times = 1:3)
sum(sum.rep)
```

```
[1] 14
```

```r
mean(sum.rep)
```

```
[1] 2.333333
```

```r
median(sum.rep)
```

```
[1] 2.5
```

```r
z = c(5, 3, 6, 7, 7, 6, 2, 1, 10)
range(z)
```

```
[1]  1 10
```

```r
# Rounding numbers
floor(5.7)  ## the greatest integer less than
```

```
[1] 5
```

```r
ceiling(5.7)  ## next integer
```

```
[1] 6
```

```r
round(x = 5.7562, digits = 0)  ## this allows you to determine how many decimals you want
```

```
[1] 6
```

```r
round(5.7562, 1)
```

```
[1] 5.8
```

```r
round(5.7562, 2)
```

```
[1] 5.76
```

```r
signif(12345678, digits = 4)  ## signif() works similarly, but for large integers
```

```
[1] 12350000
```

```r
signif(12345678, digits = 2)  ## see the notation for 12 million
```

```
[1] 1.2e+07
```

```r
# Obtaining the length of a list
length(sequence.2)
```

```
[1] 20
```

```r
length(Y)
```

```
[1] 1
```

```r
length(seq(0, 10, by = 2))
```

```
[1] 6
```

Sorting elements in a list can be useful:

```r
# sorting elements in a vector/list
z
```

```
[1]  5  3  6  7  7  6  2  1 10
```

```r
sort(z)
```

```
[1]  1  2  3  5  6  6  7  7 10
```

```r
# now suppose we want to sum the 3 largest numbers in this list
rev(sort(z))
```

```
[1] 10  7  7  6  6  5  3  2  1
```

```r
rev(sort(z))[1:3]  ## it helps working the code piece-by-piece
```

```
[1] 10  7  7
```

```
sum(rev(sort(z))[1:3])
```

```
[1] 24
```

And defining which elements in a list correspond to a value is **very** useful:

```
# which
z
```

```
[1]  5  3  6  7  7  6  2  1 10
```

```
which(z == max(z))  ## which element in z has the maximum value of z
```

```
[1] 9
```

```
z[9]
```

```
[1] 10
```

```
which(z == min(z))
```

```
[1] 8
```

```
z[8] == min(z)
```

```
[1] TRUE
```

```
# or use which.max/which.min
which.max(z)
```

```
[1] 9
```

Objects have different classes (which makes them useful depending on the context), which can be checked using:

```
class(y)
```

```
[1] "numeric"
```

```
is.numeric(y)
```

```
[1] TRUE
```

```
is.numeric(written.answer)
```

```
[1] FALSE
```

```
is.character(written.answer)
```

```
[1] TRUE
```

```
as.character(y)  ## you can represent objects from one type to another
```

```
[1] "-2"
```

```
as.character(y) + Y  ## summing a character with a number should not work
```

```
Error in as.character(y) + Y: non-numeric argument to binary operator
```

Factors are an important class of variables in R. Factors are categorical variables that have a fixed number of levels.

```
obj.colors = factor(c("black", "white", "black", "black", "pink", "white"))  ##
class(obj.colors)
```

```
[1] "factor"
```

```
levels(obj.colors)
```

```
[1] "black" "pink"  "white"
```

```r
nlevels(obj.colors)
```

```
[1] 3
```

```r
length(levels(obj.colors))  ## the length of the levels is = nlevels
```

```
[1] 3
```

Data frames

Data frames in R can be described as a type of table that can be similar to a matrix, but is different in some important aspects.

Matrix	Dataframe
Collection of data sets arranged in a two dimensional rectangular organisation.	Stores data tables that contains multiple data types in multiple column called fields.
It's m*n array with similar data type.	It is a list of vector of equal length. It is a generalized form of matrix.
It has fixed number of rows and columns.	It has variable number of rows and columns.
The data stored in columns can be only of same data type.	The data stored must be numeric, character or factor type.
Matrix is homogeneous.	DataFrames is heterogeneous.

Figure 1: Differences between matrices and data frames in R

A data frame is usually created when you read an external table into R. Let's see an example of how to create a data frame and some information about the object:

```r
dataset = data.frame(c(0:10), seq(100, 200, by = 10))
colnames(dataset) = c("Units", "Tens")
dataset
```

```
   Units Tens
1      0  100
2      1  110
```

```
3      2  120
4      3  130
5      4  140
6      5  150
7      6  160
8      7  170
9      8  180
10     9  190
11    10  200
```

```r
class(dataset)
```

```
[1] "data.frame"
```

```r
head(dataset)  ## returns parts of an object, by default the first 6 lines
```

```
  Units Tens
1     0  100
2     1  110
3     2  120
4     3  130
5     4  140
6     5  150
```

```r
dataset[,1]  ## first column
```

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

```r
dataset$Units  ## column named 'Units'
```

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

```r
dataset[,1] == dataset$Units
```

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

```
dataset[10,1] == dataset$Units[10]
```

```
[1] TRUE
```

Data frames can have column and row names, which can be attached to the R search path of the session (and called by their names) using the function *attach()*.

```
names(dataset)
```

```
[1] "Units" "Tens"
```

```
attach(dataset)
Units
```

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

```
Tens
```

```
 [1] 100 110 120 130 140 150 160 170 180 190 200
```

Missing values

Missing values are treated as a *number* in R denoted by *NA*. Missing values are commonly present in real-world data and can be disruptive to some analyses, so it is important to know how to deal with them.

```
y = c(4, NA, 7)
is.na(y)  ## this will check if there are NAs in the vector y
```

```
[1] FALSE  TRUE FALSE
```

```
# to produce a vector with the NA striped out, use:
y[!is.na(y)]  ## this reads: "values of y which are not NA
```

```
[1] 4 7
```

```r
# some functions will not work when there are missing values in the data
x = c(1:8, NA)
mean(x)
```

```
[1] NA
```

```r
mean(x, na.rm = T)  ## mean() has an argument to remove NAs
```

```
[1] 4.5
```

```r
# it is also possible to replace the NAs using the ifelse() function
ifelse(test = is.na(x), yes = 9, no = x)  ## if yes, then replace by 9
```

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

End of session

Always save the script you are working on. It might be useful by the end of a session to check the objects created:

```r
objects()
```

```
 [1] "answer"         "ctl"            "dataset"        "group"
 [5] "lm.D9"          "lm.D90"         "matrix.1"       "matrix.2"
 [9] "obj.colors"     "opar"           "sequence"       "sequence.1"
[13] "sequence.2"     "sum.rep"        "trt"            "vec.1"
[17] "vec.2"          "weight"         "written.answer" "x"
[21] "X"              "y"              "Y"              "z"
```

You can save and then read them in the next section to start from where you stopped. To save all objects in a session, use:

```r
save.image("SessionData.RData")
```