Start R

Vincent Bonhomme

2022-10-07

Contents

Chapter 1

Preamble

This book is primarily intended for students and researchers interested in R but with no prior knowledge of the language.

Dozens of similar books on the web exist, most of them are even open and free. This one has been largely guided with the help of 700+ students so far, mostly from 2014 to 2019. Also, this is how I wish R would have been taught to me, almost 20 years ago.

This book is the transcription of a 20 hours training session that will take you from scratch to a general understanding of the language, practical data manipulation and visualization. At the end of it, you should no longer be afraid of a list, writing your own functions or to dig into stack overflow archives.

Nowadays everyone is in an hurry and desire to hit the ground running with R. Even if, as for natural languages, mastering a programming language takes years. This book is written with this in mind and often points to curated resources where you can mastering a particular aspect of the language.

This edition is a complete rewriting started in 2022 of a first one, not publicly released. This edition aims to put the tidyverse even more at core but maintain a prior solid training on R basic grammar and philosophy. Until completely rewrote, I will keep updating it whenever I have time to do so.

Chapter 2

Installation

2.1 Installing R and RStudio

- You will need to install R from its official website.
- Also install RStudio, one of its IDE (integrated development environment).

You should now have a minimal yet fully working R installation. Most of what follows can directly be done in bare R yet Rstudio brings everything in the same place in a nice, user-friendly and productive environment.

2.2 Introducing RStudio

When you open RStudio, it runs R in the background. At startup you will have four windows :

For now, we will focused on the two on the left column. On top, you will have your "scripting" window, the place to write and save code. Just below, you have the "console" where you can type code directly but also throw lines or full scripts from the window above.

The typical workflow will be to: i) try code in the console, ii) when you're happy with it, save it into your script. This will help you redo whenever and wherever you want your analyses, ie to do some *reproducible* analyses.

At the console, up and down arrows will navigate through your command history. Ctrl+L will erase the console, not the history. You can also access history through the "History" tab in RStudio.

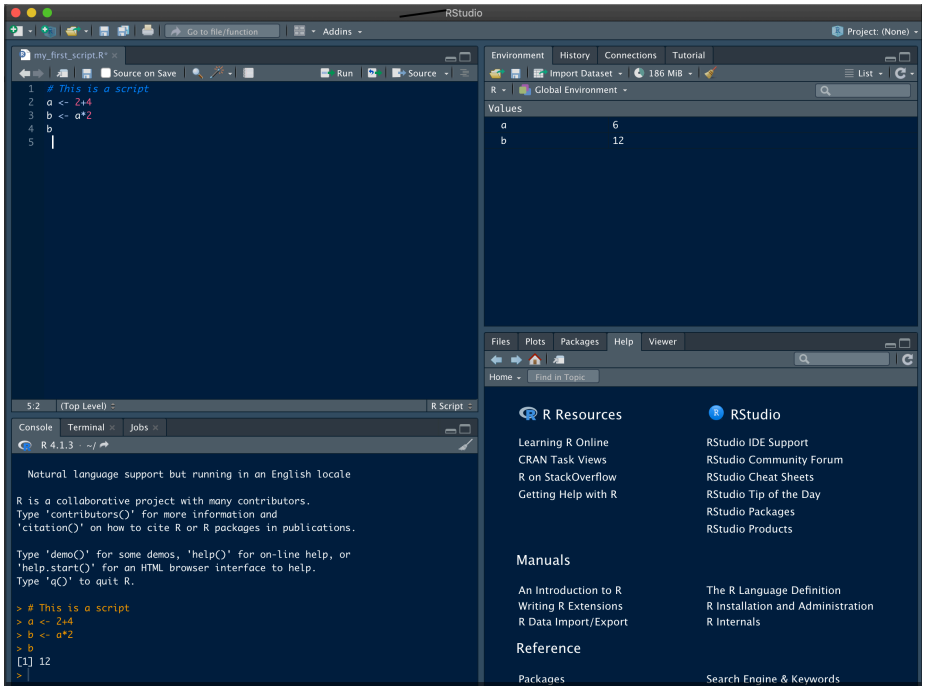

Figure 2.1: Rstudio.png

On the scripting window, Ctrl+Enter will execute the current line, Maj+Ctrl+B will execute every line from the top of your script to the current line. Use these shortcuts from the very beginning of of your journey into R! Moe of them can be found through Rstudio's menu when you access "Keyboard shortcuts help".

Chapter 3

R101

3.1 Hello world

To follow up the tradition, let's print a message on the console. Open a script, paste the lines below, save it as "hello.R" and run it. You can also type it directly into the console.

```
hello <- "Hello world!"
hello
```

```
## [1] "Hello world!"
```

As you can see, R code is pretty natural and easy to read. Usually we read it from right to left (and also from nested brackets to outer brackets).

Here, we have typed a message, explicitly created as a chain of characters because we put it into quotes (`"`)[1]. Then we use the **assignment operator** `<-` whose direction indicates that we assign it into a **variable** called `hello`.

When `hello` is **called**, it simply prints it message on the console[2].

3.2 Basic arithmetics

You're likely here to manipulate numbers not only strings, so here we go:

[1] You can use simple quotes (`'`) as well. Having two different quotes allow to use one for defining a string, and the other to use as a litteral quote, try: `print("Hello 'quoted' world")`

[2] R does actually call `print(hello)` for you

```
x <- 32*2
y <- sqrt(x)
y
```

```
## [1] 8
```

```
y^2
```

```
## [1] 64
```

R is a sophisticated calculator that you can use to do basic arithmetics. Here we assigned the result of `32*2` into a variable called `x`, then we calculated the square root of `x` and assigned it to another variable called `y`. To do so, we used the `function` `sqrt()` which is short for square root. When the `sqrt` function is *passed* with the **argument** x, the value of the latter is retrived and its square root is returned.

The `*` and the `^` are arithmetic **operators**. You can access all of them by typing `?Arithmetic` in the console. An operator roughly means "shortcut", in other words, a function call to hides its brackets.

3.3 Functions and help pages

In R, everything that has brackets is a function call[3].

All functions have a manual page that can be accessed using `?function_name`, for example `?print`. Some general topics are also accessible for example `?Arithmetic`.

Every time you discover and/or struggle a function, read the manual. Help pages are not exactly poetry: they may appear boring and disturbing at first but there is beauty in their concision and what you're looking for is probably there. Most of them also have examples that you can copy/paste to better understand what they do[4].

3.4 Vectors

So far, we have only had variables with a single value or **scalars**. A variable can be a **vector** and contain more than a single value. To create vectors, we use the function `c` which names stands for "concatenate".

[3] and more generally everything is a function call in R, even when you do not see brackets (try `"*"(1, 2)`).

[4] One can also use `example("function_name")`

```
x <- c(1, 2, 3, 4, 5, 6, 7)
x^2
```

```
## [1]  1  4  9 16 25 36 49
```

Here we assigned a new value to `x`, so the previous one (64) is lost. As you can see, when calculating `x^2`, we calculate the square of all values of `x`. Not all functions behave this way since some of them return a scalar, by construction:

```
length(x)
```

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

```
sum(x)
```

```
## [1] 28
```

```
sd(x)
```

```
## [1] 2.160247
```

Here we accessed the number of elements in `x` (with `length`), its `sum` and its standard deviation (`sd`).

Now let say we are interested in calculating the sum of integers not from 1 to 7, but from 1 to 100. As you can imagine, there must be a smarter way than typing `c(1, 2, ..., 99, 100)`. You're right, and that's the job of `seq`:

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

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

3.5 The golden rule

Possibly the most important rule in R, and in any programming language is :

> If something sounds dumb and/or repeatitive there must be a smarter way to do it.

There are several ways to avoid to do dumb and/or repeative things like having vocabulary (knowing functions to avoid paraphrasing) or the right grammar (ie defining your own functions to avoid copy/pasting entire blocks with minro changes between them). Be sure, that of this will come naturally with time and mistakes. Everything in its time, the first step is to make things work, then the aim will be to doing it in a smarter way.

Back to regular sequences, this is such a repetitive need that we have a shortcut for that, thanks to the operator `:`.

```
sum(1:100)
```

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

3.6 Function arguments

When using `seq` as above, you can do more things because there are different flavours of regular sequences. That's the purpose of functions **arguments**. You can see arguments as functions parameters or options, but we will call them arguments. They are easily recognized because they come into functions brackets. In the code above they were explicitely named so that we know, at first glance, what they do: we `start` from 1, we go `to` 100, `by` an increment of 1.

This does not come out of the blue but from the function **definition**, that is how the function was written by someone else for you to use it. Have a look to the manual. Past the summary, the second and third section are "Usage" and "Arguments". Again, everytime you feel that a function could do something that fits your needs, it is probably there and you have to find the information in the manual. For instance if you want, only even numbers, or a sequence of a length 12, it is simply variants of `seq`:

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

```
## [1]  0  2  4  6  8 10
```

```
seq(0, 10, length=4)
```

```
## [1]  0.000000  3.333333  6.666667 10.000000
```

If you compared the syntax to the Usage section of `?seq`, you will probably note that above, I omitted the name of the first two arguments (`from` and `to`), and also that the third argument was `by` and then `length`. I also abbreviated `length.out` as `length`, and yet, it works as expected. The rules to know when working with arguments follows:

- You can omit argument names as long as they come in the same order as found in the `Usage` section.
- You can abbreviate argument names as long as the abbreviation in unequivocal among all defined arguments
- If an argument is not specified, it takes its default value, as mentionned in the Usage section
- You can, yet it is never necessary nor a good idea, change the order of arguments.

In other words, these commands are strictly equivalent:

```
seq(from=1, to=5, by=1.2)
```

```
## [1] 1.0 2.2 3.4 4.6
```

```
seq(f=1, t=5, b=1.2)
```

```
## [1] 1.0 2.2 3.4 4.6
```

```
seq(1, 5, 1.2)
```

```
## [1] 1.0 2.2 3.4 4.6
```

```
seq(by=1.2, to=5)
```

```
## [1] 1.0 2.2 3.4 4.6
```

3.7 Other sequence generators

Also have a look to `rep` that replicated elements of vectors:

```
rep(1:3, each=3)
```

```
## [1] 1 1 1 2 2 2 3 3 3
```

```
rep(5:2, times=2)
```

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

We will explored them later, but if you are looking for random number generation, they are named `r+abbreviated_distribution_name`:

```
set.seed(123)
runif(5, 0, 1)
```

```
## [1] 0.2875775 0.7883051 0.4089769 0.8830174 0.9404673
```

```
rnorm(10, 0, 1)
```

```
##  [1] -1.6895557  1.2394959 -0.1089660 -0.1172420  0.1830826  1.2805549
##  [7] -1.7272706  1.6901844  0.5038124  2.5283366
```

3.8 Recycling

When you do an operation between two vectors, it is usually done element-wise:

```
x <- 1:5
y <- 10:6
x
```

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

```
y
```

```
## [1] 10  9  8  7  6
```

```
x*y
```

```
## [1] 10 18 24 28 30
```

But when the length of vectors is not compatible, R does **recycle** the shorted vector to match the length of the longer one. Usually, recycling is useful but it may also be troublesome, particularly when it does not come with a warning.

```
z <- c(0, 1)
x
```

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

```
z
```

```
## [1] 0 1
```

```
x+z # equivalent to c(1, 2, 3, 4, 5) + c(0, 1, 0, 1, 0)
```

```
## Warning in x + z: longer object length is not a multiple of shorter object
## length
```

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

3.9 Indexing

To access a particular element in an object, we will use the square bracket `object[i]`, with i being the index of the element to access:

```
x <- c(3, 1, 5, 4)
x[2]
```

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

```
x[3]/4
```

```
## [1] 1.25
```

In R, the first element is 1[5].

Indexing can be used to access a particular value and/or to change it when combined with the **assignment operator**

```
x
```

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

```
x[2] <- 1.2
x
```

```
## [1] 3.0 1.2 5.0 4.0
```

You can also access more than one element:

[5]This may sound natural but in most languages it is 0

```
x[2:4]
```

```
## [1] 1.2 5.0 4.0
```

```
x[c(1, 3)]
```

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

And change more than a single value at a time. Recycling applies here

```
x
```

```
## [1] 3.0 1.2 5.0 4.0
```

```
x[2:4] <- c(0.2, 0.4, 0.5)
x
```

```
## [1] 3.0 0.2 0.4 0.5
```

```
x[2:4] <- 99
x
```

```
## [1]  3 99 99 99
```

So far we have seed **positive indexing** but sometimes it is also useful to state which elements you do NOT want:

```
x <- 3:7
x
```

```
## [1] 3 4 5 6 7
```

```
x[-1] # all but the first
```

```
## [1] 4 5 6 7
```

```
x[-length(x)] # all but the last
```

```
## [1] 3 4 5 6
```

```
x[-c(1, length(x))] # trim both ends
```

```
## [1] 4 5 6
```

You can also use variables:

```
indices <- c(1, 4)
x <- -2:3
x
```

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

```
x[indices]
```

```
## [1] -2  1
```

3.10 Tests and logicals

You can do logical tests in R. They return a vector of type "logical".

```
x <- 1:5
x > 3
```

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

You can directly pass this vector of type `logical` as an indexing variable or turn them into `numeric` indices:

```
x[x > 3]
```

```
## [1] 4 5
```

```
above_three <- which(x>3)
x[above_three]
```

```
## [1] 4 5
```

To get the full list of relational operators, type `?Comparison`. You can also do set operations, see `?sets`; `%in%` operator is often used

```
x
```

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

```
x==4
```

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

```
x<=3
```

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

```
x!=2
```

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

```
x %in% c(2, 4)
```

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

3.11 NA

```
x[seq(1, 5, by=2)]
```

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

On the last command, we tried to access all elements from 1 to 5 with an increment of 2 (ie `{1, 3, 5}`). The `x` only has 4 elements, so the 5^th does not exist. R returns `NA` which stands for **Non available**. NAs are very common when doing data analyses and some functions may not work as expected as long as there is a single NA in the object you're doing calculations with:

```
x <- c(1, 5, NA, 6)
mean(x)
```

```
## [1] NA
```

`NA` is returned here for a good reason: the missing value could be 4 or 3 billions! If you want to ignore NAs, you must explicitely mention it to R. Some functions have a `na.rm` (remove NAs) arguments, otherwise there is the `na.omit` function:

```
mean(x, na.rm=TRUE)
```

```
## [1] 4
```

```
mean(na.omit(x))
```

```
## [1] 4
```

3.12 Other structures

Besides scalars and vectors, R can handle objects with more dimensions such as **matrices** which are 2-dimensionnal **arrays**, **lists** and **data.frame** that are rectangular lists.

3.12.1 Matrices

Let's begin with matrices:

```
mat <- matrix(1:9, nrow=3, ncol=3)
mat
```

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

Matrices are rectangular objects that contain number on which we can do many thing, including arithmetic operations:

```
sum(mat)
```

```
## [1] 45
```

```
mat/2
```

```
##      [,1] [,2] [,3]
## [1,]  0.5  2.0  3.5
## [2,]  1.0  2.5  4.0
## [3,]  1.5  3.0  4.5
```

```
colSums(mat)
```

```
## [1]  6 15 24
```

They can be indexed using the syntax: `object[i, j]`. We have one more indexing argument, since we now have two dimensions. The first being the row index, the second being the column index by convention. If you omit one, this means "take all of them". Otherwise, all other indexing rules applies.

```
mat[1, 2] # first row, second column
```

```
## [1] 4
```

```
mat[2, ] # second row
```

```
## [1] 2 5 8
```

```
mat[-2, c(1, 3)] # all rows but the second, first and third columns
```

```
##      [,1] [,2]
## [1,]    1    7
## [2,]    3    9
```

Among useful functions when working with matrices, we have:

```
dim(mat)
```

```
## [1] 3 3
```

```
nrow(mat)
```

```
## [1] 3
```

```
ncol(mat)
```

```
## [1] 3
```

```
colnames(mat) # not defined so far
```

```
## NULL
```

```
colnames(mat) <- c("a", "b", "c") # let's name columns
colnames(mat) # now defined
```

```
## [1] "a" "b" "c"
```

3.12.2 Lists

Lists are vectors than can contain objects of different type. They can be named or not:

```
x <- list(a=1, b="hello", c=mat)
x
```

```
## $a
## [1] 1
##
## $b
## [1] "hello"
##
## $c
##      a b c
## [1,] 1 4 7
## [2,] 2 5 8
## [3,] 3 6 9
```

They can be indexed with three different syntaxes: `[`, `[[` or `$`. `$` will only work for a **single** and **named** element:

```
x$mat
```

```
## NULL
```

This is mostly useful when working with `data.frame`s that are a particular flavor of lists, to retrieve a single column. We will be back at it later.

Single and double square brackets are quite confusing at first glance, but they allow to retrieve either the element and retain its list nature (`[`), or the element and dropping its list nature (`[[`):

```
x[1]
```

```
## $a
## [1] 1
```

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

```
## [1] "list"
```

```
# x[1] + 3 This won't work as we still have a list

x[[1]]
```

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

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

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

```
x[[1]]+3
```

```
## [1] 4
```

3.12.3 data.frames

Data frames are rectangular lists. They are the natural structure in data analysis because they correspond to a spreadheet.

Consequently and practically, they also are the central object in modern R, at the very core of the tidyverse grammar.

3.13 Footnotes