# QStep — Week 2 Lab: Getting Started in R

Leonardo Carella

## 1. Installing R and R Studio (to do before class)

Before the lab, you should have received instructions on how to download **R and R Studio**:

- Step 1: download R itself, through https://www.r-project.org/ (you will need to select a CRAN mirror depending on your location), or go directly to the UK mirror through https://www.stats.bris.ac.uk/R/. Select your operating system (Mac, Windows, and Linux are supported), and the version of R you want to download, depending on which version of the operating system your computer runs on.
- Step 2: download RStudio, through https://rstudio.com/products/rstudio/download/. "RStudio Desktop – Open source license" is what you want. Select the version of RStudio you want to download, depending on which version of your operating system your computer is running on.

What are these things, by the way?

- **R** is a programming language and environment for statistical computing. It's one of the most popular software for data science in academia and increasingly in the industry. It's free, open source, and collaborative: developers around the world are constantly increasing R's functionalities, making them directly available to users through packages, which are also free to install (more about that later on in the course).
- **R Studio** is an Integrated Development Environment for R: a front-end that runs your code through R, allowing you to do so from a user-friendly interface instead of coding through R's rather basic console. All the coding in this course will be done in RStudio, but **you need to have R installed for RStudio to work**.

Once you've installed R and RStudio, **open RStudio** (not R), and let's get to work.

## 2. Basic Syntax: Operators, Functions, and Objects

First thing, click on the "New File" icon (the green plus on a white square) in the top-right corner of your screen and select "R Script". This will be the blank canvas where we type in our code.

At its most basic, you can use R as a calculator: type out arithmetic operations in the *script* section (the field in the top left of the interface), highlight the chunk of code you want to run (say, 2+2), then press the "Run" button. You will get the result in the *console* section (the field in the bottom left of the RStudio interface). You can also tell R to run your code with command+shift instead of pressing "Run".

```
2+2
```

R evaluates mathematical operators in the usual order: Parentheses first, then Exponentiation, then Multiplication, then Division, then Addition, and then Subtraction (PEDMAS). Only use round parentheses (brackets and braces are used for different purposes in R), and use the dot as decimal separator instead of commas, as it's the convention in English-speaking countries.

```
-3*5
(20-10)/2.5
7^2
```

This is a good place to introduce your first *functions*. Functions in R are expressions that come with parentheses and within parentheses you pass one or more arguments. These are some simple mathematical functions, where you pass one numerical argument and they return the result of an operation:

```
abs(-10) #computes the absolute value
sqrt(81) #computes the square root
log(4) #computes the *natural* logarithm
```

In R, you can write down notes to your code by using the commenting symbol # before your text. R will not run anything that follows a hashtag on a line. In all other cases, R is not line-sensitive: you can space your code across lines, and as long as you select the whole chunk, it will still recognise it as a continuous block of code. Note also that R is not whitespace-sensitive: `2+2` and `2    +    2` will return the same answer (try it).

Sometimes you have to pass more than one argument to the function. Use commas to separate the arguments:

```
log(8, base = 2) #computes the logarithm of a number in base 2
round(123.456789, digits = 2) #rounds a number to the 2nd decimal place
```

If you want to know more about what a function does, you can type ? and then the function, and run the code. Or indeed, use the `help()` function. This call will open the *help* window in the bottom right of your interface, with descriptions and examples of how the function can be used

```
?log
?round
```

You can nest functions and operations within each other in a single expression using parentheses:

```
sqrt(10*5-1)
round(log(5+5)*2, digits = 3)
```

It is often useful to "store" values as named objects, using the **all-important** `<-` (assign) operator. These values can be numbers or text strings; in the latter case you have to surround the text with quote marks.

```
x <- 7
y <- 4+4
my_name <- "Thomas Bodley"
```

When you assign values to a named object, R does not return the value in the *console*, but it will appear in the *global environment* window in the top right of the interface. This means that you have successfully stored this value, and at any point you can simply run your object's label (e.g. `my_name`, or `x`) and R will print the value you have assigned to it. (Also note the underscore in `my_name`: when you name objects in R, the names must be continuous text.)

You can pass these named objects as arguments of functions or operations, just as we did numbers:

```
sqrt(x)
(x*y)/2
```

Obviously don't try to calculate the square root of your name, because you will get an error:

```
sqrt(my_name)
```

## 3. The Building Blocs: Vectors and Dataframes

We normally work with variables taking a series of values across a number of observations rather than a single quantity. Imagine for instance a spreadsheet with a list of countries, and a column recording their respective GDP per capita; or again a list of survey respondents and a list of responses to a question ("Agree", "Agree", "Disagree" etc.). R handles these ordered sequences of values, be them numbers or characters, as *vectors*, and you can create your own vectors using the `c()` function (c stands for combine or concatenate, there is some controversy in the R community):

```
primes <- c(2, 3, 5, 7, 11, 13, 17, 19, 23, 29)
# a numerical (integer) vector

terms <- c("Michaelmas", "Hilary", "Trinity")
# a character vector, or string
```

Indeed, you have already worked with vectors before: the named objects `x`, `y` and `my_course` were vectors of length 1, because they contained only one element. We call the number of elements in a vector its 'length'. Now we have created two longer vectors: the integer vector `primes` of length 10, and the character vector (also known as a string) `terms` of length 3.

Here are some other ways to create vectors:

```
one_to_ten <- 1:10
#creates a vector with all integers between 1 and 100

decimals <- seq(0, 10, by = 0.1)
#creates a vector with all numbers from 0 to 1 by intervals of 0.1

rep(terms, times = 10)
# creates a vector repeating the first argument (the vector
# 'terms'), n times, where n is the second argument (10).
# note: this time we haven't stored the output with <-
```

You can now pass the functions and operations described above to the whole sequence. The function is applied to each element of the original vector, and R will compile a vector with the sequence of results:

```
squares_of_primes <- primes^2
```

You can also obtain new vectors by manipulating vectors you already have, for instance, this syntax

```
primes+one_to_ten
```

returns a vector where the first element is the sum of element 1 of `primes` and element 1 of `one_to_ten`, the second element is the sum of element 2 of `primes` and element 2 of `one_to_ten`, and so on. This will be very useful when we learn how to create new variables in a dataframe.

Luckily, in this case both `primes` and `one_to_ten` are the same length (10 elements); see the homework to figure out what happens when the vectors are of different length.

Other functions can be applied to a vector, and they return a single value computed from all the elements in the vector:

```
length(primes) #returns the number of items in a vector
sum(primes) #returns the sum of elements of the vector
mean(primes) #returns the mean of elements of the vector
median(primes) #returns the median of elements of the vector
max(primes) #returns the maximum value in the vector
min(primes) #returns the minimum value in the vector
```

You can also ask R to evaluate each element of a vector according to a logical expression. This will return a vector of logical values TRUE/FALSE. *Stick a pin on this because it will be very useful when we index dataframe variables later on.* For instance:

```
one_to_ten > 6 #is the element larger than six?
one_to_ten >= 4 #is the element larger than or equal to four?
one_to_ten == 1 #is the element a one?
one_to_ten != 10 #is the element different from 10?
```

**Note the double equal in the last line of code**. When we want to express equality as a logical or mathematical statement in R, we need to use `==`. The single `=` is used within functions; otherwise, it works as an equivalent of the `<-` (assign) operator. So `one_to_ten = 1` would be creating an object called `one_to_ten` with only one element, the number 1.

Just like vectors are collections of objects, we can create collections of vectors: *dataframes*. These are different from vectors in two key respects: (1) they have two dimensions, rows and columns, and (2) they can take values of different type (numbers, characters, logicals) in different columns. You can create your first dataframe so by passing your vectors in the `data.frame()` function. By default, your vectors will be treated as columns, and the vector names will become column names.

Dataframes can take text, numerical and logical vectors. As long as they're the same length, you're good to go:

```
students <- c("Aditya", "Betty", "Charlie")
grade <- c(70, 55, 35)
pass <- grade >= 50

my_dataframe <- data.frame(students, grade, pass)
```

To visualise `my_dataframe`, use the `View()` function:

```
View(my_dataframe)
```

## 4. Setting up a Working Directory

Now let's work with a real dataset. In practice, we almost never build up dataframes from scratch; in this course, we will mostly be working with datasets compiled by other researchers.

Make sure you have downloaded the file `brexit.csv` that comes with this lab material. It might be useful to move it to a folder dedicated to the material for the course: this will be your working directory, and it might be easier to have all your data in one place.

**A working directory is the default folder on your computer where R will look for files you want to load and where it will put any files you save.** The easiest way to set up a working directory is to select the 'Session' drop-down menu at the top of the RStudio interface, select `Set Working Directory`, then select `Choose Directory`, then search for the folder you've saved the file in (say, a folder called "QStep"), and click `Open`. Something like this should appear in your console:

```
setwd("~/Desktop/QStep")
```

To check if this has worked, run

```
getwd()
```

Does the file path appear in the console? If so, you've set up your working directory and you can now skip to point 5. You will have to set up a working directory every time you close RStudio and open a new RStudio session (or when it crashes).

If it has not worked, you can manually set the working directory by copying the file path of the dataset you want to import and passing it through the `setwd()` function:

```
"/Users/yourname/Desktop/QStep"
```

- For Mac users: you can get the file path by right-clicking on the file `brexit.csv` in your documents, selecting `Get Info` and then copying what comes after `Where:` in the Info box.
- For Windows users: right-click on `brexit.csv` in your documents, select `Copy as path`. **You may have to change the backward slashes to forward slashes in the R Script.**

Now pass your file path through the `setwd()` function, making sure to enclose it within quote marks:

```
setwd("/Users/yourname/Desktop/QStep")
```

## 5. Loading and Exploring a Dataset

Once you have set a working directory, import the dataframe and store it as a dataframe called `brexit` with the `read.csv()` function and the assign operator `<-`

```
brexit <- read.csv("brexit.csv")
```

Hopefully, a new object called `brexit` will have appeared in your environment panel. **For some Mac users, this may not have worked because some of the newer Macs treat csv files as Numbers file**. In this case, you want to open quality_of_government in `Numbers`, and then export it as a `.csv` file named "brexit.csv" (`File > Export To... > CSV`).

This is the dataset we are going to work with. It was compiled by merging 2016 EU referendum data at the local authority level with information from the census. Each row corresponds to one of the 380 local authorities in Great Britain. These are the observations (in other datasets, these might be countries, or individuals, or parties etc.). The variables in the columns are as follows:

| Variable | Description |
|---|---|
| `area` | Local authority name |
| `region` | Region/Country name |
| `electorate` | Number of people on the electoral ballot |
| `valid_votes` | Number of valid votes |
| `total_votes` | Number of votes cast |
| `remain_votes` | Raw number of votes for Remain |
| `leave_votes` | Raw number of votes for Leave |
| `percent_remain` | % votes for Remain |
| `percent_leave` | % votes for Leave |
| `area_type` | Rural Urban Classification (England) |
| `percent_degree` | % adults with level 4+ qualification (degree) |
| `median_age` | median age (2016 mid-year, one decimal point) |
| `GVA` | gross value added per head |
| `percent_UK_born` | % residents born in UK |
| `percent_white` | % residents who identify as white |
| `median_earnings` | median weekly earnings in the area (£) |
| `percent_AB_socialgrade` | % professional/managerial occupations |
| `percent_C1_socialgrade` | % clerical/administrative occupations |
| `percent_C2_socialgrade` | % in skilled manual occupations |
| `percent_DE_socialgrade` | % in unskilled manual occupations/unemployed |

Some useful functions to inspect a new dataset:

```
View(brexit) # shows the whole dataset in a new window
dim(brexit) # returns number of rows and columns in the dataset
colnames(brexit) # returns a vector of variable names
summary(brexit) # returns a summary of each variable
head(brexit) # prints the first six rows of the dataset
```

## 6. Describing Variables and Handling Missing Data

**An essential operator when we work with dataframes is `$` (extract)**. It allows us to select one variable in the dataframe to work with. So, for instance if the column we're looking for is `percent_leave` (i.e. the percentage of leave vote in a local authority) in the dataframe `brexit`, we call

```
brexit$percent_leave
```

and it will print out the variable (remember to get the capital letters right! R is case-sensitive). Note that this is a *vector*: exactly like our friends `one_to_ten`, `primes` etc. and we can treat it exactly in the same way. We can apply the functions we learnt before, and some new ones as well:

```
mean(brexit$percent_leave)
median(brexit$percent_leave)
max(brexit$percent_leave)
summary(brexit$percent_leave) #prints a summary of the variable distribution
```

Annoyingly, when a variable contains missing values (denoted by `NA`), we have to remind R to remove them from our calculations by setting `na.rm = TRUE` in our function, otherwise R will be unsure of how to compute the mean between numerical and logical values. Try for instance, the variable `percent_UK_born`

```
mean(brexit$percent_UK_born) # produces NA (a missing value)
mean(brexit$percent_UK_born, na.rm = TRUE) # works fine! :)
```

To learn about categorical variables, we can use the `unique()` function to get a vector of the unique elements occurring in a vector, or the `table()` function to see how many times a value occurs.

```
unique(brexit$region)
table(brexit$region)
```

## 7. Indexing

R uses square brackets `[ ]` to select some values out of an object, such as vectors and dataframes, according to their position. For instance, if I wanted to select the first row and the second column of my dataset, I would call:

```
brexit[1,2]
```

Remember: for rectangular data like dataframes, the first dimension is the *row* and the second dimension is the *column*.

You can also pass vectors instead of single values, for instance this prints rows from 1 to 10 of columns 2 and 4:

```
brexit[1:10, c(2,4)]
```

You can also index a single column by using in combination the extract operator `$` and indexing. In this case, you obviously only have to pass one dimension, because that's the only dimension vectors have:

```
brexit$percent_leave[1]
brexit$percent_leave[1:10]
```

A very important thing that indexing allows you to do is selecting values of a variable *conditional* on the values of another variable, for instance if I wanted to know what the value of `percent_leave` is for the local authority of Oxford, I would run...

```
brexit$percent_leave[brexit$area == "Oxford"]
# Note the double equal and the quote marks!
```

What is going on 'behind the scenes' is that R is evaluating TRUE/FALSE for the logical statement `brexit$area == "Oxford"`, and then it's returning the value of `percent_leave` **for the row in which the statement is TRUE** (i.e. for the row corresponding to the Oxford local authority). Some more examples of how we can index our variables:

```
brexit$percent_leave[brexit$region == "Wales"]
#index by a condition that applies to multiple observations

brexit$percent_leave[brexit$median_age > 40]
# index by a continuous variable

mean(brexit$percent_leave[brexit$median_age > 40])
# combine indexing with mean() function

mean(brexit$percent_UK_born[brexit$median_age > 40], na.rm = T)
# within mean(), remember 'na.rm = TRUE' if your variable has missing values
```

In the homework, you will find some more example of operators and functions that can be used in combination with indexing.

## 8. Creating new variables

We can create new variables in an existing dataframe using a combination of the extract operator `$` and the assign operator `<-`. When we create new variables, it is good practice to start always with an empty variable-vector, assigning `NA` (missing) to all entries. Then we can fill it up with mathematical operations or logical statements from the variables we already have.

The logic is exactly the same as we learnt when we summed our `one_to_ten` vector with our `decimals` vector and got a new numerical vector as output, or when we got a logical `TRUE/FALSE` vector by asking R whether the elements of `one_to_ten` were larger than six. Only, this time we are attaching the vector output to an existing dataframe as a new variable.

Let's try for instance to create a new variable in our `brexit` dataframe for turnout – let us call it, imaginatively, `turnout` – defined as the ratio between the existing `total_votes` variable and the `electorate` variable:

```
brexit$turnout <- NA
#creates an empty variable
brexit$turnout <- brexit$total_votes/brexit$electorate

summary(brexit$turnout)
```

What is happening behind the scene is that R is taking the first value of `brexit$total_votes` (the number of votes cast in the local authority in row 1) and dividing it by the first value of `brexit$electorate` (the number of people on the electoral roll in the local authority in row 1), repeats this for all 380 rows, and then spits out a vector of length 380 which is attached to our dataframe.

We can combine this procedure to create new variables with indexing. For instance, let's say we want a character variable named `winner` that takes the character value "Leave" when leave vote is greater than 50%, and "Remain" when leave vote is smaller than 50%. We would run:

```
brexit$winner <- NA #creates an empty variable
brexit$winner[brexit$percent_leave > 50] <- "Leave"
brexit$winner[brexit$percent_leave < 50] <- "Remain"

# How many local authorities voted Leave? How many voted Remain?
table(brexit$winner)
```

**9. Saving your progress** As we have created new variables in this dataset, we may want to save the dataset for future use. We can create a new `.csv` (comma-separated-variables) file in the working directory using the function `write.csv()`. Careful: if you save the dataset as `brexit.csv`, it will overwrite the file you already have; so it might make sense to give it a different name, like `brexit_new.csv`.

```
write.csv(brexit, "brexit_new.csv", row.names=FALSE)

# it is advisable to specify row.names=FALSE, otherwise the function will
# save the row numbers (1,2,3...380) as an additional variable.
```

**Homework**

- Create a vector `even_numbers` of all even numbers from 0 to 100. Create a vector of their squares and call it `squares`. Create a vector of their square roots, and call it `roots`. Now bind the three vectors in a new dataframe with three columns.
- When we added `primes` and `one_to_ten`, we found that, as both vectors are of length 10, we got as a result another vector of length 10, where the first element is the sum of the first elements of the two original vectors, the second element is the sum of the second elements, and so on. What happens when we try to add vectors of different length? Go back to our vector `one_to_ten` (if needed, create it again), and let's try to add to it vectors of different length:

```
one_to_ten + 1000
one_to_ten + c(1000,2000)
one_to_ten + c(1000,2000,3000)
one_to_ten + c(1000,2000,3000,4000,5000)
```

Can you guess what's the rule? (Explanation in the homework solutions).

- Go back to the `brexit` dataset. Find out: (1) what's the mean percentage of Leave vote in a London constituency (variable `region`)? (2) what's the average Remain vote in areas classified as 'Predominantly Rural' (variable `area_type`; mind: there are missing values, so remember `na.rm = TRUE`)? (3) what's the total number of Leave votes (`leave_votes`) across all local authorities? What's the total number of Remain votes (`remain_votes`)?

- Using indexing, find out what's the highest value of `median_age` in the dataset. Then find out which local authority corresponds to that value (i.e. what's the 'oldest' local authority in our dataset).
- The chunks of code below illustrate the use of some key operator that can be helpful with indexing. Follow along with the code and then try to answer the final questions.

++ We can use the OR operator `|` to condition on multiple criteria (e.g. select the rows that take the values 'North East' OR 'North West' in the variable `region`)

```
brexit$area[brexit$region == "North East" |
                        brexit$region == "North West"]

brexit$percent_leave[brexit$region == "North East" |
                        brexit$region == "North West"]
```

++ We can do the same thing with the %in% operator, passing as argument a vector of values for which the logical statement can be true. The code below effectively asks R to select rows that take in the variable `region` any of the values in the vector c("North East", "North West"). This solution is usually more synthetic than the OR operator.

```
brexit$percent_leave[brexit$region %in% c("North East", "North West")]
```

++ The AND operator '&' also conditions on multiple criteria: but in this case BOTH must be true. Like the OR operator, it can be used within one variable or across multiple variables.

```
brexit$percent_leave[brexit$median_age > 35 &
                     brexit$median_age < 45]

brexit$percent_leave[brexit$region == "South East" &
                         brexit$area_type == "Predominantly Urban"]
```

++ Finally, values like `NA` are special values, so if you wanted to ask R to tell you something about the observations that have missing variables you cannot treat them as any other value. You need to use the `is.na()` function. For instance, if I wanted to know which local authorities are missing their values for the variable `percent_UK_born`, I have to pass:

```
brexit$area[is.na(brexit$percent_UK_born)]
```

- Create a new variable `result`, which takes four possible values: "Strong Leave" if Leave vote is 60% or over, "Weak Leave" if Leave vote is between 50 and 60%, "Weak Remain" if Leave vote is between 40 and 50%, and "Strong Remain" if Leave vote is under 40%. Remember to use the quote marks, as here you are working with character values. How many local authorities fall in each category? What's the mean percentage of residents who identify as white (variable `percent_white`) in each of the four categories?