Tutorial 1: Your first steps in R

Carlo Knotz

Table of contents

Introduction

In the previous tutorial, you have learned how you can install `R` and `RStudio` and how you install add-on packages.

But even if you managed to get everything to work, it might still have felt strange to work with computer code.

The aim of this tutorial is to help you get familiar with `R` and more used to writing code. This means that you will do a few simply warm-up exercises and learn about the main logic of `R` and `R` code.

> Tip
>
> *Hvis du ønsker å lese en norsk tekst **i tillegg**: "Lær deg R", Kapittel 2 & 4.1.1*

Writing code: Some warm-up exercises

Calculating with R

One thing that can help you get familiar with R is to simply use it as a calculator — which you can do easily because, in essence, R is just a very fancy, very powerful pocket calculator.

To do that, click into the *Console* window so that a small vertical blinking line ("cursor") appears behind the > symbol. Then type `1 + 1` and hit the Enter-key.

The result should look like this:

```
1+1
## [1] 2
```

You can of course also do much more complicated calculations such as:

```
(12*78)/(0.5-7000)+42.541-8*98
## [1] -741.5927
```

You can do all common mathematical operations that you know from (high) school in R:

- `+` adds two numbers
- `-` subtracts one number from another
- `*` multiplies two numbers
- `/` divides one number by another
- `(` and `)` are *parentheses* (to separate out a specific part of the calculation)
- `^` means "raise to the power of" — `2^2` means `2*2` (or "two squared"), `2^3` means `2*2*2`, and so on

Working with text (and your first error message)

Clearly, `R` can handle numbers quite well — which is not surprising, given that `R` is a *statistical* programming language. But `R` can also handle text, as long as the text is entered correctly.

To see what this means, let's have a look at what happens when you handle text *in*correctly: type the following into the *Console* and hit "Enter":

```
Hello world!
```

You should now receive your first **error message** (the first of many…sorry!): `Error: unexpected symbol in "Hello world"`

Here is what happened: When you enter letters or words into the `R` *Console*, `R` thinks that you are entering a **command** ("function"). It then looks into its internal library of commands and, if it does not find a command that corresponds to the word you entered, it returns an error message. `Unexpected symbol` basically means *"I don't understand what you want from me."*

You can tell `R` to treat something as text and not as a command by placing it within quotation marks. See what happens when you enter the following into your *Console*:

```
"Hello world!"
```

Saving stuff into "*Objects*"

While `R` *can* be used simply as a calculator to add two numbers, you will normally use it to do more complex calculations and transformations with large datasets — i.e., hundreds, thousands, or sometimes millions to billions of numbers. And in this usual scenario, you can obviously not enter all these numbers by hand. Plus, the results of statistical analyses ususally include several numbers and other pieces of information, and this can be hard to handle unless you can save these results somewhow for later use.

In `R`, you can save *stuff* — numbers, text, and other things — into *objects*. Objects are, in essence, little containers that store whatever you put into it and that float around in your work environment. Once you have saved something in an object, you can use it again later.

The assignment operator (<-)

To save something in an object, you use the **assignment operator**: `<-` (the "smaller than" sign, `<`, plus a dash, `-`)

For example, `x <- 5` tells `R` to "store `5` into the object `x`" (or, put differently, "assign the value of `5` to `x`").

See for yourself what happens when you type the following into your *Console* and hit enter:

```
x <- 5
```

You will now not see any output in the *Console* — instead, you should see an item appear in your work environment (on the upper right side of your screen). You see that the value of 5 has indeed been stored as the `x` object.

Try storing another number as `y`, for example:

```
y <- 7
```

As mentioned before, you can also store text. Run the following in your *Console*:

```
z <- "statistics is pointless"
```

You should now see the `z` object containing the phrase "statistics is pointless" appear in your work environment.

Shortcutting the assignment operator

When working with `R`, you will use the assignment operator a lot. Basically all the time.

It therefore makes sense to add a *keyboard shortcut* specifically for the assignment operator so that you do not have to twist your fingers into a knot every time you want to store something in an object.

This is easy to do in `RStudio`: Go to "Tools" in the taskbar at the top and choose "Modify keyboard shortcuts"; use the little search/filter field to search for "assignment". The entry "Insert assignment operator" should appear. Use the existing shortcut, or change it to one that suits you.

Using stored objects

What if you want to re-use the objects you stored? Easy. You just call them up by typing their name into the *Console*.

For example, to retrieve the number you stored in the `x` object, you just type `x` into the *Console* and hit "Enter".

Try retrieving all the stored values, `x`, `y`, and `z` (one at a time!).

You can also use these stored items, for example in calculations. Try it out, type the following into your *Console* and hit "Enter":

```
x + y
```

Obviously, mathematical operations only make sense if you work with stored *numbers*. This does not work with text. (Feel free to try it anyways! See what happens when you run `y + z`.)

Storing multiple numbers or words

Now comes a very important step: We are moving from storing single pieces of information (one "datum") to storing multiple pieces of information ("data").

To store multiple items together you use a very important **command** or "function": `c()`

`c()` stands for "combine" — you use it to combine pieces of information ("data points") in a single object. The `c()` function is easy to use. You just add the things you want to store *within the parentheses, separated by commas.*

To see how it works, run the following code to store the sequence of the numbers 1 through 4 into the object `a` (or, "assign" them to the object `a`)

```
a <- c(1,2,3,4)
```

You can also store a sequence of words as long as you put them into quotation marks and separate them with commas. Give it a try by running the following code:

```
b <- c("statistics","is","pointless")
```

On names for `R` objects

You have just learned about objects — the little containers in which you can store different values. When working in `R`, you will usually work with many different objects: Your main dataset will be one object in your *Environment*, but you may also be working with more than one dataset at a time or have other things stored in different objects. **This means it is important that you always know what each object is — you have to give them useful and informative names!**

You can choose essentially freely how to name your objects — just like you can freely choose how you name any Microsoft Word document you are working on. For example, you could name a Word Document *"BST290_CoursePaper_draft15.docx"* or *"CoursePaper_BST290_draft23.docx"*, or *"StupidStatsCourse_Paper_almostdone_Version56.docx"*, or any other name you want. The same applies to your `R` objects: You can name them, in principle, in any way you like.

But: Do make sure you always pick names that are informative so you don't get confused! For example, once you work with a full dataset, you should give it a meaningful name like `my_dataset` and not just `x` or `y`! You should also avoid naming objects like important commands — so, don't call your dataset object `mean` because there is a command in `R` to calculate the mean ("average") that is called `mean()`. You obviously don't want any confusion there…

Briefly on commands ("functions")

A bit earlier, you used the `c()`-command or `c()`-**function** to combine sets of items into an object.

Functions are a central component of the `R` language. You can think of functions as the `R` language's *verbs* — the "do stuff" words. You can tell that something is a **function** if it has parentheses: `()`.

For example,

- `c()` is the command to tell `R`: "combine the following items together"
- `mean()` is the command to tell `R` to calculate the mean ("average") of a set of numbers
- `download.file()` is the command to tell `R` — you guessed it — to download a file from the internet
- `rm()` is the command to tell `R` to remove ("delete") an object from the work environment

Functions work by the following principle:

- The name of the function — the text before the parentheses (`mean`, `download.file`,…) — specifies **what operation you want to be done**.
- Within the parentheses, you specify the object(s) that the operation should be **done with** (and, if available, any options). In the official `R` lingo, the content within the parentheses is called *arguments*.

Let's look at two functions in action (run these in your *Console*, one at a time):

```
bodyheight_data <- c(187,156,198,183,175,171)

mean(bodyheight_data)
```

```
[1] 178.3333
```

Here, you use two different functions:

- You first use `c()` to store a collection of values, measurements of the body heights of a fictional group of people, as `bodyheight_data`.
- Then you let `R` calculate the average body height in this group using the `mean()` function.

There are obviously many, many, many more functions in the `R` language. No reason to worry, you will learn them one at a time.

Another look at your stored objects

If you take another look at your work environment you should see the different objects you created so far, among them `a`, `b`, and `bodyheight_data`. You also see their respective contents, and then some gibberish in between:

- `num [1:4]` in the case of `a`
- `chr [1:3]` in the case of `b`
- `num [1:6]` in the case of `bodyheight_data`

Here is what this means: The numbers in brackets (`[1:4]` or `[1:3]`) tell you that the objects contain multiple items. As you probably remember, `a` contains four items (the numbers 1 through 4), and `b` contains three items (the three words "statistics", "is", and "pointless"). `bodyheight_data`, finally, contains six different numbers.

The abbreviations `num` and `chr` inform you about the type of data that is stored:

- `num` stands for *numeric*, which means "contains numbers"
- `chr` stands for *character*, which means "contains characters" (i.e., "single letters or words" or simply "text")

In `R` "lingo", `a` and `b` would be called **vectors** — which is a fancy way to say "collections of things". (Unfortunately, `RStudio` displays them under "Values" in the work environment, which might be confusing.) There are also other types of vectors than `numeric` and `character`; you will learn about them in a later tutorial.

As with the other objects, you can retrieve vectors by typing their name into the *Console* and hitting enter. Give it a try:

- Retrieve the `a` vector by typing `a` into your *Console* and pressing Enter.
- Now do the same with `b`.

Writing and saving your code in a *script file*

Congrats, you have already learned quite a bit about working with `R`! You know:

- What *objects* are and how you can store them with the *assignment operator* (`<-`)
- What *vectors* (a type of object) are, and that there are different kinds of vectors
- What *functions* ("commands") are and how you use them

You may also note that both your *Console* and your *Work Environment* start to get a bit crammed — and you may even have lost track of why you created the different objects and what code you used to create them.

This is a common problem — statistical analyses are usually complex, and it is easy to lose track of what is going on after a while.

But there is also a solution: Writing and saving your code in a dedicated *script file*! This part of the tutorial will show you how to do this.

From single commands to script files

So far, you have just entered a few individual functions into the `R` *Console* and then looked at the results. This is fine for a very first introduction, but later on — when things get more serious — you will write many more and also more complex functions and you will want to run them directly after each other. Also, you will usually work on any single data analysis project for a longer period of time. For example, you might do all your data management and cleaning on one day and then do the actual data analysis on the next day. In that case, you obviously do not want to have to re-enter all the code from the first day when you start again on the following day.

This means you need some place where you can write longer sequences of code and then save everything so you do not have to start from the beginning every time you need to interrupt your work. This is why you use **script files**.

Script files contain all the code that you write for a specific data analysis project.

Writing and running a script file

To make this more concrete, let's **open a script file on your own computer**: In the taskbar at the top of your screen, go to "File" -> "New File" -> "R Script" (alternatively, press "Ctrl+Shift+N" on Windows or "Command+Shift+N" on Mac).

The left-hand side of your screen should now split in two: The *Console* moves to the bottom half and an entirely empty file with a blinking cursor appears in the top half. *The new file at the top is your script file.*

First, save the file ("File" -> "Save as") and give it an informative name (e.g., `r_tutorial1.R`).

Then write the following code to generate two new vectors into your script file:

```
bodyweight_data <- c(85.3,75.6,114.2,56.9)

fruit <- c("banana","apple","orange","cherry")
```

Now you have some code (which should like in the image below) — but you may wonder how you run it? You could of course copy-paste it line by line into the *Console* below, but that would be annoying.

Instead, select the code either with your mouse or with the Shift + arrow keys — *just like you would select text in a Word document.* The result should look like in the screenshot below:

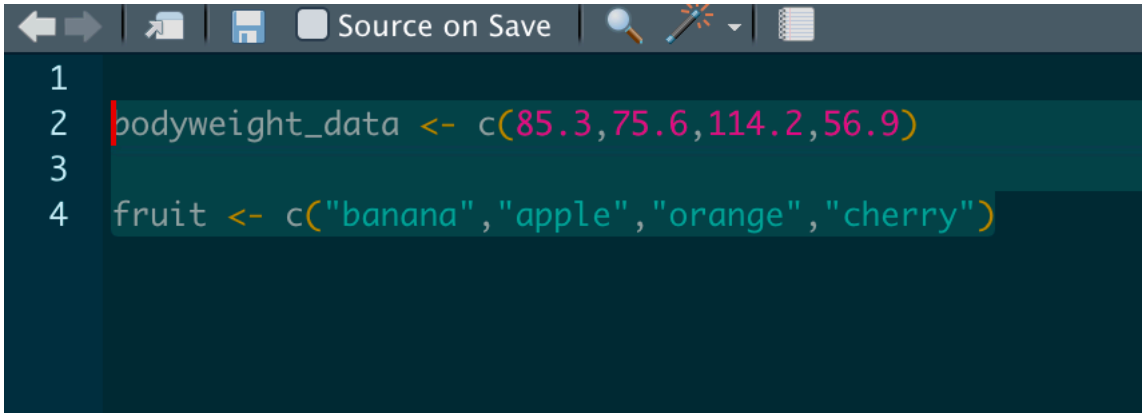

Figure 1: Selected code in a script file

To run this code, move your mouse to the top right corner of the script file (not the entire window!) — *there is a little button with a white square and green arrow that says "Run".* Click on that button to run your code.

You will see that the commands you selected to run get "transferred" to the *Console* and then executed. **This is the normal way to work in R: You write (and save!) code in your scriptfile, and you also run it from there.** The *Console* is only used for small, less important things (e.g., when you quickly want to look at a stored object or want to try out something).

You can also create a **keyboard shortcut** to run code. **This is strongly recommended because it will make your life easier.** To find out what the current shortcut to run selected code is, navigate again to the relevant menu ("Tools" -> "Modify Keyboard Shortcuts"; search for "Run current line or selection"; use the existing shortcut or modify it as desired).

Comments in a script file

Sometimes you might want to add notes or comments to your code, for example to explain to others (or your future self) what a particularly complicated piece of code does. Obviously, you then need to tell `R` to ignore these comments or you will get an error message.

You can add comments in a script file with the hash symbol (`#`). `R` will then ignore everything that comes after `#`. For example:

```
# mean(bodyweight_data) <- R ignores all of this

# Some comment <- R ignores this too

mean(bodyweight_data) # <- this code is run, but everything after the hash is ignored
```

Keeping your code neat and tidy

It is important that you keep your script files organized and easy to read — especially for yourself. You should be able to open the scriptfile for an analysis that you did two years ago and understand, at least after a bit of reading, what you did and how you did it. This means:

- Use comments to provide a title for each file plus relevant information and explanations for each bit of code
- Loosen it up: Add empty lines between code 'chunks' that belong together

For example, this is how you could organize a simple scriptfile (this code will not make sense to you now, but you should be able to understand most of it by the end of this course):

```
#########################
# Example analysis script
#########################

# Carlo Knotz (March 17, 2022)

library(tidyverse) # add-on package for data management & visualization

# Data management
#################

# Downloading data
cpds <- haven::read_dta("https://www.cpds-data.org/images/Update2021/CPDS_1960-2019_Update_2

# Data cleaning (selecting relevant variables, changing name of one)
cpds %>%
  select(country,year,openc,realgdpgr,debt_hist,gov_left1,gov_right1) %>%
  rename(growth = realgdpgr) -> cpds


# Descriptive graph
###################

# Average growth rate, by country
cpds %>%
  group_by(country) %>%
  summarise(growth = mean(growth, na.rm = T)) %>%
  ggplot(aes(y = reorder(country,growth), x = growth)) +
    geom_bar(stat = "identity") +
```

```
    labs(x = "Average economic growth rate (%)",
         y = "") +
    theme_bw()

# ...and so on...
```

Vectors, variables, and datasets

Vectors & variables

You have previously learned that R stores collections of data points as *vectors*. But you may also remember from the lecture and Kellstedt & Whitten that we, as social scientists, have also another name for collections of data points: **variables**.

Measurements of a variable are really nothing else than collections of data points (usually numbers, but sometimes also text). For example, the (fictional) set of body height measurements we created earlier could be measurements of a variable ("body height") that comes from a research project.

```
bodyheight_data
## [1] 187 156 198 183 175 171
```

The important point to remember: In R, variables are represented as vectors — as collections of data points.

Datasets

When you do a statistical analysis, you usually work with more than one variable (i.e., vector) because you usually want to know how one or more variables are related to another variable (as explained in Chapter 1 in Kellstedt & Whitten). To be able to work with two or more variables at the same time, they need to be "tied together" — and *this is, in essence, what a dataset is: Several variables/vectors tied together.*

To see what a dataset in `R` looks like, you can open one of the many "toy" datasets that are built into `R`: the "Violent Crime Rates by US State" (`USArrests`) dataset.[1]

To view this dataset, you just have to type `USArrests` into the *Console* and hit Enter. The result should look like this:

```
USArrests
##               Murder Assault UrbanPop Rape
## Alabama         13.2     236       58 21.2
## Alaska          10.0     263       48 44.5
## Arizona          8.1     294       80 31.0
## Arkansas         8.8     190       50 19.5
## California       9.0     276       91 40.6
## Colorado         7.9     204       78 38.7
## Connecticut      3.3     110       77 11.1
## Delaware         5.9     238       72 15.8
## Florida         15.4     335       80 31.9
## Georgia         17.4     211       60 25.8
## Hawaii           5.3      46       83 20.2
## Idaho            2.6     120       54 14.2
## Illinois        10.4     249       83 24.0
## Indiana          7.2     113       65 21.0
## Iowa             2.2      56       57 11.3
## Kansas           6.0     115       66 18.0
## Kentucky         9.7     109       52 16.3
## Louisiana       15.4     249       66 22.2
## Maine            2.1      83       51  7.8
## Maryland        11.3     300       67 27.8
## Massachusetts    4.4     149       85 16.3
## Michigan        12.1     255       74 35.1
## Minnesota        2.7      72       66 14.9
## Mississippi     16.1     259       44 17.1
## Missouri         9.0     178       70 28.2
## Montana          6.0     109       53 16.4
```

[1] To see what other "toy" datasets you have available, run `data()` in your *Console*.

```
## Nebraska          4.3     102       62 16.5
## Nevada           12.2     252       81 46.0
## New Hampshire     2.1      57       56  9.5
## New Jersey        7.4     159       89 18.8
## New Mexico       11.4     285       70 32.1
## New York         11.1     254       86 26.1
## North Carolina   13.0     337       45 16.1
## North Dakota      0.8      45       44  7.3
## Ohio              7.3     120       75 21.4
## Oklahoma          6.6     151       68 20.0
## Oregon            4.9     159       67 29.3
## Pennsylvania      6.3     106       72 14.9
## Rhode Island      3.4     174       87  8.3
## South Carolina   14.4     279       48 22.5
## South Dakota      3.8      86       45 12.8
## Tennessee        13.2     188       59 26.9
## Texas            12.7     201       80 25.5
## Utah              3.2     120       80 22.9
## Vermont           2.2      48       32 11.2
## Virginia          8.5     156       63 20.7
## Washington        4.0     145       73 26.2
## West Virginia     5.7      81       39  9.3
## Wisconsin         2.6      53       66 10.8
## Wyoming           6.8     161       60 15.6
```

You see that the dataset contains statistics about arrests for violent crimes in all U.S. States (in 1973).[2]

Note that:

- Each row is a state (e.g., Alabama, Alaska, etc.);
- Each column is a variable that records the arrest rate for a given crime and the share of the population living in big cities (`UrbanPop`).

Importantly, each of the variables by itself is just a sequence of numbers — or a vector — but, together, they form a dataset. And this is also how a dataset should ideally be organized: Each row is an observation, each column is a variable. A dataset that is organized like this is called a *tidy* dataset.[3] In `R`, datasets are stored as so-called `data.frames`.[4]

[2] Run `?USArrests` in the *Console* to open the documentation of the dataset, which will give you more detailed information.

[3] See also Wickham, H. (2014). Tidy data. *Journal of Statistical Software*, 59(10):1–23.

[4] There are also other types of dataset-objects such as the `tibble` (https://tibble.tidyverse.org/), but they all look and work more or less the same way.

The datasets you will work with during this course will be organized like this — but, in real life, datasets are often messy and need to be reorganized before you can really work with them. This is why this course includes a part on Data Management.

Creating a data.frame yourself

When doing a data analysis in R, you will normally work with large to very large datasets that you import in one go — typing them in by in is obviously not an option then.

But you can create datasets by yourself, and doing so is a good way to understand better how to work with this type of object.

You can create your own `data.frame` with the `data.frame()` function. When you use this function, you go variable by variable – or column by column: You first type the name and value of one column, and then repeat that for all other columns.

For example, let's assume you had collected a tiny dataset about some local students at UiS. This dataset has four observations (i.e., four rows) and three variables (columns). The rows are different four different students, and the variables include `name`, `age`, and `hometown`. The values are as shown below and we save the dataset as `studata`.

This is how you create a dataset of these four students (you should write this in your new *scriptfile*):

```
studata <- data.frame(name = c("Janne","Tore","Siri","Ola"),
                      age = c(23,21,19,22),
                      hometown = c("Stavanger","Oslo","Tromso","Bergen"))
```

When you take a closer look at the code, you see that each variable name (`name`, `age`,...) is followed by a set of values that together form the variables. Also, notice that values that are text (the names and hometowns) are put in quotation marks but not those that are true numbers (the students' ages).

When you now run this code (as described earlier), the `studata` object should appear under `Data` in the *Work Environment*. If you then click on the little blue circle with the white triangle in it (next to `studata`), you will also get an overview over the contents of the dataset: The variables and what types they are.

You can also see the structure of your new dataset more clearly if you print it out in the *Console*. To do that, you simply type the name of the stored dataset into the *Console* and press "Enter":

```
studata
##    name age  hometown
## 1 Janne  23 Stavanger
## 2  Tore  21      Oslo
## 3  Siri  19    Tromso
## 4   Ola  22    Bergen
```

You can directly see the structure of the dataset: Observations (students) in rows, variables (`name`, `age`,…) in columns.

The end!

First task: take a quick break! You just took your first steps in a statistical programming language! These first steps might have been a bit wobbly and may have felt weird — but this is only the beginning. Before you know it, writing and running code in `R` will feel a lot easier and more intuitive!