# Lab 01: Introduction to RStudio and R

Maximilian Haag Constantin Kaplaner

17.10.2022

## Contents



## What is this?

This is an `Rmarkdown` notebook document. It can be used to combine regular text, like this, and `R` code. This means, that you can, e.g. take your own notes right here in the editor while working through the lab. We will go into detail on this in the next session. For now, here is all you need to know to work with this document:

- Save your scripts with the file extension `.R`
- While this file is not a pure `.R`-Script but a combination of markdown formatted text and `R` code you don't have to mind for now.
- The code in this notebook is contained in the grey boxes that start with "`{r}"" and end with ""`. The code goes right in between.

(The following only applies if you open the `.Rmd` file notebook in RStudio, not to the compiled `.pdf` version)

- To execute these "code chunks" you can either click the little green 'play' button in the upper right hand corner of every chunk
- or you can select the code you want to execute and press Ctrl+Enter (Windows/Linux) or cmd+Enter (Mac)
- If you put your cursor in a line of code without selecting anything, the commands above will simply execute that line of code

If you want to know more about markdown and its formatting options, you can take a look at the Markdown Cheatsheet.

# Getting started with RStudio and R

## R as a fancy, complicated calculator

Since R is a statistical programming language first and foremost, it can do a lot of maths and statistics right out of the box. Let's try some.

Press the little green 'play' icon in the upper right corner of the code chunks below and examine the outputs:

```
1 + 1
```

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

```
2 - 3
```

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

```
4 * 5
```

```
## [1] 20
```

```
2 ^ 2
```

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

```
4 / 2
```

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

```
2 ^ (1 / 2)
```

```
## [1] 1.414214
```

That's pretty basic. How about something a little more wild?

```
((2 + 2) * 8) / 4
```

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

Note that, just as your regulator calculator, `R` respects brackets when evaluating a term. And, just as your regular calculator, it gets a little lost when you don't properly specify your mathematical expression. Can you spot what went wrong here?

```
((2 + 2) * 8 / 4
```

```
## Error: <text>:5:0: unexpected end of input
## 3:
## 4:
##   ^
```

Ok, now for some more maths:

```
sqrt(2)
```

```
## [1] 1.414214
```

```
log(10)
```

```
## [1] 2.302585
```

```
sum(c(2,2))
```

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

and some statistics:

```r
mean(c(2,2,2,2,10,9))
```

```
## [1] 4.5
```

```r
mean(c(1+1,2,2,2,10,9))
```

```
## [1] 4.5
```

Here, we're calculating the mean of a series of numbers (2,2,2,2,10 and 9). Note that we can also replace a number with a mathematical term (2nd expression) and `R` will automatically evaluate the term first before it calculates the mean.

## Variables and functions

So now, that we've seen that `R` can do maths (wow!), you might ask yourself, whether all this fuzz is really necessary just to use a calculator. Of course, `R` can do more than that! For example, it can remember stuff.

Say, we wanted to take our mean from above and use it at a later point in time (e.g. to compare it to another mean). For that, we need to tell `R` to store it for us:

```r
first_mean <- mean(c(2,2,2,2,10,9))
```

Can you see what's different here? First of all, we're not getting any output from out code chunk. Second, we're using `first_mean <-`. This expression tells `R` to store (`<-`) our mean in a *variable* called `first_mean`.

@TODO: RStudio environment upper right corner. @TODO: rm() objects

Note that we can choose any name we would like as long as it does not contain spaces or special characters (or start with a number). So, if we feel like it, we can also store our mean in a variable called `tiger`. And we can store the value of `tiger` in `lion`. Simple as that.

```r
tiger <- mean(c(2,2,2,2,10,9))

lion <- tiger
```

A trick we can use in RStudio to save us some typeing when we're re-using our variables is code completion. Try it out for yourself, type `t` in the code chunk below and press the `Tab` key:

While cool variable names are fun and you certainly will find them in the wild, it's usually good to choose a descriptive and informative variable name. This will help others (and future you!) understand your code. So here, using `first_mean` for our first mean is an acceptable choice. If, e.g., we wanted to calculate the avg. height between Constantin and Max, we could name our variables like this:

```r
height_constantin <- 185

height_max <- 183

avg_height <- mean(c(height_constantin, height_max))

print(avg_height)
```

```
## [1] 184
```

We've additionally done two new things here: We gave two variables to `mean` and we printed out the result.

Note that we can't just name our mean of heights `mean`[1]. Why? Because the name 'mean' is already taken by the *function* we use to calculate our mean of heights, the `mean`-*function*.

A *function* is like a little program on its own that usually takes some input, does something with it, and returns the results. Take, e.g., the `mean`-function, it

[1] Actually, we can but we really should not!

- takes a series of numbers as an input[2]
- sums up the numbers
- divides them by the number of numbers in the series
- returns the result

If we translate this into R it would look this:

```
my_own_mean <- function(numbers){

  calc_mean <- sum(numbers) / length(numbers)

  return(calc_mean)

}
```

We can call this function just like R's own mean-function:

```
my_own_mean(c(height_constantin, height_max))
```

```
## [1] 184
```

```
mean(c(height_constantin, height_max))
```

```
## [1] 184
```

There are many functions that are already integrated into R, like the mean- and print-functions we used above, but you can also write your own function that do what you tell them to (we might get to that later on in the course).

## Commenting and naming

When you're writing your own code or you're reviewing somebody else's code, it's often not clear from the beginning *why* your're doing certain things the way you're doing them. Take our example from above:

```
height_constantin <- 185

height_max <- 183

avg_height <- mean(c(height_constantin, height_max))

print(avg_height)
```

```
## [1] 184
```

We already established that this code creates two variables with Constantin's and Max' height, calculates the mean, stores in another variable and then prints out the result to us. However, if you've never heard of R or you don't know programming languages at all, that might not be obvious to you. So to remedy that we can *comment* code. In R, comments always start with #. When you type # in an R script everything after dash until the end of the line will be ignored by R and only be visible in the code itself but not in the output.

So let's take another look at our height calculation and make it accessible to others:

```
# store Constantin's height
height_constantin <- 185

# store Max' height
height_max <- 183+1 # this was after I grew another cm
```

[2]Note that these numbers are themselves wrapped in the c-function that can be used to combine variables. We'll cover that in the next session.

```r
# calculate the mean of both
avg_height <- mean(c(height_constantin, height_max))

# print out the result
print(avg_height)
```

```
## [1] 184.5
```

Note that there's two types of comments in here: Those that take up the whole line and one that's in the same line as some code. Both of these types can be used interchangeably in your code.

We can use comments to document our code for ourselves and others. Or, we can store any other information that we'd like to remember in there like a ToDo list for our coding project.

```r
# TODO
# - add another cm for Max
# - remove one cm from Constantin

# store Constantin's height
height_constantin <- 185

# store Max' height
height_max <- 183+1 # this was after I grew another cm

# calculate the mean of both
avg_height <- mean(c(height_constantin, height_max))

# print out the result
print(avg_height)
```

```
## [1] 184.5
```

Naming conventions, i.e. how we name our variables, function or files, are just as useful when it comes to documenting code. We've already seen some naming conventions above (descriptive names, `tiger` vs. `avg_height`). Others are imposed on us by the language. E.g., names can't contain spaces:

```r
height max <- 183
```

```
## Error: <text>:2:8: unexpected symbol
## 1:
## 2: height max
##           ^
```

So far, we've used an underscore, `_`, to replace spaces in our names. There's other ways of doing this like CamelCase naming. In CamelCase, we would name our variable `HeightMax`. From a functional perspective, this does not change the way our code works or performs, but it helps us read and understand our code. For this reason it's usually best to stick to either of the two and avoid things like `heightmax` to maintain readability.

*TIP*: There's multiple naming conventions and style guides for code, e.g. Google's R Style Guide or Hadley Wickham's Style Guide. It's that necessary that you follow any of them to the bone, but it's often helpful to choose a naming conventions, like `_`, and stick to it.

## Some examples of doing (familiar) things in `R`

@TODO plot

## Packages

What's cool about using a programming language instead of your calculator to do things for you is that you're usually not the first person to solve a given problem. For that reason, developers take their proven solutions and sets of function for a certain problem and pack them in to a 'package', which other can the download and use for their problems.

I, for example, am really bad at remembering jokes when I need to tell one, so I'm glad that someone took it upon themselves to solve that problem for me. First, install the `dadjoke` package:

```
install.packages('dadjoke')
```

Now, get it to tell you a joke:

```
dadjoke::dadjoke()
```

```
##
##
## You know, people say they pick their nose.
##
## But I feel like I was just born with mine.
```

Note that here we specify the package name, `dadjoke` followed by `::` and then the function name. This tells `R` 'out of the `dadjoke` package, use the function `dadjoke()`'. Note that in case of the `dadjoke` package the package and the function carry the same name. Bad example. Here's another one:

```
install.packages('cowsay')

cowsay::say('Hello, I am learning R!', by='cow')
```

Writing `packagename::` can be useful when we're using a lot of different packages and functions and we want to avoid any confusion with regard to which function is used. Additionally we may use it if we just want to use a single function out of a very big package. In any other case, we can simply *load* the package into our environment using the `library`-function. Then, we don't need to specify the packagename when calling one of it's functions:

```
library(cowsay)

say('I wrote this without specifying a packagename!', by="cat")
```

```
##
##  --------------
## I wrote this without specifying a packagename!
##  --------------
##     \
##       \
##         \
##             |\___/|
##           ==) ^Y^ (==
##             \  ^  /
##              )=*=(
##             /     \
##             |     |
##            /| | | |\
##            \| | |_|/\
##       jgs  //_// ___/
##                \_)
##
```

*TIP*: When you type the name of the package followed by `::` you can again make use of RStudio's code completion feature to get an overview of the functions contained in a package.

## Where can I get help if I'm stuck?

At this point, you may think you have no idea what you're doing or what this is good for. Lucky for us, programmers and researchers also often have no idea what they're doing. This is where online forums come into play, specifically *Stack Overflow*. *Stack Overflow*

@TODO Error msg as example for Googling

@TODO RStudio R Help

## 'Data' in `R` and the `tidyverse`

The `tidyverse` is collection of packages that has become quite popular among `R` users because it simplifies a lot of operations and code when working with data, which makes it particularly suitable to new `R` users:

> The tidyverse is an opinionated collection of R packages designed for data science. All packages share an underlying design philosophy, grammar, and data structures. (https://www.tidyverse.org/)

In this course, we will rely on packages associated with the tidyverse for a lot of exercises. While the `tidyverse` makes a lot of stuff easier for us, it also has some particularities and rules that we will examine below.

### Working with data

@TODO Excursion, dataset, dataframe (dataframes erst nächste session) @TODO show easy example datasets @TODO View function etcetc

For now, it is suffcicient that you have a rough understanding of how our data is strucutred here. In the next session, we will take a closer look at data(sets) in `R`.

### Installing the `tidyverse`

To install all of the `tidyverse` packages, run (this may take a while)

```
install.packages("tidyverse")
```

As an alternative, it is sufficient for now if you just install the `dplyr` package:

```
install.packages('dplyr')
```

### The pipe operator

@TODO

Note: If this is confusing to you, don't worry! With practice you will become more and more used to thinking about data in this way

Note: You may stumble across another type of pipe operator: `|>`. This is `R`'s own pipe operator, which has recently been introduced. A lot of times you can use both interchangeably, but there are some differences in how they work, so for now it's best to stick to `%>%`.

## Data-wrangling operations

The `tidyverse` offers five main operations to work with your data: `mutate()`, `select()`, `filter()`, `summarise()` and `arrange()`

## Some more examples using the `tidyverse`

*TIP*: If you are looking for a more comprehensive introduction to the `tidyverse`, take a look at Hadley Wickham's free book 'R for Data Science'