Lab 1: Introduction to R

David Berger

2023-08-25

Today's lab gives you some basic info on R usage. The intention is to give a very brief introduction to R, so that we can focus on the statistical practices without running into too many R programming issues. Therefore, in the forthcoming labs, some R code will be provided by your teacher. Nevertheless, it is important that you understand what this code does, and that you can reproduce the code to use it on new datasets and analyses - this should be our goal when it comes to R usage during this course.

To get more familiar with the R working environment and Rstudio, it may also be helpful to look at some of the additional material posted in the *R usage* zip-folder in the "Introduction" module on *Studium*, as well as some of the links provided in this lab. The course is quite short and will be focused exclusively on statistical problems, and most technical issues concerning R-programming will have to be left aside.

The goals for today are to:

- Install Rstudio on your computer
- Make you familiar you with the basic lay-out of Rstudio
- Make you familiar with basic concepts in R programming, such as understanding how to *assign*, *display* and *evaluate* objects and apply functions to them.
- Making sure that you can read in data into Rstudio, and set up a preferred way of working, creating and saving scripts and their associated data and outputs in a specific folder of your choice

Important

Even though many of the results of the code provided in this lab is already given in this pdf, you should run the code yourself in Rstudio/R. Toggle with the code to make sure you understand what the different bits of code do/how they work.

If you are already familiar with R, you may be able to skip through this introduction quite quickly and check out some useful links and tips at the bottom related to some of the more recent developments in R programming and data-handling - the *tidyverse*, a compilation of packages allowing fast computing, data handling and good programming practices, that are compatible with Rstudio.

Note! R usage on this course will be focused on R base functions - a good place to start!

What is R?

R is a powerful interactive computer package that is orientated towards statistical applications. It will run on Windows, Linux and Mac. It consists of a base system that can be downloaded without charge together with many contributed packages for specialist analyses. It offers:

- an extensive and coherent set of tools for statistics and data analysis
- a language for expressing statistical models
- comprehensive facilities for performing matrix calculations and manipulations, enabling many applications in multivariate data analysis and linear models
- graphical facilities for interactive data analysis and display
- an object-orientated programming language that can easily be extended
- an expanding set of publicly available libraries or packages of routines for special analyses
- libraries or packages available from the official Contributed Packages webpages are thoroughly tested by the R Core Development Team
- packages have manuals, help systems and usually include illustrative data-sets
- The R community is HUGE. There are tons of web pages offering help, tips and instructions. Googling has never been more meaningful.

Installing R on your own computer

Full instructions for installing R are given on the R Project home page at *http://www.r-project.org/* and/or *http://www.rstudio.com/*. The first step is to choose a site geographically close to you from which to download the package. Next choose the appropriate operating system, select the option *base* and download the system. Accepting the option to run the download file will install the package after downloading it. Accepting the default options for locations etc is simplest but these can be customized.

RStudio

A highly recommended open source interface to R is RStudio *click here.* RStudio groups your work into four windows (script window – usually placed top left), output window to which you send your code (console – usually placed bottom left), environment overview (top right) and files/plots/packages overview (bottom right).

The **script window** (top left)is where you will spend most of your time - this is where you code and construct your models. The most important thing is to save clean code with the script file. Nowadays using Rstudio, this is easily done by just hitting the save button in the top-left menu, and when you work in a project, all associated files, created objects and results are saved together in the project folder. Nevertheless, the script file is the most important thing to save - with this you or anyone else can recreate your analyses quickly and do not really need to save much else.

In the **output window** (bottom left) you will see the results from the code you send from your script in the *console* tab. You will spend a lot of time here looking at the results, but also with interpreting error messages. Make sure that the *console* shows the > sign when you send your code. If it does not, this is an indication that something went wrong previously, and you should check error messages to see if you need to correct something in your previous code. *Always* do this code debugging, because your next line of code you send will likely also show error messages, even though it is perfectly fine (the error is due to your previous code). It is also possible to write code straight into the *concole* and execute it. This is not necessary, but I sometimes do it if I want to try different things out without saving the code.

In the **environment window** (top right) you will see all the objects you have saved in the session. These are typically data that you read into R and saved as a dataset, and then several vectors, matrices and subsets of data that you create from the original dataset. You can click on each object to display them (this is done in a new tab in the top left window).You can also see which R packages you have loaded into the session. There are thousands of R packages, each allows you to do some specific thing that you may require to perform a specific analysis.

In the **files/plots/packages window** (bottom right) you can find files in folders that you want to import into R (typically datasets saved as excel or txt). The folder that you chose for your project is displayed by default. You can also find, read about, and install R packages needed for analysis using the *Packages* tab. Here you will also be able to read and get help about specific functions you apply (see further below) in the *Help* tab. While working, you will probably spend most time on this course looking at the *Plots* tab - here you can look at plotted data and result figures. There are options to zoom in on figures and display them in a new window, and you can also save and export them.

Working directory and Projects

The working directory in R/Rstudio is the physical place where R will look for files (e.g. datasets or scripts) that you load into R, and where it will store your output files by default (e.g. plots and scripts), unless you tell it otherwise. To determine the current working directory, type **getwd()** which yields the path to the current working directory on your computer.

To set a working directory use **set.wd()**, e.g: *set.wd(C:/Users/CharlesDarwin/Documents/Peacock)*. **You can also set your working directory by using your mouse in Rstudio** - go to *Session* in the role-down menu and choose *Set working directory*. This is the preferred way to do it on this course if you are a beginner, as Linux, Mac and Windows have slightly different ways to specify the directory with text in R.

In Rstudio you can choose to work in a **Project**. If doing so, all your data files, output files, figures, etc get associated with the project and end up in the same folder automatically if saved. You can open up the project if you restart Rstudio and take off from where you stopped last time in a click. This is a highly preferred way to work in R. When starting your work, go to *File* in the role-down menu and choose *New Project* and select an appropriate place to save your project. It makes sense to **start a new project for each lab you do for this course**. At the end of each lab, you ideally have saved code and associated output tables and figures in your Project folder that can be compiled into a report if you wanted to. A simple way to make the report if you are a beginner is to paste cleaned code together with the main tables, figures and your personal interpretation and answers into a word document and save it - but any way that works for you is fine! If you miss a lab you have to hand in such a report. The style of the report is also similar to what would be expected for answers to exam questions that are based on analyzing data. I have uploaded a template for how a report/answer to an exam question could look like on *Studium* in the module *Practice questions and extra material*.

RMarkdown

This document was created using RMarkdown. RMarkdown is a simple formatting syntax for authoring HTML, PDF, and MS Word documents, and is well integrated in Rstudio. It does not only read the R programming language, but also other languages such as *Python* and *Bash*. For more details on using RMarkdown see http://rmarkdown.rstudio.com.

There is no pressure to learn or use RMarkdown during this course, but for those of you already familiar with R, it might be worth having a look as it offers a neat way to annotate, store and present reproducible code and associated results, much like you should do to store your work during labs. I have added the RMarkdown file used to create this document to the folder of lab 1 (R use), for those that are interested and want to get an idea of how it works. You can also go to the *Help* tab in the role-down menu at the top of Rstudio. There, choose *Cheat Sheets*. This gives you an overview of how to use Rmarkdown and other resources for R (more on this below).

Getting help

Besides asking your teachers on this course, R has a very rich online help system.

- There are various online manuals that can be accessed from the help tab on the role-down menu at the top. This should be your first attempt at finding the answer to your question. You can get help on specific commands by typing a **?** followed by the command name into the Rstudio console. For example, for help on plotting graphs, try: **?plot** or **help(plot)**. To access the hypertext (HTML) version of R´s online documentation, type **help.start()**. These help files will be displayed in the *Help* window, bottom right.
- Sometimes, these help-files are quite technical, and sometimes you can find better help by googling to find a help page for a specific function that someone has been nice enough to create.
- There are many different online forums, depending on your particular type of problem, so it is best to just start googling! Be careful before posting your question on the forum, as often you should follow special rules and present reproducible code so that people can understand your problem. Most of the time though, you do not need to pose your question but will find answers to it already posted online.

Basic R functions

All commands in R are regarded as functions, they operate on arguments, e.g. **plot(x, y)** plots the vector y against the vector x – that is, it produces a scatter plot of y vs. x. Commands that you can give R are of three categories: *evaluation*, *assignment*, and *display*.

Evaluation

To evaluate a function, type its name followed by the list of arguments, which may be empty, given in brackets. For example, **sqrt()** is the name of the function that calculates square roots: **sqrt(3)** returns 1.732051

Most functions produce numerical output, such as the sqrt example above. For example, the function **rnorm(n)** generates a random sample of specified size *n* from the standard, (zero mean, unit variance) normal distribution N(0,1). Try typing **rnorm(20)** to generate a sample of 20 random observations from the standard normal distribution. You can use rnorm to generate a random sample of any normally distributed variable. For example, entering **rnorm(1000, 10, 2)** generates 1000 random draws from a normal distribution with a mean of 10 and a standard deviation of 2.

An expression to be evaluated need not be in the form of a function call; the usual arithmetic operators +, -, *, /, and ^ are all available, as are more specialized operators. For example:

```
2+7
```

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

```
2/(3+5)
```

```
## [1] 0.25
```

```
sqrt(9) + 5^2
```

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

```r
9^(1/2) + 5^2
```

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

Assignment

Above, the results of the evaluations are displayed immediately. Instead, the results of an evaluation can be assigned to a variable:

```r
x <- rbinom(10,5,0.2)
```

which creates a vector of 10 outcomes of binomial sampling (each with 5 trials, and probability of success equal to 0.2), and calls it **x**. **x** will show up as a stored object in your **environment tab** (top right). Functions can be performed on variables. For example, **max(x)** returns the largest element of the vector x.

Remember, assignments do not automatically display anything. Variable (and command) names are case-sensitive, and you should avoid using names that also serve as R functions, for example: t, c, q, F, I, T, diff, df, and pt.

Display

To see the value of an object that you have created, or that already exists in R, just type its name and hit return. For example:

```r
x
```

```
##  [1] 2 2 0 0 1 0 2 0 2 1
```

This applies equally to functions; to R, functions are just objects of a particular sort.

Vectors and Matrices

Numerical vectors: A vector of numbers can be made using the c command, for example:

```r
x<-c(2,4,6,8,10)
```

Or other commands like **seq()** or **rep()** (use **?** to find out more about a function: e.g. **?seq**). Various commands can be performed on vectors such as **sum(x)** and **mean(x)**.

Individual elements of vectors can be accessed by specifying the required index within square brackets:

```r
x[2]
```

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

```r
x[c(1,2,3)]
```

```
## [1] 2 4 6
```

gives the second, or the first, second and third elements of **x**, respectively.

Operations can be performed on each element of a vector using a single command. For example, 3*x or xˆ2. Element-wise operations between vectors can be performed similarly. For example:

```
y<-c(1,3,5,7,9)
y*x
```

```
## [1]  2 12 30 56 90
```

The commands **rbind** and **cbind** can be used to merge row or column vectors to matrices.

```
x <- c(1,2,3)
y <- c(4,5,6)
A <- cbind(x,y)
B <- rbind(x,y)
C <- t(B)
```

The last command gives the matrix transpose of **B**.

Multiplying two matrices in R renders element-wise multiplication, and *NOT* matrix multiplication. To perform matrix multiplication you use the %*% operator. Due to the dimensions of A,B, and C, you can only do element-wise multiplication of A and C and matrix multiplication between B and A/C:

```
A
```

```
##      x y
## [1,] 1 4
## [2,] 2 5
## [3,] 3 6
```

```
B
```

```
##   [,1] [,2] [,3]
## x    1    2    3
## y    4    5    6
```

```
C
```

```
##      x y
## [1,] 1 4
## [2,] 2 5
## [3,] 3 6
```

```
A*C
```

```
##      x  y
## [1,] 1 16
## [2,] 4 25
## [3,] 9 36
```

```
A%*%B
```

```
##      [,1] [,2] [,3]
## [1,]   17   22   27
## [2,]   22   29   36
## [3,]   27   36   45
```

Factors

Factors are R's data type for storing categorical variables; they are typically un-ordered but can be ordered (ordinal) too. Factors are stored as two pieces of information; a character vector of unique levels and a numeric vector of indices. For instance:

```
x <-factor(letters[c(1:5,1:5)]) # letters is a pre-existing built-in variable in R
x
```

```
##  [1] a b c d e a b c d e
## Levels: a b c d e
```

```
str(x) # compactly display the internal structure of an R object
```

```
##  Factor w/ 5 levels "a","b","c","d",..: 1 2 3 4 5 1 2 3 4 5
```

```
levels(x) #levels returns the levels of the attribute x
```

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

Data frames

A data frame is a data structure that corresponds to what often is referred to as a "data matrix" or a "data set". It is a list of vectors and/or factors of the same length with a unique set of row names. One may create a data frame from preexisting variables:

```
treatment <- factor(letters[c(1:5,1:5)])
response  <- rnorm(10)
myData    <- data.frame(treatment, response)
myData
```

```
##    treatment   response
## 1          a -0.6032768
## 2          b  0.1540309
## 3          c  1.0733590
## 4          d -0.3684141
## 5          e -0.7759124
## 6          a -2.0859416
## 7          b -0.3730146
## 8          c -0.3542732
## 9          d  0.6702998
## 10         e  0.4888600
```

Variables from the data frame are extracted using the $ notation, **myData$response** yields the elements in the original vector *response*. If one needs element number 5 in that vector, one types **myData$response[5]** or more generally **myData[5,2]**.

This last bit of code calls out the fifth row, and the second column in the data frame *myData*. This is a general way to grab entries from an R object – rows first followed by coloumns, inside square-brackets. Thus, *data.frame* can be used to organize data of different kinds and to extract particular subsets. Data frames are what you will be analyzing for most parts of this course.

Subsetting

Now that we know this, we can also create subsets of data, to perform downstream analyses on only parts of the data. For example, let's say you only want to compare treatments "a" and "e".

```
newData <- myData[myData$treatment=="a"|myData$treatment=="e",]
newData
```

```
##    treatment   response
## 1          a -0.6032768
## 5          e -0.7759124
## 6          a -2.0859416
## 10         e  0.4888600
```

The **|** operator means *or*, and == means *equal to*. There are more such operators that allow you to select data in different ways, such as **&** (*and*), **<** (*smaller than*) and **!=** (*not equal to*). There are many more ways to create subsets of data. One way is to use the function **subset()**. Another way is to use the options in the **dplyr** package in *tidyverse* (see below).

Figures

Basic scatter plot: Suppose you have two vectors x and y generated as:

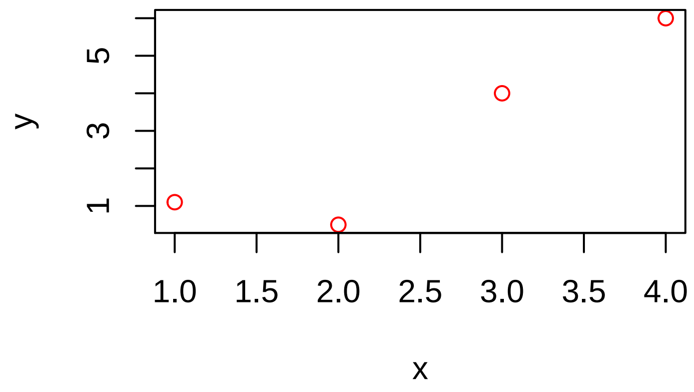
```
x<-c(1,2,3,4)
y<-c(1.1,0.5,4,6)
```

To produce a scatter plot of y against x type:

```
plot(x,y, col="red")
```

or

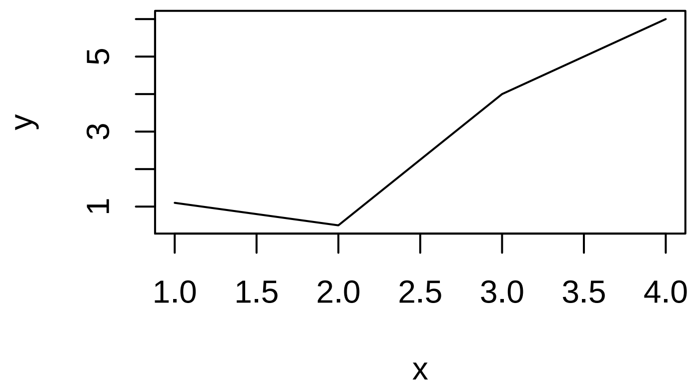
```
plot(y~x, col="red")
```



The first line of code plots whichever variable comes first within the parenthesis on the horizonal (x) axis, and the second variable on the vertical (y) axis. The second line of code is read as: *y depends on x* and results in the same graph (because the dependent variable is put on the y-axis by default). The second line of code is similar to the way we express relationships between variables when we specify statistical models in R, and you may therefore prefer this way of specifying your plot (to avoid confusion). More on statistical modelling in the forthcoming labs.

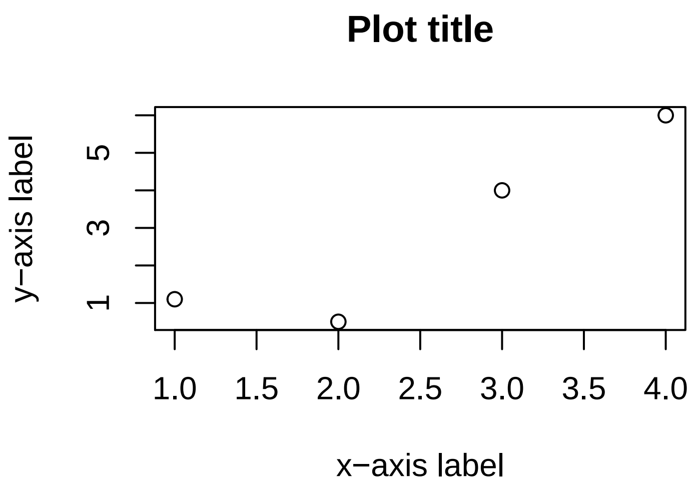
Adding the argument **type="l"** produces a line plot

```
plot(y~x,type="l")
```



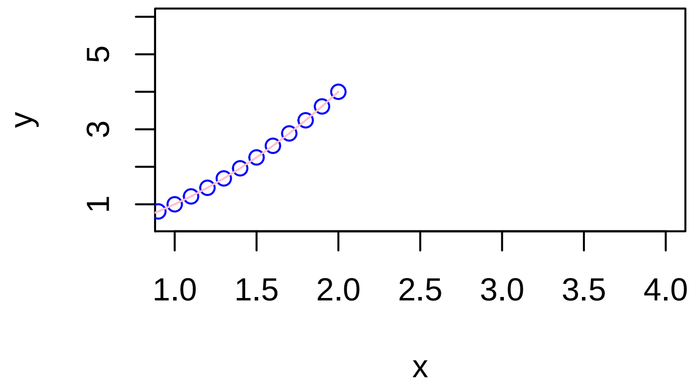
You can specify your own axes labels and title with additional arguments

```
plot(y~x, xlab = "x-axis label", ylab = "y-axis label", main="Plot title")
```



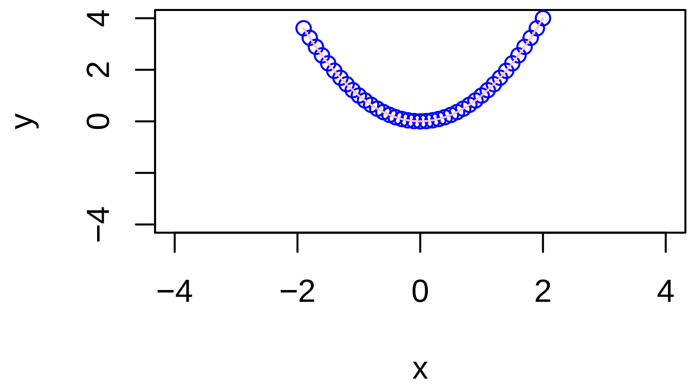
You can add more points or lines to an existing plot by using the **points()** or **lines()** commands. For example, plotting two more vectors x2 and y2 against each other in the present graphical window

```
x2 <- seq(-1.9,2,0.1)
y2 <- x2^2
plot(y~x,type="n")

points(x2,y2, col="blue")

#or

lines(x2,y2, col="pink")
```

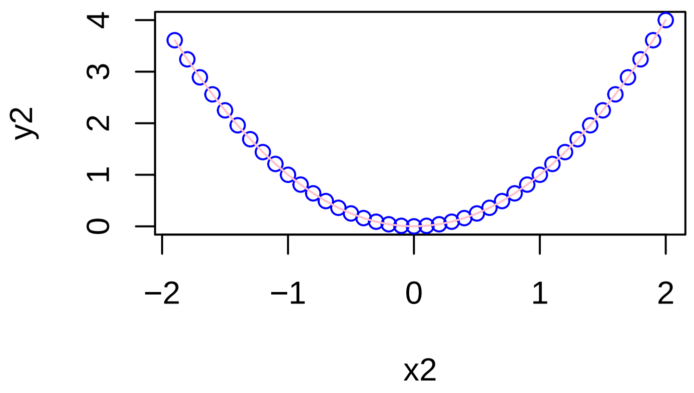


The ranges for both axes are chosen automatically based on the data you enter using **plot()**, and are not altered when you add points or lines that don't fit. If you wish to specify the range for an axis, e.g., you want the x- and y-axis to range from -4 to 4, you should instead type

```
plot(y~x,type="n", xlim=c(-4,4), ylim=c(-4,4))
points(x2,y2, col="blue")
lines(x2,y2, col="pink")
```

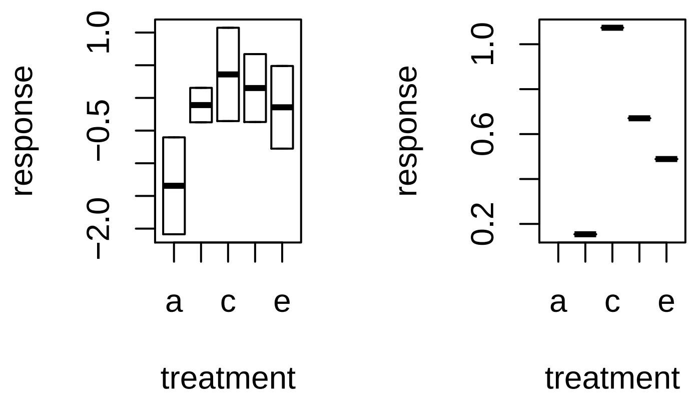


```
#using the new variables inside plot() directly makes the function
#use the data to decide appropriate scales for the axes:
plot(y2~x2,type="n")
points(x2,y2, col="blue")
lines(x2,y2, col="pink")
```



Another basic plot in R is the *box plot* used to look at the data spread in several discrete groups. We can use it to compare the five treatments (a-e) that you stored in the data frame *myData* above:

```
par(mfrow=c(1,2)) #plot several figures and arrange next to each other
boxplot(response ~ treatment, data=myData)
boxplot(response ~ treatment, data=myData[myData$response>0,])
```



You can find out more about this function by typing **?boxplot**

You can find information about how to set up general graphical parameters in R by typing: **?par**.

There are ENDLESS possibilities to do beautiful graphing in R, and many help pages about it. Here is a **link** to a beginners guide for plotting in Rstudio. There are even several books written about just making figures in R. We will mostly handle very basic and the most essential graphs on this course, directly related to the data problems we encounter.

Getting Data into R

Load existing datasets

You can load an existing R dataset with the command **load('Name_of_Dataset')**. You often do this when you work with example-analyses and use datasets that were included in the package you are trying out.

Importing data into R

You have to make sure the dataset is in your working directory. If not, you can either change the working directory in Rstudio (top role-down menu -> *Session* -> *Set working directory*), or specify the full path inside the brackets of the function used to read in the data.

Often you will want to import data stored in alternative file formats for analysis, and there are many ways to do this with R. The most common way is to have your data in excel format and then saving the data as either a plain text file (txt) or a comma separated file (csv) which are read into R. For example, suppose you have a plain (tab delimited) text file *measurements.txt* saved on your computer, with each row giving the age, height, and weight of an individual, and the first row giving the column headings:

age	height	weight
22	172	68
25	150	60
32	180	75

You can import this data into R and store it as a data frame called *measurements* with the **read.table** command

```
measurements<-read.table("measurements.txt",header=T)
```

assuming the file *measurements.txt* is in the current working directory, everything went well. You can extract single columns from the data frame as follows:

```
measurements$age
measurements$height
measurements$weight
```

R will not understand what you want if you simply type **age**, because this variable does not exist outside the data frame **measurements**. Hence, above you are telling R to look for the variable **age** inside the data frame **measurements**. A good practice when having read data into R is to get an overview of the data.frame to see that everything went as you think it did by typing:

```
str(name_of_your_dataset)

#and

head(name_of_your_dataset)
tail(name_of_your_dataset)
```

This gives you the structure of the data.frame R has created from your data, and the categories that R has assigned to the data variables (for example: is variable 3 a *factor* like I want it to be, or has R assigned it to be a *numeric* variable?). Using Rstudio, you can also find the read in data in the "Environment window" (top right) and click on it to display the data.frame created by R.

When reading a datafile into R, the option header=T tells R to use the first line of the text file as the column names. If this is changed to header=F, R will assign its own column names, and all values in the text file will be used as data values, whether this is appropriate or not. Hence you should use header=T if the first row of your text file gives the column names, and header=F if your text file contains data values only.

To import comma separated format (csv), you can use the command read.csv:

```
read.csv("measurements.csv",header=T, sep = ";")
```

The **sep** command tells R that entries are separated with semicolons.

The absolutely simplest way to important your data if you are using Rstudio, which circumvents potential issues with accessing datasets located in different places on your computer in the same R session, is to use the mouse in Rstudio. Go to the *Environment* tab (top right) in Rstudio and choose *Import Dataset*. From thereon, you can simply click your way to your datafile on your computer and choose the type of file to read in. You will also see how the data frame will look once the data is imported, so you will get an idea right away of whether you have chosen the correct options for reading in your file.

Exercise

The two versions of the *measurements* file (tab delimited - txt and comma separated - csv) are in the folder of Lab 1 on Studium. Download the files and read them into Rstudio. Using what you have learnt so far, can you perform some plots of their data? Summaries of the variables in the dataset? Calculate the body mass index (bmi) as a new variable and add it to the data frame?

Further exercises and help

These were just a few examples of R basics. If you are new to R, it is recommended to go through some additional tutorials. For example, you can look at the first chapters of the *R book* by Crawley (you can find a downloadable pdf on the net if you google). Or, you can take a free online course by *DataCamp* (free registration) *click here.* In the *R usage* folder on *Studium* you will also find some pdfs containing a summary of the most common base functions https://clausrueffler.github.io/CodingWithR/TheConsole.html

Already experienced with R? - Check out *The Tidyverse*

The tidyverse (*click here*) is a compilation of packages designed to take data science with R to the next level (at least according to its creators). All the associated packages share common programming features, making them work well together, with some functions in some packages relying on data already having been handled by another tidyverse package. This applies specifically to how some data–sets are created once read into R.

Remember that we will mostly work with *base R*, which can cause some incompatibility with things you want to do with *tidyverse.*

So right now the *tidyverse* might be something you will save for the future, but be happy to know about. Its functions are argued to be particularly helpful for handling big data, manipulating data, and graphing data, as well as increasing compatibility with other programming languages. Here is a good *online resource* for getting an introduction to the *tidyverse.*

ggplot2

ggplot2 (*click here*) is a part of the *tidyverse* and a package specifically designed to make graphing easy and powerful. Like other packages in the *tidyverse*, it has a dedicated homepage, help-sites and community, and is what many researchers prefer to use when illustrating their results. You can check out some examples by following this *link* .

Cheat-sheets

Also check out the "Cheat-sheets" for *Rstudio*, *RMarkdown*, *ggplot2* and some other *tidyverse* packages in the *R-usage* folder on *Studium.* You can get some cheat-sheets directly from the *Help* in the top role-down menu on Rstudio. You can get even more of these cheat-sheets from *tidyverse.org* - for future purposes perhaps.