Data analysis using R for Psychology

Alexander (Sasha) Pastukhov

2020-11-10

Contents

Introduction

About the seminar

This is a material for *Applied data analysis for psychology using the open-source software R* seminar. Each chapter covers a single seminar, introducing necessary ideas and is accompanied by a notebook with exercises, which you need to complete and submit. The material assumes no foreknowledge of R or programming in general from the reader. Its purpose is to gradually build up your knowledge and introduce to a typical analysis pipeline. It is based on a data that is typical for the field (repeated measures, appearance, accuracy and response time measurements, Likert scale reports, etc.), you are welcome to suggest your own data set for analysis. Even if you already performed the analysis using some other program, it would still be insightful to compare the different ways and, perhaps, you might gain a new insight. Plus, it is more engaging to work on your data.

Remember that throughout the seminar you can and should(!) always ask me whenever something is unclear, you do not understand a concept or logic behind certain code. Do not hesitate to write me in the team or (better) directly to me in the chat (in the latter case, the notifications are harder miss and we don't spam others with our conversation).

You will need to submit your assignment one day before the next seminar (Tuesday before noon at the latest), so I would have time evaluate it and provide feedback.

As a final assignment, you will need to program a full analysis pipi line for a given data set (or, if you want, for your data set). All the necessary steps will be covered by the seminar material. Please inform me, If you require a grade, as then I will create a more specific description for you to have a clear understanding of how the program will be graded.

Content of the seminar

The ultimate goal of the seminar is to give you tools to perform a complete analysis of a typical psychology research data, including data import and merging, pre-processing, aggregating, plotting, and performing statistical analysis. We will start with very basic R programming concepts (vectors, tables) before proceeding to learn about data import and visualization via the grammar of graphics. Next, we will learn about piping that makes sequential analysis easy to write and read. Then, we will see how combine piping with Tidyverse *verbs*. Next, we will see how to use, visualize, and report statistical models. Finally, you will learn the power of functional programming that provides ultimate flexibility in R.

Note on exercises

In many exercises your will be not writing the code but reading and understanding it. Your job in this case is "to think like a computer". Your advantage is that computers are very dumb, so instructions for them must be written in very simple, clear, and unambiguous way. This means that, with practice, reading code is easy for a human (well, reading a well-written code is easy, you will eventually encounter "spaghetti-code" which is easier to rewrite from scratch than to understand). In each case, you simply go through the code line-by-line, doing all computations by hand and writing down values stored in the variables (if there are too many to keep track of). Once you go through the code in this manner, it should be completely transparent for you. No mysteries should remain, you should have no doubts or uncertainty about any(!) line. Moreover, you then can run the code and check that the values you are getting from computer match yours. Any difference means you made a mistake and code is working differently from how you think it does. In any case, **if you not 100% sure about any line of code, ask me, so we can go through it together!**

In a sense, this is the most important programming skill. It is impossible to learn how to write, if you cannot read first! Moreover, when programming you will probably spend more time reading the code and making sure that it works correctly than writing the new code. Thus, use this opportunity to practice and never use the code that you do not understand completely. Thus, there is nothing wrong in using stackoverflow but do make sure you understand the code you copied!

Why R?

There are many software tools that allow you preprocess, plot, and analyze your data. Some cost money (SPSS, Matlab), some are free just like R (Python, Julia). Moreover, you can replicate all the analysis that we will perform using Python in combination with Jupyter notebooks (for reproducable analysis), Pandas (for Excel-style table) and statmodels (for statistical analysis). However, R in combination with piping and Tidyverse family of packages is optimized for data analysis, making it easy to write simple, powerful and expressive code that is very easy to understand (a huge plus, as you will discover). I will run circles around myself trying to replicate the same analysis in Python or Matlab. In addition, R is loved by mathematicians and statisticians, so it tends to have implementations for all cutting edge methods (my impression is that even Python is lagging behind it in that respect).

Tidyverse versus base R

I will be teaching what one might call a "dialect" of R, based on Tidyverse family of packages. R is extremely flexible, making it possible to redefine its own syntax. Because of that Tidyverse-based code is very different from the base R code to the point that it might look like written in a completely different language (which, in a sense, it is). Although Tidyverse, at least in my opinion, is a better way, we will start with *base R*, so that you will be able to read and understand code written outside of Tidyverse, as it is also very common.

Getting Started

Installing R

Go to r-project.org and download the current stable version of R for your platform. Run the installer, accepting all defaults. The installer will ask you whether you also want the 32-bit version to be installed alongside 64-bit. You probably won't need 32-bit, so if space is at premium you can skip it. Otherwise, it will make very little difference.

Installing R-Studio

Go to rstudio.com and download *RStudio Desktop Free* edition for your platform. Install it using defaults. The *R-Studio* is an integrated development environment for R but you need to install R separately first! The R-Studio will automatically detect latest R that you have and, in case you have several versions of R installed, you will be able to alter that choice.

I will explain the necessary details on using R-Studio throughout the seminar but the official cheatsheet is an excellent, compact, and up-to-date source of information. In fact, R Studio has numerous cheatsheets that describe individual packages in a compact form.

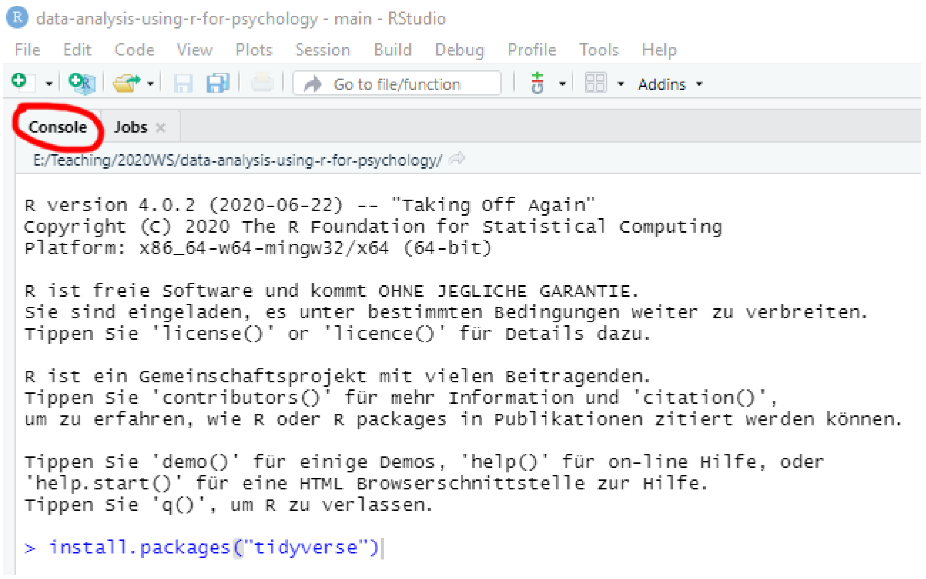
Installing RTools

If you are using Windows, we might need Rtools for building and running some packages. You do not need to install it at the beginning, but when we will need it later, just following the link above, download the latest *Rtools* version, run the installer using the defaults and **follow the instructions on that page to put Rtools on the PATH**! (I do not repeat them here, because they might change).

Installing packages

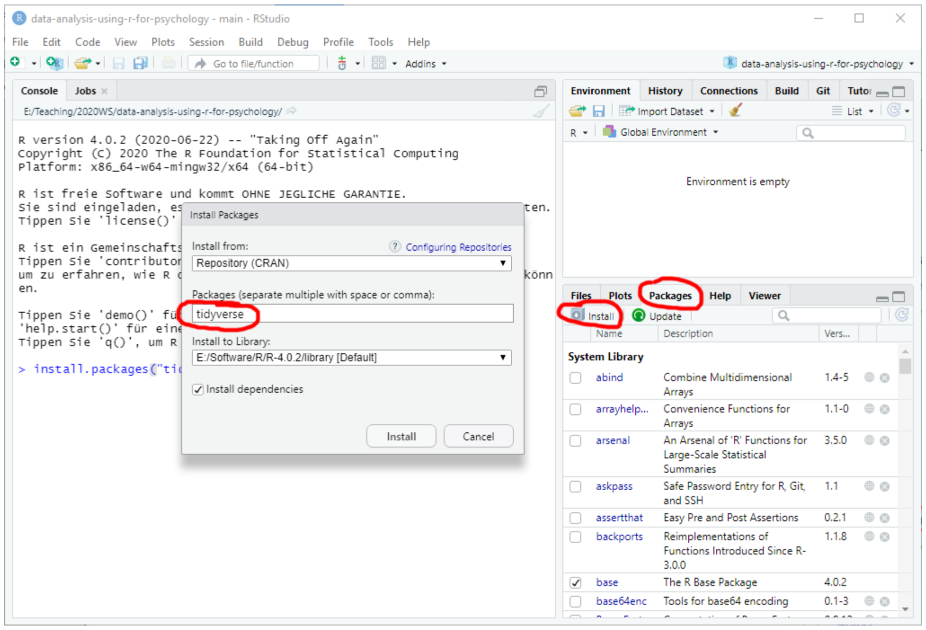
The real power of R lies in a vast community-driven family of packages suitable for any occasion. The default repository used by R and R-Studio is The Comprehensive R Archive Network (a.k.a. *CRAN*). It has very strict requirements for submitted packages, which makes it very annoying for the authors but ensures high quality. Most packages are well-documented and come with example data and code. We will use CRAN as a sole source for packages, but there are alternatives, such as Bioconductor that might have a package that is missing at CRAN. The Bioconductor relies on its own package manager, so you will need to consult the latest manual on their website.

To install CRAN package you have two alternatives: via command line function or via R-Studio package manager interface (which will call the function for you). In the former case, go to `Console` tab and type `install.packages("package-name")`, for example `install.packages("tidyverse")`, and press `Enter`.


data-analysis-using-r-for-psychology - main - RStudio
File Edit Code View Plots Session Build Debug Profile Tools Help
Go to file/function
Addins
Console
Jobs
E:/Teaching/2020WS/data-analysis-using-r-for-psychology/

```
R version 4.0.2 (2020-06-22) -- "Taking Off Again"
Copyright (C) 2020 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (64-bit)

R ist freie Software und kommt OHNE JEGLICHE GARANTIE.
Sie sind eingeladen, es unter bestimmten Bedingungen weiter zu verbreiten.
Tippen Sie 'license()' or 'licence()' für Details dazu.

R ist ein Gemeinschaftsprojekt mit vielen Beitragenden.
Tippen Sie 'contributors()' für mehr Information und 'citation()',
um zu erfahren, wie R oder R packages in Publikationen zitiert werden können.

Tippen Sie 'demo()' für einige Demos, 'help()' für on-line Hilfe, oder
'help.start()' für eine HTML Browserschnittstelle zur Hilfe.
Tippen Sie 'q()', um R zu verlassen.

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

Alternatively, go to `Packages` tab, click on `Install` button, enter the package name in the window, which has autocomplete to help you, and press `Install`.


data-analysis-using-r-for-psychology - main - RStudio
File Edit Code View Plots Session Build Debug Profile Tools Help
Go to file/function
Addins
Console
Jobs
Environment History Connections Build Git Tutor
Import Dataset
Global Environment
Environment is empty
Files Plots Packages Help Viewer
Install
Update

E:/Teaching/2020WS/data-analysis-using-r-for-psychology/

```
R version 4.0.2 (2020-06-22) -- "Taking Off Again"
Copyright (C) 2020 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (64-bit)

R ist freie Software und kommt OHNE JEGLICHE GARANTIE.
Sie sind eingeladen, es                                      ten.
Tippen Sie 'license()'

R ist ein Gemeinschafts
Tippen Sie 'contributor
um zu erfahren, wie R                                        könn
en.

Tippen Sie 'demo()' fü
'help.start()' für eine
Tippen Sie 'q()', um R

> install.packages("ti
```

Install Packages

Install from: Repository (CRAN)

Configuring Repositories

Packages (separate multiple with space or comma): tidyverse

Install to Library: E:/Software/R/R-4.0.2/library [Default]

[✓] Install dependencies

Install Cancel

	Name	Description	Vers...
System Library			
☐	abind	Combine Multidimensional Arrays	1.4-5
☐	arrayhelp...	Convenience Functions for Arrays	1.1-0
☐	arsenal	An Arsenal of 'R' Functions for Large-Scale Statistical Summaries	3.5.0
☐	askpass	Safe Password Entry for R, Git, and SSH	1.1
☐	assertthat	Easy Pre and Post Assertions	0.2.1
☐	backports	Reimplementations of Functions Introduced Since R-3.0.0	1.1.8
✓	base	The R Base Package	4.0.2
☐	base64enc	Tools for base64 encoding	0.1-3

In some cases, R will ask whether you want to install packages from source. In this case, it will grab the source code and compile the package, which takes time and requires RTools. In most cases, you can say "No" to install a pre-build binary version. The binary version will be slightly outdated but the emphasis is on *slightly*.

Please install the following packages:

- `tidyverse` : includes packages from data creation (`tibble`), reading (`readr`), wrangling (`dplyr`, `tidyr`), plotting (`ggplot2`). Plus type specific packages (`stringr` for strings, `forcats` for factors) and functional programming (`purrr`).
- `rmarkdown` : package for working with RMarkdown notebooks, which will we use to create reproducible analysis.
- `fs` : file system utilities.

Keeping R and packages up-to-date

R and packages are getting constantly improved, so it is a good idea to regularly update them. For packages, you can use `Tools / Check for Packages Updates...` menu in R-Studio. To update R and, optionally, packages, you can use installr package that can install newest R (but it keeps old version!) optionally copying your entire library of packages, updating packages, etc. For

R-Studio itself, use `Help / Check for Updates` menu and install a newer version, if it is available (it is generally a good idea to keep your R-Studio in the newest state).

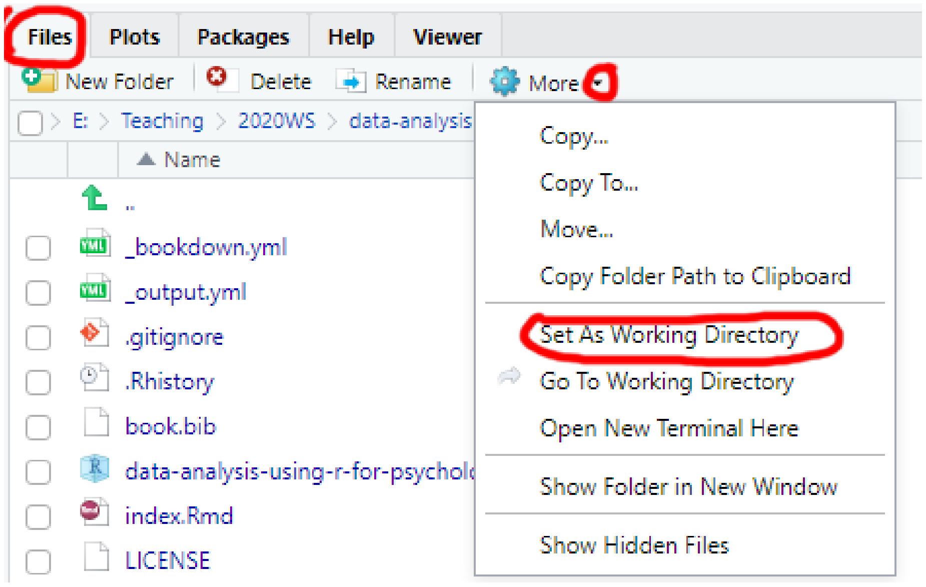
Chapter 1

Reproducable Research: Projects and RMarkdown Notebooks

Our aim is to create reproducible research and analysis. It is a crucial component of the open science movement but is even more important for your own research or study projects. Doing analysis is easy. The trick is to create a self-contained well-documented easy-to-understand reproducible analysis. An analysis that others and, most importantly, future-you can easily understand saves you time and gives you a deeper insight into the results (less mystery is better in these cases). It also makes it easier to communicate your results to other researchers or fellow students.

1.1 Projects

One of the most annoying features of R is that sees files and folders only relative to its "working directory", which is set via setwd(dir) function. What makes it particularly confusing, is that your currently open file may be in some other folder. If you simply use `File / Open`, navigate to that file and open it, it does not change your working directory. Similarly, in R-Studio you can navigate through file system using `Files` tab and open some folder you are interested in but that **does not make it a working directory**. You need to click on `More` and `Set As Working Directory` to make this work (and that trick won't work for an opened file).



In short, you may think that you are working in a particular folder but R will have its own opinion about this. Whenever this happens, it is really confusing and involves a lot of cursing as R cannot find files that you clearly see with your own eyes. To avoid this you should organize any project or seminar as an *R Project*. It assumes that all the necessary files are in the project folder, which is also the working directory. R Studio has some nice project-based touches as well, like keeping tracking of which files you have open, providing version control, etc. Bottom line, **always** create a new R-project to organize yourself, even if it involves just a single file to try something out. Remember, "Nothing is more permanent than a temporary solution!" Which is why you should **always** write your code, as if it is for a long term project (good style, comprehensible variable names, comments, etc.), otherwise your temporary solution grows into permanent incomprehensible spaghetti code.

Let us create a new project for this seminar. Use `File / New Project...`, which will give you options of creating it in a new directory (you get to come up with a name), using an existing directory (project will be named after that directory), or check it out from remote repository (something we won't talk about just yet). You can do it either way. This will be the project folder for this seminar and you will need to put all notebooks and external data files into that folder. Next time you need to open it, you can use `File / Recent Projects` menu, `File / Open Project...` menu, or simply open the `<name-of-your-project>.Rproj` file in that folder.

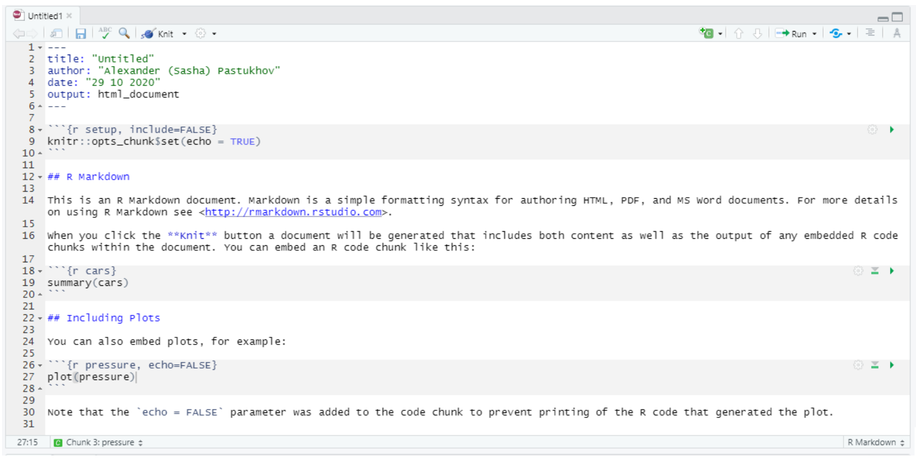
1.2 RMarkdown

RMarkdown notebooks combine formatted text and illustrations with code. When a notebook is "knitted", all the code is ran and its output, such as tables and figures, is inserted into the final document. This allows you to combine the narrative (the background, the methodology, discussion, conclusions, etc.) with the actual code that implements what you described.

Importantly, notebooks can be knitted into a variety of formats including HTML, PDF, Word document, EPUB book, etc. Thus, instead of creating plots and tables and saving them into separate files so you can copy-paste them into your Word file (and then redoing it, if something changed, and trying to find the correct code that you used the last time, and wondering why it does not run anymore…), you simply "knit" the notebook and get the current and complete research report, semester work, presentation, etc. Even more importantly, same goes for others, as they also can knit your notebook and generate its latest version in format they need. All your exercises will be based on RMarkdown notebooks, so you need to familiarize yourself with them.

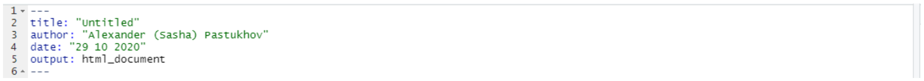
We will start by learning the markdown, which is a family of human-oriented markup languages. Markup is a plain text that includes formatting syntax and can be translated into visually formatted text. For example, HTML and LaTeX are markup languages. The advantage of markup is that you do not need a special program to edit it, any plain text editor will suffice. However, you do need a special program to turn this plain text into the document. For example, you need Latex to compile a PDF or a browser to view HTML properly. However, anyone can read your original file even if they do not have Latex, PDF reader, or a browser installed (you do need Word to read a Word file!). **Markdown** markup language was design to make formatting simple and unobtrusive, so the plain document is easy to read (you can read HTML but it is hardly fun!). It is not as feature-rich as HTML or LaTeX but covers most of your usual needs and is very easy to learn!

Create a new markdown file via `File / New File / R Markdown...` menu. Use `Seminar 1` for its title and HTML as default output format. Then you need to save the file (pressing `Ctrl + S` will suffice) and call the file `seminar-01` (R Studio will add `.Rmd` extension automatically). The file you created is not empty, as R Studio is kind enough to provide a template and example for you. Knit the notebook by clicking on `Knit` button or pressing `Ctrl + Shift + K` to see how the properly typeset text will look (it will appear in `Viewer` tab).



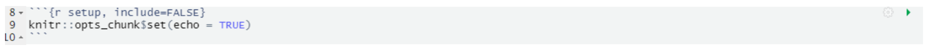
~~~
---
title: "Untitled"
author: "Alexander (Sasha) Pastukhov"
date: "29 10 2020"
output: html_document
---

```{r setup, include=FALSE}
knitr::opts_chunk$set(echo = TRUE)
```

## R Markdown

This is an R Markdown document. Markdown is a simple formatting syntax for authoring HTML, PDF, and MS Word documents. For more details
on using R Markdown see <http://rmarkdown.rstudio.com>.

When you click the **Knit** button a document will be generated that includes both content as well as the output of any embedded R code
chunks within the document. You can embed an R code chunk like this:

```{r cars}
summary(cars)
```

## Including Plots

You can also embed plots, for example:

```{r pressure, echo=FALSE}
plot(pressure)
```

Note that the `echo = FALSE` parameter was added to the code chunk to prevent printing of the R code that generated the plot.

~~~

Let us go through the default notebook that R Studio created for us.



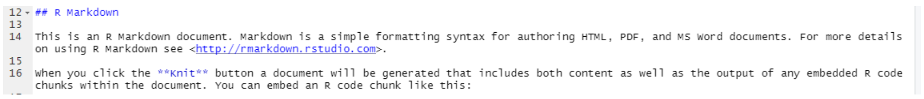
```
---
title: "Untitled"
author: "Alexander (Sasha) Pastukhov"
date: "29 10 2020"
output: html_document
---
```

The top part between two sets of `---` is a notebook header with various configuration options written in YAML (yes, we have two different languages in one file). `title`, `author`, and `date` should be self-explanatory. `output` defines what kind of output document knitr will generate. You can specify it by hand (e.g., `word_document`) or just click on drop down next to `Knit` button and pick the option you like (we will use the default HTML most of the time). These are sufficient for us but there are numerous other options that you can specify, for example, to enable indexing of headers. You can read about at yihui.org/knitr.



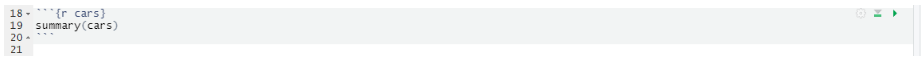
~~~
```{r setup, include=FALSE}
knitr::opts_chunk$set(echo = TRUE)
```
~~~

The next section is the "setup code chunk" that specifies default options on how the code chunks are treated by default (executed or not, their output is shown or not, etc.). By default code in chunks is run and its output is shown (`echo = TRUE`) but you can change this behavior on per-chunk basis by pressing the gear button at the top-right. The setup chunk is also a good place to import your libraries (we will talk about it later) as it is always run before any other chunks (so, even if you forgot to run it to load libraries, R Studio will do this for you).



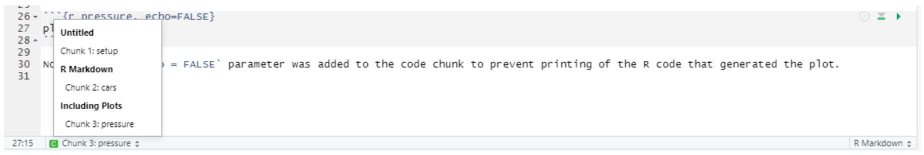
```
## R Markdown

This is an R Markdown document. Markdown is a simple formatting syntax for authoring HTML, PDF, and MS Word documents. For more details
on using R Markdown see <http://rmarkdown.rstudio.com>.

When you click the **Knit** button a document will be generated that includes both content as well as the output of any embedded R code
chunks within the document. You can embed an R code chunk like this:
```

Next, we have plain text with rmarkdown, which gets translated into formatted text when you click on `Knit` button. You can write like this anywhere outside of code chunks to explain the logic of your analysis. You should write why and how the analysis is performed but leave technical details on programming to the chunk itself, where you can comment the code.



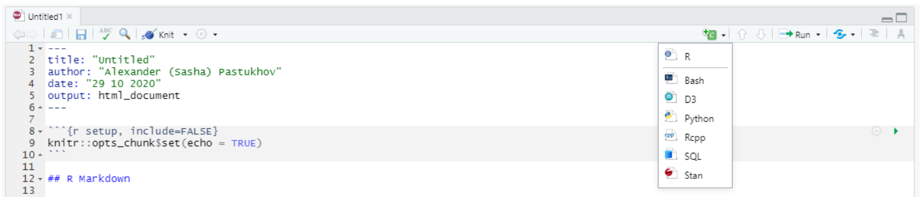
~~~
```{r cars}
summary(cars)
```
~~~

Finally, we have our first "proper" chunk of code (the "setup" chunk above is a special case). A code chunk is simply the code embedded between ` ```{r <name of the chunk} ` and the seconds set of ticks ` ``` `. Here `r` specifies that the code inside is written in R language but you can use other languages such as Python, Stan, or SQL. The `name of the chunk` is optional but I would recommend to have it, as it reminds you what this code is about and it makes it easier to navigate in large notebooks. In the bottom-left corner, you can see which chunk or section you are currently at and, if you click on it, you can quickly navigate to a different chunk. If chunks are not explicitly named, they will get labels `Chunk 1`, `Chunk 2`, etc. making it hard to distinguish them.



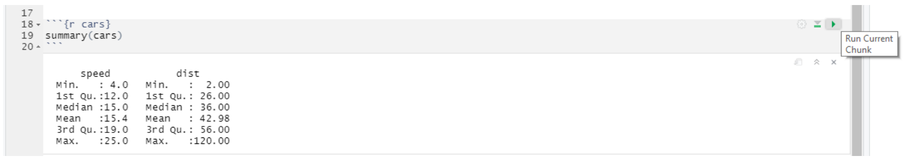
~~~
```{r pressure, echo=FALSE}
pl
``
No                  = FALSE` parameter was added to the code chunk to prevent printing of the R code that generated the plot.
~~~

There are also additional options that you can specify per chunk (whether to run the code, to show the output, what size the figures should be, etc.). Generally we won't need these options but you can get an idea about them by looking at the official manual. You can create a chunk by hand or click on "Create chunk" drop-down list (in this case, it will create the chunk at the position of the cursor)



~~~
---
title: "Untitled"
author: "Alexander (Sasha) Pastukhov"
date: "29 10 2020"
output: html_document
---

```{r setup, include=FALSE}
knitr::opts_chunk$set(echo = TRUE)
```

## R Markdown

~~~

Finally, you run **all** the code in the chunk by clicking on `Run current chunk button` at the top-right corner of the chunk or by pressing `Ctrl + Shift + Enter` when the you are inside the chunk. However, you can also run just a *single line* or only *selected lines* by pressing `Ctrl + Enter`. The cool thing about RMarkdown in RStudio is that you will see the output of that chunk right below it. This means that you can write you code chunk-by-chunk, ensure that each works as intended and only when knit the entire document. Run the chunks in your notebook to see what I mean.



~~~
```{r cars}
summary(cars)
```
~~~

```
     speed           dist       
 Min.   : 4.0   Min.   :  2.00  
 1st Qu.:12.0   1st Qu.: 26.00  
 Median :15.0   Median : 36.00  
 Mean   :15.4   Mean   : 42.98  
 3rd Qu.:19.0   3rd Qu.: 56.00  
 Max.   :25.0   Max.   :120.00  
```

1.3 Exercise

For the today's exercise, I want you to familiarize yourself with markdown. Go to markdownguide.org and look at basic and extended syntax (their cheat sheet is also very good). Write any text you want that uses all the formatting and submit the file to MS Teams.

Chapter 2

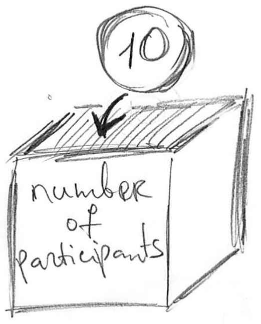
Vectors! Vectors everywhere!

Before reading the chapter, please download the exercise notebook (`Alt + Click` to download it or right-click as `Save link as...`), put it into your seminar project folder and open the project. You need both the text and the notebook with exercises to be open, as you will be switching between them.

Before we can start programming in R, you need to learn about vectors. This is a key concept in R, so your understanding of it will determine how easy it will be for you to use R. Do all of the exercises and do not hesitate to ask me whenever something is unclear. Remember, you need to master vectors before you can master R!

2.1 Variables as boxes

In programming, the concept of a variable is often described as a box you can put something in. A box has a name tag on it, which is the *name* of the variable. Whatever you put in is the *value* that you store.



This “putting in” concepts is reflected in R syntax

```
number_of_participants <- 10
```

Here, `number_of_participants` is the name of the variable (name tag for the box we will be using), `10` is the value you store, and `<-` means **“put `10` into variable `number_of_participants`”**. If you know other programming languages, you probably expected the usual assignment operator `=`. Confusingly, you can use it as well, but there are some subtle, yet important, differences in how they operate behind the scenes. We will meet `=` again when we will be talking about functions and, in particular, Tidyverse way of doing things but for now **only use `<-` operator**!

2.2 Assignment statement in detail

One **very important** thing to remember about the assignment statement `<variable> <- <value>`: the *right side* is evaluated first until the final value is established and then, and only then, it is stored in a `<variable>` specified on the left side. This means that you can use the same variable on *both* sides. Take a look at the example

```
x <- 2
print(x)
```

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

```
x <- x + 5
print(x)
```

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

We are storing value `2` in a variable `x`. In the next line, the *right side* is evaluated first. This means that the current value of `x` is substituted in its place on the right side: `x + 5` becomes `2 + 5`. This expression computed and we get `7`. Now, that the *right side* is fully evaluated, the value can be stored in `x` replacing (overwriting) the original value it had.

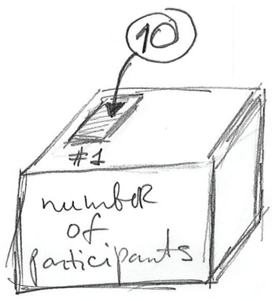
R's use of `<-` makes it easier to memorize this *right side is fully evaluated first* rule. However, as noted above, we will meet `=` operator and this one makes it look like a mathematical equation. However, assignments (storing values in a variable) have nothing in common with mathematical equations (finding values of variables to ensure equality)!

Do exercise 1.

2.3 Vectors and scalars (which are also vectors)

The box metaphor you've just learned, doesn't quite work for R. Historically, R was developed as a language for statistical computing, so it was based on concepts of linear algebra instead of being a "normal" programming language. This means that there is no conceptual divide between single values and containers (arrays, lists, dictionaries, etc.) that hold many single values. Instead, the primary data unit in R is a **vector**, which you may remember from geometry or, hopefully, from linear algebra, as an arrow that goes from 0 to a specific point in space. From computer science point of view, a vector is just a list of numbers (or some other values, as you will learn later). This means that there are no "single values" in R, there are only vectors of variable length. Special cases are vectors of length one, which are called *scalars* [1] (but they are still vectors) and zero length vectors that are, sort of, a Platonic idea of a vector without actual values. With respect to the "box metaphor", this means that we always have a box with indexed (numbered) slots in it. A simple assignment makes sure that "the box" has as many slots as the values you want to put in and stores these values one after another starting with slot #1. So, the example above `number_of_participants <- 10` creates a variable with one (1) slot and stores the value in it.

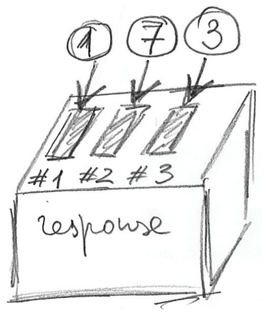
[1]Multiplication of a vector by another vector *transforms* it but for a single element vector the only transformation you can get is "scaling", hence, the name.



But, as noted above, a single value (vector with length of one) is a special case. More generally you write:

```r
response <- c(1, 7, 3)
```

Here, you create a variable (box) named `response` that has three slots in it because you want to store three values. You put values `1`, `7`, `3` into the slots #1, #2, and #3. The `c(1, 7, 3)` notation is how you create a vector in R by **c**oncatenating (or **c**ombining) values[2]. The figure below illustrates the idea:



Building on the box metaphor: If you can store something in a box, you can take it out! In the world of computers it works even better, because rather than taking something out, you just make a copy of that and store this copy somewhere else or to use it to compute things. Minimally, we would like to see what is inside of the box. For this, you can use print function:

```r
response <- c(1, 7, 3)
print(response)
```

```
## [1] 1 7 3
```

[2] I find this to be a poor choice of name but we are stuck with, so get used to it.

Or, we can make a copy of values in one variable and store them in another:

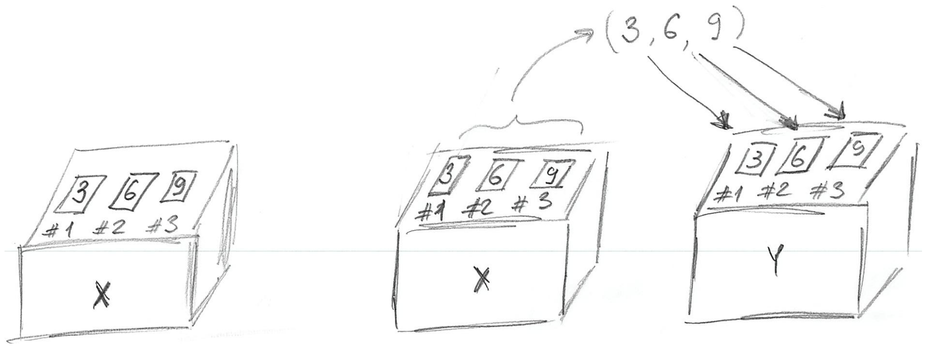
```
x <- c(3, 6, 9)
y <- x

print(x)
```

```
## [1] 3 6 9
```

```
print(y)
```

```
## [1] 3 6 9
```

Here, we create a 3-slot variable `x` so that we can put in a vector of length 3 created via concatenation `c(3, 6, 9)`. Next, we make a copy of these three values and store them in a different variable `y`. Importantly, the values in variable `x` stayed as they were. Take a look at the figure below, which graphically illustrate this:



Do exercise 2.

Remember, everything is a vector! This means that `c(3, 6, 9)` does not **c**oncatenate numbers, it **c**oncatenates three length one vectors (scalars) `3`, `6`, `9`. Thus, **c**oncatenation works on longer vectors in exactly the same way:

```
x <- c(1, 2, 3)
y <- c(4, 5)
print(c(x, y))
```

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

Do exercise 3.

2.4 Vector indexes

A vector is an ordered list of one or more values (box with one or more slots) and, sometimes, you need only one of the values. Each value (slot in the box) has its own index from 1 till N, where N is the length of the vector. To access that slot you use square brackets `some_vector[index]`. You can both get and set the value for the individual slots the same way you do it for the whole vector.

```
x <- c(1, 2, 3)
# set SECOND element to 4
x[2] <- 4

# print the entire vector
print(x)
```

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

```
# print only the third element
print(x[3])
```

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

Do exercise 4.

Unfortunately, vector indexing in R behaves in a way that may catch you by surprise. If your vectors contains five values, you would expect that index of `0` (negative indexes are discussed below) or above `5` generates an error. Not in R! Index of `0` is a special case and produces an *empty vector* (vector of zero length).

```
x <- c(1, 2, 3)
x[0]
```

```
## numeric(0)
```

If you try to get vector element using index that is larger than vector length (so `6` and above, if your vector length is 5), R will return `NA` ("Not Available" / Missing Value).

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

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

In both cases, it won't generate an error or even warn you!

When **setting** the value by index, using `0` will produce no effect, because you are trying to put a value in a vector with no slots. Oddly enough, this will generate neither an error nor a warning, so beware!

```
x <- c(1, 2, 3)
x[0] <- 5
print(x)
```

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

If you set an element those index is **larger** than vector length, the vector will be automatically expanded to that length and all the elements between the old values and the new one will be `NA` (Missing Value / Not Available).

```
x <- c(1, 2, 3)
x[10] <- 5
print(x)
```

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

This may sound too technical but I want you to learn about this because R conventions are so different from other programming languages and, also, from you would intuitively expect. If you are not aware of these highly peculiar rules, you may never realize that your code is not working properly because you will never see an error or even a warning! It should also make you more cautious and careful when programming in R. It is a very powerful language that allows you to be very flexible and expressive. Unfortunately, that flexibility means that base R won't stop you from shooting yourself in a foot. Even worse, sometimes you won't even notice that your foot is shot because R won't generate either errors or warnings, as in examples above. Good news is that things are far more restricted and consistent in Tidyverse.

Do exercise 5.

Also, you can use *negative* indexes. In that case, you **exclude** the value with that index and return or modify the rest.

```
x <- c(1, 2, 3, 4, 5)
# this will return all elements but #3
x[-3]
```

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

```
x <- c(1, 2, 3, 4, 5)
# this will assign new value (by repeating length one vector) to all elements but #2
x[-2] <- 10
x
```

```
## [1] 10  2 10 10 10
```

Given that negative indexing returns everything **but** the indexed value, what do you think will happen here?

```
x <- c(10, 20, 30, 40, 50)
x[-10]
```

Do exercise 6.

Finally, somewhat illogically, the entire vector is returned if you do not specify the index in the square brackets. Here, lack of index means "everything".

```
x <- c(10, 20, 30, 40, 50)
x[]
```

```
## [1] 10 20 30 40 50
```

2.5 Names as Index

As you've just learned, every slot in vector has its numeric (integer) index. However, this number only indicates an index of a slot but tells you nothing on how it is conceptually different from a slot with a different index. For example, if we are storing width and height in a vector, remembering their order may be tricky: was it `box_size <- c(<width>, <depth>, <height>)` or `box_size <- c(<height>, <width>, <depth>)`? Similarly, looking at `box_size[1]` tells that you are definitely using the *first* dimension but is it `height` or `width` (or `depth`)?

In R, you can use *names* to supplement numeric indexes. It allows you add meaning to a particular vector index, something that becomes extremely important when we use it for tables. There are two ways to assign names to indexes, either when you are creating the index via `c()` function or, afterwards, via names() function.

To create named vector via `c()` you specify a name before each value as `c(<name1> = <value1>, <name2> = <value2>, ...)`:

```r
box_size <- c("width"=2, "height"=4, "depth"=1)
print(box_size)
```

```
##  width height  depth
##      2      4      1
```

Note the names appearing above each value. You can now use either numeric index or name to access the value.

```r
box_size <- c("width"=2, "height"=4, "depth"=1)
print(box_size[1])
```

```
## width
##     2
```

```r
print(box_size["depth"])
```

```
## depth
##     1
```

Alternatively, you can use names() function to both get and set the names. The latter works via a *very counterintuitive* syntax `names(<vector>) <- <vector-with-names>`

```r
# without names
box_size <- c(2, 4, 1)
print(box_size)
```

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

```r
# with names
names(box_size) <- c("width", "height", "depth")
print(box_size)
```

```
##  width height  depth
##      2      4      1
```

```r
# getting all the names
print(names(box_size))
```

```
## [1] "width"  "height" "depth"
```

Because everything is a vector, `names(<vector>)` is also a vector, meaning that you can get or set just one element of it.

```
box_size <- c("width"=2, "height"=4, "depth"=1)

# modify SECOND name
names(box_size)[2] <- "HEIGHT"
print(box_size)
```

```
##  width HEIGHT  depth 
##      2      4      1
```

Finally, if you use a name that is not in the index, this is like using numeric index larger than the vector length. Is in out-of-range numeric index, there will be neither error not warning and you will get an `NA` back.

```
box_size <- c("width"=2, "height"=4, "depth"=1)
print(box_size["radius"])
```

```
## <NA> 
##   NA
```

Do exercise 7.

2.6 Slicing

So far we were reading or modifying either the whole vector or just one of its elements. However, the index you pass in square brackets (you've guess it!) is also a vector! Which means that you can construct a vector of indexes the same way you construct a vector of any values (the only restriction is that index values must integers and that you cannot mix negative and positive indexes).

```
x <- c(10, 20, 30, 40, 50)
x[c(2, 3, 5)]
```

```
## [1] 20 30 50
```

When constructing a vector index, you can put the index values in the order you require (starting from the end of it, random order, etc.) or use the same index more than once.

```
x <- c(10, 20, 30, 40, 50)
x[c(3, 5, 1, 1, 4)]
```

```
## [1] 30 50 10 10 40
```

You can also use several negative indexes to exclude multiple values and return the rest. Here, neither order nor the duplicate indexes matter. Regardless of which value you exclude first or how many times you exclude it, you still get *the rest* of the vector in its default order.

```
x <- c(10, 20, 30, 40, 50)
x[c(-4, -2, -2)]
```

```
## [1] 10 30 50
```

Note that you cannot mix positive and negative indexes as R will generate an error (at last!).

```
x <- c(10, 20, 30, 40, 50)

# THIS WILL GENERATE AN ERROR:
# "Error in x[c(-4, 2, -2)] : only 0's may be mixed with negative subscripts"
x[c(-4, 2, -2)]
```

Finally, including zero index makes no difference but generates neither an error nor a warning.

```
x <- c(10, 20, 30, 40, 50)
x[c(1, 0, 5, 0, 0, 2, 2)]
```

```
## [1] 10 50 20 20
```

You can also use names instead of numeric indexes.

```
box_size <- c("width"=2, "height"=4, "depth"=1)
print(box_size[c("height", "width")])
```

```
## height  width
##      4      2
```

However, you cannot mix numeric indexes and names. The reason is that a vector can hold only values of one type (more on that during the next seminar), so all numeric values will be converted to text (`1` will become `"1"`) and treated as names rather than indexes.

2.7 Colon Operator and Sequence Generation

To simplify vector indexing, R provides you with a shortcut to create a range of values. An expression `A:B` (a.k.a.Colon Operator) builds a sequence of integers starting with `A` and ending with **and including(!)** `B` (the latter is not so obvious, if you come from Python).

```
3:7
```

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

Thus, you can use it to easily create an index and, because everything is a vector!, combine it with other values.

```
x <- c(10, 20, 30, 40, 50)
x[c(1, 3:5)]
```

```
## [1] 10 30 40 50
```

The sequence above is increasing but you can also use the colon operator to construct a decreasing one.

```
x <- c(10, 20, 30, 40, 50)
x[c(5:2)]
```

```
## [1] 50 40 30 20
```

The colon operator is limited to sequences with steps of `1` (if end value is larger than the start value) or `-1` (if end value is smaller than the start value). For more flexibility you can use Sequence Generation function: `seq(from, to, by, length.out)`. The `from` and `to` are starting and ending values (just like in the colon operator) and you can specify either a step via `by` parameter (as, in “from A to B by C”) or via `length.out` parameter (how many values you want to generate, effectively `by = ((to - from)/(length.out - 1))`. Using `by` version:

```
seq(1, 5, by=2)
```

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

Same sequence but using `length.out` version:

```
seq(1, 5, length.out=3)
```

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

You have probably spotted the `=` symbol. Here, it is not an assignment but is used to specify values of parameters when you call a function. Thus, we are still sticking with `<-` **outside** of the function calls but are using `=` **inside** the function calls.

Do exercise 8.

2.8 Working with two vectors of *equal* length

You can also use mathematical operations on several vectors. Here, the vectors are matched element-wise. Thus, if you add two vectors of *equal* length, the *first* element of the first vector is added to the *first* element of the second vector, *second* element to *second*, etc.

```
x <- c(1, 4, 5)
y <- c(2, 7, -3)
z <- x + y
print(z)
```

```
## [1]  3 11  2
```

Do exercise 9.

2.9 Working with two vectors of *different* length

What if vectors are of *different* length? If the length of the longer vector is a *multiple* of the shorter vector length, the shorter vector is repeated N-times (where $N = length(longer\ vector)/length(shorter\ vector)$) and this length-matched vector is then used for the mathematical operation. Take a look at the results of the following computation

```
x <- 1:6
y <- c(2, 3)
print(x + y)
```

```
## [1] 3 5 5 7 7 9
```

Here, the values of `y` were repeated three times to match the length of `x`, so the actual computation was `c(1, 2, 3, 4, 5, 6) + c(2, 3, 2, 3, 2, 3)`. A vector of length 1 (scalar) is a special case because any integer is a multiple of 1, so that single value is repeated `length(longer_vector)` times before the operation is performed.

```
x <- 1:6
y <- 2
print(x + y)
```

```
## [1] 3 4 5 6 7 8
```

Again, the actual computation is `c(1, 2, 3, 4, 5, 6) + c(2, 2, 2, 2, 2, 2)`.

If the length of the longer vector **is not** a multiple of the shorter vector length, R will repeat the shorter vector N times, so that $N = ceiling(length(longer\ vector)/length(shorter\ vector))$ (where ceiling() rounds a number up) and truncates (throws away) extra elements it does not need. Although R will do it, it will also issue a warning about mismatching objects' (vectors') lengths.

```
x <- c(2, 3)
y <- c(1, 1, 1, 1, 1)
print(x + y)
```

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

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

Finally, combining any vector with null length vector produces a null length vector.

```
x <- c(2, 3)
y <- c(1, 1, 1, 1, 1)
print(x + y[0])
```

```
## numeric(0)
```

One thing to keep in mind: R does this length-matching-via-vector-repetition automatically and shows a warning only if two lengths are not multiples of each other. This means that vectors will be matched by length even if that was not your plan. E.g, imagine that your vector that contains experimental condition (e.g. contrast of the stimulus) is about all ten blocks that participants performed but your vector with responses is, accidentally, only on block #1. R will **silently(!)** replicate the responses 10 times to match their length without ever telling you to watch out. Thus, do make sure that your vectors are matched in their length, so that you are not caught surprised by this behavior (you can use function length() for this). Good news, it is much more strict in Tidyverse, which is designed to make shooting yourself in the foot much harder.

Do exercise 10.

2.10 Applying functions to a vector

Did I mention that everything is a vector? This means that when you are using a function, you are always applying it to (mapping it on) a vector. This, in turn, means that you apply the function to **all values** in one go. For example, you can compute a cosine of all values in the vector.

```
cos(c(0, pi/4, pi/2, pi, -pi))
```

```
## [1]  1.000000e+00  7.071068e-01  6.123032e-17 -1.000000e+00 -1.000000e+00
```

In contrast, in Python or C, you would need to loop over the values and compute the cosine for one value at a time (matrix-based NumPy library is a different story). Or think about Excel, where you need to extend formula over the rows but each row is computed independently (so you can deliberately or accidentally miss some rows). In R, because everything is the vector, the function is applied to every value automatically. Similarly, if you are using aggregating functions, such as `mean()` and `max()`, you can pass a vector and it will return a length-one vector with the value.

```
mean(c(1, 2, 6, 9))
```

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

2.11 Wrap up

By now you have learned more about vectors, vector indexing, and vector operations in R than you probably bargained for. Admittedly, not the most exciting topic. On top of that, there was not a single word on psychology, data analysis, or statistics! However, R is obsessed with vectors (everything is a vector!) and understanding them will make it easier to understand lists (a polyamorous cousin of a vector), tables (special kind of lists made of vectors) and functional programming. Finish this seminar by doing remaining exercises. Let's see whether R can still surprise you!

Chapter 3

Tables and Tibbles (and Tribbles)

Please download the exercise notebook (`Alt + Click` to download it or right-click as `Save link as...`), put it into your seminar project folder and open the project. You need both the text and the notebook with exercises to be open, as you will be switching between them.

3.1 Primary data types

Last time we talked about the fact that everything is a vector in R. All the examples used numeric vectors which are two of the four primary types in R.

- Real numbers (double precision floating point numbers) that can be written in decimal notation with or without a decimal point (`123.4` or `42`) or in a scientific notation (`3.14e10`). There are two special values specific to the real numbers: `Inf` (infinity) and `NaN` (not a number). The latter looks similar `NA` (Not Available / Missing Value) but is a different special case.
- Integer numbers that can be specified by adding `L` to the end of an integer number `5L`. Without that `L` a *real* value will be created (`5` would be stored as `5.0`).
- Logical or Boolean values of `TRUE` (also written as `T`) and `FALSE` (also written as `F`).
- Character values (strings) that hold text between a pair of matching `"` or `'` characters. The two options mean that you can surround your text by `'` if you need to put a quote inside: `'"I have never let my schooling interfere with my education." Mark Twain'` or by `"` if you need an apostrophe `"participant's response"`.

You can convert from one type to another and check whether a particular vector is of specific type. Note that if a vector cannot be converted to a specified type, it is "converted" to `NA` instead.

- to integer via `as.integer()` and `is.integer`. When converting
  - from a real number the fractional part is discarded, so `as.integer(1.8)` $\rightarrow$ `1` and `as.integer(-2.1)` $\rightarrow$ `2`
  - from logical value `as.integer(TRUE)` $\rightarrow$ 1 and `as.integer(FALSE)` $\rightarrow$ 0
  - from string only it is a properly formed number, e.g. `as.integer("12")` $\rightarrow$ `12` but `as.integer("_12_")` is `NA`. Note that a real number string is converted first to a real number and then to an integer so `as.integer("12.8")` $\rightarrow$ `12`.
  - from `NA` $\rightarrow$ `NA`
- to real number via `as.numeric()` / `as.double()` and `is.double()` (avoid `is.numeric()` as Hadley Wickham writes that it is not doing what you would think it should).
  - from logical value `as.double(TRUE)` $\rightarrow$ 1.0 and `as.double(FALSE)` $\rightarrow$ 0.0
  - from string only it is a properly formed number, e.g. `as.double("12.2")` $\rightarrow$ `12` but `as.double("12punkt5")` is `NA`
  - from `NA` $\rightarrow$ `NA`
- to logical `TRUE/FALSE` via `as.logical` and `is.logical()`.
  - from integer or real, zero (`0` or `0.0`) is `FALSE`, any other non-zero value is `TRUE`
  - from a string, it is `TRUE` for `"TRUE"`, ”`True"`, `"true"`, or `"T"` but `NA` if `"t"` `"TRue"`, `"truE`, etc. Same goes for `FALSE`.
  - from `NA` $\rightarrow$ `NA`
- to a character string via `as.character()` and `is.character()`
  - numeric values are converted to a string representation with scientific notation being used for large numbers.
  - logical `TRUE/T` and `FALSE/T` are converted to `"TRUE"` and `"FALSE"`.
  - `NA` $\rightarrow$ `NA`

Do exercise 1.

3.2 In vector all values must be of the same type

All values in a vector must be of the same type - all integer, all double, all logical, or all strings. This ensures that you can apply the same function or operation to the entire vector without worrying about type compatibility. This means, however, that you cannot mix different value types in a vector. If you do try to concatenate vectors of different types, all values will be converted to a more general / flexible type. Thus, if you mix numbers and logical values, you will end up with a vector of numbers. Mixing anything with strings will convert the entire vector to string. Mixing in `NA` does not change the vector type.

Do exercise 2.

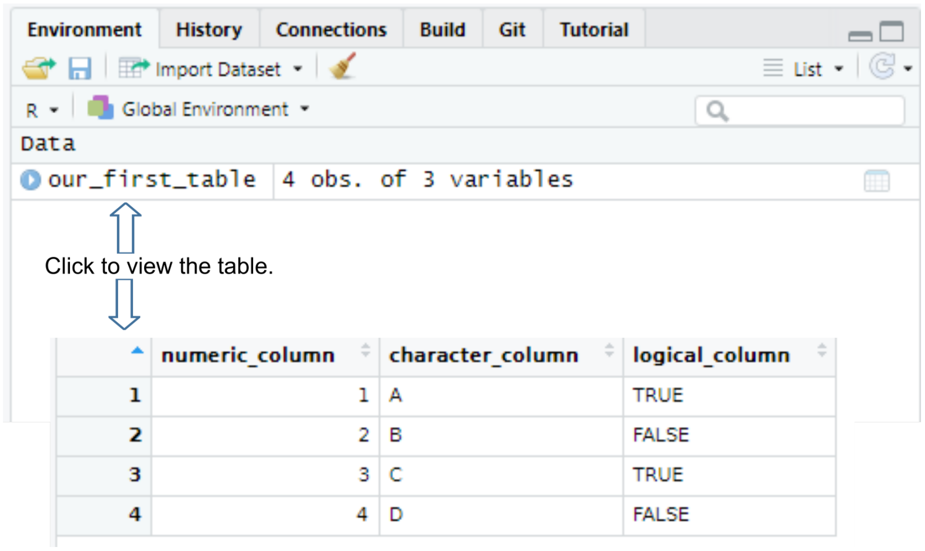
3.3 Tables, a.k.a. data frames

We have spent so much time on vectors because a data table is merely a vector (well, technically, a list) of vectors, with each vector as a column. The default way to construct a table, which are called *data frames* in R, is via data.frame() function.

```
our_first_table <- data.frame(numeric_column = c(1, 2, 3, 4),
                              character_column = c("A", "B", "C", "D"),
                              logical_column = c(TRUE, F, T, FALSE))

our_first_table
```

```
##   numeric_column character_column logical_column
## 1              1                A           TRUE
## 2              2                B          FALSE
## 3              3                C           TRUE
## 4              4                D          FALSE
```

Once you create a table, it will appear in your environment, so you can see it in the *Environment* tab and view it by clicking on it or typing `View(table_name)` in the console (not the capital V in the `View()`).



```
our_first_table   4 obs. of 3 variables
```

Click to view the table.

	numeric_column	character_column	logical_column
1	1	A	TRUE
2	2	B	FALSE
3	3	C	TRUE
4	4	D	FALSE

Do exercise 3.

Because all columns in a table **must** have the same number of rows. This is similar to the process of matching vectors' length that you have learned the last time. However, it works automatically only if length of *all* vectors is a multiple of the longest length. Thus, the example below will work, as the longest vector (`numeric_column`) is 6, `character_column` length is 3, so it will be repeated twice, and `logical_column` length is 2 so it will be repeated thrice.

```r
the_table <- data.frame(numeric_column = 1:6,                # length 6
                        character_column = c("A", "B", "C"),   # length 3
                        logical_column = c(TRUE, FALSE))       # length 2
the_table
```

```
##   numeric_column character_column logical_column
## 1              1                A           TRUE
## 2              2                B          FALSE
## 3              3                C           TRUE
## 4              4                A          FALSE
## 5              5                B           TRUE
## 6              6                C          FALSE
```

If the simple *multple-of-length* rule does not work, R (finally!) generates an error.

```
# this will generate an error: arguments imply differing number of rows
the_table <- data.frame(numeric_column = 1:7,                   # length 7
                        character_column = c("A", "B", "C"))  # length 3, cannot be multiplied by
```

Do exercise 4.

Just as with the vectors, you can extract or modify only some elements (rows, columns, subsets) of a table. There are several ways to do it, as you can extract individual elements, individual columns, individual rows, or some rows and some columns. Subsetting tables is not the most exciting and fairly confusing topic but you need to understand it as in the R code that you will encounter different notations could be used interchangeably.

3.4 Extracting a single vector / column

To access individual *vectors* (a.k.a. columns or variables) use dollar notation `table$column_name` (this should be your default way) or **double** square brackets, `table[[column_name]]` or `table[[column_index]]`. Note that you cannot use multiple indexes (`c(1, 2)`) or slicing (e.g., `1:2`) with double square brackets. The important if subtle detail is that this notation returns a *vector*, just a like one that you create via `c()` function.

```
our_first_table <- data.frame(numeric_column = c(1, 2, 3),
                              character_column = c("A", "B", "C"),
                              logical_column = c(TRUE, F, T))

# via $ notation
our_first_table$numeric_column
```

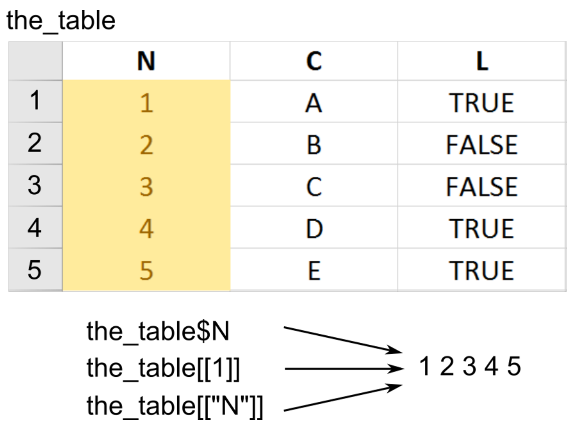
```
## [1] 1 2 3
```

```
# via name and double square brackets
our_first_table[['numeric_column']]
```

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

```
# via index and double square brackets
our_first_table[[1]]
```

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



	N	C	L
1	1	A	TRUE
2	2	B	FALSE
3	3	C	FALSE
4	4	D	TRUE
5	5	E	TRUE

3.5 Extracting part of a table

Alternatively, you can extract or access a rectangular part of the table via **single** square brackets. To get one or more columns you can use their names or indexes just as you did with vectors.

```
our_first_table <- data.frame(numeric_column = c(1, 2, 3),
                              character_column = c("A", "B", "C"),
                              logical_column = c(TRUE, F, T))

# via index
our_first_table[1]
```

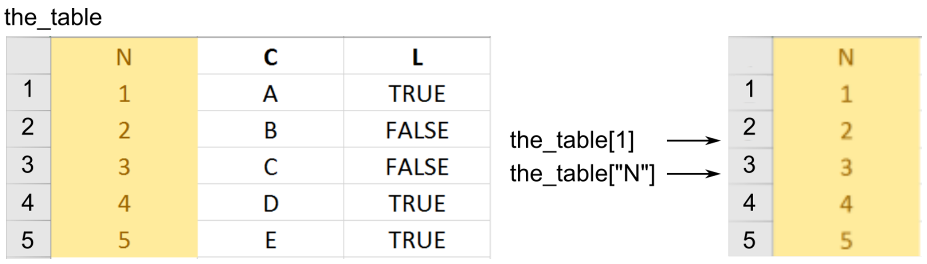
```
##   numeric_column
## 1              1
## 2              2
## 3              3
```

```
# via name
our_first_table['numeric_column']
```

```
##   numeric_column
## 1              1
## 2              2
## 3              3
```

```
# via slicing
our_first_table[1:2]
```

```
##   numeric_column character_column
## 1              1                A
## 2              2                B
## 3              3                C
```



Again, note that first two call get you a single-column table not a vector. You still need to use the column name to access its values.

```
our_first_table <- data.frame(numeric_column = c(1, 2, 3),
                              character_column = c("A", "B", "C"),
                              logical_column = c(TRUE, F, T))

table_copy <- our_first_table[1]
table_copy$numeric_column
```

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

To select a subset rows and columns you write `table[rows, column]`. If you omit either rows or columns this implies that you want *all* rows or columns.

```
our_first_table <- data.frame(numeric_column = c(1, 2, 3),
                              character_column = c("A", "B", "C"),
                              logical_column = c(TRUE, F, T))

# getting ALL rows for the FIRST column -> this gives you a VECTOR
our_first_table[, 1]
```

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

```
# getting FIRST row for ALL columns -> this gives you DATA.FRAME
our_first_table[1, ]
```

```
##   numeric_column character_column logical_column
## 1              1                A           TRUE
```

```r
# ALL rows and ALL columns, equivalent to just writing `our_first_table` or `our_first_
our_first_table[,]
```

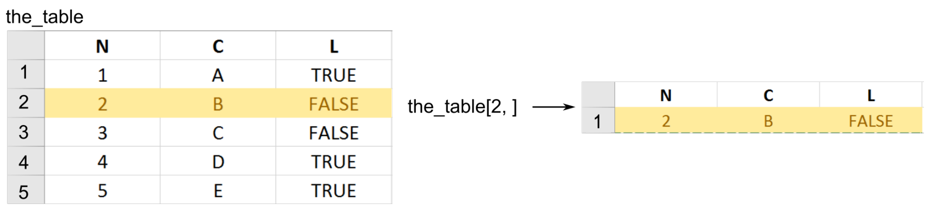
```
##   numeric_column character_column logical_column
## 1              1                A           TRUE
## 2              2                B          FALSE
## 3              3                C           TRUE
```

```r
# getting SECOND element of the THIRD column
our_first_table[2, 3]
```

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

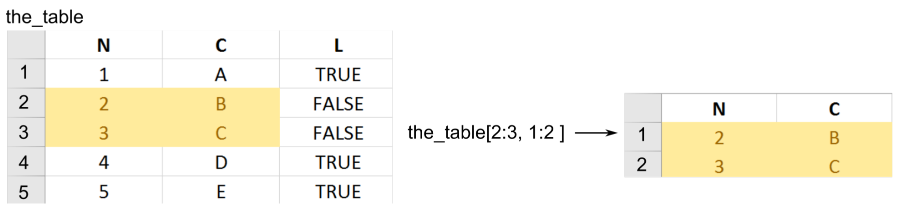
```r
# getting first two elements of the logical_column
our_first_table[1:2, "logical_column"]
```

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



	N	C	L
1	1	A	TRUE
2	2	B	FALSE
3	3	C	FALSE
4	4	D	TRUE
5	5	E	TRUE

	N	C	L
1	2	B	FALSE



	N	C	L
1	1	A	TRUE
2	2	B	FALSE
3	3	C	FALSE
4	4	D	TRUE
5	5	E	TRUE

	N	C
1	2	B
2	3	C

Do exercise 5.

3.6 Tibble, a better data.frame

3.7 Tribble, table from text

3.8 Reading example tables

3.9 Reading tables from disk