Data analysis using R for Psychology

Alexander (Sasha) Pastukhov

2020-11-25

Contents

3   Tables and Tibbles (and Tribbles)   37

4   Grammar of Graphics   55

Introduction

This is a material for *Applied data analysis for psychology using the open-source software R* seminar as taught at Institute of Psychology at University of Bamberg.

About the 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.
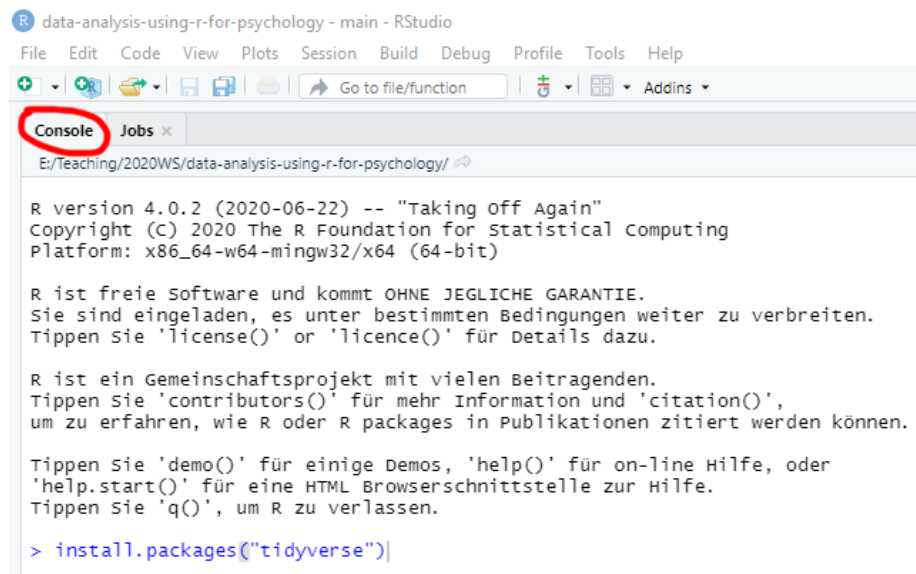
Installing RTools

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`.

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`.
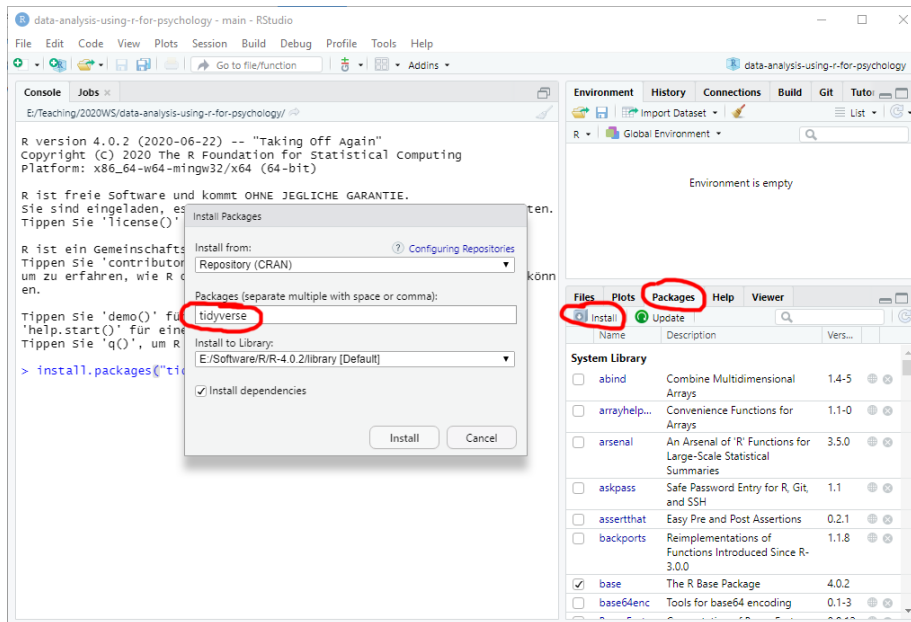
In some cases, R will ask whether you want to install packages from source. In this case, it will grab the source code and compile the package, which takes time and requires RTools. In most cases, you can say "No" to install a pre-build binary version. The binary version will be slightly outdated but 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).
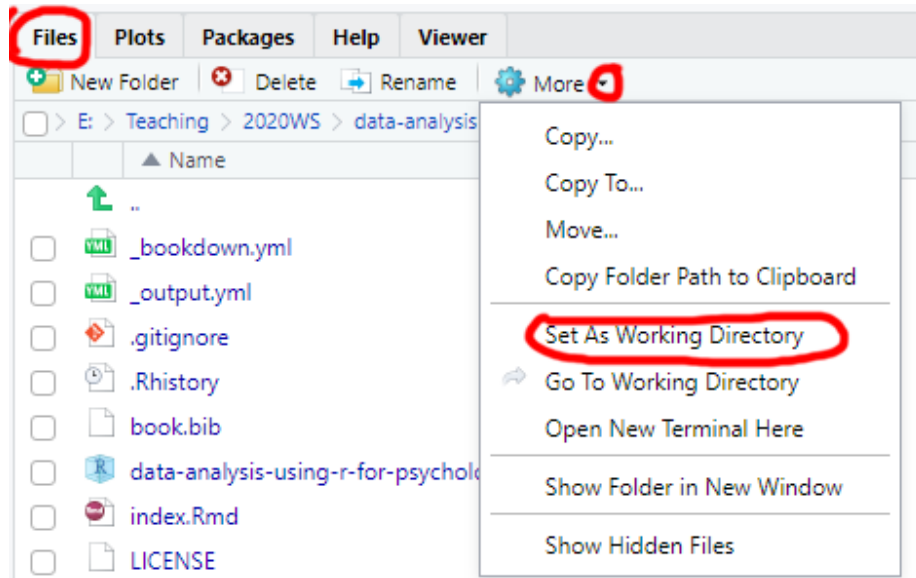
In short, you may think that you are working in a particular folder but R will have its own opinion about this. Whenever this happens, it is really confusing and involves a lot of cursing as R cannot find files that you clearly see with your own eyes. To avoid this you should organize any project or 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).
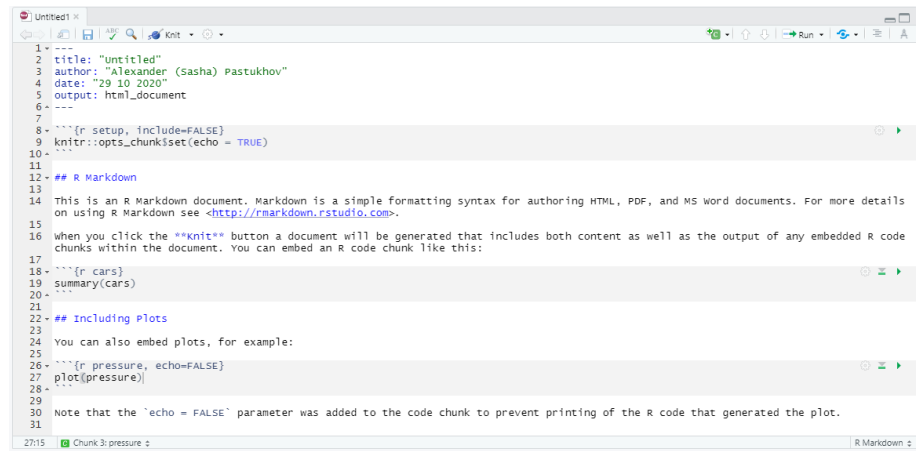
Let us go through the default notebook that R Studio created for us.

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.

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).
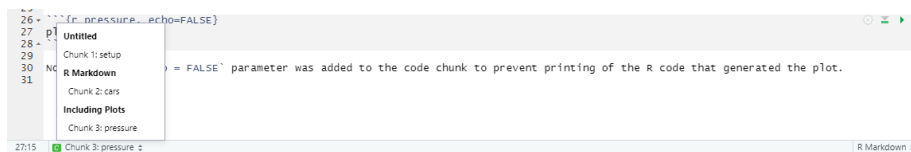
Next, we have plain text with rmarkdown, which gets translated into formatted text when you click on `Knit` button. You can write like this anywhere outside of code chunks to explain the logic of your analysis. You should write why and how the analysis is performed but leave technical details on programming to the chunk itself, where you can comment the code.
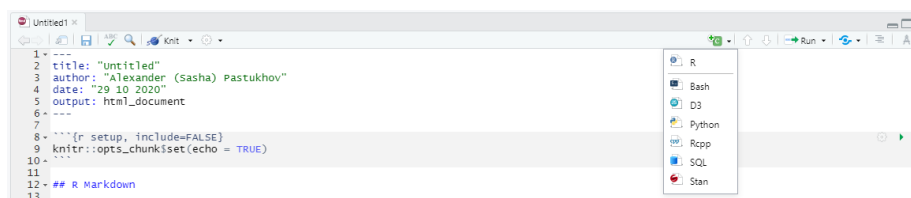
```
18 - ```{r cars}
19   summary(cars)
20 - ```
21
```

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.

```
26 - ```{r pressure, echo=FALSE}
27   p|   Untitled
28 - `
29        Chunk 1: setup
30   No   R Markdown        ) = FALSE` parameter was added to the code chunk to prevent printing of the R code that generated the plot.
31
          Chunk 2: cars
          Including Plots
          Chunk 3: pressure

27:15   Chunk 3: pressure                                                                R Markdown
```
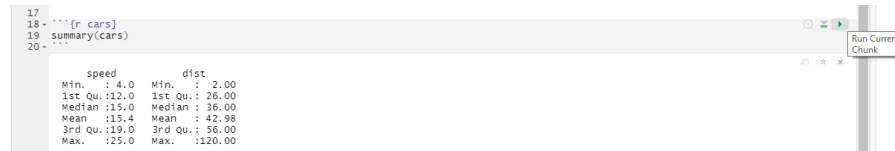
There are also additional options that you can specify per chunk (whether to run the code, to show the output, what size the figures should be, etc.). Generally we won't need these options but you can get an idea about them by looking at the official manual. You can create a chunk by hand or click on "Create chunk" drop-down list (in this case, it will create the chunk at the position of the cursor)

```
Untitled1

Knit

1 - ---
2   title: "untitled"                    R
3   author: "Alexander (Sasha) Pastukhov"
4   date: "29 10 2020"                    Bash
5   output: html_document
6 - ---                                   D3
7
8 - ```{r setup, include=FALSE}           Python
9   knitr::opts_chunk$set(echo = TRUE)
10 - ```                                  Rcpp
11
12 - ## R Markdown                        SQL
13
                                          Stan
```

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.

```
17
18 - ```{r cars}
19  summary(cars)
20 - ```
```
```
         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.

21

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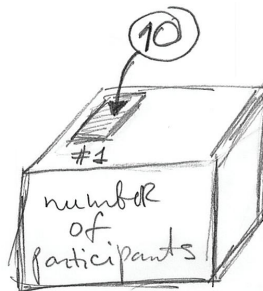
But, as noted above, a single value (vector with length of one) is a special case. More generally you write:

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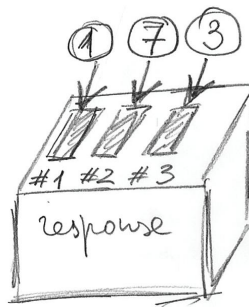
Building on the box metaphor: If you can store something in a box, you can take it out! In the world of computers it works even better, because rather than taking something out, you just make a copy of that and store this copy somewhere else or to use it to compute things. Minimally, we would like to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

```
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:
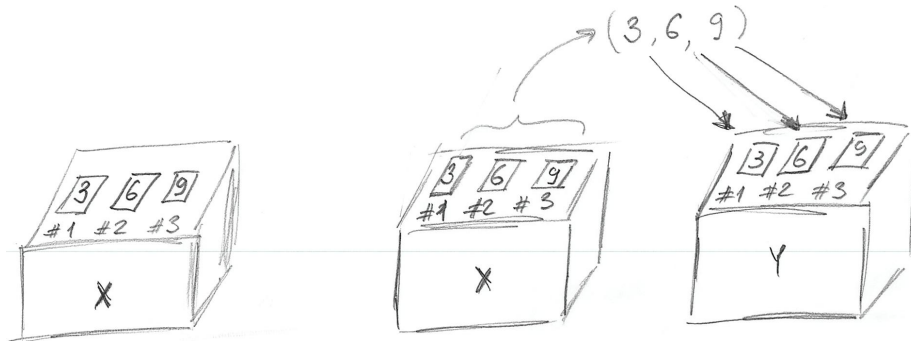
Do exercise 2.

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.

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

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

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

```r
# 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).

```r
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).

```r
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!

```r
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).

```r
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.

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

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

```r
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?

```r
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".

```r
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 im-
portant 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.

```r
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.

```r
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).

```r
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)` → 1 and `as.integer(-2.1)` → 2
  - from logical value `as.integer(TRUE)` → 1 and `as.integer(FALSE)` → 0
  - from string only it is a properly formed number, e.g. `as.integer("12")` → 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")` → 12.
  - from `NA` → `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)` → 1.0 and `as.double(FALSE)` → 0.0
  - from string only it is a properly formed number, e.g. `as.double("12.2")` → 12.2 but `as.double("12punkt5")` is `NA`
  - from `NA` → `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` → `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` → `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()`).

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.

```r
# 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 b
```

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.

```r
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
```

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

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

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

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

the_table

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

the_table$N
the_table[[1]] ⟶ 1 2 3 4 5
the_table[["N"]]

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.

```r
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
```

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

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

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

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

the_table

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

the_table[1] ⟶
the_table["N"] ⟶

	N
1	1
2	2
3	3
4	4
5	5

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
```

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

the_table

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

the_table[2,] ⟶

	N	C	L
1	2	B	FALSE

the_table

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

the_table[2:3, 1:2] ⟶

	N	C
1	2	B
2	3	C

Do exercise 5.

3.6   Using libraries

There is a better way to construct a table but to use it, we need to first import a library that implements it. As with most modern programming languages, the real power of R is not what comes bundled with it (very little, as a matter of fact) but community developed libraries that extend it. We already discussed how you install libraries. For that you use library() function (it has a sister function `require()` but it should be used inside functions and packages not in scripts or notebooks). So, to use *tidyverse* library that you already installed, you simply write

```r
library(tidyverse)
# or
library("tidyverse")
```

One thing to keep in mind is that if you import two libraries that have a function with same name, the function from the *latter* package will overwrite (mask) the function from the former. You will get a warning but if you miss it, it may be very confusing. My favorite stumbling block are functions `filter()` from `dplyr` package (we will use it extensively, as it filters a table by row) and `filter()` function from `signal` package (applies a filter to a time-series). This overwriting of one function by another can lead to very odd looking mistakes. In my case I though I was using `dplyr::filter()` and could not understand the error message I was getting, it it took me an hour to figure it out the first time. Here are the warnings I should have paid attention to.

```r
library(dplyr)
```

```
##
## Attaching package: 'dplyr'
```

```
## The following objects are masked from 'package:stats':
##
##     filter, lag
```

```
## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

```r
library(signal)
```

```
##
## Attaching package: 'signal'
```

```
## The following object is masked from 'package:dplyr':
##
##     filter
```

```
## The following objects are masked from 'package:stats':
##
##     filter, poly
```

Thus, keep that in mind or, better still, explicitly mention which package the function is coming from via `library::function()` notation. In this case, you will use the function that you are interested in and need not to worry about other functions with the same name that may conflict with it. In general, it is a good idea to *always* disambiguate function via library but in practice it may make your code hard to read by littering it with `library::` prefixes. Thus, you will need to find a balance between disambiguation and readability.

```r
library(tibble)

# imported from the library into the global environment
print(tribble(~a, 1))
```

```
## # A tibble: 1 x 1
##       a
##   <dbl>
## 1     1
```

```r
# used directly from the package
tibble::tribble(~a, 1)
```

```
## # A tibble: 1 x 1
##       a
##   <dbl>
## 1     1
```

When using a notebook (so, in our case, always) put the libraries into *setup* chunk of the notebook. This ensures that your libraries are always initialized, even if you first run some other chunk. Word of advice, keep you library list in alphabetical order. Because libraries are very specialized, you will need quite a few of them for a typical analysis. Keeping them alphabetically organized makes it easier to see whether you imported the required library and whether you need to install a new one.

3.7   Tibble, a better data.frame

Although the `data.frame()` function is the default way of creating a table, it is a legacy implementation with numerous shortcomings. Tidyverse implemented its own version of the table called tibble() that provides a more rigorous control and more consistent behavior. For example, it allows you to use any symbols for the columns names (including spaces), prints out only beginning of the table rather than entire table, etc. It also gives more warnings. If you try to access a non-existing column both `data.frame()` and `tibble()` will return `NULL` but the former will do it silently, whereas the latter will complain.

```
library(tibble)

# data.frame will return NULL silently
df <- data.frame(b = 1)
print(df$A)
```

```
## NULL
```

```
# data.frame will return NULL for a variable that does not exist

tbl <- tibble(b = 1)
print(tbl$A)
```

```
## Warning: Unknown or uninitialised column: `A`.
```

```
## NULL
```

In short, `tibble()` provides a more robust version of a `data.frame` but other-
wise behaves (mostly) identically to it. Thus, it should be your default choice
for a table.

3.8 Tribble, table from text

The tibble package also provides an easier to read way of constructing tables
via the tribble() function. Here, you use tilde to specify column names, and
then write the content row-by-row.

```
tribble(
    ~x, ~y,
    1,  "a",
    2,  "b"
)
```

```
## # A tibble: 2 x 2
##       x y
##   <dbl> <chr>
## 1     1 a
## 2     2 b
```

Do exercise 6.

3.9   Reading example tables

One of the great things about R is that most packages come with an example data set that illustrates their function. You can see the list of some of them here. In case of example data set, you need to import the library it is part of and then load them by writing `data(tablename)`. For example, to use use `mpg` data on fuel economy from `ggplot2` package, you need to import the library first, and then call `data(mpg)`.

```r
library(ggplot2)
data(mpg) # this create a "promise" of the data
print(mpg) # any action on the promise leads to data appearing in the environment
```

```
## # A tibble: 234 x 11
##    manufacturer model   displ year  cyl trans   drv    cty  hwy fl    class
##    <chr>        <chr>   <dbl> <int> <int> <chr>   <chr> <int> <int> <chr> <chr>
##  1 audi         a4        1.8  1999    4 auto(l~ f        18   29 p     comp~
##  2 audi         a4        1.8  1999    4 manual~ f        21   29 p     comp~
##  3 audi         a4        2    2008    4 manual~ f        20   31 p     comp~
##  4 audi         a4        2    2008    4 auto(a~ f        21   30 p     comp~
##  5 audi         a4        2.8  1999    6 auto(l~ f        16   26 p     comp~
##  6 audi         a4        2.8  1999    6 manual~ f        18   26 p     comp~
##  7 audi         a4        3.1  2008    6 auto(a~ f        18   27 p     comp~
##  8 audi         a4 quat~  1.8  1999    4 manual~ 4        18   26 p     comp~
##  9 audi         a4 quat~  1.8  1999    4 auto(l~ 4        16   25 p     comp~
## 10 audi         a4 quat~  2    2008    4 manual~ 4        20   28 p     comp~
## # ... with 224 more rows
```

3.10   Reading csv files

So far we covered creating a table by hand via `data.frame()`, `tibble()`, or `tribble()` functions and loading an example table from a package via `data()` function. More commonly, you will need to read a table from an external file. These files can come in many formats because they are generated by different experimental software. Below, you will see how to handle those but my recommendation is to always store your data in a csv (Comma-separated values) files. These are simple plain text files, which means you can open them in any text editor, with each line representing a single row (typically, top row contains column names) with individual columns separated by some symbol or symbols. Typical separators are a comma (hence, the name), a semicolon (this is frequently used in Germany, with comma serving as a decimal point), a tabulator, or even a space symbol. Here is an example of such file

```
Participant,Block,Trial,Contrast,Correct
A1,1,1,0.5,TRUE
A1,1,2,1.0,TRUE
A1,1,2,0.05,FALSE
...
```

that is turned into a table when loaded

	Participant	Block	Trial	Contrast	Correct
1	A1	1	1	0.50	TRUE
2	A1	1	2	1.00	TRUE
3	A1	1	2	0.05	FALSE

There are several ways of reading CSV files in R. The default way by using read.csv() function that comes has different versions optimized for different combinations of the decimal point and separator symbols, e.g. `read.csv2()` assumes a comma for the decimal point and semicolon as a separator. However, a better way is to use readr library that implements same functions. Names of the functions are slightly different with underscore replacing the dot, so `readr::read_csv()` is a replacement for `read.csv()`. These are faster (although it will be noticeable only on large data sets), do not convert text to factor variables (we will talk about factors later but this default conversion by `read.csv()` can be very confusing), etc.

However, most important difference between `read.csv()` and `read_csv()` is they constraint the content of a CSV file. `read.csv()` has not assumptions about which columns are in the file and what their value types are. It simply reads them as is, *silently* guessing their type.

```
results <- read.csv("data/example.csv")
results
```

```
##   Participant Block Trial Contrast Correct
## 1          A1     1     1     0.50    TRUE
## 2          A1     1     2     1.00    TRUE
## 3          A1     1     2     0.05   FALSE
```

You can use `read_csv()` the same way and it will work the same way but warn you about the table structure it deduced.

```
results <- readr::read_csv("data/example.csv")
```

```
##
## -- Column specification --------------------------------------------------
```

```
## cols(
##   Participant = col_character(),
##   Block = col_double(),
##   Trial = col_double(),
##   Contrast = col_double(),
##   Correct = col_logical()
## )
```

```
results
```

```
## # A tibble: 3 x 5
##   Participant Block Trial Contrast Correct
##   <chr>       <dbl> <dbl>    <dbl> <lgl>
## 1 A1              1     1     0.5  TRUE
## 2 A1              1     2     1    TRUE
## 3 A1              1     2     0.05 FALSE
```

This annoying *Column specification* print out, which gets even more annoying if you need to read many CSV files, is there for a reason: it wants to annoy you! Because the only way to turn it off and stop being annoyed is to specify the column structure yourself via `col_types` parameter. As a matter of fact, the print out you get is where, so you can take a look at it, adjust it, if necessary, and copy-paste to the `read_csv` call. By default, it suggested `double` values for `Block` and `Trial` but we know they are integers, so we can copy-paste the suggested structure, replace `col_double()` with `col_integer()` and read the table without a warning.

```
library(readr)
results <- read_csv("data/example.csv",
                    col_types = cols(Participant = col_character(),
                                     Block = col_integer(), # read_csv suggested col_d
                                     Trial = col_integer(), # read_csv suggested col_d
                                     Contrast = col_double(),
                                     Correct = col_logical()))
results
```

```
## # A tibble: 3 x 5
##   Participant Block Trial Contrast Correct
##   <chr>       <int> <int>    <dbl> <lgl>
## 1 A1              1     1     0.5  TRUE
## 2 A1              1     2     1    TRUE
## 3 A1              1     2     0.05 FALSE
```

You may feel that this a lot of extra work just to suppress an annoying but, ultimately, harmless warning. Your code will work with or without it, right?

Well, *hopefully* it will but you probably want to *know* that it will work not *hope* for it. Imagine that you accidentally overwrote your experimental data file with data from a different experiment (that happens more often than one would want). You still have `results.csv` file in your project folder and so the `read.csv()` will read it as is (it does not know what should be in that file) and your analysis code will fail in some mysterious ways at a much later point (because, remember, if you try to access a column/variable that does not exist in the table, you just get `NULL` rather than an error). You will eventually trace it back to the wrong data file but that will cost time and nerves. However, if you specify the column structure in `read_csv()` it will show warning, if the file does not match the description. It would warn about wrong column names (`TheBlock` in the example below) and about wrong type (it does not like `TRUE`/`FALSE` in a column it expected to find integers in).

```
library(readr)
results <- read_csv("data/example.csv",
                    col_types = cols(Participant = col_character(),
                                     TheBlock = col_integer(), # read_csv suggested col_double()
                                     Trial = col_integer(), # read_csv suggested col_double() bu
                                     Contrast = col_double(),
                                     Correct = col_integer())))
```

```
## Warning: The following named parsers don't match the column names: TheBlock
```

```
## Warning: 3 parsing failures.
## row     col   expected actual                   file
##   1 Correct an integer  TRUE  'data/example.csv'
##   2 Correct an integer  TRUE  'data/example.csv'
##   3 Correct an integer  FALSE 'data/example.csv'
```

Personally, I would prefer for `read_csv()` to fail in cases like these but having a nice red warning is already very helpful to quickly detect the problem with your data (and if your data is wrong, your whole analysis is meaningless). Thus, *always* use `read_` rather than `read.` functions and *always* specify the table structure. The lazy, and my preferred, way to do it, is to first read the file without specifying the structure and copy-paste-edit the warning column-specification message into the code.

Do exercise 7, you need face_rank.csv file for it. Download it and place it in the project folder. *Warning*, if you use Chrome or any Chromium-based browsers like MS Edge, Opera, etc. they might, for some odd reason, automatically *rename it* into *face_rank.xls* during the download. Just rename it back to *face_rank.csv*, because the file is not converted to an Excel, it is only the extension that gets changed (why? No idea, ask Google!).

3.11   Reading Excel files

There are several libraries that allow you to read Excel files directly. My personal preference is readxl package, which is part of the Tidyverse. Warning, it will be installed as part of the Tidyverse (i.e., when you typed `install.packages(tidyverse)`) but you still need to import it explicitly via `library(readxl)`. Because an Excel file has many sheets, by default the `read_excel()` function reads the *first* sheet but you can specify it via a `sheet` parameter using its index `read_excel("my_excel_file.xls", sheet=2)` or name `read_excel("my_excel_file.xls", sheet="addendum")`.

Do exercise 8, you need face_rank.xlsx file for it.

You can read about further options at the package's website but I would generally discourage you from using Excel for your work and, definitely, for your data analysis. Because, you see, Excel is very smart and it can figure out the *true* meaning and type of columns by itself. The fact that you might disagree is your problem. Excel knows what is best for you. The easiest way to screw a CSV file up is to open it in Excel and immediately save it. The file name will remain the same but Excel will "adjust" the content as it feels is better for you (you don't need to be consulted with). If you think I am exaggerating, read this article at The Verge on how Excel messed up thousands of human genome data tables by turning some values into dates (because why not?). So now the entire community is *renaming* some genes because it is easier to waste literally thousands of man-hours on that than to fix Excel. In short, friends don't let friends use Excel.

3.12   Reading files from other programs

World is a very diverse place, so you are likely to encounter a wide range of data files generated by Matlab, SPSS, SAS, etc. There are two ways to import the data. First, that I would recommend, use the original program (Matlab, SPSS, SAS, etc.) to export data as a CSV file. Every program can read and write CSV, so it a good common ground. Moreover, this is simple format with no embedded formulas (as in Excel), service structures, etc. Finally, if you store your data in CSV, you do not need a special program to work with it. In short, unless you have a good reason, store your data in CSV files.

However, sometimes you have a file but you do not have the program (Matlab, SPSS, SAS, etc.). This is the second way, when you can use various R libaries, starting with foreign, which can handle most typical cases, e.g., SPSS, SAS, State, or Minitab. The problem here, is that all the programs differ in the internal file formats and what exactly is included. For example, when importing from an SPSS sav-file via read.spss you will get a list with various components rather than a data.frame. You can force the function to convert everything to a

single table via `to.data.frame=TRUE` option but you may lose some information. Bottom line, you need to be extra careful when importing from other formats and the safest way is to ensure complete and full export of the data to a CSV from the original program.

3.13 Wrap up

We have started with vectors and now extended them to tables. Next time, we will look at how to visualize the data using The Grammar of Graphics approach.

Chapter 4

Grammar of Graphics

In previous seminar, you have learned about tables that are the main way of representing data in psychological research and in R. In the following seminars, you will learn how to manipulate the data in these tables: change it, aggregate or transform individual groups of data, use it for statistical analysis. But before that you need to understand how to store your data in the table in the optimal way. First, I will introduce the idea of *tidy data*, the concept that gave Tidyverse its name. Next, we will see how tidy data helps you visualize the relationships between variables. Don't forget to download the notebook.

4.1 Tidy data

The tidy data follows three rules:

* variables are in columns,
* observations are in rows,
* values are in cells.

This probably sound very straightforward to the point that you wonder "Can a table not by tidy?" As a matter of fact *a lot* of typical results of psychological experiments are not tidy. Imagine an experiment where participants rated a face on symmetry, attractiveness, and trustworthiness. Typically (at least in my experience), the data will stored as follows:

Participant	Face	Symmetry	Attractiveness	Trustworthiness
1	M1	6	4	3
1	M2	4	7	6
2	M1	5	2	1
2	M2	3	7	2

This is a very typical table optimized for *humans*. A single row contains all responses about a single face, so it is easy to visually compare responses of individual observers. Often, the table is even wider so that a single row holds all responses from a single observer (in my experience, a lot of online surveys produce data in this format).

Participant	M1.Symmetry	M1.Attractiveness	M1.Trustworthiness	M2.Symmetry	M2.Attract
1	6	4	3	4	7
1	5	2	1	3	7

So, what is wrong with it? Don't we have variables in columns, observations in rows, and values in cells? Not really. You can already see it when comparing the two tables above. The *face* identity is a variable, however, in the second table it is hidden in column names. Some columns are about face `M1`, other columns are about `M2`, etc. So, if you are interested in analyzing symmetry judgments across all faces and participants, you will need to select all columns that end with `.Symmetry` and figure out a way to extract the face identity from columns' names. Thus, face *is* a variable but is not a column in the second table.

Then, what about the first table, which has `Face` as a column, is it tidy? The short answer: Not really but that depends on your goals as well! In the experiment, we collected *responses* (these are numbers in cells) for different type of *judgments*. The latter are a variable but it is hidden in column names. Thus, a *tidy* table for this data would be

Participant	Face	Trustworthiness	Judgment	Response
1	M1	3	Symmetry	6
1	M1	3	Attractiveness	4
1	M2	6	Symmetry	4
1	M2	6	Attractiveness	7
2	M1	1	Symmetry	5
2	M1	1	Attractiveness	2
2	M2	2	Symmetry	3
2	M2	2	Attractiveness	7

This table is (very) tidy and it makes it easy to group data by every different combination of variables (e.g. per face and judgment, per participant and judgment), perform statistical analysis, etc. However, it may not always be the best way to represent the data. For example, if you would like to model `Trustworthiness` using `Symmetry` and `Attractiveness` as predictors, when the first table is more suitable. At the end, the table structure must fit your needs, not the other way around. Still, what you probably want is a *tidy* table because it is best suited for most things you will want to do with the data and because it makes it easy to transform the data to match your specific needs (e.g., going from the third table to the first one via pivoting).

Most data you will get from experiments will not be tidy. We will spent quite some time in learning how to tidy it up but first let us see how an already tidy data makes it easy to visualize relationships in it.

4.2 ggplot2

ggplot2 package is my main tool for data visualization in R. ggplot2 tends to make really good looking production-ready plots (this is not a given, a default-looking Matlab plot is, or used to be when I used Matlab, pretty ugly). Hadley Wickham was influenced by works of Edward Tufte when developing ggplot2. Although the aesthetic aspect goes beyond our seminar (although, if you are interested, I might make a Christmas-special on that), if you will need to visualize data in the future, I strongly recommend reading Tufte's books. In fact, it is such an informative and aesthetically pleasing experience that I would recommend reading them in any case.

More importantly, ggplot2 uses a grammar-based approach of describing a plot that makes it conceptually different from most other software such as Matlab, Matplotlib in Python, etc. A plot in *ggplot2* is described in three parts:

1. Aesthetics: Relationship between data and visual properties that define working space of the plot (which variables map on axes, color, size, fill, etc.).
2. Geometrical primitives that visualize your data (points, lines, error bars, etc.) that are *added* to the plot.
3. Other properties of the plot (scaling of axes, labels, annotations, etc.) that are *added* to the plot.

You always need the first one. But you do not need to specify the other two, even though a plot without geometry in it looks very empty. Let us start with a very simple artificial example table below.

| Condition	Intensity	Response
A	1	-1.1488362
B	1	-2.5639601
C	1	-3.8566105
A	2	0.6440975
B	2	0.6815639
C	2	-3.1919283
A	3	2.3663966
B	3	-0.7310656
C	3	-1.0401717
A	4	2.9989712
B	4	2.2163216
C	4	-0.0383730
A	5	4.3878768
B	5	2.0751273
C	5	0.0036814
A	6	3.9971447
B	6	3.1575005
C	6	1.3639655
A	7	4.8905494
B	7	3.2260466
C	7	1.4765708
A	8	7.3918176
B	8	5.3472123
C	8	3.3756461

We plot this data by 1) *defining aesthetics* (mapping `Intensity` on to x-axis, `Response` on y-axis, and `Condition` on color) and 2) *adding* lines to the plot (note the plus in `+ geom_line()`).

```
ggplot(data=simple_tidy_data, aes(x = Intensity, y = Response, color=Condition)) +
  geom_line()
```
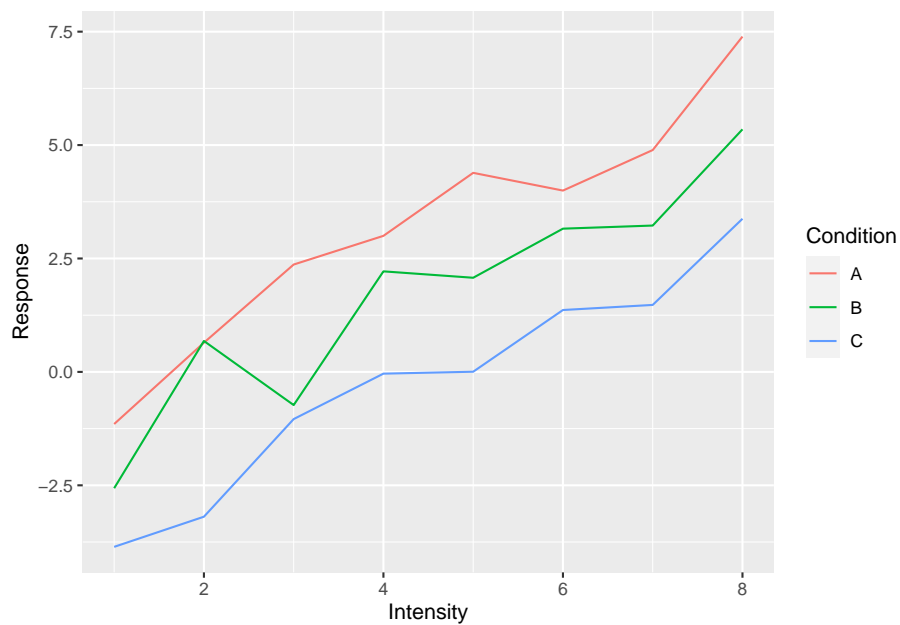
As I already wrote, technically, we only thing you need to define is aesthetics, so let us not add anything to the plot (drop the `+geom_line()`).

```
ggplot(data=simple_tidy_data, aes(x = Intensity, y = Response, color=Condition))
```
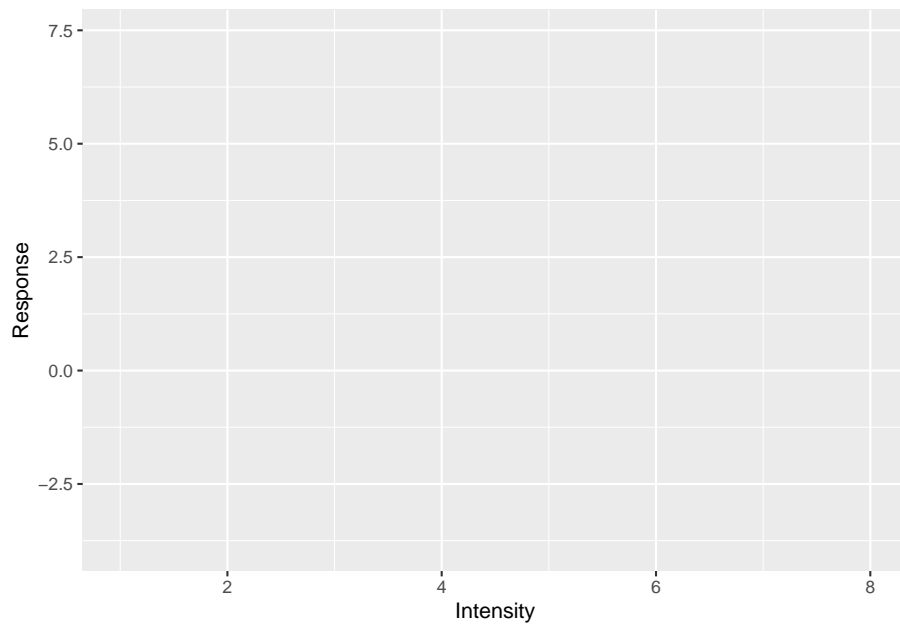
Told you it will look empty and yet you can see *ggplot2* in action. Notice that axes are labeled and their limits are set. You cannot see the legend (they are not plotted without corresponding geometry) but it is also ready. This is because our initial call specified the most important part: how the individual variables map on various properties even before we told *ggplot2* which visuals we will use to plot the data. When we specified that x-axis will represent the `Intensity`, ggplot2 figured out the range of values, so it knows *where* it is going to plot whatever we decide to plot. Points, lines, bar, error bars and what not will span only that range. Same goes for other properties such as color. We wanted *color* to represent the condition. Again, we may not know what exactly we will be plotting (points, lines?) or even how many different visuals we will be adding to the plot (just lines? points + lines? points + lines + linear fit?) but we do know that whatever visual we add, if it can have color, its color *must* represent condition for that data point. The beauty of *ggplot2* is that it analyses your data and figures out how many colors you need and is ready to apply them *consistently* to all visuals you will later add. It will ensure that all points, bars, lines, etc. will have consistent coordinates scaling, color-, size-, fill-mapping that are the same across the entire plot. This may sound trivial but typically (e.g., Matlab, Matplotlib), it is *your* job to make sure that all these properties match and that they represent the same value across all visual elements. And this is a pretty tedious job, particularly when you decide to change your mappings and have to redo all individual components by hand. In *ggplot2*, this dissociation between mapping and visuals means you can tinker with one of them at a time. E.g. keep the visuals but change grouping or see if effect of condition is easier to see via line type, size or shape of the point? Or you can keep the mapping and see whether adding another visual will make the plot easier to understand. Note that some mapping also *groups* your data, so when you use group-based visual information (e.g. a linear regression line) it will know what data belongs together and so will perform this computation per group.

Let us see how you can keep the relationship mapping but add more visuals. Let us add both lines and points.

```
ggplot(data=simple_tidy_data, aes(x = Intensity, y = Response, color=Condition)) +
  geom_line() +
  geom_point() # this is new!
```
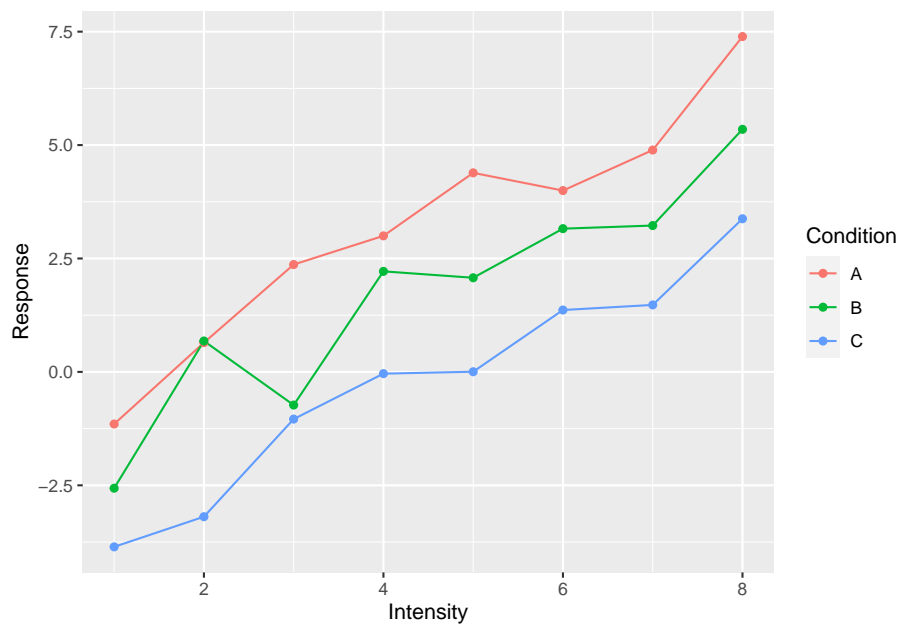
In the plot above, we *kept* the relationship between variables and properties but said "Oh, and throw in some points please". And ggplot2 knows how to add the points so that they appear at proper location and in proper color. But we want more!
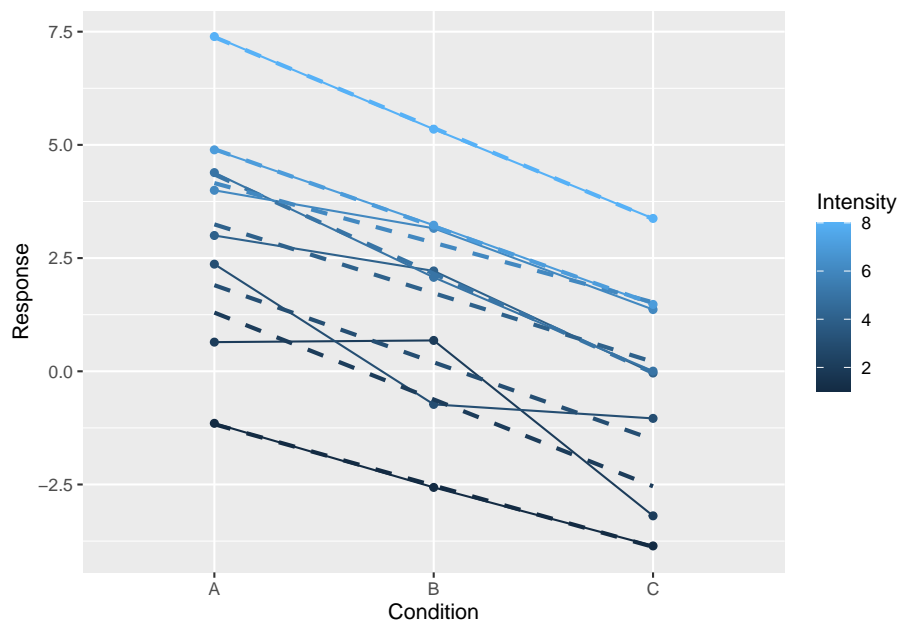
```
ggplot(data=simple_tidy_data, aes(x = Intensity, y = Response, color=Condition)) +
  geom_line() +
  geom_point() +
  # a linear regression over all dots in the group
  geom_smooth(method="lm", formula = y ~ x, se=FALSE, linetype="dashed")
```
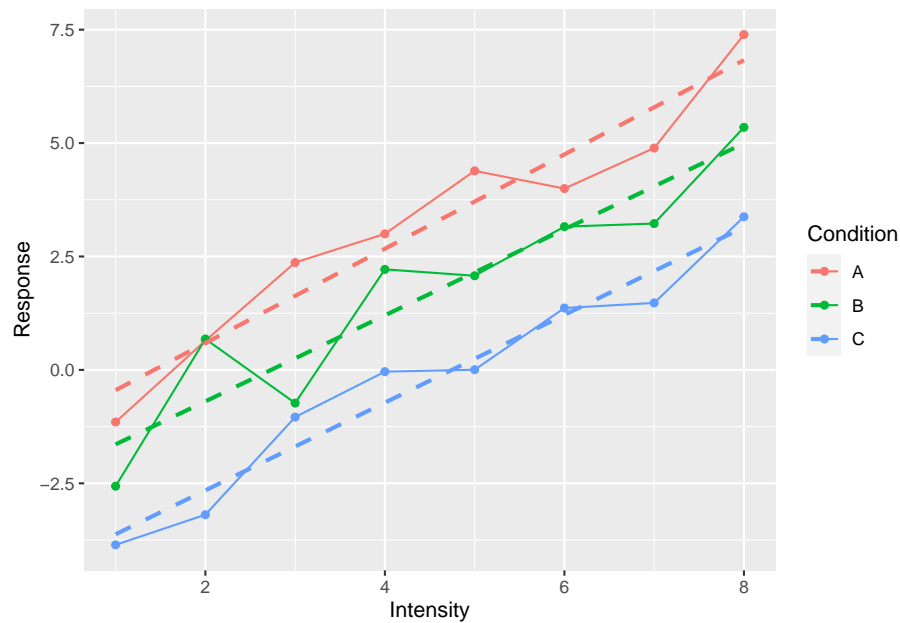
Now we added a linear regression line that helps us to better see the relationship between `Intensity` and `Response`. Again, we simply wished for another visual to be added (`method="lm"` means that we wanted to average data via linear regression with `formula = y ~ x` meaning that we regress y-axis on x-axis with no further covariates, `se=FALSE` means no standard error stripe, `linetype="dashed"` just makes it easier to distinguish from the solid data line).

Or, we can keep the *visuals* but see whether changing *mapping* would make it more informative (we need to specify `group=Intensity` as continuous data is not grouped automatically).

```
ggplot(data=simple_tidy_data, aes(x = Condition, y = Response, color=Intensity, group=
  geom_line() +
  geom_point() +
  geom_smooth(method="lm", se=FALSE,  formula = y ~ x, linetype="dashed")
```

Or, we can check whether splitting into several plots helps.

```
ggplot(data=simple_tidy_data, aes(x = Intensity, y = Response, color=Condition)) +
  geom_line() +
  geom_point() +
  geom_smooth(method="lm", formula = y ~ x, se=FALSE, linetype="dashed") +
  facet_grid(. ~ Condition) # makes a separate subplot for each group
```
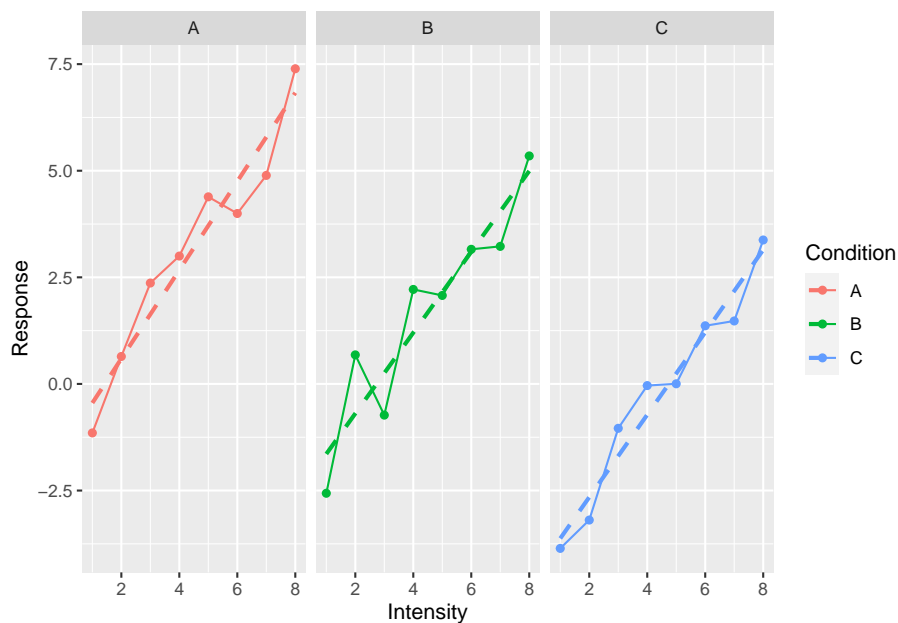
Again, note that all three plots live on the same scale for x- and y-axis, making
them easy to compare (you fully appreciate this magic if you ever struggled with
ensuring optimal and consistent scaling by hand in Matlab). I went through so
many examples to stress how ggplot allows you to think about the aesthetics
of variable mapping *independently* of the actual visual representation (and vice
versa). So now lets us explore *ggplot2* by doing exercises.

I recommend using ggplot2 reference page and cheatsheet when you are doing
the exercises.

4.3   Auto efficiency: continuous x-axis

We start by visualizing how car efficiency, measured as miles-per-gallon, is af-
fected by various factors such as production year, size of the engine, type of
transmission, etc. The data is in table mpg, which is part of the *ggplot2* pack-
age. Thus, you need to first import the library and then load the table via
data() function. Take a look at the table description to familiarize yourself with
the variables.

First, let us look at the relationship between car efficiency in the city cycle (`cty`),
engine displacement (`displ`), and drive train type (`drv`) using color points.
Reminder, the call should look as

`[ggplot2](https://ggplot2.tidyverse.org/reference/ggplot.html)(data_table_name, [aes](`

```
geom_primitive1() +
geom_primitive2() +
...
```

Think about which variables are mapped on each axes and which is best depicted as color.

Do exercise 1.

Do you see any clear dependence? Let us try to making it more evident by adding geom_smooth geometric primitive.

Do exercise 2.

Both engine size (displacement) and drive train have a clear effect on car efficiency. Let us visualize the number of cylinders (`cyl`) as well. Including it by mapping it on the *size* of geometry.

Do exercise 3.

Currently, we are mixing together cars produced at different times. Let us visually separate them by turning each year into a subplot via facet_wrap function.

Do exercise 4.

The dependence you plotted does not look linear but instead is saturating at certain low level of efficiency. This sort of dependencies could be easier to see on a logarithmic scale. See functions for different scales and use logarithmic scaling for y-axis.

Do exercise 5.

Note that by now we managed to include *five* variables into our plots. We can continue this by including transmission or fuel type but that would be pushing it, as too many variables can make a plot confusing and cryptic. Instead, let us make it prettier by using more meaningful axes labels (`xlab()`, `ylab()` functions) and adding a plot title (labs).

Do exercise 6.

4.4   Auto efficiency: discrete x-axis

The previous section use a continuous engine displacement variable for x-axis (at least that is my assumption on how you mapped the variables). Frequently, you need to plot data for discrete groups: experimental groups, conditions, treatments, etc. Let us practice on the same mpg data set but visualize relationship between the drive train (`drv`) and highway cycle efficiency (`hwy`). Start by using point as visuals.

Do exercise 7.

One problem with the plot is that all points are plotted at the same x-axis location. This means that if two points share the location, they overlap and appear as just one dot. This makes it hard to understand the density: one point can mean one point, or two, or a hundred. A better way to plot such data is by using box or]violin](https://ggplot2.tidyverse.org/reference/geom_violin.html) plots. Experiment by using them instead of points.

Do exercise 8.

Again, let's up the ante and split plots via both number of cylinders and year of manufacturing. Use facet_grid function to generate grid of plots.

Do exercise 9.

Let us again improve our presentation by using better axes labels and figure title.

Do exercise 10.

4.5   Mammals sleep: single variable

Now lets us work on plotting a distribution for a single variable using mammals sleep dataset. For this, you need to map `sleep_total` variable on x-axis and plot a histogram. Explore the available options, in particular `bins` that determines the bins number and, therefore, their size. Note that there is no "correct" number of bins to use. *ggplot2* defaults to 30 but a small sample would be probably better served with fewer bins and, vice versa, with a large data set you can afford hundred of bins.

Do exercise 11.

Using a histogram gives you exact counts per each bin. However, the appearance may change quite dramatically if you would use fewer or more bins. An alternative way to represent the same information is via smoothed density estimates. They use a sliding window and compute an estimate at each point but also include points *around* it and weight them according to a kernel (e.g. a Gaussian one). This makes the plot look smoother and will mask sudden jumps in density (counts) as you, effectively, average over many bins. Whether this approach is better for visualizing data depends on the sample you have and message you are trying to get across. It is always worth checking both (just like it is worth checking different number of bins in histogram) to see which way is the best for your specific case.

Do exercise 12.

Let us return to using histograms and plot a distribution per `vore` variable (it is `carnivore`, `omnivore`, `herbivore`, or `NA`). You can map it on the `fill` color of the histogram, so that each *vore* kind will be binned separately.

Do exercise 13.

The plot may look confusing because by default *ggplot2* colors values for each group differently but stacks all of them together to produce the total histogram counts. One way to disentangle the individual histograms is via facet_grid function. Use it to plot `vore` distribution in separate rows.

Do exercise 14.

That did the trick but there is an alternative way to plot individual distributions on the same plot by setting `position` argument of geom_histogram to `"identity"` (it is `"stack"` by default).

Do exercise 15.

Hmm, shouldn't we have more carnivores, what is going on? Opacity is the answer. A bar "in front" occludes any bars that are "behind" it. Go back to the exercise and fix that by specifying `alpha` argument that controls transparency. It is `1` (completely opaque) by default and can go down to `0` (fully transparent as in "invisible"), so see which intermediate value works the best.

4.6 Mapping for all visuals versus just one visual

In the previous exercise, you assigned a constant value to `alpha` (transparency) argument. You could do this in *two* places, inside of either `ggplot()` or `geom_histogram()` call. In the former case, you would have set `alpha` level for *all* geometric primitives on the plot, whereas in the latter you do it only for the histogram. To better see the difference, reuse your code for plotting city cycle efficiency versus engine size (should be exercise #6) and set `alpha` either for all visuals (in `ggplot2`) or in some visuals (e.g. only for points) to see the difference.

Do exercise 16.

4.7 Mapping on variables versus constants

In the previous exercise, you assigned a constant value to `alpha` (transparency) argument. However, transparency is just a property just like `x`, `color`, or `size`. Thus, there are *two* ways you can use them:

- *inside* `aes(x=column)`, where `column` is column in the table you supplied via `data=`
- *outside* of `aes` by stating `x=value`, where value is some constant value or a variable *that is not in the table*.

Test this but setting the `size` in the previous plot to a constant outside of aesthetics or to a variable inside of it.

Do exercise 17.

4.8   Themes

Finally, if you are not a fan of the way the plots look, you can quickly modify this by using some other theme. You can define it yourself (there are lots of options you can specify for your theme) or can use one of the ready-mades. Explore the latter option, find the one you like the best.

Do exercise 18.

4.9   You ain't seen nothing yet

What you explored is just a tip of the iceberg. There are many more geometric primitive, annotations, scales, themes, etc. It will take an entire separate seminar to do *ggplot2* justice. However, the basics will get you started and you can always consult reference, books (see below), or me once you need more.

4.10   Further reading

If plotting data is part of your daily routine, I recommend reading ggplot2 book. It gives you an in-depth view of the package and goes through many possibilities that it offers. You may need all of them but I find useful to know that they exists (who knows, I might need them one day). Another book worth reading is Data Visualization: A Practical Introduction. by Kieran Healy.

4.11   Extending ggplot2

There are 80 (as of 19.11.2020) extensions that you find at ggplot2 website. They add more ways to plot your data, more themes, animated plots, etc. If you feel that *ggplot2* does not have the geometric primitives you need, take a look at the gallery and, most likely, you will find something that fits your bill.

One package that is *not* in the gallery is patchwork. It was created "to make it ridiculously simple to combine separate ggplots into the same graphic". It is a bold promise but authors do make good on it. It is probably the easiest way to combine multiple plots but you can also consider cowplot and gridExtra packages.

4.12 ggplot2 cannot do everything

There are many different plotting routines and packages for R but I would recommend to use *ggplot2* as your main tool. However, that does not mean that it must be your only tool, after all, CRAN is brimming with packages. In particular, *ggplot2* is built for plotting data from a single tidy table, meaning it is less optimal for plotting data in other cases. E.g., you can use it to combine information from several tables in one plot but things become less automatic and consistent. Similarly, you can plot data which is stored in non-tidy tables or even in individual vectors but that makes it less intuitive and more convoluted. No package can do everything and *ggplot2* is no exception.