

# Data analysis using R for Psychology and Social Science

Alexander (Sasha) Pastukhov

2023-10-09



# Contents

<b>Introduction</b>	<b>7</b>
Goal of the book . . . . .	8
Why R? . . . . .	9
Why Tidyverse . . . . .	9
About the seminar itself . . . . .	10
Thinking like a computer . . . . .	10
About the material . . . . .	11
<b>Software</b>	<b>13</b>
Installing R . . . . .	13
Installing R-Studio . . . . .	13
Installing RTools . . . . .	13
Installing packages . . . . .	13
0.1 Minimal setup of R-Studio . . . . .	15
Minimal set of packages . . . . .	17
Keeping R and packages up-to-date . . . . .	17
<b>1 Reproducible Research: Projects and Markdown Notebooks</b>	<b>19</b>
1.1 Projects . . . . .	19
1.2 Quattro and RMarkdown notebooks . . . . .	21
1.3 Exercise . . . . .	25
<b>2 Vectors! Vectors everywhere!</b>	<b>27</b>
2.1 Variables as boxes . . . . .	27
2.2 Assignment statement in detail . . . . .	28
2.3 Vectors and singlar values (scalars, which are also vectors) . . .	29
2.4 Vector indexes (subsetting) . . . . .	31
2.5 Names as an Index . . . . .	33
2.6 Slicing . . . . .	35
2.7 Colon Operator and Sequence Generation . . . . .	36
2.8 Working with two vectors of <i>equal</i> length . . . . .	37
2.9 Working with two vectors of <i>different</i> length . . . . .	38
2.10 Applying functions to a vector . . . . .	39

2.11	Wrap up . . . . .	40
<b>3</b>	<b>Tables and Tibbles (and Tribbles)</b>	<b>41</b>
3.1	Primary data types . . . . .	41
3.2	All vector values must be of the same type . . . . .	43
3.3	Lists . . . . .	43
3.4	Subsetting, a fuller version . . . . .	43
3.5	Yet another subsetting via \$ . . . . .	47
3.6	Tables, a.k.a. data frames . . . . .	48
3.7	Subsetting tables . . . . .	50
3.8	Using libraries . . . . .	52
3.9	Tibble, a better data.frame . . . . .	55
3.10	Tribble, table from text . . . . .	55
3.11	Reading example tables . . . . .	56
3.12	Reading csv files . . . . .	56
3.13	Reading Excel files . . . . .	60
3.14	Reading files from other programs . . . . .	61
3.15	Writing and reading a single object . . . . .	61
3.16	Wrap up . . . . .	62
<b>4</b>	<b>Functions! Functions everywhere!</b>	<b>63</b>
4.1	Functions . . . . .	63
4.2	Writing your own function . . . . .	65
4.3	Scopes: Global versus Local variables . . . . .	66
4.4	Function with two parameters . . . . .	68
4.5	Table as a parameter . . . . .	69
4.6	Named versus positional arguments . . . . .	69
4.7	Default values . . . . .	70
4.8	Nested calls . . . . .	71
4.9	Piping . . . . .	72
4.10	Function is just a code stored in a variable . . . . .	74
4.11	Functions! Functions everywhere! . . . . .	74
4.12	Using (or not using) explicit return statement . . . . .	75
<b>5</b>	<b>ggplot2: Grammar of Graphics</b>	<b>77</b>
5.1	Tidy data . . . . .	77
5.2	ggplot2 . . . . .	79
5.3	Auto efficiency: continuous x-axis . . . . .	86
5.4	Auto efficiency: discrete x-axis . . . . .	87
5.5	Mammals sleep: single variable . . . . .	88
5.6	Mapping for all visuals versus just one visual . . . . .	89
5.7	Mapping on variables versus constants . . . . .	89
5.8	Themes . . . . .	90
5.9	You ain't seen nothing yet . . . . .	90
5.10	Further reading . . . . .	90
5.11	Extending ggplot2 . . . . .	90

5.12	ggplot2 cannot do everything . . . . .	90
<b>6</b>	<b>Tidyverse: dplyr</b>	<b>93</b>
6.1	Tidyverse philosophy . . . . .	93
6.2	<code>select()</code> columns by name . . . . .	95
6.3	Conditions . . . . .	98
6.4	Logical indexing . . . . .	100
6.5	<code>filter()</code> rows by values . . . . .	100
6.6	<code>arrange()</code> rows in a particular order . . . . .	101
6.7	<code>mutate()</code> columns . . . . .	102
6.8	<code>summarize()</code> table . . . . .	104
6.9	Work on individual groups of rows . . . . .	104
6.10	Putting it all together . . . . .	107
6.11	Should I use Tidyverse? . . . . .	107
<b>7</b>	<b>Working with Factors</b>	<b>109</b>
7.1	How to write code . . . . .	109
7.2	Implementing a typical analysis . . . . .	110
7.3	Factors . . . . .	112
7.4	Forcats . . . . .	114
7.5	Plotting group averages . . . . .	116
7.6	Plotting our confidence in group averages via quantiles . . . . .	118
7.7	Dealing with Likert scales . . . . .	121
<b>8</b>	<b>Tidying your data: joins and pivots</b>	<b>125</b>
8.1	Joining tables . . . . .	125
8.2	Pivoting . . . . .	132
8.3	Pivot longer . . . . .	133
8.4	Practice using Likert scale data . . . . .	135
8.5	Pivot wider . . . . .	139
<b>9</b>	<b>Controlling computation flow</b>	<b>145</b>
9.1	<code>rep()</code> . . . . .	145
9.2	Repeating combinations . . . . .	146
9.3	For loop . . . . .	147
9.4	Conditional statement . . . . .	151
9.5	<code>ifelse()</code> . . . . .	153
9.6	<code>case_when</code> . . . . .	154
9.7	Breaking out of the loop . . . . .	155
9.8	Using for loop to load and join multiple data files . . . . .	156
9.9	Apply . . . . .	156
9.10	Purrr . . . . .	158
<b>10</b>	<b>Missing data</b>	<b>161</b>
10.1	Making missing data explicit (completing data) . . . . .	161
10.2	Dropping / omitting NAs . . . . .	163

<b>11 Working with strings</b>	<b>169</b>
11.1 Warming up . . . . .	169
11.2 Formatting strings via <code>glue()</code> . . . . .	169
11.3 Formatting strings via <code>paste()</code> . . . . .	171
11.4 Formatting strings via <code>sprintf()</code> . . . . .	172
11.5 Extracting information from a string . . . . .	173
11.6 Splitting strings via <code>separate()</code> . . . . .	173
11.7 Extracting a substring when you know its location . . . . .	177
11.8 Detecting a substring using regular expressions . . . . .	178
11.9 Extracting substring defined by a regular expression . . . . .	179
11.10 Replacing substring defined by a regular expression . . . . .	180
<b>12 Sampling and simulations</b>	<b>181</b>
12.1 Estimating mean of a normal distribution via resampling . . . . .	181
12.2 Repeating computation via for loop . . . . .	185
12.3 Repeating computation via <code>purrr</code> . . . . .	185
12.4 Bootstrapping via <code>boot</code> library . . . . .	186
12.5 Confidence about proportion of responses for Likert scale . . . . .	187
<b>13 (Generalized) Linear regression and Resampling</b>	<b>191</b>
13.1 Linear regression: simulating data . . . . .	192
13.2 Linear regression: statistical model . . . . .	192
13.3 Linear regression: bootstrapping predictions . . . . .	196
13.4 Logistic regression: simulating data . . . . .	197
13.5 Logistic regression: fitting data . . . . .	198
13.6 Logistic regression: bootstrapping uncertainty . . . . .	200

# Introduction

This book will (try to) teach you how to perform typical data analysis tasks: reading data, transforming and cleaning it up so you can visualize it and perform a statistical analysis. It covers most topics that you need to get you started but it cannot cover them all. One advantage of R is a sheer size of its ecosystem with new incredible libraries appearing very much on a daily basis. Covering them all is beyond the scope of any book, so instead I will concentrate on (trying to) building a solid understanding of things that you need to extend your R knowledge. Because of that some early chapters (e.g., on vectors, tables, or functions) might feel boring and too technical making you wonder why didn't we start with some exciting and useful analysis, working our way down to finer details. I have tried that<sup>1</sup> but, unfortunately, philosophy of R is about having many almost identical ways of achieving the same end. If you do not learn these finer details, you waste time wondering why seemingly the same code works in one case but fails in mysterious ways in the other one<sup>2</sup>. Therefore, please bear with me and struggle through vectors (which are everywhere!), oddities and inconsistencies of subsetting, and learning how to write a function before you even started to use them properly. I can only promise that, from my personal experience, this is definitely worth the effort.

An important note: this book will *not* teach you statistics or machine learning beyond several examples at the very end. The reason for this is that it teaches data preparation and both statistics and machine learning are 90% about data preparation. This is most obvious in machine learning where data acquisition, cleaning, feature engineering, etc. to make it suitable for analysis take most of your time. The actual machine learning part boils down to trying various<sup>3</sup> standard machine learning methods on it and picking one that gives best out-of-sample performance. That last part is so automated by now that it requires little knowledge beyond details of a specific package. Knowing how methods work is obviously beneficial but, and I hate to write this, not *that* critical for machine learning (not so for statistics or deep learning!). Same is true for statistical methods, although where time is split between preparing data for statistical

---

<sup>1</sup>As a matter of fact, this was my approach when learning R.

<sup>2</sup>Talking from a personal experience here.

<sup>3</sup>Who am I kidding, all!

analysis and interpreting and comparing models. As with machine learning, running statistical models itself is easy and automatic. If you know (enough of) statistics, you will have little trouble understanding how to work with these packages and functions. If you do not, no amount of reading of manuals will make it clearer.

## Goal of the book

The goal is for you to be able to program a typical analysis pipeline

- Loading data from csv/excel/online files or other data manipulation programs like SPSS
- Preprocessing data by in the entire table or per group (certain combination of columns)
  - manipulating text information
  - converting variables to factors
  - transforming numeric data
  - filtering data
  - handling missing data
- Simplifying data structure by
  - selecting relevant columns
  - reshaping table between long (tidy) and wide (human-readable) formats
- Summarizing data
- Plotting data using ggplot2 and its extensions
- Performing basic statistical analysis, reporting and visualizing model predictions alongside the data.
- Putting it all together in a single notebook that can be compiled (“knitted”) into a final document: report, presentation, seminar work, thesis, etc.

The programming concepts that you learn will include

- Variables and data types
- Vectors, lists, and tables that are foundation of R data
- Writing functions
- Conditional statements
- Loops
- Functional programming, i.e., applying the same function to multiple data points at the same time

Please note that exercises are distributed through each chapter embedded in the text and you should do them at that time point. They designed to clarify concepts and apply the knowledge that were presented before, so doing them immediately would be most helpful



## 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 analyses that we will perform using Python in combination with Jupyter notebooks (for reproducible analysis), Pandas (for Excel-style table), and statmodels (for statistical analysis). R is hardly perfect. For example, its subsetting system is confusing and appears to follow “convenience over safety” approach that does not sit particularly well with me. However, R in combination with piping (an easy way to perform a series of computations on the same table) and Tidyverse family of packages makes it incredibly easy to write simple, powerful and expressive code, which 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 (or base R). In addition, R is loved by mathematicians and statisticians, so it tends to have implementations for all cutting edge statistical methods (but Python is your go to for machine and deep learning).

## Why Tidyverse

The material is heavily skewed towards using Tidyverse family of packages. It looks different enough from base R to the point that one might call it a “dialect” of R<sup>4</sup>. Learning Tidyverse means that you have twice as many things to learn: I will always introduce both base R and Tidyverse version. Tidyverse is the main reason I use R (rather than Python or Julia) as it makes data analysis a breeze and makes your life so much easier. This is why I want you to learn its ways. At the same time, there is plenty of useful code that uses base R, so you need to know and understand it as well.

As a matter of fact, R is so rich and flexible that there many dialects and, therefore, plenty of opinion differences<sup>5</sup>. For example, `data.table` package re-implements the same functionality as base R and Tidyverse in a very compact way. I does not fit my style but it might be something that feels natural to you, so I encourage you to take a look. There are also other packages to handle things like laying out your figures or working with summary tables that might suit you better. Point is, these material barely scratches the surfaces in terms of tools and approaches that you can use. View it as a starting point for your exploration not the complete map.

Another thing to keep in mind is that Tidyverse is under very active development. This means that parts of this material could be outdated by the time you read it. E.g., `dplyr` `do()` verb was superseded by a `group_modify()` function, a warning generated by `readr` package was adapted for humans but now requires an extra step to be used for column specification, we are now on the *third* set

---

<sup>4</sup>R is extremely flexible, making it possible to redefine its own syntax.

<sup>5</sup>Just ask about “base R vs. Tidyverse” on Twitter and see the thread set itself on fire!

of pivoting functions, etc. None of the changes are breaking and deprecation process is deliberately slow (e.g., `do()` still works), so even when outdated the code in the book should still work for quite some time. However, you should keep in mind that things *might* have changed, so it is a good idea to check an official manual from time to time.

## About the seminar itself

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. Each chapter covers a single seminar, introducing necessary ideas and is accompanied by a notebook with exercises, which you need to complete and submit. To pass the seminar, you will need to complete *all* assignments. You do not need to complete or provide correct solutions for *all* the exercises to pass the course and information on how the points for exercises will be converted to an actual grade (if you need one) or “pass” will be available during the seminar.

The material assumes no foreknowledge of R or programming in general from a 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.) and 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, or you simply got stuck. 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).

## Thinking like a computer

In some exercises you will not be writing 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 a very simple, clear, and unambiguous way<sup>6</sup>. 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

---

<sup>6</sup>Not true anymore in general with large-language models but still true for R programming.

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 the code 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 **never** use the code you do not understand (do not blindly copy-paste)!

## About the material

The material is **free to use** and is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives V4.0 International License.



# Software

## Installing R

Go to [r-project.org](http://r-project.org) and download a current stable version of R for your platform. Run the installer, accepting all defaults.

## Installing R-Studio

Go to [posit.co](http://posit.co) and download *RStudio Desktop* 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 via *Tools / Global Options...* menu.

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

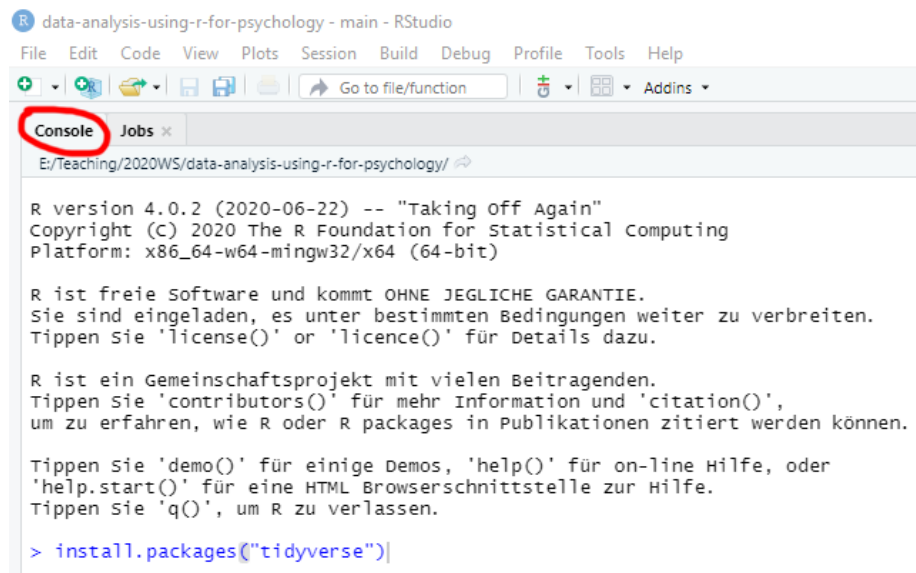
If you are using Windows, you might need Rtools for building and running some packages. You do not need to install it at the beginning, but if you will need it later, just following the link above, download the latest *Rtools* version, run the installer using the *defaults* and follow the instructions on that page to put Rtools on the PATH!

## 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 for you. We will use CRAN as a sole source of packages. However, there are alternatives, such as Bioconductor that might have a package that is missing on CRAN. The Bioconductor relies on its own package manager, so you will need to consult the latest manual on their website.

To install a 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**.



The screenshot shows the RStudio interface with the 'Console' tab selected. The console output displays the R version (4.0.2), copyright information, and a list of helpful commands. The command `> install.packages("tidyverse")` is entered at the prompt. The 'Console' tab label is circled in red.

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

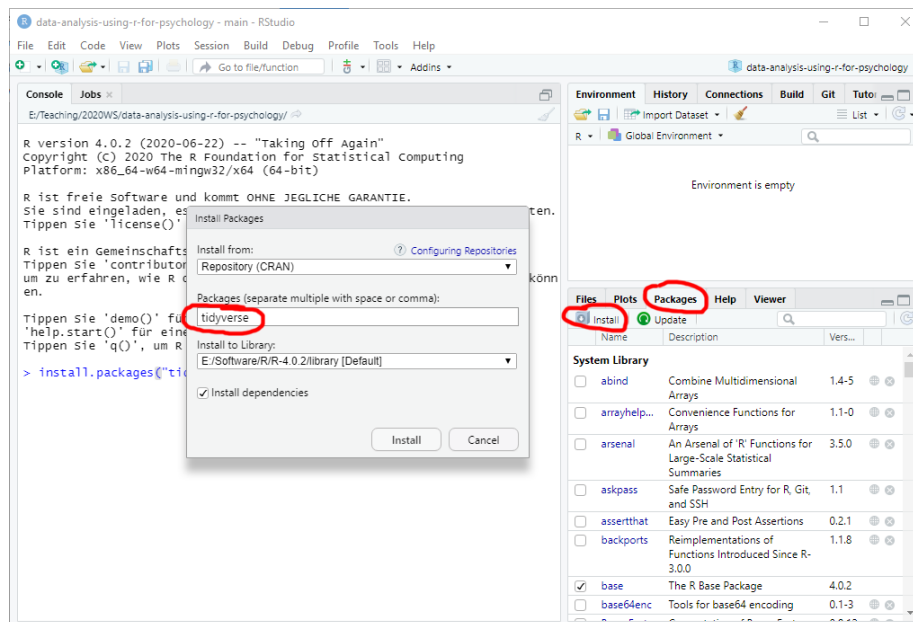
R ist freie Software und kommt OHNE JEGliche GARANTIE.
Sie sind eingeladen, es unter bestimmten Bedingungen weiter zu verbreiten.
Tippen Sie 'license()' or 'licence()' für Details dazu.

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

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

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

Alternatively, go to *Packages* tab, click on *Install* button, enter a package name in the window (it has autocomplete to help you), and press *Install*.

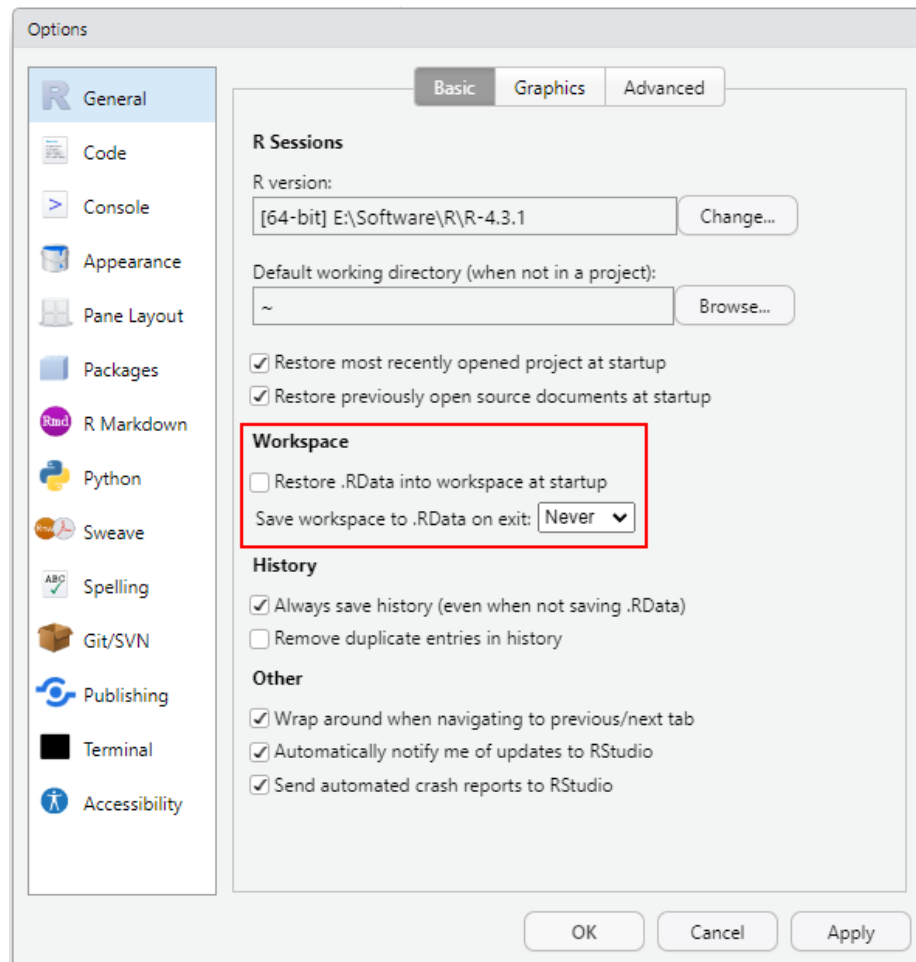


Sometimes, R will ask whether you want to install packages from source. In this case, it will grab the source code and compile the package, which takes time and requires RTools. In most cases, you can say “No” to install a pre-build binary version. The binary version will be slightly outdated but the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

On other occasions, R-Studio will suggest restarting R Session as packages that need to be updated in use. You can do that but, in my experience, this could become a repetitive experience if one of the packages is used R Studio itself (so it starts it in a new session, realizes that it is in use, suggests to restart the session, etc.) My solution is to close *all* R Studio windows and use R directly. For Windows, you can find it the Start Menu, just make sure that you are using the correct version. Then, I use `install.packages()` to install and update everything I need.

## 0.1 Minimal setup of R-Studio

A big advantage of R-Studio is that it comes as a turn-key solution with reasonable default settings. However, there is one convenience feature that I strongly recommend turning off: automatic storing and saving of the workspace. Go to *Tool / Global Options...* and in *General* setting tab unselect “Restore .RData into workspace at startup” and set “Save workspace to .RData on exit:” to *Never*.



The feature that you just turned off sounds great on paper: When it is enabled the entire workspace (state of the program) is saved and automatically restored when you start RStudio again. Very convenient, as you start off at the same state, with same loaded libraries and variables/tables/functions as you have left. However, in my experience, the price you pay is that you may no longer remember how did you get to that state, as it is common to try out things while writing and editing the analysis. This leaves libraries that are loaded (but not explicitly in the code), variables created or modified by hand (with no trace of this in the program), etc. So while things will continue to work on your computer, same notebook or code will fail on a different one. And debugging problems like these is one of the toughest cases taking me ages to solve every time I make a mistake like that. The price you pay is that all the computation must be repeated but the advantage of it is that it gives you another opportunity to check that analysis works as it should and your previous results were not a fluke.



## Minimal set of packages

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. On Windows, 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. It is easy to use even in R itself, as it creates an extra menu to make interface simpler. 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

## Reproducible Research: Projects and Markdown Notebooks

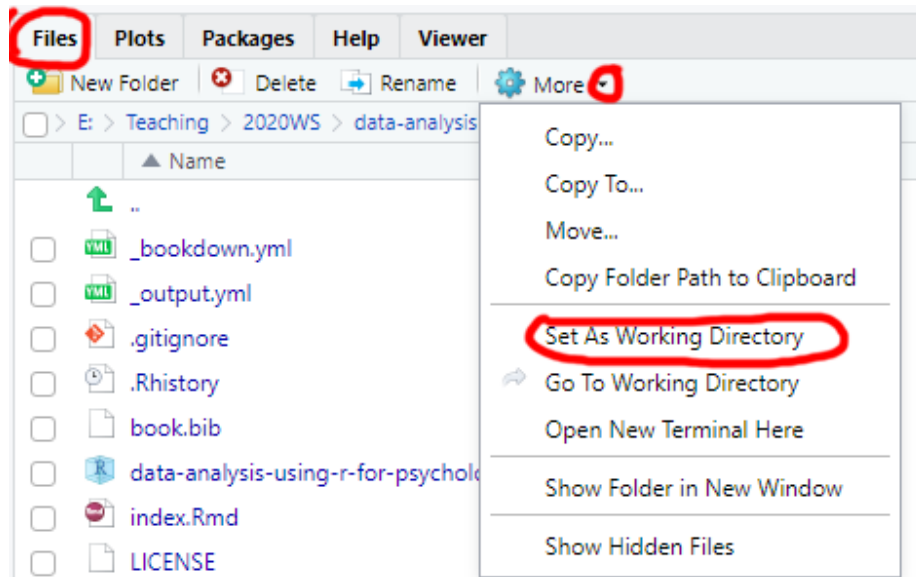
Our aim is to create reproducible research and analysis. It is a crucial component of the open science movement but is even more important for your own research or study projects. You want to create a self-contained well-documented easy-to-understand reproducible analysis. A complete self-sufficient code 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 cases like these). It also makes it easier to communicate your results to other researchers or fellow students.

You should also *always* consider posting your results and analysis online at public repositories such as OSF, Science Data Bank, or GitHub. This not only help others but forces you into making such data + analysis archive more thoroughly. That, in turns, makes it easier for future-you to return to the data and the analysis. Using a GitHub (private) repository is a good idea even if you are not planning to collaborate with others as it gives you a version-controlled code in a cloud and makes synchronizing it between different machines easy.

### 1.1 Projects

One of the most annoying features of R is that it looks for 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 (that trick won't work for an opened file).



In short, it may *look* 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 can clearly see with your own eyes. To avoid this, you should organize any program, project or seminar as an *R Project*, which assumes that all 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 (if you know git, this is very convenient). You can do it either way. This will be a 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 `the.Rproj` file in that folder.

## 1.2 Quattro and RMarkdown notebooks

There are two very similar notebook formats in R: an older R Markdown and a new Quattro. On the one hand, the latter is the future, so it would make sense to use Quattro notebooks. On the other hand, there is little practical difference for you, as from a R-only end-user point of view, they differ mostly in how they specify chunk options (more on that later).

Both RMarkdown and Quattro rely on a markdown language to combine formatted text, figures, references (via bibtex) and cross-references with code<sup>1</sup>. 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, comments, discussion, conclusions, etc.) with the actual code that implements what you described. And, you can be sure that the figures and numbers are the latest correct version.

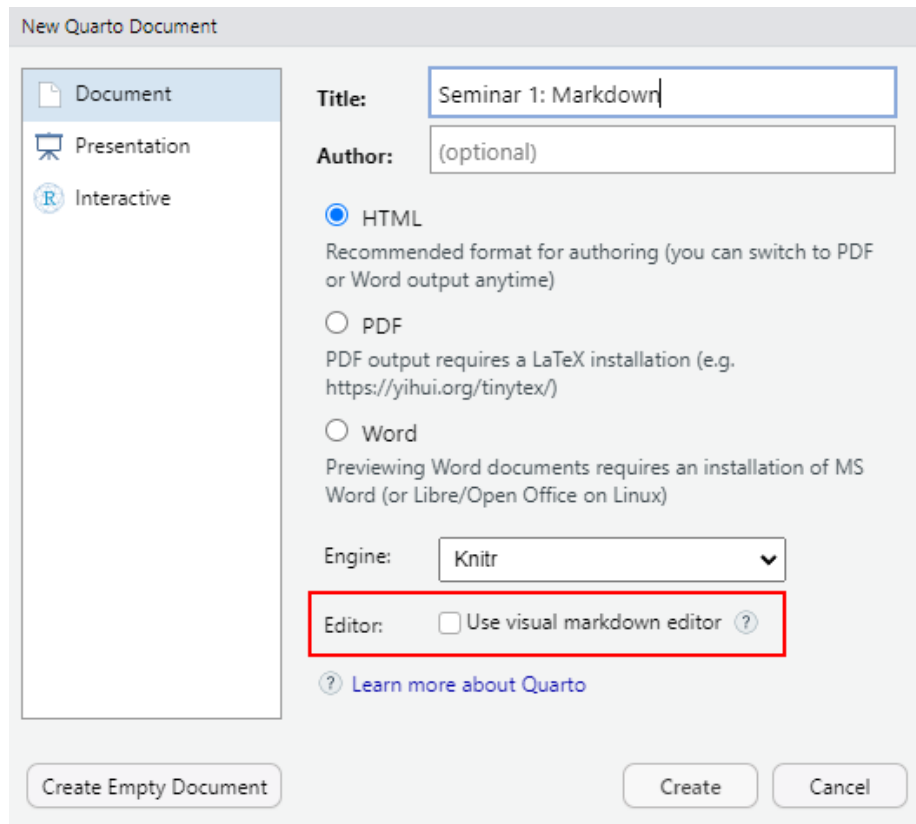
Notebooks can be knitted into a variety of formats including HTML, PDF, Word document, EPUB book, etc. Thus, instead of creating plots and tables to save them into separate files so you can copy-paste them into your Word file (and then redoing this, if something changed, and trying to find the correct 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 exercises will involve using 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 designed to make formatting simple and unobtrusive, so the plain document is easier 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 / Quattro Document...* menu. Use **Seminar 1: Markdown** for its title and HTML as default output format. The new RStudio makes you life easier by giving you a WYSIWYG visual editor option, which you can use later, but for this seminar disable it and learn how to use markdown directly

---

<sup>1</sup>This material was prepared using RMarkdown, knitr, and bookdown



Save the file (press **Ctrl + S** or use *File/Save* menu) calling it `seminar-01` (R Studio will add `.qmd` extension automatically). The file you created is not empty, as R Studio is kind enough to provide an example for you (interestingly, you get a different example if you create RMarkdown file instead). 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 a *Viewer* tab).

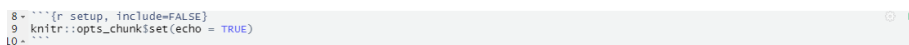
### 1.2.1 RMarkdown template



Let us go through the default RMarkdown notebook that R Studio created for us (it has more content than a default Quattro document) that you can see above or can create yourself via *File / New File / R Markdown...* (not *R notebook!*).



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 this at [yihui.org/knitr](http://yihui.org/knitr).



The next section is the “setup code chunk” that specifies default options for how the code chunks are treated by default (whether they are executed, whether the output, warnings, or messages are shown, 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 this 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).

```

12 ## R Markdown
13
14 This is an R Markdown document. Markdown is a simple formatting syntax for authoring HTML, PDF, and MS word documents. For more details
15 on using R Markdown see <http://rmarkdown.rstudio.com>.
16
17 When you click the "Knit" button a document will be generated that includes both content as well as the output of any embedded R code
18 chunks within the document. You can embed an R code chunk like this:

```

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

```

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 second set of ticks `````. Here `r` specifies that the code inside is written in R language but you can use other languages such as Python (via reticulate package), Stan, or SQL. The `name of the chunk` is optional but I would recommend to specify 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 npressure, echo=FALSE}
27 plot(npressure)
28
29 # Chunk 1: setup
30 # R Markdown
31 # = FALSE parameter was added to the code chunk to prevent printing of the R code that generated the plot.
32 # Including Plots
33 # Chunk 2: cars
34 # Chunk 3: pressure

```

There are 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)

```

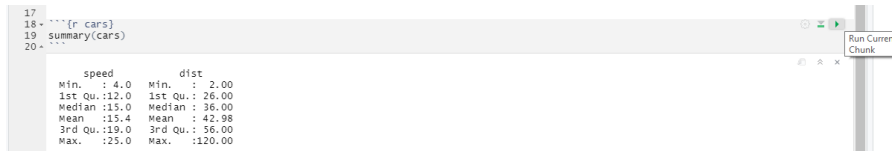
1 ---
2 title: "Untitled"
3 author: "Alexander (sasha) Pastukhov"
4 date: "29 10 2020"
5 output: html_document
6 ---
7
8 ```{r setup, include=FALSE}
9 knitr::opts_chunk$set(echo = TRUE)
10 ```
11
12 ## R Markdown
13

```

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.



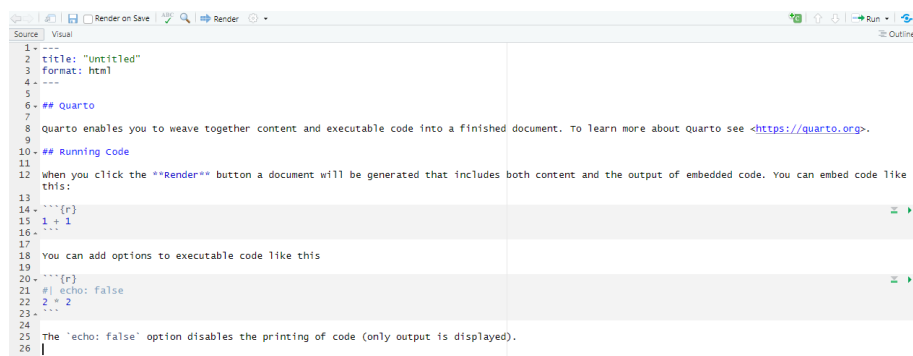
```

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

## 1.2.2 Quatro template



```

1 -> ---
2 -> title: "Untitled"
3 -> format: html
4 -> ---
5 ->
6 -> ## Quarto
7 ->
8 -> Quarto enables you to weave together content and executable code into a finished document. To learn more about quarto see <https://quarto.org>.
9 ->
10 -> ## Running Code
11 ->
12 -> When you click the "Render" button a document will be generated that includes both content and the output of embedded code. You can embed code like
13 -> this:
14 -> ```{r}
15 -> 1 + 1
16 -> ...
17 ->
18 -> You can add options to executable code like this
19 ->
20 -> ```{r}
21 -> #| echo: false
22 -> 2 * 2
23 -> ...
24 ->
25 -> The 'echo: false' option disables the printing of code (only output is displayed).
26 ->

```

Working with Quarto notebook is very similar but certain YAML header options are different (e.g., `format: html` instead of `output: html_document`) and chunk settings are inside the chunk (`#| echo: false`) instead of being inside curly brackets (`{r echo=FALSE}`). These differences become important when you go in deeper and want to build a website or write a book using Quarto but won't matter much for this seminar.

## 1.3 Exercise

For the today's exercise, I want you to familiarize yourself with markdown. Create a Quarto notebook for this. Go to [markdownguide.org](https://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 `.qmd` file to MS Teams. Please note that if you use an external image, you must submit it (zip everything into a single file) and the path that you use must be *relative*. Remember, if your file path is "c:/Documents/R-seminar/funny.png" chances are I do not have that file and that set of folders on my computer and your markdown won't render for me.



## 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 using R for analysis, 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 in general. 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, a concept of a variable is often described as a box you can put something in. A box has a name tag on it, which is the *name* of the variable. Whatever you put in is the *value* that you store.



This “putting in” concepts is reflected in R syntax

```
number_of_participants <- 10
```

Here, `number_of_participants` is the name of the variable (name tag for the box that 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 in R 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
```

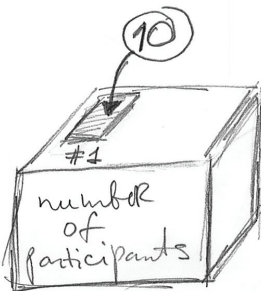
We are storing value 2 in a variable `x`. In the next line, the *right side* is evaluated first. This means that the current value of `x` is substituted in its place on the right side: `x + 5` becomes `2 + 5`. This expression computed and we get 7. Now, that the *right side* is fully evaluated, the value 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 singular values (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 is based on concepts of linear algebra instead of being a “normal” programming language like Python or C. 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*. 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*<sup>1</sup> (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 values you want to put in and stores these values one after another starting with slot #1<sup>2</sup>. Therefore, the example above `number_of_participants <- 10` creates a *vector* variable with one (1) slot and stores the value in it.



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

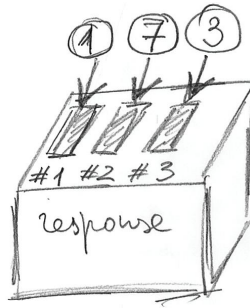
```
response <- c(1, 7, 3)
```

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 concatenating (or combining) values<sup>3</sup>. The figure below illustrates the idea:

<sup>1</sup>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.

<sup>2</sup>If you have experience with programming languages like Python, C, or Java: indexes in R start with 1, not with 0!

<sup>3</sup>I find this to be a very poor choice of name but we are stuck with it, so your only option is to get used to it.



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
```

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

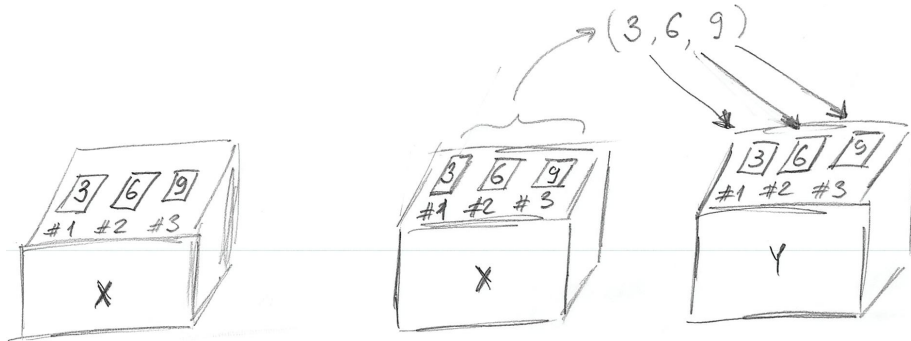
```
x <- c(3, 6, 9)
y <- x
print(x)
```

```
## [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 concatenate numbers, it concatenates three length one vectors (scalars) 3, 6, 9. Thus, concatenation 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 (subsetting)

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

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

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

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

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

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

Do exercise 4.

Unfortunately, vector indexing in R behaves in a way that may<sup>4</sup> catch you by surprise. Or, even worse, you will not even notice that your indexing does not work and screwed up your analysis. If your vector contains five values, you would expect that an index of 0 (negative indexes are special and will be discussed below) or above 5 generates an error. Not in R! Index of 0 is a special case and produces an *empty vector* (vector of zero length).

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

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

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

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

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

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

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

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

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

If you set an element with index **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 (“Not Available” / Missing Value).

```
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 what 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, remember, 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

---

<sup>4</sup>Who am I kidding? Will!



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 bleeding 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.

You can also use *negative* indexes. In that case, you *exclude* the value with that index and return or modify the rest<sup>5</sup>.

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

## [1] 1 2 4 5

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

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

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

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

Do exercise 6.

Finally, somewhat counterintuitively, the entire vector is returned if you do not specify an index in the square brackets. Here, lack of index means “everything”.

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

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

## 2.5 Names as an Index

As you've just learned, every slot in vector has its numeric (integer) index. However, this number only indicates an index (position) 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

---

<sup>5</sup>People who use Python, please be aware that negatives indexes in R and in Python behave completely differently!

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

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

```
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.

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

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

```
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>`

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

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

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

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

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

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

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

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

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

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

Finally, if you use a name that is not in the index, this is like using numeric index larger than the vector length. Just as for out-of-range numeric index, there will be neither error nor warning and you will get an NA back.

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

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

Do exercise 7.

## 2.6 Slicing

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

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

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

When constructing a vector index, you can put index values in the order you require (normal ascending order, 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 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:
x[c(-4, 2, -2)]
```

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

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 next time), 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**<sup>6</sup>.

```
3:7
```

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

---

<sup>6</sup>The latter is not so obvious, if you come from Python

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, 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? Unfortunately, R has a solution and that is not an error or a warning. Rather, if the length of the longer vector is a *multiple* of the shorter vector length, the shorter vector is “recycled”, i.e., repeated  $N$ -times (where  $N = \text{length}(\text{longer vector})/\text{length}(\text{shorter vector})$ ) and this length-matched vector is then used for the mathematical operation. Please note that this generate neither an error, nor a warning! This is another example of “convenience over safety” approach in R, so you should be very careful and *always* double-check length of your vectors. Otherwise, your code will work incorrectly and, *if you are lucky*, you might notice this.

To see how recycling works, 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  $\text{length}(\text{longer\_vector})$  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  $N = \text{ceiling}(\text{length}(\text{longer vector})/\text{length}(\text{shorter vector}))$  (where  $\text{ceiling}()$  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 (yay!) 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<sup>7</sup>.

```
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. Imagine that your vector, which 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 for block #1. R will **silently(!)** replicate the responses 10 times to match their length without ever(!) telling you about this. 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 a 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 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 values and compute 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

---

<sup>7</sup>No idea!

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()` or `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 or data analysis! 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 (on which R is built on). Finish this seminar by doing remaining exercises. Let's see whether R can still surprise you!

Do exercises 11-17.



## 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<sup>1</sup> is a vector in R. All examples in that chapter used numeric vectors which are two of the four primary types in R.

- **numeric**: Real numbers (*double precision floating point numbers* to be more precise) 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, which you will get if you divide a non-zero number by zero) and **NaN** (not a number). The latter looks similar to **NA** (Not Available / Missing Value) but is a different special case and are generated when you are trying to perform a mathematically undefined operation like diving by zero by zero or computing sine of infinity (see R documentation for details).
- **integer**: 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**: Logical or Boolean values of **TRUE** and **FALSE**. They can also be written as **T** and **F** but this practice is discouraged by the Tidyverse style

---

<sup>1</sup>Terms and conditions apply.

guide.

- **character**: 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". Note that a string **is not** a vector of characters. This would make a lot of sense but it is not. This is why indexing or slicing will not work and you need to use special functions that we will cover in a later chapter.

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.

- Convert to integer via `as.integer()`, use `is.integer()` to check whether vector consist only of integers. 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 if 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
- Convert to real number via `as.numeric()` / `as.double()`, check a value via `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 if it is a properly formed number, e.g. `as.double("12.2")` → 12.2 but `as.double("12punkt5")` is NA
  - from NA → NA
- Convert to logical TRUE/FALSE via `as.logical` and check a value via `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
- Convert to a character string via `as.character()` and check via `is.character()`<sup>2</sup>
  - numeric values are converted to a string representation with scientific notation being used for large numbers.

---

<sup>2</sup>Be aware that `str()` function has nothing to do with strings but displays a structure of an object. A very unfortunate choice of name but we are stuck with it.

- logical TRUE/T and FALSE/F are converted to "TRUE" and "FALSE".
- NA → NA

Do exercise 1.

## 3.2 All vector 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, which is why they are also called *atomic* vectors. This ensures that you can apply the same function or operation to the entire vector without worrying about type compatibility. However, this means 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 Lists

(Atomic) vectors are homogeneous lists of primary types items, whereas lists are lists that can contain *any* item including vectors of different primary type data, other lists, or other objects<sup>3</sup>. Otherwise, you work with them *almost* the same way as with vectors. You create a list via `list()` function and then you can concatenate lists via the same `c()` function, access individual elements via numeric index or names, use slicing, etc. But...

## 3.4 Subsetting, a fuller version

Unfortunately, this is the point where we need to talk about subsetting (accessing individual elements of a vector or of a list via indexes or names) yet again. This is because R has several<sup>4</sup> different ways of doing this that look similar and, to make things worse, *sometimes* produce identical results. I do not expect you to memorize or even fully appreciate all the intricate details (I am not even sure I know them all). But I do need you to understand the fundamental difference between the two ways of doing subsetting and to be aware of potential issues that can easily cause confusion. If the latter is the case, return to this section, read the official manual, or read the subsetting section in “Advanced R” book by Hadley Wickham.

---

<sup>3</sup>If you are familiar with Python, you can think of R lists as Python lists and dictionaries mixed together.

<sup>4</sup>Advanced R lists six(!) ways to subset atomic vectors, three(!) subsetting operators, and they can interact!

You already know subsetting via *single* square brackets: `x[1]`. This is called *preserving* subsetting because it returns a *part of an object* (vector or list) that your requested *as is*. I.e., if you used them to access part of a vector, you get a vector. If you used them to access a list or a table, you get a list or a table.

In addition, you can use *double* square brackets: `x[[1]]`. These are called *simplifying* subsetting because they extract *content* at that location. If you have a list of vectors `l` then `l[2]` (preserving) would return a single item list (with a vector as an item inside that list) but `l[[2]]` would return a vector stored at that location. A metaphor based on @RLangTip: “If list `x` is a train carrying objects, then `x[[5]]` is the object in car 5; `x[5]` is a train consisting only of car 5.”

Note that the simplifying notation allows you to extract content of only *one* item. Therefore, you cannot use it with slicing (extracting content of many items), negative indexes (even if you will end up with just one index), or zero index. The good news is that R will actually generate an error message instead of failing silently, although error messages will be both mysterious and *different* depending on exact circumstances. To deepen your feeling of arbitrariness of R design: using a non-existent *name* index will generate an error for vectors but not for lists (you get `NULL`)<sup>5</sup>.

Confusing? Let us go through some examples, so you can see the difference these two kinds of brackets make for vectors and lists.

### 3.4.1 Subsetting for (atomics) vectors

First, we create a named vector

```
x <- c("a"=1, "b"=2, "c"=3)
```

Preserving subsetting via `[]` returns a part of original vector, notice that the *name* associated with an index is retained.

```
x[2]
```

```
## b
```

```
## 2
```

Simplifying subsetting via `[[ ]]` extracts value at the location. Because the *name* is associated with an *index*, not with the content, it gets stripped off.

```
x[[2]]
```

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

Trying to extract multiple items will fail for *simplifying* subsetting.

```
x[[1:2]]
```

---

<sup>5</sup>Insert a hair-pulling picture of your preference here.

```
## Error in x[[1:2]]: attempt to select more than one element in vectorIndex
```

Same is true for negative indexes, even though we are excluding two out of three indexes, so end up with content from just one item.

```
x[[-(1:2)]]
```

```
## Error in x[[-(1:2)]]: attempt to select more than one element in vectorIndex
```

And the zero index generates an error as well but the error message is different.

```
x[[0]]
```

```
## Error in x[[0]]: attempt to select less than one element in get1index <real>
```

Using of a non-existent name will generate an error for *simplifying* but not for *preserving* subsetting.

```
x["a"]
```

```
## a
```

```
## 1
```

This also works but strips the name off.

```
x[["a"]]
```

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

This fails silently by returning NA.

```
x["d"]
```

```
## <NA>
```

```
## NA
```

But this will generate an error.

```
x[["d"]]
```

```
## Error in x[["d"]]: subscript out of bounds
```

**General rule:** when using vectors, always use `[]`. The only potential usage-case I can see is wanting to strip a name off a single item (not sure I ever needed to do this). If, for some reason, this is what you need, you should write a comment explaining that. Otherwise, everyone including future-you will be confused and think that this is a typo.

### 3.4.2 Subsetting for lists[#list-subsetting]

First, we create a named list that includes another list as one of the items

```
l <- list("a"=1, "b"=2, "c"=list(3, 4, 5))
```

Preserving subsetting via `[]` returns a *list* that is a part of original list, again the *name* associated with an index, so it is retained in the returned list.

```
l[2]
```

```
## $b
## [1] 2
```

Simplifying subsetting via `[[ ]]` extracts value at the location. This is why you get a *vector* (of length 1).

```
l[[2]]
```

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

What about our list inside of the list at position 3? Using preserving subsetting give us a list inside of the list (car of the train). Again, note that the top level list retains the name.

```
l["c"]
```

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

Using simplifying subsetting return a list that was inside the list at that location (object inside that car). Note that, as with the vector before, the name gets stripped off, as it belongs to the index (car of the train) and not the content (object inside the car).

```
l[["c"]]
```

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

Again, using multiple indexes will fail.

```
l[[1:2]]
```

```
## Error in l[[1:2]]: subscript out of bounds
```

Same for negative indexe.

```
l[[-(1:2)]]
```

```
## Error in l[[-(1:2)]]: attempt to select more than one element in integerOneIndex
```

And for the zero index (but notice a *different* error message).

```
l[[0]]
```

```
## Error in l[[0]]: attempt to select less than one element in getIndex <real>
```

Using a non-existent name index for lists with preserving subsetting (`[]`) will return a single item list with `NULL` as a content<sup>6</sup> but just `NULL` for simplifying one (`[]`). And none of them will generate an error.

```
l["d"]
```

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

```
## NULL
```

```
l[["d"]]
```

```
## NULL
```

**General rule:** use `[]` if you are interested in list or table content, e.g., you need a column from the table. Use `[]` if you interested in using a smaller (part of the original) list or table.

If this is the point when you want to start running around screaming or bang your head against the screen, please tell me, cause I would definitely join you. Let us do some exercises that might help to build some intuition. However, as you probably noticed already, subsetting in R is fairly haphazard, so you probably will still end up being confused or have a code that does not work for seemingly mysterious reasons.

Do exercise 3.

## 3.5 Yet another subsetting via \$

I am terribly sorry but I need to deepen your confusion and despair but telling you that R has yet *another* way of subsetting via a \$ sign. In fact, this is the most common way of subsetting for lists and tables that you will encounter almost everywhere. It is a shortcut version for *partial matching* of simplifying subsetting: `l$c` is the same as writing `l[["c", exact = FALSE]]`. The last new part `exact = FALSE` means that you can use an *incomplete* name after the \$ and R will guess what you mean. So, if you made a typo while meaning some completely different column name, R will second-guess your intentions and give you a variable that has the same beginning (no error, don't even hope).

---

<sup>6</sup>And NA as a name. Don't ask.

```
l <- list("aa" = 1, "bb" = 2, "bc" = 3)
l$a
```

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

The good news is that it won't autocomplete ambiguous cases. The bad news is that it won't generate an error and will return a NULL (just like `l[]` would).

```
l$b
```

```
## NULL
```

```
l[["b"]]
```

```
## NULL
```

My advice would be to stick to `l[]` notation despite the ubiquitous use of `$` out where. `l[]` is slightly more strict and more universal: You can hard code a column/item name or put it into a variable for indexing. Only hard-coding works for `$`.

```
l[["bb"]]
```

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

```
column_of_interest <- "bb"
l[[column_of_interest]]
```

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

### 3.6 Tables, a.k.a. data frames

R tables are a mix between lists and matrices. You can think about it as a list of vectors (columns) of *identical* length. 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 (note the capital V in the `View()`).



Click to view the table.

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

Do exercise 4.

All columns in a table **must** have the same number of items (rows). If the vectors are of different length, R will match their length via recycling of shorter vectors. 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.

```
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
```

```
##   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 *multiple-of-length* rule does not work, R generates an error (finally!).

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

```
## Error in data.frame(numeric_column = 1:7, character_column = c("A", "B", : argument
```

Do exercise 5.

### 3.7 Subsetting tables

One way to think about a table as a list of columns (vectors). Hence, both preserving (`[]`) and simplifying (`[[[]]`) subsetting will work as you would expect returning either a `data.frame` (`[]`) or a *vector* that was a column your were interested in (`[[[]]`).

The preserving returns a *table* with *one* column.

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

```
# via index
```

```
our_first_table[1]
```

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

```
# via name
```

```
our_first_table['numeric_column']
```

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

```
# via slicing
```

```
our_first_table[1:2]
```

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

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

The simplifying returns a *vector*.

```
# via $ notation
our_first_table$numeric_column
```

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

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

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

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

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

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]]  
the\_table[["N"]]

1 2 3 4 5

The only new thing is that, because tables are two-dimensional, you can use preserving subsetting to extract or access a rectangular region within a table. To select a subset rows and columns you write `table[rows, columns]`. If you omit either rows or columns this implies that you want *all* rows or columns.

```
# getting ALL rows for the FIRST column -> confusingly this gives you a VECTOR,
# so even though you used [] they work as simplifying subsetting
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`
# this gives you DATA.FRAME
our_first_table[,]
```

```
## numeric_column character_column logical_column
## 1          1          A          TRUE
## 2          2          B          FALSE
## 3          3          C          TRUE
# getting SECOND element of the THIRD column, returns a VECTOR
our_first_table[2, 3]
```

```
## [1] FALSE
# getting first two elements of the logical_column, returns a VECTOR
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 6.

### 3.8 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 in what comes bundled with it (very little, as a matter of fact) but in community developed libraries that extend it. We already discussed how you install libraries. To use a library in your code, you use `library()`

function<sup>7</sup>. So, to use *tidyverse* library that you already installed, you simply write

```
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)<sup>8</sup>. This overwriting of one function by another can lead to very odd looking mistakes. In my case I think that I am using `dplyr::filter()` and get confused by error messages that I get (they are not really informative). The first time I did this, it took me an hour to figure it out. Here are the warnings I should have paid attention to.

```
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 cluttering it with `library::` prefixes. Thus, you will need to find a balance between disambiguation and readability.

```
library(tibble)

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

---

<sup>7</sup>It has a sister function `require()` but it should be used inside functions and packages not in scripts or notebooks

<sup>8</sup>There is also a filter function in stats library.

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

# 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 the first chunk of the notebook. In case of Quarto and Rmarkdown notebooks, if you call this chunk “setup”, RStudio will run it at least once before any other chunk, ensuring that your libraries are always initialized (take a look at the very first chunk in the exercise notebook). Word of advice, keep your library list in alphabetical order. Libraries are very specialized, so 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.

Another note, sometimes you may need to install packages and it might be tempting to include `install.packages("library-I-needed")` into the notebook or R script. Do not do this, as it will run installation every time you or somebody else run that chunk/script or knit the notebook, which could waste time, lead to errors (if you are offline), mess up other packages, etc. RStudio is smart enough to detect missing packages and will display an alert about this, offering to install it.



### 3.9 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 symbol in columns names (including spaces), prints out only the 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 give you a warning but only if you use the `$` notation<sup>9</sup>.

```
library(tibble)

# data.frame will return NULL silently
df <- data.frame(b = 1)
print(df[["A"]])

## NULL

# tibble will return NULL for a variable that does not exist for [[]]
tbl <- tibble(b = 1)
print(tbl[["A"]])

## NULL

# but a warning here
tbl$A

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

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

### 3.10 Tribble, table from text

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

```
tribble(
  ~x, ~y,
  1,  "a",
```

---

<sup>9</sup>Arbitrariness strikes again! But this also means that `$` is safer to use with tibbles but not with data.frames.

```
2, "b"
)

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

Do exercise 7.

### 3.11 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 an example data set, you need to import the library that it is part of and then load them by writing `data(tablename)`. For example, to use `mpg` data on fuel economy from `ggplot2` package, you need to import the library first, and then call `data(mpg)`.

```
library(ggplot2)
data(mpg) # this creates 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~ f      18    29 p    comp~
## 2 audi          a4         1.8  1999    4 manu~ f      21    29 p    comp~
## 3 audi          a4         2    2008    4 manu~ f      20    31 p    comp~
## 4 audi          a4         2    2008    4 auto~ f      21    30 p    comp~
## 5 audi          a4         2.8  1999    6 auto~ f      16    26 p    comp~
## 6 audi          a4         2.8  1999    6 manu~ f      18    26 p    comp~
## 7 audi          a4         3.1  2008    6 auto~ f      18    27 p    comp~
## 8 audi          a4 quattro  1.8  1999    4 manu~ 4      18    26 p    comp~
## 9 audi          a4 quattro  1.8  1999    4 auto~ 4      16    25 p    comp~
## 10 audi         a4 quattro  2    2008    4 manu~ 4      20    28 p    comp~
## # i 224 more rows
```

### 3.12 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 is `read.csv()` function that 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 re-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, the most important difference between `read.csv()` and `read_csv()` is that the latter can constrain 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 will inform (warn) you about the table structure it deduced.

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

```
## Rows: 3 Columns: 5
## -- Column specification -----
## Delimiter: ","
## chr (1): Participant
## dbl (3): Block, Trial, Contrast
## lgl (1): Correct
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
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 warning, which gets even more annoying if you need to read many CSV files, is there for a reason: it wants to annoy you! You can turn it off via `show_col_types = FALSE` but I strongly recommend against this. Instead, you should specify the column structure yourself via `col_types` parameter. The simplest way to do this is via `spec()` function, as suggested by the printout.

```
spec(results)
```

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

This is a specification that the reader prepared for you. 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 just *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()` and `read_csv()` both 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 a 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, you can see the details via `problems()` function).

```

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() but
    Contrast = col_double(),
    Correct = col_integer()))

```

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

```

## Warning: One or more parsing issues, call `problems()` on your data frame for details,
## e.g.:
##   dat <- vroom(...)
##   problems(dat)

```

```
problems()
```

Personally, I would prefer for `read_csv()` to loudly fail with an error 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, run `spec()` function on it, and copy-paste the output into the code adjusting as necessary.

You will need `face_rank.csv` file for exercise 8. 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 (it does not happen as I write this, but it was an issue at some point of time). Just rename it back to `face_rank.csv`, because the file itself is not converted to an Excel, it is only the extension that gets changed<sup>10</sup>.

Do exercise 8.

Note that you have another option with online files, you can use their URL in-place of the filename and R will take care of downloading it for you (but, obviously, you have to be online for this trick to work).

```
read_csv("https://alexander-pastukhov.github.io/data-analysis-using-r-for-psychology/d
```

### 3.13 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")`.

For the exercise 9, you need `face_rank.xlsx` file for it. Download it or use the URL directly. Also, think about which library you need to do to perform the task. Add this library to the “setup” chunk at the beginning, not to the exercise chunk itself!

Do exercise 9.

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 (at least in Germany) 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

---

<sup>10</sup>why? No idea, ask Google!

of human genome data tables by turning some values into dates because why not<sup>11</sup>? 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.

### 3.14 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 a 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.) to export data into a CSV. The second way is to use various R libraries, starting with `foreign`, which can handle most typical cases, e.g., SPSS, SAS, State, or Minitab. The problem here is that all 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 table (`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.

For the next exercise, you need `band_members.sav` file. You need `read.spss()` function from library `foreign` (again, if you need to import it, do it in the “setup” chunk).

Do exercise 10.

### 3.15 Writing and reading a single object

There is another option for saving and reading data in R via `saveRDS()` and `readRDS` functions<sup>12</sup>. `saveRDS()` saves an arbitrary object (vector, list, table, model fit, etc.), whereas `readRDS` reads it back. This is useful, if your table is very large, so using CSV is inefficient, or if you want to save something that is not a table, again, something like a model fit for later use. This is an R-specific format, so you will not be able to use this file with other program, so

<sup>11</sup>Best joke on that that I've heard so far is: Optimist thinks that the glass is half full, pessimist thinks it is half empty, Excel thinks it is second of February.

<sup>12</sup>This is another example of odd inconsistencies in R. Why `readRDS()` instead of `loadRDS()`? Or `writeRDS()` if you read it.

you probably will not use this frequently but it is a good option to keep in mind.  
Do exercise 11.

### 3.16 Wrap up

We have started with vectors and now extended them to tables. Our final stop, before getting into exciting visualizations, are functions, the other most important pillar of R.

## Chapter 4

# Functions! Functions everywhere!

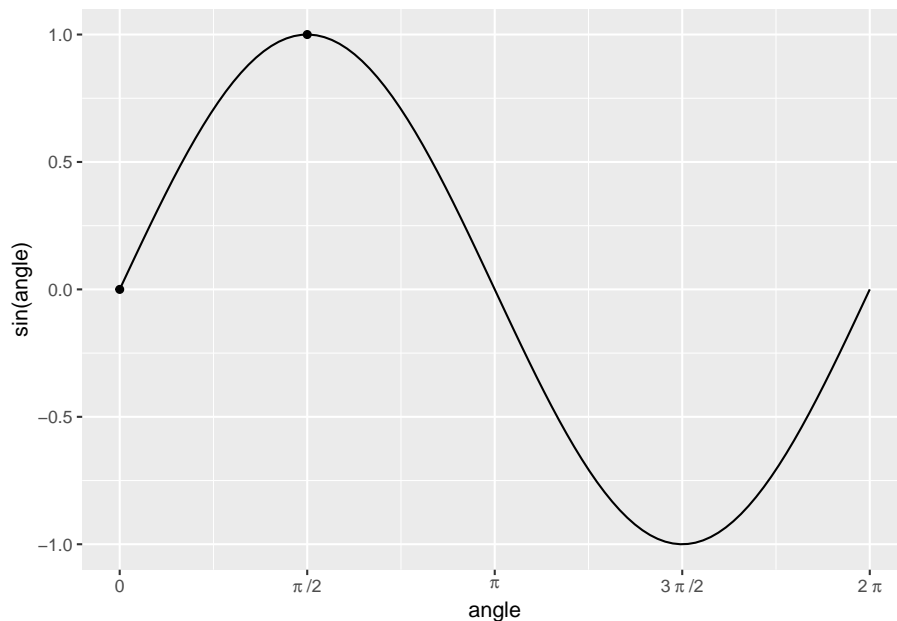
In this chapter you will learn about *functions* in R, as they are the second most important concept in R and are everywhere (just like vectors and lists). You will also learn how to *pipe* your computation through series of functions without creating a mess of temporary variables or nested calls. Don't forget to download the notebook.

### 4.1 Functions

In the previous chapters, you have learned that you can store information in variables — “boxes with slots” — as vectors or as tables (bundles of vectors of equal length). To use these stored values for computation you need functions. In programming, function is an *isolated* code that receives some (optional) input, performs some action on it, and, optionally, returns a value<sup>1</sup>. The concept of functions comes from mathematics, so it might be easier to understand them using R implementation of mathematical functions. For example, you may remember a sine function from trigonometry. It is typically abbreviated as **sin**, it takes a numeric value of angle (in radians) as its *input* and returns a corresponding value between -1 and 1 (this is its *output*):  $\sin(0) = 0$ ,  $\sin(\pi/2) = 1$ , etc.

---

<sup>1</sup>Function that does not return a value, probably, generates its output to a console, external file, etc. There is little point in running a function that does not affect the world.



In R, you write a function using the following template<sup>2</sup>

```
name_of_the_function <- function(parameter1, parameter2, parameter3, ...){
  ...some code that computes the value...
  return(value)
}
```

A `sin` function with a single parameter `angle` would look something like this

```
sin <- function(angle){
  ...some math that actually computes sinus of angle using value of angle parameter ..
  return(sin_of_angle)
}
```

Once we have the function, we can use it by *calling* it. You simply write `sin(0)`<sup>3</sup> and get the answer!

```
sin(0)
```

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

As you hopefully remember, all simple values are (atomic) vectors, so instead of using a scalar 0 (merely a vector of length of one) you can write and apply this function to (compute sinus for) every element in the vector.

<sup>2</sup>Did you notice the `<-` assignment? Yes `name_of_the_function` is just a variable in which you *literally* store R-code for execution later. I will explain this in detail below.

<sup>3</sup>I've cheated here by using R implementation of `sin()`.



```
sin(seq(0, 3.141593, length.out = 5))
```

```
## [1] 0.000000e+00 7.071068e-01 1.000000e+00 7.071066e-01 -3.464102e-07
```

You can think of functions parameters as local function variables those values are set before the function is called. A function can have any number of parameters, including zero<sup>4</sup>, one, or many parameters. For example, an arctangent `atan2` function takes 2D coordinates (y and x, in that order!) and returns a corresponding angle in *radians*.

```
atan2(c(0, 1), c(1, 1))
```

```
## [1] 0.0000000 0.7853982
```

A definition of this function would look something like this

```
atan2 <- function(y, x){
  ...magic that uses values of y and x parameters...
  ...to compute the angle_in_rad value...
  return(angle_in_rad);
}
```

Do exercise 1.

## 4.2 Writing your own function

Let us start practicing computing things in R and writing functions at the same time. We will begin by implementing a very simple function that doubles a given number. We will write this function in steps. First, think about how you would name<sup>5</sup> this function (meaningful names are your path to readable and reusable code!) and how many parameters it will have. Write the definition of the function without any code inside of wiggly brackets (so, no actual computation or a return statement at the end of it).

Do exercise 2.1

Next, think about the code that would *double-the-value* based on the parameter. This is the code that will eventually go inside the wiggly brackets. Write that code (just the code, without the bits from exercise 2.1) in exercise 2.2 and test it by creating a variable with the same name as your parameter inside the function. E.g., if my parameter name is `the_number`, I would test it as

```
the_number <- 1
...my code to double the value using the_number variable...
```

Do exercise 2.2

---

<sup>4</sup>This, probably, means that the function always does the same thing or a random thing and you cannot influence this.

<sup>5</sup>`double_or_nothing?`

By now you have your formal function definition (exercise 2.1) and the actual code that should go inside (exercise 2.2). Now, we just need to combine them by putting the code inside the function and *returning* the value. You can do this two ways:

- 1) you can store the results of the computation in a separate local variable and then return that variable,
- 2) return the results of the computation directly

```
# 1) first store in a local variable, then return it
result <- ...some computation you perform...
return(result)
```

```
# 2) returning results of computation directly
return(...some computation you perform...)
```

Do it *both* ways in exercises 2.3 and 2.4. Call the function using different inputs to test that it works.

Do exercise 2.3 and 2.4

More practice is always good, so write a function that converts an angle in *degrees* to *radians*. The formula (hint: Constants) is

$$rad = \frac{deg \cdot \pi}{180}$$

Decide whether you want to have an intermediate local variable inside the function or to return the results of the computation directly. My personal preference for writing function is to follow the inside-out sequence when writing functions as in exercise 2. Decide on function definition and parameter names, then create variables with same names as parameters to write and debug the code. Once I am sure the code is working, I put it into a function, *restart R session* or clear the workspace to get rid of temporary variables, and run the function to check that everything works as it should<sup>6</sup>.

Do exercise 3

### 4.3 Scopes: Global versus Local variables

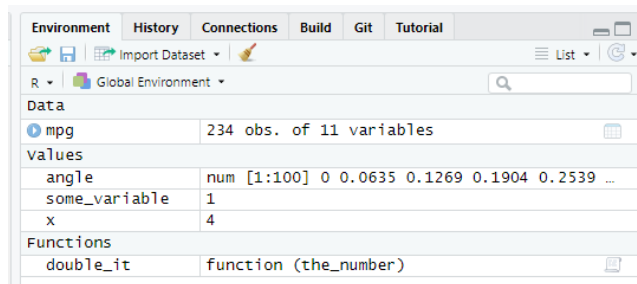
I suggested that you use a variable to store the results of double-it-up computation before returning it. But why did I call it *local*? This is because each function has its own *scope* (environment with variables and functions) that is (mostly) independent from the scope of the global script. Unfortunately, environment scopes in R are different and more complicated than those in other

---

<sup>6</sup>If you have experience with a proper debugger, RStudio has you (somewhat) covered: you can debug your code with all the usual tools but only R-scripts, not the notebooks that we are using here (at least it is much more complicated for the latter).

programming languages, such as Python, C/C++, or Java, so always pay attention and be careful in the future.

The *global* scope/environment is the environment for the code *outside* of functions. All *global* variables and functions, the ones that you define in the code *outside* of functions (typing in the console, running scripts, running chunks of code in notebooks, etc.), live where. You can see what you have in the global scope at any time by looking at *Environment* tab (note the *Global Environment* tag).



In my case, it has one table (`mpg`, all tables go under *Data*), three vectors (`angle`, `some_variable`, and `x`, all vectors go under *Values*), and an example function from exercise #2 that I created (`double_it`, all functions go under *Functions*, makes sense). However, it has no access to parameters of the functions and variables that you define inside these function. When you run a function, it has its own scope/environment that includes *parameters* of the function (e.g., `the_number` for my `double_it` function and the value it was assigned during the call), any *local* variables you create inside that function (e.g., `result <- ...some computation you perform...` creates such local variable), and a **copy(!)** of all *global* variables<sup>7</sup>. In the code below, take a look at the comments that specify the accessibility of variables between global script and functions (ignore glue for a moment, it *glues* a variable's value into the text, so I use it to make printouts easier to trace)<sup>8</sup>

```
# this is a GLOBAL variable
global_variable <- 1

i_am_test_function <- function(parameter){
```

<sup>7</sup>You can write to a parent (e.g., global) environment from inside the function via `<-` operator. However, this should be your last resort, as it makes the code brittle and harder to understand and debug, because it depends not only on local things but on global variables that may or may not be where at any given moment.

<sup>8</sup>These rules — a **copy** of a global scope is accessible inside a function but function scope is inaccessible from outside — should be how *you* write your code. However, R's motto is "anything is possible", so be aware that *other people* may not respect these rules. This should not be an issue most of the time but if you are curious how R can let you break *all* the rules of good responsible programming and do totally crazy things, read *Advanced R* book by Hadley Wickham. However, it is really **advanced**, aimed primarily at programmers who develop R and packages, not at scientists who use them.

```

# parameter live is a local function scope
# its values are set when you call a function
print(glue(" parameter inside the function: {parameter}"))

# local variable created inside the function
# it is invisible from outside
local_variable <- 2
print(glue(" local_variable inside the function: {local_variable}"))

# here, a_global_variable is a LOCAL COPY of the the global variable
# of the same name. You can use this COPY
print(glue(" COPY of global_variable inside the function: {global_variable}"))

# you can modify the LOCAL COPY but that won't affect the original!
global_variable <- 3
print(glue(" CHANGED COPY of global_variable inside the function: {global_variable}")
}

print(glue("global_variable before the function call: {global_variable}"))

## global_variable before the function call: 1
i_am_test_function(5)

## parameter inside the function: 5
## local_variable inside the function: 2
## COPY of global_variable inside the function: 1
## CHANGED COPY of global_variable inside the function: 3
# the variable outside is unchanged because we modify its COPY, not the variable itself.
print(glue("UNCHANGED global_variable after the function call: {global_variable}"))

## UNCHANGED global_variable after the function call: 1

```

Do exercise 4 to build understanding of scopes.

## 4.4 Function with two parameters

Let us write a function that takes *two* parameters — *x* and *y* — and computes *radius* (distance from (0,0) to (*x*, *y*))<sup>9</sup>. The formula is

$$R = \sqrt{x^2 + y^2}$$

This is very similar to exercises 2 and 3, with number of parameters being the only difference. Again, I would suggest first writing and debugging the code

<sup>9</sup>This function is complementary to `atan2()`, as two of them allow transformation of coordinates from Cartesian to polar coordinate system

in a separate chunk and then putting it inside of the formal function definition (and the testing it again!)

Do exercise 5.

## 4.5 Table as a parameter

So far, we only passed vectors (a.k.a. values) to functions but you can pass any object including tables<sup>10</sup>. Let us use `mpg` table from the *ggplot2* package. Write a function that takes a table as a *parameter*. This means that the function should not assume that a table with this name exists in the global environment. Do not use `mpg` as a parameter name (makes it confusing and hard to understand which table global or local you actually mean), call it something else<sup>11</sup>. The function should compute and return average miles-per-gallon efficiency of city `cty` and highway `hwy` test cycles for each car (so, you need to compute 234 values). Do it *two* ways. First, compute and return *a vector* based on the table passed as parameter. Second, do the same computation but add the result *to the table that function received as a parameter* (call the new column `avg_mpg`) and return the entire table (remember, modifying table is not enough, as you are working on a copy).

Do exercise 6.

Let us write another function that computes mean efficiency for a particular cycle, either city or highway, over all the cars (so single value as the output). For this, the function will take *two* parameters: 1) the table itself and 2) a *string* (text variable) with the name of the column. Then, you use column name to access it via simplifying `[[ ]]` subsetting. To summarize, your function takes 1) a table and 2) a string with a column name and returns a single number (mean for the specified column). E.g.

```
average_efficiency(mpg, "cty") # should return 16.85897
```

Do exercise 7.

## 4.6 Named versus positional arguments

Remember than we talked about the assignment statement, I noted that you should always use `<-` operator for assignments but warned you that you will encounter an alternative `=` operator when we will use functions. You also may have noticed me using it, for example, in `seq(0, 2*pi, length.out = 100)`. Why did I use name for the third parameter and not the other two? Why did I have to use `length.out=` at all? This is because in R you can pass arguments

<sup>10</sup>And even functions themselves, not just what they computed! This is part of *functional programming* you will learn about later.

<sup>11</sup>I tend to use `df`, if it is reasonably easy to deduce what sort of `data.frame` I mean.

by *position* (first two arguments in that example) or by *name* (that would be `length.out =`).

Then you pass arguments by *position*, values are put into corresponding arguments in the order you supplied them in. I.e., the first value goes into the first parameter, the second into the second, etc. Then you specify arguments by *name*, you explicitly state which argument gets which value `arg = value`. Here, the exact order does not matter and you can put arguments in the order you need (i.e., an order that makes understanding the code easier). You can also mix these two ways and, R being R, you can really mix them interleaving positional and named arguments any way you like. That being a case, never use named arguments before positional ones. Each time you use a named argument, its position is taken out and that *changes* position index for remaining positional arguments. This is an almost certain way to put a value into a wrong parameter and, at the same time, create an error that is really hard to find. Thus, despite all the flexibility that R gives you to confuse yourself and others, use only all-positional-followed-by-all-named-arguments order, e.g., `seq(0, 2*pi, length.out = 100)`.

When should you use which? That depends solely on which way is clearer or possible. For a single parameter or widely used mathematical functions, like `mean` or `sin` there is little point in using names (particularly, because the only argument is named `x`). At the same time, I *always* use named parameters for `atan2` simply because I am never 100% sure about the order (i.e., `x` followed by `y` occurs far more often). At the same time, formulas for statistical models in R have a very specific look and are typically a first argument, so it is probably redundant to specify that `formula = y ~ x`. Returning to the `seq()` function, the `length.out` is actually its *fourth* argument with `by` (an alternative way to define how many values you will get in the sequence) being third. But because both parameters define, essentially, the same thing, it is *impossible* to specify `length.out` using positional arguments only. Finally, if a function has more than a couple of parameters, it is probably a good idea to use named arguments. In short, use you own better judgement on whatever makes understanding the code easier.

Do exercise 8.

## 4.7 Default values

When you write a function, you can combine simplicity of its use with flexibility via *default values*. I.e., *some* parameters can have sensible or commonly used values but, if desired, a user can specify their own values to modify functions behavior. For example, function `mean` has *three* parameters. You always need to specify the first one (`x`, a vector of values you are computing the mean of). But you can also specify trimming via `trim` and whether it should ignore

NA via `na.rm`<sup>12</sup>. By default, you do not trim (`trim = 0`) and do not ignore NA (`na.rm = FALSE`). These defaults are sensible as they produce a typically expected behavior. At the same time their existence means that you can fine-tune your mean computation the way you require.

When writing your own function, you specify the default values when you define an argument. E.g., here, the second parameter `r`, the radius, has a default value of 1, so you can only specify the direction of the vector to compute (x, y) coordinates for a (default) *unit* vector.

```
polar_to_cartesian <- function(angle, r=1) {
  # ...
}
```

Do exercise 9.

## 4.8 Nested calls

What if you need to call several functions in a single chain to compute a result? Think about the function from exercise #3 that converts degree to radians. Its most likely usage scenario is to convert degrees to radians and use that to compute sine (or some other trigonometric function). There are different ways you can do this. For example, you can store the angle in radians in some temporary variable (e.g., `angle_in_radians`) and then pass it to the sine function during the next call.

```
angle_in_radians <- deg2rad(90) # function returns 1.570796, this value is stored in angle_in_radians
sin(angle_in_radians) # returns 1
```

Alternatively, you can use the value returned by `deg2rad()` *directly* as a parameter for function `sin()`

```
sin(deg2rad(90)) # returns 1
```

In this case, the computation proceeds in an *inside-out* order: The innermost function gets computed first, the function that uses its return value is next, etc. Kind of like assembling a Russian doll: you start with an innermost, put it inside a slightly bigger one, now take that one put it inside the next, etc. Nesting means that you do not need to pollute you memory with temporary variables<sup>13</sup> and make your code more expressive as nesting explicitly informs the reader that intermediate results are of no value by themselves and are not saved for later use.

Do exercise 10.

<sup>12</sup>If you do not ignore NA then the mean of a vector with at least one NA in it is NA as well

<sup>13</sup>The worst case scenario is when you use same `temp` variable for things like that, forget to initialize / change its value properly at some point, spend half-a-day trying to understand why your code does not generate an error but results don't make sense.

## 4.9 Piping

Although nesting is better than having tons of intermediate variables, lots of nesting can be mightily confusing. Tidyverse has an alternative way to chain or, in Tidyverse-speak, pipe a computation through a series of function calls. The magic operator is `|>` (that’s the pipe). There is another pipe operator that you are very likely to encounter: `%>%`. This is the “original” pipe introduced by `magrittr` library and widely used, particularly in Tidyverse, before R version 4.1.0 provided a built-in `|>` solution. As far as you are concerned, these two operators are synonyms and can be used interchangeably (same goes when you read somebody else’s code). However, you do need to import either `tidyverse` or one of its libraries to enable `%>%` and `|>` is an official future, so I strongly recommend to using `|>` in your code<sup>14</sup> Here is how a pipe transforms our nested call

```
sin(deg2rad(90)) # returns 1

deg2rad(90) |> sin() # also returns 1

90 |> deg2rad() |> sin() # also returns 1
```

Do exercise 11.

All functions you used in the exercise had only one parameter because **single-output** `|>` **single-input** piping is very straightforward. But what if one of the functions takes more than one parameter? By default, `|>` puts the piped value into the *first* parameter. Thus `4 |> radius(3)` is equivalent to `radius(4, 3)`. Although this default is very useful (the entire Tidyverse is build around this idea of piping a value into the *first* parameter to streamline your experience), sometimes you need that value as some *other* parameter (e.g., when you are pre-processing the data before piping it into a statistical test, the latter typically takes **formula** as the first parameter and **data** as second). For this, you can use a special underscore variable `_` for `|>` and a dot variable `.` for `%>%`. The latter is more flexible, as it can be used for both named and positional arguments many times, whereas the former is more restrictive and can be used only with named parameters. Both of these variables are hidden<sup>15</sup> temporary variable that holds the value you are piping through via either `|>` or `%>%`<sup>16</sup>. Thus, for a standard pipe `|>`

```
z <- 4
w <- 3
z |> radius(w) # equivalent to radius(z, w)
```

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

<sup>14</sup>It also uses 1 symbol less!

<sup>15</sup>As in: It won’t show up in your Global Environment tab.

<sup>16</sup>Yes, it is the same variable that you use over-and-over again but it is clean up after each call, so you are not in danger of incidentally using a value from a previous call.



```
# Note the underscore and that we use all named parameters!
z |> radius(x = _, y = w) # equivalent to radius(z, w)
```

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

```
z |> radius(x = w, y = _) # equivalent to radius(w, z)
```

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

```
# Using it without name of the parameter generates a mistake
```

```
z |> radius(_, w)
```

```
# Using it twice is also not allowed
```

```
z |> radius(x = _, y = _)
```

```
## Error: pipe placeholder can only be used as a named argument (<text>:2:6)
```

For the magrittr pipe `%>%` you can use it anyway you want and even multiple times.

```
z %>% radius(w) # equivalent to radius(z, w)
```

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

```
# Note the underscore and that we use all named parameters!
```

```
z %>% radius(x = ., y = w) # equivalent to radius(z, w)
```

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

```
z %>% radius(x = w, y = .) # equivalent to radius(w, z)
```

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

```
# Now positional parameters
```

```
z %>% radius(., w) # equivalent to radius(z, w)
```

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

```
z %>% radius(w, .) # equivalent to radius(w, z)
```

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

```
# Now same value twice
```

```
z %>% radius(., .) # equivalent to radius(z, z)
```

```
## [1] 5.656854
```

As you can see `%>%` is more flexible but, to be honest, I have realized that I never really bumped into the limitations of the standard pipe `|>` until I started to compile these examples. Thus, my advice is to stick to `|>` and use `%>%` only if you actually need it (R is so liberal that at least some restrictions are good.)

Do exercise 12.

## 4.10 Function is just a code stored in a variable

Did you spot the assignment `<-` operator in function definitions and wondered about this? Yes, this means that you are literally storing function code in a variable. So when you call this function by name, you are asking to run the code stored inside that variable. You can appreciate that fact by typing a name of a function you wrote *without* round brackets, e.g. `double_the_value` or `radius`, and, voila, you will see the code. This means that function name is not really a name (as in some other programming languages), rather it is a name of a variable function's code is stored in. So you can use a variable *with a function code inside* just the way you treat any other variable. For example, you can copy it (do `radius2 <- radius` and then `radius2` will work exactly the same way). Or pass it as an argument of another function (effectively, copy its code into a parameter), which will be very handy when you learn about bootstrapping (needs data and a function) or about applying/mapping functions to data.

## 4.11 Functions! Functions everywhere!

*Every* computation you perform in R is implemented as a function, even if it does not look like a function call. For example, `+` addition operator is a function. Typically, you write `2 + 3`, so no round brackets, no comma-separated list of parameters, it *looks* different. But this is just a special implementation of a function call known as function operator that makes code more readable for humans. You can actually call `+` function the way you call a normal function by using backticks around its name.

```
2 + 3
```

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

```
# note the `backticks` around +
`+`(2, 3)
```

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

Even the assignments operator `<-` is, you've guessed it, a function

```
`<-`(some_variable, 1)
some_variable
```

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

This does not mean that you should start using operators as functions (although, if it helps to make a particular code clearer, then, why not?), merely to stress that there is only one way to program any computation in R — as a function — regardless of how it may appear in the code. Later on, you will learn how to apply functions to vectors or tables (or, a Tidyverse version of that, how to use functions to map inputs to outputs), so it helps to know that you can apply *any* function, even the one that does not look like a function.

## 4.12 Using (or not using) explicit return statement

In the code above I have always used the `return` function. However, explicit `return(some_value)` can be omitted if it is the *last* line in a function, you just write the value (variable) itself:

```
some_fun <- function(){  
  x <- 1  
  return(x)  
}  
  
some_other_fun <- function(){  
  x <- 2  
  x  
}  
  
yet_another_fun <- function(){  
  3  
}  
  
some_fun()
```

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

```
some_other_fun()
```

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

```
yet_another_fun()
```

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

The lack of `return` function in the final line is actually an officially recommended style of Tidyverse but, personally, I am on the fence about this approach because “explicit is better than implicit”. This omission may be reasonable if it comes at the very end of a long pipe but, in general, I would recommend using an explicit `return()`.



## Chapter 5

# ggplot2: Grammar of Graphics

In previous chapters, you have learned about tables that are the main way of representing data in psychological research and in R. In the following ones, you will learn how to manipulate 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 (or, at least, recommended) 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 relationships between variables. Don't forget to download the notebook.

### 5.1 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 be 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 be 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.Attractiveness
1	6	4	3	4	7
2	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 *judgments*. The latter are a variable but it is hidden in column names. Thus, a *tidy* table for this data would be

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

This table is (very) tidy and it makes it easy to group data by every different combination of variables (e.g., by face and judgment, by 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 to begin with 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 learning how to tidy it up but first let us see how an already tidy data makes it easy to visualize relationships in it.

## 5.2 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, 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. You need to get used to it but once you do, you probably will never want to go back.

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 individual 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. I simulate a response as

$$Response = Normal(\mu = 1, \sigma = 0.2) - Normal(\mu = 2 * ConditionIndex, \sigma = 0.4) + Normal(\mu = Intensity, \sigma = 0.4)$$

Condition	Intensity	Response
A	1	-0.4168134
B	1	-2.5816069
C	1	-4.8843430
A	2	0.4557152
B	2	-2.1316479
C	2	-2.1967557
A	3	2.6754808
B	3	0.2503168
C	3	-2.1646617
A	4	3.0076320
B	4	0.5210436
C	4	-0.4479474
A	5	4.3557439
B	5	0.7999901
C	5	-0.4812187
A	6	4.6898423
B	6	3.5706158
C	6	1.7524839
A	7	5.5386521
B	7	3.9014439
C	7	1.7979598
A	8	7.8400025
B	8	5.5283233
C	8	3.0614365

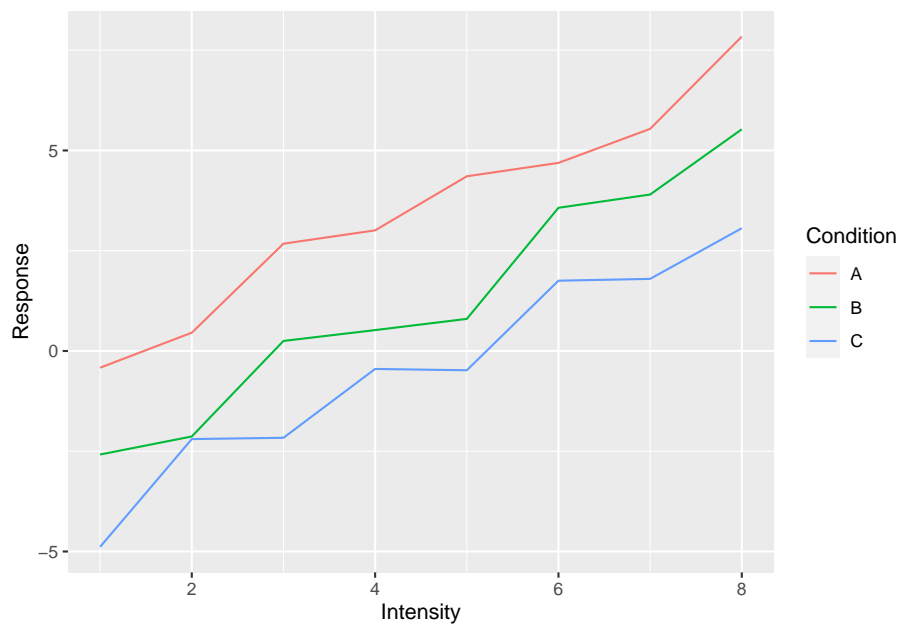
We plot this data by 1) *defining aesthetics* (mapping **Intensity** on to x-axis, **Response** on y-axis, and **Condition** on color) and 2) *adding* lines to the plot (note the plus<sup>1</sup> in `+ geom_line()`).

```
ggplot(data=simple_tidy_data, aes(x = Intensity, y = Response, color=Condition)) +  
  geom_line()
```

---

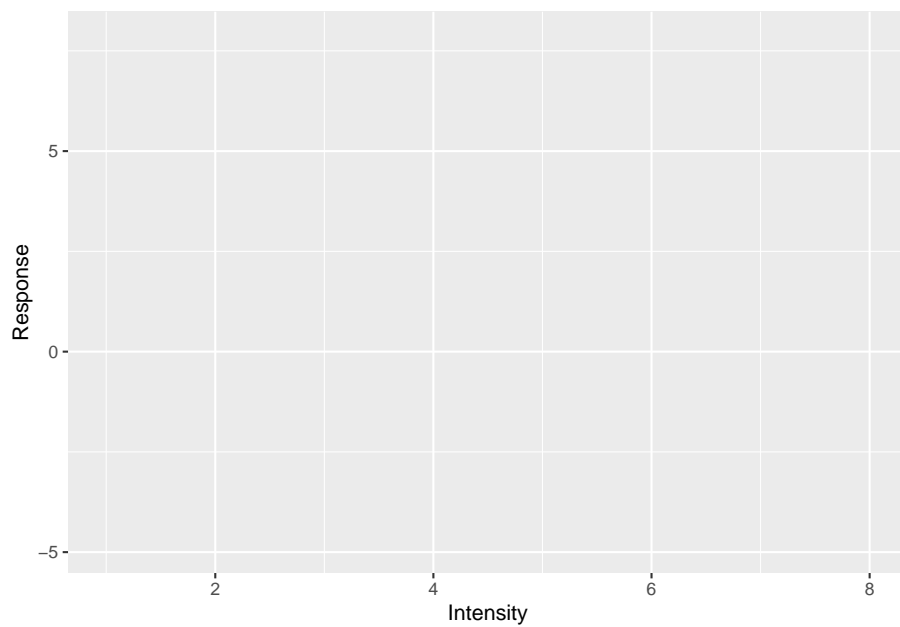
<sup>1</sup>One of a potentially confusing bits is usage of `+` in `ggplot2` but of pipe `|>` everywhere else. The difference is deliberate and fundamental. Pipe `|>` passes the output to the next function, `+` adds something to the already existing plot.





As I already wrote, technically, the only thing you need to define is aesthetics, so let us not add anything to the plot dropping the `+ geom_line()`.

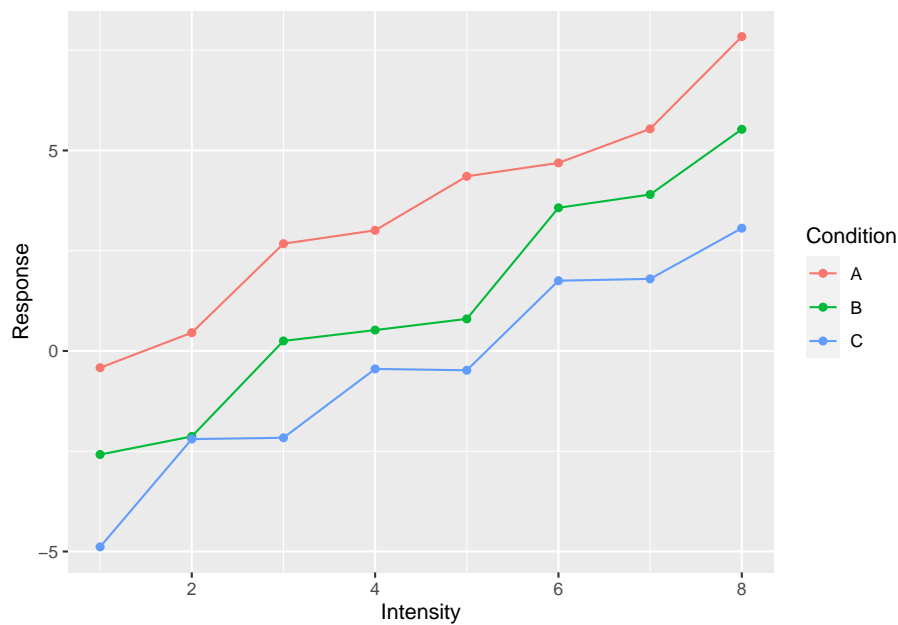
```
ggplot(data=simple_tidy_data, aes(x = Intensity, y = Response, color=Condition))
```



Told you it will look empty and yet you can already 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 behind the scenes. This is because our initial call specified the most important part: how individual variables map on various properties even before we tell *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, and the axis's label. Points, lines, bars, 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 add later. 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 mappings also *group* 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.

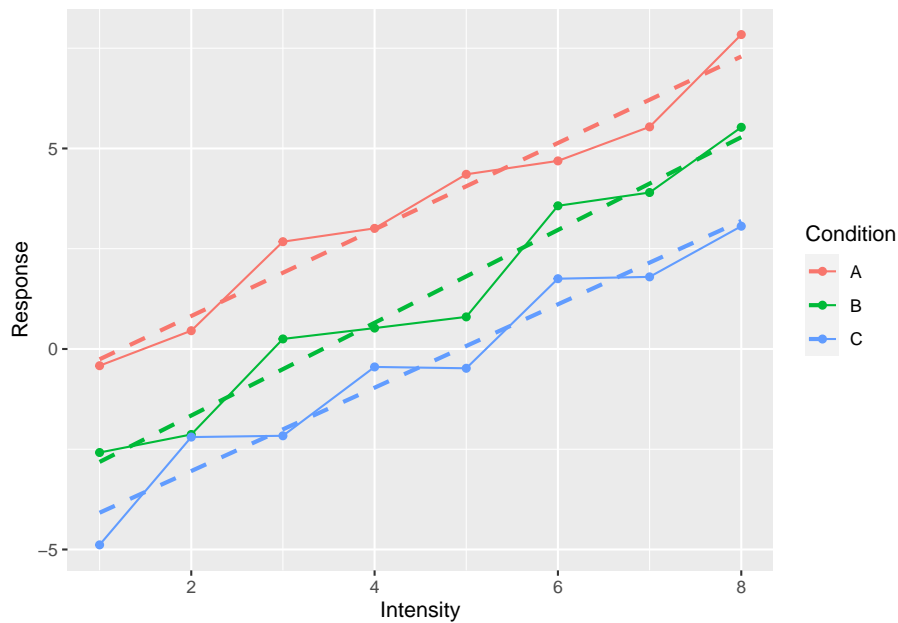
Let us see how you can keep the relationship mapping but add more visuals. Let us add both lines and points.

```
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, please, throw in some points as well”. And ggplot2 knows how to add the points so that they appear at proper location and in proper color. But we want more!

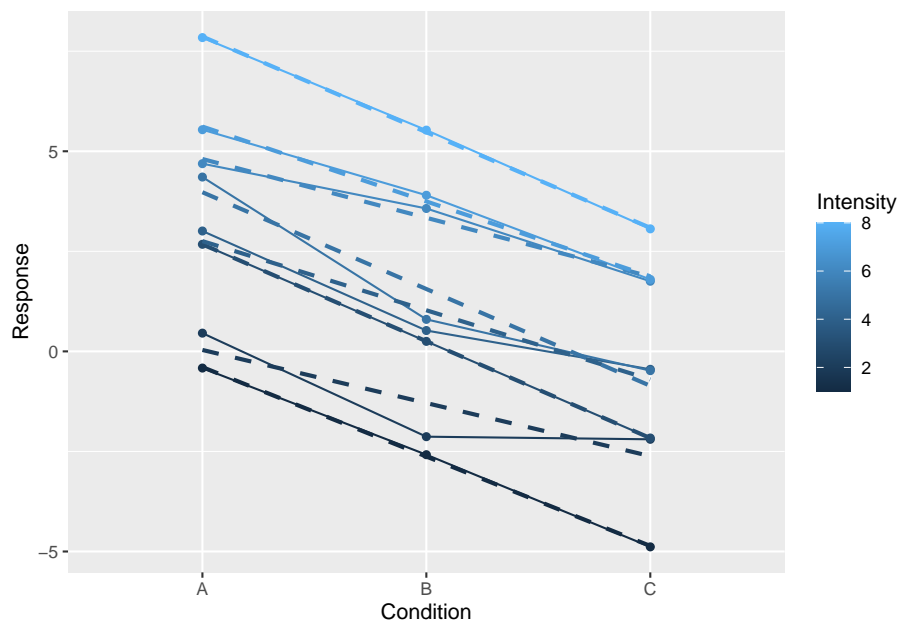
```
ggplot(data=simple_tidy_data, aes(x = Intensity, y = Response, color=Condition)) +
  geom_line() +
  geom_point() +
  # a linear regression over all dots in the group
  geom_smooth(method="lm", formula = y ~ x, se=FALSE, linetype="dashed")
```



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 linear fit from the solid data line).

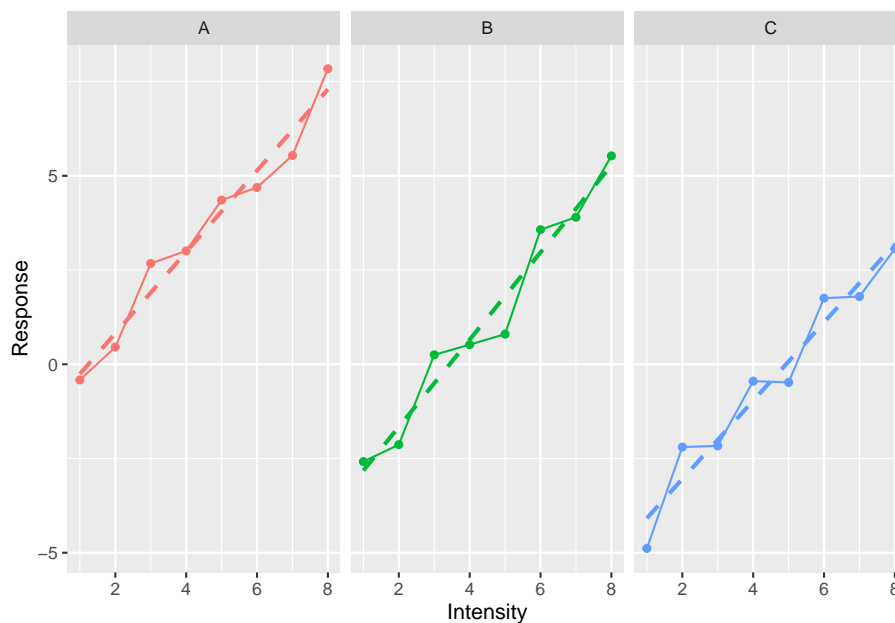
Or, we can keep the *visuals* but see whether changing *mapping* would make it more informative (we need to specify `group=Intensity` as continuous data is not grouped automatically).

```
ggplot(data=simple_tidy_data, aes(x = Condition, y = Response, color=Intensity, group=Intensity)) +
  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
  theme(legend.position = "none") # we don't need the legend as conditions are split between fac
```



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).

Now let's explore *ggplot2* by doing exercises. I recommend using ggplot2 reference page and cheatsheet when you are doing the exercises.

### 5.3 Auto efficiency: continuous x-axis

We start by visualizing how car efficiency, measured as miles-per-gallon, is affected by various factors such as production year, size of the engine, type of transmission, etc. The data is in the table `mpg`, which is part of the *ggplot2* package. 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

```
ggplot(data_table_name, aes(x = var1, y = var2, color = var3, shape = var4, ...)) +
  geom_primitive1() +
  geom_primitive2() +
```

...

Think about aesthetics, i.e., 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.

## 5.4 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 plots<sup>2</sup>. 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.

## 5.5 Mammals sleep: single variable

Now let's 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 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 hundreds 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 by default). 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.

---

<sup>2</sup>You can also use an excellent package called *ggbeeswarm* that distributes dots to show the distribution.



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<sup>3</sup>. 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.

## 5.6 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 `ggplot()` call) or in some visuals (e.g. only for points) to see the difference.

Do exercise 16.

## 5.7 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.

---

<sup>3</sup>Perhaps it is me, but I find this more confusing than helpful.

## 5.8 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.

## 5.9 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.

## 5.10 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 not 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.

## 5.11 Extending ggplot2

There are 128 (as of 05.10.2023) 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 I use particularly often 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.

## 5.12 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 and 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.



## Chapter 6

# Tidyverse: dplyr

Now that you understand vectors, tables, functions and pipes, and you know what our end goal is (a tidy table), we can start with data wrangling and Tidyverse way of doing it. All functions discussed below are part of `dplyr`<sup>1</sup> “grammar of data manipulation” package. Grab the exercise notebook!

### 6.1 Tidyverse philosophy

Data analysis is different from “normal” programming as it mostly involves a series of sequential operations on the same table. You might load the table, transform some variables, filter data, select smaller subset of columns, aggregate by summarizing across different groups of variables before plotting it or formally analyzing it via statistical tests. Tidyverse is built around this serial nature of data analysis of piping a table through a chain of functions. Accordingly, Tidyverse functions take a *table* (`data.frame` or `tibble`) as their *first* parameter, which makes piping simpler, and return a *modified table* as an output. This *table-in* → *table-out* consistency makes it easy to pipe these operations one after another. For me, it helps to think about Tidyverse functions as *verbs*: Actions that I perform on the table at each step.

Here is quick teaser of how such sequential piping works. Below, we will examine each verb/function separately and I will also show you how same operations can be carried out using base R. Note that I put each verb/function on a separate line. I don’t need to do this, but it makes it easier to understand how many different operations you perform (number of lines), how complex they are (how long individuals lines of code are), and makes them easy to read line-by-line. Note that even though we have many lines, it is fairly easy to follow the entire code sequence.

---

<sup>1</sup>The name should invoke an image of data pliers. According to Hadley Wickham, you can pronounce it any way you like.

```
# miles-per-gallon to kilometers-per-liter conversion factor
mpg2kpl <- 2.82481061

mpg |>
  # we filter the table by rows,
  # only keeping rows for which year is 2008
  filter(year == 2008) |>

  # we change cty and hwy columns by turning
  # miles/gallon into liters/kilometer
  mutate(cty_kpl = cty / mpg2kpl,
         hwy_kpl = hwy / mpg2kpl) |>

  # we create a new column by computing an
  # average efficiency as mean between city and highway cycles
  mutate(avg_kpl = (cty_kpl + hwy_kpl) / 2) |>

  # we convert kilometers-per-liter to liters for 100 KM
  mutate(avg_for_100 = 100 / avg_kpl) |>

  # we reduce the table to only two columns
  # class (of car) and avg_mpg
  select(class, avg_for_100) |>

  # we group by each class of car
  # and compute average efficiency for each group (class of car)
  group_by(class) |>
  summarise(class_avg = mean(avg_for_100), .groups = "keep") |>

  # we round the value to just one digit after the decimal point
  mutate(class_avg = round(class_avg, 1)) |>

  # we sort table rows to go from best to worst on efficiency
  arrange(class_avg) |>

  # we rename the class_avg_lpk to have a more meaningful name
  rename("Average liters per 100 KM" = class_avg) |>

  # we kable (Knit the TABLE) to make it look nicer in the document
  knitr::kable()
```

class	Average liters per 100 KM
compact	11.6
midsize	12.0
subcompact	12.6
2seater	14.0
minivan	15.3
suv	18.1
pickup	19.5

## 6.2 select() columns by name

Select verb allows you to select/pick columns in a table using their names. This is very similar to using columns names as indexes for tables that you have learned in seminar 3.

First, let us make a shorter version of `mpg` table by keeping only the first five rows. Note that you can also pick first N rows via `head()` function or `slide_head()` function from `dplyr` itself .

```
short_mpg <- mpg[1:5, ]

# same "first five rows" but via head() function
short_mpg <- head(mpg, 5)

# same "first five rows" but via dplyr::slice_head() function
short_mpg <- slice_head(mpg, n = 5)

knitr::kable(short_mpg)
```

manufacturer	model	displ	year	cyl	trans	drv	cty	hwy	fl	class
audi	a4	1.8	1999	4	auto(l5)	f	18	29	p	compact
audi	a4	1.8	1999	4	manual(m5)	f	21	29	p	compact
audi	a4	2.0	2008	4	manual(m6)	f	20	31	p	compact
audi	a4	2.0	2008	4	auto(av)	f	21	30	p	compact
audi	a4	2.8	1999	6	auto(l5)	f	16	26	p	compact

Here is how you can select only `model` and `cty` columns via preserving `[]` sub-setting

```
short_mpg[, c("model", "cty")]
```

model	cty
a4	18
a4	21
a4	20
a4	21
a4	16

And here is how it is done via `select()`.

```
short_mpg |>
  select(model, cty)
```

model	cty
a4	18
a4	21
a4	20
a4	21
a4	16

The idea of Tidyverse functions is to adopt to you, so you *can* use quotes or pass a vector of strings with column names. All calls below produce the same effect, so pick the style you prefer (mine, is in the code *above*) and stick to it<sup>2</sup>.

```
short_mpg |>
  select(c("model", "cty"))
```

```
short_mpg |>
  select("model", "cty")
```

```
short_mpg |>
  select(c(model, cty))
```

As you surely remember, you can use negation to select *other* indexes within a vector (`c(4, 5, 6)[-2]` gives you `[4, 6]`). For the single brackets `[]` this mechanism does not work with column *names* (only with their indexes). However, `select` has you covered, so we can select everything *but* `cty` and `model`

```
short_mpg |>
  select(-cty, -model)
```

manufacturer	displ	year	cyl	trans	drv	hwy	fl	class
audi	1.8	1999	4	auto(l5)	f	29	p	compact
audi	1.8	1999	4	manual(m5)	f	29	p	compact
audi	2.0	2008	4	manual(m6)	f	31	p	compact
audi	2.0	2008	4	auto(av)	f	30	p	compact
audi	2.8	1999	6	auto(l5)	f	26	p	compact

In the current version of `dplyr`, you can do the same negation via `!` (a **logical not** operator, you will meet later), moreover, it is now a recommended way of writing the selection<sup>3</sup>. The `-` and `!` are not synonyms and the difference is subtle but important, see below.

```
# This will NOT produce the same result as above
# Note that the model column is still in the table
```

<sup>2</sup>In general, bad but consistent styling is better than an inconsistent mix of good styles.

<sup>3</sup>At least, `-` is not mentioned anymore, even though it still works.



```
short_mpg |>
  select(!cty, !model)
```

manufacturer	model	displ	year	cyl	trans	drv	hwy	fl	class	cty
audi	a4	1.8	1999	4	auto(l5)	f	29	p	compact	18
audi	a4	1.8	1999	4	manual(m5)	f	29	p	compact	21
audi	a4	2.0	2008	4	manual(m6)	f	31	p	compact	20
audi	a4	2.0	2008	4	auto(av)	f	30	p	compact	21
audi	a4	2.8	1999	6	auto(l5)	f	26	p	compact	16

However, if you stick to putting all column names into a vector, as with the direct selection above, you can use negation with names as strings, you can negate a vector of names, etc. Again, it is mostly a matter of taste with consistency being more important than a specific choice you make.

*# will produce the same result as for "-"*

```
short_mpg |>
  select(!c("cty", "model"))
```

```
short_mpg |>
  select(!"cty", !"model")
```

```
short_mpg |>
  select(!c(cty, model))
```

Unlike vector indexing that forbids mixing positive and negative indexing, `select` does allow it. However, **do not use it**<sup>4</sup> because results can be fairly counter-intuitive and, on top of that, `-` and `!` work somewhat differently. Note the difference between `!` and `-`: In the former case only the `!model` part appears to have the effect, whereas in case of `-` only `cty` works.

```
short_mpg |>
  select(cty, !model)
```

cty	manufacturer	displ	year	cyl	trans	drv	hwy	fl	class
18	audi	1.8	1999	4	auto(l5)	f	29	p	compact
21	audi	1.8	1999	4	manual(m5)	f	29	p	compact
20	audi	2.0	2008	4	manual(m6)	f	31	p	compact
21	audi	2.0	2008	4	auto(av)	f	30	p	compact
16	audi	2.8	1999	6	auto(l5)	f	26	p	compact

```
short_mpg |>
  select(cty, -model)
```

<sup>4</sup>Unless you know what you are doing and that is the simplest and clearest way to achieve this.

cty
18
21
20
21
16

To make things even, worse `select(-model, cty)` work the same way as `select(cty, !model)` (sigh...)

```
short_mpg |>
  select(-model, cty)
```

manufacturer	displ	year	cyl	trans	drv	cty	hwy	fl	class
audi	1.8	1999	4	auto(l5)	f	18	29	p	compact
audi	1.8	1999	4	manual(m5)	f	21	29	p	compact
audi	2.0	2008	4	manual(m6)	f	20	31	p	compact
audi	2.0	2008	4	auto(av)	f	21	30	p	compact
audi	2.8	1999	6	auto(l5)	f	16	26	p	compact

So, bottom line, do not mix positive and negative indexing in `select`! I am showing you this only to signal the potential danger.

Do exercise 1.

Simple names and their negation will be sufficient for most of your projects. However, I would recommend taking a look at the official manual just to see that `select` offers a lot of flexibility (selecting range of columns, by column type, by partial name matching, etc), something that might be useful for you in your work.

## 6.3 Conditions

Before we can work with the next verb, you need to understand conditions. Conditions are statements about values that are either `TRUE` or `FALSE`. In the simplest case, you can check whether two values (one in a variable and one hard-coded) are equal via `==` operator

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

```
## [1] TRUE
print(x == 3)
```

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

For numeric values, you can use all usual comparison operators including *not equal* `!=`, *less than* `<`, *greater than* `>`, *less than or equal to* `<=` (note the order of symbols!), and *greater than or equal to* `>=` (again, note the order of symbols).

Do exercise 2.

You can negate a statement via *not* ! symbol as !TRUE is FALSE and vice versa. However, note that round brackets in the examples below! They are critical to express the *order* of computation. Anything *inside* the brackets is evaluated first. And if you have brackets inside the brackets, similar to nested functions, it is the innermost expression that get evaluated first. In the example below, x==5 is evaluated first and logical inversion happens only after it. In this particular example, you may not need them but I would suggest using them to ensure clarity.

```
x <- 5
print(!(x == 5))
```

```
## [1] FALSE
print(!(x == 3))
```

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

Do exercise 3.

You can also combine several conditions using *and* & and *or* | operators<sup>5</sup>. Again, note round brackets that explicitly define what is evaluated first.

```
x <- 5
y <- 2

# x is not equal to 5 OR y is equal to 1
print((x != 5) | (y == 1))
```

```
## [1] FALSE
# x less than 10 AND y is greater than or equal to 1
print((x < 10) & (y >= 1))
```

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

Do exercise 4.

All examples above used scalars but you remember that *everything is a vector*, including values that we used (they are just vectors of length one). Accordingly, same logic works for vectors of arbitrary length with comparisons working element-wise, so you get a vector of the same length with TRUE or FALSE values for each *pairwise* comparison.

Do exercise 5.

---

<sup>5</sup>There are also && and || operators that work only on scalars, i.e., on single values only.

## 6.4 Logical indexing

In the second seminar, you learned about vector indexing when you access *some* elements of a vector by specifying their index. There is an alternative way, called *logical indexing*. Here, you supply a vector of equal length with *logical values* and you get elements of the original vector whenever the logical value is TRUE

```
x <- 1:5
x[c(TRUE, TRUE, FALSE, TRUE, FALSE)]
```

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

This is particularly useful, if you are interested in elements that satisfy certain condition. For example, you want all *negative* values and you can use condition `x<5` that will produce a vector of logical values that, in turn, can be used as index

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

```
## [1] -2 -5 -1
```

You can have conditions of any complexity by combining them via *and* `&` and *or* `|` operators. For example, if you want number below -1 or above 3 (be careful to have space between `<` and `-`, otherwise it will be interpreted as assignment `<-`).

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

```
## [1] -2 5 -5
```

Do exercise 6.

Sometimes you may want to know the actual index of elements for *which* some condition is TRUE. Function `which()` does exactly that.

```
x <- c(-2, 5, 3, -5, -1)
which( (x < -1) | (x > 3) )
```

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

## 6.5 filter() rows by values

Now that you understand conditions and logical indexing, using `filter()` is very straightforward: You simply put condition that describes rows that you want to *retain* inside the `filter()` call. For example, we can look at efficiency only for two-seater cars.

```
mpg |>
  filter(class == "2seater")
```

manufacturer	model	displ	year	cyl	trans	drv	cty	hwy	fl	class
chevrolet	corvette	5.7	1999	8	manual(m6)	r	16	26	p	2seater
chevrolet	corvette	5.7	1999	8	auto(l4)	r	15	23	p	2seater
chevrolet	corvette	6.2	2008	8	manual(m6)	r	16	26	p	2seater
chevrolet	corvette	6.2	2008	8	auto(s6)	r	15	25	p	2seater
chevrolet	corvette	7.0	2008	8	manual(m6)	r	15	24	p	2seater

You can use information from any row, so we can look for midsize cars with four-wheel drive.

```
mpg |>
  filter(class == "midsize" & drv == "4")
```

manufacturer	model	displ	year	cyl	trans	drv	cty	hwy	fl	class
audi	a6 quattro	2.8	1999	6	auto(l5)	4	15	24	p	midsize
audi	a6 quattro	3.1	2008	6	auto(s6)	4	17	25	p	midsize
audi	a6 quattro	4.2	2008	8	auto(s6)	4	16	23	p	midsize

Do exercise 7.

Note that you can emulate `filter()` in a very straightforward way using single-brackets base R, the main difference is that you need to prefix every column with the table name, so `mpg[["class"]]` instead of just `class`<sup>6</sup>.

```
mpg[mpg[["class"]] == "midsize" & mpg[["drv"]] == "4", ]
```

manufacturer	model	displ	year	cyl	trans	drv	cty	hwy	fl	class
audi	a6 quattro	2.8	1999	6	auto(l5)	4	15	24	p	midsize
audi	a6 quattro	3.1	2008	6	auto(s6)	4	17	25	p	midsize
audi	a6 quattro	4.2	2008	8	auto(s6)	4	16	23	p	midsize

So why use `filter()` then? In isolation, as a single line computation, both options are equally compact and clear (apart from all the extra `table[["..."]]` in base R). But pipe-oriented nature of the `filter()` makes it more suitable for chains of computations, which is the main advantage of Tidyverse.

## 6.6 `arrange()` rows in a particular order

Sometimes you might need to sort your table so that rows go in a particular order<sup>7</sup>. In Tidyverse, you arrange rows based on values of specific variables. This verb is very straightforward, you simply list all variables, which must be used for sorting, in the order the sorting must be carried out. I.e., first the table is sorted based on values of the first variable. Then, for equal values of that

<sup>6</sup>You can sidestep this issue via `with()` function, although I am not a big fan of this approach.

<sup>7</sup>In my experience, this mostly happens when you need to print out or view a table.

variable, rows are sorted based on the second variable, etc. By default, rows are arranged in ascending order but you can reverse it by putting a variable inside of `desc()` function. Here is the `short_mpg` table arranged by city cycle highway efficiency (ascending order) and engine displacement (descending order, note the order of the last two rows).

```
short_mpg |>
  arrange(cty, desc(displ))
```

manufacturer	model	displ	year	cyl	trans	drv	cty	hwy	fl	class
audi	a4	2.8	1999	6	auto(l5)	f	16	26	p	compact
audi	a4	1.8	1999	4	auto(l5)	f	18	29	p	compact
audi	a4	2.0	2008	4	manual(m6)	f	20	31	p	compact
audi	a4	2.0	2008	4	auto(av)	f	21	30	p	compact
audi	a4	1.8	1999	4	manual(m5)	f	21	29	p	compact

Do exercise 8.

You can arrange a table using base R via `order()` function that gives index of ordered elements and can be used inside of preserving subsetting via single brackets `[]` notation. You can control for ascending/descending of a specific variable using `rev()` function that is applied *after* ordering, so `rev(order(...))`.

```
short_mpg[order(short_mpg[["cty"]], rev(short_mpg[["displ"]])), ]
```

manufacturer	model	displ	year	cyl	trans	drv	cty	hwy	fl	class
audi	a4	2.8	1999	6	auto(l5)	f	16	26	p	compact
audi	a4	1.8	1999	4	auto(l5)	f	18	29	p	compact
audi	a4	2.0	2008	4	manual(m6)	f	20	31	p	compact
audi	a4	2.0	2008	4	auto(av)	f	21	30	p	compact
audi	a4	1.8	1999	4	manual(m5)	f	21	29	p	compact

Do exercise 9.

## 6.7 mutate() columns

In Tidyverse, `mutate` function allows you to both add new columns/variables to a table and change the existing ones. In essence, it is equivalent to a simple column assignment statement in base R.

```
# base R
short_mpg[["avg_mpg"]] <- (short_mpg[["cty"]] + short_mpg[["hwy"]]) / 2

# Tidyverse equivalent
short_mpg <-
  short_mpg |>
  mutate(avg_mpg = (cty + hwy) / 2)
```

Note two critical differences. First, `mutate()` takes a table as an input and

returns a table as an output. This is why you start with a table, pipe it to `mutate`, and assign the results *back* to the original variable. If you have more verbs/lines, it is the output of the *last* computation that is assigned to the variable on the left-hand side of assignment<sup>8</sup>. Look at the listing below that indexes each line by when it is executed.

```
some_table <-      # 3. We assign the result to the original table, only once all the code below has
  some_table |>    # 1. We start here, with the original table and pipe to the next computation
  mutate(...)      # 2. We add/change columns inside of the table. The output is a table which we v
```

Second, you are performing a computation *inside* the call of the `mutate()` function, so `avg_mpg = (short_mpg$cty + short_mpg$hwy) / 2` is a *parameter* that you pass to it (yes, it does not look like one). This is why you use `=` rather than a normal assignment arrow `<-`. Unfortunately, you *can* use `<-` inside the `mutate` and the computation will work as intended but, for internal-processing reasons, the *entire statement*, rather than just the left-hand side, will be used as a column name. Thus, use `<-` *outside* and `=` *inside* of Tidyverse verbs.

```
short_mpg |>
  select(cty, hwy) |>
  mutate(avg_mpg = (cty + hwy) / 2,      # column name will be avg_mpg
         avg_mpg <- (cty + hwy) / 2) |> # column name will be `avg_mpg <- (short_mpg$cty + short
  knitr::kable()
```

cty	hwy	avg_mpg	avg_mpg <- (cty + hwy)/2
18	29	23.5	23.5
21	29	25.0	25.0
20	31	25.5	25.5
21	30	25.5	25.5
16	26	21.0	21.0

As shown in the example above, you can perform several computations within a single `mutate` call and they are executed one after another, just as they would be when using base R.

Do exercise 10.

Finally, `mutate` has two cousin-verbs called `transmute` and `add_column`. The former — `transmute` — works the same way but *discards* all original columns that were not modified. You probably won't use this verb all too often but I want you to be able to recognize it, as its name and function are very similar to `mutate` and the two are easy to confuse.

<sup>8</sup>R does have `->` statement, so, technically, you can pipe your computation and then assign it to a variable `table |> mutate() -> table`. However, this style is generally discouraged as starting with `table <- table |>` makes it clear that you modify and store the computation, whereas `table |>` signals that you pipe the results to an output: console, printed-out table, plot, etc.

```
short_mpg |>
  transmute(avg_mpg = (cty + hwy) / 2) |>
  knitr::kable(align = "c")
```

avg_mpg
23.5
25.0
25.5
25.5
21.0

The latter — `add_column` — is *similar* to `mutate` if you need to add a new column rather than to modify a new one. Its advantage is that it will produce an error, if you try to overwrite an existing column. Its disadvantage is that it does not appear to respect data grouping (see below), which can be very confusing. In short, stick to `mutate` unless you need either of these two functions specifically.

## 6.8 `summarize()` table

This verb is used when you *aggregate* across *all* rows, reducing them to a *single value*. Some examples of aggregating functions that are probably already familiar to you are mean, median, standard deviation, min/max. However, you can “aggregate” by taking a first or a last value or even by putting in a constant. Important is that you should assign a *single* value to the column when using `summarize`.

If you use `summarize` on an *ungrouped* table (these are the only tables we’ve been working on so far), it keeps only the computed columns, which makes you wonder “what’s the point?”

```
mpg |>
  summarise(avg_cty = mean(cty),
            avg_hwy = mean(hwy))
```

```
## # A tibble: 1 x 2
##   avg_cty avg_hwy
##   <dbl>   <dbl>
## 1    16.9    23.4
```

## 6.9 Work on individual groups of rows

The real power of `summarize` and of `mutate` becomes evident when they are applied to the data that is *grouped by* certain criteria. `group_by()` verb groups rows of the table based on values of variables you specified. Behind the scenes, this turns your single table into set of tables, so that your Tidyverse verbs are



applied to *each* table *separately*. This ability to parse your table into different groups of rows (all rows that belong to a particular participant, or participant and condition, or rows per block, or per trial), change that grouping on the fly, return back to the original full table, etc. makes analysis a breeze. Here is how we can compute average efficiency not across all cars (as in the code above) but for each car class separately.

```
mpg |>
  # there are seven different classes of cars, so group_by(cars) will
  # create seven hidden independent tables and all verbs below will be
  # applied to each table separately
  group_by(class) |>

  # same mean computation but per table and we've got seven of them
  summarise(avg_cty = mean(cty),
            avg_hwy = mean(hwy),
            .groups = "drop") |>
  knitr::kable()
```

class	avg_cty	avg_hwy
2seater	15.40000	24.80000
compact	20.12766	28.29787
midsize	18.75610	27.29268
minivan	15.81818	22.36364
pickup	13.00000	16.87879
subcompact	20.37143	28.14286
suv	13.50000	18.12903

Note that we compute a *single* value per table but because we do it for *seven* tables, we get *seven* rows in our resultant table. And `group_by` makes it easy to group data in any way you want. Are you interested in manufacturers instead car classes? Easy!

```
mpg |>
  group_by(manufacturer) |>
  # same mean computation but per table and we've got seven of them
  summarise(avg_cty = mean(cty),
            avg_hwy = mean(hwy),
            .groups = "drop") |>
  knitr::kable()
```

manufacturer	avg_cty	avg_hwy
audi	17.61111	26.44444
chevrolet	15.00000	21.89474
dodge	13.13514	17.94595
ford	14.00000	19.36000
honda	24.44444	32.55556
hyundai	18.64286	26.85714
jeep	13.50000	17.62500
land rover	11.50000	16.50000
lincoln	11.33333	17.00000
mercury	13.25000	18.00000
nissan	18.07692	24.61538
pontiac	17.00000	26.40000
subaru	19.28571	25.57143
toyota	18.52941	24.91176
volkswagen	20.92593	29.22222

How about efficiency per class *and* year? Still easy!

```
mpg |>
  group_by(class, year) |>
  # same mean computation but per table and we've got seven of them
  summarise(avg_cty = mean(cty),
            avg_hwy = mean(hwy),
            .groups = "drop") |>
  knitr::kable()
```

class	year	avg_cty	avg_hwy
2seater	1999	15.50000	24.50000
2seater	2008	15.33333	25.00000
compact	1999	19.76000	27.92000
compact	2008	20.54545	28.72727
midsize	1999	18.15000	26.50000
midsize	2008	19.33333	28.04762
minivan	1999	16.16667	22.50000
minivan	2008	15.40000	22.20000
pickup	1999	13.00000	16.81250
pickup	2008	13.00000	16.94118
subcompact	1999	21.57895	29.00000
subcompact	2008	18.93750	27.12500
suv	1999	13.37931	17.55172
suv	2008	13.60606	18.63636

The `.groups` parameter of the `summarize` function determines whether grouping you use should be dropped (`.groups = "drop"`, table become a single ungrouped table) or kept (`.groups = "keep"`). You will get a warning, if you do not specify `.groups` parameter, so it is a good idea to do this explicitly.

In general, use `.groups = "drop"` as it is better to later regroup table again then work on a grouped table without realizing it (leads to some weird looking output and a tricky to chase problem). You can also explicitly drop grouping via `ungroup()` verb.

Finally, there is a cousin verb `rowwise()` that groups by row. I.e., every row of the table becomes its own group, which could be useful if you need to apply a computation per row and the usual `mutate()` approach does not work (however, this is something of an advanced topic, so it is more about recognizing it than using it).

Do exercise 11.

You can replicate the functionality of `group_by + summarize` in base R via `aggregate()` and `group_by + mutate` via `by` functions. However, they are somewhat less straightforward in use as they rely on functional programming (which you haven't learned about yet) and require both grouping and summary function within a single call. Hence, we will skip on those.

## 6.10 Putting it all together

Now you have enough tools at your disposal to start programming a continuous analysis pipeline!

Do exercise 12.

## 6.11 Should I use Tidyverse?

As you saw above, whatever Tidyverse can do, base R can do as well. So why use a non-standard family of packages? If you are using each function in isolation, there is probably not much sense in this. Base R can do it equally well and *each individual* function is also compact and simple. However, if you need to *chain* your computation, which is almost always the case, Tidyverse's ability to pipe the entire sequence of functions in a simple consistent and, therefore, easy to understand way is a game-changer. In the long run, pick your style. Either go "all in" with Tidyverse (that is my approach), stick to base R, or find some alternative package family (e.g., `data.table`). However, as far as the book is concerned, it will be almost exclusively Tidyverse from now on.



## Chapter 7

# Working with Factors

Let us start with a “warm up” exercise that will require combining various things that you already learned. Download `persistence.csv` file (you can use the URL directly, but that will cause R to download it every time you call the function, so let’s not overstress the server) and put it into *data* subfolder in your seminar project folder. This is data from a Master thesis project by my student Kristina Burkel, published as an article in *Attention, Perception, & Psychophysics*. The work investigated how a change in object’s shape affected perceptual stability during brief interruptions (50 ms blank intervals). The research question was whether the results will match those for one other two history effects, which work at longer time scales. Such match would indicate that both history effects are likely to be produced by the same or shared neuronal representations of 3D rotation. Grab the exercise notebook before we start.

### 7.1 How to write code

From now on, you will need to implement progressively longer analysis sequences. Unfortunately, the longer and the more complex the analysis is, the easier it is to make a mistake that will ruin everything after that stage. And you will make mistakes, simply because no one is perfect and everyone makes them. I make them all the time. Professional programmers make them. So the skill of programming is not about writing the perfect code on your first attempt, it is writing your code in an iterative manner, so that any mistake you make (and, again, you will make them!) will be spotted and fixed immediately, before you continue adding more code. It should be like walking blind through uncertain terrain: One step a time, no running, no jumping, as you have no idea what awaits you.

What does this mean in practical terms? In a typical analysis (such as in the exercise below), you will need to do many things: read data, select columns,

filter it, compute new variables, group data and summarize it, plot it, etc. You might be tempted to program the whole thing in one go but it is a terrible idea. If your step #2 does not do what you think it should, your later stages will work with the wrong data and tracing it back to that step #2 may not be trivial (it almost never is). Instead, implement one step at a time and check that the results look as they should. E.g., in the exercise below, read the table. Check, does it look good, does it even have the data? Once you are sure that your reading bit works, proceed to columns selection. Run this two-step code and then check that it works and the table looks the way it should. It does (it has only the relevant columns)? Good, proceed to the next step.

**Never** skip these checks! Always look at the results of each additional step, do not just *hope* that they will be as they should. They might, they might not. In the latter case, if you are lucky, you will see that and are in for a long debugging session. But you may not even notice that computation is subtly broken and use its results to draw erroneous conclusions. It may feel overly slow to keep checking yourself continuously but it is a *faster* way to program in a long term. Moreover, if you do it once step at a time, you actually *know*, not hope, that it works.

I've spent three paragraphs on it (and now adding even the forth one!), because, in my opinion, this approach is the main difference between novice and experienced programmers (or, one could go even further and say between good and bad programmers). And I see this mistake of writing everything in one go repeated again and again irrespective of the tool people use (you can make a really fine mess using SPSS!). I makes a mess every time a deviate from this approach! So, pace yourself and let's start programming in earnest!

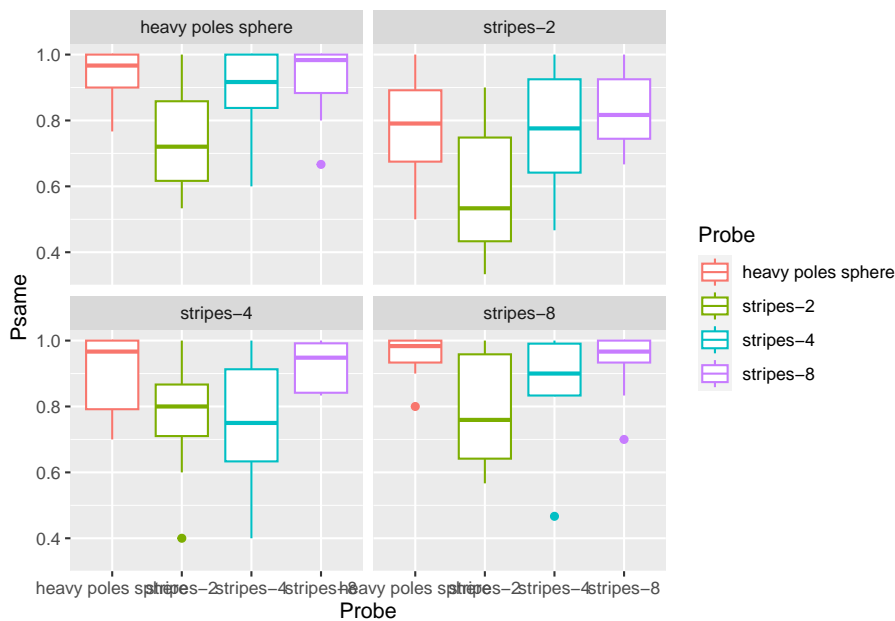
## 7.2 Implementing a typical analysis

In the first exercise, I want you to implement the actual analysis performed in the paper. Good news is that by now you know enough to program it! Note that steps 1-4 (everything but plotting) should be implemented as a *single pipeline*. Start with the first action and keep adding more verbs to it but you should end up with a single chained computation. This is actually more important than you might think: Putting read-and-preprocessing into a single pipe means no lurking temporary / intermediate variables that you may have changed in the mean time. A single pipe ensures that you are guaranteed to get the same table you run it. Splitting it into a few smaller chunks means you can (which means you will) run them out-of-order, run one computation an extra time, forget that you modified the data and you need to reload it, etc. I have been caught by this way more often than I would like to admit, so having a single definitive computation is always a good idea.

1. Load the data in a table. Name of the variable is up to you. Typically, I use names like `data`, `reports`, `results`, etc. Don't forget to specify

- columns' type.
2. Exclude `filename` column (it duplicates `Participant` and `Session` columns).
  3. Compute a new variable `SameResponse` which is `TRUE` when `Response1` and `Response2` match each other (in the experiment, that means that an object was rotating in the same direction before and after the intervention).
  4. For every combination of `Participant`, `Prime` and `Probe` compute proportion of same responses. You can do this in two ways. Recall that `as.integer(TRUE)` is 1 and `as.integer(FALSE)` is 0. Thus, you can either compute proportion as mean or compute the sum of same responses and divide it by total number of trials. Use function `n()` for the latter, it returns the total number of rows in the table or the group. Try doing it both ways.
  5. Plot the results with `Probe` variable on x-axis, proportion of same responses on y-axis, and use `Prime` to facet plots. Use box plots (or violin plots) to visualize the data. Try adding color, labels, etc. to make plots look nice.

Your final plot should look something like this



Do exercise 1.

When you examine the plot, you can see some sort of non-monotonic dependence with a dip for "stripes-2" and "stripes-4" objects. In reality, the dependence is monotonic, it is merely the order of values on the x-axis that is wrong. The

correct order, based on the area of an object covered with dots, is "heavy poles sphere", "stripes-8", "stripes-4", "stripes-2". Both `Prime` and `Probe` are *ordinal* variables called *factors* in R. Thus, to fix the order and to make object names a bit better looking, we must figure out how to work with factors in R.

## 7.3 Factors

Factors are categorical variables, thus variables that have a finite fixed and known set of possible values. They can be either *nominal* (cannot be ordered) or *ordinal* (have a specific order to them). An example of the former is the drive train (`drv`) variable in `mpg` table. There is a finite set of possible values ("f" for front-wheel drive, "r" for rear wheel drive, and "4" for a four-wheel drive) but ordering them makes no sense. An example of an ordinal variable is a Likert scale that has a finite set of possible responses (for example, "disagree", "neither agree, nor disagree", "agree") with a specific fixed order (participant's support for a statement is progressively stronger so that "disagree" < "neither agree, nor disagree" < "agree").

You can convert *any* variable to a factor using `factor()` or `as.factor()` functions. The latter is a more limited version of the former, so it makes little sense to ever use it. Below, I will only use `factor()`.

When you convert a variable (a vector) to factor, R:

1. figures out all unique values in this vector
2. sorts them in an *ascending* order
3. assigns each value an integer index, a.k.a. "level"
4. uses the actual value as a "label".

Here is an example of this sequence: there four levels sorted alphabetically (note that R prints out not only the vector but also its levels).

```
letters <- c("C", "A", "D", "B", "A", "B")
letters_as_factor <- factor(letters)
letters_as_factor
```

```
## [1] C A D B A B
## Levels: A B C D
```

You can extract levels of a factor variable by using the function of the same name

```
levels(letters_as_factor)
```

```
## [1] "A" "B" "C" "D"
```

You can specify the order of levels either during the `factor()` call or later using `forcats` library (more on that later). For example, if we want to have levels in



the reverse order we specify it via `levels` parameter. Note the opposite order of levels.

```
letters <- c("C", "A", "D", "B", "A", "B")
letters_as_factor <- factor(letters, levels = c("D", "C", "B", "A"))
letters_as_factor
```

```
## [1] C A D B A B
## Levels: D C B A
```

We can also specify `labels` of individual labels instead of using values themselves. Note that the labels must match levels in *number* and *order*.

```
responses <- c(1, 3, 2, 2, 1, 3)
responses_as_factor <- factor(responses, levels = c(1, 2, 3), labels = c("negative", "neutral", "positive"))
responses_as_factor
```

```
## [1] negative positive neutral neutral negative positive
## Levels: negative neutral positive
```

You can see *indexes* that were assigned to each level by converting `letters_as_factor` to a numeric vector. In this case, R throws away labels and returns indexes.

```
as.numeric(letters_as_factor)
```

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

However, be careful when level labels are numbers. In the example below, you might think that `as.numeric(tens)` should give you `[20, 40, 30]`<sup>1</sup> but these are labels, not levels that go from 1 to 3! If you need to convert labels to numbers, you have to do it in two steps `as.numeric(as.character(tens))`: `as.character()` turns factors to strings (using labels) and `as.numeric()` converts those labels to numbers (if that conversion can work).

```
tens <- factor(c(20, 40, 30))
print(tens)
```

```
## [1] 20 40 30
## Levels: 20 30 40
```

```
print(as.numeric(tens))
```

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

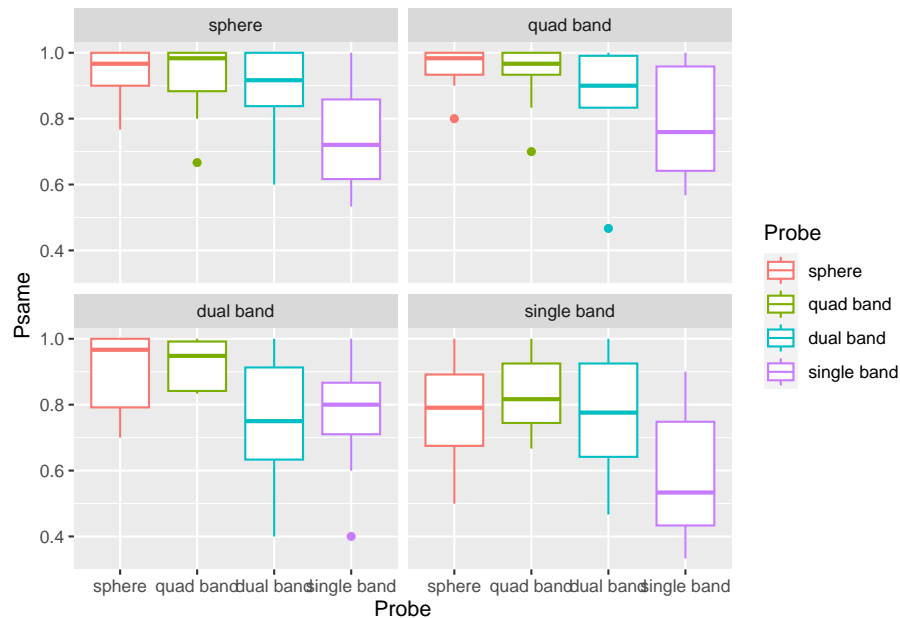
```
print(as.numeric(as.character(tens)))
```

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

For the next exercise, copy-paste the code from exercise #1 and alter it so the labels are "sphere" (for "heavy poles sphere"), "quad band"

<sup>1</sup>At least I tend to always think that.

(for "stripes-8"), "dual band" ("stripes-4"), "single band" (for "stripes-2") and levels are in that order. Your plot should look something like this.



Do exercise 2.

## 7.4 Forcats

Tidyverse has a package `forcats`<sup>2</sup> that makes working with factors easier. For example, it allows to reorder levels either by hand or automatically based on the order of appearance, frequency, value of other variable, etc. It also gives you flexible tools to changes labels either by hand, by lumping some levels together, by anonymising them, dropping unused levels, etc. In my work, I mostly use reordering (`fct_relevel()`) and renaming (`fct_recode()`) of factors by hand. You will need to use these two functions in exercise #3. However, if you find yourself working with factors, it is a good idea to check other `forcats` functions to see whether they can make your life easier.

To reorder factor by hand, you simply state the desired order of factors, similar to they way you specify this via `levels=` parameters in `factor()` function. However, in `fct_relevel()` you can move only *some* factors and others are “pushed to the back”.

<sup>2</sup>The package’s name is anagram of *factors*.

```
letters <- c("C", "A", "D", "B", "A", "B")
letters_as_factor <- factor(letters, levels = c("B", "C", "D", "A"))
print(letters_as_factor)
```

```
## [1] C A D B A B
## Levels: B C D A
```

```
# specifying order for ALL levels
```

```
letters_as_factor <- fct_relevel(letters_as_factor, "D", "C", "B", "A")
print(letters_as_factor)
```

```
## [1] C A D B A B
## Levels: D C B A
```

```
# specifying order for just ONE level, the rest are "pushed back"
```

```
# "A" should now be the first level and the rest are pushed back in their original order
```

```
letters_as_factor <- fct_relevel(letters_as_factor, "A")
print(letters_as_factor)
```

```
## [1] C A D B A B
## Levels: A D C B
```

You can also put a level at the very back, as second level, etc. `fct_relevel()` is very flexible, so check reference whenever you use it.

To rename individual levels you use `fct_recode()` by providing `new = old` pairs of values.

```
letters_as_factor <- factor(c("C", "A", "D", "B", "A", "B"))
letters_as_factor <- fct_recode(letters_as_factor, "_A_" = "A", "_C_" = "C")
print(letters_as_factor)
```

```
## [1] _C_ _A_ D B _A_ B
## Levels: _A_ B _C_ D
```

Note that this allows you to merge levels by hand.

```
letters_as_factor <- factor(c("C", "A", "D", "B", "A", "B"))
letters_as_factor <- fct_recode(letters_as_factor, "_AC_" = "A", "_AC_" = "C")
print(letters_as_factor)
```

```
## [1] _AC_ _AC_ D B _AC_ B
## Levels: _AC_ B D
```

For exercise #3, redo exercise #2 but using `fct_relevel()` and `fct_recode()`. You still need to use `factor()` function to convert `Prime` and `Probe` to factor but do not specify levels and labels. Use `fct_relevel()` and `fct_recode()` inside `mutate()` verbs to reorder and relabel factor values (or, first relabel and then reorder, whatever is more intuitive for you). The end product (the plot) should be the same.

Do exercise 3.

## 7.5 Plotting group averages

Let us keep practicing and extend our analysis to compute and plots averages for each condition (**Prime**×**Probe**) over all participants. Use preprocessing code from exercise #3 but, once you compute a proportion per **Participant**×**Prime**×**Probe**, you need to group data over **Prime**×**Probe** to compute average performance across observers. Advice, *do not* reuse the name of the column, e.g., if you used **Psame** for proportion per **Participant**×**Prime**×**Probe**, use some *other* name for **Prime**×**Probe** (e.g. **Pavg**). Otherwise, it may turn out to be very confusing (at least, this is a mistake a make routinely). Take a look at the code below, what will the **Range** values be?

```
tibble(ID = c("A", "A", "B", "B"),
       Response = c(1, 2, 4, 6)) |>

  group_by(ID) |>
  summarise(Response = mean(Response),
            Range = max(Response) - min(Response))
```

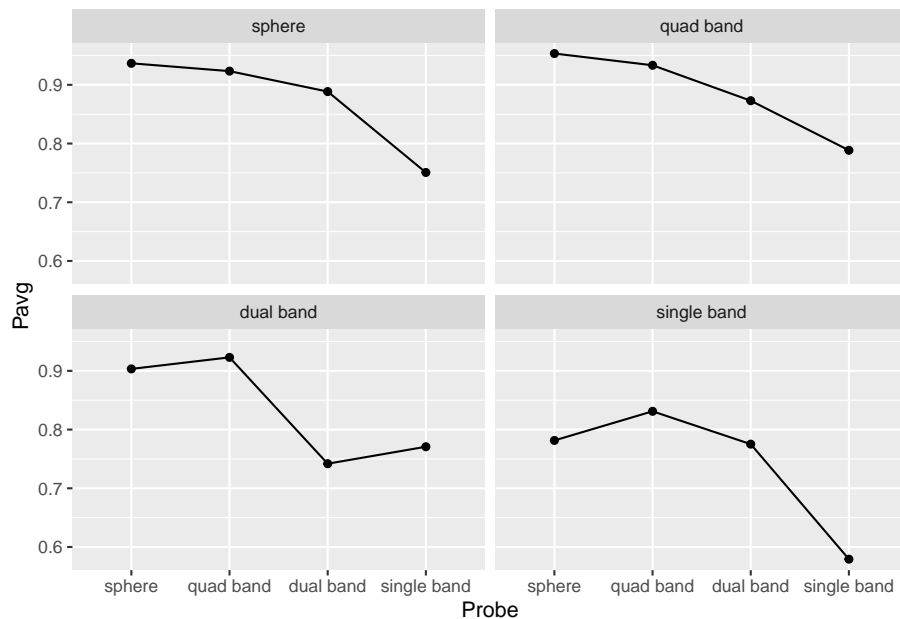
I routinely assume that they should be 1 for "A" (because 2-1) and 2 for "B" (6-4). Nope, both are 0 because by the time **Range = max(Response) - min(Response)** is executed, original values of **Response** are overwritten by **Response = mean(Response)**, so it has just **one** value, the mean. And **min()** and **max()** of a single value is that value, so their difference is 0. It is obvious once you carefully consider the code but it is *not* obvious (at least to me) straightaway. In short, be **very careful** when you are reusing column names. Better still, do not reuse them, be creative, come up with new ones!

Getting back to the exercise, compute average performance per **Prime**×**Probe**. Store the result of the computation in a new variable (I've called it **persistence\_avg**) and check that results makes sense, e.g. you have just three columns **Prime**, **Probe**, and **Pavg** (or however you decided to name the column). They should look like this:

Prime	Probe	Pavg
sphere	sphere	0.9366667
sphere	quad band	0.9233333
sphere	dual band	0.8885185
sphere	single band	0.7507407
quad band	sphere	0.9533333
quad band	quad band	0.9333333
quad band	dual band	0.8729630
quad band	single band	0.7885185
dual band	sphere	0.9033333
dual band	quad band	0.9229630
dual band	dual band	0.7418519
dual band	single band	0.7707407
single band	sphere	0.7814815
single band	quad band	0.8311111
single band	dual band	0.7751852
single band	single band	0.5792593

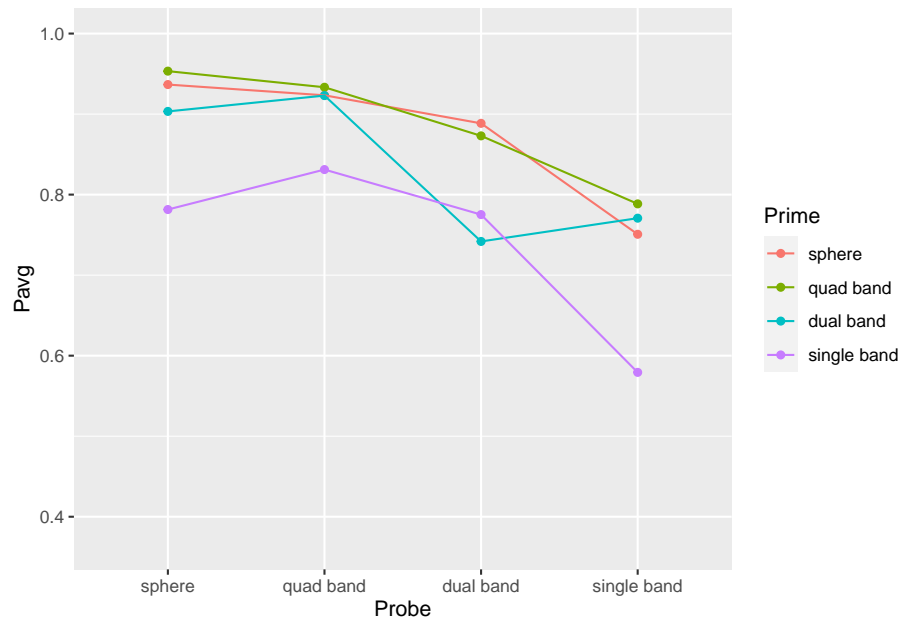
Do exercise 4.

Then, plot the results. Use `geom_point()` plus `geom_line()` to plot the mean response. The plot should look like this (hint, drop color mapping and map `Prime` to `group` property).



Do exercise 5.

Tweak code from exercise 4 to plot all lines on the same plot and use color property to distinguish between different primes.



Do exercise 6.

## 7.6 Plotting our confidence in group averages via quantiles

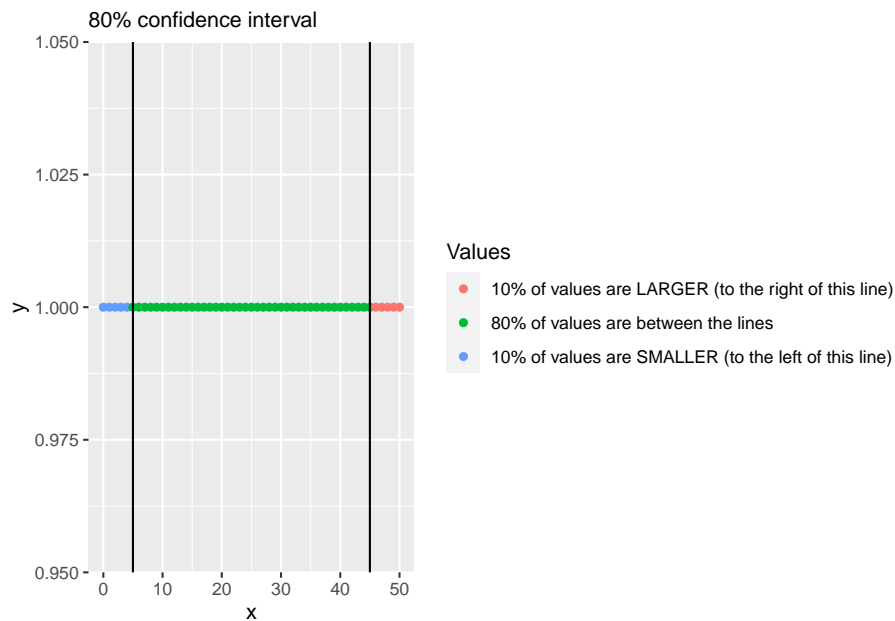
From the plots above, you get a sense that identities of the probe and prime (objects before and after the interruption) matter. Single band appears to be the poorest prime (its line is lowest) and probe (its dots are lower than the rest). Conversely, sphere is an excellent prime (line at the top) and probe (dots are very high). However, averages that we plotted is just a point estimate for most likely effect strength but they alone cannot tell us whether differences in objects' shape do matter. For this, you need to perform statistical analysis but to get at least a feeling of how confident can you be about these differences, you need to plot a measure of variability associated with that *statistics*. I.e., [1, 5, 9] and [4, 5, 6] both have identical mean of 5 but their standard deviation is 4.5 *times* different (4.51 vs. 1). In the second case, the true mean is likely to be somewhere very near to 5, whereas we would have much less confidence in the former one.

One way to characterize the mean is by computing its standard error. However, it is best used when actual data is distributed normally or, at least, symmetrically around the mean, i.e., the distance from an observation to the mean could

be the same irrespective of whether it is larger or smaller. This is a luxury you can expect only for variables that live on  $\pm\infty$  range (support) or if the practically observed range of values is very far from either the floor or the ceiling. Adult height is an example of the latter: You cannot have height below 0 but an average adult height is far enough from that limit so its distribution is normal and symmetric enough. Unfortunately, a lot of data that we collect in psychology or social science research does not fit this description: Binomial data with yes/no or correct/incorrect responses lives on 0..1 range, response times have long right tail because they cannot be negative or even particularly short (200 ms would be a realistic floor for key presses, ~120 ms for eye saccadic response under *very* specific experimental conditions.) End I did not mention Likert scale data because it is an ordered categorical type data, so you cannot use raw data to compute even the mean, let alone its error, so I will show you how to visualize it later.

In our case the outcome variable is a *proportion* limited to 0 to 1 range. From practical point of view this means that our measure of variability is unlikely to be symmetric relative to the mean (with a unique exception of the mean exactly 0.5). I.e., think about a group average  $P_{avg} = 0.8$ , points *below* that average can be further away from the mean (up to 0.8) than points *above* it (0.2 away at most). This compression is called either a ceiling (when you get squashed by the upper range) or flooring (when you cannot go below certain value) effect. Thus, we need a measure that would take this asymmetry into account. Later on you will learn how to do it using bootstrapping but we will start with a simpler approach of just using quantiles of a distribution to understand its variability.

To compute this quantiles-based interval, you need to compute its lower and upper limits separately via quantiles. A quantile for 0.1 (10%) returns a value, so that 10% of all values in the vector are below it, the quantile of 0.9 (90%) means that only 10% of values are above it (or 90% are below). So, an 80% confidence intervals includes values that are between 10% and 90% or, alternatively, between 0.1 and 0.9 quantiles.



To compute this, R has function `quantile()`.

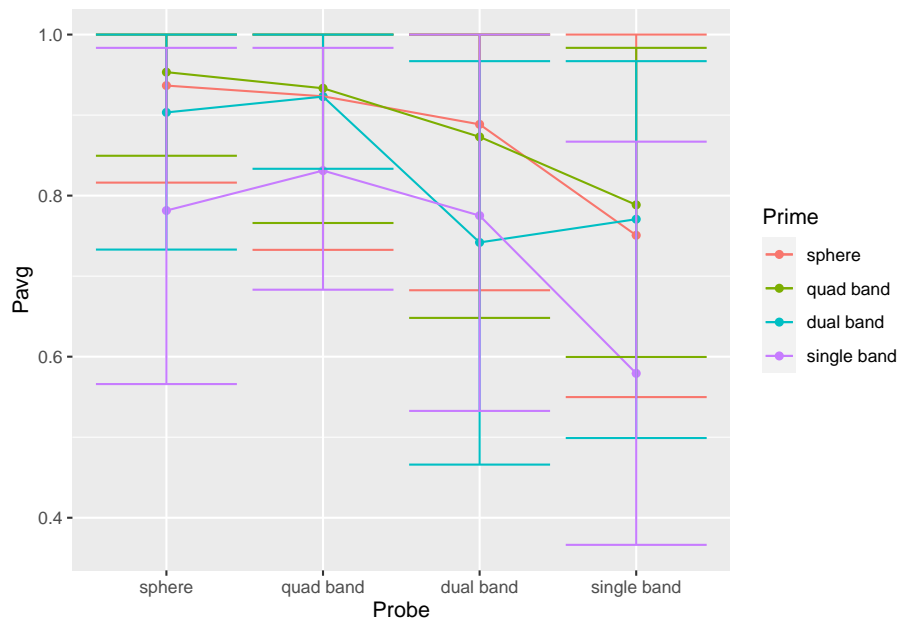
```
x <- 0:50
quantile(x, 0.1)
```

```
## 10%
##    5
```

Modify code from exercise #5 to compute two additional variables/columns for lower and upper limits of the 89%<sup>3</sup> interval (think about what these limits are for 89% interval). Then, use `geom_errorbar()` to plot 89% interval (you will need to map the two variable you computed to `ymin` and `ymax` properties). The plot should look like this (hint, drop color mapping and map `Prime` to `group` property).

<sup>3</sup>Why 89%? Because it is a prime number! If you think that it sounds arbitrary, you are perfectly correct. But so is using 95% and for that one you do not even have the “prime number” excuse!





Do exercise 7.

## 7.7 Dealing with Likert scales

Likert scales are one of the most popular ways to measure responses in psychology. And, at least this is my opinion of the literature, they tend to be misanalyzed and misreported (IMHO). It is quite common to report an average and a standard error of a response across all participants, but these numbers cannot be actually computed (because these are ordered categorical values, not numbers) and even if you trick the computer in doing so (by pretending that levels are actual numbers) these numbers are ill-suited to characterize the response, as you will see below. The proper analysis of Likert scale data requires a use of the “Item Response Theory” and although it is very straightforward, it is outside of the scope of this book. Instead, we will look at how you can visualize the responses in the meaningful way, although the complete story will have to wait until we learn about bootstrapping.

Below, we will use data collected using Intrinsic Motivation Inventory (IMI), where participants needed to respond on a 7-point Likert scale indicating indicate how true a statement (such as “I enjoyed doing this activity very much” or “I was pretty skilled at this activity.”) is for them:

1. Not at all true
2. Hardly true
3. Slightly true

4. Somewhat true
5. Mostly true
6. Almost completely true
7. Very true

Before we visualize the data, we need to read and preprocess it. Perform the following using a single pipe:

- Read the file `likert-scale.csv` specifying column types
- Convert *Condition* column to factor where 1 is “game” and 2 is “experiment”
- Convert *Response* column to factor using levels described above.

Here is how the first five rows of the table look for me:

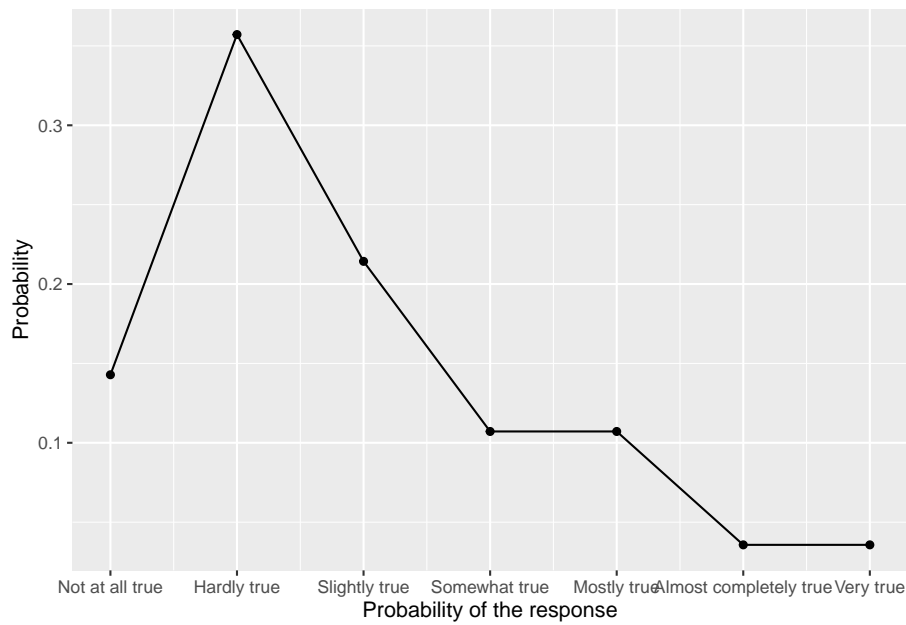
Participant	Condition	Response
TMM1990w	game	Slightly true
VEH1985w	experiment	Slightly true
CEN2000w	experiment	Hardly true
YWH1992w	game	Mostly true
IDK1985w	experiment	Hardly true

Do exercise 8.

Hopefully, using actual labels instead of original number makes it clear that computing mean and standard error is impossible (what is an average of “Slightly true” and “Hardly true”?). Instead, we need to count the number of response per response level, convert that to a proportion of responses, and plot them to see the distribution. You know almost everything you need to perform this. Here is the logic:

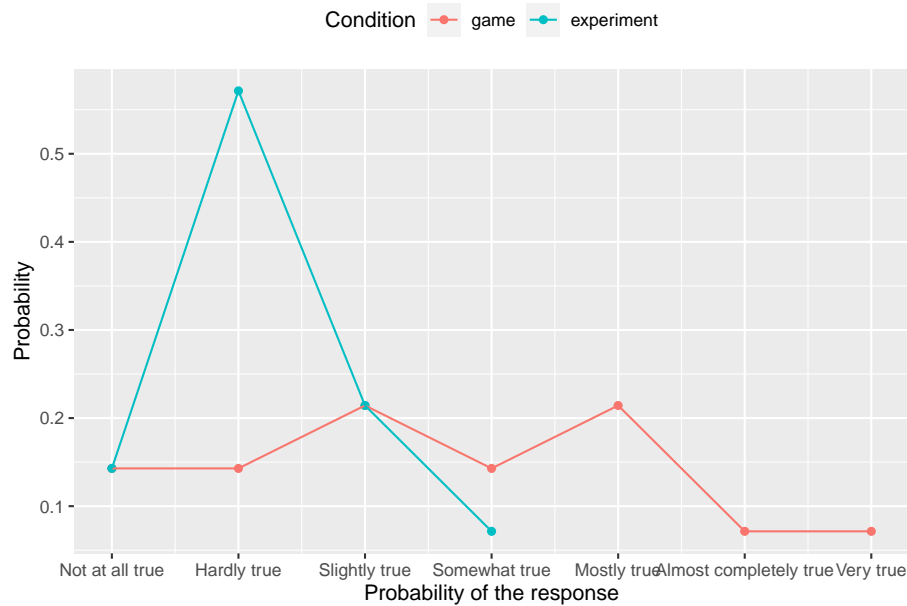
- group data per response level
- summarize the data by computing number of entries per group, don’t forget to ungroup table. You can also combine the two steps to count responses per level.
- compute proportion per response by dividing number of responses for each level by the total number of responses.

Once you computed the aggregates, plot them using `geom_point()` and `geom_line()`. The only catch here, is that `geom_line()` cannot work with discrete x-axis, so when specifying the aesthetics for x-axis, you need to use `as.integer(Response)` instead of just `Response` and you will need to use `scale_x_continuous()` to specify breaks (1 to 7) and labels (levels of our Likert scale). The end results should look as follows:



Do exercise 9.

The study in question used to experimental conditions, so it would be informative to see whether this produce difference in responses. The logic of computation is (almost) the same: You need to group by both response level and experimental condition when counting responses but by condition alone when computing the proportion (why?). When visualizing, you only need to add color aesthetics to distinguish between the groups. The end results should be as follows:



Do exercise 10.

Note that we are missing information on uncertainty about proportion of response (i.e., error bars), but I will show you how to compute them once we get to bootstrapping. Also note that even if you would convert ordinal levels to real numbers, using mean and standard error would still be a bad idea. Note how squashed the distribution is to the lower side (particularly, when averaging across groups). Mean is a nice descriptor for symmetric distributions as then it coincides with both mode (most probable, highest point) and median (point that splits distribution half/half). But for skewed distributions, all three statistics — mode, mean, and median — will be at different locations and you should either report all or think about better ways to present the data. Same goes for a standard error which is a nice descriptor for a normally distributed data, less so for symmetric data (you'll be missing information about exact shape and probably thinking about a normal one), not very good for skewed data. Here, data does not lie equally on both sides, so a symmetric standard error will overestimate variance on the left side (there is not much space to vary on the left), while simultaneously underestimating it on the right side. In short, always look at the data before compacting it to various statistic, even if such practice is common (this does not make it a good practice).

## Chapter 8

# Tidying your data: joins and pivots

It is fun to work with tidy complete data. Unfortunately, more often than not you will need to preprocess and tidy it up before you can analyze it. In addition to dplyr, Tidyverse has tidyr package that helps you with some of the problems. Grab the exercise notebook and let's get started.

### 8.1 Joining tables

Sometimes, results of a study are stored in separate tables. For example, demographic information can be stored separately from responses, as the former is the same for all trials and conditions, so it makes little sense to duplicate it. For the actual analysis, you may need to add this information, merging or, in SQL/Tidyverse-speak, joining two tables (I'll use term “join” from hereon).

Examples make joining operations much easier to understand, so let us create two tables.

```
demographics <- tibble(ID = c("A", "B", "C", "D"),
                       Age = c(20, 19, 30, 22))
reports <-
  tibble(ID = c("A", "B", "C", "D", "A", "B", "C", "D")) |>
  mutate(Report = round(runif(length(ID), 1, 7)), 2) |>
  arrange(ID)
```

To join two tables, you require “key” columns: columns that contain values that will be used to identify matching rows in the two tables. You can use multiple and differently named columns for that purpose but we will start with a simplest case: a single column with same name “ID”. A join function (`inner_join()`)

Table 8.1: demographics (left) and report (right) tables

ID	Age	ID	Report	2
A	20	A	5	2
B	19	A	3	2
C	30	B	6	2
D	22	B	5	2
		C	4	2
		C	7	2
		D	5	2
		D	1	2

Table 8.2: Joining/merging tables with fully matching keys via join (left) and merge (right). Note how a different order of tables results in a different order of columns.

ID	Age	Report	2	ID	Report	2	Age
A	20	5	2	A	5	2	20
A	20	3	2	A	3	2	20
B	19	6	2	B	6	2	19
B	19	5	2	B	5	2	19
C	30	4	2	C	4	2	30
C	30	7	2	C	7	2	30
D	22	5	2	D	5	2	22
D	22	1	2	D	1	2	22

here, see below for details on different joins) takes a first row in `demographics` table that has "A" in ID column and will look for all rows in table `reports` that has `ID == "A"`. Then, it will do the same for all remaining rows for `demographics` table, one row at a time. It takes three parameters: table `x` (first table), table `y` (second table), and `by` - a vector with key column names argument<sup>1</sup>. A call to `merge()` function is very similar and produces identical results. Note how column order has changed for `merge()`, because I used a different order of tables, but the content itself is the same.

```
via_join <- inner_join(demographics, reports, by = "ID")
via_merge <- merge(reports, demographics, by = "ID")
```

Things are easy when every key in the first table has its counterpart in the second one and vice versa. Things get tricky, if that is not the case. Which is why there are *four* different ways to join tables (note that they will all produce

<sup>1</sup>Theoretically, you can skip it and *dplyr* will do its best to find matching columns. However, this is a dangerous thing to leave out, as your intuition and *dplyr*'s matching rules may lead to different results. I strongly recommend to always specify key columns.

Table 8.3: Now demographics (left) and reports (right) table have unique keys ("D" for demographics, "E" for reports) without a match in the second table.

ID	Age	ID	Report
A	20	A	1.779618
B	19	A	4.881113
C	30	B	3.071987
D	22	B	3.094037
		C	4.848295
		C	5.465657
		E	5.239866
		E	3.980308

identical results for fully matching set of keys, as in the example above):

- **inner** join: uses only key values that are present in both tables.
- **full** join: uses all key values from both tables, if matching keys are missing in one of the tables, rows are filled with NA.
- **left** join: uses only key values that are present in the left (first) table, if matching keys are missing in the *right* table, , rows are filled with NA.
- **right** join: mirror twin of the left join, uses only key values that are present in the right (second) table, if matching keys are missing in the *left* table, , rows are filled with NA.

To see each join in action, let us slightly modify the **reports** table to include ID "E" instead of "D". Now, the "D" is missing the second table but "E" is missing in the **demographics**:

```
reports <-
  tibble(ID = c("A", "B", "C", "E", "A", "B", "C", "E")) |>
  mutate(Report = runif(length(ID), 1, 7)) |>
  arrange(ID)
```

**Inner join** is most conservative and excludes any non-matching keys, note that rows with *both* ID == "D" and ID == "E" are missing. This is the *default* behavior for the merge function.

```
inner_tidy <- inner_join(reports, demographics, by = "ID")
inner_base <- merge(reports, demographics, by = "ID")
```

In contrast, **full join** is the most liberal as it includes all rows from both tables, filling in missing values with NA (e.g., see Report for ID == "D" and Age for ID == "E"). In base R merge() function, you turn an inner join into a full one using all = TRUE.

```
full_tidy <- full_join(demographics, reports, by = "ID")
full_base <- merge(demographics, reports, by = "ID", all = TRUE)
```

Table 8.4: Inner join. Only rows for matching keys are merged

ID	Report	Age
A	1.779618	20
A	4.881113	20
B	3.071987	19
B	3.094037	19
C	4.848295	30
C	5.465657	30

Table 8.5: Full join. All rows are merged, ‘NA’ are used for missing values in rows from a complementary table

ID	Age	Report
A	20	1.779618
A	20	4.881113
B	19	3.071987
B	19	3.094037
C	30	4.848295
C	30	5.465657
D	22	NA
E	NA	5.239866
E	NA	3.980308



Table 8.6: Left join. All rows from demographics tables are used and missing matching rows are filled with ‘NA’

ID	Age	Report
A	20	1.779618
A	20	4.881113
B	19	3.071987
B	19	3.094037
C	30	4.848295
C	30	5.465657
D	22	NA

Table 8.7: Right join. All rows from reports tables are used and missing matching rows are filled with ‘NA’

ID	Age	Report
A	20	1.779618
A	20	4.881113
B	19	3.071987
B	19	3.094037
C	30	4.848295
C	30	5.465657
E	NA	5.239866
E	NA	3.980308

**Left join** uses only rows from the left (first) table, dropping extra rows from the second one. Note NA in Report column for ID == "D" and no rows for ID == "E". To do a left join via `merge()`, you need to specify `all.x = TRUE`.

```
left_tidy <- left_join(demographics, reports, by = "ID")
left_base <- merge(demographics, reports, by = "ID", all.x = TRUE)
```

**Right join** is a mirror twin of the left join, so now rows for ID == "D" are missing and there are missing values for ID == "E". You include `all.y = TRUE` for a right join via `merge()`.

```
right_tidy <- right_join(demographics, reports, by = "ID")
right_base <- merge(demographics, reports, by = "ID", all.y = TRUE)
```

As noted above, you can also use more than one key.

```
demographics <- tibble(ID = c("A", "B", "A", "B"),
  Gender = c("M", "F", "F", "M"),
  Age = c(20, 19, 30, 22))
reports <- tibble(ID = c("A", "B", "A", "B"),
```

Table 8.8: Two identically named key columns: ID and Gender.

ID	Gender	Age	ID	Gender	Report
A	M	20	A	M	4.565688
B	F	19	B	F	5.629244
A	F	30	A	F	5.254527
B	M	22	B	M	3.571175

Table 8.9: Joining/merging two tables by ID and Gender.

ID	Gender	Age	Report
A	M	20	4.565688
B	F	19	5.629244
A	F	30	5.254527
B	M	22	3.571175

```
Gender = c("M", "F", "F", "M"),
Report = runif(length(ID), 1, 7))
```

```
inner_multi_tidy <- inner_join(demographics, reports, by = c("ID", "Gender"))
inner_multi_base <- merge(demographics, reports, by = c("ID", "Gender"))
```

Finally, key columns can be named *differently* in two tables. In this case, you need to “match” them explicitly. For dplyr joins, you use a named vector to match pairs of individual columns. For `merge()`, you supply two vectors: one for columns in the first table (parameter `by.x`) and one columns in the the second one (parameter `by.y`). Here, you need to be careful and check that columns order matches in both parameters.

```
demographics <- tibble(VPCode = c("A", "B", "A", "B"),
  Sex = c("M", "F", "F", "M"),
  Age = c(20, 19, 30, 22))
```

```
reports <- tibble(ID = c("A", "B", "A", "B"),
  Gender = c("M", "F", "F", "M"),
  Report = runif(length(ID), 1, 7))
```

```
inner_diff_tidy <- inner_join(demographics, reports, by=c("VPCode"="ID", "Sex"="Gender"))
inner_diff_base <- merge(demographics, reports, by.x=c("VPCode", "Sex"), by.y=c("ID", "Gender"))
```

As you saw from examples above, dplyr joins and `merge()` produce identical results. However, I would recommend to use the former, simply because function names make it explicit which kind of join you perform (something you can figure out only by checking additional parameters of `merge()`).

Table 8.10: Differently named key columns. VPCode in demographics table corresponds to ID in reports; sex in demographics corresponds to Gender in reports

VPCode	Sex	Age	ID	Gender	Report
A	M	20	A	M	2.255894
B	F	19	B	F	3.460513
A	F	30	A	F	6.447326
B	M	22	B	M	5.604981

Table 8.11: Joining tables by matching VPCode to ID and Sex to Gender.

VPCode	Sex	Age	Report
A	F	30	6.447326
A	M	20	2.255894
B	F	19	3.460513
B	M	22	5.604981

Download files IM.csv and GP.csv that you will need for exercise #1. These are participants responses on two questionnaires with each participant identified by their ID (**Participant** in *IM.csv* and **Respondent** in *GP.csv*), **Condition** (which experimental group they belong to), and their **Gender**. Read both tables and join them so that there are no missing values in the table (some participants are missing in *GP.csv*, so there are *three* joins that can do this, which one will you pick?). Then, turn **Condition** and **Gender** into factors, so that for **Condition** levels are "control" (2) and "game" (1) and for **Gender** levels are "female" (1) and "male" (2). Your final table should look as follows (I've dropped most of the columns for IM.csv, so they would fit to the page):

Participant	Condition	Gender	IM01_01	IM01_02	IM01_03	GP01_01	GP02_01	GP02_02	GP02_03
wilKv	game	female	3	2	5	1	1	1	
9QQjf	control	female	3	4	4	1	4	4	
qcp03	control	female	2	3	6	1	1	1	
eV1RY	control	female	2	5	6	1	1	5	
JymeM	control	female	2	1	7	1	1	1	
hxrts	game	female	1	3	3	1	1	3	

Do exercise 1.

Repeat the same exercise but use `merge()`.

Do exercise 2.

Now let us practice joining and simulating data as well. Create two tables that need to be joined by a single key column. In the first table, Use `rnorm()` function to generate normally distributed data with mean of 175 and standard deviation

Table 8.12: Wide non-tidy table.

Participant	Face	Symmetry	Attractiveness	Trustworthiness
1	M-1	6	4	3
1	M-2	4	7	6
2	M-1	5	2	1
2	M-2	3	7	2

of 7.6 for column **Height** (what range would you expect to cover 95% of the data?). In the second table, use same normal distribution but with mean of 80 and standard deviation of 10 for column **Weight**. When fill in key column for both tables, do it so that *inner* and *right* join would give the same final table but *left* and *full* would give you a longer one (test this explicitly!). After joining two tables, plot **Height** against **Weight** and superimpose linear regression fit. Are two columns correlated? Should they be given how we generated the data?

Do exercise 3.

## 8.2 Pivoting

Recall the idea of tidy data:

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

And, also recall, that quite often data is stored in a wide format that is easier for humans read.

Here, **Symmetry**, **Attractiveness**, **Trustworthiness** are different face properties participants responded on, whereas values are **Response** they gave. You can work with a table like that but it is often more convenient to have a column **Scale** that will code which face property participants respond on and a column **Response** to hold values. Then, you can easily split or group your data by property while performing the same analysis on all of them.

You can do pivoting using base R `reshape()` function. But pivoting is a fairly confusing business, so Tidyverse alone offers three different solutions starting with `reshape2` package<sup>2</sup>, and continuing with original `gather()` and `spread()` functions in `tidyr` to modern `pivot_longer()` and `pivot_wider()` functions in the same package.

---

<sup>2</sup>It uses `melt` and `cast` functions for this.

## 8.3 Pivot longer

`pivot_longer()` takes a table, which you can pipe to the function Tidyverse-style, and a vector of column names that need to be transformed. All *column names* go to one new column and all the *values* go to another new column. Defaults names of these two columns are, respectively, "name" and "value" but you can specify something more suitable via, respectively, `names_to` and `values_to` parameters.

There are many more bells-and-whistles (name and value transformations, removing a prefix via regular expressions, etc.), so I recommend looking at the manual and a vignette. However, in most cases these four parameters will be all you need, so let us see `pivot_longer` in action.

I assume that table presented above is in `widish_df` table defined above. The columns that we want to transform are `Symmetry`, `Attractiveness`, `Trustworthiness`. Thus, the simplest call with all defaults is

```
long_tidy <- tidyr::pivot_longer(widish_df,
                                cols=c("Symmetry", "Attractiveness", "Trustworthiness"))
```

Participant	Face	name	value
1	M-1	Symmetry	6
1	M-1	Attractiveness	4
1	M-1	Trustworthiness	3
1	M-2	Symmetry	4
1	M-2	Attractiveness	7
1	M-2	Trustworthiness	6
2	M-1	Symmetry	5
2	M-1	Attractiveness	2
2	M-1	Trustworthiness	1
2	M-2	Symmetry	3
2	M-2	Attractiveness	7
2	M-2	Trustworthiness	2

When you compare the two tables, you will see that original three columns  $\times$  four rows are now stretched into twelve rows and name-value pairs are consistent across the two tables<sup>3</sup>. As noted above, we can improve on that by specifying proper names for new columns.

```
long_tidy <- tidyr::pivot_longer(widish_df,
                                cols=c("Symmetry", "Attractiveness", "Trustworthiness"),
                                names_to = "Scale",
                                values_to = "Response")
```

<sup>3</sup>By the way, this simple check may seem as a trivial point but this is a kind of simple sanity check that you should perform routinely. This way you *know* rather than *hope* that transformation did what it should. I also check value is a few rows to make sure that I didn't mess things up. Catching simple errors early saves you a lot of time!

Participant	Face	Scale	Response
1	M-1	Symmetry	6
1	M-1	Attractiveness	4
1	M-1	Trustworthiness	3
1	M-2	Symmetry	4
1	M-2	Attractiveness	7
1	M-2	Trustworthiness	6
2	M-1	Symmetry	5
2	M-1	Attractiveness	2
2	M-1	Trustworthiness	1
2	M-2	Symmetry	3
2	M-2	Attractiveness	7
2	M-2	Trustworthiness	2

If you want to stick to base R, you can pivot longer via `reshape()` function that can do both pivot to longer and wider tables. It is more flexible and, therefore, much more confusing (at least for me). Here are some parameters that we need to specify in order to emulate `pivot_longer()` call above:

- **direction**: Either "long" (here), or "wide".
- **idvar**: names of variables that identify multiple records in the long format. In contrast, `pivot_longer()`, assumes that all columns that you did not transform are identity columns.
- **varying**: names of columns that will be turned into a single variable that contains only **values** in a new long table. Corresponds to `cols` argument in `pivot_longer()`
- **v.names**: name of the column with values. Corresponds to `values_to` parameter of `pivot_longer()`.
- **timevar**: sort of corresponds to `names_to` parameter of `pivot_longer()`, so it is the name of the column where *indexes* or *labels* of transformed columns will go.
- **times**: by default, `reshape()` does not put column names into `timevar` column but uses their relative *indexes* instead. E.g., `Symmetry` column will get index of 1, `Attractiveness` will get 2, `Trustworthiness` will be 3. You can replace these indexes with *any* labels. Below, I used the same labels as column names but I could have used *any* three values for this.

```
long_base <- reshape(widish_df,
  direction = "long",
  idvar = c("Participant", "Face"),
  varying = c("Symmetry", "Attractiveness", "Trustworthiness"),
  v.names="Response",
  times = c("Symmetry", "Attractiveness", "Trustworthiness"),
  timevar = "Scale")
```

For using `reshape()`, I strongly suggest experimenting with its parameters to get a feeling for how it should be (and should not be!) used. As you can see, it is

Table 8.13: Same table pivoted via reshape.

Participant	Face	Scale	Response
1	M-1	Symmetry	6
1	M-2	Symmetry	4
2	M-1	Symmetry	5
2	M-2	Symmetry	3
1	M-1	Attractiveness	4
1	M-2	Attractiveness	7
2	M-1	Attractiveness	2
2	M-2	Attractiveness	7
1	M-1	Trustworthiness	3
1	M-2	Trustworthiness	6
2	M-1	Trustworthiness	1
2	M-2	Trustworthiness	2

more involved and uses fewer defaults than `pivot_longer()`, so you need to make sure you understand it.

## 8.4 Practice using Likert scale data

Let us put this new knowledge to practice, using GP.csv file. This is a questionnaire on gaming habits, which was conducted prior to an experiment to check whether two groups of participants assigned to *Game* and *Experiment* conditions have similar gaming habits. We would like to visually inspect responses to individual items in a questionnaire appear for different conditions, as this will tell us whether we should expect a difference. For this, we will reuse our approach for summarizing and plotting ordinal responses we did during the last seminar. Split the computations below into two pipelines. One that loads and pre-processes the data (steps 1-4). Another one that produces a summary and stores it into a different table (steps 5-6, see previous seminar if you forgot how we did it). Advice, implement it one step at a time, checking the table and making sure that you get expected results before piping it and adding the next operation.

1. Read the file, make sure you specify column types.
2. Convert **Condition** column to a factor with “game” (1) and “control” (2).
3. Pivot all GP.. columns. You should get a table with five columns: **Respondent**, **Condition**, **Gender**, **Item** (column where original column names go), and **Response** (a column where original go). Hint, you can use slicing : to specify the range of columns or `starts_with()` function to specify a common prefix. Try both approaches.
4. Convert **Response** column to a factor assuming a seven-point scale, the levels are

5. Not at all
6. Slightly
7. Somewhat
8. Moderately
9. Quite a bit
10. Considerably
11. Very much
12. Count responses for each item and condition.
13. Convert counts to proportion of responses for each item and condition.

Your first table, in long format, should look like this (I show only first four rows)

Respondent	Condition	Gender	Item	Response
NV5UJ	game	1	GP01_01	Not at all
NV5UJ	game	1	GP02_01	Not at all
NV5UJ	game	1	GP02_02	Not at all
NV5UJ	game	1	GP02_03	Somewhat

And your second table, with aggregated results, should be like this

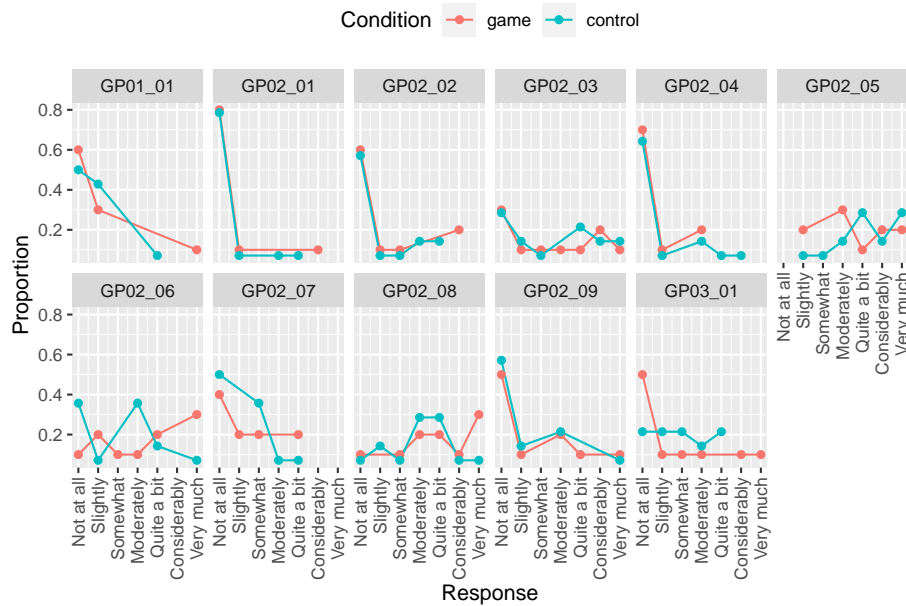
Condition	Item	Response	n	Proportion
game	GP01_01	Not at all	6	0.6
game	GP01_01	Slightly	3	0.3
game	GP01_01	Very much	1	0.1
game	GP02_01	Not at all	8	0.8

Do exercise 4.

Repeat the exercise but now using base R reshape() function. ::: {.infobox  
 .practice} Do exercise 5. :::

Now you have a table with proportion of response for each condition and item, let's plot them to compare visually. Use **Response** for x-axis, **Proportion** for y-axis, **Condition** for color, think about how facet the plot.





Do exercise 6.

Note that our computation and plot have a problem: whenever a response level was not used by any participant, it is missing both in the table and in the plot. However, we should have an entry for every response level but with 0 for both count and proportion, 0 counts for a response level is not a missing or absent response level. A way to solve this problem is by using function `complete()` that adds missing combinations of the data based on a subset of columns you specified and fills in values for the remaining columns either with `NA` or with values that you specified. Here is an example for a table that contains columns `Class` and `Count`. Note that I defined `Class` as a factor with five levels (“A” till “E”) and the latter is missing from the table. `Complete` adds the row and fill the remaining column with ‘NA’.

```
class_df <-
  tibble(Class = c("A", "B", "C", "D"),
    Count = c(20, 19, 30, 22)) |>
  mutate(Class = factor(Class, levels = c("A", "B", "C", "D", "E")))

class_df |>
  complete(Class)
```

```
## # A tibble: 5 x 2
##   Class Count
##   <fct> <dbl>
## 1 A      20
```

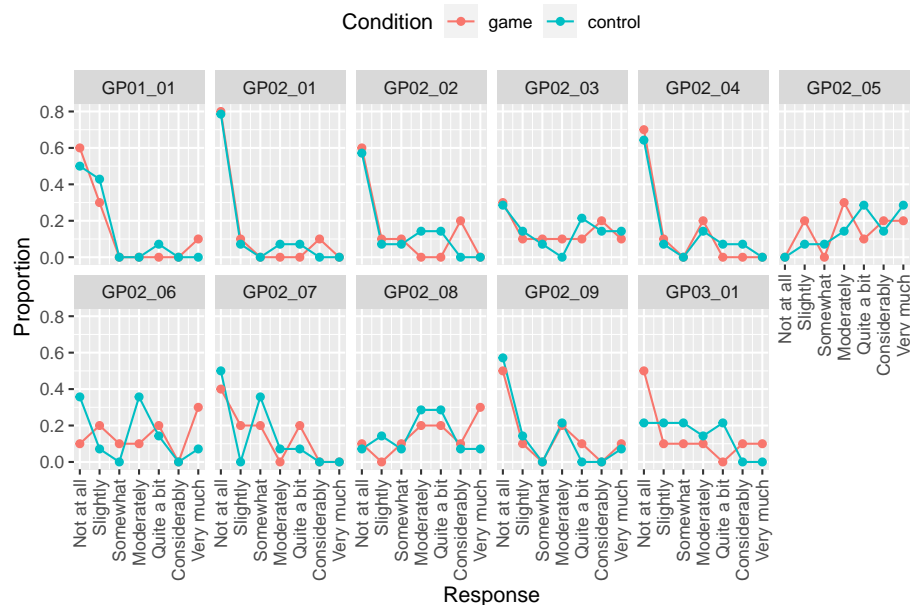
```
## 2 B      19
## 3 C      30
## 4 D      22
## 5 E      NA
```

Although in some cases, filling with NA may be reasonable, here, we want to fill in the count and it should be 0 (the fact that a particular class is missing means that we never counted it). Note that `fill` parameter takes a named **list** not a named vector (Why is that? If about data types of different columns and if you are still not sure why, reread on difference between atomic vectors and lists).

```
class_df |>
  complete(Class, fill = list(Count = 0))
```

```
## # A tibble: 5 x 2
##   Class Count
##   <fct> <dbl>
## 1 A      20
## 2 B      19
## 3 C      30
## 4 D      22
## 5 E       0
```

Here is how the plot look once we completed the table



Do exercise 7.

Table 8.14: Our table is wide again!

Participant	Face	Symmetry	Attractiveness	Trustworthiness
1	M-1	6	4	3
1	M-2	4	7	6
2	M-1	5	2	1
2	M-2	3	7	2

## 8.5 Pivot wider

You can also always go from a long to a wide representation via `pivot_wider()` or `reshape` functions. The logic is reverse, you need to specify which columns *identify* different rows that belong together, which columns contain column names, and which contain their values. For our example table on face ratings, the names of the columns are in the column **Scale** and values are in **Response**. But what about columns that identify the rows that belong together? In our case, these are **Participant** and **Face**, so all rows from a *long* table that have same combination of **Participant** and **Face** values should be merged together into a single row. If you do not explicitly specify `id_cols`, then by default, `pivot_wider()` will use *all other remaining columns* to identify which rows belong together. This is irrelevant in this toy example, as **Participant** and **Face** is all we have left anyhow but below I will show you how things can get confusing and how to overcome this.

Let us undo our previous wide-to-long transformation and get our original wide table back!<sup>4</sup>

```
wide_again_tidy <-
  long_tidy |>
  pivot_wider(names_from = "Scale", values_from="Response")
```

Or, using explicit `id_cols`

```
wide_again_tidy <-
  long_tidy |>
  pivot_wider(id_cols = c("Participant", "Face"), names_from = "Scale", values_from="Response")
```

You can pivot wider using `reshape()` as well. However, note that, as of 07.10.2023, it works correctly *only* with data frames, so if you have a tibble (as I do), you need to convert it to a data frame via `data.frame()` or `as.data.frame()` (drop the `as.data.frame()` in the example above to see what I mean). Otherwise, you need to specify

- `direction = "wide"`

<sup>4</sup>I used `table` as an explicit first argument for `pivot_longer()` but piped it to `pivot_wider()`, why? To remind you that these two ways are the interchangeable and that both put the table as a parameter into the function.

Table 8.15: Our table is wide again but via reshape()!

	Participant	Face	Response_Symmetry	Response_Attractiveness	Response_Trustworthiness
1	1	M-1	6	4	
4	1	M-2	4	7	
7	2	M-1	5	2	
10	2	M-2	3	7	

- **idvar** : different rows that belong together. Same as **id\_cols** for `pivot_wider()` but no defaults here.
- **timevar** : same as **names\_from** for `pivot_wider()`, column with values that will be used as column names.
- **v.names** : same as **values\_from**.
- **sep**: the new column names will be constructed as **v.names + sep + timevar**. By default **sep**=".".

The main difference, as compared to `pivot_wider()`, is how the column names are constructed. With `reshape()` function, the **v.names + sep + timevar** rule means that you end up with column names such as `Response.Symmetry` instead of just `Symmetry`.

```
wide_again_base <- reshape(as.data.frame(long_tidy),
  direction = "wide",
  idvar = c("Participant", "Face"),
  timevar = "Scale",
  v.names = "Response",
  sep = "_")
```

Let us take a look at the importance of **id\_cols**. Imagine that we have *another* column, say, response times. So, our long table will look like this

```
long_tidy_rt <-
  long_tidy |>
  ungroup() |>
  mutate(RT = round(rgamma(n(), 4, 3), 2))
```

Participant	Face	Scale	Response	RT
1	M-1	Symmetry	6	1.38
1	M-1	Attractiveness	4	0.90
1	M-1	Trustworthiness	3	0.57
1	M-2	Symmetry	4	1.45
1	M-2	Attractiveness	7	1.81
1	M-2	Trustworthiness	6	2.64
2	M-1	Symmetry	5	0.39
2	M-1	Attractiveness	2	0.72
2	M-1	Trustworthiness	1	0.78
2	M-2	Symmetry	3	0.67
2	M-2	Attractiveness	7	1.57
2	M-2	Trustworthiness	2	0.64

For `pivot_wider`, if we do not specify which columns identify rows that belong together, RT will be used as well. But, because it is different for every response, each row in the original table will be unique and we will end up with a weird looking table with lots of NAs.

```
wide_odd_rt <-
  pivot_wider(long_tidy_rt, names_from = "Scale", values_from="Response")
```

Participant	Face	RT	Symmetry	Attractiveness	Trustworthiness
1	M-1	1.38	6	NA	NA
1	M-1	0.90	NA	4	NA
1	M-1	0.57	NA	NA	3
1	M-2	1.45	4	NA	NA
1	M-2	1.81	NA	7	NA
1	M-2	2.64	NA	NA	6
2	M-1	0.39	5	NA	NA
2	M-1	0.72	NA	2	NA
2	M-1	0.78	NA	NA	1
2	M-2	0.67	3	NA	NA
2	M-2	1.57	NA	7	NA
2	M-2	0.64	NA	NA	2

To remedy that, we need to specify id columns explicitly, so that `pivot_wider()` can ignore and *drop* the rest:

```
wide_rt <-
  pivot_wider(long_tidy_rt,
    id_cols = c("Participant", "Face"),
    names_from = "Scale",
    values_from="Response")
```

Participant	Face	Symmetry	Attractiveness	Trustworthiness
1	M-1	6	4	3
1	M-2	4	7	6
2	M-1	5	2	1
2	M-2	3	7	2

For practice, let us take adaptation data and turn it onto a wide format that is easier for humans to read. In the original form, the table is a long format with a row for each pair of prime and probe stimuli.

Participant	Prime	Probe	Nsame	Ntotal
LUH-1992-M	Sphere	Sphere	22	119
LUH-1992-M	Sphere	Quadro	23	118
LUH-1992-M	Sphere	Dual	15	120
LUH-1992-M	Sphere	Single	31	115

Let us turn it into a wider table, so that a single row corresponds to a single prime and four new columns contain proportion of same responses for individual probes. The table should look like this (use `round()` function to reduce the number of digits):

Participant	Prime	Sphere	Quadro	Dual	Single	Average
LUH-1992-M	Sphere	0.18	0.19	0.12	0.27	0.19
LUH-1992-M	Quadro	0.21	0.22	0.14	0.34	0.23
LUH-1992-M	Dual	0.25	0.30	0.27	0.48	0.32
LUH-1992-M	Single	0.34	0.30	0.48	0.39	0.38

The overall procedure is fairly straightforward (should be a single pipe!):

1. Read the file (don't forget to specify column types!)
2. Compute `Psame` proportion of same responses given number of total responses for each `Probe`.
3. Pivot the table wider, think about your id columns. Also try this without specifying any and see what you get.
4. Compute an average stability across all probes and put it into a new `Average` column.
5. Pipe it to the output, using `knitr::kable()`.

Use `pivot_wider()` in exercise 8.

Do exercise 8.

Repeat the analysis but now using the `reshape()` function.

Do exercise 9.

Let us practice more and create group average summary as a square 5×4 table with a single row per *Prime* and four columns for *Probe* plus a column that says which prime the row corresponds to. As a value for each cell, we want to code a *median* value. The table should look like this:

Prime	Sphere	Quadro	Dual	Single
Sphere	0.13	0.13	0.17	0.32
Quadro	0.19	0.12	0.21	0.38
Dual	0.15	0.30	0.27	0.48
Single	0.34	0.30	0.48	0.51

You know almost everything you need, so think about how you would implement this as a *single* pipeline. Hints: to match my table you will definitely to convert **Prime** and **Probe** to factors to ensure consistent ordering (otherwise, they will be sorted alphabetically), you will need to group individual combinations of prime and probe before computing a summary statistics. And, of course, you will need to pivot the table wider (use your preferred method).

Do exercise 10.





## Chapter 9

# Controlling computation flow

Grab the exercise notebook before we start.

One of the most powerful features of R is that it is vector-based. Remember, everything is a vector (or a list). In the previous seminars you saw you can apply a function, a filter, or perform a computation on all values of a vector or all rows in a table in a single call. However, sometimes, you need to go over one value or row at a time explicitly. For example, if you working with a time-series, it might be easier to use an explicit for loop to compute current value based on previous state of the system. However, such instances are fairly rare, so the general rule for R is “you probably do not need a loop”. Below, we will go through various tools that render explicit loops redundant, but cover the loops themselves as well as conditional control statements (if-else) and functions (`ifelse`, `case_when`).

Note that the material does not cover a `while () {<do-code>}` loop mostly because it is rarely used in data analysis<sup>1</sup> and I could not come up with examples that would not be very artificial looking. At the same time, if you do need it, it has a very simple structure, so understanding and using it should not be a challenge.

### 9.1 `rep()`

The most basic repetition mechanism in R is `rep()` function. It takes a vector and repeats it specified number of `times`.

---

<sup>1</sup>I tried to remember when was the last time I personally used it in R and could not remember, if I ever did this.

```
rep(c(1, 2, 3), times = 4)
```

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

Alternatively, you can repeat each element specified number of times before repeating the next one via `each` parameter. The difference between options lies only in the *order* of elements in the new vector. As you can see both vectors have the same length and each individual value is repeated four times.

```
rep(c(1, 2, 3), each = 4)
```

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

You can specify length of the output vector via `length.out`. When combined with `times` it can be useful for producing truncated vectors. E.g., when we repeat a three element vector but we want to get ten values. Using `times` only, we can get either nine (`times = 3`) or twelve (`times = 4`), not ten. `length.out = 10` makes it happen.

```
rep(c(1, 2, 3), times=4, length.out = 10)
```

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

However, you can also use subsetting of a new vector to achieve the same end.

```
rep(c(1, 2, 3), times = 4)[1:10]
```

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

You should be more careful when combining `length.out` with `each`, as each value is repeated `each` times and, if `length.out` is longer, the same sequence is repeated again. Could be confusing and you might get a very unbalanced repeated sequence.

```
rep(c(1, 2, 3), each = 8, length.out = 10)
```

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

Do exercise 1.

## 9.2 Repeating combinations

To create a table with all combinations of values, you can use either base R `expand.grid()` or `tidyr`'s implementation `expand_grid()`. The latter is a bit more robust and can expand even tables and matrices (but see the documentation for subtle differences in implementation and output).

The usage is very straightforward, you provide column names and values and you get all combinations of their values.

```
knitr::kable(grid_base)
```

gender	handidness	colorblindness
female	right	TRUE
male	right	TRUE
female	left	TRUE
male	left	TRUE
female	right	FALSE
male	right	FALSE
female	left	FALSE
male	left	FALSE

`expand_grid()` works the same they but for the order of values within columns.

```
expand_grid(gender=c("female", "male"),
            handidness=c("right", "left"),
            colorblindness=c(TRUE, FALSE))
```

```
knitr::kable(grid_tidy)
```

gender	handidness	colorblindness
female	right	TRUE
female	right	FALSE
female	left	TRUE
female	left	FALSE
male	right	TRUE
male	right	FALSE
male	left	TRUE
male	left	FALSE

Do exercise 2.

## 9.3 For loop

You can loop (iterate) over elements of a vector or list via a for loop, which is very similar to for-loops in other programming languages. However, use of the `for` loop in R is fairly rare, because vectors are a fundamental building block of R and, therefore, it is inherently vectorized (you can do the same thing to all values, not to one value at a time). In a sense, `for` loop is very un-R, so if you find yourself using it, consider whether there is a simpler or more expressive way to do this. At the same time, if `for` loop *is* the simplest, clearest, or more robust way to write you code, by all means, use it!

The general format is

```
for(loop_variable in vector_or_list){
  ...some operations using loop_variable that
```

```
    changes its value on each iteration using
    values from the vector or list...
}
```

Note the curly brackets. We used them before to put the code inside a function. Here, we use them to put the code inside the loop. The loop (the code inside curly brackets) is repeated as many times as the number of elements in a vector or a list with a loop variable<sup>2</sup> getting assigned each vector/list value on each iteration. Thus, to print each value of a vector we can do

```
for(a_number in c(1, 5, 200)){
  print(a_number)
}
```

```
## [1] 1
## [1] 5
## [1] 200
```

Here, we have three elements in a vector, therefore the code inside curly brackets is repeated three times with the variable `a_number` taking each value in turn. I.e., `a_number` is equal to 1 on a first iteration, 5 on a second, 200 on the third. Note that the code above is equivalent to just assigning one value at a time to `a_number` and calling `print()` function three times.

```
a_number <- 1
print(a_number)
```

```
## [1] 1
a_number <- 5
print(a_number)
```

```
## [1] 5
a_number <- 200
print(a_number)
```

```
## [1] 200
```

As you can see, it does not really matter *how* you assign a value to a variable and repeat the code. However, the `for` loop approach is much more flexible, is easier to read and to maintain. I.e., let us assume that you decided to alter the `print()` call with `cat()` instead. In the `for` loop version there is just one line to take care of. In the copy-paste version, there are three lines you need to alter. And imagine if you need to repeat the same code a hundred or thousand times, copy-paste is clearly not a viable solution.

Also note that you might be interested in repeat the code inside the `for` loop

---

<sup>2</sup>Just a reminder, the loop variable can have *any* name. Often, you see people using `i` but I would strongly recommend going for a more meaningful name.

given number of times but are not interested in a loop variable and values that it takes. For example, we might want to repeat `print("Ho!")` three times (because it is Christmas time and “Ho! Ho! Ho!” is what Santa Clause says). In this case, we still need a vector with three elements that we can loop over but we do not care what these three elements are.

```
for(temp in 1:3){
  print("Ho!")
}
```

```
## [1] "Ho!"
## [1] "Ho!"
## [1] "Ho!"
```

Note that we are not using variable `temp` inside of the loop and it has no effect on the code inside the curly brackets. Which is why we can use *any* three values, as it is their total number that matters, not the values themselves.

```
for(temp in c("A", "B", "C")){
  print("Ho!")
}
```

```
## [1] "Ho!"
## [1] "Ho!"
## [1] "Ho!"
```

or

```
for(temp in seq(100, 300, length.out=3)){
  print("Ho!")
}
```

```
## [1] "Ho!"
## [1] "Ho!"
## [1] "Ho!"
```

In short, number of elements in the vector determines how many times the code inside the loop will be repeat. Vector elements are stored in the loop variable on each iteration and you can either use it (`a_number` example above) or ignore them (“Ho! Ho! Ho!” example).

Do exercises 3 and 4.

One for a typical scenarios for the loop variable is when it is used as an *index* to access an element of a vector or a list. You can build a vector of indexes via `start:stop` sequence tool we used for slicing. You can compute a length of an object via `length()` function. For a `data.frame` or a `tibble`, you can figure out number of rows and columns via, respectively, `nrow()` and `ncol()` functions.

```
vector_of_some_numbers <- c(1, 5, 200)
for(index in 1:length(vector_of_some_numbers)){
```

```
print(vector_of_some_numbers[index])
}
```

```
## [1] 1
## [1] 5
## [1] 200
```

Do exercise 5.

You can also *nest* loops

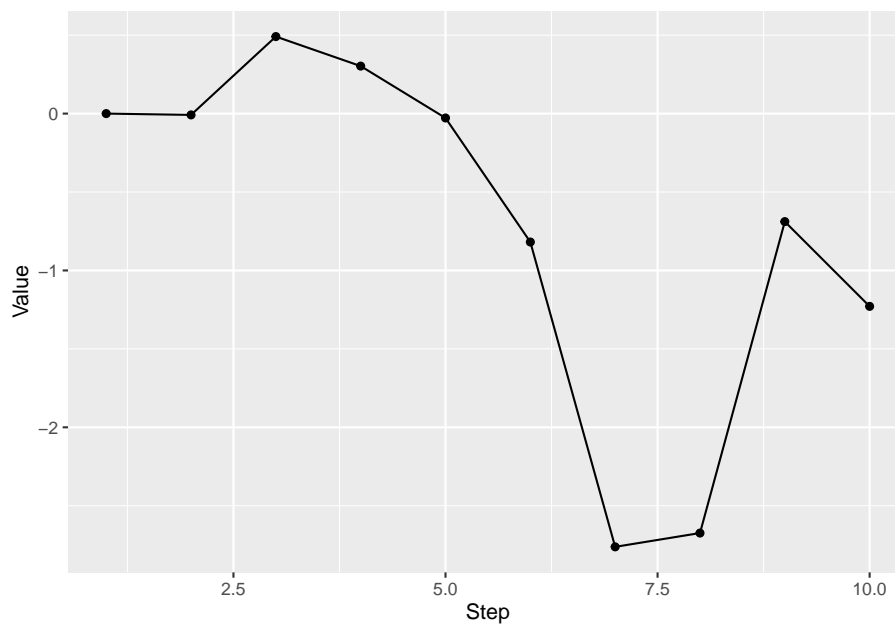
```
for(letter in c("A", "B", "C")){
  for(number in 1:2){
    cat(letter, number, "\n")
  }
}
```

```
## A 1
## A 2
## B 1
## B 2
## C 1
## C 2
```

Do exercise 6.

One scenario when loops are particularly advantageous is when a current value depends on a previous one (or many previous values). In this case, a new value depends on the previous one and, in turn, determines the following ones and they cannot be computed before it is known. This means that vectorized approach that applies the computation to all values in parallel will not work and we need to perform the computation in a sequential manner.

In the next exercise, use `for` loop to create a random walk. It should start at zero (so, initial value for your variable should be zero). For each next step, draw a random value from a normal distribution using *previous* value as a mean (decide on the standard deviation yourself). The function you are looking for is `rnorm()`. Generate a ten-step random walk, that might look like this (you can my plot replicate exactly, if you `set.seed()` to 1977):



Do exercise 7.

## 9.4 Conditional statement

As in all other programming languages, you can control the flow of execution using if-else statements. The general usage is as follows.

```
if (some_condition) {  
  # code runs if some_condition is TRUE  
} else {  
  # code runs if some_condition is FALSE  
}
```

The rules are the same as for logical indexing, so we can define our condition using mathematical comparisons.

```
x <- -3  
if (x > 0) {  
  cat("X is definitely greater than zero")  
} else {  
  cat("X is not greater than zero.")  
}
```

## X is not greater than zero.

However, be aware that the if uses a *single* logical value. If you have more than

one value, it stops with an error<sup>3</sup>.

```
x <- c(10, -3)
if (x < 0) {
  cat("Bingo!")
} else {
  cat("I don't like this x.")
}
```

```
## Error in if (x < 0) {: the condition has length > 1
```

As in logical indexing, you can combine several conditions using logical operators such as *and* `&&` or *or* `||`.

For example, we can check whether `x` is smaller than zero but larger than -10.

```
x <- -3
if ((x < 0) && (x > -10)) {
  cat("Bingo!")
} else {
  cat("I don't like this x.")
}
```

```
## Bingo!
```

However, be aware that R has both `&/|` and `&&/||` versions of *and* and *or* logical operators (single versus double symbols). Both perform logical operations on vectors but double-symbol works only for a single value. E.g., single-symbol returns TRUE/FALSE for each value of `x`

```
x <- c(10, -3, -11)
(x < 0) & (x > -10)
```

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

`&&` and `||` will generate an error if vectors are longer<sup>4</sup>.

```
# This works
x <- -3
(x < 0) && (x > -10)
```

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

```
# This generates an error
x <- c(10, -3, -11)
(x < 0) && (x > -10)
```

<sup>3</sup>If you use an older R-version, it will use *only the first one* but, at least, it will give you a warning. Point being that it is generally a good idea to use the latest R distribution.

<sup>4</sup>This is a new and very nice behavior introduced recently. Prior to that both `&&` and `||` would *silently* ignore all elements but the first, making them very dangerous to use, so that I recommended against their use in the previous version of the book. Now it is the other way around, so they are the preferred way when working with scalars.



```
## Error in (x < 0) && (x > -10): 'length = 3' in coercion to 'logical(1)'
```

Let us combine `for` loop with `if-else` operator. Generate a vector of ten normally distributed values (again, `rnorm()` is the function). Loop over them in a `for` loop and `print()` or `cat()` "Positive" if a value is larger than zero and "Not positive" if not. The results should look like this<sup>5</sup>

```
## [1] 0.03003683 1.22390943 1.71573769 -0.89994016 0.55507190 0.42319195
## [7] 0.82993426 -1.28614375 1.21511589 -0.05815403

## [1] "Positive"
## [1] "Positive"
## [1] "Positive"
## [1] "Not positive"
## [1] "Positive"
## [1] "Positive"
## [1] "Positive"
## [1] "Not positive"
## [1] "Positive"
## [1] "Not positive"
```

Do exercise 8.

## 9.5 ifelse()

As I wrote above, strength of R lies in built-in vectorization that typically renders loops unnecessary. Although we used the `for` loop to apply comparison to one element at a time, R has function `ifelse()` that performs the same task on vectors. Its format is `ifelse(logical-vector, values-if-true, values-if-false)` where the logical vector of `TRUE` and `FALSE` values can be produced via a comparison. For example, we can classify letters into "Vowel" and "Consonant" using `%in%` to check whether each letter is *in* the `vowels` list (note the `%`, this distinguished `%in%` for matching from `in` of the `for` loop).

```
vowels <- c("a", "e", "i", "o", "u")
letters <- c("a", "b", "c", "d", "e", "f")
ifelse(letters %in% vowels, "Vowel", "Consonant")
```

```
## [1] "Vowel"      "Consonant" "Consonant" "Consonant" "Vowel"      "Consonant"
```

Note that vectors `values-if-true` and `values-if-false` must not be of equal length. For example, one can use `ifelse()` to replace only vowels in the original vector but keep consonant letter as is by using `letters` vector for `values-if-false`

---

<sup>5</sup>Want exact same numbers as I do? Use `set.seed(164)`. This function, see `set.seed()` makes random generators start at a specific point, so that the you will get the *same* sequence of *random* numbers as I did.

```
ifelse(letters %in% vowels, "Vowel", letters)
```

```
## [1] "Vowel" "b"      "c"      "d"      "Vowel" "f"
```

However vectors `values-if-true` and `values-if-false` ideally should either match the length of the logical comparison vector or have a single value. Otherwise, they will be recycled which might lead to confusing results. E.g., here, can you figure out when `"Consonant1"` and `"Consonant2"` are used?

```
ifelse(letters %in% vowels, "Vowel", c("Consonant1", "Consonant2"))
```

```
## [1] "Vowel"      "Consonant2" "Consonant1" "Consonant2" "Vowel"
## [6] "Consonant2"
```

Use `ifelse()` to replicate exercise 8 without a loop.

Do exercise 9.

## 9.6 case\_when

Function `ifelse()` is useful when you have just two outcomes. You can use nested `ifelse` calls, just like you can use nested `if-else` statements but it becomes awkward and hard to read very fast. Instead, you can use function `case_when()` from `dplyr` library. Here, you can use as many conditions as you want. The general format is

```
case_when(Condition1 ~ Value1,
          Condition2 ~ Value2,
          ...
          ConditionN ~ ValueN)
```

For example, the internet tells me that letter `y` is complicated, as it can be either a consonant or a vowel depending on a word. Thus, we can label it as “Complicated” via `case_when()` (note the vectorized nature of it, as it applies this comparison to one element of the vector at a time)

```
vowels <- c("a", "e", "i", "o", "u")
letters <- c("a", "b", "c", "d", "e", "f", "y")
case_when(letters == "y" ~ "Complicated",
          letters %in% vowels ~ "Vowel",
          TRUE ~ "Consonant")
```

```
## [1] "Vowel"      "Consonant"  "Consonant"  "Consonant"  "Vowel"
## [6] "Consonant"  "Complicated"
```

Note the last condition that is always `TRUE`, it looks odd but this serves as a “default” branch: if you got that far, all other conditions must be `FALSE`, so this *must* be `TRUE` (you have no other options left).

Do exercise 10.

## 9.7 Breaking out of the loop

The code inside the for loop is typically repeated for every value of the vector that you have supplied. However, there is a way to break out of it using a **break** statement. This stops execution of the code *inside* of the loop immediately and continues with the code immediately *after* the loop.

```
for(y in c(1, 2, 3)){
  print("This will be executed")
  break
  print("But this won't be")
}
```

```
## [1] "This will be executed"
print("Now to the code AFTER the loop")
```

```
## [1] "Now to the code AFTER the loop"
```

Note that in this case, the code was executed (incompletely!) only once. Typically, **break** is used in combination with the **if-else** statement to break out of the loop, if a certain condition is met. Let us practice. Again, generate ten normally distributed numbers, loop over them and print each one. However, **break** after the *fifth* value. For this, you need to loop over indexes of values (that go from 1 to the `length()` of your vector). Thus, your loop variable will contain an index of each element of `x` (because of that I called it `ix`) and you need to use it to get a value at this position within the vector. If that index is equal to 5, break out of the loop. For the ten values I've generated above, the output will be the following ("Done for today" is printed after the loop, also I could have first printed a value and then exited the loop, that is a analysis logic question, not a technical one).

```
set.seed(164)
x <- rnorm(10)
for(ix in 1:length(x)){
  if (ix == 5) break
  print(x[ix])
}
```

```
## [1] 0.03003683
## [1] 1.223909
## [1] 1.715738
## [1] -0.8999402
print("Done for today")
```

```
## [1] "Done for today"
```

Do exercise 11.

## 9.8 Using for loop to load and join multiple data files

Your analysis starts with loading the data. Quite often, data for individual participants is stored in different files but with identical structure, so you need code that figures out which files you need to load, loads them one at a time and then binds them to one final table. Using a `for` loop is not the most elegant way to implement this but it does the job and gives you another example of how loops can be useful. I will walk you through details and then you We will implement the code that loads and merges individual files for persistence study. Download the `persistence.zip` and unzip into `Persistence` subfolder (we do not want to create a mess in your main folder!).

First, you need to have a character vector with relevant file names. Package you are looking for is `fs` (for File System). It has everything you need to work with the file system, including working with file names (dropping or adding path, extension, etc.), creating/moving/deleting files, checking whether file exists, and what not. One function that I use the most is `dir_ls()` that list files in a specified folder. The two parameters you need are `path` to your folder (you can and should use a relative path) and, optionally, `glob` filter string. The latter is a globbing wildcard pattern, where `*` stands for “any sequence of characters” and `?` stand for “one arbitrary character. For a csv file, this pattern would be `*.csv`”. Test this single function call using appropriate `path` and `glob` parameters and make sure you get all the files in *Persistence* folder.

Next, you need to create a full table variable (I, typically, call it `results` or `reports`) and initialize it to an empty `data.frame()` (or an empty `tibble`). You loop over file names, read one file at a time (don’t forget to specify column types), and then use `bind_rows()` to combine the full table and the new table you loaded. Note that `bind_rows()` returns a *new* table, so you need to assign it back to the original full table variable. Once you are done, your table should have 5232 rows and twelve columns.

Participant	Session	Block	Trial	OnsetDelay	Bias	Shape1	Shape2
AKM1995M	2019-06-12-14-07-17	0	0	0.5746952	left	stripes-8	stripes-4
AKM1995M	2019-06-12-14-07-17	0	1	0.5741707	left	stripes-4	heavy poles spl
AKM1995M	2019-06-12-14-07-17	0	2	0.5082200	left	stripes-2	stripes-2
AKM1995M	2019-06-12-14-07-17	0	3	0.6065058	right	stripes-8	stripes-2
AKM1995M	2019-06-12-14-07-17	0	4	0.5359504	left	stripes-2	heavy poles spl
AKM1995M	2019-06-12-14-07-17	0	5	0.6435367	right	stripes-4	stripes-4

Do exercise 12.

## 9.9 Apply

As noted above, `for` loops do the job by might not be the most elegant way of doing things. In R, you can apply a function to each row or column of a matrix.

In addition, there are more case-specific versions of it, such `lapply`.

The function is called `apply` because you *apply* it to values of a vector. In a sense, you have been applying functions the whole time by calling them. For example, we might compute a sinus of a sequence of numbers as

```
sin(seq(0, pi, length.out = 5))
```

```
## [1] 0.000000e+00 7.071068e-01 1.000000e+00 7.071068e-01 1.224606e-16
```

Or, we can *apply* sinus function to a number sequence (note that I pass the name of the function alone `sin` but do not call it, so no round brackets!)

```
sapply(seq(0, pi, length.out = 5), sin)
```

```
## [1] 0.000000e+00 7.071068e-01 1.000000e+00 7.071068e-01 1.224606e-16
```

You might ask, what is then the point to use `apply`? Not much for simple vector cases like this, but it is very useful when you have two dimensional data, as you can apply a function along horizontal (rows) or vertical (columns) margin. For example, imagine you need to compute an average (or median, or any other quantile) of each row or column in a matrix (something you might do fairly often for posterior samples in Bayesian statistics).

Let us create a simple 3 by 4 matrix of normally distributed random numbers.

```
a_matrix <- matrix(rnorm(12), nrow = 3)
a_matrix
```

```
##           [,1]      [,2]      [,3]      [,4]
## [1,] -1.306076e-05 -0.9815949  0.3981706 -0.1809577
## [2,]  1.880631e-02 -0.4210554 -1.2544423 -0.2319481
## [3,]  2.429102e+00 -1.1964317  0.2265607  0.5361628
```

We would expect median value of any row or column to be 0 but because we have so few data points, they will be close but not exactly zero. Computing median for each row (we should get *three* numbers)

```
apply(a_matrix, MARGIN = 1, FUN = median)
```

```
## [1] -0.0904854 -0.3265017  0.3813618
```

Similarly for a column (here, it should be *four* numbers)

```
apply(a_matrix, MARGIN = 2, FUN = median)
```

```
## [1]  0.01880631 -0.98159487  0.22656072 -0.18095773
```

I will not go into any further details on these functions, concentrating on similar functionality by `purrr` package. However, if you find yourself working with matrices or needing to apply a function to rows of a data frame, `apply` might be a simpler solution. Keep this option in mind, if you feel that either looping or `purrr` looks inadequate.

## 9.10 Purrr

Package `purrr` is part of the `tidyverse`. It provides functional programming approach similar to `apply` but it easier to use (IMHO) and it has a more explicit and consistent way to describe and combine the output. Language-wise, you do not *apply* a function, you use it to map inputs on outputs<sup>6</sup>

The basic `map()` function always returns a list but you can explicitly state that you expect function to return a number (`map_dbl()`) and all outputs will be combined into a numeric vector. And, unlike `apply`, `map_dbl()` will generate an error if outputs cannot be converted to numeric.

The basic call is similar to `apply` but is easier to use as you can explicitly address current value via `.` variable (as in other parts of `Tidyverse`) and you can write a “normal” function call, prefixing it with a `~`. Here is the example of computing the sinus again. First, same a `apply` via function name only

```
map_dbl(seq(0, pi, length.out = 5), sin)
```

```
## [1] 0.000000e+00 7.071068e-01 1.000000e+00 7.071068e-01 1.224606e-16
```

Now, we a magic tilde `~`. Note an explicit call to `sin()` function with `.` as an argument.

```
map_dbl(seq(0, pi, length.out = 5), ~sin(.))
```

```
## [1] 0.000000e+00 7.071068e-01 1.000000e+00 7.071068e-01 1.224606e-16
```

Again, using `map_dbl()` in this case looks as a complete overkill. So let us do something more relevant. Let us implement loading and merging of persistence study files. You already know how to get a vector with names of relevant files. Now you can use `map()` function on this vector to generate a list of tables and then combine them into a single table by piping it to `list_rbind()` which binds tables by rows (there is a twin `list_cbind()` function that binds table by columns). When using `~` call notation, remember `.` would then correspond to a single value from the vector of file names (so, a *single* filename). Again, you should get a single table with twelve columns and 5232 rows. You need a *single* pipeline for the entire operation.

Do exercise 13.

You have just *mapped* inputs on outputs using `read_csv()` but functional programming is particularly useful, if you program your own functions. Let us program a function that takes a filename, loads the file and returns total number of trials/rows (if you forgot how to compute number of rows in a table, see above). Once you have a function, use it with `map_dbl` and a vector of persistence filenames. You should get a vector of ten values. Now we can easily see

---

<sup>6</sup>Means the same thing but this is a linear algebra way of expressing of what a function does.

that there was something wrong with one of the files and we must pay attention to the amount of data that we have.

```
## [1] 528 528 480 528 528 528 528 528 528 528
```

Do exercise 14.





## Chapter 10

# Missing data

Grab an exercise notebook before we start!

Sometimes data is missing. It can be missing *explicitly* with `NA` standing for Not Available / Missing data. Or, it can be missing *implicitly* when there is no entry for a particular condition. In the latter case, the strategy is to make missing values explicit first (discussed below).

Then (once) you have missing values, represented by `NA` in R, you must decide how to deal with them: you can use this information directly as missing data can be diagnostic in itself, you can impute values using either a sophisticated statistical methods or via a simple average/default value strategy, or you can exclude them from the analysis. Every option has pros and cons, so think carefully and do not use an option whose effects you do not fully understand as it will compromise the rest of your analysis.

### 10.1 Making missing data explicit (completing data)

To make implicit missing data explicit, *tidyr* provides a function `complete()` that you already met. It figures out all combinations of values for columns that you specified, finds missing combinations, and adds them using `NA` (or some other specified value) for other columns. Imagine a toy incomplete table (no data for Participant 2 and Face M-2).

We can complete that table by specifying columns that define all required combinations.

```
complete_df <- complete(incomplete_df, Participant, Face)
```

For *non-factor* variables (`Participant` is numeric and `Face` is character/string),

Table 10.1: Table with no data for Face M2 for Participant 2.

Participant	Face	Symmetry	Attractiveness	Trustworthiness
1	M-1	6	4	3
1	M-2	4	7	6
2	M-1	5	2	1

Table 10.2: Completed table with explicit NAs

Participant	Face	Symmetry	Attractiveness	Trustworthiness
1	M-1	6	4	3
1	M-2	4	7	6
2	M-1	5	2	1
2	M-2	NA	NA	NA

complete finds all unique values for each column and finds all combinations of these elements. However, if a variable is a factor, complete uses its levels, even if not all levels are present in the data. E.g., we can use **Face** as a factor with three levels: “M-1”, “M-2”, and “F-1”. In this case, information is missing for both participants (neither have responses on face “F-1”) and should be filled with NAs. This approach is useful if you know all combinations that *should* be present in the data and need to ensure the completeness.

```
extended_df <-
  incomplete_df |>
  # converting Face to factor with THREE levels (only TWO are present in the data)
  mutate(Face = factor(Face, levels = c("M-1", "M-2", "F-1"))) |>
  # completing the table
  complete(Participant, Face)
```

Do exercise 1.

You can also supply default values via **fill** parameter that takes a named

Table 10.3: Completed missing data including F-1 face.

Participant	Face	Symmetry	Attractiveness	Trustworthiness
1	M-1	6	4	3
1	M-2	4	7	6
1	F-1	NA	NA	NA
2	M-1	5	2	1
2	M-2	NA	NA	NA
2	F-1	NA	NA	NA

Table 10.4: Completed missing data with non-NA values.

Participant	Face	Symmetry	Attractiveness	Trustworthiness
1	M-1	6	4	3
1	M-2	4	7	6
2	M-1	5	2	1
2	M-2	0	0	NA

list, e.g., `list(column_name = default_value)`. However, I'd like to remind you again that you should only impute values that “make sense” given the rest of your analysis. Zeros here are for illustration only and, in a real-life scenario, would ruin your inferences either by artificially lowering symmetry and attractiveness of the second face or (if you are lucky) will break and stop the analysis that expects only values within 1-7 range (rmANOVA won't be bothered at that would be the first scenario),

```
filled_df <-
  incomplete_df |>
  complete(Participant, Face, fill=list(Attractiveness=0, Symmetry=0))
```

Do exercise 2.

The `complete()` is easy to use convenience function that you can easily replicate yourself. To do this, you need to create a new table that lists all combinations of variables that you are interested in (you can use either `expand.grid()` or `expand_grid()` for this) and then left joining the original table to it (why left join? Could you use another join for the same purpose?). The results is the same as with a `complete()` itself.

Do exercise 3.

## 10.2 Dropping / omitting NAs

There are two approaches for excluding missing values. You can exclude all incomplete rows which have missing values in *any* variable via `na.omit()` (base R function) or `drop_na()` (tidyr package function). Or you can exclude rows only if they have NA in a specific columns by specifying their names.

For a table you see below

First, we can ensure only complete cases via `na.omit()`

```
na.omit(widish_df_with_NA)
```

or via `drop_na()`

```
widish_df_with_NA |>
  drop_na()
```

Table 10.5: Table with missing values.

Participant	Face	Symmetry	Attractiveness	Trustworthiness
1	M-1	6	NA	3
1	M-2	NA	7	NA
2	M-1	5	2	1
2	M-2	3	7	2

Table 10.6: Complete cases via `na.omit()`

Participant	Face	Symmetry	Attractiveness	Trustworthiness
2	M-1	5	2	1
2	M-2	3	7	2

Second, we drop rows only if **Attractiveness** data is missing.

```
widish_df_with_NA |>
  drop_na(Attractiveness)
```

Practice time. Create you own table with missing values and exclude missing values using `na.omit()` and `drop_na()`.

Do exercise 4.

`drop_na()` is a very convenient function but you can replicate it functionality using `is.na()` in combination with filter `dplyr` function or logical indexing. Implement code that excludes rows if they contain **NA** in a specific column using these two approaches.

Do exercises 5 and 6.

Recall that you can write your own functions in R that you can use to create convenience wrappers like `drop_na()`. Implement code that uses logical indexing as a function that takes table (`data.frame`) as a first argument and name a of a single column as a second, filters out rows with **NA** in that column and returns the table back.

Do exercise 7.

As noted above, you can also impute values. The simplest strategy is to use

Table 10.7: Complete cases via `drop_na()`

Participant	Face	Symmetry	Attractiveness	Trustworthiness
2	M-1	5	2	1
2	M-2	3	7	2

Table 10.8: Complete Attractiveness via `drop_na()`

Participant	Face	Symmetry	Attractiveness	Trustworthiness
1	M-2	NA	7	NA
2	M-1	5	2	1
2	M-2	3	7	2

Table 10.9: Missing values filled with 0 and -1

Participant	Face	Symmetry	Attractiveness	Trustworthiness
1	M-1	6	0	3
1	M-2	-1	7	NA
2	M-1	5	2	1
2	M-2	3	7	2

either a fixed or an average (mean, median, etc.) value. `tidyr` function that performs a simple substitution is `replace_na()`<sup>1</sup> and, as a second parameter, it takes a named list of values `list(column_name = value_for_NA)`. For our toy table, we can replace missing `Attractiveness` and `Symmetry` values with some default value, e.g. 0 and -1 (this is very arbitrary, just to demonstrate how it works, do not do things like these for real analysis unless you know what you are doing!)

```
widish_df_with_NA |>
  replace_na(list(Attractiveness = 0, Symmetry = -1))
```

Do exercise 8.

Unfortunately, `replace_na()` works only with constant values and does not handle grouped tables very well<sup>2</sup>. So to replace an NA with a mean value of a *grouped* data, we need to combine some of our old knowledge with an `ifelse(condition, value_if_true, value_if_false)` function you learned about before. Recall that this function is a vectorized cousin of the if-else that takes 1) a vector of logical values (`condition`), 2) a vector values that are returned if `condition` is true, 3) a vector of values that are returned if `condition` is false. Note that the usual rules of vector length-matching apply, so if the three vectors have different length, they will be automatically (and silently) adjusted to match the length of `condition` vector. As with all computations, you can use original values themselves. Here is how to replace only negative values but keep the positive ones:

```
v <- c(-1, 3, 5, -2, 5)
ifelse(v < 0, 0, v)
```

<sup>1</sup>There is also an inverse function `na_if()` that converts a specific value to an NA.

<sup>2</sup>At least I wasn't able to figure out how to do this.

Table 10.10: adaptation\_with\_na.csv with missing values

Participant	Prime	Probe	Nsame	Ntotal
ma2	Sphere	Sphere	NA	119
ma2	Sphere	Quadro	23	NA
ma2	Sphere	Dual	NA	120
ma2	Sphere	Single	31	115
ma2	Quadro	Sphere	25	120
ma2	Quadro	Quadro	26	120

Table 10.11: adaptation\_with\_na.csv with imputed values

Participant	Prime	Probe	Nsame	Ntotal	Psame
ma2	Sphere	Sphere	36	119	0.2983741
ma2	Sphere	Quadro	23	120	0.1916667
ma2	Sphere	Dual	36	120	0.2983741
ma2	Sphere	Single	31	115	0.2695652
ma2	Quadro	Sphere	25	120	0.2083333
ma2	Quadro	Quadro	26	120	0.2166667

```
## [1] 0 3 5 0 5
```

We, essentially, tell the function, “if the condition is false, use the original value”. Now, your turn! Using the same vector and `ifelse()` function, replace negative values with a mean value of the positive values in the vector.

Do exercise 9.

Now that you know how to use `ifelse()`, replacing `NA` with a mean will be (relatively) easy. Use `adaptation_with_na` table and replace missing information using participant-specific values.

We have missing data in different columns, so we have to use different for each case. Here is one way to approach this problem. We cannot know the number of trials for a specific Prime  $\times$  Probe combination, but we can replace missing values for `Ntotal` with a participant-specific median value (a “typical” and integer number of trials but do not forget about `na.rm` option, see manual for details). `Nsame` is trickier. For this, compute proportion of same response for each condition `Psame = Nsame / Ntotal`. This will produce missing values whenever `Nsame` is missing. Now, replace missing `Psame` values (`is.na()`) with a mean *Psame per participant* (again, watch out for `na.rm`!) using `ifelse()` (you can use it inside `mutate()`). Finally, compute missing values for `Nsame` from `Psame` and `Ntotal` (do not forget to round them, so you end up with integer number of trials). This entire computation should be implemented as a single pipeline. You will end up with a following table.

Do exercise 10.





## Chapter 11

# Working with strings

When working with strings, I strongly suggest consulting a manual and vignettes of the `stringr` package. It has many functions that cover most needs. Grab exercise notebook before we start.

### 11.1 Warming up

Before we start working with strings, let us warm up by preprocessing `band-adaptation.csv` that we will be working with.

1. Read it (try specifying the URL instead of the local filename). Do not forget to specify column types!
2. compute proportion of “same” responses as a using `Nsame` (number of “same” responses) and `Ntotal` (total number of trials).
3. Convert `Prime` and `Probe` column to factors with the order “Sphere”, “Quadro”, “Dual”, “Single”.
4. Compute median and median absolute deviation from the median for `Psame` for all combinations of `Prime` and `Probe`.

Your table should look as follows:

Do exercise 1.

### 11.2 Formatting strings via `glue()`

The table above gives us information about *median* probability of seeing the same rotation and about its absolute deviation from the median. However, it would be more convenient for a reader if we combine these two pieces of information into a single entry in form for of “ $\pm$ ”. Plus, it would be easier to see the pattern in a square table with one `Prime` per row and one `Probe` per column. The table I have in mind look like this:

Table 11.1: bands\_df

Prime	Probe	Pmedian	Pmad
Sphere	Sphere	0.13	0.06
Sphere	Quadro	0.13	0.04
Sphere	Dual	0.17	0.10
Sphere	Single	0.32	0.08
Quadro	Sphere	0.19	0.07
Quadro	Quadro	0.12	0.12
Quadro	Dual	0.21	0.19
Quadro	Single	0.38	0.07
Dual	Sphere	0.15	0.15
Dual	Quadro	0.30	0.14
Dual	Dual	0.27	0.15
Dual	Single	0.48	0.16
Single	Sphere	0.34	0.18
Single	Quadro	0.30	0.20
Single	Dual	0.48	0.12
Single	Single	0.51	0.18

Table 11.2: Probability of persistence, median  $\pm$  MAD

Prime	Sphere	Quadro	Dual	Single
Sphere	$0.13 \pm 0.06$	$0.13 \pm 0.04$	$0.17 \pm 0.1$	$0.32 \pm 0.08$
Quadro	$0.19 \pm 0.07$	$0.12 \pm 0.12$	$0.21 \pm 0.19$	$0.38 \pm 0.07$
Dual	$0.15 \pm 0.15$	$0.3 \pm 0.14$	$0.27 \pm 0.15$	$0.48 \pm 0.16$
Single	$0.34 \pm 0.18$	$0.3 \pm 0.2$	$0.48 \pm 0.12$	$0.51 \pm 0.18$

You already know how to perform the second step (pivoting table wider to turn `Probe` factor levels into columns). For the first step, you need to combine two values into a string. There are different ways to construct this string via `sprintf()`, `paste()`, or via `glue` package. We will start with Tidyverse's `glue()` and explore base R functions later.

`glue` package is part of the Tidyverse, so it should be already installed. However, it is not part of *core* tidyverse, so it does not get imported automatically via `library(tidyverse)` and you need to import it separately or use `glue::` prefix. Function `glue()` allows you to “glue” values and code directly into a string. You simply surround *any* R code by wiggly brackets inside the string and the result of the code execution is glued in. If you use just a variable, its value will be glued-in. But you can put *any* code inside, although, the more code you put, the harder it will be to read and understand it.

```
answer <- 42
bad_answer <- 41
glue::glue("The answer is {answer}, not {abs(bad_answer / -4)}")
```

```
## The answer is 42, not 10.25
```

Use the table that you prepared during exercise 1 to compute a new column with “ $\pm$ ” (you will want to use `round()` function to restrict values to just 2 digit after the decimal point). Think about *when* you want to perform this computation to make it easier (before or after pivoting?) and which column(s?) do you need to pivot wider.

Do exercise 2.

## 11.3 Formatting strings via `paste()`

Base R has functions `paste()` and `paste0()` that concatenate a vector of strings into a single string. If you recall, vector values can only be of one (most flexible) type. Therefore, if you have a vector that intersperses strings with other values, they will be *first* converted to strings anyhow. The difference between `paste()` and `paste0()` is that the former puts a separator string in-between each value (defaults to “ ” but you can define your own via `sep` argument), whereas `paste0()` uses no separator. We can replicate our `glue()` example.

```
answer <- 42
bad_answer <- 41
paste("The answer is ", answer, ", not ", abs(bad_answer / -4), sep = "")
```

```
## [1] "The answer is 42, not 10.25"
```

```
paste0("The answer is ", answer, ", not ", abs(bad_answer / -4))
```

```
## [1] "The answer is 42, not 10.25"
```

Redo exercise 2 but using one of the paste functions instead of the glue().

Do exercise 3.

## 11.4 Formatting strings via `sprintf()`

For detailed string formatting, base R has a `sprintf()` function that provides a C-style string formatting (same as Python's original string formatting and a common way to format a string in many programming languages). The general function call is `sprintf("string with formatting", value1, value2, value)`, where values are inserted into the string. In "string with formatting", you specify where you want to put the value via `%` symbol that is followed by an *optional* formatting info and the *required* symbol that defines the **type** of the value. The type symbols are

- **s** for string
- **d** for an integer
- **f** for a float value using a “fixed point” decimal notation
- **e** for a float value using a scientific notation (e.g., `1e2`).
- **g** for an “optimally” printed float value, so that scientific notation is used for very large or very small values (e.g., `1e+5` instead of `100000` and `1-e5` for `0.00001`).

Here is an example of formatting a string using an integer:

```
sprintf("I had %d pancakes for breakfast", 10)
```

```
## [1] "I had 10 pancakes for breakfast"
```

You are not limited to a single value that you can put into a string. You can specify more locations via `%` but you must make sure that you pass the matching number of values. If there fewer parameters when you specified in the string, you will receive an *error*. If there are too many, only a *warning*<sup>1</sup>. Before running it, can you figure out which call will actually work (and what will be the output) and which will produce an error or a warning?

```
sprintf("I had %d pancakes and either %d or %d stakes for dinner", 2)
sprintf("I had %d pancakes and %d stakes for dinner", 7, 10)
sprintf("I had %d pancake and %d stakes for dinner", 1, 7, 10)
```

In case of real values you have two options: `%f` and `%g`. The latter uses scientific notation (e.g. `1e10` for `10000000000`) to make a representation more compact. When formatting floating numbers, you can specify the number of decimal points to be displayed.

```
e <- 2.71828182845904523536028747135266249775724709369995
sprintf("Euler's number is roughly %.4f", e)
```

---

<sup>1</sup>Talk about consistency...

```
## [1] "Euler's number is roughly 2.7183"
```

Note that as most functions in R, `sprintf()` is vectorized so when you pass a vector of values it will generate a *vector* of strings with one formatted string for a value.

```
sprintf("The number is %d", c(2, 3))
```

```
## [1] "The number is 2" "The number is 3"
```

This means that you can use `sprintf()` to work on column both in base R and inside `mutate()` Tidyverse verb.

```
tibble(Number = 1:3) |>
  mutate(Message = sprintf("The number is %d", Number)) |>
  knitr::kable()
```

Number	Message
1	The number is 1
2	The number is 2
3	The number is 3

Redo exercise #2 but use `sprintf()` instead of `glue()`.

Do exercise 4.

## 11.5 Extracting information from a string

Previous exercises dealt with combining various bits of information into a single string. Often, you also need to do the opposite: extract bits of information from a single string. For example, in the toy table on face perception, we have been working with, **Face** column code gender of the face "M" (table is short but you can easily assume that faces of both genders were used) and the second is its index (1 and 2). When we worked with persistence, **Participant** column encoded year of birth and gender, whereas **Session** contained detailed information about year, month, day, hour, minutes, and seconds all merged together. There are several ways to extract this information, either by extracting one piece at a time via `substr()` or string processing library `stringr`. Alternatively, you can split a string column into several columns via `separate()` or use `extract()` function.

## 11.6 Splitting strings via `separate()`

Function `separate()` is part of `tidyr` and its use is very straightforward: you pass 1) the name of the column that you want to split, 2) names of the columns it needs to be split into, 3) a separator symbol or indexes of splitting positions. Examples using the face table should make it clear. Reminder, this is the original wide table and we want to separate **Face** into **FaceGender** and **FaceIndex**.

```
widish_df <-
  tibble(Participant = c(1, 1, 2, 2),
         Face = rep(c("M-1", "M-2"), 2),
         Symmetry = c(6, 4, 5, 3),
         Attractiveness = c(4, 7, 2, 7),
         Trustworthiness = c(3, 6, 1, 2))

knitr::kable(widish_df)
```

Participant	Face	Symmetry	Attractiveness	Trustworthiness
1	M-1	6	4	3
1	M-2	4	7	6
2	M-1	5	2	1
2	M-2	3	7	2

As there is a very convenient “dash” between the two, we can use it for a separator symbol:

```
widish_df |>
  separate(Face, into=c("FaceGender", "FaceIndex"), sep="-")
```

Participant	FaceGender	FaceIndex	Symmetry	Attractiveness	Trustworthiness
1	M	1	6	4	3
1	M	2	4	7	6
2	M	1	5	2	1
2	M	2	3	7	2

Note that the original `Face` column is gone. We can keep it via `remove=FALSE` option

```
widish_df |>
  separate(Face, into=c("FaceGender", "FaceIndex"), sep="-", remove=FALSE)
```

Participant	Face	FaceGender	FaceIndex	Symmetry	Attractiveness	Trustworthiness
1	M-1	M	1	6	4	3
1	M-2	M	2	4	7	6
2	M-1	M	1	5	2	1
2	M-2	M	2	3	7	2

We also do not need to extract *all* information. For example, we can extract only face gender or face index. To get only the gender, we only specify *one* `into` column and add `extra="drop"` parameter, telling `separate()` to drop any extra piece it obtained:

```
widish_df |>
  separate(Face, into=c("Gender"), sep="-", remove=FALSE, extra="drop")
```

Participant	Face	Gender	Symmetry	Attractiveness	Trustworthiness
1	M-1	M	6	4	3
1	M-2	M	4	7	6
2	M-1	M	5	2	1
2	M-2	M	3	7	2

Alternatively, we can explicitly *ignore* pieces by using NA for their column name:

```
widish_df |>
  separate(Face, into=c("Gender", NA), sep="-", remove=FALSE)

widish_df |>
  separate(Face, into=c("Gender", NA), sep="-", remove=FALSE) |>
  knitr::kable()
```

Participant	Face	Gender	Symmetry	Attractiveness	Trustworthiness
1	M-1	M	6	4	3
1	M-2	M	4	7	6
2	M-1	M	5	2	1
2	M-2	M	3	7	2

What about keeping only the *second* piece in a FaceIndex column? We ignore the first one via NA

```
widish_df |>
  separate(Face, into=c(NA, "Index"), sep="-", remove=FALSE)

widish_df |>
  separate(Face, into=c(NA, "Index"), sep="-", remove=FALSE) |>
  knitr::kable(align = "c")
```

Participant	Face	Index	Symmetry	Attractiveness	Trustworthiness
1	M-1	1	6	4	3
1	M-2	2	4	7	6
2	M-1	1	5	2	1
2	M-2	2	3	7	2

Let's practice. Use `separate()` to preprocess persistence data and create two new columns for hour and minutes from `Session` column. Do it in a single pipeline, starting with reading all files (use tidyverse `read_csv()` and specify column types!) and renaming `Shape1` (`Prime`) and `Shape2` (`Probe`) columns. Your results should look like this, think about columns that you drop or keep (this is only first four rows, think of how you can limit your output the same way via `head()` or `slice_head()` functions):

Participant	Hour	Minutes	Block	Trial	OnsetDelay	Bias	Prime	Probe
AKM1995M	14	07	0	0	0.5746952	left	stripes-8	stripes-4
AKM1995M	14	07	0	1	0.5741707	left	stripes-4	heavy poles sphere
AKM1995M	14	07	0	2	0.5082200	left	stripes-2	stripes-2
AKM1995M	14	07	0	3	0.6065058	right	stripes-8	stripes-2

Do exercise 5.

As noted above, if position of individual pieces is fixed, you can specify it explicitly. Let us make out toy table a bit more explicit

Participant	Face	Symmetry	Attractiveness	Trustworthiness
1	M-01	6	4	3
1	F-02	4	7	6
2	M-01	5	2	1
2	F-02	3	7	2

For our toy faces table, the first piece is the gender and the last one is its index. Thus, we tell `separate()` starting position each pieces, starting with the *second* one:

```
widish_df |>
  separate(Face, into=c("FaceGender", "Dash", "FaceIndex"), sep=c(1, 2))
```

```
widish_df |>
  separate(Face,
    into = c("FaceGender", "Dash", "FaceIndex"),
    sep = c(1, 2),
    remove = FALSE) |>
  knitr::kable()
```

Participant	Face	FaceGender	Dash	FaceIndex	Symmetry	Attractiveness	Trustworthiness
1	M-01	M	-	01	6	4	
1	F-02	F	-	02	4	7	
2	M-01	M	-	01	5	2	
2	F-02	F	-	02	3	7	

Here, I've create `Dash` column for the separator but, of course, I could have omitted it via `NA` column name.

```
widish_df |>
  separate(Face, into=c("FaceGender", NA, "FaceIndex"), sep=c(1, 2))
```

```
widish_df |>
  separate(Face,
    into = c("FaceGender", NA, "FaceIndex"),
    sep = c(1, 2)) |>
  knitr::kable()
```



### 11.7. EXTRACTING A SUBSTRING WHEN YOU KNOW ITS LOCATION 177

Participant	FaceGender	FaceIndex	Symmetry	Attractiveness	Trustworthiness
1	M	01	6	4	3
1	F	02	4	7	6
2	M	01	5	2	1
2	F	02	3	7	2

Practice time! Using same persistence data extract birth year and gender of participants from **Participant** code (however, keep the code column). Put a nice extra touch by converting year to a number (`separate()` splits a string into strings as well) and gender into a factor type with better labels. Here is how should look like:

Participant	BirthYear	Gender	Hour	Minutes	Block	Trial	OnsetDelay	Bias	Prime	Probe
AKM1995M	1995	Female	14	07	0	0	0.5746952	left	stripes-8	stripes-4
AKM1995M	1995	Female	14	07	0	1	0.5741707	left	stripes-4	heavy po
AKM1995M	1995	Female	14	07	0	2	0.5082200	left	stripes-2	stripes-2
AKM1995M	1995	Female	14	07	0	3	0.6065058	right	stripes-8	stripes-2

Do exercise 6.

## 11.7 Extracting a substring when you know its location

Base R provides a function extract a substring (or many substrings) via `substr()` function (you can also its alias `substring()`). It takes a string (or a vector of strings) and vectors with **start** and **stop** indexes of each substring.

```
face_img <- c("M01", "M02", "F01", "F02")
substr(face_img, 2, 3)
```

```
## [1] "01" "02" "01" "02"
```

Repeat exercise 6 but use `substr()` to extract each column (**BirthYear** and **Gender**) from the participant code.

Do exercise 7.

Tidyverse has its own stringr library for working with strings. Its uses a consistent naming scheme **str\_<action>** for its function and covers virtually all tasks that are related to working with strings. stringr equivalent of `substr()` is `str_sub()` that behaves similarly.

```
face_img <- c("M01", "M02", "F01", "F02")
str_sub(face_img, 2, 3)
```

```
## [1] "01" "02" "01" "02"
```

Repeat exercise 7 but using `str_sub()` function.

Do exercise 8.

## 11.8 Detecting a substring using regular expressions

Some people, when confronted with a problem, think “I know, I’ll use regular expressions.” Now they have two problems.

*Jamie Zawinsk*

One of the most powerful ways to work with strings is via regular expressions that allow you to code a flexible pattern that is matched to a substring within a string. For example, you can detect whether a string contains a number without knowing where it is located. Here a pattern “`\\d{3}`” means that we are looking for 3 (hence the `{3}`) digits (hence the `\\d`). The base R has functions `grepl()`<sup>2</sup> and `grep()` that, correspondingly, return a vector of logical values of whether the pattern was match or index of vector elements for which that matched.

```
QandA <- c("What was the answer, 42, right?", "No idea! What could it be, 423?")
# returns logical vector for each element
grepl("\\d{3}", QandA)
```

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

```
# returns index of elements for which pattern was matched
grep("\\d{3}", QandA)
```

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

Stringr library has its own version with a more obvious name `str_detect()` that acts similar to `grepl()`, i.e., returns vector of logical values on whether the pattern was matched. Note, however, the reverse order of arguments, as `str_` function always take (a vector of) strings as a first parameter

```
str_detect(QandA, "\\d{3}")
```

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

You can also look for 1 or more digits (which is `+`)

```
str_detect(QandA, "\\d+")
```

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

Or for a specific word

```
str_detect(QandA, "What")
```

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

Or for a specific word only at the beginning (`^`) of the string

---

<sup>2</sup>GREP stands for Global Regular Expression Print.

```
str_detect(QandA, "^What")
```

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

When it comes to regular expressions, what I have shown you so far is not even a tip of an iceberg, it is a tip of a tip of an iceberg at best. They are very flexible, allowing you to code very complicated patterns but they are also hard to read and, therefore, hard to debug<sup>3</sup>. For example, this is a regular expression to check validity on an email address<sup>4</sup>

```
(?:[a-z0-9!#$%&'*/+=?^_`{|}~-]+(?:\.[a-z0-9!#$%&'*/+=?^_`{|}~-]+)*)|"(?:[\x01-\x08\x0b\x0c\x0e-\x1f\x21-\x2f\x3a-\x3f\x41-\x4f\x51-\x5f\x61-\x6f\x71-\x7f\x81-\x8f\x91-\x9f\xA1-\xAf\xB1-\xBf\xC1-\xCf\xD1-\xDf\xE1-\xEf\xF1-\xFf\x21-\x7f])"
```

Still, if you need to work with text they are indispensable, so you should remember about them. When facing an actual task grab a cheatsheet and use an online expression tester to debug the pattern.

In the next exercise, use a regular expression to `filter()` out Primes and Probes that *end* with a *single* digit. I know that all of them end with a single digit, if digit is in them, so you can make a very simple expression that would do the job. But I want you to practice working with the cheatsheet, so it must specify that only one digit is allowed and that it must be the last symbol. When your pattern works, you should end up with a table where all Primes and Probes are "heavy poles sphere".

Do exercise 9.

## 11.9 Extracting substring defined by a regular expression

You can not just detect a substring defined by a regular expression but also extract it. The advantage is that you may not know how many symbols are in the substring or where it starts, so regular expressions give you maximal flexibility. The function for this is `str_extract()` that works very similar to `str_detect()` but returns an actual detected substring instead of just `TRUE` or `FALSE`. Use it to extract the participants' unique code, the first three letters of the `Participant` column. Again, here you can simply use a `substr()` but I want you to write a pattern that matches 1) one or more 2) upper case letters 3) at the beginning of the string.

Do exercise 10.

<sup>3</sup>Hence, the quote at the beginning of the section.

<sup>4</sup>This is an official RFC 5322 standard that should work on *almost* all of the valid email addresses.

## 11.10 Replacing substring defined by a regular expression

Another manipulation is to replace an arbitrary substring with a fixed one. The base R provides functions `sub()` that replaces only the *first* occurrence of the matched pattern and `gsub()` that replaces *all* matched substring. Stringr equivalents are `str_replace()` and `str_replace_all()`. The main difference, as with `grepl()` versus `str_detect()` is the order of parameters: for `str_detect()` input string is the first parameter, followed by a pattern and a replacement string, whereas for `grepl()` it is pattern, replacement, input string order.

As an exercise, use `sub()` and `str_replace()` to anonymize the birth year of our participants. You need to replace the four digits that represent their birth year with a *single* "-". The table should look as follows:

Participant	Hour	Minutes	Block	Trial	OnsetDelay	Bias	Prime	Probe
AKM-M	14	07	0	0	0.5746952	left	stripes-8	stripes-4
AKM-M	14	07	0	1	0.5741707	left	stripes-4	heavy poles sphere
AKM-M	14	07	0	2	0.5082200	left	stripes-2	stripes-2
AKM-M	14	07	0	3	0.6065058	right	stripes-8	stripes-2

Do exercise 11.

Now, repeat the exercise by replace *any* single digit with '-'. Which functions do you use to produce the same results as in exercise 11?

Do exercise 12.

## Chapter 12

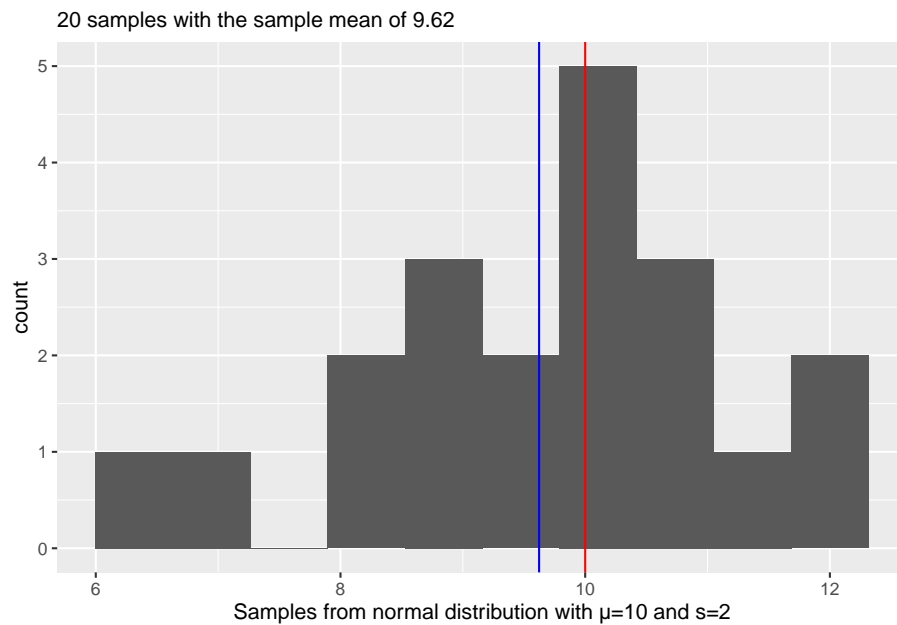
# Sampling and simulations

An important trick in your toolbox is a skill to resample and simulate data. The latter, sampling from predefined distributions, allows you develop your analysis routine and ensure that it can correctly recover the anticipated effects even before you have collected the data and perform a power analysis. Resampling your data paves way for non-parametric bootstrapping and permutation testing that helps you whenever assumptions of parametric tests are violated or when you require an estimate that is not easy to derive analytically.

Grab exercise notebook before reading on.

### 12.1 Estimating mean of a normal distribution via resampling

Let us start very simple. Your task will be to generate samples from a normal distribution and then use resampling approach to estimate the original mean. Step one is simple, decide on mean and standard deviation of the normal distribution and generate 20 samples using `rnorm()` function (`r<distribution` functions generate random number based on distribution and its parameters). Check your results visually by plotting a histogram and adding a red vertical line to indicate the true mean of the distribution. We also need to see the difference between the true mean and the sample mean, so include a blue vertical line to indicate the *sample* mean. Finally, it is always nice to have both visual and textual information in the plots, so add information about the true mean, number of samples, and sample mean to the plot's title. Run your code several times to appreciate variability of the data and, therefore, of the sample mean. Your plot should look something like this (my number of `set.seed` is 1745).



Do exercise 1.

In the real life, we do not know the true mean which is why we need to collect the data to begin with. We also know that our sample mean is different from the true mean<sup>1</sup> and we would like to know how much can we trust that value. In other words, we would like to know how much the *sample mean* would vary if we would draw some *other* samples from the same distribution. Theoretically, you want to draw samples from that “true” distribution directly. Practically, you do not have access to it, apart from replicating your experiment or study many times. Instead, you can make an educated guess about shape and parameters of this distribution. This is a parametric approach used to compute estimators analytically, e.g., from the Student t Distribution. This is the way it is done in the `t.test()`.

```
t.test(samples, mu = 10)
```

```
##
## One Sample t-test
##
## data: samples
## t = -1.1076, df = 19, p-value = 0.2819
## alternative hypothesis: true mean is not equal to 10
## 95 percent confidence interval:
```

<sup>1</sup>Mean is an unbiased estimator, so if we draw infinite samples, their distribution will center on the true mean but mean for each sample will be either (a tad or a lot) large or smaller than the true one.

```
##      8.911319 10.335209
## sample estimates:
## mean of x
##      9.623264
```

The other approach is to assume that your sample and, therefore, the data you collected is representative, so frequency of individual values in your sample is proportional to their probability, i.e., the more often you see a particular value, the more likely it is. In this case, sampling from the data is just like sampling from the true distribution. This is obviously a strong assumption, particularly for small samples, however, this approach can work with any data, regardless of its distribution, and can be used to estimate statistic that is not easy to derive analytically. Thus, below we will use a brute force approach that relies on sheer computer power to compute the same confidence interval as one analytically computed by the t-test through resampling of the data that you generated.

You will need three functions for this. First, the function that samples your data: `sample()`. It takes the original data (first parameter `x`) and randomly samples `size` items from it either with or without replacement (controlled by `replace` parameter that defaults to `FALSE`, so no replacement). In our case we want to get a sample of the size as the original data and we want to sample *with* replacement. *With replacement* means that once a value is drawn from the sample, it is recorded and then *put back in*, so it can be drawn again. *With replacement* procedure means that probability of drawing a particular value is always the same, whereas *without replacement* their probabilities change with every draw simply because there fewer and fewer values left to sample from.

For our purposes, we want to resample data and compute its mean. Write the code that does just that. Run the chunk several times to see how computed mean value changes due to resampling. As an exercise, set `replace = FALSE` and, *before running the chunk*, think what value do you expect and whether and how it would change when run the chunk again.

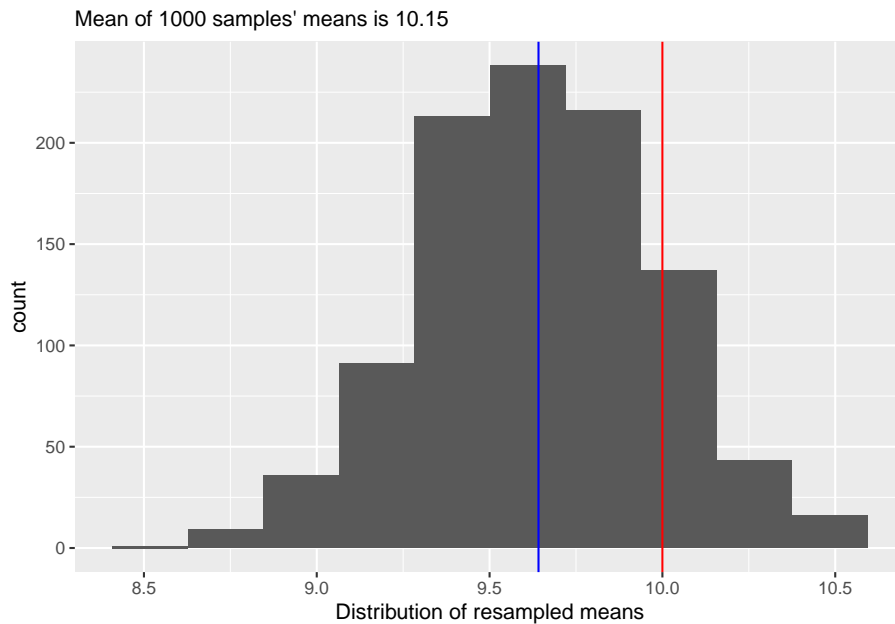
Do exercise 2.

Our resampling-and-computing-mean code is very simple and brief. However, it is generally a good idea to pack it into a function with a meaningful name. Do just that: turn the code of exercise 2 into a function (think about function parameters and what it should return) and call it to check that everything works (when you pass a sample to it, it should return a resampled mean for it).

Do exercise 3.

Our second step is to repeat our first step many (say, 1000) times. The base R function that helps you to do this is `replicate()`. That takes number of repetitions (first parameter `n`) and an arbitrary R code that returns a value (our step one). Once you run it, you will get a vector of 1000 means from resampled data. Plot the histogram, overlaying the true and

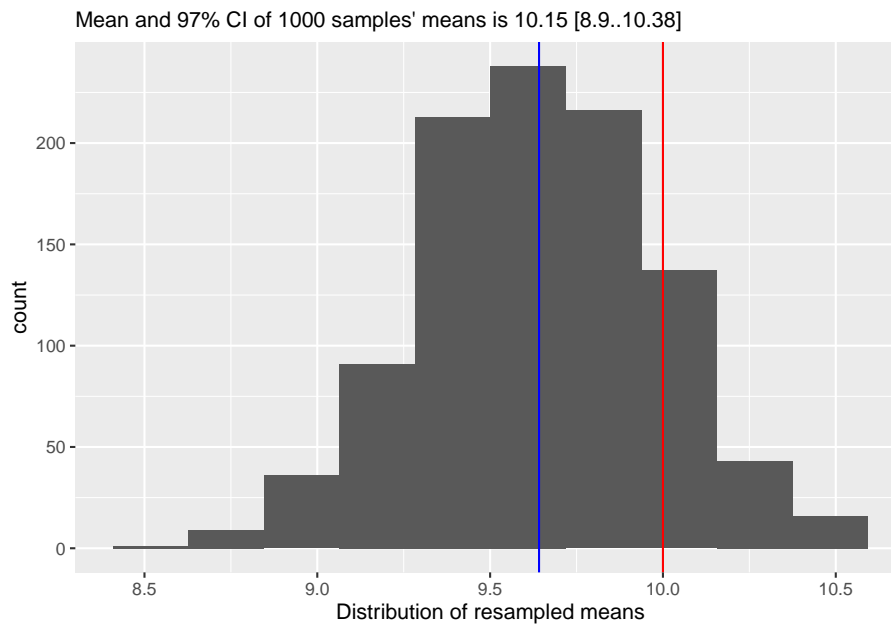
average samples' means (so mean of the means of samples, not a just a mean of all samples!) as a reference. Your plot should look like this



Our final step is to use `quantile()` function to compute 95% confidence interval. `quantile()` function takes a vector and computes a value that is greater than `probs` fraction of values in that vector. E.g., if `probs=c(0.25, 0.75)`, it will return a two values, so that 25% of values are smaller than the first one and 75% of them are smaller than the second. Or, to put it differently, 50% of all values are with `probs=c(0.25, 0.75)`. In our case, we want to compute 97%<sup>2</sup> confidence interval, i.e., 97% of all values should be between the lower and upper confidence interval values. Once you run the code, you should see that 97% confidence interval from resampling is very similar to what the t-test reported (you want get the same values due to random sampling but they should also be close to the t-test's analytic estimate). Add this information to the caption (often, this information is put directly into the text, but I find it simpler if all quantitative information is together and easy to find)

<sup>2</sup>Because 97 is a prime number and 95 is not, so 97 is a bit less arbitrary than 95!





::: {.infobox .practice} Do exercise 4. :::

## 12.2 Repeating computation via for loop

As we discussed in the a chapter on loops and repetitions, whereas many ways to repeat a computation in R. Let's replicate the sampling part of the code, using for loop. This is not the best way to perform our current task, but it is a safer approach if your computation is heavy and takes a long time, as it is easier to perform bookkeeping, retain data and restart the computation if something goes wrong (e.g., you run out of memory or file space), compared to functional programming via replicate or purrr, where you might need to start from scratch.

Think about how you will define number of iterations, whether you need to use the loop variable, how you concatenate new sample mean to the vector, etc.

Do exercise 5.

## 12.3 Repeating computation via purrr

Practice makes perfect, so let us replicate the code repeating via purrr library. Think about which map function is best for the job, whether you need to use the special . variable, etc.

Do exercise 6.

## 12.4 Bootstrapping via boot library

The approach that we used is called “bootstrapping” and R is all about giving you options, so it has a boot library to simplify and automate bootstrapping and the confidence interval computation. You do not need to install it (`boot` comes with base R) but you need to import it via `library(boot)`.

The key function is `boot()`. It has plenty of parameters that allow you to fine tune its performance but the three key compulsory parameters are

- **data**: your original data you want to use for bootstrapping.
- **statistic**: function(s) that compute desired statistic, such is mean in our case.
- **R**: the number of bootstrap replicates (we used 1000 when we did this by hand).

For non-parametric bootstrapping, like the one we used above, you will need to write the `statistic` function yourself even if you want to compute a statistic for which functions already exist, like mean or standard deviation. This is because `statistic` function must take at least two arguments: 1) the data that you passed and 2) how it should be resampled. By default, the second parameter will contain indexes of elements in the data. Note that bootstrap resamples *with* replacement, so the same index can appear more than once, i.e., the same element is drawn more than once (just as we did above).

Your statistic function should like as following, of course with a better name and an actual code inside.

```
your_statistic_function <- function(data, indexes){
  # here you compute desired statistic subsetting data using indexes
}
```

Once you have this function, you can bootstrap samples via

```
booted_samples <- boot(samples, statistic = your_statistic_function, R = 1000)
```

Next, use function `boot.ci()` to compute the confidence interval, which takes your bootstrapped samples as a first parameter. You can also specify the confidence interval you are interested in (`conf`, defaults to 0.95 but we want 97!) and `type` of the confidence interval. The one we computed above is called percentile (`type="perc"`), so this is the type you should specify<sup>3</sup>. Once you run the code the output should be similar to that below.

```
## BOOTSTRAP CONFIDENCE INTERVAL CALCULATIONS
## Based on 1000 bootstrap replicates
##
## CALL :
```

---

<sup>3</sup>It also supports other kinds of intervals, including bias corrected and accelerated CIs, so once you get serious about bootstrapping, you can use `boot` for more advanced analysis as well.

## 12.5. CONFIDENCE ABOUT PROPORTION OF RESPONSES FOR LIKERT SCALE<sup>187</sup>

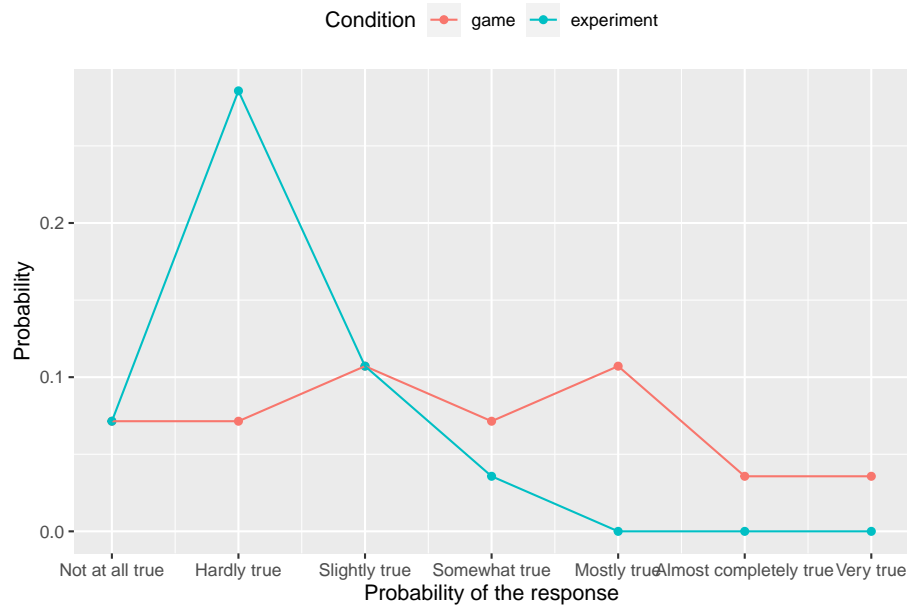
```
## boot.ci(boot.out = booted_samples, type = "perc")
##
## Intervals :
## Level      Percentile
## 95%      ( 8.939, 10.242 )
## Calculations and Intervals on Original Scale
```

As you can see, we very similar results as above (but for variation due to sampling). Thus, either approach will work but, in most cases, `boot` is more flexible solution (but do read on bootstrapping before using advanced options).

Do exercise 7.

## 12.5 Confidence about proportion of responses for Likert scale

We have looked at how one should present information on Likert scale responses a couple of times already. Let us return to it to see how you can compute not just a proportion of response level report, but also a percentile confidence interval for it, to see how reliable are these numbers. We will use a smaller file with just one scale — `likert-scale.csv` — and your first task is to reuse your old from chapter on factors but making sure that your counts are complete as we did in chapter on `tidyr`. Package the code that counts and computes proportions per condition and response level into a function. It should take a data frame with original data as the first input and a second parameter `resample` that should be `FALSE` by default (we will use it later). The function should return the summary table and you can reuse that you can plot using the same code as in chapter on factors. Also, you can reuse your original code for reading and preprocessing the data, *before* it is used to compute counts and proportions. Your plot should look almost exactly as before but for zeros where uncounted response levels were not plotted previously.



Do exercise 8.

Modify your function so that if `resample` parameter is `TRUE`, it samples the *entire* table with replacement. The most suitable function for that is `sample_frac()` from `dplyr`, as you can easily specify the `size` of data as 1 (as many row as in the original table). Or you can use `sample_n()` but here you must specify the desired number of rows explicitly. Don't forget about replacement! Test your updated function by calling it with `resample = TRUE` and checking that you get *different* averages every time.

Do exercise 9.

As with resampling the mean, now you need to repeat this many (1000, 2000) times over. Here, the prior approaches won't do, as they expect a single number (statistic) whereas our function returns a table of them. Our solution would be to use `map()` function from `purrr` library and store these tables in a list that we can turn into a single table via `list_rbind()`. My suggestion would be to first program the `list_of_tables <- map(...)` part alone for just of couple iterations. Then you can check each table inside the list (remember how to use simplifying subsetting?) and if they look good and *different*, you can combine them via `list_rbind()`. Once you are sure that the computation works correctly, you can run it for more iterations to get more samples. Advice, read on `.progress` parameter of the `map()` as it makes waiting for the code to finish a bit less stressful.

Do exercise 10.

## 12.5. CONFIDENCE ABOUT PROPORTION OF RESPONSES FOR LIKERT SCALE189

We are on the final stretch! Your table with bootstrapped counts and proportions contains many samples for each combination of condition and response level (e.g., 1000). Use `quantile()` function to compute lower and upper limits for the 97% confidence interval. Think about which dplyr verbs do you need for this. The table with confidence intervals should look like this (but for variability due to sampling).

Condition	Response	LowerCI	UpperCI
game	Not at all true	0.0000000	0.1791071
game	Hardly true	0.0000000	0.1785714
game	Slightly true	0.0000000	0.2500000
game	Somewhat true	0.0000000	0.1785714
game	Mostly true	0.0000000	0.2500000
game	Almost completely true	0.0000000	0.1071429
game	Very true	0.0000000	0.1071429
experiment	Not at all true	0.0000000	0.2142857
experiment	Hardly true	0.1071429	0.4648214
experiment	Slightly true	0.0000000	0.2500000
experiment	Somewhat true	0.0000000	0.1076786
experiment	Mostly true	0.0000000	0.0000000
experiment	Almost completely true	0.0000000	0.0000000
experiment	Very true	0.0000000	0.0000000

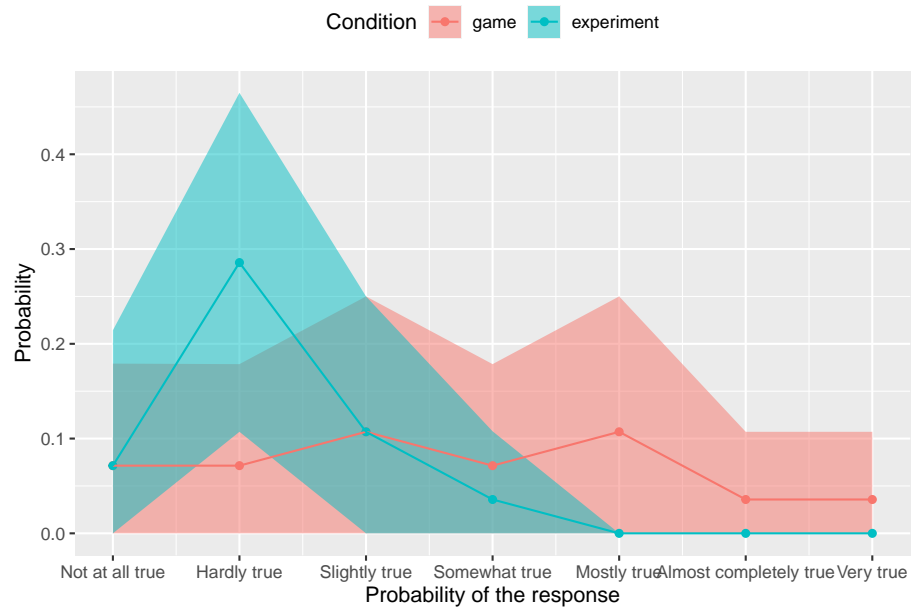
Do exercise 11.

The only thing left to do is to use this in combination with `geom_ribbon()` or `geom_errorbar()` to add 97% CIs to our original plot. Now you have two tables for a single plot and you have two ways to handle this. First, you can join them into a single table. Or, you can pass the table with CIs to the geom itself. In the latter case, you need to use explicit named parameter `data = likert_CI` (or whatever the name of your table is) as geoms in ggplot2 expect aesthetics as the first parameter. Also, your CIs table does not have the `Probability` column that you use as aesthetics for `y` and ggplot2 will complain. The solution is to set `y` to one of the limits, as it is not used in the geom itself and this will make no actual difference<sup>4</sup>.

The final plot should look like this. Note that with so little data our uncertainty about the actual proportion is quite low.

---

<sup>4</sup>But it is still puzzling for me, why would ggplot2 insist on irrelevant aesthetic in cases like these.



Do exercise 12.

The procedure and the plot above are my preferred way of reporting Likert scale data. You can use other approaches but always keep in mind that this is *ordinal* data and should be treated as such, even if you label individual levels with numbers.

## Chapter 13

# (Generalized) Linear regression and Resampling

As I have warned you at the very beginning, this seminar will not teach you statistics. The latter is a big topic in itself and, in my experience, if you know statistics and you know which specific tool you need, figuring out how to use it in R is fairly trivial (just find a package that implements the analysis and read the docs). Conversely, if your knowledge of statistics is approximate, knowing how to call functions will do you little good. The catch about statistical models is that they are very easy to run (even if you implement them by hand from scratch) but they are easy to misuse and very hard to interpret<sup>1</sup>.

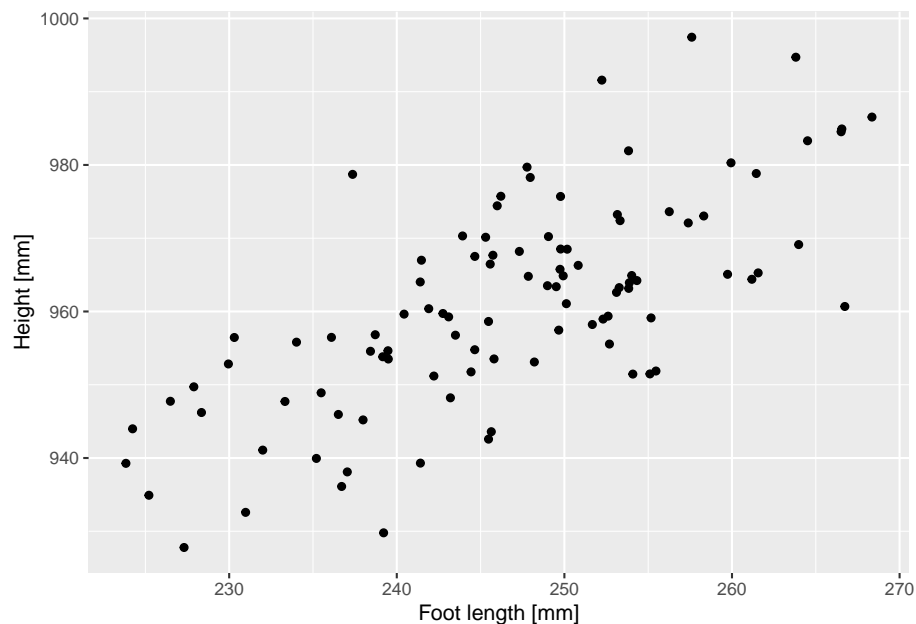
To make things worse, computers and algorithms do not care. In absolute majority of cases, statistical models will happily accept any input you provide, even if it is completely unsuitable, and spit out numbers. Unfortunately, it is on you, not on the computer, to know what you are doing and whether results even make sense. The only solution to this problem: do not spare any effort to learn statistics. Having a solid understanding of a basic regression analysis will help you in figuring out which statistical tools are applicable and, even more importantly, which will definitely misguide you. This is why I will give an general overview with some examples simulations but I will not explain here when and why you should use a particular tool or how to interpret the outputs. Want to know more? Attend my *Bayesian Statistics* seminar or read an excellent Statistical Rethinking by Richard McElreath that the seminar is based on.

---

<sup>1</sup>In the Bayesian Statistics we spend three seminars learning how to understand and interpret a simple linear multiple regression with just two predictors. And the conclusion is that even in this simple case, you are not guaranteed to fully understand it. And if you think that you can easily interpret an interaction term even for two continuous predictors...

### 13.1 Linear regression: simulating data

Our first statistical model will be linear regression. When you experiment with analysis, it is easier to notice if something goes wrong if you already know the answer. Which is why let us simulate the data with a linear relationship in it: overall height versus foot length. A conference paper I have found online suggests that foot length distribution is  $246.726 \pm 10.3434$  mm (mean and standard deviation, assume normal distribution) and the formula for estimate height is  $Height = 710 + 4 \cdot Foot + \epsilon^2$  where  $\epsilon$  (residual error) is normally distributed around the zero with a standard deviation of 10. Generate the data (I used 100 points) putting it into a table with two columns (I called them `FootLength` and `Height`) and plot them to match the figure below. You can set seed to 826 to replicate me exactly.



Do exercise 1.

### 13.2 Linear regression: statistical model

Let us use a linear regression model to fit the data and see how accurately we can estimate both intercept and slope terms. R has a built-in function for linear regression model — `lm()` — that uses a common formula design to specify relationship. This formula approach is widely used in R due to its power and simplicity. The syntax for a full factorial linear model with two predictors is  $y \sim x1 + x2 + x1:x2$  where individual effects are added via  $+$

---

<sup>2</sup>I have rounded off the numbers a little to make it easier to read.



and : mean “interaction” between the variables. There are many additional bells and whistles, such as specifying both main effects and an interaction of the variables via asterisk (same formula can be compressed to  $y \sim x1 * x2$ ) or removing an intercept term (setting it explicitly to zero”  $y \sim 0 + x1 * x2$ ). You can see more detail in the online documentation but statistical packages frequently expand it to accommodate additional features such as random effects. However, pay extra attention to the new syntax as different packages may use different symbols to encode a certain feature or, vice versa, use the same symbol for different features. E.g., | typically means a random effect but I was once mightily confused by a package that used it to denote variance term instead.

Read the documentation for `lm()` and `summary()` functions. Fit the model and print out the full summary for it. Your output should look like this

```
##
## Call:
## lm(formula = Height ~ FootLength, data = height_df)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -24.0149  -7.0855   0.7067   6.0991  26.6808
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept)  725.26822    24.12584   30.062  < 2e-16 ***
## FootLength    0.95535     0.09776    9.772 3.78e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 10.13 on 98 degrees of freedom
## Multiple R-squared:  0.4935, Adjusted R-squared:  0.4884
## F-statistic: 95.49 on 1 and 98 DF,  p-value: 3.783e-16
```

As you can see, our estimate for the intercept term is fairly close  $725 \pm 24$  mm versus 710 mm we used in the formula. However, the foot length slope is small than it should be ( $0.96 \pm 0.09$  versus 4) showing that for such noisy data 100 samples is not enough to reliably estimate it.

Do exercise 2.

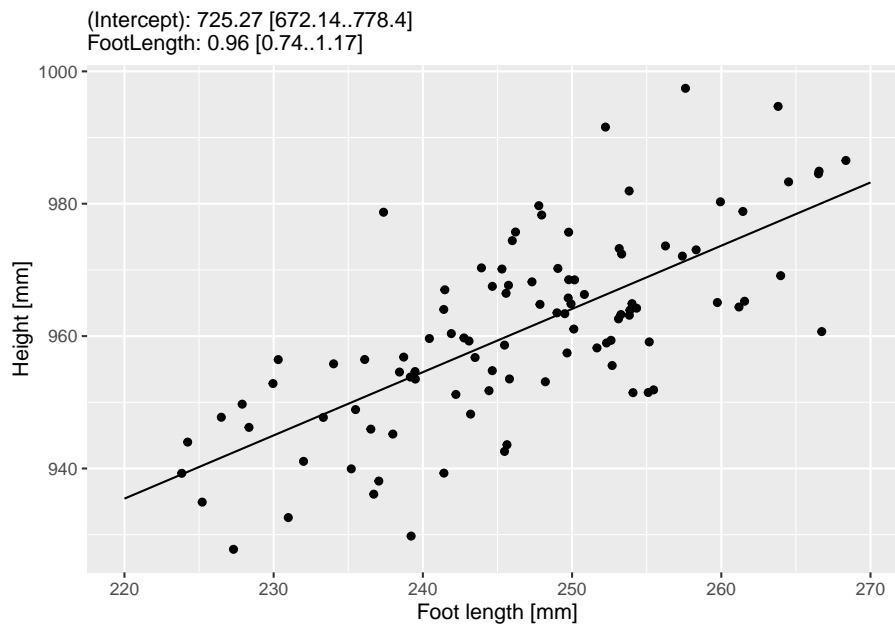
I think it is nice to present information about the model fit alongside the plot, so let us prepare summary about both intercept and slope terms in a format *estimate [lower-97%-CI-limit..upper-97%-CI-limit]*. You can extract estimates themselves via `coef()` function and and their confidence intervals via `confint()`. In both cases, names of the terms are specified either as names of the vector or rownames of matrix. Think how would you handle this. My approach is to convert matrix with confidence intervals to a data frame (converting directly to tibble removes row names that we need later), turn row names into a column

via `rownames__to__column()`, convert to a tibble so I can rename ugly converted column names, add estimates as a new column to the table, and relocate columns for a consistent look. Then, I can combine them into a new variable via string formatting (I prefer glue). You need one(!) pipeline for this. My summary table looks like this

Term	Estimate	LowerCI	UpperCI	Summary
(Intercept)	725.2682157	672.1395797	778.396852	(Intercept): 725.27 [672.14..778.4]
FootLength	0.9553541	0.7400622	1.170646	FootLength: 0.96 [0.74..1.17]

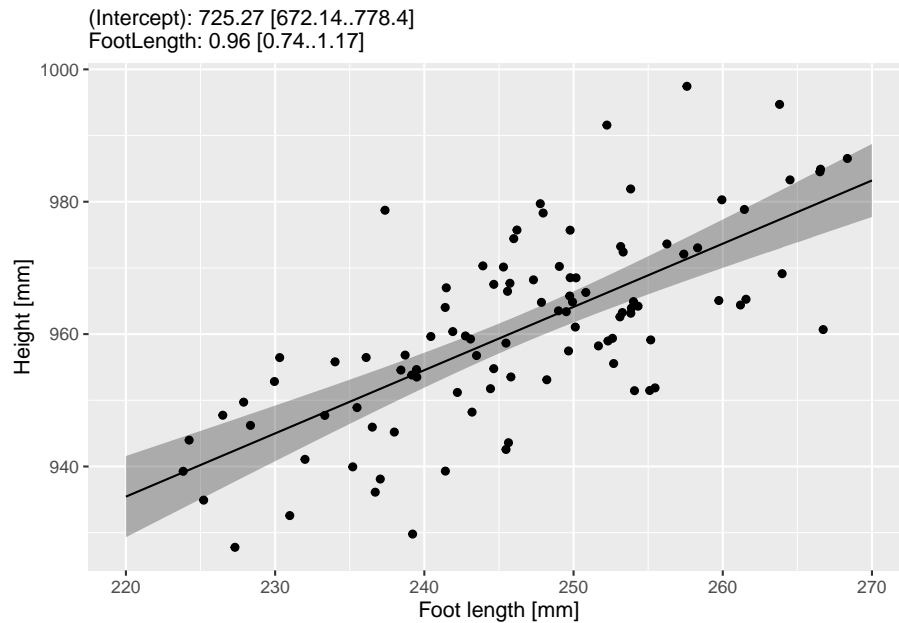
Do exercise 3.

Statistical model is only as good as its predictions, so whenever you fit a statistical model to the data, you should compare its predictions to the data visually. `ggplot2` provides an easy solution to this via `geom_smooth()` that you have met earlier. For didactic purposes, let us use a slightly longer way of generating predictions ourselves and plotting them alongside the data. R provides a simple interface to generating prediction for a fitted model via `predict()` function: you pass a fitted model to it and it will generate prediction for every original data point. However, you can also generate data for data points not present in the data (so-called “counterfactuals”) by passing a table to `newdata` parameter. We will use the latter option. Generate a new table with a single column `FootLength` (or, however you called it in the original data) with a sequence of number going from the lowest to the highest range of our values (from about 220 to 270 in my case) in some regular steps (I picked a step of 1 but think of whether choosing a different one would make a difference). Pass this new data as `newdata` to `predict`, store predictions in the `Height` column (now the structure of your table with predictions matches that with real data) and use it with `geom_line()`. Here’s how my plot looks like.



Do exercise 4.

We can see the trend, but when working with statistical models, it is important to understand uncertainty of its predictions. For this, we need to plot not just the prediction for each foot length but also the confidence interval for the prediction. First, we will do it an easy way as `predict()` function has options for that as well via its `interval` parameter (we want `"confidence"`). Use it to generate 97% confidence interval for each foot length you generated and plot it as a `geom_ribbon()`.



Do exercise 5.

### 13.3 Linear regression: bootstrapping predictions

Let us replicate these results but use bootstrap approach. One iteration consists of:

1. Randomly sample original data table *with replacement*.
2. Fit a linear model to that data.
3. Generate predictions for the interval, the same way we did above, so that you end up with a table of **FootLength** (going from 220 to 270) and (predicted) **Height**.

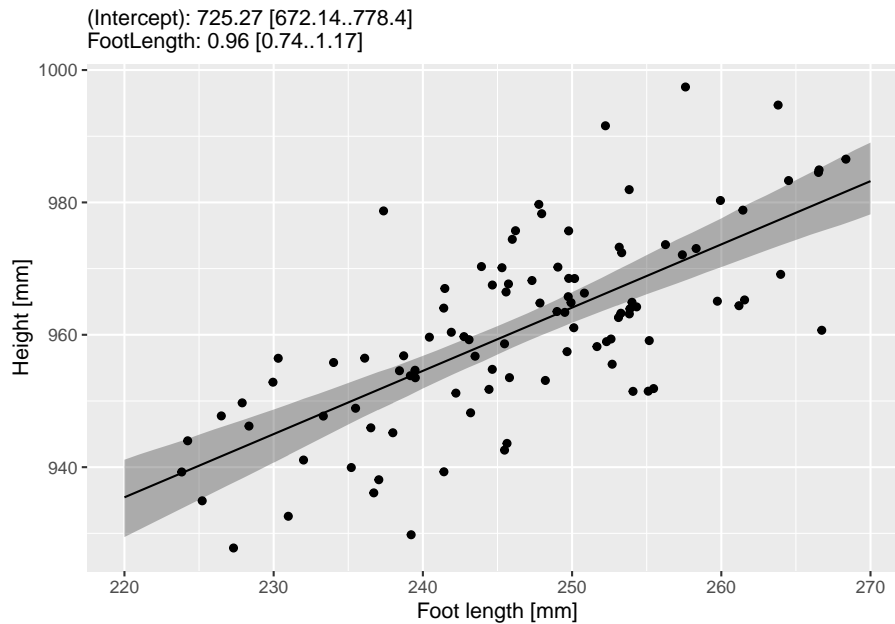
Write the code and put it into a function (think about which parameters it would need). Test it by running it the function a few times. Values for which column should stay the same or change?

Do exercise 6.

Once you have this function, things are easy. All you need is to follow the same algorithm as for computing and visualizing confidence intervals for the Likert scale:

1. Call function multiple times recording the predictions (think `map()` and `list_rbind()`)

2. Compute 97% confidence interval for each foot length via quantiles
3. Plot them as `geom_ribbon()` as before.



Do exercise 7.

## 13.4 Logistic regression: simulating data

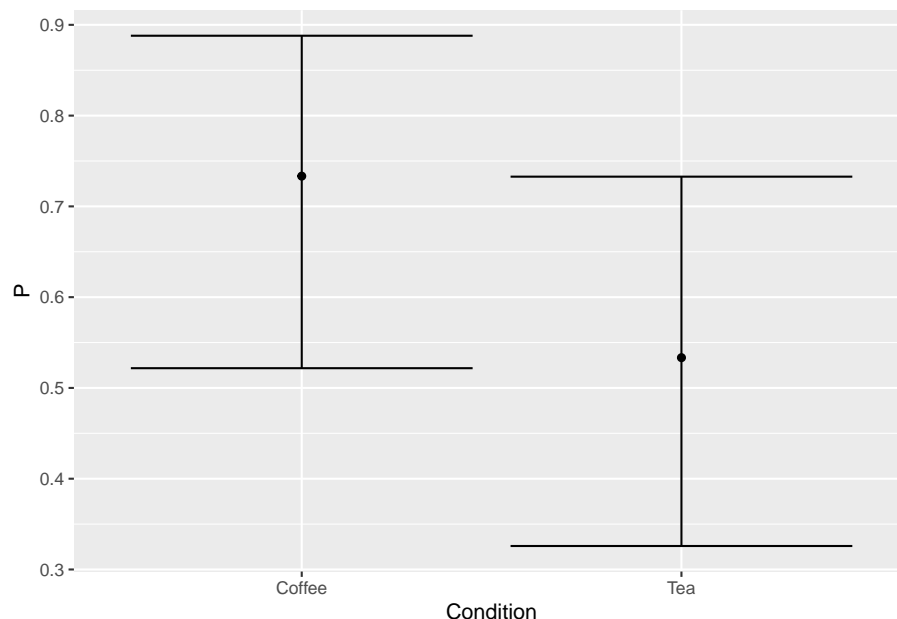
Let us practice more but this time we will using binomial data. Let us assume that we measure success in playing video games for people drank tea versus coffee (I have no idea if there is any effect at all, you can use liquids of your liking for this example). Let us assume that we have measure thirty participants in each group and average probability of success was 0.4 for tea and 0.7 for coffee<sup>3</sup> groups. You very much know the drill by now, so use `rbinom()` (you want twenty 0/1 values, so figure out which `size` and which `n` you need) to generate data for each condition, put it into a single table with 60 rows (`bind_rows()` might be useful) and two columns (`Condition` and `Success`). Your table should look similar to this (my seed is 12987)

Condition	Success
Tea	0
Tea	1
Tea	1
Tea	0
Tea	1

<sup>3</sup>Yes, I am a coffee person.

Do exercise 8.

Now, let us visualize data. You need to compute average and 97% confidence interval for each condition. The average is easy (divide total number of successes for condition by total number of participants) but confidence interval is trickier. Luckily for us, package `binom` has us covered. It implements multiple methods for computing it. I used `binom.exact()` (use `?binom.exact` in the console to read the manual, once you loaded the library). Your plot should look like this (or similar, if you did not set seed). Note that our mean probability for **Tea** condition is higher than we designed it (sampling variation!) and confidence intervals are asymmetric. The latter is easier to see for the coffee condition (redo it with probability of success 0.9 to make it more apparent) and is common for data on limited interval.



Do exercise 9.

### 13.5 Logistic regression: fitting data

Let us fit the data using generalized linear models — `glm` — with `"binomial"` family. That latter bit and the name of the function are the only new bits, as formula works the same way as in `lm()`. Print out the summary and it should look like this.

```
##
## Call:
## glm(formula = Success ~ Condition, family = "binomial", data = game_df)
```

```
##
## Coefficients:
##              Estimate Std. Error z value Pr(>|z|)
## (Intercept)   1.0116     0.4129   2.450   0.0143 *
## ConditionTea -0.8781     0.5517  -1.592   0.1115
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##      Null deviance: 78.859  on 59  degrees of freedom
## Residual deviance: 76.250  on 58  degrees of freedom
## AIC: 80.25
##
## Number of Fisher Scoring iterations: 4
```

Do exercise 10.

Note that this is *logistic regression*, so the estimates need are harder to interpret. Slope term for condition **Tea** is in the units of log-odds and we see that it is negative, meaning that model predict fewer successes in tea than in coffee group. Intercept is trickier as you need inverted logit function (for example, implemented as `inv.logit()` in *boot* package) to convert it to probabilities. Here, 0 corresponds to probability of 0.5, so 1 is somewhere above that. Personally, I find these units very confusing, so to make sense of it we need to use estimates (`coef()` function will work here as well) to compute *scores* for each condition and then translate them to probabilities via `inv.logit()`. Coffee is our “baseline” group (simple alphabetically), so  $\text{logit}(\text{Coffee}) = \text{Intercept}$  and  $\text{logit}(\text{Tea}) = \text{Intercept} + \text{ConditionTea}$ . In statistics, link function is applied to the left side, but we apply its inverse to the right side. I just wanted to show you this notation, so you would recognize it the next time you see it.

Your code should generate a *table* with two columns (**Condition** and **P** for Probability). It should look like this

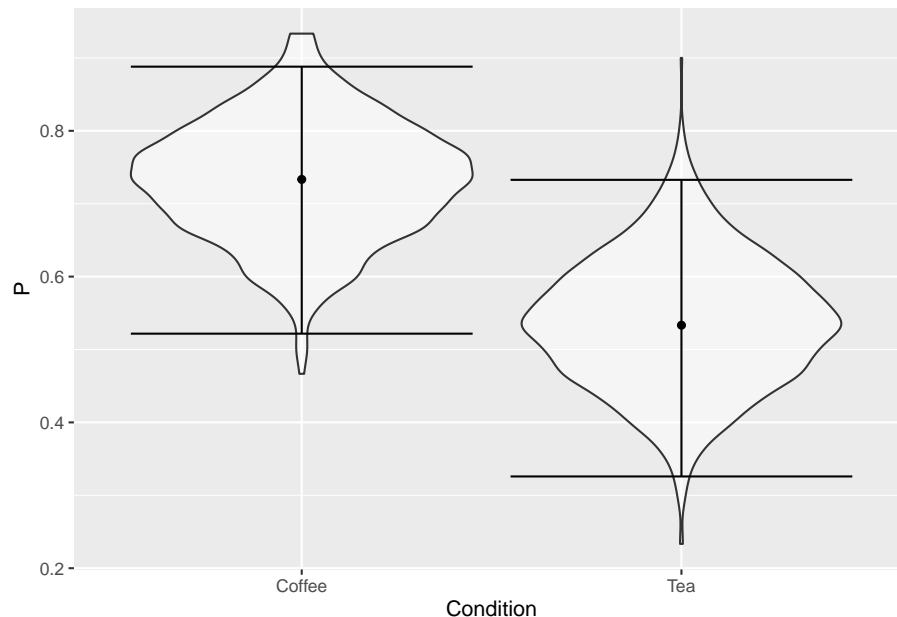
```
## # A tibble: 2 x 2
##   Condition      P
##   <chr>      <dbl>
## 1 Coffee    0.733
## 2 Tea      0.533
```

Do exercise 11.

## 13.6 Logistic regression: bootstrapping uncertainty

Let us bootstrap *predicted* probability of success following the template we used already twice but with a slight twist. Write a function (first write the inside code, make sure it works, then turn it into a function) that samples our data, fits the model, generates and returns model with prediction. The only twist is that we will sample each condition separately. `sample_frac()` but you will need to group data by condition before that. Also, pass iteration index (the one you are purring over) as a parameter to the function and store in a separate column **Iteration**. This will allows us to identify individual samples, making it easier to compute the difference between the conditions later on.

Repeat this computation 1000 times and you will end up with a table with two columns (**Condition** and **P**) and 2000 rows (1000 per condition). Instead of computing aggregate information, visualize distributions using `geom_violin()`. Here's how the plot should look like.

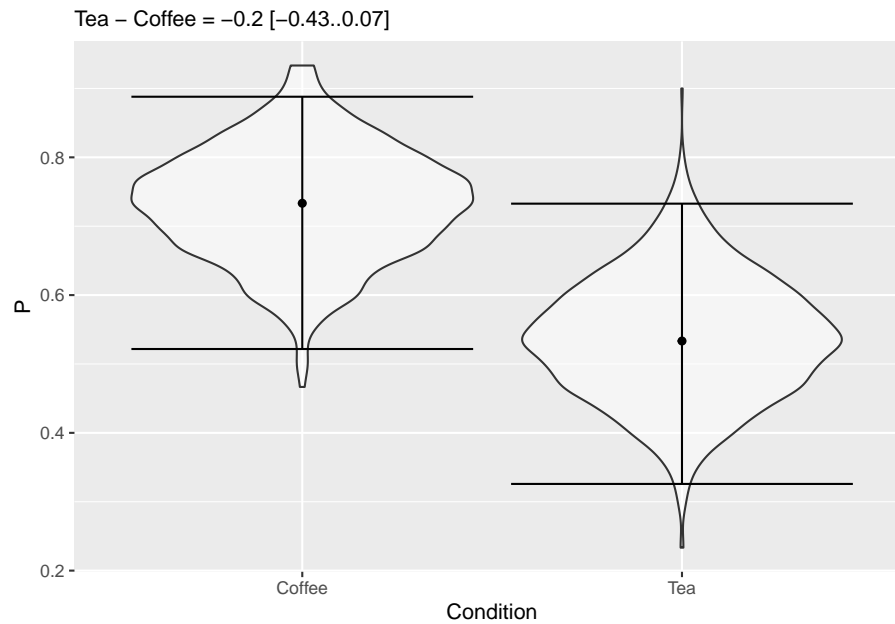


Do exercise 12.

As a final step, let us compute average and 97% confidence interval for the *difference* between the conditions. You have the samples but in a long format, so you need to make table wide. The only catch is that pivot function will be confused by the absence of ID rows, so you need to create an extra “RowIndex” column first, putting in a unique row index going from 1 to the number of rows. So you will end up with three columns: **RowIndex**, **Coffee**, and **Tea**. Compute a



new column with difference between tea and coffee, compute and nicely format the statistics putting into the figure caption. Hint: you can pull the difference column out of the table to make things easier.



Do exercise 13.