

Data Processing and Analysis in *R* (2023)

Oliver Lindemann

2024-10-09

Contents

Overview	4
1. Introduction	6
1.1. Why learning <i>R</i> ?	6
1.2. What do I learn in this course?	6
1.3. What is RStudio?	6
1.4. Code examples	8
1.5. Trial-and-error learning	9
2. Finding Help	10
I. Basics	12
3. Getting started	12
3.1. Installing <i>R</i> and <i>RStudio</i> on your PC	12
3.2. Creating a Project	12
4. Basics of Programming in R	13
4.1. Comments in Program Code	13
4.2. Variables and Objects	14
4.3. Basic operations	16
4.4. Non-numerical data types	17
4.5. Comparing values	19
4.6. Functions	19
II. Data Structures	25
5. Atomic Vectors	25
5.1. Number sequences	26

5.2. Manipulating vectors	28
5.3. Subsetting Vectors	30
5.4. Named vectors	33
6. Lists	34
6.1. Subsetting lists	35
6.2. Named lists	35
6.3. The \$ operator	36
7. Data Frames	37
7.1. Subsetting	40
7.2. Lists and data frames	42
7.3. Summary: Data structures in R	43
8. Packages	43
8.1. Installing a package	44
8.2. Using a package	45
8.3. Getting help for packages	46
9. Importing Data	47
9.1. Import via RStudio interface	47
9.2. Import via Console or Script	48
III. Data Wrangling	50
10. Tidyverse package collection	50
10.1. Tibble is the new Data Frame	52
10.2. Naming convention (“_” or “.”)	52
11. Transforming Data	53
11.1. Selecting rows: <code>Filter()</code>	54
11.2. Sorting rows: <code>arrange()</code>	56
11.3. Selecting & Arranging Variables: <code>select()</code>	58
11.4. Add new variables: <code>mutate()</code>	61
12. Summarizing & Aggregating Data	63
12.1. Aggregating data with <code>summarise()</code>	63
12.2. Grouping data by <code>group_by()</code>	64
12.3. Summary	67
13. Structuring Tabular Data	68
13.1. Tidy data	68
13.2. Long and Wide Data	70

IV. Data Processing & Visualizing	73
14. Missing Values	73
14.1. NA: Not Available	73
14.2. Missings in data frames	75
15. Factors	78
15.1. Creating factors	78
15.2. Advantages of factors	79
16. Pipes	82
16.1. The pipe operator: %>%	83
16.2. Motivation for pipes	83
16.3. Building a processing pipeline	84
17. Visualizing Data	86
17.1. Plots with base R	88
17.2. ggplot2: Plotting in Tidyverse	90
17.3. Geoms or plot types	93
A. Suggested Readings	107
A.1. Cheat sheets	107
B. Interesting Packages	107
RMarkdown	108
Shiny Apps	108
C. Compatibility Issues and Workarounds	108
C.1. Can't install <i>Tidyverse</i> on MacOS or Linux	108
D. Course Exam	109
Examination procedure	109
What is allowed to use and what will be consider as an attempt of fraud? . . .	110

Overview



This course is an introduction in basic data science skills for psychologists. Students will learn to develop data analysis scripts in the programming language R. This course is also the basis for later courses using R for inferential statistics.

R is a free software environment for statistical computing and data visualization that has become in the last couple of years the de-facto-standard tool for advanced statistics and reproducible research in psychology and social sciences.

Learning Objectives

At the end of the course the student will be able to develop scripts in the programming language R. The acquired data science skills comprise

- basics of programming in R
- reading data files in different formats
- pre-processing data such as data wrangling, transformation, aggregation and filtering
- data visualization
- descriptive statistics

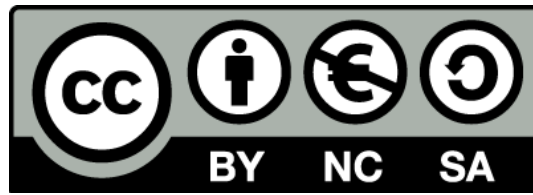
Course requirements

For information about the structure of the course, the weekly assignments, the course exam and a schedule with the lectures and the practical skills group meetings, please visit the Canvas pages of this block.

Manual

This manual guides you through the *R* course and explains all concepts you need to know and points to relevant online resources.

This tutorial is written by [Oliver Lindemann](#), Erasmus University Rotterdam, using [RMarkdown](#) and [bookdown](#). It's **free to use** and licensed under the [Creative Commons BY-NC-SA 4.0](#) License. This manual is influenced by the two tutorials [Advanced R](#) by Hadley Wickham and [Data Skills for Reproducible Science](#) of the University of Glasgow.



1. Introduction

1.1. Why learning *R*?

R is a computer language designed by statisticians. I think it is fair to say that *R* became in the recent year the “lingua franca” of data science and statistics. *R* is an open source software and an extremely powerful tool for any kind of task in scientific computing. It is not only great for data transformation, data visualization and statistical analyses. It has also the capacity to deal with big data or to do text mining and analysis of internet pages and to do machine learning and more.

R has a very large and active open source user community. For that reason, it is very easy to find online help if you have questions and if you need support for analysis problems that are difficult to handle (see section 2). Moreover, the language is constantly improving and hundreds of data scientists have developed extensions for *R* (so-called *packages*) that implement various classical and novel statistical methods. You can do indeed any statistical analysis you can think of in *R*.

Please keep in mind: *R*’s learning curve is steep partially due to its syntax and the rules of the programming language that you have to learn. *R* reads less like English, and in general is more difficult for beginners to wrap their heads around. However, once you got familiar with *R*, it makes processing of data much easier as compared to tools such as *Excel* or *SPSS*.

1.2. What do I learn in this course?

This course is merely a first introduction to *R*. It does not require any pre-knowledge in computer programming (For those of you who have already experienced with any kind of programming language, I guess it’s not a big deal to learn *R*).

This course will teach you the minimum about computer programming that you need to practice data science and get started with doing data analyses in *R*. We will use *R* (and RStudio) predominantly as an interactive environment for data science that enables you to read, process and analyse data. If you want to develop your *R* skills beyond basic usage and if you want to use *R* as a programming language to develop your own programs and/or packages, I point you to the advanced literature in the [appendix A](#).

1.3. What is RStudio?

As said before, *R* is just a programming language, which executes commands and produces certain output. To interact with the programming language, we need a program that enables us to enter the commands and edit scripts that should be executed by *R* scripts.

Moreover, we need a program that display the output of our *R* programs, that is, to show results of our calculation or display graphics that we made with *R*.

RStudio is a program which does this (and more) for us. RStudio is a so-called integrated development environment (*IDE*) for *R* programming. It is the interface that we use in this course to interact with the programming environment *R* and it looks like this

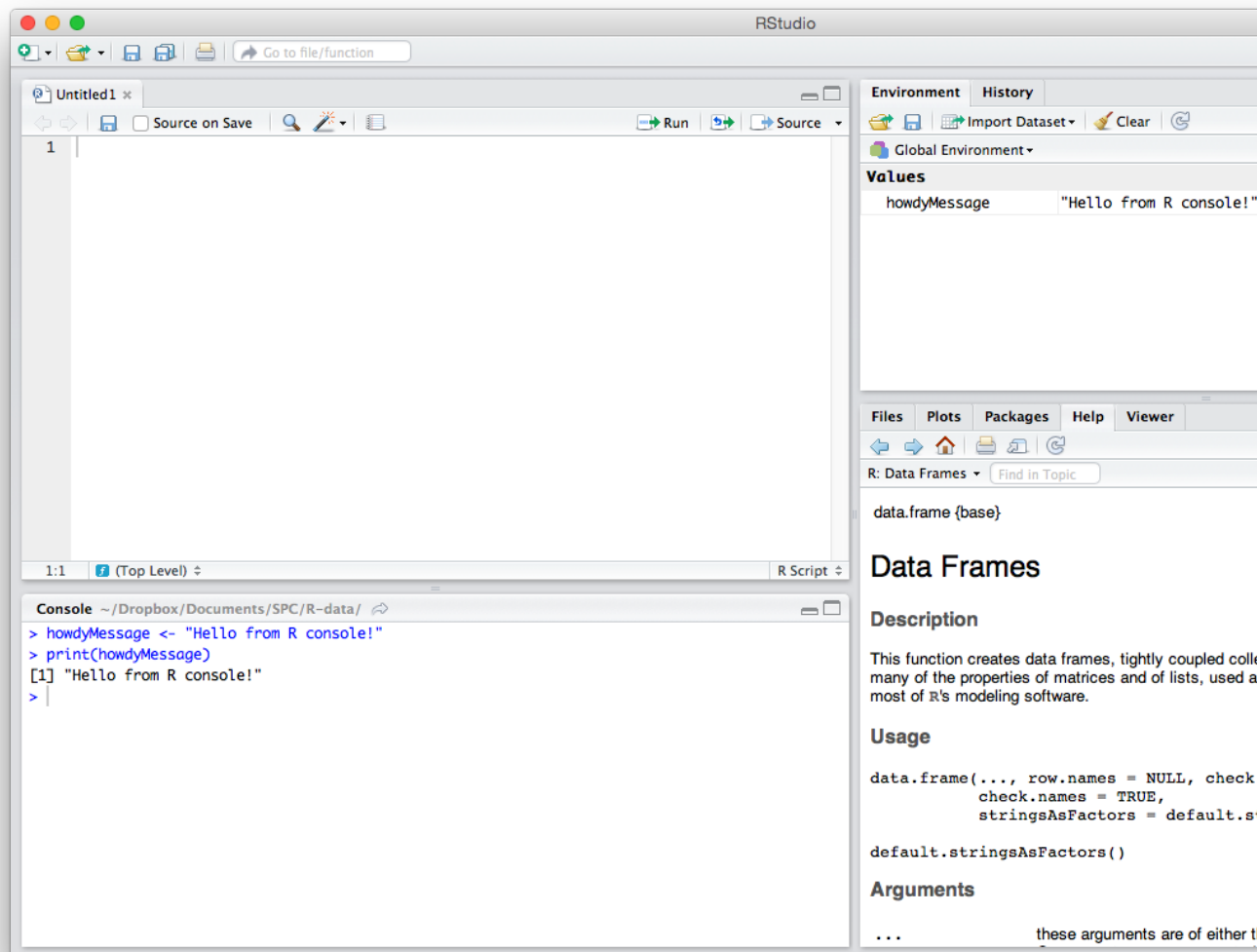


Figure 1: RStudio Window

1.3.1. The RStudio Interface

If you open RStudio, you get a window which look like Figure 1. RStudio has

- a console that you will use to enter commands and try out code (appearing as the **bottom left** window)
- there is a script editor on the **top left**
- a window with the “Environment” tab (**top right** window), which shows functions and objects you have created and which are currently in memory, and
- a window that shows plots, files packages, and help documentation (**bottom right**)

You will learn more how to use this program during this course.



If you have already programming experiences and a strong preference for a certain coding editor or IDE, you may use your preferred coding environment and integrate the *R* command line software to run your scripts.

Using coding editors is never as comfortable as working with the specialized tool *RStudio*. However, some nerds want the puristic command line experiences.

1.4. Code examples

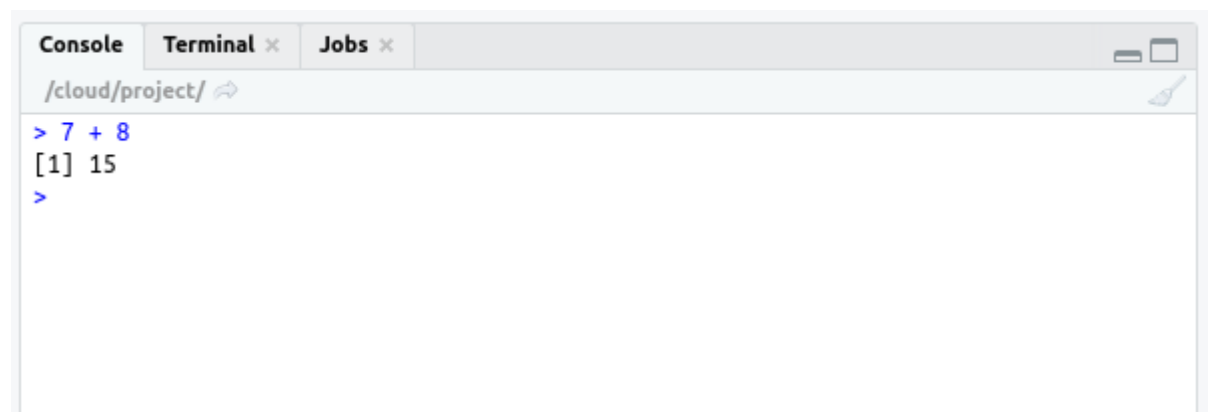
The manual comprises many code examples. A code example always looks like this:

```
7 + 8
```

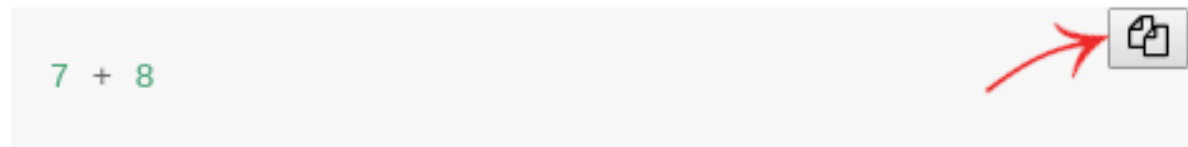
The output that *R* returns in the console after processing the command is indicated in the manual with **##** and looks like this:

```
## [1] 15
```

It is suggest to execute the code examples in your own *R*-console and see what the results are. To do so, copy & paste the code in the console and press enter. *R* executes the code and give you feedback. In RStudio it like this:



Hint: You can copy the code examples from the online manual by clicking on the ‘copy-button’, which appears if you move with the mouse cursor to right-top corner of the code example field.



1.4.1. Text boxes

There are different types of text boxes:



Important information.



Additional information for **advanced learners**. This is only for nerds and not relevant for most students. If you want, can simply ignore it.



This icon indicates an **exercise**. Solutions to some exercise problems are provided and help to check whether you understood the learning material. In the online version the solutions are hidden and will be displayed only on request.

Solution

```
# This is a solution to an exercise problem.
```

1.5. Trial-and-error learning

Trial-and-error is very essential for learning to program. I therefore suggest: *Play around with the code examples and see happens if you change them.*

You will face many error and warning messages as a consequence. But that's fine! Learning to understand error messages is, in my view, the key to learn to program. Thus, read the error messages always carefully (instead of freaking out) and try to figure out what the origin of this underlying error is. This is not always easy and finding a bug in a program code (called *debugging*) is often very time-consuming. However, it is an important (if not the most important) skill for any data scientist.

Don't worry, if you have the feeling at the beginning that you spend most of your time with debugging—you have to know, is quite normal.

2. Finding Help

Learning to programming means trying things out, change stuff in the code and see what happens. Unfortunately, try-and-error is the only way to become a good data analyst or programmer.

Please note, you will face **very often** error messages! Don't get scared, **read the error messages carefully** and try to understand what the cause of the problem is.

There is another type of mistake: Sometimes *R* commands or scripts do not show an error message, but if you look carefully you notice that the *R* doesn't do what you want or reveals very strange and impossible results. Checks therefore the effects of your steps always very carefully. This is can be sometimes a trick situation. Learning to find the mistake (i.e. the bug) in your code can become very hard. That is, finding and solving this problems in the script (called “debugging”) is a key skill that you need to develop throughout this course.

1. If you're struggling how a function works, remember the `?function` command and use this built-in help documentation of *R* and RStudio (see section 4.6.2).
2. Look at the [suggested readings and online learning resources](#) in the Appendix A.
3. Note, the internet is your best friend while learning to programming (and afterwards).
 - Sometimes to helps to find pieces of code of other users online to “*copy & paste*” and “*change & see what happens*”. For this you can use in a normal web search engine (e.g. [DuckDuckGo](#) or [Google](#) or other).
 - If you encounter an error message and you do not understand the reasons for this error, it is very help copy & paste the error message search for it online. You are probably not the first one who had this problem
 - It is also highly recommended use question-and-answer-sites for programmers. The most prominent examples is here [stackoverflow.com](#). Just enter [r] followed by your question and see, if other user have already asked a similar Question.
4. [Cheat sheets](#) are nice overviews of *R* functions and helpful, if you can't remember correctly how to use the functions. They are however not self-explaining.



Difficulties with the exercises and assignments

If you are having difficulties with the exercises or assignments and you can not solve the problem yourself, ask your question during the **tutorial sessions**. The

idea of the tutorial sessions is to provide hands-on support you need to do the exercises and answer all your questions. Make use of this sessions.

You might also use the **Canvas discussions** of this course and ask others students, which likely stumble over the same issue. If many students use this course discussion of exchange, I can become a very productive tool.

Part I.

Basics

3. Getting started

To get started to working with *R* on your own PC or laptop, you have to install *R* and *RStudio*. Details about how to install it on your computer can be found below and in one of the instruction videos.

3.1. Installing *R* and *RStudio* on your PC



Install *R*

Depending on the operating system (OS), download the latest release of [R for Windows](#) or [R for MacOSX or Linux](#) and install it. If you use Linux, you can install the package `r-base` via your package manager.



Download *RStudio* and install it.

Use the correct installer for your operation system from [RStudio homepage](#).

If you face problems with the installation you might find [Step-by-Step Guide for Installing R and R Studio](#) useful. However, there are also several other of tutorials online.

3.2. Creating a Project

To get started with your own programming in *RStudio* and with this course you need to create a project that you use for all your exercises during this course.



Open *RStudio* and click in the menu **File - New Project, New Directory - New Project**, select a directory name (e.g. 'R-practical') and select a subdirectory, where you want to save the project. Click **Create Project**.

Now, you are ready to started working with *R*...

Even though you can program and edit *R* files without making a *RStudio* project, I **strongly recommend** to work always within a project. That is, the first you should do, when opening *RStudio* is the check, if you are in the right project or make new project, if

you start a new data analysis. By the way, `_RStudio_` automatically opens the last used project.

Why should I use projects? When working with *R*, you are running your program code from a certain **working directory** (You can change the working directory [like this](#)). From this directory you access all your data files and *R* scripts. It's therefore crucial that you have files and data in the correct folder structure and that you are in the right directory. A project saves the last working directory and the state of your previous work. That is very handy.

4. Basics of Programming in R

Learning to work with *R* requires a little knowledge about basic concepts of computer programming. This is maybe a bit technical at the beginning. However, having an insight in this basics makes it later much easy to deal with real data analysis in *R* later.

Thus, the section covers the basics of the *R* programming language. If you are already familiar with any another programming language, I guess these things are not new for you.

4.1. Comments in Program Code

Before we have a look at the first *R* code examples, you should know that *R* ignores everything that follows a hashtag `#`. That is, text after a `#` will not be executed. Hashtags are used to add comments to *R*-code. The comments help to document and explains the code and make it thus more readable for humans. Comments in *R* code are printed in italic font and a different color by *RStudio* (and in the manual). See this example:

```
# Example of code with comments  
x = a / b      # dividing a by b and storing the result in x
```

That is, if you try out the code examples, you can simple ignore the comment.



If you later develop your own analyses scripts in *R*, it is **strongly** suggested to make use of comments. Only well documented code is readable code. Only if you comment your code, you will be able to remember after a few days why you did what and how your codes works.

4.2. Variables and Objects

Everything that is stored in *R* is called an **object**. The most basic example for an object is an numerical **variable**. For instance,

```
a <- 9
b <- 2724.54
third_variable <- 71
```

In this example, the numerical 9, 2724.54 and 71 are stored in the variables with the names `a`, `b` and `third_variable`. If you want to print out the content of a certain object or variable, you can use the `print()`-command together with the variable.

```
print(b)
```

```
## [1] 2724.54
```

... or even simpler, just enter the variable name in the console

```
third_variable
```

```
## [1] 71
```

This [video](#) also explains objects in *R*.

4.2.1. Variable Environment

All variables are sorted in the data workspace or environment. In RStudio, you can find the current content of the workspace with all variables the right top panel called “environment”.

Alternatively, the command `ls()` lists all objects variables in the R console.

```
ls()
```

```
## [1] "a"           "b"           "hide"        "is_latex"
## [5] "third_variable" "unhide"
```

4.2.2. Equal Sign Assignments (=)

As you have seen above, to assign a value to an object (i.e., variable) with the left arrow `<-`. In RStudio, the shortcut `alt & -` is very handy for quickly typing this sign. Check it out.

Instead of typing an left arrow `<-` you can use the equal sign `=`. This two comments have the identical effect:

```
x <- 13
x = 13
```

Many data scientists find the equal-sign syntax easier to type and more natural to read. If you also prefer using `=` instead of `<-`, just do it. It's not a problem.

You just need to know, that the coding style guidelines in the *R* community suggest to use the `<-` syntax. (see e.g. [here](#) for reasons). Most examples and documentations that you find online use therefore the arrow as the assignment operator.

To avoid any confusion, the current manual uses the `<-` operator.

4.2.3. Left and Right Hand Side Assignments

In *R*, there are two different ways to assign a value to a variable; you can either assign from left to right or from right to left in *R*. That is, instead of using `<-` or `=` to assign a value to a variable (left-hand-side assignment), you can also work with so-called the right-hand-side assignments (`->`). These comments illustrate the principle:

```
x <- 13      # left-hand-side assignment
13 -> x      # right-hand-side assignment

a <- b       # a receives the value of b
a -> b       # b receives the value of a
```

Left-hand-side assignments are most common among users, because they are similar to what we are used to in mathematical equations. You will see later in the course (when working with **pipes**), why right-hand-side assignment can be very elegant and efficient for data science.



What happens in the command below? What are the changes variable environment?

```
E = U = R = 2
```

Solution

```
# 1. The value 2 is assigned to R.  
# 2. U is assigned to R  
# 3. E is assigned to U  
# That is, you have three variables with the value 2
```

4.3. Basic operations

You use *R* as a calculator and perform basic operations on numbers and variables numbers. Just enter the following examples the console and check the results:

```
a <- 8  
  
12 + 6  
a * 2  
210 / 3  
(a - 3) / 4  
4**3      # 4 to the power of 5
```

Of course, the result of a calculations can be stored in new variables, for instance:

```
x <- a * 7  
x
```

```
## [1] 63
```

Look at the following example, which increases the value of the variable `x` by 1. To do so, it stores the result of the operation `x + 1` again in `x`:

```
x <- 9  
x <- x + 1  
x
```

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



See the code example below. If `r` represents the radius of a circle, what is `x` ?

```
r = 19  
pi*(r**2) -> x
```

Solution


```
# The area, which is pi times radius squared.
```

4.4. Non-numerical data types

4.4.1. Character

Besides numeric values as in the example above, *R* can store text (called **character**) in variables.

```
txt <- "Erasmus"
```

You can see the new variable in the current workspace (see “environment”-panel).

Certain operations can be only performed with certain data types. For instance, it does not make sense to subtract or add to text string, right? See what happens:

```
a <- "Hello"  
b <- "World"
```

```
a + b
```

```
## Error in a + b : non-numeric argument to binary operator
```

If you want to concatenate two strings you have to use the function **paste**:

```
paste(a, b)
```

```
## [1] "Hello World"
```



R can sometimes convert one atomic data type into another data type. Function that convert data types start in *R* with “**as.**”. For instance,

- **as.character()** converts as numbers into a string
- **as.numeric()** converts a character into number. see section [4.4.3](#)

4.4.2. Logicals

Another often reoccurring data type is call *logical* (in other programming languages it's other call *boolean*). A logical can be only **TRUE** or **FALSE** (typed in capitals without quotation marks); it is in other words a binary variable.

Logicals are very important, because are returned by *R* if you make a comparison of the numbers (see section 4.5.)

Negation: ! In *R*, **!** is used for the negation of logicals and reads like “*is not*”. It converts, in other words, a **TRUE** to **FALSE** and **FALSE** to **TRUE**:

```
!TRUE    # reads like "is not true"
```

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

```
!FALSE    # reads like "is not false"
```

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

4.4.3. Combining different data types

Please note, that you can preform the same operation that not combing a numerical and text variable is, of course, not possible. Try what happens if you type:

```
a <- 4
b <- "12"
a * b
```

The third line results in an error, because **a** is a number and **b** is a text object. Combining this different data types is not possible, because *R* does not know what to do.



Why does this work?

```
a * as.numeric(b)
```

Solution

```
# b is a string, but it will be converted to a number (by the function
# as.numeric). Multiplying two number is, of course, a valid operation.
```

rite the R code for the following equation:

4.5. Comparing values

You can compare numerical variables using comparators. For example,

```
a <- 27
b <- 30
a > b
```

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

You see that this expression is incorrect, because $27 < 30$.

Further comparisons are:

```
a < b
a == b    # Are a and b are equal?
a != b    # Is a unequal to b?
```

Note that *R* needs two equal signs `==` for an “is-equal-comparison”. One equal sign is interpreted by *R* as an assignment, similar to `<-` (see section 4.2.2).

4.6. Functions

As other programming languages, *R* has **functions** to perform a specific tasks in a program, for instance, to process or manipulate data. A function “returns” a results of the command or the processing. For instance, the function `ls()` lists current variables in the workspace.



To execute a function, you always have call the function name **followed by parentheses**.

Forgetting the parentheses results in strange behaviour. (That is, *R* displays the definition of the function, if you just entering the function name.)

4.6.1. Arguments

Often, functions take one or more arguments. The arguments are for instance data to be processed or certain information that are required to execute the task. For instance,

```
log(64)
```

```
## [1] 4.158883
```

returns the logarithm natural of 13, $\log_e(13)$, and prints it out in the RStudio console. You can also save the result directly in another variable, because you want to do some further processing later with it or you can combine functions. For example:

```
x <- log(64)
```

See also [this video explaining functions in R](#).

You can also feed the result of a function directly into another function. For example:

```
sqrt(log(64))
```

`sqrt()` is the command for the square root. This nesting of functions and expressions makes it the implement mathematical expression in R very straightforward. For instance, is



Write the R code for the following equation:

$$x = \sqrt{15^{\frac{\log_e(12)}{1 + \sin(4.5\pi)}}}$$

Hint: `sin()` is function for sinus. The variable `pi` is always predefine in R, even if you can not see it in the environment panel.

Solution

```
x <- sqrt(15*log(12) / 1 + sin(4.5*pi))
# Sorry, I know the example is not a very meaningful.
```

Importantly: Functions expect a specific data type as argument. If the type of the argument is not correct, you will receive an error. For instance, the function `sin()` (the sinus) expects a number not a text string. Look what happens:

```
x = "1987"
sin(x)
```

Required and optional arguments

Some arguments of functions are required (that is, you have to specify this them) and others are optional. Optional arguments will often use a default value (normally specified in the help documentation), if you do not enter any value.

For instance, if you just call `log(64)` you get the natural logarithm $\log_e(64)$. If you want to have the logarithm to a different base, e.g. \log_2 or \log_{10} , you have to specify the optional argument `base` of the `log` function:

```
log(64, base=2)
```

```
## [1] 6
```

```
log(64, base=10)
```

```
## [1] 1.80618
```

4.6.2. Built-in help for functions

R provides hundreds functions. As you will see late in section 8, you can install additional packages for specific tasks, which provide basically collections of extra functions. That is, the amount of functions that you can use in *R* is almost endless.

But how do you know what a specific functions is doing and which arguments you can or have to define to use it correctly. Fortunately, *R* has a **built-in help system**. You can look up all the arguments that a function takes by using the help documentation by using the format “`?function`”.

As an example, let’s look at the help documentation for the function `rnorm()` which randomly generates a set of numbers with a normal distribution.

4.6.3. Example

Display the documentation of the function `rnorm` by typing

```
?rnom
```

The help documentation should appear in the bottom right panel. In the usage section of the documentation, we see that `rnorm()` takes the following form:

- `rnorm(n, mean = 0, sd = 1)`

In the arguments section, there are explanations for each of the arguments. `n` is the number of random numbers we want to create, `mean` is the mean of the data points we will create and `sd` is the standard deviation of the set. In the details section it notes that if no values are entered for `mean` and `sd` it will use a default of 0 and 1 for these values. Because there is no default value for `n` it must be specified otherwise the code won't run.

Let's try an example and just change the required argument `n` to ask *R* to produce 5 random numbers.

```
rmnorm(n = 5)
```

```
## [1] 1.34598307 -1.93423140 -0.04383764 0.32993987 1.15514765
```

The random numbers have a mean of 0 and an SD of 1. Of course, since these are random numbers, you have probably got different numbers. Run the function again and you will see that you get each time different "random" numbers.

Now we can change the additional arguments to produce a different set of numbers.

```
rmnorm(n = 5, mean = 100, sd = 15)
```

```
## [1] 104.02543 112.61202 85.72245 114.08902 100.46111
```

This time *R* has still produced 5 random numbers, but now this set of numbers has a mean of 100 and a standard deviation of 15 as specified (this is the distribution of the scores in many IQ tests). Always remember to use the help documentation to help you understand what arguments a function requires.

4.6.4. Argument names

In the above examples, we have written out the argument names in our code (e.g., `n`, `mean`, `sd`), however, this is not strictly necessary. The following two lines of code would both produce the same result (although each time you run `rmnorm()` it will produce a slightly different set of numbers, because it's random, but they would still have the same mean and SD):

```
rmnorm(n = 7, mean = 10, sd = 2)  
rmnorm(7, 10, 2)
```

Importantly, if you do not write out the argument names, R will use the default order of arguments, that is for `rnorm` it will assume that the first number you enter is `n`. the second number is `mean` and the third number is `sd`.

If you write out the argument names then you can write the arguments in whatever order you like:

```
rnorm(sd = 2, n = 7, mean = 10)
```

When you are first learning R, you may find it useful to write out the argument names as it can help you remember and understand what each part of the function is doing. However, as your skills progress you may find it quicker to omit the argument names and you will also see examples of code online that do not use argument names so it is important to be able to understand which argument each bit of code is referring to (or look up the help documentation to check).

In this manual, I will mostly write out the argument names the first time, however, in subsequent uses they may be omitted.



The assign a value to an named argument of the function, you have to use the equal sign `=` and not `<-`.

4.6.5. Tab auto-complete

One very useful feature of R Studio is the tab auto-complete for functions (see Figure 2). If you write the name of the function and then press the tab key, R Studio will show you the arguments that function takes along with a brief description. If you press enter on the argument name it will fill in the name for you, just like auto-complete on your phone. This is incredibly useful when you are first learning R and you should remember to use this feature frequently.

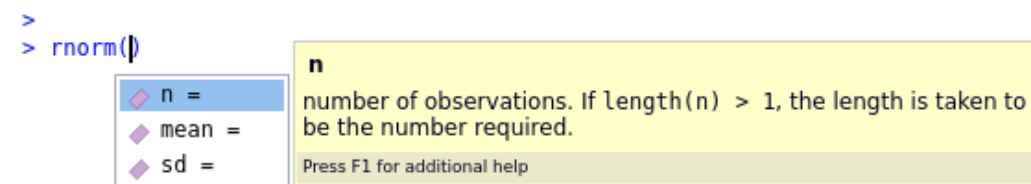


Figure 2: The tab auto-complete of RStudio.



Get the documentation of the optional argument `sd` via the autocomplete function in RStudio. Compare it with the `?`-documentation.

4.6.6. Important Base R functions

When you install *R* you will have access to a range of functions including options for data wrangling and statistical analysis. These functions that are included in the default installation are typically referred to as Base *R*. There is a very useful overview that shows the most important Base *R* functions.

An overview of this functions is online available as pdf and call the “[Base R cheat sheet](#)”.

Don’t worry, you don’t have to know this functions. You just have to know **how to figure out** which function is needed to accomplish you task and where you find to detailed information how to use this function (see section 2).

4.6.7. User-defined functions

The real power of a programming language is that you can develop your own functions that accomplish reoccurring tasks or that this way facilitate or structures your analysis.

Moreover, there are many collections of additional functions for specific tasks available online. They extend your *R* functionality and have been programmed by other data analysts or statisticians. This extensions are called in *R* “**packages**”.

Later in the course we will a package called [Tidyverse](#), which extends the *R* by many convenient functions for data handling. But for now, let’s keep it simple and just focus for now on the usages of the basic built-in functions in *R*.



If you want to develop your own functions (or even packages), please have a look, for instance, at the [online tutorial](#) of Grolemund and Wickham or see the [suggested reading](#).

Part II.

Data Structures

5. Atomic Vectors

A dataset consists typically not only of a single number or a single data point. We need to be able to deal with collections of data points or datasets. *R* knows different types of objects that store datasets.

We'll start with the most basics type of data structure in *R*, which is an **atomic vector**. A vector is merely a list of data points. They are very simple and they do show up everywhere. If you look closely enough, you'll see later that almost all data structures in *R* are built from atomic vectors.

You can make an atomic vector by grouping some values of data together with the command `c()` (which stands for concatenate).

```
vec <- c(1,2,3,4,5,6)
vec
```

```
## [1] 1 2 3 4 5 6
```

```
vector_of_logicals <- c(FALSE, TRUE, TRUE, FALSE)
vector_of_logicals
```

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

```
days_of_week <- c("Monday", "Tuesday", "Wednesday", "Thursday", "Friday",
                  "Saturday", "Sunday")
days_of_week
```

```
## [1] "Monday" "Tuesday" "Wednesday" "Thursday" "Friday" "Saturday"
## [7] "Sunday"
```

You can make vectors of any type of data, but **one atomic vector comprises only data of one type!** That is, each atomic vector stores its values as a one-dimensional vector, and each atomic vector can only store one data type. If you mix numbers (*integer*, *doubles*), *logicals*, and text (*characters*), *R* converts them to one data type. This should be always avoided, because it will cause problems later once you want to calculate with the data.

```
x <- c("a", 3, "c", 6.5)
x
```

```
## [1] "a" "3" "c" "6.5"
```

As you see, all elements have been converted to character. That's bad. If you want to mix different data types in one collection, you have to use `list()` as described in chapter 6.

Concatenating two vectors

You can use the `c()`-command also to concatenate two vectors:

```
x <- c(1,2,3,4,5,6)
y <- c(20,30,40)
z <- c(x, y)
z
```

```
## [1] 1 2 3 4 5 6 20 30 40
```

Vector length

The command `length` returns the number of elements in an atomic vector.

```
length(days_of_week)
```

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

5.1. Number sequences

You can generate a simple ascending or descending sequence of integer numbers like this:

```
x <- 1:100 # ascending sequence
x
```

```
## [1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
## [19] 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36
## [37] 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54
## [55] 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72
## [73] 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90
## [91] 91 92 93 94 95 96 97 98 99 100
```

```
y <- 49:-50 # descending sequence
y
```

```
## [1] 49 48 47 46 45 44 43 42 41 40 39 38 37 36 35 34 33 32
## [19] 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14
## [37] 13 12 11 10 9 8 7 6 5 4 3 2 1 0 -1 -
2 -3 -4
## [55] -5 -6 -7 -8 -9 -10 -11 -12 -13 -14 -15 -16 -17 -18 -19 -
20 -21 -22
## [73] -23 -24 -25 -26 -27 -28 -29 -30 -31 -32 -33 -34 -35 -36 -37 -
38 -39 -40
## [91] -41 -42 -43 -44 -45 -46 -47 -48 -49 -50
```

What does the numbers in brackets in the output mean? *R* indicates before each output line the position of the next printed element in the vector (e.g. [19]). That's why the console output of vectors always start with `## [1]`.

5.1.1. Sequence generating functions

R come with a lot of methods that generate sequence. For instance, for instance the function `seq()` has be already used in the previous assignment. Let's have a look at some further features of the function. You can also produce floating points with `seq()`:

```
seq(10, 12, by=.2)
```

```
## [1] 10.0 10.2 10.4 10.6 10.8 11.0 11.2 11.4 11.6 11.8 12.0
```

If you want to make a vector of a particular length (with equalled spaced numbers) you can you the optional argument `length`. For example,

```
seq(0,1, length=12) # 12 numbers between 0 and 1
```

```
## [1] 0.00000000 0.09090909 0.18181818 0.27272727 0.36363636 0.45454545
## [7] 0.54545455 0.63636364 0.72727273 0.81818182 0.90909091 1.00000000
```

Random numbers

Random numbers sequences are very useful to illustrate probabilistic phenomena in statistics and are required in psychology for mathematical modelling and simulations. *R* can generate random numbers from different probability functions.

Drawing random numbers from a normal distribution has already been discussed in section 4.6.3. The function `runif()` is used draw random numbers from a uniform distribution:

```
runif(10) # 10 random numbers between 0 and 1
```

```
## [1] 0.03637661 0.06360644 0.63611516 0.39051917 0.37345532 0.51225228  
## [7] 0.32133011 0.23485957 0.82402430 0.38025419
```

```
runif(10, min= 10, max = 99) # 10 random numbers between 10 and 99
```

```
## [1] 33.90631 54.17126 76.92904 86.29597 45.98563 15.12934 31.67234 42.22995  
## [9] 40.19823 36.98721
```



More information about random number generation and the probability distributions that are implemented in base *R* can be found by calling `?RNG` and `?distributions`.

5.2. Manipulating vectors

If you apply an operation on a vector, *R* applies it to each element. You have seen already some examples.

Some examples

```
v <- c(1,2,3,4,5,6)  
v + 100
```

```
## [1] 101 102 103 104 105 106
```

```
v**3 # to the power of 3
```

```
## [1] 1 8 27 64 125 216
```

```
sqrt(v) # square root
```

```
## [1] 1.000000 1.414214 1.732051 2.000000 2.236068 2.449490
```

Please note, for functions it's not always like this. Some functions do indeed apply an operation to each element and return as result a vector like `sqrt()`. However, not all functions work element-wise. Some functions do a calculation with all number and return for instance a single value. Good examples for this are:

```
v <- 200:3000
mean(v) # calculate the mean of the numbers 200 to 300
```

```
## [1] 1600
```

```
max(v) # the largest number
```

```
## [1] 3000
```



What is returned, if you compare vectors with a single number?

Solution

Examples:

```
v <- 1:10
v <= 3 # smaller or equal to three
```

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

```
v == 5
```

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

```
v <- c(1, 4, 6, 9, 2, 3, 5)
v < 5
```

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

Answer: *R* returns a vector of logicals that results for the comparison the number with each element.

5.2.1. Basic operations & comparisons with two vectors

Basic operations between two vectors are useful only between vectors of the same length. In this case, *R* combines or compares each element of the first with each element of the second vector:

```
a <- c(5, 8, 2, 4, 1, 10, 7, 3, 9, 6)
b <- 1:10
a + b      # c(5+1, 8+2, 2+3, 4+4, ..)
```

```
## [1] 6 10 5 8 6 16 14 11 18 16
```

```
a * b      # c(5/1, 8/2, 2/3, 4/4, ..)
```

```
## [1] 5 16 6 16 5 60 49 24 81 60
```

```
a > b      # c(5>1, 8>2, 2>3, 4>4, ..)
```

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

Avoid to combine two vectors of different length. The result is not very intuitive and typically not useful. If that happens, you get a warning by *R*.



If two vectors are not of the same length, *R* loops over the shorter vector. Check out what happens here:

```
c(2, 1000) + c(100, 101, 103)
```

Other object types that are commonly used in *R* are **arrays** and **matrices**. Matrices store values in a two-dimensional array, that is, you have rows and columns. These data types are crucial for those who use *R* for linear algebra or want to handle very large datasets. *R* furthermore knows n-dimensional arrays. However, in this course we will not further discuss these data types.

5.3. Subsetting Vectors

Often you want to access only subsets, that is, parts or single elements of a vector. Let's explore the different types of subsetting with a simple vector of floating point numbers *x*.

```
x <- c(2.1, 4.2, 3.3, 5.4)
```

Please note, in this example vector the number after the decimal point represents the original position in the vector. That might make it easy to understand the following examples.

If you want to have only certain elements of a vector you have to use the operator `[]`. The `[]` operator can be called

- with positive integers
- with negative integers
- with logical vectors
- with nothing

5.3.1. Positive integers

Positive integers return elements at the specified positions:

```
x[2]           # access only the 2nd element
```

```
## [1] 4.2
```

```
x[c(2, 4)]     # access the 2nd and 4th element
```

```
## [1] 4.2 5.4
```



This vector has only 4 elements. But what happens if you ask for the 5th, not existing element? Check the documentation (?) what type of data the resulting ‘value’ is.

Solution

```
x[5]
```

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

or another example

```
x[c(1, 3, 5)]
```

```
## [1] 2.1 3.3 NA
```

Type ?NA to get the documentation: NA stands for “Not Available” and is an important basic “data type” that you need to know.

5.3.2. Negative integers

Negative integers exclude elements at the specified positions

```
x[-2]          # exclude 2nd element
```

```
## [1] 2.1 3.3 5.4
```

```
x[-c(3, 2)]    # exclude 3rd and 2nd element
```

```
## [1] 2.1 5.4
```

5.3.3. Logical vectors

Logical vectors select elements where the corresponding logical value is `TRUE`.

```
x[c(TRUE, FALSE, TRUE, FALSE)]
```

```
## [1] 2.1 3.3
```

It returns the elements 1 and 3, because these elements where `TRUE` in the logical vector. This is probably the most useful type of subsetting, because you can write an expression that generate a logical vector. For instance, if you want to have all elements that are larger 3

```
x[x > 3]
```

```
## [1] 4.2 3.3 5.4
```

Importantly, if you use logical vectors for subsetting, the logical should have the same length!



In `x[y]`, what happens if `x` and `y` have different lengths? This is, for instance, explained [here](#).

5.3.4. With nothing

Nothing returns the original vector.

```
x[]
```

```
## [1] 2.1 4.2 3.3 5.4
```

Why shall I do this? This is not useful for 1D vectors in this example, but, as you'll see shortly in chapter 7, it's very useful for data frames.

5.4. Named vectors

R object can have **attributes**. An attribute is a piece of information that you can attach to an atomic vector (or any *R* object). The attribute won't affect any of the values in the object, and it will not appear when you display your object. You can think of an attribute as “metadata”; it is just a convenient place to put information associated with an object. *R* will normally ignore this metadata, but some *R* functions will check for specific attributes.

For now the only important attribute is **name**. It's sometimes convenient to add names to values that you want to store. For instance, if you save the body length of three subjects (“Peter”, “John”, “Maria”) in a vector you could do it like this:

```
body_len <- c(1.84, 1.92, 1.71, 1.63)
names(body_len) <- c("Peter", "John", "Maria", "Linda")
```

The object `body_len` has a **names** attribute and *R* will display the names above the elements whenever you print the vector:

```
body_len
```

```
## Peter  John Maria Linda
##  1.84  1.92  1.71  1.63
```

This will give you just the names:

```
names(body_len)
```

```
## [1] "Peter" "John"  "Maria" "Linda"
```

5.4.1. Subsetting named vectors by `[name]`

If the vector is named, you can also use the names or a vector of names to access selected elements. This is a convenient feature and can produce more readable code:

```
body_len["Linda"]
```

```
## Linda
##  1.63
```

```
body_len[c("Maria", "Peter")]
```

```
## Maria Peter  
## 1.71 1.84
```

6. Lists

Lists are like atomic vectors because they group data into a one-dimensional set. However, lists do not group together individual values; lists group together *R* objects, such as atomic vectors and other lists. For example, you can make a list that contains a numeric vector of length 31 in its first element, a character vector of length 1 in its second element, and a new list of length 3 in its third element. To do this, use the `list()` function.

`list()` creates a list the same way `c()` creates a vector. Separate each element in the list with a comma:

```
a_list <- list(100:111, "Erasmus", c(FALSE, TRUE, FALSE))  
a_list
```

```
## [[1]]  
## [1] 100 101 102 103 104 105 106 107 108 109 110 111  
##  
## [[2]]  
## [1] "Erasmus"  
##  
## [[3]]  
## [1] FALSE TRUE FALSE
```

The double-bracketed indexes, `[[.]]`, tell you which element of the list is being displayed. The single-bracket indexes, `[]`, tell you which subelement of an element is being displayed. For example, 100 is the first subelement of the first element in the list. "Erasmus" is the first sub-element of the second element. This two-system notation arises because each element of a list can be *any R-object*, including a new vector (or list) with its own indexes.

As you can imagine, the structure of lists can become quite complicated, but this flexibility makes lists a useful all-purpose storage tool in *R*: you can group together anything with a list.

6.1. Subsetting lists

If you want to access the third element in the list you have to call:

```
a_list[[3]]
```

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

In our example, this is a vector with 3 logicals.

Please keep in mind that **you have to use the double-bracketed indexes `[[...]]`, if you want to access the content of a specific object in a list**. Single-brackets, e.g. `a_list[3]`, don't raise an error, but returns something slightly different. This behaviour is often confusing for beginners.



But this actually very consistent. `a_list[3]` returns a list with one object, because `a_list[c(3,2)]` would return also a list (with two objects) and because subsetting atomic vectors with single brackets also return vectors. This check this out:

```
typeof(a_list[2]) and typeof(a_list[[2]])
```

6.2. Named lists

This subsetting with double-brackets is not very intuitive. At least in my view. I try to avoid it always and prefer to use names in lists. Lists produce readable code, if you use `name` attribute and if assign useful names to the elements in the list. You assign names to lists, like you did it with vectors in section 5.4. In our example, it would be:

```
b_list <- list(100:111, "Erasmus", c(FALSE, TRUE, FALSE))
names(b_list) <- c("my_sequence", "university", "x")
```

But this can be done also in one line of code and is even easier and even more readable, right?

```
b_list <- list(my_sequence=100:111, university="Erasmus", x=c(FALSE, TRUE, FALSE))
b_list
```

```
## $my_sequence
## [1] 100 101 102 103 104 105 106 107 108 109 110 111
##
## $university
```

```
## [1] "Erasmus"
##
## $x
## [1] FALSE TRUE FALSE
```

6.3. The \$ operator

If you want to access a specific element you can call it by its name using the \$ operator. See what you get, if you type:

```
b_list$my_sequence
b_list$university
```

You can also add a new element to list simply by using the \$ operator followed by a new name

```
b_list$my_new_element <- 15.6
```



Have a look in RStudio in the Environment viewer and see how `a_list` and `b_list` are displayed.

- Click on the arrow to get more information.
- Double click the name of one list.

What should I see?

1. You see details of the three objects/elements of the list in the environment tab.
2. Details are displayed in a separate panel.



Make a list called `lst` with three objects

- a vector called `subjects` containing the names John, Maria and Linda
- a vector called `body_len` containing the length in meter 1.92, 1.71 and 1.63
- a vector called `body_cm` containing the calculated length in centimeter

Solution

```
lst <- list(subjects=c("John", "Maria", "Linda"),
            body_len=c(1.92, 1.71, 1.63))
lst$body_cm = lst$body_len * 100
print(lst)
```

```
## $subjects
## [1] "John" "Maria" "Linda"
##
## $body_len
## [1] 1.92 1.71 1.63
##
## $body_cm
## [1] 192 171 163
```

Table 1: A tabular dataset.

Subject_id	Gender	IQ	International_student
1	female	102	FALSE
2	female	121	FALSE
3	male	112	TRUE
4	male	103	FALSE

7. Data Frames

So far, we have worked in this course with very basic data structures. In psychology we have often a specific type of data so-called tabular data. Tabular data are data that can be stored in a table with different variables arranged in columns. The rows of the table comprise different cases. You do know this kind of data from spreadsheet or statistics software like *MS Excel* or *SPSS*. For example:

In *R*, **tabular data are stored in data frames**. Data frames are the two-dimensional version of a list. They are far and away the most useful storage structure for data analysis, and they provide an ideal way to store that above.

Data frames group vectors together into a two-dimensional table. Each vector becomes a column in the table. As a result, each column of a data frame can contain a different type of data; but within a column, every cell must be the same type of data.

data frame	1	"R"	TRUE
	2	"S"	FALSE
	3	"T"	TRUE
	numeric	character	logical

Figure 3: Data frames

Data frames store data as a sequence of columns. Each column can be a different data type. Every column in a data frame must be the same length.

Creating a data frame by hand takes a lot of typing, but you can do it (if you like) with the `data.frame` function. Give `data.frame` any number of vectors, each separated

with a comma. Each vector should be set equal to a name that describes the vector. `data.frame` will turn each vector into a column of the new data frame:

```
df <- data.frame(Subject_id = c(1,2,3, 4),
                  Gender = c("female", "female", "male", "male" ),
                  IQ=c(102, 121, 112, 103),
                  International_student = c(FALSE, FALSE, TRUE, FALSE))
```

You'll need to make sure that each vector is the same length. The resulting data frame looks like this:

```
print(df)
```

```
##   Subject_id Gender  IQ International_student
## 1           1 female 102                   FALSE
## 2           2 female 121                   FALSE
## 3           3   male 112                    TRUE
## 4           4   male 103                   FALSE
```



Remember: Use the = assignment inside functions

Since you `data.frame` is a function and `Subject_id`, `Gender`, `IQ` and `International_student` are arguments of this function, you have to use the equal sign `=` to map the vectors to variable names and not the left hand side assignment `<-`.

Variable overview

To function `str()` gives you a side-wise overview of the variables in a data frame. This show the variable names, the data types and the first few data points:

```
str(df)
```

```
## 'data.frame':    4 obs. of  4 variables:
## $ Subject_id      : num  1 2 3 4
## $ Gender           : chr  "female" "female" "male" "male"
## $ IQ               : num  102 121 112 103
## $ International_student: logi  FALSE FALSE TRUE FALSE
```

Descriptive summary

A descriptive summary of all variables a data frame is very handy and important to check your data quality and to exclude potential technical issue in your dataset. It's very simply to do by:

```
summary(df)
```

```
##      Subject_id      Gender      IQ      International_student
## Min.      :1.00   Length:4      Min.      :102.0   Mode :logical
## 1st Qu.:1.75   Class :character  1st Qu.:102.8   FALSE:3
## Median :2.50   Mode  :character  Median :107.5   TRUE :1
## Mean    :2.50                                Mean    :109.5
## 3rd Qu.:3.25                                3rd Qu.:114.2
## Max.    :4.00                                Max.    :121.0
```

You should always do quality checks like this, if you read in tabular data.



Quality check

Below you see summarizing information about dataset call `demo_data`. Without knowing the data in detail, what is strange in this dataset?

```
nrow(demo_data)
```

```
## [1] 200
```

```
summary(demo_data)
```

```
##      variable_a      variable_b      variable_c
## Min.      : -173.220   Min.      : -2.68079   Min.      :10
## 1st Qu.:    6.334     1st Qu.: -0.80996     1st Qu.:10
## Median :   86.670     Median : -0.00754     Median :10
## Mean    :   73.167     Mean    : -0.09343     Mean    :10
## 3rd Qu.:  148.005     3rd Qu.: 0.55309     3rd Qu.:10
## Max.    :  312.665     Max.    : 2.42227     Max.    :10
##                                     NA's    :131
```

Solution

The summary of this dataset shows that

1. you have many missing values in `variable_b` (131 out of 200 rows).
2. `variable_b` comprises always of the same value 10 and has thus no variance. This might be a problem for you later statistical analyses.

7.1. Subsetting

7.1.1. Selecting columns by name

You can access a specific column/variable of the data frame with **\$ operator** as you know it from **lists**. For example:

```
df$IQ
```

```
## [1] 102 121 112 103
```

As you see above, if you access a variable of a data frame with the **\$** operator, you get an **atomic vector**. This means that you can use subsetting of vectors as described in section 5.3.

```
g <- df$Gender      # g is a vector with the genders
g[2]                # print the gender at the second positions
```

```
## [1] "female"
```

But this can be done also in one expression.

```
df$Gender[2]        # get the Gender in the second row position
```

```
## [1] "female"
```



- Calculate the median of the IQs of the subjects in the data frame.
- Make a vector of logicals that is **TRUE** for all students that have an IQ larger than the mean IQ (and **FALSE** otherwise). Include this vector as new variable **is_clever** to our data frame.

Solution

```
m_iq = mean(df$IQ)
m_iq
```

```
## [1] 109.5
```

```
df$is_clever <- df$IQ > m_iq
df
```

```
##   Subject_id Gender  IQ International_student is_clever
## 1         1 female 102                FALSE    FALSE
## 2         2 female 121                FALSE     TRUE
## 3         3 male  112                 TRUE     TRUE
## 4         4 male  103                FALSE    FALSE
```


7.1.2. Selecting by row and column indices

Cells in an data frame can be also access via row and column indices and the `[]` operator. The syntax looks like this: `[row_index, column_index]`.

```
# Check what you get
df[1, 3]           # returns from the 1st row the column 3
df[2:4, 3]         # returns from row 2-4 the column 2
df[c(1, 4), c(2,3)] # returns from row 1 and 4 the column 2 and 3
```

If you omit one of the indices *R* takes all rows or columns of the omitted index.

```
df[, 2]           # all rows of column 2
df[4, ]           # all columns of row 2 and 4
```

Consequently, `df[,]` or `df[]` should return the entire data frame...and this exactly what *R* is doing.



- Access all columns from the 1st and 3rd row of `df`.
- Do this also in an reversed order of the columns. That is, the first column should be `International_student`.

Solution

```
df[c(1, 3), ]
```

```
## Subject_id Gender IQ International_student is_clever
## 1          1 female 102                FALSE    FALSE
## 3          3  male 112                TRUE     TRUE
```

```
df[c(1, 3), 4:1]
```

```
## International_student IQ Gender Subject_id
## 1                FALSE 102 female          1
## 3                 TRUE 112  male           3
```

7.1.3. Subsetting with logicals

Section 5.3.3 demonstrated that you can select subsets with logicals. This subsetting with logical vectors is very useful for data frames and very often used. For example, if you want to have the data of the females only:

```
selected_rows <- df$Gender == "female" # logical vector with TRUE for females
df[selected_rows, ]
```

```
##   Subject_id Gender  IQ International_student is_clever
## 1           1 female 102                   FALSE    FALSE
## 2           2 female 121                   FALSE     TRUE
```

Alternatively you can write it in one line:

```
df[df$Gender == "female", ]
```



Make a data frame with the clever people only.

Solution

```
clever <- df[df$is_clever, ] # this clever is already a logical vector
clever
```

```
##   Subject_id Gender  IQ International_student is_clever
## 2           2 female 121                   FALSE     TRUE
## 3           3  male 112                   TRUE      TRUE
```

7.2. Lists and data frames

Since lists and data frame can store multiple atomic vectors and they can actually both store tabular-like data. You can therefore convert between this two types using an `as.-converter` function. For instance, the list `lst` that we made in section 6.3 can be converter to a data frame.

```
lst <- list(subjects=c("John", "Maria", "Linda"),
            body_len=c(1.92, 1.71, 1.63))

df2 <- as.data.frame(lst)
df2
```

```
##   subjects body_len
## 1     John    1.92
## 2    Maria    1.71
## 3    Linda    1.63
```

Note, converting a list to a data frame makes only sense, if the data in the list are indeed tabular, that is, if all objects in the list are atomic vectors of the same length.

Finally, a data frame can be converted into a list using `as.list()`. See what you get, if you convert the above data frame `df`:

```
as.list(df)

## $Subject_id
## [1] 1 2 3 4
##
## $Gender
## [1] "female" "female" "male"    "male"
##
## $IQ
## [1] 102 121 112 103
##
## $International_student
## [1] FALSE FALSE  TRUE FALSE
##
## $is_clever
## [1] FALSE  TRUE  TRUE FALSE
```

7.3. Summary: Data structures in R

You can store data in *R* with five different objects, which let you store different types of values in different types of relationships. The most common data structures in the analysis of psychological data are **vectors**, **lists** and **data frames**. For the sake of completeness, you should know that **matrices** and **n-dimensional arrays** also exist in *R*—they are however not relevant for this course.

Importantly, data frames can store data of different types (e.g. number, characters, text, factors, logical). Each column is conceptually an atomic vector and can thus only store data of one type. Data frames store the most common forms of data used in statistics, which are tabular data.

8. Packages

When you install *R* you will have access to a range of functions including basic options for data wrangling and statistical analyses. The functions that are included in the default installation are typically referred to as **Base R** and there is a useful [cheat sheet](#) that shows many Base *R* functions here. We've been using a couple of functions from base *R*, such as `seq()` or `rnorm()`—

However, the amazing power of *R* is that it is extendable. That is, if a command or method doesn't exist anyone can create a so-called *package* that contains one or multiple functions (and/or data) to perform new tasks. The large active *R* user community is always creating new packages to expand *R*'s capabilities. You may find it useful to think

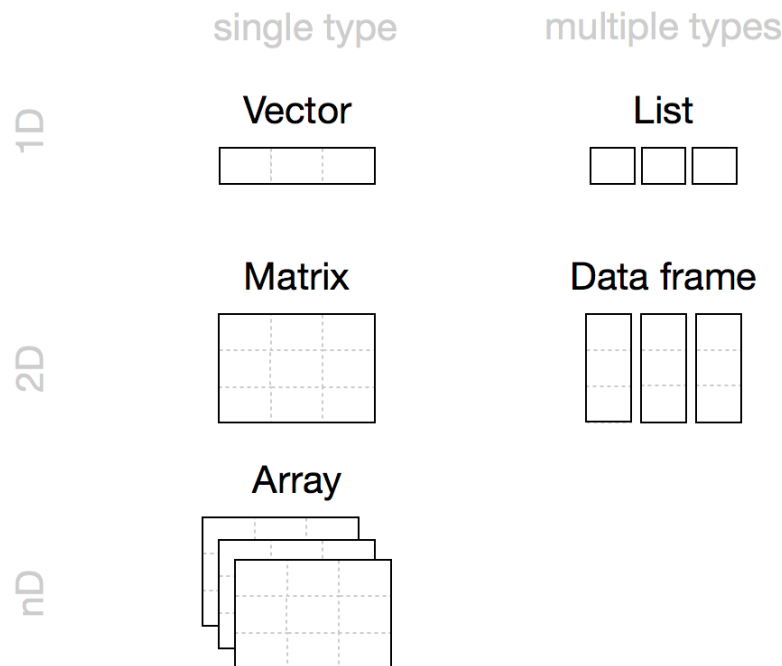


Figure 4: Overview of data structures in R

of Base *R* as the default apps that come on your phone and packages as additional apps that you need to download separately.

In *R*, the fundamental unit of shareable code is a package. A package bundles together code, data, documentation, and tests, and is easy to share with others. They increase the power of *R* by improving existing base *R* functionalities, or by adding new ones.

8.1. Installing a package

The traditional place to get *R* packages is from [CRAN, the Comprehensive R Archive Network](#), which is where you have downloaded *R*.

However, you don't need to go to *CRAN*'s website to install packages. It's much simpler and can be accomplished within the console using the command `install.packages("package-name-in-quotes")`. *R* searches online on the *CRAN* servers for a packages with that name and downloads and installs it locally. This works, of course, only if you have an internet connection. Let's install a small, fun package `praise`. You need to use quotes around the package name:

```
install.packages("praise")
```



The next chapters will make use of two packages that help us to important data in *R*. This packages are called `readr` and `readxl`. Please install these two packages.

Solution

```
install.packages("readr")
install.packages("readxl")
```



If you want to install multiple packages at once, you have to use a vector of the package names you want to install as argument, e.g., `install.packages(c("package-a", "package-b"))`.

8.2. Using a package

So far, we've just installed packages. That is, the packages are now available on the local computer. Before we can use the new functionality, we need to tell *R* to activate the package for this session, or better said, to load the library to memory. This will be done by the function `library()`. For example, to use the `praise`-package we call first:

```
library("praise")
```

Now that the `praise` package is loaded, you can use the function in that package. The new function is called `praise()` and returns a praise to make you feel better.

```
praise()
```

```
## [1] "You are stellar!"
```

(As you see, you can do really useful things with R ;-)

But what's the difference between a package and a library?

Sometimes there is a confusion between a package and a library. Please don't get confused. You can find many people calling "libraries" to packages (like me, sometimes). To be precise, a library is a the collection of functions that we use in *R* and that will be loaded into memory with the `library()`-command. A package is the bundle of software that you have to install to get the library. A package can contain more than just the library, for instance, documentation, example scripts and data.

The `::` operator

Once you have load the package with `library()` *R* knows all functions of that package and you can run the function just by calling it's name (as you know it).

If you call a function, it's also possible to mention explicitly the package from which the function stems. This is done with the `::` operator that separates the package and function name: `package-name::function-name()`. For example, calling the function `praise()` from the package `praise` would look like this:

```
praise::praise()
```

Why should I use the `::` syntax?

You don't have to use `::` and I guess that you will hardly need it for you own analyses. However, there are some reasons why you should know that this syntax exists:

1. In manuals and documentations people often use this syntax to make clear from which package a particular function stems and which package needs to be installed to make use of this function.
2. If you use the `::` operator, you don't have to call `library()`. That is, if you need a function just once, it might be convenient to save a line of code.
3. It is possible that you use two packages that introduce the same name for different functions. This causes a naming conflict and one package masks the function of the other package. To get access to both functions in the different packages, you need to use the `::` operator.



Mask conflicts occur for instance in the `tidyverse` package collection, that we'll use next week. Mask conflicts will be depicted as warnings while loading (importing) a library.

8.3. Getting help for packages

If you want to know more about a package, you can use *R*'s built-in help function: `help(package="package-name")`.



Try to access the documentation of the function `read_excel` of the newly installed `readxl`-package.

9. Importing Data

A crucial function of a statistical software is of course the ability to read data from files. There are many different types of files that you might work with when doing data analysis. These different file types are usually distinguished by the three letter extension following a period at the end of the file name. Here are some examples of functions (and packages!) that you can use to import data from different file types:

Extension	File Type	Reading	Writing
.Rdata, .Rds	R's own data format	<code>load()</code> or <code>readRDS()</code>	<code>save()</code> or <code>saveRDS()</code>
.csv	Comma-separated values	<code>readr::read_csv()</code>	<code>readr::write_csv()</code>
.tsv, .txt	Tab-separated values	<code>readr::read_tsv()</code>	<code>readr::write_tsv()</code>
.xls, .xlsx	Excel workbook	<code>readxl::read_excel()</code>	–
.sav	SPSS data	<code>haven::read_sav()</code>	<code>haven::write_sav()</code>
.dta	Stata data	<code>haven::read_dta()</code>	<code>haven::write_dta()</code>

As you see, different packages that are dealing with data import. You have already installed the packages `readxl` and `readr` in the previous exercise.

Example data: You find some example data files in the `data` folder.

9.1. Import via RStudio interface

R offers several ways to import data from a file. One way is RStudio's import functions (see this week's instruction video).

.Rdata: R's own data files (.Rdata) can be very easily imported just by clicking on the file in the file browser.



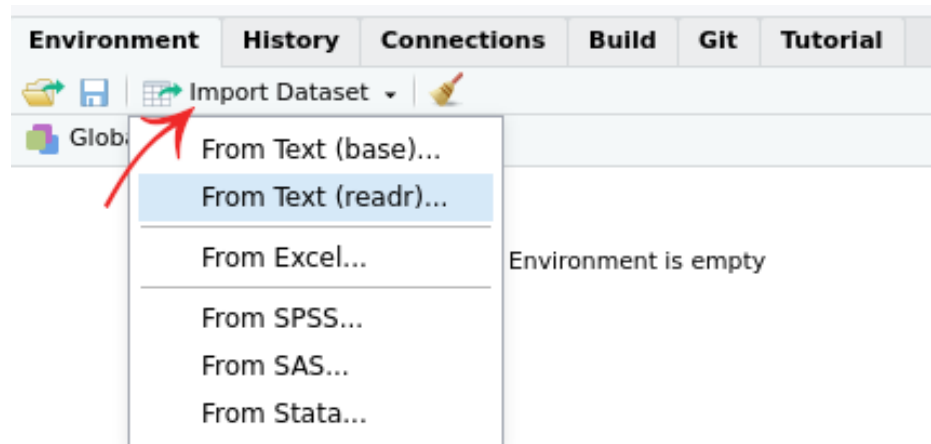
You find a file `babynames.Rdata` among the example data. It contains the frequencies of names given to babies in the US since 1880.

Load the dataset (via double click). How many rows does this file have?

Solution

Almost two million rows: 1924664

Other file types To import data from other (non-*R*) file types, you can use the import panel of RStudio.



9.2. Import via Console or Script

.Rdata

To load an *R* data file via a script or the console, you called `load()` and the file path as argument. A file path consists of the subdirectory (relative to your current directory) and the filename separated by a `/` (backslash). For instance, if the data file is `babynames.Rdata` located in the subfolder `data`, this command should load the dataset:

```
load("data/babynames.Rdata")
```

Hint: You can use the autocompletion function of RStudio and type only `load(",` followed TAB.



Since file paths are always relative to the **current working directory**, you need to check in which directory you are currently in (or in which folder the current script is).

The console displays the current working directory on top of the console panel. Also the command `getcwd()` gives you this information. The autocompletion function is also handy here.

csv- and xlsx-files and other files

When you read in CSV files, it is best practice to use the `readr::read_csv()` function. Note that you would normally want to store the result of the `read_csv()` function to an object, as so:

```
library(readr) # load the package (just needed the first time)
csv_data <- read_csv("data/exam_anxiety.csv")
```

The function `read_excel()` from the package `readxl` is handy to read in Excel spreadsheet data.

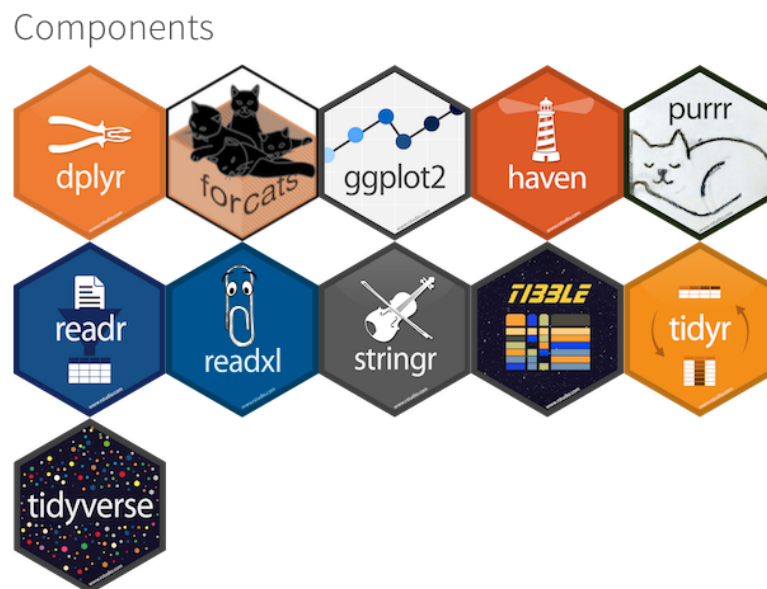
A summary of data import function can be found in this [data import cheat sheet](#).

Part III.

Data Wrangling

10. Tidyverse package collection

Tidyverse (<https://www.tidyverse.org/>) is a collection of R packages created by data scientist [Hadley Wickham](#), who has massively influenced the development *R* in the recent decade. *Tidyverse* contains several core packages (for example, `dplyr`, `tidyr`, `readr`, `purrr`, `ggplot2`, and `tibble`). It intends to facilitates the handling and visualization of data. After Wickham released these packages, they became heavily used in the *R* community and are, in my view, the new quasi-standard of *R* programming.



In the core ten packages of *Tidyverse* (see figure above), you find everything you need to preprocess and prepare you data for statistical analyses. We will therefore make use of *Tidyverse* in the rest of this courses.



Why not doing everything with base-*R*?

Of course, all things that we do with *Tidyverse* can be also somehow done with the base *R* commands, which we know already. However, the commands of *Tidyverse* are very powerful and, most importantly, they result in a computer code that is much more readably than the ugly base-*R* code.

The readability of the *R*-scripts is very important, because make it makes the data analyses less error-prone.

Installing

Tidyverse is actually not really a package, but a meta-package, that is, a collection of packages. That is, if you install *Tidyverse*, you install several packages. **It will therefore takes some time to install it**, especially on old machines.

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

Remark: **MacOS** or **Linux** users that get an error when installing *Tidyverse* need to install first some additional programs (see [workaround C.1](#)).

Importing

As with all *R* -packages, if you want to make use of it in yours scripts, you need to import it first:

```
library(tidyverse)
```

```
## -- Attaching core tidyverse packages -----
## v dplyr      1.1.4      v readr      2.1.5
## v forcats    1.0.0      v stringr   1.5.1
## v ggplot2    3.5.1      v tibble    3.2.1
## v lubridate  1.9.3      v tidyr     1.3.1
## v purrr      1.0.2
## -- Conflicts -----
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
## i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to
##
## Attaching packages              tidyverse 1.3.0
## ggplot2 3.3.2      purrr 0.3.4
## tibble 3.0.3      dplyr 1.0.0
## tidyr 1.1.0      stringr 1.4.0
## readr 1.3.1      forcats 0.5.0
## Conflicts                ## tidyverse_conflicts()
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

If you import the *Tidyverse*-package with the `library()`-command you see that several core packages will be imported. The names of this packages are not really important here, you should just be aware that you can also import the single sub-packages separately. You see that often in code examples, if you check the help pages of *R* or if you google for solutions in data processing.

10.1. Tibble is the new Data Frame

Tidyverse introduces a new type of `data.frame`, called `tibble`. You use tibbles exactly like data frames and it does not really matter for you, whether you have a tibble or a data frame. *R* converts a data frame automatically into a tibble whenever that is required and possible. I use the two labels therefore sometimes interchangeably.

You just need to know that a `tibble` is a modern version of a data frame, because it will show up in the *R* output and in some scripts and documentations. Actually, we have already worked with the tibbles, because the data frames that you created when importing csv-files with the function `read_csv` are tibbles and not base-*R* data frames. Why? The package `readr` that we use for the import is part of the *Tidyverse*.

The output of tibble is somewhat nicer than the output of data frames. But the difference is really minor. Check it out:

```
library(readr) # load the package (just needed the first time)
csv_data <- read_csv("data/exam_anxiety.csv")
print(csv_data)
```

The first row shows that it is a tibble. The second row depicts the variable names and the third row shows the data type of the variable (`dbl` stands for “double” that is a numerical).



You can convert a data frame to a tibble with the function `as_tibble()` and a tibble into a `data.frame` with `as.data.frame()`.

10.2. Naming convention (“_” or “.”)

If you are unsure whether a function is from *Tidyverse* or part of base-*R*, the function name often gives you a hint. *Tidyverse* functions use always an underscore sign (`_`) in the function names; for example `as_tibble()`, the function to convert sometimes into a tibble. Classic base-*R* uses the dot (`.`) in names, like in `as.numeric()` or `as.data.frame()`. This has historical reasons, because *R* is inspired by the [programming language S](#), a language from the 70th. In that time the underscore was not a key on all computer keyboards.

Table 3: First row of the example dataset 'teaching method'.

student	teaching_method	city	time	statistics	cog_psy
1	method_A	Groningen	before	6	10
2	method_B	Nijmegen	before	11	13
3	control	Rotterdam	before	12	10
4	method_A	Utrecht	before	10	9
5	method_B	Nijmegen	before	9	8
6	control	Utrecht	before	8	11

I personally also prefer to use the underscore sign (as you have seen in the examples of this manual), because it's better readable and in line with naming conversions in other programming languages often used for data processing like [Python](#), [Julia](#) or [MATLAB](#). Since more and more *R*-users work with *Tidyverse* the naming convention is currently also changing in the *R*-community—that is at least my impression.

11. Transforming Data

Example data

To illustrate the *Tidyverse*-functions for data transformation, we use an example dataset `teaching_method.csv` from that [data folder](#).

The fake dataset looks like this (only 6 first rows are depicted):

Students from universities in different Dutch cities were taught in statistics and cognitive psychology. Two teaching methods were compared (method A & B). There was in addition a control condition with no teaching at all. The knowledge about the subjects was tested before and after the courses with a test of 40 questions and 4 choices. The results of the test (number of correct answers) for the two subjects are shown in the variables `statistics` and `cog_psy`.

Let's load *Tidyverse* and the dataset:

```
library(tidyverse)
teach <- read_csv("data/teaching_method.csv")
```

The import dataset is a tibble and looks in the console of RStudio like this:

```
print(teach)
```

```
## # A tibble: 60 x 6
##   student teaching_method city      time  statistics cog_psy
##   <dbl> <chr>          <chr>    <chr>    <dbl>    <dbl>
## 1     1      1 method_A      Groningen before         6      10
## 2     2      2 method_B      Nijmegen  before        11      13
## 3     3      3 control        Rotterdam before        12      10
## 4     4      4 method_A      Utrecht   before        10       9
## 5     5      5 method_B      Nijmegen  before         9       8
## 6     6      6 control        Utrecht   before         8      11
## 7     7      7 method_A      Leiden    before         8      12
## 8     8      8 method_B      Utrecht   before         9      12
## 9     9      9 control        Leiden    before         8      10
## 10    10    10 method_A      Groningen before        10      12
## # i 50 more rows
```

11.1. Selecting rows: Filter()



We have already selected rows or cases of data frames with base-*R* (see for last assignment or [section 7.1.3](#)). If we want to select all students from Rotterdam, we would do it in base-*R* like this:

```
teach[teach$city == "Rotterdam", ]
```

With *Tidyverse* this can be done much more easily with function `filter()`.

```
filter(teach, city == "Rotterdam")
```

Of course, you can use any logical comparisons that are possible in *R*. For example, if you want to have all rows, which are not a control condition or all rows where `cog_psy` is smaller 15:

```
filter(teach, teaching_method != "control")
filter(teach, cog_psy < 15)
```

This table gives an overview of logical comparisons in *R*:

Logic in R - ?Comparison, ?base::Logic			
<	Less than	!=	Not equal to
>	Greater than	%in%	Group membership
==	Equal to	is.na	Is NA
<=	Less than or equal to	!is.na	Is not NA
>=	Greater than or equal to	&, , !, xor, any, all	Boolean operators

11.1.1. Multiple criteria

logical “AND”

The great thing with *Tidyverse* is that you can very easy combine the filtering conditions. For instance, all cases in which the `city` is “Rotterdam” and the `time` is “after” and the `teaching_method` is not “control”.

```
filter(teach, city=="Rotterdam" & time=="after" & teaching_method!="control")
```

```
## # A tibble: 4 x 6
##   student teaching_method city      time  statistics cog_psy
##   <dbl>   <chr>          <chr>   <chr>    <dbl>   <dbl>
## 1      13 method_A      Rotterdam after      26      40
## 2      17 method_B      Rotterdam after      30      33
## 3      19 method_A      Rotterdam after      34      32
## 4      29 method_B      Rotterdam after      22      23
```

The sign `&` is used in *R* as logical *and*.

logical “OR”

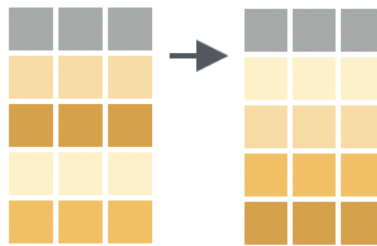
The sign `|` is used in *R* as logical *or*. For example, to select all rows, in which have test score in `statistics` lower 8 *or* in `cog_psy` lower 6.

```
filter(teach, statistics < 8 | cog_psy < 6)
```

```
## # A tibble: 6 x 6
##   student teaching_method city      time    statistics cog_psy
##   <dbl> <chr>          <chr>    <chr>      <dbl>   <dbl>
## 1      1 method_A      Groningen before         6      10
## 2     27 control       Rotterdam before         4      12
## 3     29 method_B      Rotterdam before        12       5
## 4     18 control       Leiden    after          6      16
## 5     21 control      Nijmegen after           7      11
## 6     27 control       Rotterdam after          4      11
```

More information about logical operators in *R* can be found for instance [here](#)

11.2. Sorting rows: `arrange()`



`arrange()` works similarly to `filter()` except that instead of selecting rows, it changes their order. It takes a data frame and a set of column names (or more complicated expressions) to order by.

```
teach <- arrange(teach, statistics)  # re-arrange & override to old data frame
teach
```

```
## # A tibble: 60 x 6
##   student teaching_method city      time    statistics cog_psy
##   <dbl> <chr>          <chr>    <chr>      <dbl>   <dbl>
## 1     27 control       Rotterdam before         4      12
## 2     27 control       Rotterdam after          4      11
## 3      1 method_A      Groningen before         6      10
## 4     18 control       Leiden    after          6      16
```



```
## 5      21 control      Nijmegen after      7      11
## 6       6 control      Utrecht  before     8      11
## 7       7 method_A     Leiden   before     8      12
## 8       9 control      Leiden   before     8      10
## 9      13 method_A     Rotterdam before     8      13
## 10     16 method_A     Nijmegen before     8       9
## # i 50 more rows
```

Use `desc()` to re-order by a column in descending order:

```
teach <- arrange(teach, desc(statistics))
teach
```

```
## # A tibble: 60 x 6
##   student teaching_method city      time  statistics cog_psy
##   <dbl> <chr>      <chr>    <chr>    <dbl>    <dbl>
## 1     28 method_A     Leiden  after     37      35
## 2     19 method_A     Rotterdam after     34      32
## 3      4 method_A     Utrecht  after     32      37
## 4     17 method_B     Rotterdam after     30      33
## 5     10 method_A     Groningen after     29      35
## 6      2 method_B     Nijmegen after     28      40
## 7     11 method_B     Utrecht  after     28      31
## 8     16 method_A     Nijmegen after     28      30
## 9     25 method_A     Groningen after     28      32
## 10      5 method_B     Nijmegen after     27      29
## # i 50 more rows
```

If you want to sort first by `city` and then by the `statistics` test score, you just provide more than one column name. Each additional listed column will be used to break ties in the values of preceding columns:

```
teach <- arrange(teach, city, statistics) # sort for city then for statistics
teach <- arrange(teach, city, statistics) # sort for statistics and then for city
```



Check out the different types of sorting and use RStudio's object viewer to see differently sorted data frames.

11.3. Selecting & Arranging Variables: `select()`

11.3.1. Select a Variable



Selecting variables from a data frame was already easy in base-*R* (see [section 7.1.1](#)). *Tidyverse* offers the function `select()` for this and you will see that it's even easier. `select()` more powerful when it come to selecting multiple variables and it improves the readability of your code.

It's not uncommon to get datasets with a large amount of variables. In this case, the first challenge is often narrowing in on the variables you're actually interested in. `select()` allows you to rapidly zoom in on a useful subset using operations based on the names of the variables.

Select the variable for our data frame `teach`:

```
select(teach, cog_psy)
```

```
## # A tibble: 60 x 1
##   cog_psy
##   <dbl>
## 1     35
## 2     32
## 3     37
## 4     33
## 5     35
## 6     40
## 7     31
## 8     30
## 9     32
## 10    29
## # i 50 more rows
```

To select multiple variables just add further variables names as argument, separated by a comma:

```
select(teach, cog_psy, student, city)
```

```
## # A tibble: 60 x 3
##   cog_psy student city
##   <dbl>   <dbl> <chr>
## 1     35     28 Leiden
## 2     32     19 Rotterdam
## 3     37      4 Utrecht
## 4     33     17 Rotterdam
## 5     35     10 Groningen
## 6     40      2 Nijmegen
## 7     31     11 Utrecht
## 8     30     16 Nijmegen
## 9     32     25 Groningen
## 10    29      5 Nijmegen
## # i 50 more rows
```

11.3.2. Re-arranging the Order of Variables

As you see the variables appear in the order you mentioned them in the command. In other words, you can use `select` also to re-arrange the order of your columns.

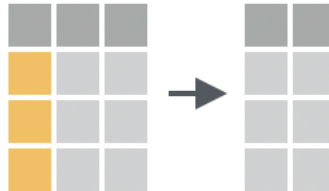
When re-arranging the order, another option is to use the `everything()` helper, which stands for all remaining variable. This is useful if you have a handful of variables you'd like to move to the start (or end) of the data frame.

```
select(teach, city, everything())
```

```
## # A tibble: 60 x 6
##   city      student teaching_method time  statistics cog_psy
##   <chr>      <dbl> <chr>          <chr>      <dbl>   <dbl>
## 1 Leiden      28 method_A      after        37     35
## 2 Rotterdam   19 method_A      after        34     32
## 3 Utrecht      4 method_A      after        32     37
## 4 Rotterdam   17 method_B      after        30     33
## 5 Groningen   10 method_A      after        29     35
## 6 Nijmegen     2 method_B      after        28     40
## 7 Utrecht    11 method_B      after        28     31
## 8 Nijmegen    16 method_A      after        28     30
```

```
## 9 Groningen      25 method_A      after      28      32
## 10 Nijmegen      5 method_B      after      27      29
## # i 50 more rows
```

11.3.3. Dropping a Variable



If you want to exclude a column you just write `-` followed by the variable name:

```
select(teach, -city)
```

```
## # A tibble: 60 x 5
##   student teaching_method time  statistics cog_psy
##   <dbl> <chr>          <chr>      <dbl>    <dbl>
## 1      28 method_A      after        37      35
## 2      19 method_A      after        34      32
## 3       4 method_A      after        32      37
## 4      17 method_B      after        30      33
## 5      10 method_A      after        29      35
## 6       2 method_B      after        28      40
## 7      11 method_B      after        28      31
## 8      16 method_A      after        28      30
## 9      25 method_A      after        28      32
## 10      5 method_B      after        27      29
## # i 50 more rows
```

11.3.4. Select a Range of Variables

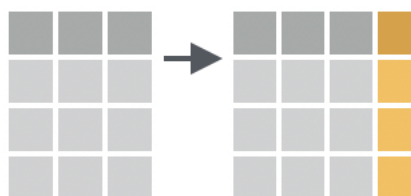


If you want to select all variable from column `city` to column `statistics`, it's very handy to use this `:`-operator. This operator only works for variable names for you use *Tidyverse*.

```
select(teach, city:statistics)
```

```
## # A tibble: 60 x 3
##   city      time statistics
##   <chr>    <chr>      <dbl>
## 1 Leiden   after         37
## 2 Rotterdam after         34
## 3 Utrecht  after         32
## 4 Rotterdam after         30
## 5 Groningen after         29
## 6 Nijmegen after         28
## 7 Utrecht  after         28
## 8 Nijmegen after         28
## 9 Groningen after         28
## 10 Nijmegen after         27
## # i 50 more rows
```

11.4. Add new variables: mutate()



Very often you want to add new columns that are functions of existing columns. That's the job of `mutate()`.

`mutate()` always adds new columns at the end of your dataset. For instance, if you want to calculate the test scores (i.e., number of correct answers) into percentage, we have multiple the score with $\frac{N_{\text{questions}}}{100}$? We know the test had 40 questions, thus:

```
teach_extended <- mutate(teach, percent_statistics = statistics / 40 * 100)
teach_extended
```

```
## # A tibble: 60 x 7
##   student teaching_method city      time  statistics cog_psy percent_statistics
##   <dbl> <chr>          <chr>    <chr>    <dbl>    <dbl>          <dbl>
## 1      28 method_A      Leiden  after      37      35           92.5
## 2      19 method_A      Rotterdam after      34      32           85
## 3       4 method_A      Utrecht  after      32      37           80
## 4      17 method_B      Rotterdam after      30      33           75
## 5      10 method_A      Groningen after      29      35           72.5
## 6       2 method_B      Nijmegen after      28      40           70
## 7      11 method_B      Utrecht  after      28      31           70
## 8      16 method_A      Nijmegen after      28      30           70
## 9      25 method_A      Groningen after      28      32           70
## 10     5 method_B      Nijmegen after      27      29           67.5
## # i 50 more rows
```

You probably remember from last week, you can also do this with base *R*:

```
teach$percent_statistics <- teach$statistics / 40 * 100
```

However, `mutate()` becomes really convenient (and again produces simpler code), if you need to add several columns. Just like `arrange()` or `select()`, you can add further variables by adding another argument to the function call (separated with a comma):

```
teach_extended <- mutate(teach,
  percent_statistics = statistics / 40 * 100,
  percent_cog_psy = cog_psy / 40 * 100)
```

Check out how the resulting data frame `teach_extended` looks like. Remember that when you're in RStudio, the easiest way to see all the columns is the object viewer using `View(teach_extended)`.

Hint for data frames with many variables: To check a data frame with many variables in the console, the function `glimpse()` produces a nice vertical overview of all columns and the first data.

glimpse

```
glimpse(teach_extended)
```

```
## Rows: 60
## Columns: 8
## $ student      <dbl> 28, 19, 4, 17, 10, 2, 11, 16, 25, 5, 1, 8, 13, 22, ~
## $ teaching_method <chr> "method_A", "method_A", "method_A", "method_B", "me-
## $ city          <chr> "Leiden", "Rotterdam", "Utrecht", "Rotterdam", "Gro-
## $ time          <chr> "after", "after", "after", "after", "after", "after-
## $ statistics     <dbl> 37, 34, 32, 30, 29, 28, 28, 28, 28, 27, 26, 26, 26,-
## $ cog_psy        <dbl> 35, 32, 37, 33, 35, 40, 31, 30, 32, 29, 32, 36, 40,-
## $ percent_statistics <dbl> 92.5, 85.0, 80.0, 75.0, 72.5, 70.0, 70.0, 70.0, 70.-
## $ percent_cog_psy  <dbl> 87.5, 80.0, 92.5, 82.5, 87.5, 100.0, 77.5, 75.0, 80-
```



Take the command above and calculate and add further variables. We want to have also for each row

- the average score of both test (**mean_score**) and
- the average percentage correct in both tests (**percent_correct_both**).

Solution

```
teach_extended <- mutate(teach, percent_statistics = statistics/40*100,
  percent_cog_psy = cog_psy/40*100,
  mean_score = (statistics+cog_psy)/2,
  percent_correct_both = mean_score/40*100)
glimpse(teach_extended)
```

```
## Rows: 60
## Columns: 10
## $ student      <dbl> 28, 19, 4, 17, 10, 2, 11, 16, 25, 5, 1, 8, 13, 22,-
## $ teaching_method <chr> "method_A", "method_A", "method_A", "method_B", "-
## $ city          <chr> "Leiden", "Rotterdam", "Utrecht", "Rotterdam", "G-
## $ time          <chr> "after", "after", "after", "after", "after", "aft-
## $ statistics     <dbl> 37, 34, 32, 30, 29, 28, 28, 28, 28, 27, 26, 26, 2-
## $ cog_psy        <dbl> 35, 32, 37, 33, 35, 40, 31, 30, 32, 29, 32, 36, 4-
## $ percent_statistics <dbl> 92.5, 85.0, 80.0, 75.0, 72.5, 70.0, 70.0, 70.0, 7-
## $ percent_cog_psy  <dbl> 87.5, 80.0, 92.5, 82.5, 87.5, 100.0, 77.5, 75.0, ~
## $ mean_score      <dbl> 36.0, 33.0, 34.5, 31.5, 32.0, 34.0, 29.5, 29.0, 3-
## $ percent_correct_both <dbl> 90.00, 82.50, 86.25, 78.75, 80.00, 85.00, 73.75, ~
```

12. Summarizing & Aggregating Data

12.1. Aggregating data with summarise()

Create summary statistics for the dataset can be done with the command `summarise()`. Check the data transformation [cheat sheet](#) for various summary functions. Some common ones are: `mean()`, `sd()`, `n()`, `sum()` or `quantile()`.

For instance,

```
summary_df <- summarise(teach,
  n_obs=n(),
  m_statistics = mean(statistics),
  m_cog_psy = mean(cog_psy),
  useless = sum(statistics+cog_psy) )
print(summary_df)
```

```
## # A tibble: 1 x 4
##   n_obs m_statistics m_cog_psy useless
##   <int>      <dbl>      <dbl>   <dbl>
## 1     60        15.4        18.2    2017
```

As you see this command generates a data frame (that we save in `summary_df`) with the number of observations (`n_obs`), the mean grade for statistics and cognitive psychology (`m_statistics` & `m_cog_psy`) and a `useless` variable that is sum off all scores in statistics and cognitive psychology. The mean grade in `statistics` is lower than in `cog_psy`.

Ok, fine, but you may think now: “I could get the same information much simpler with the base-*R*. Why shall I use this strange `summarise()` thing?” The actual power of it becomes clear if used together with the `group_by()` command described in the next section.

12.2. Grouping data by `group_by()`

If you want that all *Tidyverse*-operations will be done separately for different parts of a tibble or data frame, you can group your data and use a grouped tibble. You group a data with the `group_by()` command:

```
teach_grouped <- group_by(teach, city)
print(teach_grouped)
```

```
## # A tibble: 60 x 6
## # Groups:   city [5]
##   student teaching_method city      time  statistics cog_psy
##   <dbl>   <chr>          <chr>   <chr>      <dbl>   <dbl>
## 1      28 method_A      Leiden  after        37      35
## 2      19 method_A      Rotterdam after        34      32
## 3       4 method_A      Utrecht  after        32      37
## 4      17 method_B      Rotterdam after        30      33
## 5      10 method_A      Groningen after        29      35
## 6       2 method_B      Nijmegen after        28      40
```



```
## 7      11 method_B      Utrecht  after      28      31
## 8      16 method_A      Nijmegen after      28      30
## 9      25 method_A      Groningen after     28      32
## 10     5 method_B      Nijmegen after     27      29
## # i 50 more rows
```

You actually see nothing special happening and the new data frame `teach_grouped` looks exactly like the `teach`. The curial difference is for *R* that the new data frame is grouped. You see that in the second line of the output. The variable `city` defines the group. There are in total 5 groups.

If we now apply the `summarise()` command on the new group data, we get the aggregated data for separately for each group :

```
summary_df <- summarise(teach_grouped,    # use the new grouped df
  n_obs=n(),
  m_statistics = mean(statistics),
  m_cog_psy = mean(cog_psy),
  useless = sum(statistics+cog_psy) )
print(summary_df)
```

```
## # A tibble: 5 x 5
##   city      n_obs m_statistics m_cog_psy useless
##   <chr>    <int>      <dbl>      <dbl>    <dbl>
## 1 Groningen    12        15.9        20.2     433
## 2 Leiden       12        12.8        15.7     341
## 3 Nijmegen     12        14.8        17.9     393
## 4 Rotterdam    12        15.7        17.4     397
## 5 Utrecht      12        17.9        19.8     453
```

You see the aggregated data for each city. In each city we have 12 measures. In our fake dataset, the statistics grades seems to be on average higher in Utrecht than in Leiden.

Grouping for multiple variables

Like with the other *Tidyverse* functions you can add variables and further group the dataset. Let's get the aggregated data for each `teaching_method` and in each `city`:

```
teach_grouped <- group_by(teach, city, teaching_method)
summary_df <- summarise(teach_grouped,
  n_obs=n(),
  m_statistics = mean(statistics),
```

```

      m_cog_psy = mean(cog_psy),
      useless = sum(statistics+cog_psy) )
summary_df

```

```

## # A tibble: 14 x 6
## # Groups:   city [5]
##   city      teaching_method n_obs m_statistics m_cog_psy useless
##   <chr>      <chr>          <int>      <dbl>      <dbl>    <dbl>
## 1 Groningen control            2         12         11        46
## 2 Groningen method_A          6        17.8        21.7       237
## 3 Groningen method_B          4         15        22.5       150
## 4 Leiden     control            8         9.88        11.9       174
## 5 Leiden     method_A          4        18.5        23.2       167
## 6 Nijmegen   control            4        10.2        11.8        88
## 7 Nijmegen   method_A          2         18        19.5        75
## 8 Nijmegen   method_B          6        16.8        21.5       230
## 9 Rotterdam  control            4          8         11         76
## 10 Rotterdam method_A          4        20         23.5       174
## 11 Rotterdam method_B          4        19         17.8       147
## 12 Utrecht   control            2        11.5        12         47
## 13 Utrecht   method_A          4        19.5        22        166
## 14 Utrecht   method_B          6         19         21        240

```

You see that the conditions are not evenly distributed over the cities. In Leiden were more students (8) in the control condition with no teaching than in Utrecht (2). This explains possibly the difference in the statistic grades.



Calculate the mean statistics grade separate for all `teaching_methods` and for the time before and after the course.

Solution

```

summarize(group_by(teach, time, teaching_method), m_statistics = mean(statistics))

```

```

## # A tibble: 6 x 3
## # Groups:   time [2]
##   time      teaching_method m_statistics
##   <chr>      <chr>          <dbl>
## 1 after     control            10.2
## 2 after     method_A            28.5
## 3 after     method_B            24.1
## 4 before    control            9.7
## 5 before    method_A             9
## 6 before    method_B            11

```



Are those who are better in statistics also better in cognitive psychology?

Use a simple median-split comparison: Calculate the `median()` of all statistics grades and calculate the mean cognitive psychology grade for those who have higher grades than the median in statistics and those who have lower grades.

Solution

```
teach2 <- mutate(teach, good_in_stats = statistics > median(statistics))
teach2_grouped <- group_by(teach2, good_in_stats)
summarize(teach2_grouped, m_cog_psy = mean(cog_psy))
```

```
## # A tibble: 2 x 2
##   good_in_stats m_cog_psy
##   <lgl>         <dbl>
## 1 FALSE         10.7
## 2 TRUE          28.7
```

..and for the geeks in one line of code:

```
summarize(group_by(teach, statistics > median(statistics)),
           m_cog_psy = mean(cog_psy))
```

```
## # A tibble: 2 x 2
##   `statistics > median(statistics)` m_cog_psy
##   <lgl>                             <dbl>
## 1 FALSE                             10.7
## 2 TRUE                              28.7
```

TRUE is the group with a grade > median(statistics). FALSE is the low stats group.

Ungrouping a grouped data frame

You can remove the grouping of a data frame with `ungroup()`.

12.3. Summary

The last two chapters introduced the following six *Tidyverse* -commands, which all stems actually from the subpackage `dplyr` in the *tidyverse*.

- `filter()` Include or exclude certain cases (rows)
- `arrange()` Change the order of cases (rows)
- `select()` Include or exclude certain variables (columns)
- `mutate()` Create new variables (columns)
- `summarise()` Derive aggregate variables for groups of observations subsequent
- `group_by()` Organize the data into groups

These six functions are known as 'single table verbs' because they only operate on one table at a time. Although the operations of these functions may seem very simplistic, it's amazing what you can accomplish when you string them together: Some people claimed that 90% of data analysis can be reduced to the operations described by these six functions.

Again, you don't have to remember everything in these chapters - the important thing is that you know where to come and look for help when you need to do particular tasks. Being good at coding really is just being good at knowing what to copy and paste. You may also find the [cheat sheet](#) about data transformation useful at this point.

13. Structuring Tabular Data

13.1. Tidy data

For almost all statistical analyses, we have to our collected data in a tabular format. The previous chapters introduced how to handle tabular data. Please keep in mind that reshaping and re-aggregating data to have them in a format that is appropriate for your statistical software is not always that simple and takes typically much more time than the actual statistics. You have to think therefore carefully about the way how to represent you data in a table or how to structure your dataset, so that it can be efficiently processed in *R*.

Tabular data that are well structured and in good shape for data analysis have been called by Wickham (2014¹) **tidy data**. This term, by the way, also explains why Wickham labelled his famous *R* package collection *Tidyverse*.

It is very important to notice, that you can represent the same data in multiple ways. The *tidyverse* package comes with example data (`table`, `table2`, `table3`, `table4a`, `table4b`) that illustrate this. The tables below show the same data organised in four different ways. Each dataset shows the same values of four variables `country`, `year`, `population`, and `cases`, but each dataset organises the values in a different way.

Table 4: Example data 'table1'

country	year	cases	population
Afghanistan	1999	745	19987071
Afghanistan	2000	2666	20595360
Brazil	1999	37737	172006362
Brazil	2000	80488	174504898
China	1999	212258	1272915272

¹Wickham, H. (2014). Tidy Data. *Journal of Statistical Software*, 59(10), 1–23. doi: [10.18637/jss.v059.i10](https://doi.org/10.18637/jss.v059.i10)

Table 5: ‘table2’ and ‘table3’ contain exactly the same data as ‘table1’

country	year	type	count
Afghanistan	1999	cases	745
Afghanistan	1999	population	19987071
Afghanistan	2000	cases	2666
Afghanistan	2000	population	20595360
Brazil	1999	cases	37737
Brazil	1999	population	172006362
Brazil	2000	cases	80488
Brazil	2000	population	174504898
China	1999	cases	212258
China	1999	population	1272915272
China	2000	cases	213766
China	2000	population	1280428583

country	year	rate
Afghanistan	1999	745/19987071
Afghanistan	2000	2666/20595360
Brazil	1999	37737/172006362
Brazil	2000	80488/174504898
China	1999	212258/1272915272
China	2000	213766/1280428583

China	2000	213766	1280428583
-------	------	--------	------------

Same data but differently structured:

Or spread across two tables:

These are all representations of the same underlying data, but they are not equally easy to use. The first dataset—the tidy dataset—will be much easier to work with inside *R*.

There are three interrelated rules which make a dataset tidy:

1. Each variable must have its own column.
2. Each observation must have its own row.
3. Each value must have its own cell.

In our example, table 4 (data frame: `table1`) is the only representation where each column is a variable. Figure 5 also depicts this principles.

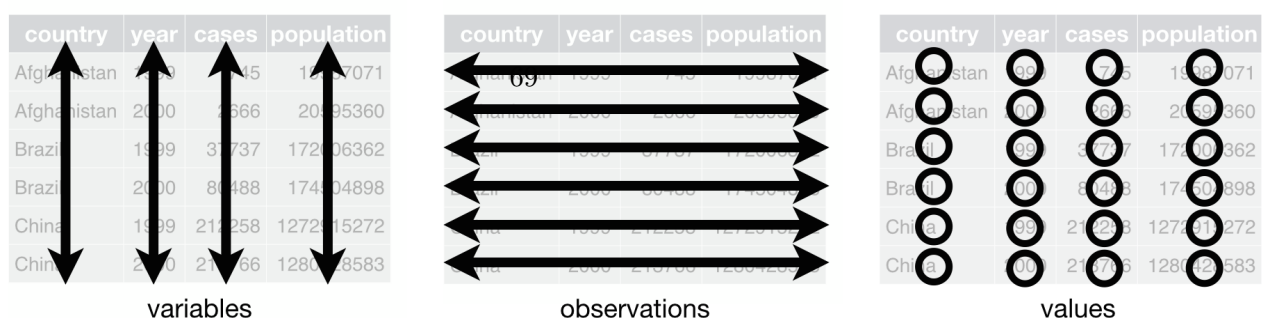


Table 8: The data spread over two tables: ‘table4a’ (left) and ‘table4b’ (right). ‘table4a’ contains the cases and ‘table4b’ the populations.

country	1999	2000
Afghanistan	745	2666
Brazil	37737	80488
China	212258	213766

country	1999	2000
Afghanistan	19987071	20595360
Brazil	172006362	174504898
China	1272915272	1280428583

2. You have to clean up the messy data structure and resolve at least one of the two common problems:
 - a. One variable might be spread across multiple columns.
 - b. One observation might be scattered across multiple rows.

13.2. Long and Wide Data

Tidy data can be organized in different ways. A common problem is for instance that you can treat different measurements as either

- different variables or
- as the same variable under different condition.

That is, sometimes it’s not easy to say what the columns/variables in your dataset really should be. In data science, we distinguish between long and wide data structures. Let’s have a look at an example. The dataset `weight` comprises the weights for four overweighted subjects before and after a diet and their age. There are two representations of the same data possible:

As you see in the **long data** format at the **right**, the time is coded as a variable and gets thus its own column. If you intend to do, for instance, a one-way factorial ANOVA with the independent variable `time` and the dependent variable `weight`, this long format directly illustrates the involved variables or factors (IV & DV). And in fact, most functions that do inferential statistics in *R* require this type of long data representation (e.g., the package `afex` for ANOVAs).

In contrast, in the **wide data** format at the **left** all measurements that belong to one person/case get their variable (in our example, `before` and `after`). One person/case is not spread across multiple rows. *SPSS* uses, for instance, this representation and needs always a *wide format* for all repeated measurements. That is, a wide data format represents all repeated measurements of the same person/case (such as weight at time 1 & 2) in different columns/variables. Only between subject variables, such as the `age`, get in the wide data format its own column. By the way: also the *t*-test function in *R* (`t-test()`) needs a wide data format.

13.2.1. Pivoting functions

Tidyverse offers the functions `pivot_longer()` and `pivot_wider()` to convert data between the two formats. For example, the file `weight.csv` [data folder](#) contains the data

Table 11: The same tidy data structures in a wide (left) or long (right) format.

Subject	Age	Before	After
Peter	23	105.0	98.5
Frank	29	102.5	97.5
Bob	34	96.5	95.5
Jim	25	91.0	87.0

Subject	Age	Time	Weight
Peter	23	Before	105.0
Peter	23	After	98.5
Frank	29	Before	102.5
Frank	29	After	97.5
Bob	34	Before	96.5
Bob	34	After	95.5
Jim	25	Before	91.0
Jim	25	After	87.0

```
## # A tibble: 4 x 4
##   Subject    Age Before After
##   <chr>    <dbl> <dbl> <dbl>
## 1 Peter      23   105   98.5
## 2 Frank      29  102.   97.5
## 3 Bob        34   96.5   95.5
## 4 Jim        25    91    87
```

Wide to long It's simple and straightforward to convert the wide data to a long format:

```
weight_long <- pivot_longer(weight, c("Before", "After"), names_to="Time", values_to="Weight")
```

Please note, the second argument of `pivot_longer()` is the columns to be rearranged and has to be a vector of the names (`c(...)`). The single names of the columns have to be in quotes ("`...`"). The resulting data frame is now in long format:

```
print(weight_long)
```

```
## # A tibble: 8 x 4
##   Subject    Age Time    Weight
##   <chr>    <dbl> <chr>    <dbl>
```

```
## 1 Peter      23 Before 105
## 2 Peter      23 After  98.5
## 3 Frank      29 Before 102.
## 4 Frank      29 After  97.5
## 5 Bob        34 Before 96.5
## 6 Bob        34 After  95.5
## 7 Jim        25 Before 91
## 8 Jim        25 After  87
```

Long to wide The function `pivot_wider()` does the opposite, that is, it transforms a long data format into a wide format.



Try to figure out how `pivot_wider()` works and use this function to convert the data frame `weight_long` back to a wide format.

Solution

```
wide_again <- pivot_wider(weight_long, names_from="Time", values_from="Weight")
print(wide_again)
```

```
## # A tibble: 4 x 4
##   Subject Age Before After
##   <chr>   <dbl> <dbl> <dbl>
## 1 Peter    23   105   98.5
## 2 Frank    29  102.   97.5
## 3 Bob     34   96.5   95.5
## 4 Jim     25    91    87
```

Hint: Further information about pivoting can be found in the build-in help of *R* if you type `vignette("pivot")` and look at the help panel in RStudio.



Old versions of *Tidyverse* (<1.0) don't have this pivoting functions and used instead the functions `gather()` and `spread()`. The old functions still work, but I suggest to use the new `pivot_wider()` and `pivot_longer()` functions.

Part IV.

Data Processing & Visualizing

14. Missing Values

R has a distinct data type to indicate that an object, a variable or aN element of a vector has no value or that this data point is missing.

14.1. NA: Not Available

NA stands for “Not Available” and is used to indicate a missing value. For instance, if you have a vector of the body heights of three subjects and, let’s say, you don’t know the height of the second person, it should look like this in *R*:

```
height = c(1.69, NA, 1.82)
```

Importantly, **any operation and any comparison with NA has the result NA**. That is, if you make an element-wise operation or comparison with the vector, *R* returns always NA for any missing elements.

```
height * 100 # height in cm
```

```
## [1] 169 NA 182
```

```
large_guys = height > 1.80 # find all the large guys  
print(large_guys)
```

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

If you want to figure out which elements in a vector is NA, you have to use `is.na()` function:

```
is.na(height)
```

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

Since you use the `!`-sign in *R* to make a negation (like in the comparison “*is not equal*” `!=`; see section 4.4.2 about logicals or section 4.5 about comparisons), you can use this to find all elements in a vector that are **NOT** NA:

```
!is.na(height)
```

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

Aggregating/processing vectors with missing values Since all operations with missing values return NA, all functions that calculate a single value from a vector (e.g. `sum()`, `mean()` or `sd()`) return always NA, if only one single value in the vector is NA.

```
mean(height)
```

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

That is a problem, because we typically just want to ignore NA and calculate the results without the missing value. To do so, we have to exclude all NA values. We could do this manually by selecting only the elements that are not NA (e.g. using `filter()` or base-*R* [subsetting](#)). However, most vector-base functions in *R* offer the option to remove all NAs. This option is called `na.rm` and if it's set to TRUE, *R* excludes the NA before doing the calculation:

```
mean(height, na.rm = TRUE)
```

```
## [1] 1.755
```

manually deleting NA

```
# manually deleting NA
mean(height[!is.na(height)])
```

```
## [1] 1.755
```

```
# but this is easier, right?
mean(height, na.rm = TRUE)
```

```
## [1] 1.755
```

14.2. Missings in data frames

The datasets that we have used in the previous chapter were always complete and did not contain any missing data point. In real life, however, you often encounter the situation that you are missing some measurements in some variables. In other words, not all subjects provide a measurement in all columns. This might cause problems for the data processing and we need to deal with that issue.

The **data folder** contains a second (more realistic) version of the teaching method dataset called `teaching_method2.csv`. Let's have a look at it.

```
teach <- read_csv("data/teaching_method2.csv")
summary(teach)
```

```
##      student      teaching_method      city      time
## Min.   : 1.0    Length:60          Length:60    Length:60
## 1st Qu.: 8.0    Class :character    Class :character    Class :character
## Median :15.5    Mode  :character    Mode  :character    Mode  :character
## Mean   :15.5
## 3rd Qu.:23.0
## Max.   :30.0
##
##      statistics      cog_psy
## Min.   : 4.00    Min.   : 5.00
## 1st Qu.: 9.00    1st Qu.:10.00
## Median :12.00    Median :12.00
## Mean   :15.39    Mean   :18.41
## 3rd Qu.:23.00    3rd Qu.:31.00
## Max.   :37.00    Max.   :40.00
## NA's   :3        NA's   :2
```

As you see in the summary, we have 3 missing values the variable `statistics` and `cog_psy`. In the data viewer it look like this.

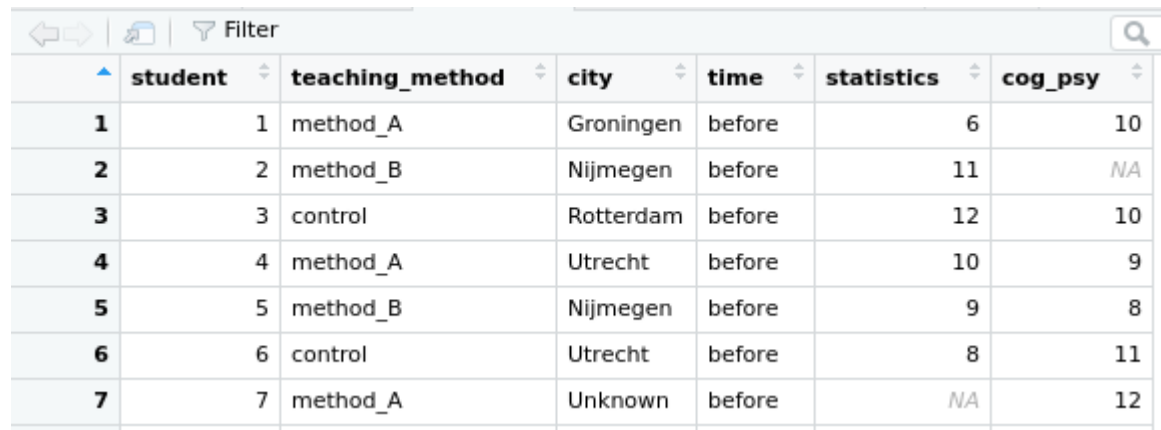


Calculate the mean of `statistics` and `cog_psy` using the `summarise()` function of *Tidyverse*?

Solution

```
summarise(teach, m_stats = mean(statistics, na.rm=TRUE),
           m_cog_psy = mean(cog_psy, na.rm=TRUE))
```

```
## # A tibble: 1 x 2
##   m_stats m_cog_psy
##   <dbl>    <dbl>
## 1   15.4      18.4
```



	student	teaching_method	city	time	statistics	cog_psy
1	1	method_A	Groningen	before	6	10
2	2	method_B	Nijmegen	before	11	NA
3	3	control	Rotterdam	before	12	10
4	4	method_A	Utrecht	before	10	9
5	5	method_B	Nijmegen	before	9	8
6	6	control	Utrecht	before	8	11
7	7	method_A	Unknown	before	NA	12

Figure 6: Screen shot of the data frame displayed in RStudio Viewer

14.2.1. Defining missing values

As you have seen above, when using `read_csv()`, *R* detected automatically the missing values in the variables `statistics` and `cog_psy` and inserted instead `NA`. This was the case because there was simply no value in the data file.

However, very often scientists use specific values to indicate in their raw data that there is a missing value (e.g. `??`, `None`, `Missing`, `-99`). For example, if you look carefully on the screenshot of the data viewer above (figure 6), the `city` in row 7 is `"Unknown"`, which is obviously a missing value. *R* doesn't know that, of course, and we have to indicate that the data point `"Unknown"` represents a missing value and not a Dutch city.

Data reading functions have the optional argument `na`, which calls for a vector of the values/labels that are representing missing values in this dataset. For example:

```
teach <- read_csv("data/teaching_method2.csv", na = c("Unknown"))
```


As you see, `city` in row 7 is now a missing.

If you have multiple values that represent a missing value, it could look like this:

```
df <- read_csv("demo.csv", na = c("Unknown", "??", "None", -99, "no idea", "who knows"))
```

14.2.2. Drop rows with missing values

Tidyverse has the function `drop_na()` to remove rows from a data frame that contain missing values. This command excludes all(!) rows, in which one or more variables are `NA`:



	student	teaching_method	city	time	statistics	cog_psy
1	1	method_A	Groningen	before	6	10
2	2	method_B	Nijmegen	before	11	NA
3	3	control	Rotterdam	before	12	10
4	4	method_A	Utrecht	before	10	9
5	5	method_B	Nijmegen	before	9	8
6	6	control	Utrecht	before	8	11
7	7	method_A	NA	before	NA	12

Figure 7: Screen shot of the data frame displayed in RStudio Viewer. 'Unknown' is now defined as missing.

```
teach_no_na <- drop_na(teach)
```

Often you only want to remove rows, in which only certain variables are NA. Like in all *Tidyverse* functions, you can specify variables to which it a certain operations should be apply. For instance, if you want to remove all rows in which the variables `city` or `statistics` are NA.

```
drop_na(teach, city, statistics)
```



Calculate the again mean of `statistics` and `cog_psy` with `summarise()`, but this time use also `drop_na()`? Try to do it in one line of code.

Solution

```
summarise(drop_na(teach), m_stats = mean(statistics),
           m_cog_psy = mean(cog_psy))
```

```
## # A tibble: 1 x 2
##   m_stats m_cog_psy
##   <dbl>    <dbl>
## 1   15.6     18.4
```

14.2.3. Replacing missing values

If we know that a missing value in a certain variable actually stands for an particular meaningful value, we can replace all NAs values in that variable by using `mutate()` and `replace_na()`-function:

```
teach <- mutate(teach,
  city = replace_na(city, "Amsterdam"))
```

This replaces all missing values in the variable with “*Amsterdam*”. However, be careful and use `replace_na()` only, if you are absolutely sure that you can replace the missing values without messing up your data.

15. Factors

In *R*, factors are used to work with categorical variables. A categorical variable are often represented as text variables in raw data files (e.g. the words “*Male*” or “*Female*”).



A factor is defined as a variable that has a fixed and **known** set of possible values or factor levels.

15.1. Creating factors

Examples of categorical factors are the variables `teaching_method`, `time` and `city` in our dataset `teaching_method2.csv`. You see that the variables are character variables `<chr>` when reading the CSV-file into *R*.

```
teach <- read_csv("data/teaching_method2.csv", na = c("Unknown"))
glimpse(teach)
```

```
## Rows: 60
## Columns: 6
## $ student      <dbl> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, ~
## $ teaching_method <chr> "method_A", "method_B", "control", "method_A", "method~
## $ city          <chr> "Groningen", "Nijmegen", "Rotterdam", "Utrecht", "Nijm~
## $ time          <chr> "before", "before", "before", "before", "before", "bef~
## $ statistics     <dbl> 6, 11, 12, 10, 9, 8, NA, 9, 8, 10, 16, 11, 8, 12, 13, ~
## $ cog_psy        <dbl> 10, NA, 10, 9, 8, 11, 12, 12, 10, 12, 8, 13, 13, 8, 10~
```

We want to treat these variable as factors, because they have a fixed and known number of factor levels. To tell *R* that variables in a data frame (or tibble) are factors, we make use of the function `factor()` and `mutate()` to ‘replace’ a particular text variable with a factor representation of this variable:

```

teach_with_factors <- mutate(teach,
                             teaching_method=factor(teaching_method),
                             city=factor(city),
                             time=factor(time))
glimpse(teach_with_factors)

## Rows: 60
## Columns: 6
## $ student      <dbl> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, ~
## $ teaching_method <fct> method_A, method_B, control, method_A, method_B, contr~
## $ city          <fct> Groningen, Nijmegen, Rotterdam, Utrecht, Nijmegen, Utr~
## $ time          <fct> before, before, before, before, before, before, before~
## $ statistics     <dbl> 6, 11, 12, 10, 9, 8, NA, 9, 8, 10, 16, 11, 8, 12, 13, ~
## $ cog_psy        <dbl> 10, NA, 10, 9, 8, 11, 12, 12, 10, 12, 8, 13, 13, 8, 10~

```

The three variables are now factors (<fct>). The same could be achieved with base-*R*, but it is less convenient.

Making factors with base-*R* is less readable

```

teach_with_factors <- teach
teach_with_factors$teaching_method = factor(teach_with_factors$teaching_method)
teach_with_factors$city = factor(teach_with_factors$city)
teach_with_factors$time = factor(teach_with_factors$time)

```

15.2. Advantages of factors

The Advantages of using factors can be best illustrated with a typical example of a categorical variable in psychology: the *gender*. People often use numbers to code to genders and enter for instance for the females 1 and males with 2.

```

df_no_factor = data.frame(gender=c(1,2,3,2,1, 1,2,2,2,1))
print(df_no_factor)

```

```

##      gender
## 1         1
## 2         2
## 3         3
## 4         2
## 5         1
## 6         1
## 7         2
## 8         2
## 9         2
## 10        1

```

15.2.1. Predefined factor levels

One advantage of factors is that you can define the possible factor levels (like `...,levels=c(1, 2),...`) and when an row/observation has a values that does not fit the predefined levels, the values will be set automatically to NA. You can do it with the function argument `levels`.

```
df_factor <- mutate(df_no_factor,
  gender = factor(gender, levels=c(1, 2)))
print(df_factor)
```

```
##      gender
## 1         1
## 2         2
## 3      <NA>
## 4         2
## 5         1
## 6         1
## 7         2
## 8         2
## 9         2
## 10        1
```

Gender 3 became not not NA.

15.2.2. Meaningful factor labels

You can introduce useful factor labels (or descriptions) for each level, which make your analysis script and, importantly, the produced output more readable and more transparent.

Don't forget, often the factor levels are represented in a raw data file just by codes. This data is not self-explaining and I'm sure that you can't remember in a few weeks whether the females or the males have the value 1. I therefore **strongly** suggest to add self-explaining labels for all levels of the factor. You can do it with the function argument `labels`.

```
df_factor <- mutate(df_no_factor,
  gender = factor(gender, levels=c(1, 2), labels=c("female", "male")))
print(df_factor)
```



```
##      gender
## 1  female
## 2    male
## 3    <NA>
## 4    male
## 5  female
## 6  female
## 7    male
## 8    male
## 9    male
## 10 female
```

15.2.3. Statistics with categorical variables

Inferential statistics needs to know which variable is categorical and which is continuous. Almost all packages that do inferential statistics (see [Appendix B](#)) use factors to indicate categorical variables.

15.2.4. Nice output

The output of some functions is nicer and better organized. If we do a summary of the data frame without factors:

```
summary(df_no_factor)
```

```
##      gender
## Min.      :1.0
## 1st Qu.:1.0
## Median :2.0
## Mean     :1.7
## 3rd Qu.:2.0
## Max.     :3.0
```

and with factors,

```
summary(df_factor)
```

```
##      gender
## female:4
## male   :5
## NA's   :1
```

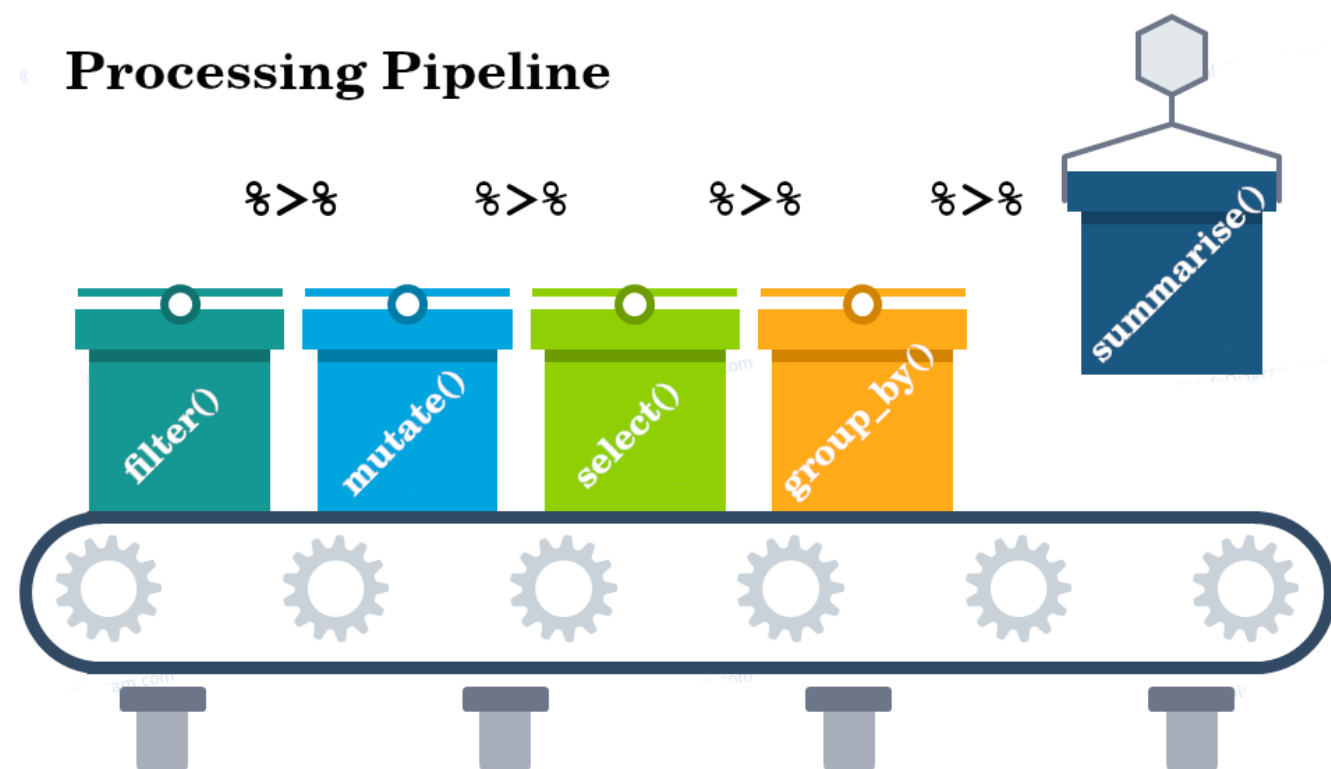
you see that only the summary of the factor gender is meaningful and it provide is need information of all factor levels and amount of observations in each level.



Factors are also very useful when you need to display or process character variables in a predefined non-alphabetical order, for instance, `c("Mon", "Tue", "Wed", "Thr", "Fri", "Sat", "Son")`. However, this aspect is less relevant for social sciences and it will be therefore not discussed here. (but see for instance [this tutorial](#) or the documentation of factors in *Tidyverse*, the [forcats package](#)).

16. Pipes

You have seen in the previous chapters that if you have a raw dataset you need to apply typically multiple operations and processing steps on a dataset. The resulting data of one operations, will be used a input data for the next operations. That is, when processing and analysing data, we typically setup a processing pipeline.



Pipes are a powerful tool for clearly expressing a sequence of multiple operations. Pipes are a way to order your code in a more readable format.

16.1. The pipe operator: %>%

Tidyverse has the operator %>%, which sends an object (e.g. data frame) as input for the following function:

For example, instead writing

```
# classical function call
drop_na(my_data)           # example a
mutate(my_data, new_var = a + b) # example b
```

you can write

```
# the same with pipes
my_data %>% drop_na()           # example a
my_data %>% mutate(new_var = a + b) # example b
```

The two versions of example a and b do exactly the same. It's just a different syntax. Instead having the data frame as first argument of the functions `drop_na()` or `mutate()` (first two example lines), you “push it” or “pipe it” to the following function with %>%. That is, *R* uses the forwarded object as input for function and you can (or have to!) omit mentioning the data inside the function call (last two example lines).



The package that defines pipes in *Tidyverse* is called `magrittr`.

16.2. Motivation for pipes

Why shall I use pipes? Pipes can be very handy, because you can not only send data to functions, but also the results of functions (i.e., the resulting data) to the following functions. You can build with this technology very efficiently a complex processing pipeline and your code remains well structured and readable (which this is always the aim of a good data science practice).

```
# classical function call
data_with_na <- drop_na(my_data)           # step a
mutate(data_with_na, new_var = a + b) # step b

# the same with pipes
my_data %>% drop_na() %>% mutate(new_var = a + b) # step a and b
```



Don't forget, pipes work only with functions that are developed for this purpose. Fortunately, this is the case for all *Tidyverse*-functions.

16.3. Building a processing pipeline

Let's have an example and use our dataset `teaching_method2.csv` (see section 11). Assume we want to calculate the mean percent correct in cognitive psychology for all cities and teaching methods. We want also to exclude the cities "*Leiden*" and "*Utrecht*". You know we have missing values and need to deal with it.



Try to do this analysis yourself, before you go on.

16.3.1. Intermediate steps

To do it, we need to read the data and process it in multiple steps. For each step we create a new data frame and use the previous data frame as input. You are already familiar with that approach.

One solution to the problem above looks, for example, like this:

```
#reading the data
teach <- read_csv("data/teaching_method2.csv", na = c("Unknown"))

#remove nan values
teach2 <- drop_na(teach)

# filtering the cities
teach3 <- filter(teach2, city != "Leiden", city != "Utrecht")

# calculating percent correct for cog.psy-
teach4 <- mutate(teach3, percent_cog_psy= cog_psy * 40/100)

# group data for city and teaching method
teach4_grouped <- group_by(teach4, city, teaching_method)

# making the summary
summary_df<- summarise(teach4_grouped,
                        m_cog_psy = mean(percent_cog_psy) )
```

We produced in total 6 data frames (`teach`, `teach2`, `teach3`, `teach4`, `teach4_grouped`, `summary_df`), but we are actually only interested in the last data frame, which look like this:

```
print(summary_df)

## # A tibble: 9 x 3
## # Groups:   city [3]
##   city      teaching_method m_cog_psy
##   <chr>      <chr>          <dbl>
## 1 Groningen control          4.4
## 2 Groningen method_A        7.84
## 3 Groningen method_B         9
## 4 Nijmegen control          4.7
## 5 Nijmegen method_A         7.8
## 6 Nijmegen method_B        9.28
## 7 Rotterdam control          4.4
## 8 Rotterdam method_A         9.4
## 9 Rotterdam method_B         7.1
```

All the other five intermediate data frames are not really wanted as result. This is very inefficient and occupies a lot of memory.

16.3.2. Overwrite the original

Instead of creating intermediate objects at each step, we could overwrite the original object:

```
teach <- read_csv("data/teaching_method2.csv", na = c("Unknown"))
teach <- drop_na(teach)
teach <- filter(teach, city != "Leiden", city != "Utrecht")
teach <- mutate(teach, percent_cog_psy = cog_psy * 40/100)
teach <- group_by(teach, city, teaching_method)
teach <- summarise(teach, m_cog_psy = mean(percent_cog_psy) )
```

This is less typing (and less thinking), so you're less likely to make mistakes. However, there are two problems:

1. Debugging is painful: if you make a mistake you'll need to re-run the complete pipeline from the beginning.
2. The repetition of the object being transformed (we've written `foofoo` six times!) obscures what's changing on each line.

16.3.3. Build a processing pipeline with pipe %>%

Finally, we can use the pipe:

```
summary_df <- read_csv("data/teaching_method2.csv", na = c("Unknown")) %>%
  drop_na() %>%
  filter(city != "Leiden", city != "Utrecht") %>%
  mutate(percent_cog_psy= cog_psy * 40/100) %>%
  group_by(city, teaching_method) %>%
  summarise(m_cog_psy = mean(percent_cog_psy) )
```

If you compare this code with the code in which we produced multiple *intermediate data frames*, I think you see the benefit. It does not only do the same computations, it's also faster executed (which you noticed however only if you have really big datasets).

```
print(summary_df)
```

```
## # A tibble: 9 x 3
## # Groups:   city [3]
##   city      teaching_method m_cog_psy
##   <chr>      <chr>          <dbl>
## 1 Groningen control          4.4
## 2 Groningen method_A        7.84
## 3 Groningen method_B         9
## 4 Nijmegen  control          4.7
## 5 Nijmegen  method_A        7.8
## 6 Nijmegen  method_B        9.28
## 7 Rotterdam control          4.4
## 8 Rotterdam method_A        9.4
## 9 Rotterdam method_B        7.1
```

17. Visualizing Data

This chapter will give merely a superficial overview on plotting and some examples how to create graphs in *R*. Unfortunately, we do not have the time to have a detailed discussion of data visualizations. **Data visualization will be not part of the exam.**

There are excellent resources online that nicely explain how to do graphs with *R* and *Tidyverse*. For instance:

- [Chapter 3](#) and [Chapter 28](#) of *R for Data Science*
- [R Graphics Cookbook](#)

- [ggplot2 documentation](#) and [ggplot2 cheat sheet](#)

It's often very useful find *R*-code of plots that are similar to a graph you want to create. You can then copy & paste the code, include your data and modify code that it suits your purposes. An overview of plots will be provided below. Websites with galleries of nice figures can be found online, for example:

- [The R Graph Gallery](#)
- [Top 50 ggplot2 Visualizations](#)
- [Plotly](#)

Simulating example data

For the example plots in this chapter, we use a simulated dataset with scores of two IQ tests (IQ_A & IQ_B) from 1000 students. 300 students study at “*University A*”, 500 at “*University B*” and 200 at “*University C*”.

The following code generates the data:

```
# Just copy & paste, you don't have to understand the code.
simulation <- data.frame(school = c(rep("University A", 300),
                                   rep("University B", 500),
                                   rep("University C", 200))) %>%
  mutate(school = factor(school),
         # random IQ data for each person
         IQ_A = rnorm(1000, 100, 15),
         # make people from University A more clever
         IQ_A = ifelse(school=="University A", IQ_A + 20, IQ_A),
         # random different outcome of IQ test B
         IQ_B = IQ_A + rnorm(1000, 0, 25) )
```



For those how are interested in the simulation. It makes the following assumptions:

- IQ scores are normally distributed with a mean of 100 and standard deviation of 15.
- The two IQ tests are strongly correlated (see last line of code)
- Students from University A are 20 IQ points more clever than those from B & C.

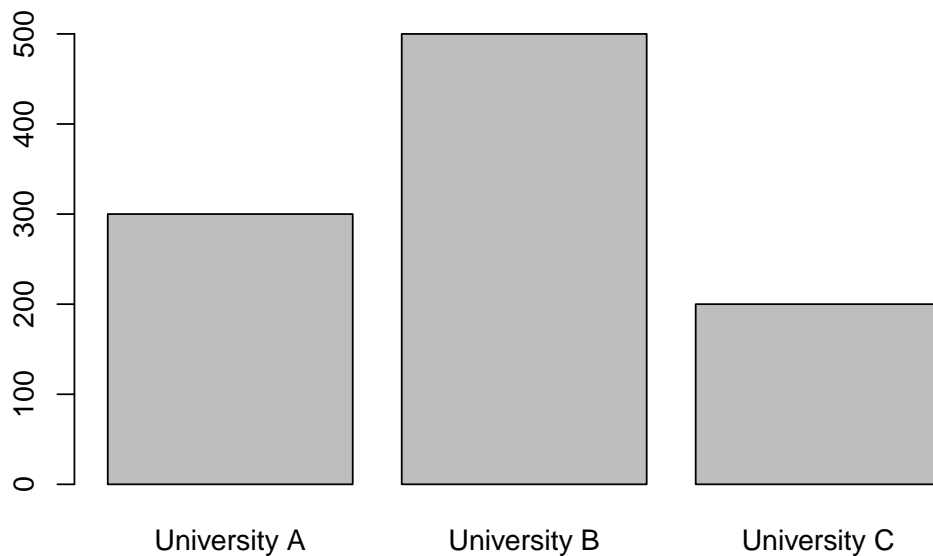
17.1. Plots with base R

R has some basic plotting functions, but they're somewhat difficult to use and aesthetically not very nice. They can be useful to have a quick look at data while you're working on a script, though. The function `plot()` usually defaults to a sensible type of plot, depending on whether the arguments `x` and `y` are categorical or continuous.

17.1.1. Plot with a categorical variable

`plot()` with a categorical factor variable (`school`) shows how often the factor levels occur.

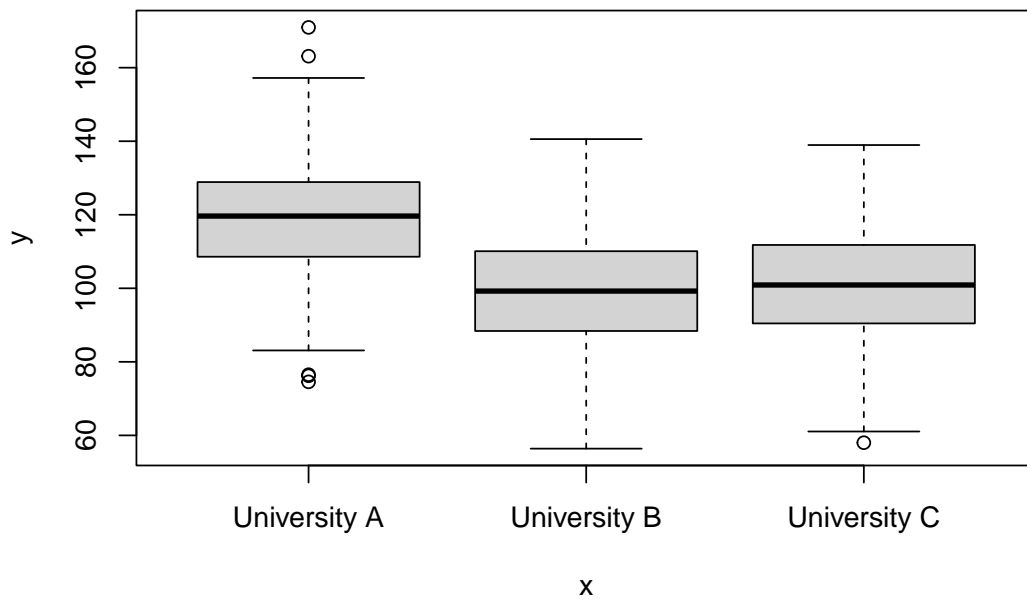
```
plot(x = simulation$school)
```



17.1.2. Plot a categorical and a continuous variable

`plot()` with a categorical variable `x` and a continuous variable `y` shows a box plot with the distribution of continuous data in the factor levels.

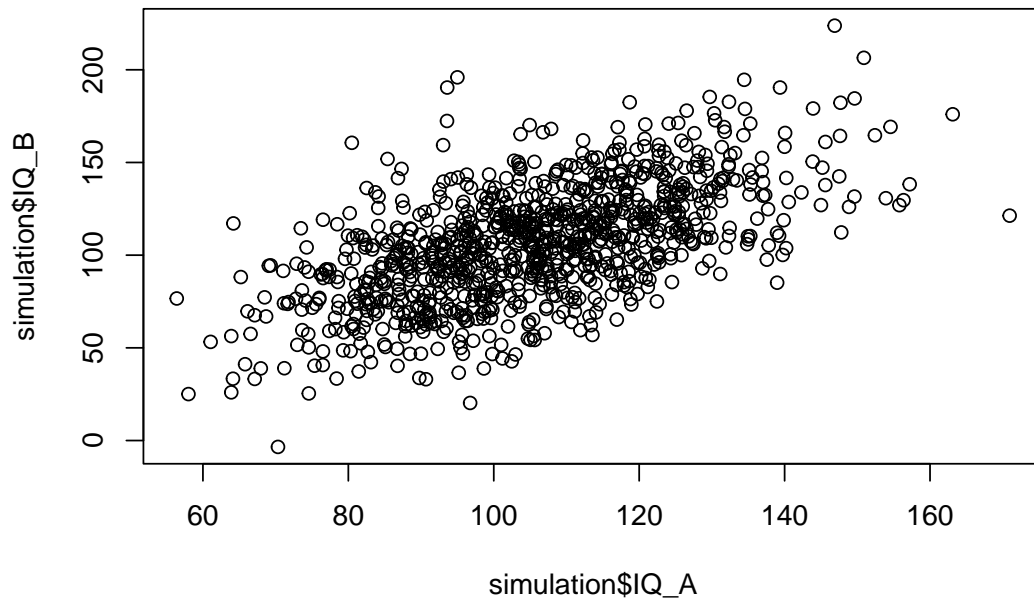

```
plot(x = simulation$school, y = simulation$IQ_A)
```



17.1.3. Plot two continuous variables

`plot()` with a categorical variable `x` and a continuous variable `y` shows a scatter plot of the two variables. You see that `IQ_A` and `IQ_B` are correlated.

```
plot(x = simulation$IQ_A, y = simulation$IQ_B)
```

**Excursus: Correlation coefficient**

By the way: To calculate the correlation coefficient r , you can use the function `cor()`.

```
cor(simulation$IQ_A, simulation$IQ_B)
```

```
## [1] 0.5945371
```

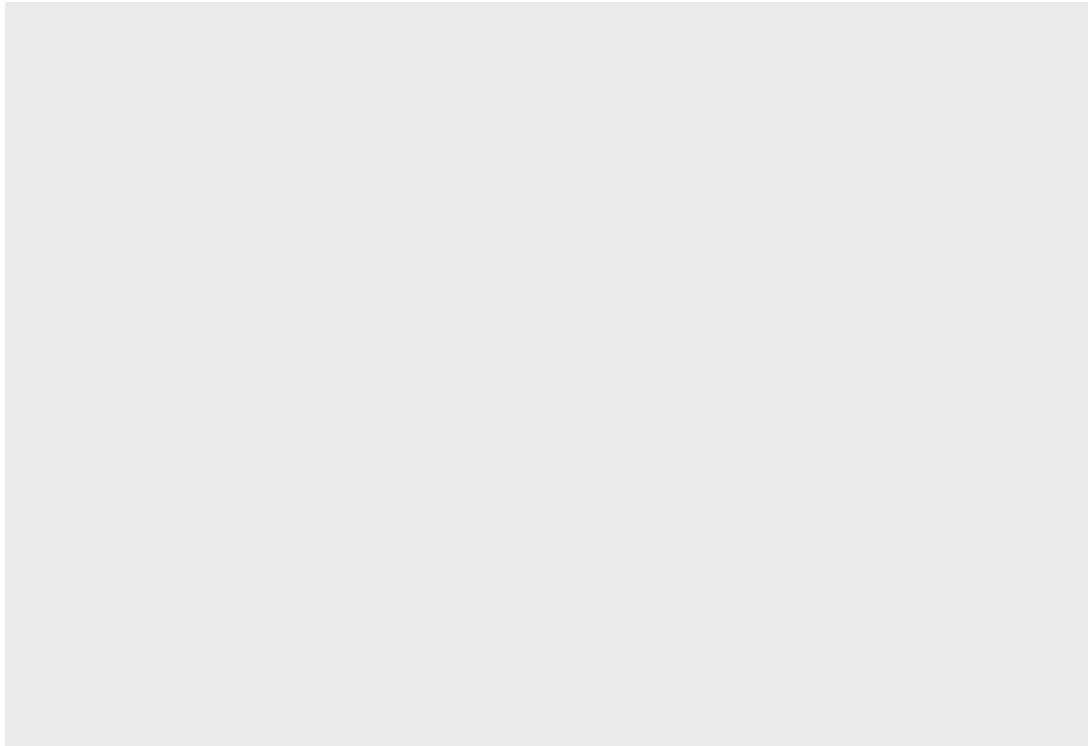
17.2. ggplot2: Plotting in Tidyverse

While the functions above are nice for quick visualisations, it's difficult to make publication-ready plots. The package `ggplot2`, which is also part of the *Tidyverse*, is one of the most common packages to visualise data in R.

`ggplot2` creates plots using a “grammar of graphics” where you add **geoms** in layers. It can be complex to understand, but it's very powerful once you have a mental model of how it works. The chapter will just illustrate the principle. Please have a look at the suggested literature above, if you want to know more.

Let's start with a totally empty plot layer created by the `ggplot()` function with no arguments.

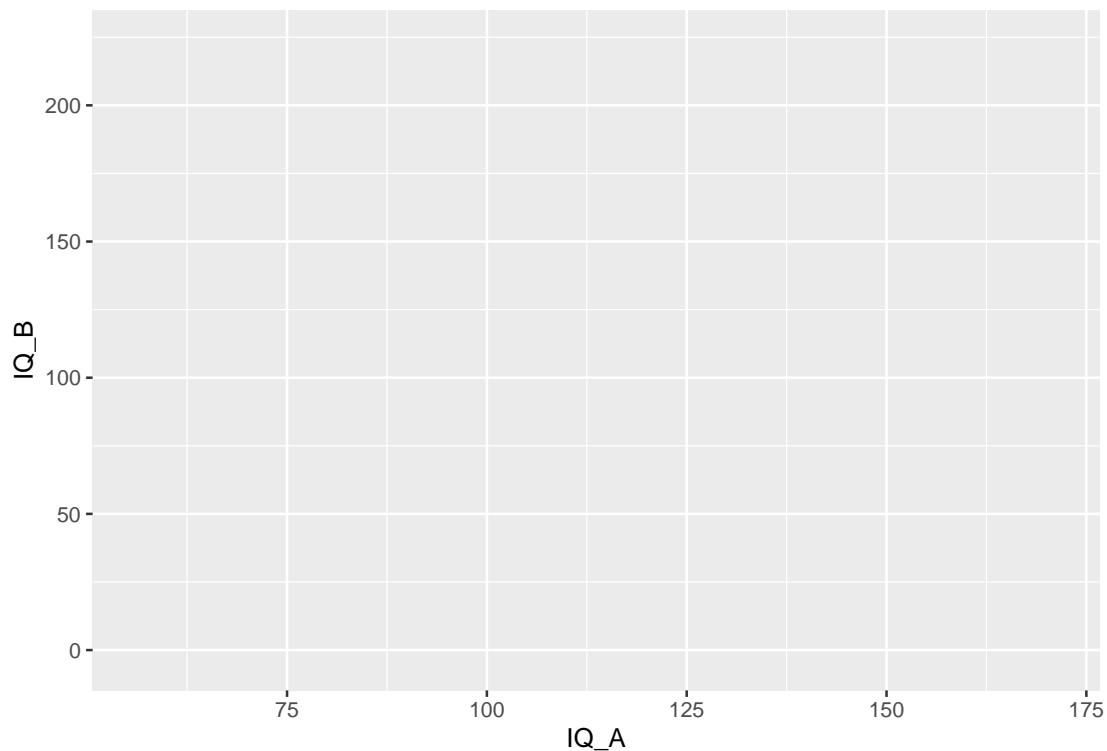
```
ggplot()
```



Like with all *Tidyverse* functions, the first argument to `ggplot()` is the data frame you want to plot. We use our simulated data `simulation`.

Aesthetic mappings: The second argument is the mapping that specifies which variables corresponds to which properties of the plot, such as the `x`-axis, the `y`-axis, line `colour` or `linetype`, point `shape`, or object `fill`. These mappings are called the *aesthetics* and are defined in *R* by the `aes()` function. Just adding this to the `ggplot()` function creates the labels and ranges for the `x` and `y` axes. They usually have sensible default values, given your data, but you can also change them.

```
ggplot(simulation, mapping = aes(x=IQ_A, y=IQ_B))
```

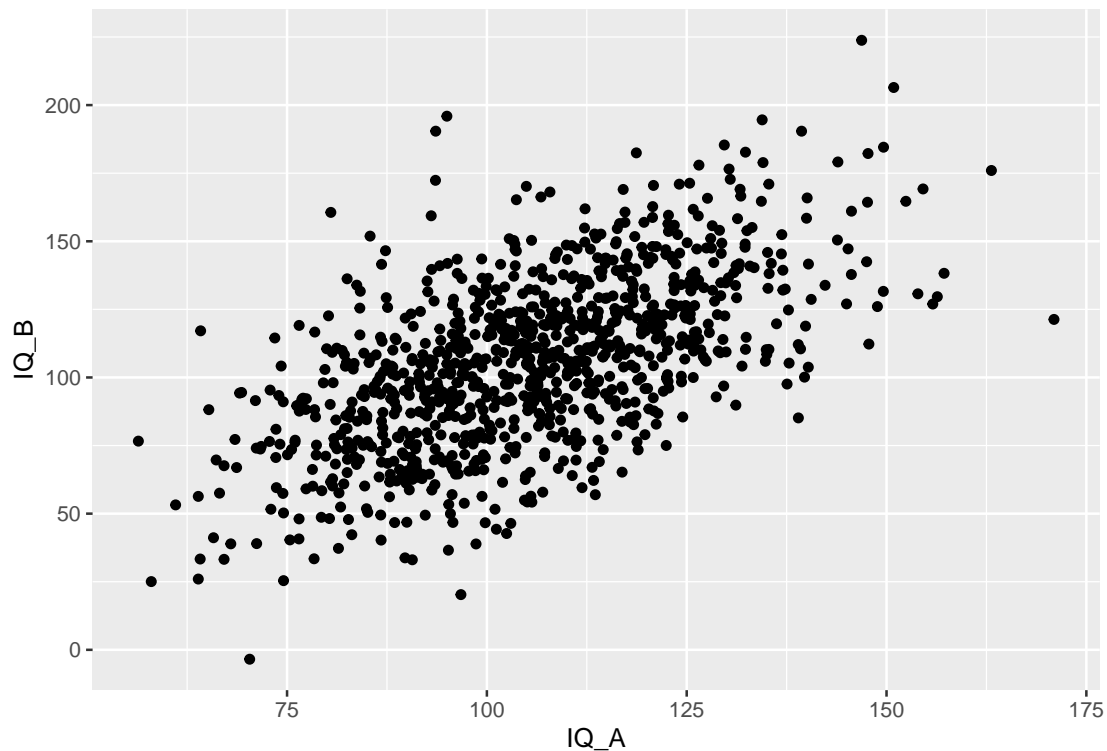


People often omit the argument name `mapping` and just put the `aes()` function directly as the second argument to `ggplot`. That's also fine. You can also omit `x` and `y` as arguments inside `aes()`, but I suggest to use argument names if you more than one variable, because it improves the readability.

The function above does not plot anything yet and you only see scaled axes with the variable names. *R* does not yet know *what* or better *how* to plot the data. This will be specified in `ggplot2` by “geoms” or plot styles. You literally add them with the `+` symbol. You can also add other plot attributes, such as labels, or change the theme and the font size (see the advanced literature).

If we add, for example, the `geom_point()`, we tell *R* to plot the data with the defined aesthetic mapping as point:

```
ggplot(simulation, mapping = aes(x=IQ_A, y=IQ_B)) + geom_point()
```



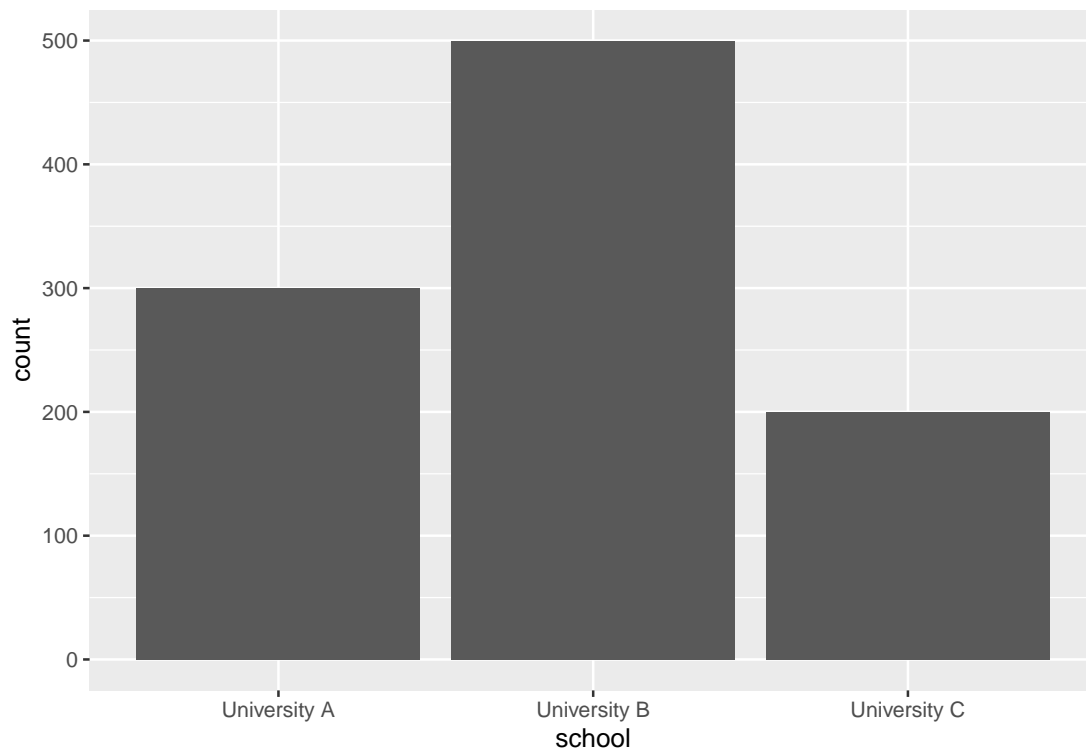
17.3. Geoms or plot types

There are many geoms, and they can take different arguments to customise their appearance. Lets have a look a common plot types. Look carefully at the code to see the different mappings, `aes()`, and geoms, `geom_?()`. This ways you get hopefully a bit of an inside how the functions work. As you know, if you want information about these functions, you can always look at the function documentation by typing `?function_name`.

17.3.1. Bar plot

Bar plot are done pity easy by adding (+) the geom `geom_bar()`:

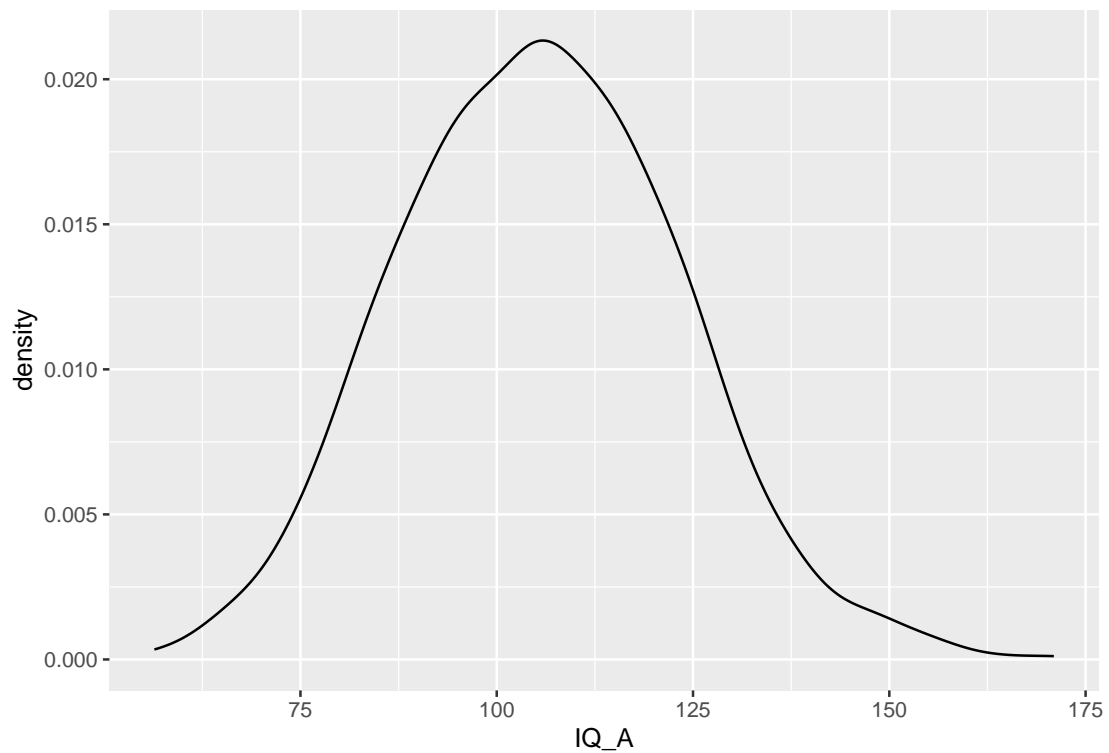
```
ggplot(simulation, aes(school)) +  
  geom_bar()
```



17.3.2. Density plots

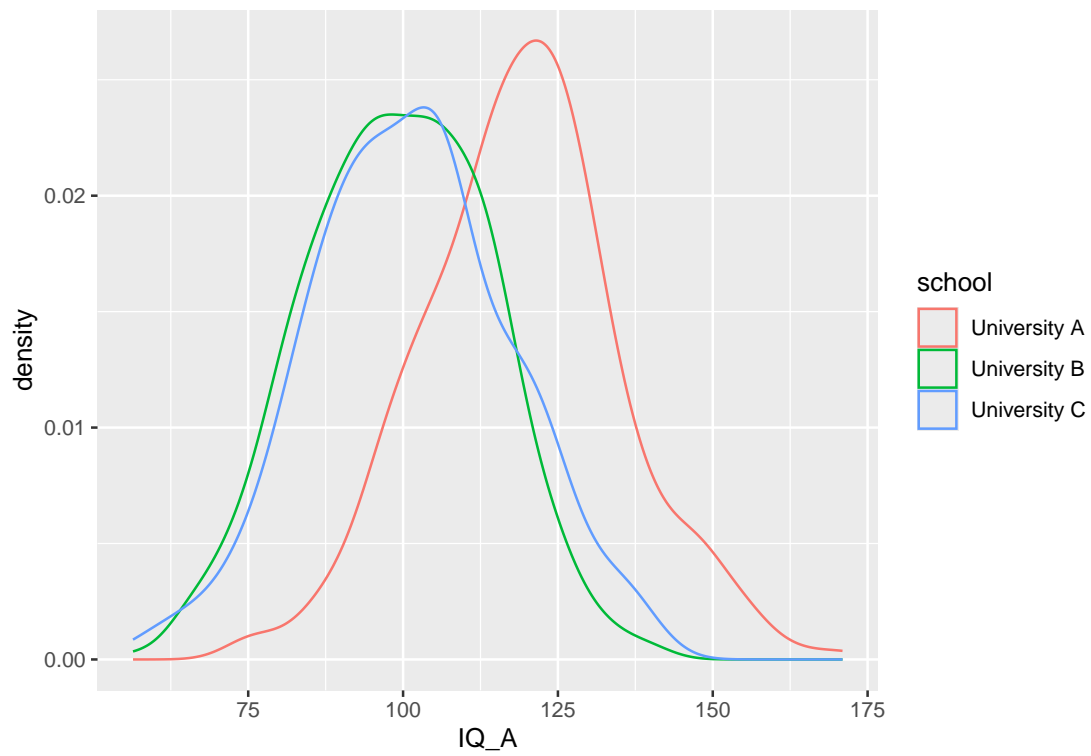
Density plots are good for one continuous variable, but only if you have a fairly large number of observations.

```
ggplot(simulation, aes(IQ_A)) +  
  geom_density()
```

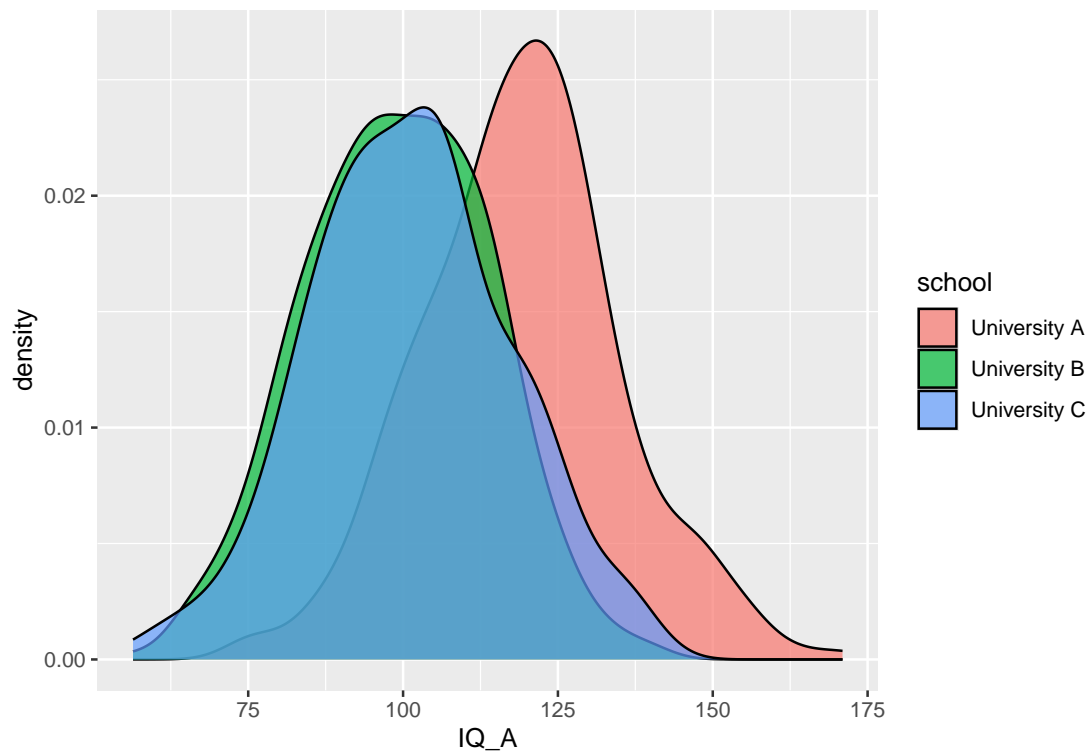


You can represent subsets of a variable by assigning the category variable to the argument `group`, `fill`, or `color`.

```
ggplot(simulation, aes(IQ_A, color=school)) +  
  geom_density(alpha=0.7)
```



```
ggplot(simulation, aes(IQ_A, fill=school)) +  
  geom_density(alpha=0.7)
```

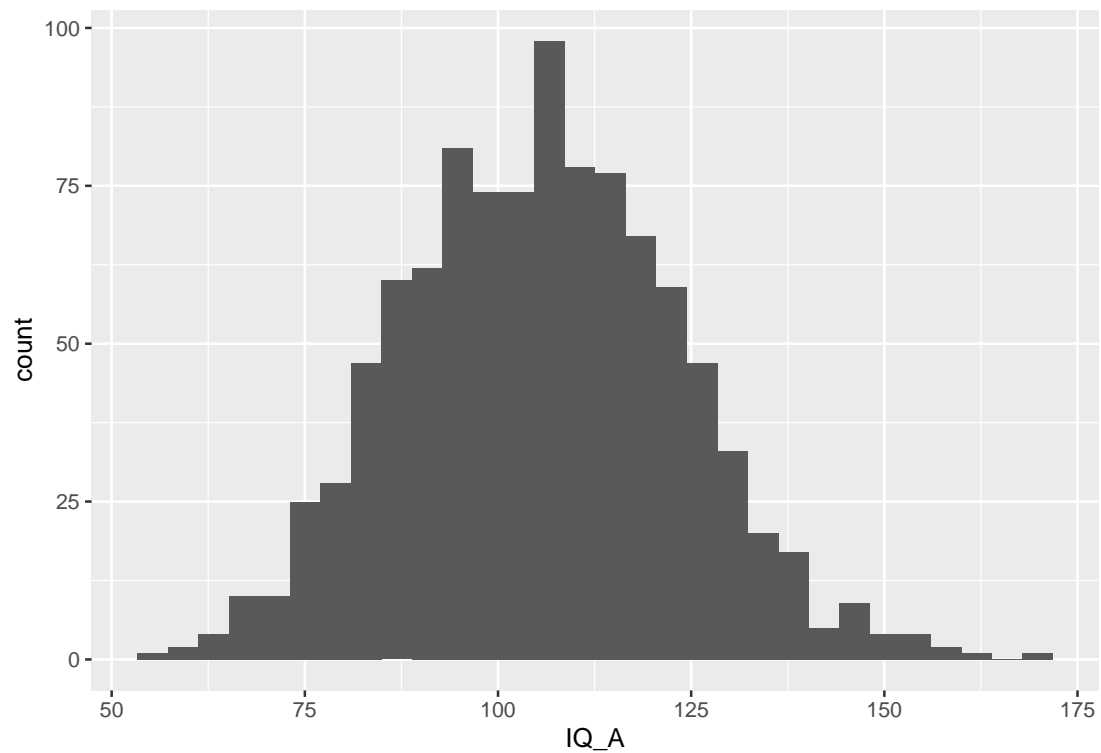



17.3.3. Histogram

Histograms are useful to get an overview on the shape of the distribution of your continuous data.

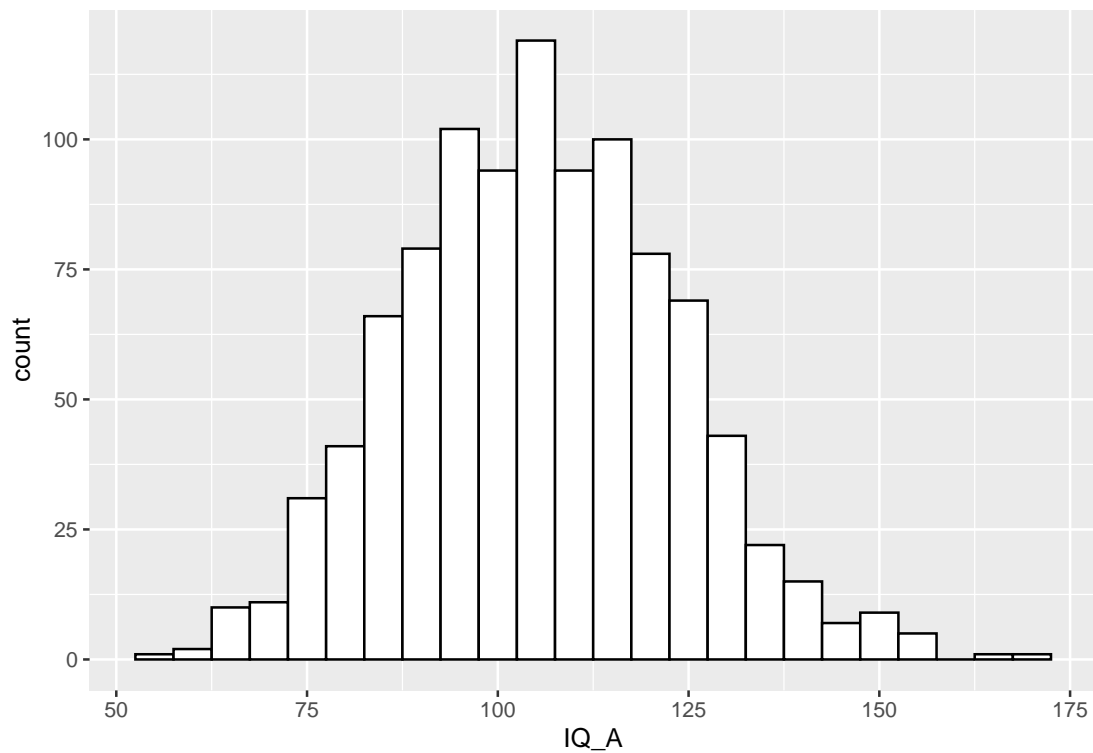
```
ggplot(simulation, aes(IQ_A)) +  
  geom_histogram()
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



If you don't like the colours, don't panic:

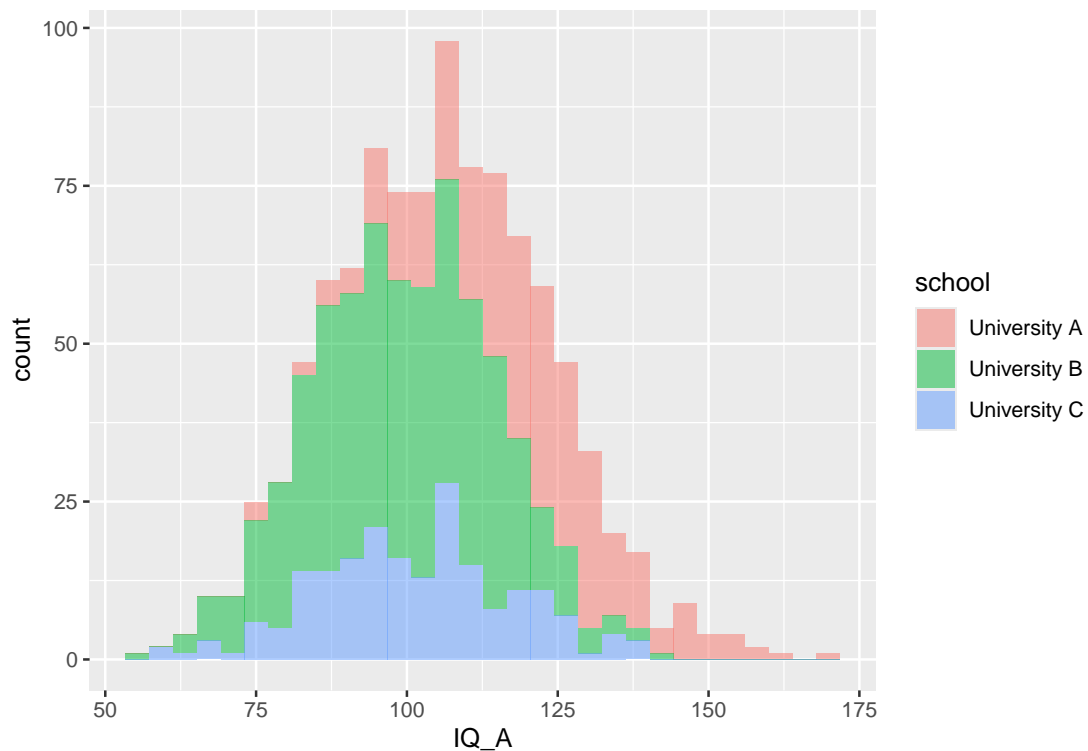
```
ggplot(simulation, aes(IQ_A)) +  
  geom_histogram(binwidth = 5, fill = "white", color = "black")
```



Again, you can add the factor variable the mapping (`aes()`) of the plot and make different histograms for each factor level.

```
ggplot(simulation, aes(IQ_A, fill = school)) +  
  geom_histogram(alpha = 0.5)
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```

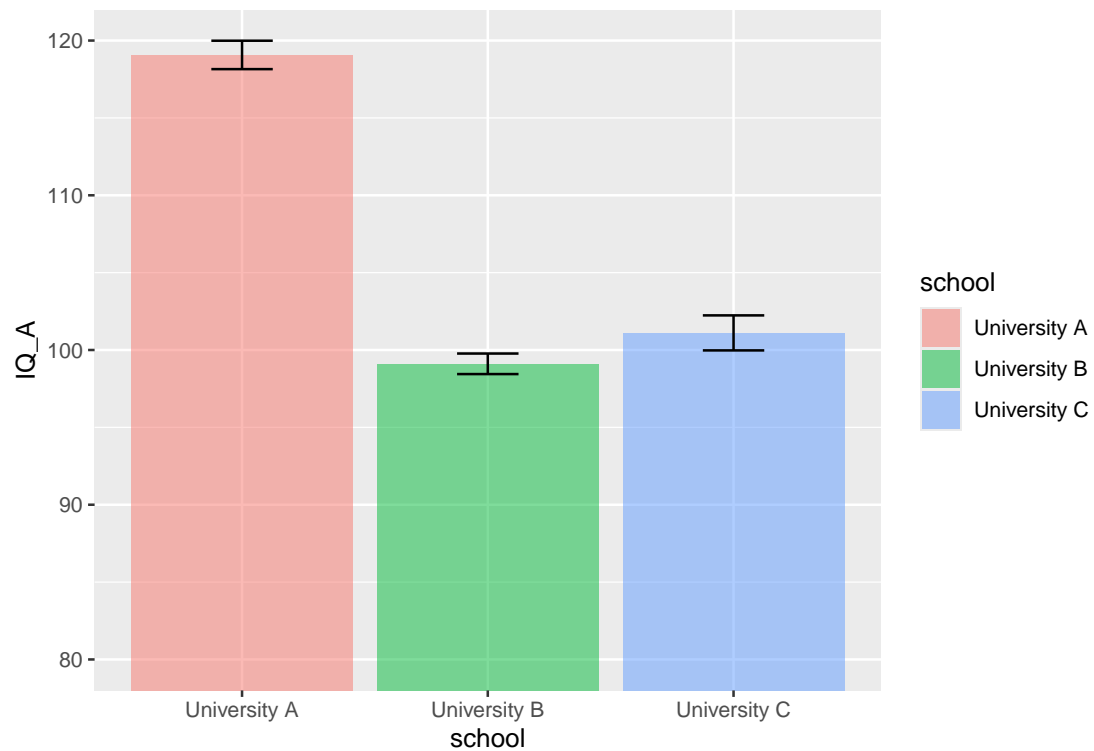


17.3.4. Column plots

Column plots are the most common way to depict grouped continuous data. If your data are already aggregated with `summarise()` (see chapter 12), you can use `geom_bar` (and `geom_col` and `geom_errorbar`) to make this plots.

If not, you can use the function `stat_summary` to calculate the mean and standard error and send those numbers to the appropriate geom for plotting.

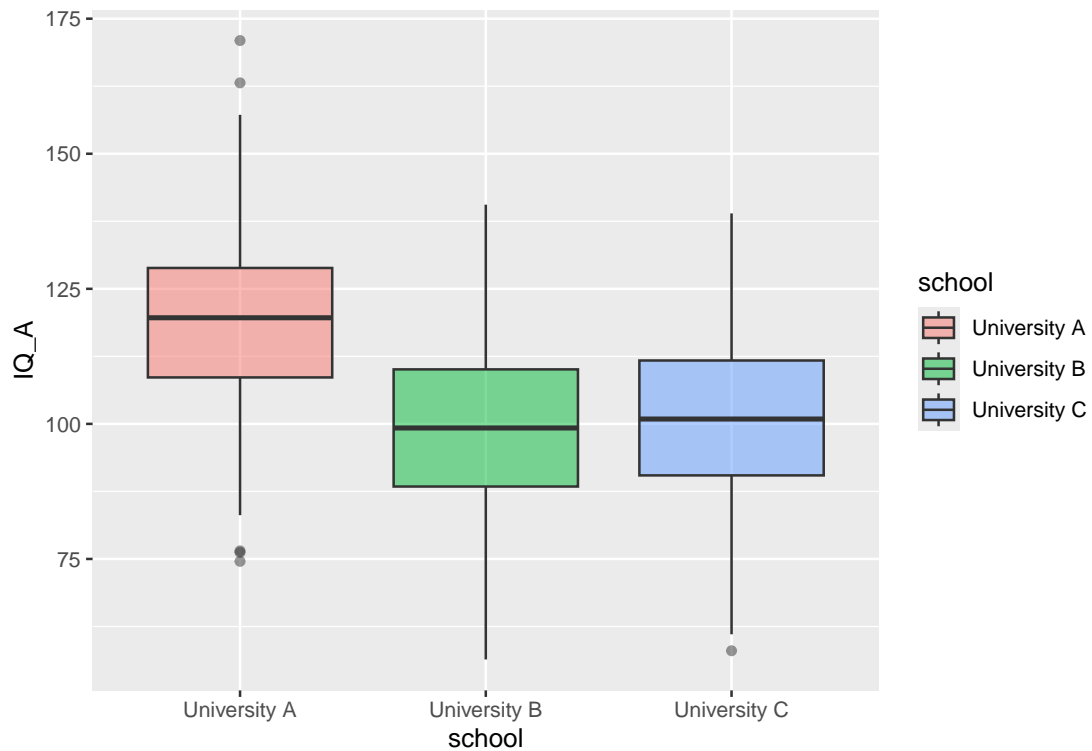
```
ggplot(simulation, aes(x=school, y=IQ_A, fill=school)) +
  stat_summary(fun = mean, geom = "col", alpha = 0.5) +
  stat_summary(fun.data = mean_se, geom = "errorbar", width = 0.25) +
  coord_cartesian(ylim = c(80, 120))
```



17.3.5. Box plots

Boxplots are, in my view, a much better way to depict grouped continuous data, because they provide some information about the distribution and about outliers. Unfortunately, they are less common in psychology and social sciences. With `ggplot2`, it's very easy to do:

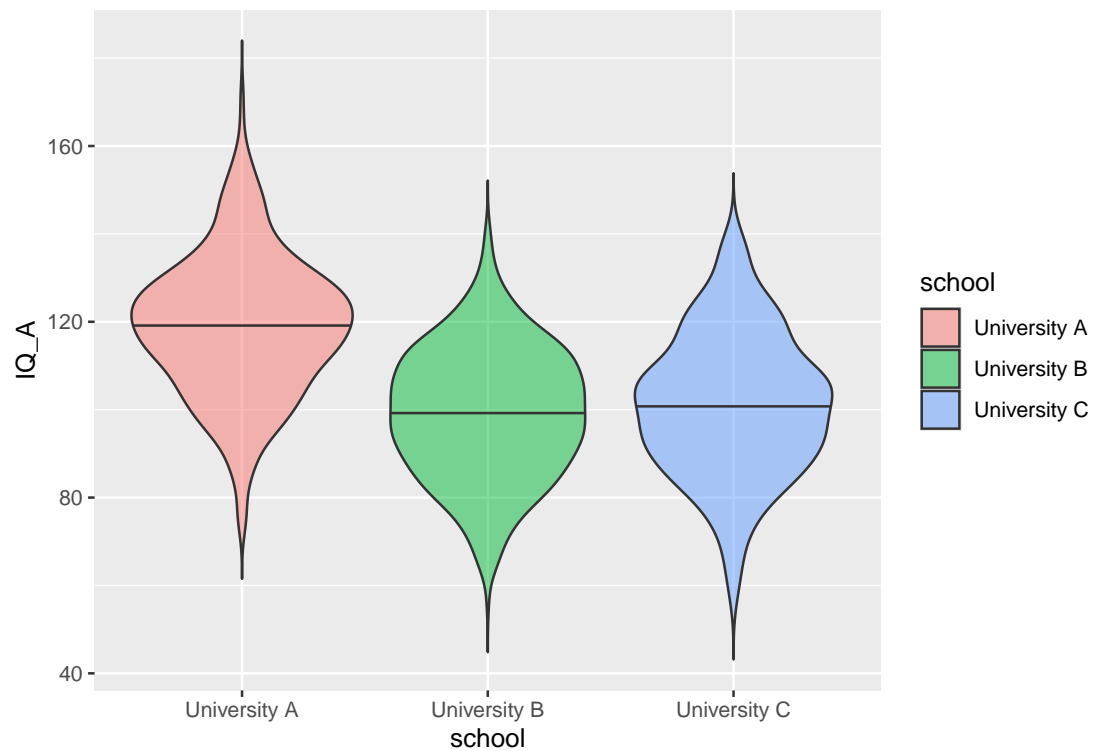
```
ggplot(simulation, aes(x=school, y=IQ_A, fill=school)) +  
  geom_boxplot(alpha = 0.5)
```



17.3.6. Violin pots

Violin pots are like sideways, mirrored density plots. They give even more direct information about distribution and are, in my view, useful when you have non-normal distributions.

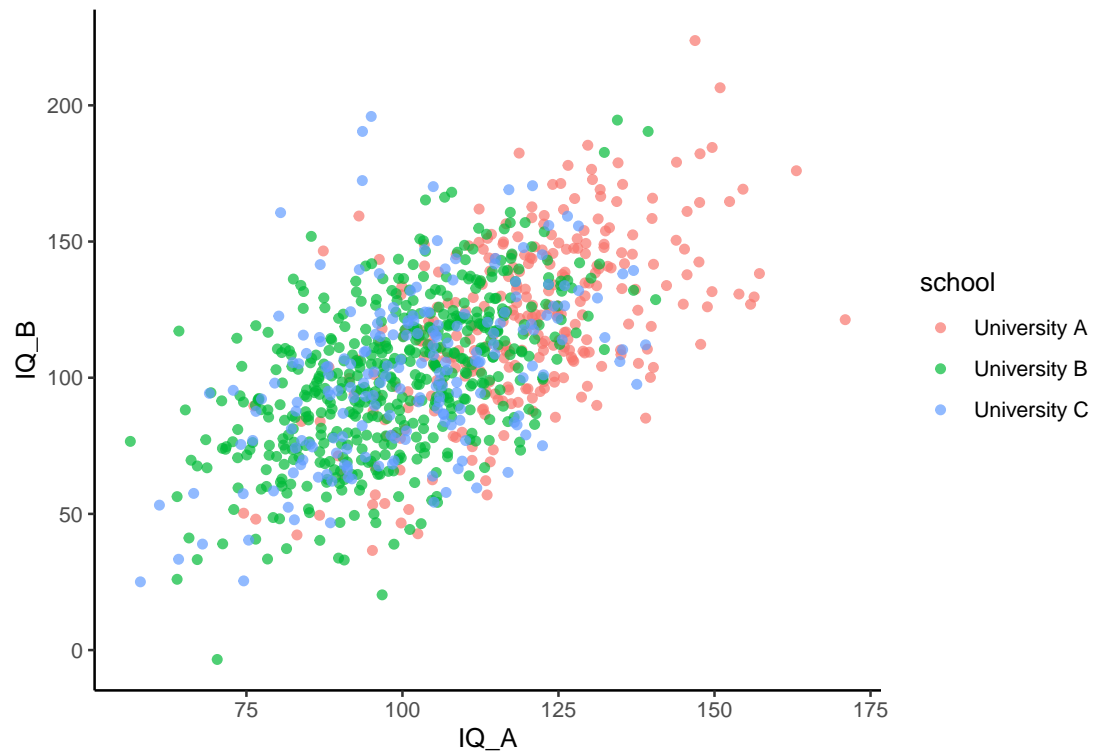
```
ggplot(simulation, aes(x=school, y=IQ_A, fill=school)) +  
  geom_violin(draw_quantiles = .5, trim = FALSE, alpha = 0.5)
```



17.3.7. Scatter plot

Let's plot IQ_A on the x and IQ_B on the y-axes and take also the `school` into account:

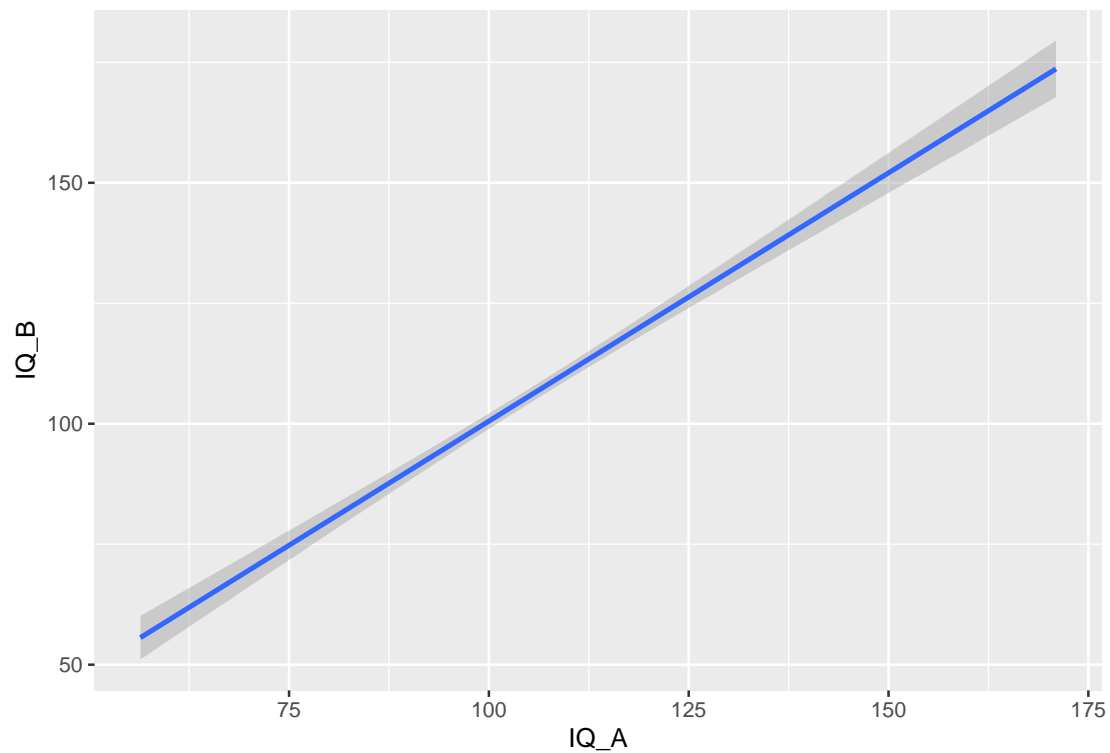
```
ggplot(simulation, aes(IQ_A, IQ_B, color=school)) +  
  geom_point(alpha = 0.7) +  
  theme_classic()
```



17.3.8. Line graph

To illustrate the correlation between the two variables, we can plot line graph that depicts the regression line.

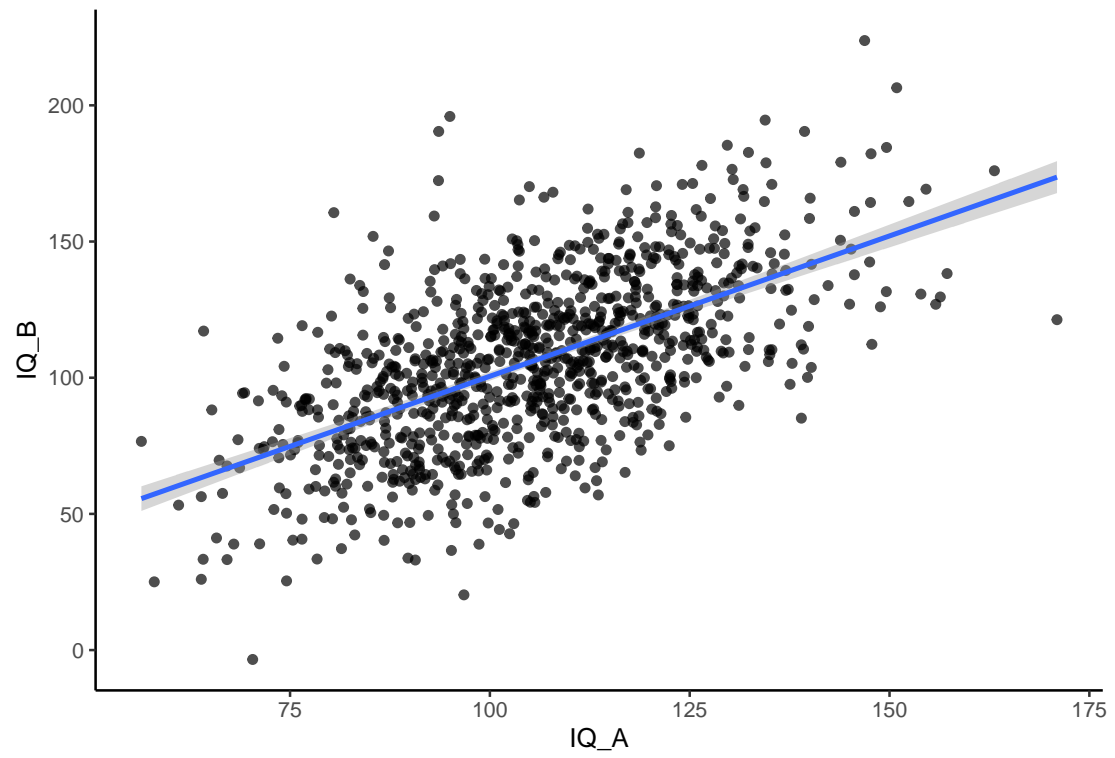
```
ggplot(simulation, aes(IQ_A, IQ_B)) +  
  geom_smooth(formula = y ~ x, method="lm")
```

17.3.9. Combining two plot types

Finally, it's also possible to combine two plot types, such as a line and scatter plot:

```
ggplot(simulation, aes(IQ_A, IQ_B)) +  
  geom_point(alpha = 0.7) +  
  geom_smooth(formula = y ~ x, method="lm") +  
  theme_classic()
```



A. Suggested Readings

The following free books and online resources provide further introductions to data science skills with *R*.

- [R for Data Science](#) by Grolemund & Wickham
- [Hands-On Programming with R](#) by G. Grolemund
- [Advanced R](#) by Hadley Wickham
- [The Tidyverse Cookbook](#) by G. Grolemund
- [Data Skills for Reproducible Science](#) from the University Glasgow

Further resources

- [R codebook](#)
- [Introduction to Open Data Science](#)
- [Exploratory Data Analysis with R](#)
- [The Art of Data Science](#)
- [Data Visualization: A practical introduction](#)

A.1. Cheat sheets

- [RStudio IDE](#)
- [Base R](#)
- [Data import](#) with `readr`
- [Data transformation](#) with `dplyr` & `tidyr`
- [Data tidying](#) with `tidyr`
- [Data visualization](#) with `ggplot`
- and more..

B. Interesting Packages

Name	Description
lubridate	Easy dealing with dates and times, see tutorial
afex	Analysis of factorial experiments
effectsize	Indices of effect size and standardized parameters
pwr	Basic functions for power analysis
easypower	Sample size estimation for experimental designs

Name	Description
lavaan	Factor analysis and structural equation modeling
lme4	Linear mixed-effects models
BayesFactor	Computation of Bayes factors for common designs
metafor	Meta-analysis package
apaTables	Create American Psychological Association (APA) style tables
rtweet	Collecting Twitter data

Examples Analyses

Most packages come with well documented example analyses called “*vignettes*”. They are done with RMarkdown. Just type `vignette()` to see a list of the vignettes of all packages installed on your machine. For instance, to see an example ANOVA done with the `afex` packages, install `afex` and run `vignette("afex_anova_example", package="afex")`.

RMarkdown

[R Markdown](#) is perfect for generating reproducible data analyses and to write manuscripts and reports. Write *R* code in your markdown reports. When you run `render`, *R* Markdown will replace the code with its results and then export your report as an HTML, pdf, or MS Word document, or a HTML or pdf slideshow. This manual has been, for instance, written with *R* Markdown and an extension called [bookdown](#).

Shiny Apps

With [Shiny Apps](#) you can make interactive web apps. It’s great to explore data and share findings with non-programmers.

C. Compatibility Issues and Workarounds

C.1. Can’t install *Tidyverse* on MacOS or Linux

Several libraries are mandatory for successfully installing the *Tidyverse*. Before you call the `install.package()` command in R, you need to have some programs installed. That is, if you encountered issues, install them like this:

MacOS

Use a terminal and install via brew:

```
brew install libxml2 openssl
```

Linux

Terminal command:

```
sudo apt install -y libcurl4-openssl-dev libssl-dev libxml2-dev
```

D. Course Exam

The course ends with a little exam on campus. Please look at your timetable on Canvas for time and location. The exam consist of data analysis tasks that you have to do. The exam will be graded.

Examination procedure

- You can bring your **own laptop** and use it for the exam. In this case, please ensure that you have internet access on the Campus. Alternatively, you can **use one computer in the PC pool**. In this case, you might have a preference to use a particular computer. If so, please contact me or one of the invigilators at the beginning of the exam. Selecting a particular place might be beneficial, because using a computer that you have work with before ensures that you have your project with the package **tidyverses** installed.
- When the exam starts, you can download the exam sheet with the assignments via **Canvas**.
- Enter all your responses to the assignments in one single *R*-script. Check out the **practice exam** on Canvas to get an impression how an exam and an ideal response *R*-script could look like.
- You have **3 hours** time to do this and to **upload your script to Canvas**. Please note that there is a strict deadline of 3 hours and you can not upload your script afterwards.
- During the exam, an invigilator will come by and carry out an identity check. To do this, have your ID, driver's license, residence permit or possibly a physical student card ready on the top left corner of your table. The invigilator will wear gloves and a mask.

What is allowed to use and what will be consider as an attempt of fraud?

- You are allowed to use the internet and any materials that you have on your computer.
- However, it's **not** allowed to directly ask others for help, to exchange any code or to communicate with someone online. That is, **any usage of emails, chat clients/websites, the posting of questions in an online forum/newsgroups as well as the use of mobile phones will be consider as an attempt of fraud.**