Automated Data Processing with R

Christian Neuwirth & Maximilian Elixhauser

10 Oktober, 2023

Table of Contents

# Preface

This web-book is a text book with exercises that together form the learning materials for “Automated Data Processing with R”, an elective module of the UNIGIS distance learning program in Geoinformatics at the University of Salzburg.

The web-book is published under an open licence. I welcome everybody to explore the contents and to work through the exercises. Only the related discussion forum and the assignments can exclusively be accessed by actively enrolled UNIGIS students, who signed up for this module.

# 1 Introduction to R

## 1.1 About this module

This module will provide you with the fundamental skills in basic programming in R. We will start with some core concepts of programming that are the building blocks of programming in any language. This includes **Datatypes**, **Operators**, **Variables**, **Functions**, **Control Structures** and **Libraries**.

On this basis, we will explore more complex data types like **Data Frames** and **Tibbles** as well as methods to **Read and Write** spatial and non-spatial datasets. In many cases the available data will not be suitable for your analyses. You will learn how to **filter, query, subset, join and re-shape** data to fit your needs.

After you have learned how to manipulate data, you will get to know methods to **visualize** data by means of diagrams (e.g. box plots, scatterplots, line plots etc.) and maps.

Upon the completion of this module, you will have the fundamental skills in R programming as a basis for more advanced methods like Geospatial Data Analysis (is covered by the module “Spatial Statistics” in the MSc program) and Machine Learning.

This module is partly based on the teaching materials [granolarr](https://github.com/sdesabbata/granolarr) worked out by [Stefano de Sabbata](https://sdesabbata.github.io/) at the [University of Leicester](https://le.ac.uk/). For more information take a look at the Webbook [R for Geographic Data Science](https://sdesabbata.github.io/r-for-geographic-data-science/). The chapters on **Statistical Analysis** and **Machine Learning** are recommended as a follow up read for those who are willing to delve into more advanced applications of R.

## 1.2 R programming language

R is a language that is applied in diverse fields of data science and analysis. Typical applications include…

* data wrangling
* statistical analysis
* machine learning
* data visualisation and maps
* processing spatial data
* geographic information analysis
* and many more.

Apart from its widespread use, there are a number of other reasons to learn R…

* R is free and open source.
* R has more comprehensive functionality than most proprietary solutions.
* R is available for Windows, Mac OS and Linux
* R is a general-purpose programming language, so you can use it to automate analyses and create new custom functions that extend default features.
* Because R is open source, it has a large user community, so it is easy to get help.

R is a so-called **high level programming language** or **scripting language**. This means that R code is not compiled into a computer readable format, but interpreted by an **interpreter.** An interpreter is a computer program that directly interprets and executes instructions written in a programming language.

In order to make sure that the interpreter can understand the program code, the programmer must stick to the grammar of the programming language; i.e. the interpreter expects commands to appear in a predefined order. This grammar is often regarded as **Syntax**.

In this lesson we will focus on some key principles of the R syntax and logic.

## 1.3 Installation and Setup

Before you can run your code, you have to install R together with an **Integrated Development Environment (IDE)** on your machine:

1. Download R from [R Archive Network (CRAN.)](http://cran.r-project.org)
2. Follow the instructions and install the most up to date version on your machine (chose ‘base’ as well as 32-bit or 64-bit dependent on the bit-version of your operating system).

The IDE is where you write, test and execute your R programs. I strongly recommend using **RStudio Desktop**, which is [freely available for download.](https://www.rstudio.com/)

The following video gives a brief overview of key functions of RStudio.



Figure 1.1: Video (6:09 min): RStudio for the Total Beginner.

In case you are facing any technical issues, please turn to the discussion forum!

## 1.4 Interpreting values

Now that you have installed RStudio and R on your machine, it is time to run some code in RStudio. When you open RStudio, you will find the **Console Window** (see Fig. 1.2). When values and operations are inputted in the *Console*, the interpreter returns the results of its interpretation of the expression.



Figure 1.2: Console Window in RStudio

Type in a numeric value (e.g. 3) and press Enter. The interpreter returns a value in brackets and the input value. The value in brackets indicates that the input is composed of one single entity.

What if you type in a text value (e.g. test) and press Enter?

**See solution!**

The interpreter returns an error, because this datatype is unknown. Text is commonly reffered to as **String** or **String of Characters**. When apostrophes (i.e. "Test") are added, the interpreter knows that this is a String.

If you start your input with a hash symbol (#) the interpreter will consider that line as a comment. For instance, if you type in **# Test Test Test**, you will see that nothing is returned as an output. Comments are extremely important as they allow you to add explanations in plain language. Comments are fundamental to allow other people to understand your code and it will save you time interpreting your own code.

## 1.5 Simple data types

In the previous section you have already see **numeric** and **character (string)** data types. **Logical** is a third simple data type provided with R.

R provides three core data types

* numeric
  + both integer and real numbers
* character
  + i.e., text, also called *strings*
* logical
  + TRUE or FALSE

The type ‘logical’ encodes the values TRUE or FALSE. Together these three simple data types are the building blocks R uses to encode information.

If you type a simple numeric operation in the console (e.g. 2 + 4), the interpreter will return a result. This indicates that operations (e.g. mathematical calculations) can be carried out on these types.

Logical operations return values of type ‘logical’. What value is returned in the console when you type and execute the expression 2 < 3?

**See solution!**

The interpreter returns ‘TRUE’, because it is ‘true’ that 2 is less than 3.

## 1.6 Numeric operators

R provides a series of basic numeric operators.

| Operator | Meaning | Example | Output |
| --- | --- | --- | --- |
| + | Plus | 5 + 2 | 7 |
| - | Minus | 5 - 2 | 3 |
| \* | Product | 5 \* 2 | 10 |
| / | Division | 5 / 2 | 2.5 |
| %/% | Integer division | 5 %/% 2 | 2 |
| %% | Module | 5 %% 2 | 1 |
| ^ | Power | 5^2 | 25 |

Whereas mathematical operators are self-explanatory, the operators ‘Module’ and ‘Integer division’ may be new to some of you. Integer division returns an integer quotient:

5%/%2

## [1] 2

Note: In this web book, two hash symbols (##) highlight the values returned by the R Console. The code above returns a value of 2. The number in squared brackets [1] indicates the line number of the return.

Execute 5 %% 2 to test the ‘Module’ operator.

**See solution!**

The ‘Module’ returns the remainder of the division, which is ‘1’ in the example above.

## 1.7 Logical operators

R also provides a series of basic logical operators to create logical expressions.

| Operator | Meaning | Example | Output |
| --- | --- | --- | --- |
| == | Equal | 5 == 2 | FALSE |
| != | Not equal | 5 != 2 | TRUE |
| > (>=) | Greater (or equal) | 5 > 2 | TRUE |
| < (<=) | Less (or equal) | 5 <= 2 | FALSE |
| ! | Not | !TRUE | FALSE |
| & | And | TRUE & FALSE | FALSE |
| | | Or | TRUE | FALSE | TRUE |

Logical expressions are typically used to execute code dependent on the occurrence of conditions.

What logical values are returned by the following expressions:

* *(3 != 5) | (3 == 4)*
* *(2 >= 3) | (3 < 7)*
* *(2 == 9) & (2 < 4)*

Type and execute (Enter button) in the RStudio console to validate your assumptions.

## 1.8 References

Apart from [Stefano de Sabbata’s](https://sdesabbata.github.io/) teaching materials, a number of other sources are use in this module. Most of them are freely available online:

* The Grammar Of Graphics – All You Need to Know About ggplot2 and Pokemons by Pascal Schmidt. see [Online Tutorial](https://thatdatatho.com/grammar-of-graphics-ggplot2/)
* ggplot2 - Overview. see [Online Documentation](https://ggplot2.tidyverse.org/)
* Getting started with httr2 - httr2 quickstart guide. [Online Tutorial](https://httr2.r-lib.org/articles/httr2.html)
* *Programming Skills for Data Science: Start Writing Code to Wrangle, Analyze, and Visualize Data with R* by Michael Freeman and Joel Ross, Addison-Wesley, 2019. See book [webpage](https://www.pearson.com/us/higher-education/program/Freeman-Programming-Skills-for-Data-Science-Start-Writing-Code-to-Wrangle-Analyze-and-Visualize-Data-with-R/PGM2047488.html) and [repository](https://programming-for-data-science.github.io/).
* *R for Data Science* by Garrett Grolemund and Hadley Wickham, O’Reilly Media, 2016. See [online book](https://r4ds.had.co.nz/).
* *Discovering Statistics Using R* by Andy Field, Jeremy Miles and Zoë Field, SAGE Publications Ltd, 2012. See book [webpage](https://www.discoveringstatistics.com/books/discovering-statistics-using-r/).
* *Machine Learning with R: Expert techniques for predictive modeling* by Brett Lantz, Packt Publishing, 2019. See book [webpage](https://subscription.packtpub.com/book/big_data_and_business_intelligence/9781788295864).
* *Introduction to spatial data in R* by Nils Riach and Rafael Hologa. See [online tutorial](https://thinking-spatial.org/courses/angewandte_geodatenverarbeitung/kurs03/).
* *The Art of R Programming: A Tour of Statistical Software Design* by Norman Matloff, No Starch Press, 2011. See book [webpage](https://nostarch.com/artofr.htm)
* *An Introduction to R for Spatial Analysis and Mapping* by Chris Brunsdon and Lex Comber, Sage, 2015. See book [webpage](https://us.sagepub.com/en-us/nam/an-introduction-to-r-for-spatial-analysis-and-mapping/book258267)
* *Geocomputation with R* by Robin Lovelace, Jakub Nowosad, Jannes Muenchow, CRC Press, 2019. See [online book](https://bookdown.org/robinlovelace/geocompr/).
* *The RStudio Cheatsheets -*  [Collection on R Studio Website](https://www.rstudio.com/resources/cheatsheets/).
* *The terra package.* see [Online Documentation](https://rspatial.org/pkg/index.html)
* *Sf - Simple Features for R.* see [Online Documentation](https://r-spatial.github.io/sf/)

# 2 Core concepts

In this lesson, we will focus on three fundamental concepts in programming:

1. Variables
2. Functions
3. Libraries

## 2.1 Variables

Variables are used to **store data**. Variables can be defined using an *identifier*, i.e. a variable name (e.g., a\_variable), on the left of an *assignment operator* <-, followed by the object to be linked to the identifier such as a *value* (e.g. 1):

a\_variable <- 1

The value of the variable can be invoked by simply specifying the **identifier**.

a\_variable

## [1] 1

In order to save your code, you can create an **R Script** in RStudio (File/New File/R Script). Select the code in the R Script Window and push ‘Run’ to execute the code.

Note: The code is executed line by line in a sequential order!

Variables allow you to save the result of any computations performed in the code and retrieve it later in the code for further analyses. For instance, you can declare a variable such as,

a\_variable <- 1

manipulate the value of the variable as

a\_variable <- a\_variable + 10

and later in the code assign the value to a different variable

another\_variable <- a\_variable

At this point, the question may arise, why bother using variables instead of simply typing the numbers? The answer is that variables make your code reusable and safe you lots of time.

Let us consider the following example:

Meteorologists monitor water temperature gradients in the Pacific Ocean to better understand El Niño weather patterns and to forecast extreme weather conditions associated with it. In a given year the water temperature at location A is 22°C and 26°C at location B. We could simply calculate the difference by executing the arithmetic operation ‘26 - 22’ in the console window of RStudio. However, temperatures are measured in real-time, i.e. we have to calculate temperature gradients repeatedly.

To speed up the process we could write code that does the calculation (temperature at location A - temperature at location B). This piece of code takes two variables (temperature at location A and B) as an input. As a result, we only need to update these two variables; the algorithm (simple subtraction in our example) is reusable.

Of course, gains in efficiency are minor given that the calculus is simple. In a more practical application, however, the algorithm is likely being composed of many lines of code that evaluate El Niño occurrence risk based on sensor records.

1. Create a new R script in RStudio (File/New File/R Script).
2. Declare two variables (temp\_A and temp\_B) and assign arbitrary temperature values to it.
3. Declare a third variable (diff) and assign the difference between the other variables as a value.
4. Run your script (select your code and click Run).

**See solution!**

temp\_A <- 24

temp\_B <- 28

diff <- temp\_A - temp\_B

When executing the code in Rstudio, you should see that something has changed in the panel on the top right, which is the **Environment Panel**. The Environment Panel shows that we now have three slots of memory with identifiers named diff, temp\_A and temp\_B that have values of -4, 24 and 28. If we invoke the name of the identifier in the code (e.g. type *diff* and run), the value that is stored in that slot gets returned.

To clear your workspace memory, push the broom icon in the menu of the Environment Panel.

## 2.2 Algorithms and functions

*An* **algorithm** *or effective procedure is a mechanical rule, or automatic method, or program for performing some mathematical operation* (Cutland, 1980).

A **program** is a specific set of instructions that implement an abstract algorithm.

The definition of an algorithm (and thus a program) can consist of one or more **functions**. Functions are a set of instructions that perform a task, i.e. functions help structuring code into functional units. These functional units are reusable in the code. Some of them receive values as inputs, some return output values.

Programming languages usually provide pre-defined functions that implement common algorithms (e.g., to find the square root of a number or to calculate a linear regression).

For instance, the pre-defined function ‘sqrt()’ calculates the square root of an input value. ‘sqrt()’ (as every function in R) is invoked by specifying the *function name* and the *arguments* (input values) between simple brackets:

sqrt(2)

## [1] 1.414214

Each input value corresponds to a *parameter* that was specified in the definition of the function. Sometimes the *parameter* name must be specified. This will get clearer when you write your own functions later in the module.

‘round()’ is another function that is predefined in R:

round(1.414214, digits = 2)

## [1] 1.41

Note that the name of the second parameter (‘digits’) needs to be specified. The parameter ‘digits’ indicates the number of digits we want to keep after the dot.

The return value of a function can be stored in a variable:

sqrt\_of\_two <- sqrt(2)  
sqrt\_of\_two

## [1] 1.414214

The output value is stored in the memory slot with the identifier ‘sqrt\_of\_two’. We can use the identifier ‘sqrt\_of\_two’ as an argument in other functions as

sqrt\_of\_two <- sqrt(2)  
round(sqrt\_of\_two, digits = 3)

## [1] 1.414

The first line calculates the square root of ‘2’ and stores it in a variable with identifier ‘sqrt\_of\_two’. The second line rounds the value stored in ‘sqrt\_of\_two’ to three digits after the dot.

Can you store the output of the ‘round()’ function in a second variable?

**See solution!**

sqrt\_of\_two <- sqrt(2)

rounded\_sqrt\_of\_two <- round(sqrt\_of\_two, digits = 3)

Functions can also be used as arguments of functions. For instance, we can use the function ‘sqrt()’ as the first argument in function ‘round()’:

round(sqrt(2), digits = 3)

## [1] 1.414

In this case the intermediate step of storing the square root of ‘2’ in a variable was skipped.

Using functions as arguments in other functions is often discouraged as it makes code hard to understand.

Moreover, in order to improve readability of R code, it is recommended to consider naming conventions when creating identifiers for variables and functions:

* R is a **case sensitive** language
  + UPPER and lower case are not the same
  + a\_variable is different from a\_VARIABLE
* names can include
  + alphanumeric symbols
  + . and \_
* names must start with
  + a letter

## 2.3 Libraries

Once a number of related, reusable functions are created, they can be collected and stored in **libraries** (a.k.a. *packages*).

To date there are more than 10,000 R libraries available, which can be downloaded and installed by means of the function ‘install.packages()’. After installing the library the function ‘library()’ is used to make it available to a script.

Libraries can be of any size and complexity, e.g.:

* base: base R functions, including the sqrt function above
* sf: A package that provides [simple feature access](https://en.wikipedia.org/wiki/Simple_Features).

The use of libraries in R can be illustrate by means of the stringr library, which provides a consistent and well-defined set of functions for manipulating strings. Assuming that the library has already been installed on your computer, you can load the library as

library(stringr)

Otherwise, you can download and install the library by calling the function

install.packages('stringr') #Note: the function takes an argument of type string ('')

Alternatively, you can download and install libraries (a.k.a. packages) using the ‘Install Packages Menu’ in RStudio (Tools/Install Packages…). In the upper dropdown list you can choose between ‘install from CRAN’ and ‘install from Package Archive file’. The large majority of libraries are available with [CRAN - Comprehensive R Archive Network](https://cran.r-project.org/), which is a collection of libraries and other R resources.

Once the library is installed and loaded, a new series of functions is available within your environment. For instance, the function ‘str\_length’ returns the number of letters included in a string:

str\_length("UNIGIS")

## [1] 6

‘str\_detect()’ does return ‘TRUE’, if the first argument (a string) contains the second argument (letter as type string). Otherwise, the function returns ‘FALSE’:

str\_detect("UNIGIS", "I")

## [1] TRUE

The function ‘str\_replace\_all’ replaces all the instances of the first argument that are identical with the second argument by a third argument:

str\_replace\_all("UNIGIS", "I", 'X')

## [1] "UNXGXS"

You may list all the functions available with library ‘stringr’ using the built in function [‘ls()’](https://www.rdocumentation.org/packages/base/versions/3.6.2/topics/ls):

ls("package:stringr")

# 3 Data Structures

In this lesson I will introduce a series of more complex data types that are built on top of the already discussed simple data types ‘numeric’, ‘character’ (string) and ‘logical’ (see [Lesson 1](#intro) ‘Simple data types’).

In this lesson, you will get to know the following data structures in R:

1. Vectors
2. Matrices and Arrays
3. Lists
4. Data Frames

## 3.1 Vectors

A **Vector** is an ordered list of values. Vectors can be of any simple type:

- numeric  
- character  
- logic

However all items in a vector have to be of the same type. A vector can be of any length.

Defining a **vector variable** is similar to the declaration of simple type variables, except that the vector is created by a return function named ‘c()’ that combines values into a vector:

# Declare a vector variable of strings  
a\_vector <- c("Birmingham", "Derby", "Leicester",  
 "Lincoln", "Nottingham", "Wolverhampton")  
a\_vector

## [1] "Birmingham" "Derby" "Leicester" "Lincoln"   
## [5] "Nottingham" "Wolverhampton"

Note that the second line of the returned elements starts with [5], as the second line starts with the fifth element of the vector.

There are also other functions to create vectors such as ‘seq()’:

#create vector of real numbers of interval 0.5 in a range between 1 and 7  
a\_vector <- seq(1, 7, by = 0.5)  
a\_vector

## [1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0 4.5 5.0 5.5 6.0 6.5 7.0

or ‘rep()’:

#create vector with 4 identical character string values  
a\_vector <- rep("Ciao", 4)  
a\_vector

## [1] "Ciao" "Ciao" "Ciao" "Ciao"

Alternatively, numeric vectors can be created by using the following syntax:

#create a vector of integer numbers between 1 and 10  
a\_vector <- (1:10)  
a\_vector

## [1] 1 2 3 4 5 6 7 8 9 10

### 3.1.1 Vector element selection

Each element of a vector can be retrieved by specifying the related **index** between square brackets, after the identifier of the vector. The **first element** of the vector **has index 1**. The following, code retrieves a value of ‘5’, which is the third element of the vector with identifier ‘a\_vector’:

a\_vector <- (3:8)  
a\_vector[3]

## [1] 5

A vector of indexes can be used to retrieve more than one element:

a\_vector <- (3:8)  
a\_vector[c(2, 4)]

## [1] 4 6

The values 4 and 6 are returned. These values have the indices 2 and 4 in vector ‘a\_vector’. Note that the vector containing the indices 2 and 4 is created on the fly (without assigning the return value to a variable).

Now try by yourself. Create a vector that looks like

east\_midlands\_cities <- c("Derby", "Leicester", "Lincoln", "Nottingham")

, select the last three cities out of the four cities in ‘east\_midlands\_cities’ and assign the returned values to a new vector named ‘selected\_cities’.

**See solution!**

east\_midlands\_cities <- c("Derby", "Leicester", "Lincoln", "Nottingham")

my\_indexes <- 2:4

selected\_cities <- c(east\_midlands\_cities[my\_indexes])

### 3.1.2 Functions on vectors

In R, functions can be used on a vector variable in the same way they are used on simple variables. In this case, the function will be applied to each element of the vector. The output will be a new vector containing the same number of elements as the input vector.

For instance, adding a number of ten to a vector of numbers between 1 and 5 will result in a vector of numbers between 11 and 15:

a\_numeric\_vector <- 1:5  
a\_numeric\_vector <- a\_numeric\_vector + 10  
a\_numeric\_vector

## [1] 11 12 13 14 15

Accordingly, an sqrt() function applied to the same vector will return a vector containing the square root of every element as a result:

a\_numeric\_vector <- 1:5  
a\_numeric\_vector <- sqrt(a\_numeric\_vector)  
a\_numeric\_vector

## [1] 1.000000 1.414214 1.732051 2.000000 2.236068

We can also produce a vector of type ‘logical’ by using a condition:

a\_numeric\_vector <- 1:5  
a\_numeric\_vector <- a\_numeric\_vector >= 3  
a\_numeric\_vector

## [1] FALSE FALSE TRUE TRUE TRUE

While the condition in the example above returns an evaluation of the conditional statement for every element, the function ‘any’ and ‘all’ are **overall expressions**. The function ‘any()’ returns TRUE, if any of the elements satisfy the condition:

a\_numeric\_vector <- 1:5  
any(a\_numeric\_vector >= 3)

## [1] TRUE

The function ‘all’ returns TRUE, if all of the elements satisfy the condition:

a\_numeric\_vector <- 1:5  
all(a\_numeric\_vector >= 3)

## [1] FALSE

A **factor** is a data type similar to a vector. However, the values contained in a factor can only be selected from a set of **levels**. Factors will not be covered in the module. For more information on this data type turn to the online tutorial [Programming with R](https://swcarpentry.github.io/r-novice-inflammation/12-supp-factors.html)

## 3.2 Multi-dimensional data types

### 3.2.1 Matrices

So far, you have learned about one dimensional data types. **Matrices** are collections of numbers arranged in a two-dimensional rectangular layout.

To create a matrix, two arguments should be provided to the function matrix. The first argument is a vector of values. The second specifies the number of rows and columns:

a\_matrix <- matrix(c(3, 5, 7, 4, 3, 1), c(3, 2))  
a\_matrix

## [,1] [,2]  
## [1,] 3 4  
## [2,] 5 3  
## [3,] 7 1

R offers a large number of operators and functions for matrix algebra. For instance, standard mathematical operators are applicable:

x <- matrix(c(3, 5, 7, 4, 3, 1), c(3, 2))  
x

## [,1] [,2]  
## [1,] 3 4  
## [2,] 5 3  
## [3,] 7 1

y <- matrix(c(1, 2, 3, 4, 5, 6), c(3, 2))  
y

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

z <- x\*y  
z

## [,1] [,2]  
## [1,] 3 16  
## [2,] 10 15  
## [3,] 21 6

In the example above, variables x and y are initiated as matrices. The product of x and y is returned in variable z.

A more comprehensive list of matrix algebra operations is provided by [Quick-R](https://www.statmethods.net/advstats/matrix.html).

### 3.2.2 Arrays

Variables of the type **array** are higher-dimensional matrices. Just like matrices, to create an array two arguments are required. The first argument is a vector containing the values. The second argument is a vector specifying the depth of each dimension. The following code returns a 3-dimensional array:

a3dim\_array <- array(1:24, dim=c(4, 3, 2))  
a3dim\_array

## , , 1  
##   
## [,1] [,2] [,3]  
## [1,] 1 5 9  
## [2,] 2 6 10  
## [3,] 3 7 11  
## [4,] 4 8 12  
##   
## , , 2  
##   
## [,1] [,2] [,3]  
## [1,] 13 17 21  
## [2,] 14 18 22  
## [3,] 15 19 23  
## [4,] 16 20 24

Note that an array could also have only one dimension. Such an array would look like a vector. Nevertheless, it is stored with additional attributes like dim, has different options and behaves differently!

### 3.2.3 Selection

Subsetting of matrices and arrays works in a very similar way as seen for vectors. However, as these are multi-dimensional objects, one value (or index) needs to be specified for each one of the dimensions.

In the example, below we are subsetting the second row and the first and second column of a\_matrix:

a\_matrix <- matrix(c(3, 5, 7, 4, 3, 1), c(3, 2))  
a\_matrix

## [,1] [,2]  
## [1,] 3 4  
## [2,] 5 3  
## [3,] 7 1

a\_matrix[2, c(1, 2)]

## [1] 5 3

Accordingly, to subset an array with three dimensions, three indices are required:

an\_array <- array(1:12, dim=c(3, 2, 2))  
an\_array

## , , 1  
##   
## [,1] [,2]  
## [1,] 1 4  
## [2,] 2 5  
## [3,] 3 6  
##   
## , , 2  
##   
## [,1] [,2]  
## [1,] 7 10  
## [2,] 8 11  
## [3,] 9 12

an\_array[2,c(1,2),2]

## [1] 8 11

As an exercise, create an arbitrary 3-dimensional array, retrieve 2 elements from it and write those elements to a new vector variable. Then retrieve 4 elements from the 3-dimensional array and write it to a new matrix variable.

**See solution!**

a3dim\_array <- array(1:24, dim=c(4, 3, 2))

a3dim\_array

a\_vector <- a3dim\_array[3, c(1, 2), 2]

a\_vector

a\_matrix <- a3dim\_array[c(3, 4), c(1, 2), 2]

a\_matrix

### 3.2.4 List

Variables of the type **list** can contain elements of different types (including vectors and matrices), whereas elements of vectors are all of the same type.

In the following example, I created a list containing the simple types ‘character’ and ‘numeric integer’:

employee <- list("Christian", 2017)  
employee

## [[1]]  
## [1] "Christian"  
##   
## [[2]]  
## [1] 2017

employee[[1]] # Note the double square brackets for selection

## [1] "Christian"

In contrast to vectors, matrices or arrays, the selection of list elements requires the use of **double square brackets**.

A specific type of list is the so called named list. In **named lists**, each element has a name, and elements can be selected using their name after the symbol $:

employee <- list(employee\_name = "Christian", start\_year = 2017)  
employee

## $employee\_name  
## [1] "Christian"  
##   
## $start\_year  
## [1] 2017

employee$employee\_name

## [1] "Christian"

### 3.2.5 Data Frame

**Data frames** are commonly used in R to encode tables of data. A data frame is equivalent to a named list where all elements are vectors of the same length. The code below creates a data frame named ‘employees’ that is composed of three vectors:

employees <- data.frame(  
 EmployeeName = c("Maria", "Pete", "Sarah"),  
 Age = c(47, 34, 32),  
 Role = c("Professor", "Researcher", "Researcher"))

You can retrieve the tabular structure of the data frame ‘employees’ by invoking the identifier in the code:

employees

## EmployeeName Age Role  
## 1 Maria 47 Professor  
## 2 Pete 34 Researcher  
## 3 Sarah 32 Researcher

Eventually data frames are tables. Each named element is a column of the table.

Given the precondition that data frames are composed of named lists where elements are **vectors**, is it possible to mix simple types in a vector/in a data frame column?

**See solution!**

Elements of a vector (data frame column) must be of the same type (character, logical or numeric). In the example above, the column ‘EmployeeName’ contains only character strings, the column ‘Age’ contains only numeric etc.

As the columns in a data frame have the same length, one element is present for each row of the table. Meaning the first element of vector ‘EmployeeName’ in data frame ‘employees’ is the Name of the first employee. The first element in vector ‘Age’ in data frame ‘employees’ is the age of the first employee etc.

You can rename the columns of a data frame by means of a function called ‘names()’:

**names(name of data frame)[column index as number] = “new column name”**

The selection of values from a data frame is similar to what we have seen for vectors and lists. However, you have to consider the two-dimensional shape of data frames. As such, you will generally need to specify two indices in order to retrieve values from a table.

The following example retrieves the first element of the first column in data frame ‘employees’:

employees[1, 1]

## [1] "Maria"

We can also retrieve entire rows…

employees[1, ]

## EmployeeName Age Role  
## 1 Maria 47 Professor

…and columns:

employees[ ,1]

## [1] "Maria" "Pete" "Sarah"

Alternatively, columns can be selected by means of dollar signs and columns names:

employees$Age

## [1] 47 34 32

This returns the vector ‘Age’. Accordingly, we can use square brackets to retrieve elements of the vector:

employees$Age[1]

## [1] 47

To further modify the data frame, we can change elements (e.g. change the age of ‘Pete’ from 34 to 33)…

employees$Age[2] <- 33

…or insert new columns as vectors (new column name Place):

employees$Place <- c("Salzburg", "Salzburg", "Salzburg")  
employees

## EmployeeName Age Role Place  
## 1 Maria 47 Professor Salzburg  
## 2 Pete 33 Researcher Salzburg  
## 3 Sarah 32 Researcher Salzburg

Operations can be performed on columns in the same way as on vectors. As an exercise, create a new variable, which stores the current year…

*current\_year <- as.integer(format(Sys.Date(), "%Y"))*

…use the column ‘Age’ in data frame ‘employees’ to calculate the year of birth for every employee…

current\_year - employees$Age

…and insert the year of birth as a new column.

**See solution!**

#Instantiate data frame employees

employees <- data.frame( EmployeeName = c("Maria", "Pete", "Sarah"), Age = c(47, 34, 32), Role = c("Professor", "Researcher", "Researcher"))

#Use Sys.Date to retrieve the current year

current\_year <- as.integer(format(Sys.Date(), "%Y"))

#Calculate employee year of birth

employees$Year\_of\_birth <- current\_year - employees$Age

employees

# 4 Spatial Data Structures

In the realm of geoinformatics, spatial data stands as a cornerstone, offering a lens through which we can view, analyze, and interpret the world around us. At this point in your Master’s, you are already familiar with the basic spatial entities of: points, lines, and polygons. These fundamental structures, while simple in concept, form the bedrock of complex spatial analyses and visualizations.

R, with its rich ecosystem of packages, offers a unique perspective on spatial data. Packages like **sf**, **terra**, **stars**, and **spatstat** have been game-changers, allowing us to handle spatial vector, raster and multidemensional data with unprecedented ease and flexibility.

In this lesson, you will get to know the following spatial data structures in R:

1. Vector data structures based on the simple feature specification implemented in package [sf](https://r-spatial.github.io/sf/)
2. Raster data structures as provided by the package [terra](https://rspatial.org/pkg/)

You will also learn how to retrieve, assign and modify coordinate systems, projections and transformations of spatial data structures.

## 4.1 Vector Data Structures

Spatial data structures are the foundation upon which geospatial information is built. They provide a systematic framework for organizing and representing geographical entities, ensuring that they can be efficiently processed, analyzed, and visualized. In this section, we’ll delve into vector data structures, which are essential for representing discrete spatial entities.

The most common geometry types are points, lines and polygons and their respective “multi” versions.

**Points** are fundamental in geospatial analysis. They are the simplest spatial entities, representing a singular location in space. They have no dimensions, meaning they don’t possess length, width, or area. Using the sf package, you can create and manipulate point data with ease:

# Create a point  
point <- sf::st\_point(c(5, 5))  
  
# Convert to sf object  
point\_sf <- sf::st\_sf(geometry = sf::st\_sfc(point))

The operator :: is used to indicate that the functions st\_point, st\_sf and st\_sfc are situated within the library sf. This helps avoiding ambiguities in the case functions from different loaded libraries have identical names.

The function st\_point creates a simple feature object from a numeric vector. The object is of the same nature as the numeric vector c(5,5). To convert to an sf-object, the function st\_sf is used.

Sf-object are similar in structure as data frames. However, in contrast to data frames, sf-objects have an additional geometry column. The sf-object point\_sf (created by the code above) contains a single point geometry and no fields:

point\_sf

## Simple feature collection with 1 feature and 0 fields  
## Geometry type: POINT  
## Dimension: XY  
## Bounding box: xmin: 5 ymin: 5 xmax: 5 ymax: 5  
## CRS: NA  
## geometry  
## 1 POINT (5 5)

In principle sf-objects can be treated like data frames. Accordingly, data frame syntax is used to assign fields (table columns) to geometries:

point\_sf$name <- c("my location")

To stay in GIS terms, records (rows) in a sf-object table may be called features. Features can be composed of multiple geometries:

# Create three points at locations 3/4, 5/3 and 7/1 as multipoint geometry  
point\_multi <- sf::st\_multipoint(matrix(c(3, 5, 7, 4, 3, 1), c(3, 2)), dim = "XY")  
  
# Convert to sf object  
point\_sf\_multi <- sf::st\_sf(geometry = sf::st\_sfc(point\_multi))  
  
plot(point\_sf\_multi)



Plotting the metadata of sf-object point\_sf\_multi in the console reveals that Geometry type is Multipoint and that no coordinate reference system has been defined (CRS: NA):

point\_sf\_multi

## Simple feature collection with 1 feature and 0 fields  
## Geometry type: MULTIPOINT  
## Dimension: XY  
## Bounding box: xmin: 3 ymin: 1 xmax: 7 ymax: 4  
## CRS: NA  
## geometry  
## 1 MULTIPOINT ((3 4), (5 3), (...

Alternatively, the function st\_crs may be used extract coordinate system information from an sf-object:

sf::st\_crs(point\_sf\_multi)

## Coordinate Reference System: NA

In case when a CRS is missing or the wrong CRS is set, the function st\_crs is used:

# Assign WGS84 as CRS  
# 4326 is the EPSG code of WGS84  
sf::st\_crs(point\_sf\_multi) <- 4326

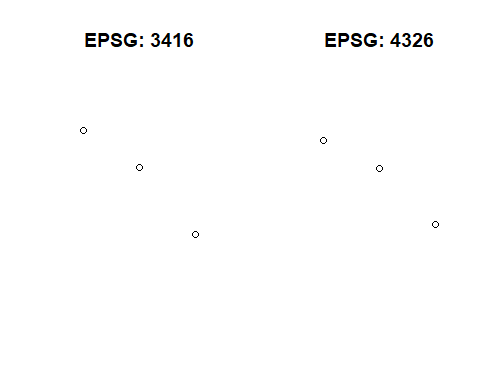
EPSG-Codes of other reference systems can be found [here.](https://spatialreference.org/ref/epsg/?search=4326&srtext=Search)

Now we transform the sf-object to [EPSG 3416](https://spatialreference.org/ref/epsg/3416/) (Austrian Lambert Projection) with function st\_transform:

# Assign NAD27 as CRS  
point\_sf\_multi\_transform <- sf::st\_transform(point\_sf\_multi, 3416)

Eventually we can compare the two sf-objects with different coordinate systems:

par(mfrow=c(1,2)) # both plots together  
plot(sf::st\_geometry(point\_sf\_multi\_transform), main = "EPSG: 3416")  
plot(sf::st\_geometry(point\_sf\_multi), main = "EPSG: 4326")



The same syntax and functions can be used to deal with **line or polygon** data. The following drop-downs contain two simple examples.

**Create Line Feature!**

Lines are sequences of points. They’re instrumental in representing pathways, routes, or any linear feature. Here’s how you can create a line using sf:

# Create a line from a matrix of coordinates  
line <- st\_linestring(matrix(1:6, 3, 2))  
  
# Convert to spatial feature  
line\_sf <- st\_sf(geometry = st\_sfc(line))  
  
# Plot the line  
plot(line\_sf)



**Create Polygon Feature!**

Polygons are closed shapes, perfect for representing areas with defined boundaries. Here’s a demonstration using sf and its st\_polygon function:

# Create a matrix of coordinates  
coords <- matrix(c(2,2, 4,4, 4,2, 2,2), ncol = 2, byrow = TRUE)  
  
# Create a list of matrices (in this case, just one matrix)  
list\_of\_coords <- list(coords)  
  
# Create the polygon  
polygon <- st\_polygon(list\_of\_coords)  
  
# Convert to spatial feature  
polygon\_sf <- st\_sf(geometry = st\_sfc(polygon))  
  
# Plot the polygon  
plot(polygon\_sf)



The code above successfully creates and plots a polygon. However, imagine if the last coordinate in the coordinate matrix (2,2) was mistakenly omitted.

Consider following Questions:

1. What error would you expect to encounter if the last coordinate was omitted?
2. Why is the last coordinate crucial for the creation of the polygon?

**Solution!**

1. If the last coordinate was omitted, you would encounter the following error:

Error in MtrxSet(x, dim, type = "POLYGON", needClosed = TRUE) : polygons not (all) closed

A breakdown of what the error message is conveying:

* MtrxSet(x, dim, type = "POLYGON", needClosed = TRUE): This is the internal function being called to set or validate the matrix representation of the polygon.
* type = "POLYGON": This indicates that the data structure being worked on is in fact a Polygon.
* needClosed = TRUE: This is a condition set within the function to ensure that polygons are closed. It checks if the starting and ending coordinates of the polygon are the same.
* polygons not (all) closed: This is the main error message, indicating that one or more polygons in your data are not closed, i.e., their starting and ending coordinates don’t match.

In practical terms, if you’re creating or manipulating polygons, you need to ensure that each polygon’s last coordinate is the same as its first coordinate. If not, many spatial operations, analyses, or visualizations might produce incorrect or unexpected results!

1. The last coordinate is crucial because it ensures that the polygon is closed, meaning its starting and ending coordinates are the same.

## 4.2 Raster Data Structures

Whereas man-made infrastructures (streets, buildings, sewage systems etc.) can clearly be delineated and modeled by discrete features, our natural-physical environment (temperature, soil moisture etc.) tends to be continuous by nature and best represented by raster data structures.

Both packages, sf and terra can handle raster and vector data. Due to its comprehensive toolset and integration with the tidyverse ecosystem (tidyverse will be covered in lesson Data Manipulation), sf is predominantly used for discrete vector data structures. Until quite recently, the package [raster](https://cran.r-project.org/web/packages/raster/index.html) has been the most popular resource to work with continuous data in R. This package is being replaced by terra (see [here](https://psfaculty.plantsciences.ucdavis.edu/plant/AdditionalTopics_Transition.pdf) for more information).

Accordingly, in this chapter we will focus on the more modern terra package that offers several advantages over its predecessor:

* Efficiency: Terra is optimized for speed and uses less memory, making it more efficient for large datasets.
* Flexibility: It supports raster, vector, and time-series data, providing a one-stop solution for various spatial data types.
* Ease of Use: With a simplified and consistent syntax, terra is easier to pick up for newcomers.
* Comprehensive Functions: From raster algebra to resampling and reclassification, terra offers a wide array of functionalities.
* Integration: It’s designed to work seamlessly with other R packages, making it easier to integrate into larger workflows.

### 4.2.1 Working with SpatRaster objects

The terra SpatRaster Object can be created using function rast:

# Load the terra package  
library(terra)  
  
x <- terra::rast()  
  
x

## class : SpatRaster   
## dimensions : 180, 360, 1 (nrow, ncol, nlyr)  
## resolution : 1, 1 (x, y)  
## extent : -180, 180, -90, 90 (xmin, xmax, ymin, ymax)  
## coord. ref. : lon/lat WGS 84

Per default the SpatRaster Object is initialized with a global extent and a spatial resolution of 1 degree. The coordinate reference system is WGS84.

Alternatively, additional arguments may be provided in the function to customize the SpatRaster Object:

x <- terra::rast(ncol=100, nrow=100, xmin=797422, xmax=807387, ymin=5298037, ymax=5306341, crs = "+proj=utm +zone=32 +ellps=WGS84 +datum=WGS84 +units=m +no\_defs ")  
  
x

## class : SpatRaster   
## dimensions : 100, 100, 1 (nrow, ncol, nlyr)  
## resolution : 99.65, 83.04 (x, y)  
## extent : 797422, 807387, 5298037, 5306341 (xmin, xmax, ymin, ymax)  
## coord. ref. : +proj=utm +zone=32 +datum=WGS84 +units=m +no\_defs

In the example above, arguments such as the number of grid rows and columns (nrow and ncol) as well as the grid extent (xmin, xmax, ymin and ymax) were defined. The raster cell resolution is a result of these inputs (x=99.65, y=83.04).

According to the [documentation of the rast function](https://www.rdocumentation.org/packages/terra/versions/1.7-46/topics/rast), the coordinate reference system can be specified in PROJ.4, WKT or authority:code notation. In the given example WGS84 UTM 33N is encoded in PROJ.4. To find the desired encoding, it is recommended to first search for a CRS on the [Spatial Reference Website](https://spatialreference.org/). The PROJ.4 is one out of many formats (e.g. EPSG code, WKT, GML etc.) that are provided in the search results.

The SpatRaster Object x has an extent that covers the City of Salzburg, which is completely within UTM Zone 33N. Search for the PROJ.4 encoding of WGS84 UTM 33N on the [Spatial Reference Site.](https://spatialreference.org/)

**Solution!**

+proj=utm +zone=33 +ellps=WGS84 +datum=WGS84 +units=m +no\_defs

In order to change the coordinate reference system from WGS84 UTM 32N to WGS84 UTM 33N and to change the spatial resolution to 100m, the terra-functions project and res can be used:

y <- terra::project(x, "+proj=utm +zone=33 +ellps=WGS84 +datum=WGS84 +units=m +no\_defs")  
terra::res(y) <- 100  
y

## class : SpatRaster   
## dimensions : 91, 105, 1 (nrow, ncol, nlyr)  
## resolution : 100, 100 (x, y)  
## extent : 347863.1, 358363.1, 5291599, 5300699 (xmin, xmax, ymin, ymax)  
## coord. ref. : +proj=utm +zone=33 +datum=WGS84 +units=m +no\_defs

Note that the number of rows and column was changed to enable a raster resolution of 100m. Also the bounding box coordinates of the SpatRaster Object have changed, which indicates a successful projection of the raster grid from UTM 32 to UTM 33.

Given that the SpatRaster Object has an undefined coordinate reference system, it can be defined by means of [terra-function crs](https://rdrr.io/cran/terra/man/crs.html).

So far we have created two SpatRaster Objects (identifiers x and y). They only consist of a skeleton, meaning that they have location, extend, a spatial gird resolution and a certain number of grid rows and columns. However, there are not yet cell-values associated with it:

#check whether objects have values  
terra::hasValues(x)

## [1] FALSE

terra::hasValues(y)

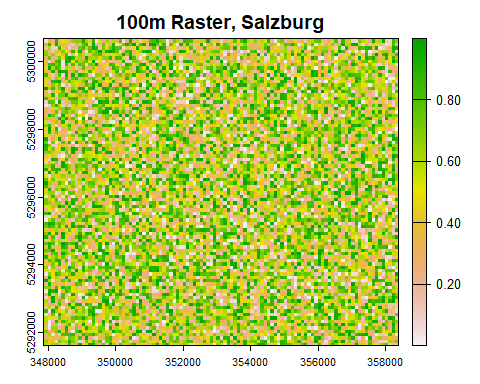
## [1] FALSE

The function hasValues() returns a logical value FALSE, because no values were assigned to objects x and y. The code below shows how to assign values to empty SpatRaster Object y:

#assign random value between 0 and 1 to cells of SpatRaster Object y  
terra::values(y) <- stats::runif(terra::ncell(y),0,1)  
  
#get values with index 1 to index 5  
terra::values(y)[1:5]

## [1] 0.04426076 0.35778215 0.24688722 0.88376651 0.40688890

#plot grid, plot() is a generic function to plot R objects  
plot(y, main='100m Raster, Salzburg')



The function runif of package stats takes three arguments, n, min and max, to generate n random numbers in a range between min and max. In the example above argument n is derived from terra function ncell, which returns the number of cells of SpatRaster Object y as an integer value. As a result we get a numeric vector of random values whose length corresponds to the number of grid cells of SpatRaster Object y. Accordingly, we can assign the numeric vector values to the grid.

When assigning or accessing values, it is crucial to know the origin and orientation of raster values. Create a SpatRaster Object with 4 cells and assign a numeric vector that consists of 4 values c(1,2,3,4) to it. Try to access individual grid values by means of the following syntax:

raster-obj[<row index>, <column index>]

Where is the origin of the grid? In what order are the values c(1,2,3,4) stored in the vector grid.

**Solution!**

r <- terra::rast(ncol=2, nrow=2)  
terra::values(r) <- c(1,2,3,4)  
plot(r)  
r[1,1]

r[1,1] returns value 1. Accordingly, the upper left corner is the origin of the SpatRaster Object. Values are stored left to right and top-down, i.e. r[2,2] returns value 4.

Note that the layer number lyr.1 is returned in the console, when accessing individual values of a SpatRaster Object. This implies that SpatRaster Objects can handle space-time and multivariable data. A useful example for integrating space and time in SpatRaster Objects is provided by Dominic Royé in his blog post [Use of multidimensional spatial data.](https://dominicroye.github.io/en/2022/use-of-multidimensional-spatial-data/) For more sophisticated data cube applications, the use of the [stars package](https://r-spatial.github.io/stars/) is recommended.

In this lesson you learned to handle spatial vector and raster structures in R. To get to know these structures, we built vector and raster objects from scratch. In many instances, however, objects may be created in R by loading vector and raster file formats (e.g. .shp or .tif). This topic will be covered in lesson Read and Write Data.

# 5 Control structures

In this lesson, you will get to know control structures as a significant structural element of code. Control structures allow our code to adapt its behavior depending on the value of the current variables in the environment.

We distinguish between two types of control structures:

1. **Conditional statements**, which allow executing instructions only if a certain condition is satisfied.
2. **Loops**, which allow to repeat one or more instructions multiple times. Loops are commonly used to apply the same operation on a series of values that are stored in sequences such as vectors or lists.

## 5.1 If

The most fundamental conditional statement in R is the structure ‘if’, which is used to execute one or more instructions only if a certain condition is TRUE.

In order to include an if-structure in your code, you need to use the following syntax:

a\_value <- -7  
if (a\_value < 0) {  
 cat("Negative")  
}

## Negative

The **statement** ‘cat(“Negative”)’ gets executed and the text “Negative” is printed out, because the **condition** (a\_value < 0) is TRUE.

The function [cat()](https://www.rdocumentation.org/packages/base/versions/3.6.2/topics/cat) concatenates and prints string inputs (“Negative” in the example above). Alternatively, you can use the function ‘print()’ to write variable values to the console window.

These functions are highly useful to inspect whether variables take on expected values!

Note that every conditional statement (e.g. ‘a\_value < 0’) returns a logical value that is either TRUE or FALSE.

Remove the negative sign from the conditional statement in the code above.

**See solution!**

The condition yields FALSE. The statement is not executed.

## 5.2 Else

In many cases, we want the interpreter to do something if the condition is satisfied or do something *else*, if the condition is not satisfied. In this case, we can use ‘if’ together with ‘else’:

a\_value <- -7  
if (a\_value < 0){  
 cat("Negative")  
} else {  
 cat("Positive")  
}

## Negative

In the example above, the **condition** ‘a\_value < 0’ is TRUE, **statement 1** ‘cat(“Negative”)’ gets executed and **statement 2** ‘cat(“Positive”)’ is ignored. If you change ‘a\_value’ to a positive value, the interpreter will ignore statement 1 and execute statement 2.

Note that statement 1 and statement 2 are in curly brackets! The indentation of statements is good programming practice, however does not affect the functioning of the code.

However, inserting a line break before ‘else’ returns an error. The reason for this behavior is explained in [a forum thread](https://stackoverflow.com/questions/37929184/if-else-does-the-line-break-between-and-else-really-matters).

## 5.3 Code blocks

Conditional structures have a wide range of applications. Almost everything what a computer does requires an input. Each time you click a button the computer responds accordingly. The code that dictates the response typically has an if-else control structure or something very similar that tells the computer what to do depending on the input it got. Obviously in most cases the response won’t be defined by a single instruction, but a code block that is composed of multiple instructions. **Code blocks** allow encapsulating **several** statements in a single group. The condition in the following example yields TRUE and the code block is executed:

first\_value <- 8  
second\_value <- 5  
if (first\_value > second\_value) {  
 cat("First is greater than second\n")   
 difference <- first\_value - second\_value  
 cat("Their difference is", difference)  
}

## First is greater than second  
## Their difference is 3

The line ‘cat(“First is greater than second”)’ prints a text (string) and inserts a line break. The next line calculates the difference between first and second value. The third line in the code block concatenates two inputs (“Their difference is” and variable ‘difference’) and prints them to the console window.

‘if’ and ‘else’ are so called **reserved words**, meaning they cannot be used as variable names.

## 5.4 Loops

The second family of control structures that we are going to discuss in this lesson are loops. Loops are a fundamental component of (procedural) programming. They allow repeating one or more instructions multiple times.

There are two main types of loops:

* **conditional** loops are executed as long as a defined condition holds true
  + construct while
  + construct repeat
* **deterministic** loops are executed a pre-determined number of times
  + construct for

### 5.4.1 While and repeat

The *while* construct can be defined using the while reserved word, followed by a condition between simple brackets, and a code block. The instructions in the code block are re-executed as long as the result of the evaluation of the condition is TRUE.

current\_value <- 0  
while (current\_value < 3) {  
 cat("Current value is", current\_value, "\n")  
 current\_value <- current\_value + 1  
}

## Current value is 0   
## Current value is 1   
## Current value is 2

Go through the example above and try to verbalize the consecutive steps.

**See solution!**

1. The variable ‘current\_value’ takes on a value of zero.
2. The condition of the while-loop returns TRUE.
3. The ‘cat()’ function is executed and prints a text as well as ‘current\_value’.
4. The variable ‘current\_value’ is incremented by +1.
5. The condition of the while-loop returns TRUE (current\_value = 1), the code block is executed again (see 3 and 4).
6. current\_value = 2, the code block is executed again (see 3 and 4).
7. current\_value = 3, the condition returns FALSE, the loops ends.

The same procedure can alternatively be implemented by means of the *repeat* construct:

current\_value <- 0   
repeat {   
 cat("Current value is", current\_value, "\n")   
 current\_value = current\_value + 1   
if (current\_value == 3){ #if (variable == 3)...   
 break #the loop will break!  
 }  
}

## Current value is 0   
## Current value is 1   
## Current value is 2

The break statement is executed and stops (breaks) the repeat loop (also applicable to while or for loops) once the variable current\_value is equal to three.

### 5.4.2 For

The *for* construct can be defined using the for reserved word, followed by the definition of an **iterator**. The iterator is a variable, which is temporarily assigned with the current element of a vector, as the construct iterates through all elements of the vector. This definition is followed by a code block, whose instructions are re-executed once for each element of the vector.

cities <- c("Derby", "Leicester", "Lincoln", "Nottingham")  
for (city in cities) {  
 cat("Do you live in ", city, "?\n", sep="")  
}

## Do you live in Derby?  
## Do you live in Leicester?  
## Do you live in Lincoln?  
## Do you live in Nottingham?

In the first iteration of the for-loop the text string “Derby” is assigned to the iterator ‘city’. The function ‘cat()’ uses the iterator value as an input. In the second iteration, the text string “Leicester” is assigned to the iterator ‘city’ … etc.

The code block below illustrates another example.

cities <- c("Derby", "Leicester", "Lincoln", "Nottingham")  
letter\_cnt <- c()  
for (city in cities) {  
 letter\_cnt <- c(letter\_cnt, nchar(city))  
}  
print(letter\_cnt)

## [1] 5 9 7 10

The for-loop iterates over the elements in vector ‘cities’. The base function ‘nchar()’ counts the number of letters of every city name and appends the count to a new vector ‘letter\_cnt’.

Note that with every iteration a new value is appended to the right side of the vector. The syntax for appending elements to a vector in R is…

*name vector <- c(name vector, element to append)*

There are some cases in which, for some reason, you just want to execute a certain sequence of steps a pre-defined number of times. In such cases, it is common practice to create a vector of integers on the spot. In the following example the for-loop is executed 3 times as it iterates over a vector composed of the three elements 1, 2, and 3 (vector is created on the spot by 1:3):

for (i in 1:3) {  
 cat("This is exectuion number", i, ":\n")  
 cat(" See you later!\n")  
}

## This is exectuion number 1 :  
## See you later!  
## This is exectuion number 2 :  
## See you later!  
## This is exectuion number 3 :  
## See you later!

Replace the vector 1:3 by a vector 3:5. What is different?

**See solution!**

The for-loop is still executed 3 times.

However, the iterator ‘i’ returns the values 3, 4, and 5.

## 5.5 Loops with conditional statements

Now that we have seen both types of control structures, **conditional statements** and **loops**, we can combine these structures. R, as most other programming languages, allows you to include conditional statements within a loop or a loop within a conditional statement.

A simple example is this bit of code that defines a countdown:

#Example: countdown!  
for (i in 3:0) {  
 if (i == 0) {  
 cat("Go!\n")  
 } else {  
 cat(i, "\n")  
 }  
}

## 3   
## 2   
## 1   
## Go!

The deterministic loop runs 4 time on the values 3, 2, 1, and 0. If the iterator ‘i’ takes on a value of 0 the print “Go!” otherwise print the current value of the iterator ‘i’. The result will be 3, 2, 1, Go!

**See another example!**

library(tidyverse)  
  
cities <- c("Salzburg", "Linz", "Wien", "Eisenstadt", "Innsbruck", "Graz")  
  
for (city in cities){  
 if (str\_starts(city, "S")){  
 print("City name starts with S")  
 } else{  
 print("City name starts with other letter")  
 }  
   
}

## [1] "City name starts with S"  
## [1] "City name starts with other letter"  
## [1] "City name starts with other letter"  
## [1] "City name starts with other letter"  
## [1] "City name starts with other letter"  
## [1] "City name starts with other letter"

We need to load the library ‘tidyverse’ to make use of the function ‘str\_starts()’. You may have to install ‘tidyverse’ (see Libraries in lesson [core Concepts](#core)).

‘cities’ is a vector of strings that includes the names of some Austrian federal capitals. The for-loop iterates over these vector elements. The function ‘str\_starts’ takes the value of the iterator ‘city’ as well as a string “S” as inputs. If the city starts with letter S, the function returns TRUE and “City name starts with S” is printed to the console window, otherwise the function returns FALSE and “City name starts with other letter” is printed.

# 6 Functions

In the past few lessons, we have been using functions without looking at them in much detail (e.g. functions like ‘str\_starts()’ or ‘cat()’). In this lesson, we are going to look inside those functions to see how functions work and how to create custom functions.

Moreover, you will get to know the difference between variables with global and variables with local scope.

## 6.1 Defining functions

The syntax for defining a function does not look too different from the syntax that we have been using to define a variable or to create a conditional statement. We start by defining an identifier (e.g. add\_one) on the left of an assignment operator (<-). This is followed by the so-called **corpus of the function.**

The corpus starts with the reserved word ‘function’ followed by the **parameter(s)** (e.g. input\_value in the example below) between simple brackets and the instruction(s) to be executed in a code block. The value of the last statement is returned as output:

add\_one <- function (input\_value) {  
 output\_value <- input\_value + 1  
 output\_value  
}

After being defined, a function can be invoked by specifying the **identifier** and necessary parameters. The function above takes a single numeric value as an input and returns that value incremented by +1. So if we invoke the function with an input value ‘3’, the function returns 4:

add\_one(3)

## [1] 4

## 6.2 More parameters

A function can be defined as having two or more **parameters**. Parameter names are separated by **commas** in the definition of the function.

A function always takes as input as many values as the number of parameters specified in the definition, otherwise an error is generated

The function ‘area\_rectangle’ includes two parameters (height and width), calculates an area value by multiplying the inputs and returns the area as a single numeric value:

area\_rectangle <- function (height, width) {  
 area <- height \* width  
 area  
}  
area\_rectangle(3, 2)

## [1] 6

In a few cases, it makes sense to define default parameters in a function. Create a new R script, copy the function definition above and change the parameter definition ‘function (height, width)’ to ‘function (height, width = 3)’. Now invoke the function by only specifying one input value. The function should return a value ‘YOUR INPUT \* 3’. If you call the same function by specifying two values as inputs, the default value ‘width=3’ is overwritten.

## 6.3 More return values

In order to let a function return multiple values, you can append return values to a list and return the list. The following functions ‘rectangle\_metrics’ calculates area and perimeter of a rectangle based on two inputs (rectangle height and width) and writes the two outputs to a new list ‘return\_vals’:

rectangle\_metrics <- function (height, width) {  
 area <- height \* width  
 perimeter <- 2\*height + 2\*width  
   
 return\_vals <- list(area, perimeter)  
   
 return\_vals  
   
}

We can retrieve the two return values by specifying their list indexes [[1]] and [[2]]:

cat("This is the first return value - area: ", rectangle\_metrics(3, 2)[[1]])

## This is the first return value - area: 6

cat("This is the second return value - perimeter: ", rectangle\_metrics(3, 2)[[2]])

## This is the second return value - perimeter: 10

If you execute one of the functions above in a new R Script, you will see that the function appears in the Environment Window of RStudio in the same way as when we define a variable. When the function is invoked by using its identifier, the R interpreter will retrieve the respective function from the memory and execute it.

## 6.4 Functions and control structures

In the last lesson, you have learned that loops can contain conditional statements and that conditional statement can contain loops. In the same way, the corpus of a function can contain both loops and conditional statements.

The following code shows an example of a function that is using a loop to calculate the factorial of a number. A factorial of a number is simply the product of all the numbers less than or equal to that number (e.g. factorial of 3 = 1 \* 2 \* 3 = 6).

factorial <- function (input\_value) {  
 result <- 1  
 for (i in 1:input\_value) {  
 cat("current:", result, " | i:", i, "\n")  
 result <- result \* i  
 }  
 result  
}  
factorial(3)

## current: 1 | i: 1   
## current: 1 | i: 2   
## current: 2 | i: 3

## [1] 6

The function takes a single numeric value as an input, defines a variable named ‘result’ that is equal to ‘1’ and then creates a loop over all the numbers from 1 (variable ‘result’) to the ‘input value’. In the loop, the current value of result is multiplied by the value of the iterator ‘i’.

Although it is technically feasible, you would normally not define a function within conditional statements or within a loop.

## 6.5 Scope

Parameters of a function effectively are internal variables of the function. They work as a bridge between the overall environment in which you are working and the internal environment, which is only known to the function. They receive the value(s) provided as arguments. When you call the function and make those values available within the function itself. The distinction between the overall environment (global) that we have seen so far and a sort of internal environment (local) of the function brings us to the concept of scope.

The **scope of a variable** is the part of code in which the variable is ‘visible’.

You have learned that a variable is saved in the memory. You can ‘see’ the variable, which means that you can use the identifier to invoke the variable in the code.

When you define a function, the corpus of the function is the scope of the variables that are defined in this function. That means you can make use of these variable within the function, but you cannot invoke these variables outside of the function (variables are not ‘seen’ outside the function).

In R, the scope of variables is defined as follows:

* a variable defined in a script (global) can be referred to from within a definition of a function in the same script
* a variable defined within a definition of a function (local) will **not** be referable from outside the definition
* scope does **not** apply to if or loop constructs, meaning that variables defined within a loop or control structure are referable from everywhere in the code.

Let us take an example. In the case below, x\_value is **global** to the function times\_x. new\_value and input\_value are **local** to the function times\_x. Referring to new\_value or input\_value from outside the definition of times\_x would result in an error. However, we can refer to x\_value from inside function times\_x:

x\_value <- 10  
times\_x <- function (input\_value) {  
 new\_value <- input\_value \* x\_value  
 new\_value  
}  
times\_x(2)

## [1] 20

Referring to external global variables in a function is possible, but can be dangerous. At the time of execution, one cannot be sure what the value of the global variable is. For instance, other processes might have changed its value, which affects the behavior of the function. In order to fix this problem, define the variable ‘x\_value’ as a default attribute of function ‘times\_x’.

**See solution!**

times\_x <- function (input\_value, x\_value = 10) {

new\_value <- input\_value \* x\_value

new\_value

}

The lessons so far have covered some fundamental concepts of R programming. The [Base R Cheatsheet](https://github.com/rstudio/cheatsheets/blob/main/base-r.pdf) contains a concise summary of most important operations at a glance.

# 7 Data manipulation

In most instances the structure of the available data will not meet the specific requirements to perform the analyses you are interested in. Data analysts typically spend the majority of their time cleaning, filtering, restructuring data as well as harmonizing and joining data from different sources.

This lesson introduces to the most common data wrangling operations by means of the dplyr library (part of the Tidyverse libraries), which offers a grammar for data manipulation.

You will also get to know [tibbles](https://tibble.tidyverse.org/), which is another data structure in R. Tibbles are basically a lightweight version of data frames (see [Tibbles in R for Data Science](https://r4ds.had.co.nz/tibbles.html)).

## 7.1 Preparation

If not yet installed on your machine, install the libraries tidyverse as well as nycflights13 (see Libraries in lesson [core Concepts](#core)).

The code below, loads a sample dataset (a [tibble](https://tibble.tidyverse.org/)) from the library nycflights13 into the variable flights\_from\_nyc. We will use this sample data in this lesson.

library(nycflights13)  
flights\_from\_nyc <- nycflights13::flights

The operator :: is used to indicate that the function flights (that returns our sample dataset) is situated within the library nycflights13. This helps avoiding ambiguities in the case functions from different loaded libraries have identical names.

In order to run the following data wrangling examples on your machine, add both lines above as well as the code snippets provided in the upcoming examples to a new R script file.

Once you have loaded the flights table, open the **Environment Tab** in RStudio and double-click variable flights\_from\_nyc to inspect the variable contents.

Alternatively, you may inspect flights\_from\_nyc by writing it to the console.

## 7.2 Data manipulation

The library dplyr provides a number of functions to investigate basic characteristics of inputs. For instance, the function count() can be used to count the number of rows of a data frame or tibble. The code below uses flights\_from\_nyc as input to the function.

library(tidyverse)  
library(knitr)  
  
flights\_from\_nyc %>%  
 dplyr::count() %>%  
 knitr::kable()

| n |
| --- |
| 336776 |

In the code example above, we use the so called **pipe operator** %>%, which is included in tidyverse, as well as a function named kable() of library knitr to render the output.

The pipe operator allows us to link a sequence of analysis steps. In the example above, flights\_from\_nyc is passed into function count() and the output is passed into function kable() to render the result\_df

The pipe operator is a powerful tool to simplify your code. See [this video](https://www.youtube.com/watch?v=sohARFx6aTo) to learn more about it.

The function count() can also be used to count the number of rows of a table that has the same value for a given column, usually representing a category.

In the example below, the column name origin is provided as an argument to the function count(), so rows representing flights from the same origin are counted together – EWR is the Newark Liberty International Airport, JFK is the John F. Kennedy International Airport, and LGA is LaGuardia Airport.

flights\_from\_nyc %>%  
 dplyr::count(origin) %>%  
 knitr::kable()

| origin | n |
| --- | --- |
| EWR | 120835 |
| JFK | 111279 |
| LGA | 104662 |

As you can see, the code above is formatted in a way similar to a code block, although it is not a code block. The code goes to a new line after every %>%, and space is added at the beginning of new lines. That is very common in R programming (especially when functions have many parameters) as it makes the code more readable.

### 7.2.1 Summarise

To carry out more complex aggregations, the function summarise() can be used in combination with the function group\_by() to summarise the values of the rows of a data frame or tibble. Rows having the same value for a selected column (in the example below, the same origin) are grouped together, then values are aggregated based on the defined function (using one or more columns in the calculation).

In the example below, the function sum() is applied to the column distance to calculate a new column mean\_distance\_traveled\_from (the mean distance travelled by flights starting from each airport).

flights\_from\_nyc %>%  
 dplyr::group\_by(origin) %>%  
 dplyr::summarise(  
 mean\_distance\_traveled\_from = mean(distance)  
 ) %>%  
 knitr::kable()

| origin | mean\_distance\_traveled\_from |
| --- | --- |
| EWR | 1056.7428 |
| JFK | 1266.2491 |
| LGA | 779.8357 |

### 7.2.2 Select and filter

The function select() can be used to select a subset of **columns**. For instance in the code below, the function select() is used to select the columns origin, dest, and dep\_delay. The function slice\_head is used to include only the first n rows in the output.

flights\_from\_nyc %>%  
 dplyr::select(origin, dest, dep\_delay) %>%  
 dplyr::slice\_head(n = 5) %>%  
 knitr::kable()

| origin | dest | dep\_delay |
| --- | --- | --- |
| EWR | IAH | 2 |
| LGA | IAH | 4 |
| JFK | MIA | 2 |
| JFK | BQN | -1 |
| LGA | ATL | -6 |

The function filter() can instead be used to filter **rows** based on a specified condition. In the example below, the output of the filter step only includes the rows where the value of month is 11 (i.e., the eleventh month, November).

flights\_from\_nyc %>%  
 dplyr::select(origin, dest, year, month, day, dep\_delay) %>%  
 dplyr::filter(month == 11) %>%  
 dplyr::slice\_head(n = 5) %>%  
 knitr::kable()

| origin | dest | year | month | day | dep\_delay |
| --- | --- | --- | --- | --- | --- |
| JFK | PSE | 2013 | 11 | 1 | 6 |
| JFK | SYR | 2013 | 11 | 1 | 105 |
| EWR | CLT | 2013 | 11 | 1 | -5 |
| LGA | IAH | 2013 | 11 | 1 | -6 |
| JFK | MIA | 2013 | 11 | 1 | -3 |

Notice how filter is used in combination with select. All functions in the dplyr library can be combined, in any other order that makes logical sense. However, if the select step didn’t include month, that same column couldn’t have been used in the filter step.

### 7.2.3 Mutate

The function mutate() can be used to add a new column to an output table. The mutate step in the code below adds a new column air\_time\_hours to the table obtained through the pipe, that is the flight air time in hours, dividing the flight air time in minutes by 60.

flights\_from\_nyc %>%  
 dplyr::select(flight, origin, dest, air\_time) %>%  
 dplyr::mutate(  
 air\_time\_hours = air\_time / 60  
 ) %>%  
 dplyr::slice\_head(n = 5) %>%  
 knitr::kable()

| flight | origin | dest | air\_time | air\_time\_hours |
| --- | --- | --- | --- | --- |
| 1545 | EWR | IAH | 227 | 3.783333 |
| 1714 | LGA | IAH | 227 | 3.783333 |
| 1141 | JFK | MIA | 160 | 2.666667 |
| 725 | JFK | BQN | 183 | 3.050000 |
| 461 | LGA | ATL | 116 | 1.933333 |

Run the mutate example above in a new script and replace dplyr::mutate by dplyr::transmute. What happens to your results?

**See solution!**

The transmute function adds a new column to the table and drops existing ones.

### 7.2.4 Arrange

The function arrange() sorts a tibble or data frame by ascending order of the values in the specified column. If a negative sign is specified before the column name, the descending order is used. The code below would produce a table showing all the rows when ordered by descending order of air time.

flights\_from\_nyc %>%  
 dplyr::select(origin, dest, air\_time) %>%  
 dplyr::arrange(-air\_time) %>%  
 dplyr::slice\_head(n = 5) %>%  
 knitr::kable()

| origin | dest | air\_time |
| --- | --- | --- |
| EWR | HNL | 695 |
| JFK | HNL | 691 |
| JFK | HNL | 686 |
| JFK | HNL | 686 |
| JFK | HNL | 683 |

In the examples above, we have used slice\_head to present only the first n rows in a table, based on the existing order.

### 7.2.5 Exercise: data manipulation

The Food and Agriculture Organization (FAO) is a specialized agency of the United Nations that leads international efforts to defeat hunger. On their [Website](http://www.fao.org/home/en/) they provide comprehensive datasets on global crop and livestock production. Your task in this exercise is to create a table that shows national African sorghum production in 2019.

1. Create an RScript and install or load the libraries tidyverse and knitr, if not done yet.
2. Bulk download [African Crop and Livestock Production](http://www.fao.org/faostat/en/#data/QCL) data as CSV:



Figure 7.1: FAO Data Download

1. Read data from comma-separated CSV (“Production\_Crops\_Livestock\_E\_Africa\_NOFLAG.csv”) into your Script.

fao\_data <- read.csv(directory as string, header = TRUE, sep = ",")

1. Use the pipe operator to perform the following operations:
   1. Select columns Area, Item, Element, Unit and Y2019
   2. Filter rows that contain sorghum production (Item == “Sorghum” & Element == “Production”)
   3. Sort the table based on yield in descending order (arrange)
   4. remove rows including No Data by means of function drop\_na()
   5. render the table using the function kable() of library knitr

See [my solution](Solution_Exercise_DataWrangling1.R)!

## 7.3 Join

A join operation combines two tables into one by matching rows that have the same values in the specified column. This operation is usually executed on columns containing identifiers, which are matched through different tables containing different data about the same real-world entities. For instance, the **table below** (data frame city\_telephone\_prexix) presents the telephone prefixes for two cities. That information can be combined with the data present in the **second table below** (data frame city\_info\_wide) through a join operation on the columns containing the city names. As the two tables do not contain all the same cities, if a full join operation is executed, some cells have no values assigned.

city\_telephone\_prexix <- data.frame(  
 city = c("Leicester", "Birmingham", "London"),  
 telephon\_prefix = c("0116", "0121", "0171")  
 ) %>%  
 tibble::as\_tibble()  
  
city\_telephone\_prexix %>%  
 knitr::kable()

| city | telephon\_prefix |
| --- | --- |
| Leicester | 0116 |
| Birmingham | 0121 |
| London | 0171 |

city\_info\_wide <- data.frame(  
 city = c("Leicester", "Nottingham"),  
 population = c(329839, 321500),  
 area = c(73.3, 74.6),  
 density = c(4500, 4412)  
 ) %>%  
 tibble::as\_tibble()  
  
city\_info\_wide %>%  
 knitr::kable()

| city | population | area | density |
| --- | --- | --- | --- |
| Leicester | 329839 | 73.3 | 4500 |
| Nottingham | 321500 | 74.6 | 4412 |

Note that data frames in the code above are converted to [tibbles](https://tibble.tidyverse.org/). This step is needed as the function kable() takes tibbles as an input.

The dplyr library offers different types of join operations, which correspond to the different SQL joins illustrated in the image below.



Figure 7.2: Join types

Please take your time to understand the examples below and check out the [related dplyr help pages](https://www.rdocumentation.org/packages/dplyr/versions/0.7.8/topics/join) before continuing. The first four examples execute the exact same *full join* operation using three different syntaxes: with or without using the pipe operator and specifying the by argument or not. Note that all those approaches to writing the join are valid and produce the same result. The choice about which approach to use will depend on the code you are writing. In particular, you might find it useful to use the syntax that uses the pipe operator when the join operation is itself only one stem in a series of data manipulation steps. Using the by argument is usually advisable unless you are certain that you aim to join two tables with all and exactly the column that have the same names in the two table.

Note how the result of the join operations is *not* saved to a variable. The function knitr::kable is added after each join operation through a pipe %>% to display the resulting table in a nice format.

# Option 1: without using the pipe operator  
# full join verb  
dplyr::full\_join(  
 # left table  
 city\_info\_wide,  
 # right table  
 city\_telephone\_prexix,  
 # columns to match  
 by = c("city" = "city")  
 ) %>%  
 knitr::kable()

| city | population | area | density | telephon\_prefix |
| --- | --- | --- | --- | --- |
| Leicester | 329839 | 73.3 | 4500 | 0116 |
| Nottingham | 321500 | 74.6 | 4412 | NA |
| Birmingham | NA | NA | NA | 0121 |
| London | NA | NA | NA | 0171 |

# Option 2: without using the pipe operator  
# and without using the argument "by"  
# as columns have the same name  
# in the two tables.  
# Same result as Option 1  
# full join verb  
dplyr::full\_join(  
 # left table  
 city\_info\_wide,  
 # right table  
 city\_telephone\_prexix  
 ) %>%  
 knitr::kable()

| city | population | area | density | telephon\_prefix |
| --- | --- | --- | --- | --- |
| Leicester | 329839 | 73.3 | 4500 | 0116 |
| Nottingham | 321500 | 74.6 | 4412 | NA |
| Birmingham | NA | NA | NA | 0121 |
| London | NA | NA | NA | 0171 |

# Option 3: using the pipe operator  
# and without using the argument "by"  
# as columns have the same name  
# in the two tables.  
# Same result as Option 1 and 2  
# left table  
city\_info\_wide %>%  
 # full join verb  
 dplyr::full\_join(  
 # right table  
 city\_telephone\_prexix  
 ) %>%  
 knitr::kable()

| city | population | area | density | telephon\_prefix |
| --- | --- | --- | --- | --- |
| Leicester | 329839 | 73.3 | 4500 | 0116 |
| Nottingham | 321500 | 74.6 | 4412 | NA |
| Birmingham | NA | NA | NA | 0121 |
| London | NA | NA | NA | 0171 |

# Option 4: using the pipe operator  
# and using the argument "by".  
# Same result as Option 1, 2 and 3  
# left table  
city\_info\_wide %>%  
 # full join verb  
 dplyr::full\_join(  
 # right table  
 city\_telephone\_prexix,  
 # columns to match  
 by = c("city" = "city")  
 ) %>%  
 knitr::kable()

| city | population | area | density | telephon\_prefix |
| --- | --- | --- | --- | --- |
| Leicester | 329839 | 73.3 | 4500 | 0116 |
| Nottingham | 321500 | 74.6 | 4412 | NA |
| Birmingham | NA | NA | NA | 0121 |
| London | NA | NA | NA | 0171 |

# Left join  
# Using syntax similar to Option 1 above  
# left join  
dplyr::left\_join(  
 # left table  
 city\_info\_wide,   
 # right table  
 city\_telephone\_prexix,  
 # columns to match  
 by = c("city" = "city")  
 ) %>%  
 knitr::kable()

| city | population | area | density | telephon\_prefix |
| --- | --- | --- | --- | --- |
| Leicester | 329839 | 73.3 | 4500 | 0116 |
| Nottingham | 321500 | 74.6 | 4412 | NA |

# Right join  
# Using syntax similar to Option 2 above  
# right join verb  
dplyr::right\_join(  
 # left table  
 city\_info\_wide,   
 # right table  
 city\_telephone\_prexix  
 ) %>%  
 knitr::kable()

| city | population | area | density | telephon\_prefix |
| --- | --- | --- | --- | --- |
| Leicester | 329839 | 73.3 | 4500 | 0116 |
| Birmingham | NA | NA | NA | 0121 |
| London | NA | NA | NA | 0171 |

# Inner join  
# Using syntax similar to Option 3 above  
# left table  
city\_info\_wide %>%  
 # inner join  
 dplyr::inner\_join(  
 # right table  
 city\_telephone\_prexix  
 ) %>%  
 knitr::kable()

| city | population | area | density | telephon\_prefix |
| --- | --- | --- | --- | --- |
| Leicester | 329839 | 73.3 | 4500 | 0116 |

### 7.3.1 Exercise: join

In the previous exercise we have created a table that shows national African sorghum production in 2019. In this exercise we will join crop production statistics with a table that contains national boundaries and visualize sorghum production quantities in a simple map.

1. Create an RScript and install and load the libraries tidyverse, knitr, ggplot2 and maps, if not done yet.
2. Copy the [code from the previous exercise](Solution_Exercise_DataWrangling1.R) into your new RScript. Note that the result of the pipe operations is *not* saved to a variable. Save it to a variable.
3. Use the ggplot2 function map\_data to convert the built in sample dataset world (comes with library maps) to a data frame:

world\_ctry <- map\_data("world")

1. Inspect the structure of this data frame. Every row represents a node (defined by long/lat) of a polygon feature (national boundaries).
2. Join tables (left table: geographic features, right table: sorghum production statistics) based on country names. Make sure to choose a join method (full\_join, inner\_join, left\_join or right\_join) that allows for retaining all the geographic features.

[My exercise solution](Solution_Exercise_DataWrangling2.R) creates a simple output map. In a subsequent lesson we will cover visualization methods in more detail.

Take a look at the [dplyr Cheatsheet](https://www.rstudio.com/resources/cheatsheets/) which shows the most important dplyr operations at a glance.

# 8 Spatial Data Manipulation

## 8.1 Introduction

* Briefly discuss the importance of spatial data wrangling in geoinformatics and related fields.
* Quick highlight of the difference between traditional data wrangling and spatial data wrangling.

## 8.2 Spatial Aggregation

* Aggregating spatial data based on certain attributes or spatial units.
* Introduce the sf and terra packages for aggregation tasks.

# Example code for spatial aggregation using sf and terra

## 8.3 Merging and Appending Spatial Data

* Combining multiple spatial datasets.
* Introduce the sf package for these tasks.

# Example code for merging and appending using sf

## 8.4 Handling Missing Data

* Discuss methods to identify, visualize, and impute missing data.
* Introduce the **tidyverse** suite for handling missing data in spatial datasets. (but again, watchout for **httr1** and **httr2** + (**time of writing 12.09.2023**)

# Example code for handling missing data using tidyverse

## 8.5 Spatial Data Cleaning

* Detecting and correcting topological errors.
* Introduce the sf and spatstat packages for these tasks.

# Example code for spatial data cleaning using sf and spatstat

## 8.6 Multidimensional Data Wrangling with stars

* Handling and manipulating multidimensional arrays.
* Discuss the importance of stars in the R spatial ecosystem.

# Example code for wrangling multidimensional data using stars

## 8.7 Challenges in Spatial Data Structures

* Issues with data integrity and accuracy.
* Handling different coordinate reference systems (CRS). (If you get for example a old or exotic dataset)

## 8.8 Dealing with large spatial datasets.

* Dive Deeper: Strategies for optimizing spatial data structures for large datasets and high-performance computing.

## 8.9 Case Study: A Complete Spatial Data Wrangling Workflow

* Would be a quite interesting change of wind: a comprehensive case study integrating various spatial data wrangling techniques…
* This could be based on a real-world scenario, such as urban planning, environmental monitoring, or transportation.
* Alternatively, a set of exercises allowing our students to apply the techniques learned in the chapter.

## 8.10 Introduction

* Briefly explain the difference between data wrangling and data manipulation in the spatial context.
* Highlight the importance of spatial data manipulation in GIS and spatial analysis workflows.
* Introduce the primary packages for this chapter: sf, terra, stars, and spatstat.

## 8.11 Filtering Spatial Data

* Filtering sf Objects
* Demonstrate how to filter spatial features in sf objects based on attribute conditions.
* Introduce the sf package for vector data filtering.

# Example code for filtering sf objects

## 8.12 Filtering Rasters with terra

* Show how to filter raster data based on cell values or spatial conditions using terra.

# Example code for filtering rasters with terra

## 8.13 Spatial Joins with sf (though.. we could also put that in Chapter 10 and make 12 bigger with spatial stat)

* Explain the concept of spatial joins.
* Demonstrate how to join two spatial datasets based on their spatial relationship using sf.

# Example code for spatial joins with sf

## 8.14 Geometric Operations with sf

### 8.14.1 Intersect

* Demonstrate how to find the common spatial elements between two datasets using sf.

# Example code for intersect operation with sf

### 8.14.2 Buffer

* Show how to create a buffer zone around a spatial feature using sf.

# Example code for buffer operation with sf

### 8.14.3 Union and Difference

* Explain and demonstrate the union and difference operations between two spatial datasets using sf.

# Example code for union and difference operations with sf

## 8.15 Coordinate Transformations and Reprojections with sf and terra

* Discuss the importance of having datasets in the same coordinate system.
* Demonstrate how to reproject spatial data using both sf for vector data and terra for raster data.

# Example code for coordinate transformations with sf and terra

## 8.16 Raster Operations with terra

### 8.16.1 Resampling

* Show how to change the resolution of raster datasets using terra.

# Example code for raster resampling with terra

### 8.16.2 Raster Algebra

* Demonstrate basic raster algebra operations like addition, subtraction, and multiplication of raster layers using terra.

# Example code for raster algebra with terra

## 8.17 Applied Spatial Statistics with spatstat

(I think, this will be yours Christian :) But it might be a pretty good idea to showcase it in chapter 12, atleast basics.): - Introduction to spatstat: - Briefly introduce the spatstat package and its capabilities in handling advanced spatial statistics. - Point Pattern Analysis (ppa): - Dive deeper into point pattern analysis, discussing its importance and applications in spatial statistics. - Spatial Clustering: - Discuss the concept of spatial clustering and its relevance in identifying patterns in spatial data. - Illustrate with examples how spatstat can be used for spatial clustering analysis. - Nearest Neighbor Analysis: - Introduce Nearest Neighbor as a crucial type of spatial query often used in spatial statistics to analyze the spatial distribution of features. - Discuss its applications in evaluating the spatial arrangement of points, assessing spatial randomness, and identifying - patterns such as clustering or dispersion. - Provide practical examples demonstrating how to perform Nearest Neighbor analysis using spatstat, and discuss the interpretation of the results.

## 8.18 Comparative Analysis of Packages

* After that intense usage of packages, Investigate and compare the performance, ease of use, and capabilities of sf, terra, stars, and spatstat for various spatial operations. Compiled languages (for example via C, C++) and packages and their difference in speed.

## 8.19 Exercise

* Provide exercises that allow readers to apply the techniques learned in the chapter, ensuring they get hands-on experience with each of the introduced packages.

## 8.20 Topological Relationships

* Discuss spatial relationships like:
* Adjacency (what’s next to what)
* Connectivity (how things are connected)
* Containment (what contains what)
* Importance of understanding topological relationships in spatial analysis and how one might implement that in R.

### 8.20.1 Resampling Raster Data in Terra

#### 8.20.1.1 What is Resampling?

Resampling is the process of changing the spatial resolution of a raster dataset. This is useful when you want to align multiple rasters that have different resolutions or when you want to change the level of detail in your analysis.

Step 1: Check the Original Resolution Before resampling, it’s important to know the original resolution of your raster. You can use the res() function to check this.

#```{r}

#res(elev) # shows us the resolution of 0.5. 0.5 #plot(elev) #``` Where res is the new resolution.

Step 2: Choose a New Resolution Decide what the new resolution should be. This could be higher or lower than the original, depending on your needs.

# Resample the raster to the new resolution  
new\_res <- c(0.5, 0.5) # Example new resolution, for example double the size.

Step 3: Perform Resampling You can use the resample() function to change the resolution. The method can be “nearest”, “bilinear”, etc., depending on the type of interpolation you want to use.

#```{r}

#elev\_resampled <- resample(elev, new\_res, method=“bilinear”) # Create a target raster with the new resolution #target\_raster <- rast(elev, res=new\_res) # Resample the original raster to match the target raster #elev\_resampled <- resample(elev, target\_raster, method=“bilinear”) # Check the new resolution #res(elev\_resampled)

#```

Masking: To mask a raster using another raster or vector, you can use the mask() function.

#{r} #r\_masked <- mask(r, mask\_layer) # Where mask\_layer is another raster or a vector layer

### 8.20.2 Basic Raster Operations

#### 8.20.2.1 Cropping:

What is Raster Cropping? Raster cropping is the process of selecting a specific area (or “extent”) from a larger raster dataset. This is useful when you want to focus your analysis on a particular region.

Why Do We Need to Check the Original Raster’s Extent? Before cropping, it’s crucial to know the extent of the original raster. This ensures that the area you want to crop actually exists within the original raster. If the extents don’t overlap, you’ll get an error.

How to Check the Extent? You can use the ext() function from the terra package to check the extent of a raster.

#{r} # Check the extent of the original raster #ext(elev) # - How to Crop? Once you know the extent, you can specify a new extent within that range for cropping.

#```{r} # Define a new extent that is within the original #raster’s extent #crop\_extent <- ext(-1, 1, -1, 1) # These coordinates #are within the original extent

# 9 Crop the raster

#elev\_cropped <- crop(elev, crop\_extent) #elev\_cropped

#``` ### Common Pitfalls - Non-overlapping Extents: Make sure the cropping extent overlaps with the original raster’s extent, otherwise, you’ll get an error. - Coordinate Systems: Ensure that the raster and the extent are in the same coordinate system!

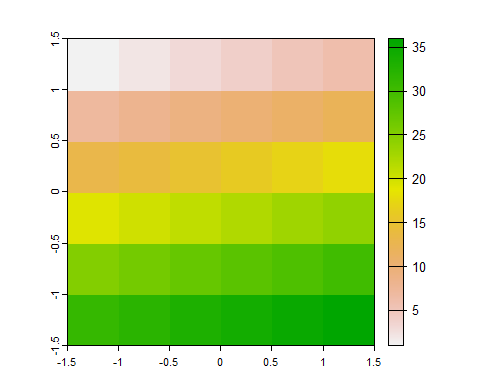
new:

**Reading Raster Data**

# Load the terra package  
library(terra)  
  
# use the elevation tif file from "spData"  
# "The raster data represents elevation in meters and uses WGS84 as a coordinate reference system."  
elev <- system.file("raster/elev.tif", package = "spData")  
elev <- rast(elev)  
elev

## class : SpatRaster   
## dimensions : 6, 6, 1 (nrow, ncol, nlyr)  
## resolution : 0.5, 0.5 (x, y)  
## extent : -1.5, 1.5, -1.5, 1.5 (xmin, xmax, ymin, ymax)  
## coord. ref. : lon/lat WGS 84 (EPSG:4326)   
## source : elev.tif   
## name : elev   
## min value : 1   
## max value : 36

plot(elev)

 Once you’ve read the raster data, you can explore its properties using various functions:

* ncol(r): Number of columns
* nrow(r): Number of rows
* res(r): Resolution (pixel size)
* crs(r): Coordinate Reference System

# 10 Read and write data

In previous exercises we have **read** data from a CSV file into our script. Similarly we can also **write** code outputs to file.

In this lesson you will learn to read and write plain-text and spatial vector file formats. Moreover, we will retrieve online data by means of a data API.

## 10.1 Read and write tabular data

A series of formats are based on plain-text files.

For instance…

* comma-separated values files .csv
* semi-colon-separated values files .csv
* tab-separated values files .tsv
* other formats using custom delimiters
* fix-width files .fwf

The [readr library](https://readr.tidyverse.org/) (also part of [Tidyverse](https://www.tidyverse.org/packages/#import)) provides a series of functions that can be used to load from and save to such data formats. For instance, the read\_csv function reads a comma delimited (CSV) file from the path provided as the first argument.

The code example below reads a CSV file that contains global fishery statistics provided by the [World Bank](https://datacatalog.worldbank.org/dataset/world-development-indicators) and queries Norwegian entries. The function writes\_csv writes these entries to a new CSV file.

library(tidyverse)  
  
fishery\_data <- readr::read\_csv("data/capture-fisheries-vs-aquaculture.csv")  
#print(fishery\_data)  
  
#print(typeof(fishery\_data$))  
  
fishery\_data %>%  
 dplyr::filter(Entity == "Norway") %>%  
 readr::write\_csv("data/capture-fisheries-vs-aquaculture-norway.csv", append=FALSE) %>%  
   
 dplyr::slice\_head(n = 3) %>%  
 knitr::kable()

| Entity | Code | Year | Aquaculture production (metric tons) | Capture fisheries production (metric tons) |
| --- | --- | --- | --- | --- |
| Norway | NOR | 1960 | 1900 | 1609362 |
| Norway | NOR | 1961 | 900 | 1758413 |
| Norway | NOR | 1962 | 200 | 1572913 |

In order to run the script, [download the CSV file](data/capture-fisheries-vs-aquaculture.csv). Then copy and run the code in a new R-script.

Other important packages for reading tabular data are [readxl](https://readxl.tidyverse.org/) for Excel (.xls and .xlsx) and [haven](https://haven.tidyverse.org/) for SPSS, Stata and SAS data.

## 10.2 Read and write vector data

The library sf makes it easy to read and write vector datasets such as shapefiles. The name (sf stands for **simple features**) already implies that sf supports simple feature access via R.

Simple features is a widely supported data model that underlies data structures in many GIS applications including QGIS and PostGIS. A major advantage of this is that using the data model ensures your work is cross-transferable to other set-ups, for example importing from and exporting to spatial databases.

In order to load vector data in an R-Script, we can use the function st\_read(). In the code block below, a [shapefile](data/nc.zip) ([North Carolina sample data](https://r-spatial.github.io/sf/reference/nc.html)) is loaded and assigned to a variable nc.

The next line creates a basic map in sf by means of plot(). By default this creates a multi-panel plot, one sub-plot for each variable of the object.

library(sf)  
  
nc <- sf::st\_read("data/nc.shp")  
  
plot(nc)



The library sf represents features as records in a data frame or tibble with a geometry list-column. The example below renders three features (rows) of variable nc including the geometry column as well as the attributes AREA (feature area) and NAME (name of county):

nc %>%  
 dplyr::select(AREA, NAME, geometry) %>%  
 dplyr::slice\_head(n = 3) %>%  
 knitr::kable()

| AREA | NAME | geometry |
| --- | --- | --- |
| 0.114 | Ashe | MULTIPOLYGON (((-81.47276 3… |
| 0.061 | Alleghany | MULTIPOLYGON (((-81.23989 3… |
| 0.143 | Surry | MULTIPOLYGON (((-80.45634 3… |

sf also includes a number of operations to manipulate the geometry of features such as st\_simplify:

sf::st\_simplify(nc) %>%  
 plot(., max.plot = 1)



You may have recognized that a dot (.) is used as a parameter in the function plot(). The dot represents the piped value. In the example above the dot is used to define the simplified geometry of nc as first parameter of function plot() and max.plot = 1 as the second.

In the next example, the st\_geometry() retrieves the geometry attribute from variable nc, function st\_centroid() calculates the centroid of the polygon geometry (counties) and function st\_write writes the centroid point geometry to file.

sf::st\_geometry(nc) %>%  
 sf::st\_centroid() %>%  
 sf::st\_write("data/nc-centroids.shp", delete\_dsn = TRUE) %>%  
 plot(pch = 3, col = 'red')



The online book [Geocomputation with R](https://r.geocompx.org/) offers a more comprehensive explanation of available geometric, attribute and spatial data operations. For a quick overview, you may turn to the [sf cheatsheets](https://github.com/rstudio/cheatsheets/blob/master/sf.pdf).

In order to test the code on your machine, [download](data/nc-centroids.zip) the North Carolina dataset and install the libraries sf and Rcpp before you run the code in an R-Script.

The plot() function offers a large number of arguments that can be used to customize your map. Replace ‘Area’ in the map above by a more meaningful map title. Turn to the [documentation](https://r-spatial.github.io/sf/reference/plot.html) for more information.

See [my solution](Solution_Miniexercise_plot.R)!

Similar R functions are also available for raster data (see package [raster: Geographic Data Analysis and Modeling](https://cran.r-project.org/web/packages/raster/raster.pdf))

## 10.3 Data API

API is the acronym for Application Programming Interface, which is a software intermediary that allows two applications to talk to each other.

By means of an API we can read, write and modify information on the web. The following video briefly introduces the technology behind it.

## PhantomJS not found. You can install it with webshot::install\_phantomjs(). If it is installed, please make sure the phantomjs executable can be found via the PATH variable.

Figure 10.1: Video (3:13 min): REST API concepts and examples.

The most important take away messages are:

1. With a REST API web data is accessible through a URL (Client-Server call via HTTP protocol)
2. The HTTP **Get** Method delivers data (a Response) - i.e. is used to read data, the HTTP **Post** Method is used to create new REST API resources (write data).
3. URL Parameters are used to filter specific data from a response.
4. Typically, APIs can return data in a number of different formats.
5. JSON is a very popular format for transferring web data.
6. The two primary elements that make up JSON are keys and values.

The library [httr2](https://httr2.r-lib.org/) offers functions to programmatically implement API calls in an R script. We will make use of this library to let our R script interact with the [APIs](https://openweathermap.org/api) offered by [OpenWeather Map](https://openweathermap.org/). that contain historical and real-time weather data for retrieval. In the upcoming example we will make a call to the [Current Weather API](https://openweathermap.org/current), which is one out of currently [10 free APIs](https://openweathermap.org/price) provided by OpenWeather Map.

For accessing the data with httr2, we’ll construct a URL composed of a reference to the data source (base) and parameters to filter the desired data subset (lat and lon). In httr2, the URL and query parameters can be separated, simplifying the process. The parameters are passed as a named list to the request() function:

library(httr2)  
  
# Define the base URL and parameters  
base <- "http://api.openweathermap.org/data/2.5/weather"  
lat <- "47.81"   
lon <- "13.03"  
apiKey <- "3f87141421b32590d50416aae5ca780c"  
  
# Construct the full URL with query parameters using sprintf  
full\_url <- sprintf("%s?lat=%s&lon=%s&appid=%s", base, lat, lon, apiKey)  
  
# Display the constructed URL  
print(full\_url)

## [1] "http://api.openweathermap.org/data/2.5/weather?lat=47.81&lon=13.03&appid=3f87141421b32590d50416aae5ca780c"

In the code above, the function sprintf() assembles the base URL and parameters. The URL (of type string) will then pass the request() function to create a request object, and req\_perform() is used to execute the HTTP Get method.

Note: The OpenWeather API requires an API Key to be passed as a parameter in the call. Get your [personal Key](https://openweathermap.org/appid) to implement your own API requests.

If your code returns an error 401, this most likely indicates that your key is not activated yet. According to the [FAQs](https://openweathermap.org/faq), it may take a couple of hours until your key is active.

By default, the req\_perform() function returns a response object. Printing a response object provides useful information such as the actual URL used (after any redirects), the HTTP status, and the content type.

You can extract important parts of the response with various helper functions such as resp\_status() and resp\_body\_json():

# Create the request and perform it  
req <- request(full\_url)  
resp <- req\_perform(req)  
  
# Check the status code of the response  
print(resp\_status(resp))

## [1] 200

# View the content structure of the response in JSON  
str(resp\_body\_json(resp))

## List of 13  
## $ coord :List of 2  
## ..$ lon: num 13  
## ..$ lat: num 47.8  
## $ weather :List of 1  
## ..$ :List of 4  
## .. ..$ id : int 802  
## .. ..$ main : chr "Clouds"  
## .. ..$ description: chr "scattered clouds"  
## .. ..$ icon : chr "03d"  
## $ base : chr "stations"  
## $ main :List of 6  
## ..$ temp : num 296  
## ..$ feels\_like: num 296  
## ..$ temp\_min : num 294  
## ..$ temp\_max : num 298  
## ..$ pressure : int 1023  
## ..$ humidity : int 80  
## $ visibility: int 10000  
## $ wind :List of 2  
## ..$ speed: num 0.51  
## ..$ deg : int 0  
## $ clouds :List of 1  
## ..$ all: int 40  
## $ dt : int 1696937749  
## $ sys :List of 5  
## ..$ type : int 1  
## ..$ id : int 6877  
## ..$ country: chr "AT"  
## ..$ sunrise: int 1696915126  
## ..$ sunset : int 1696955456  
## $ timezone : int 7200  
## $ id : int 2766824  
## $ name : chr "Salzburg"  
## $ cod : int 200

The Current Weather API call returned a number of real-time weather variables such as temperature, air pressure or humidity for the location of Salzburg. Current weather data may alternatively be retrieved by City ID or City Name (see the [documentation](https://openweathermap.org/current) to get an overview of available API parameters).

Inspect the response object by means of a function called headers(). What methods are allowed when accessing the Current Weather API?

**See solution!**

Only GET and POST methods are allowed.

Other APIs allow to update existing REST API resources (PUT method) or to delete a REST API resource (DELETE method).

To facilitate subsequent analyses and data visualization, we can convert the content of the return object to a data frame. With httr2, you can directly retrieve the JSON content and convert it to a data frame:

# Using resp\_body\_json to get the JSON content of the response  
response\_content <- resp\_body\_json(resp)  
response\_df <- as.data.frame(response\_content)  
response\_df

## coord.lon coord.lat weather.id weather.main weather.description weather.icon  
## 1 13.03 47.81 802 Clouds scattered clouds 03d  
## base main.temp main.feels\_like main.temp\_min main.temp\_max main.pressure  
## 1 stations 295.78 296.19 294.16 297.9 1023  
## main.humidity visibility wind.speed wind.deg all dt sys.type sys.id  
## 1 80 10000 0.51 0 40 1696937749 1 6877  
## sys.country sys.sunrise sys.sunset timezone id name cod  
## 1 AT 1696915126 1696955456 7200 2766824 Salzburg 200

Copy the code snippets above to a new R-Script. **Make sure to replace the key in the code example by your own API key!** If you need help, please turn to the discussion forum.

Note: If you want more advanced JSON manipulation or flattening, you can still use the jsonlite library. But for the basic conversion shown here, it’s not necessary.

[Tidyverse](https://www.tidyverse.org/packages/) also provides other packages for reading data such as [DBI](https://cran.r-project.org/web/packages/DBI/) for relational databases [jsonlite](https://cran.r-project.org/web/packages/jsonlite/) for JSON and [xml2](https://cran.r-project.org/web/packages/xml2/) for XML.

… more to come!

## 10.4 Additional Thoughts so far for revision

### 10.4.1 **7.4 Advanced JSON Handling with httr2**

While touched on converting API responses to data frames using **jsonlite**, we could delve deeper into JSON handling:

* **Parsing Nested JSON**: Often, APIs return nested JSON structures. Teach how to handle and flatten these structures to convert them into usable data frames.
* **JSON to List**: Before converting to a data frame, JSON can be converted to a list in R. This can be useful for certain types of analyses or data manipulations.

### 10.4.2 **7.5 Caching Data with httr2**

Given that **httr2** has caching capabilities, this can be a valuable addition:

* **Why Cache?**: Explain the benefits of caching, especially when dealing with rate-limited APIs or to speed up development and analysis by not fetching data repeatedly.
* **Implementing Caching**: Show how to use **req\_cache()** in **httr2** to cache API responses.

### 10.4.3 **7.6 Error Handling and Retries with httr2**

Building on the capabilities of **httr2**:

* **Understanding HTTP Errors**: Briefly explain common HTTP errors like 404 (Not Found) or 429 (Too Many Requests) and why they might occur when working with APIs.
* **Implementing Retries**: Show how to use **req\_retry()** in **httr2** to automatically retry if a request fails.

### 10.4.4 **7.7 Advanced API Techniques**

* **Pagination**: Many APIs return data in pages, especially if there’s a lot of data. Teach how to handle paginated responses and fetch all data.
* **Rate Limiting**: Explaining what rate limiting is and how to handle it, possibly by introducing pauses in requests or by using caching.

### 10.4.5 **7.8 Introduction to Web Scraping** (Altough, this might be too much.. and even if it’s important, for data sourcing, it’s kind of a grey area, limited by rate limits or considerations with ToS.)

While APIs are a clean way to get data, not all websites offer them. We could introduce the concept of web scraping:

* **What is Web Scraping?**: Briefly explain the concept and why it might be necessary.
* **Using rvest for Web Scraping**: Introduce the **rvest** package, which is part of the tidyverse and is used for web scraping. We could provide a simple example of scraping data from a website.
* **Morality of Web Scraping**: via. Ethical, Legal, Technical Considerations

# 11 Data visualization

R has a very rich set of graphical functions. The [R Graph Gallery](https://www.r-graph-gallery.com/) provides a large number of examples (including code).

In this lesson you will get to know the [ggplot2 library](https://ggplot2.tidyverse.org/), which is the most popular library for creating graphics in R. You will learn to create standard graphs such as histograms, boxplots or scatterplots as well as maps by means of the [ggplot2 library](https://ggplot2.tidyverse.org/).

## 11.1 The Grammar of Graphics

The [ggplot2 library](https://ggplot2.tidyverse.org/) is part of [Tidyverse](https://www.tidyverse.org/) and offers a series of functions for creating graphics declaratively, based on the concepts outlined in the Grammar of Graphics by [Leland Wilkinson](https://en.wikipedia.org/wiki/Leland_Wilkinson).

The grammar of graphics is a schema that enables us to concisely describe the components of a graphic. These components are called **layers of grammatical elements.** Overall, the grammar comprises seven layers:

1. Data - The data element is the dataset itself.
2. Aesthetics - This layer defines how variables are mapped onto scales (see description below).
3. Geometries - This element determines how our data is being displayed (e.g. bars, points, lines etc.)
4. Facets - Faceting splits the data into subset and displays the same graph for every subset.
5. Statistics - These are statistics derived from the data (add mean, median, quartile, etc.).
6. Coordinates - This element determines the transformation of axes (e.g. change spacing of displayed data)
7. Themes - This element determines the graphics background.

The **aesthetics layer** offers a number of different options to map data onto visual variables. A **visual variable** is an aspect of a **mark** that can be controlled to change its appearance.

Visual variables are:

* Size
* Shape
* Orientation
* Colour (hue)
* Colour value (brightness)
* Texture
* Position (map variable to x or y axis)

For instance, in Figure 11.1 variables ‘Gdp per capita’ and ‘Life Expectancy’ are mapped onto the x and y axes (visual variable **position**), variables ‘national population’ and ‘world regions’ are mapped onto visual variables **size** and **color**.



Figure 11.1: Visual variables color and size

In order to make that concept clearer, a number of examples will be presented in upcoming sections.

The basic concept behind the grammar of graphics is described in an [article](https://byrneslab.net/classes/biol607/readings/wickham_layered-grammar.pdf) by Hadley Wickham.

## 11.2 Visualization of distributions

As already announced above, functions in the ggplot2 library are structured according to the Grammar of Graphics. To create a graph in ggplot2, we need to provide input data, specify visual variables by means of an aesthetics element (aes()), specify the geometry of marks (e.g., geom\_point) and apply transformations (axis spacing) and themes (background theme of the graph).

We start the analysis with a simple histogram, to explore the distribution of air quality [**data**](data/AirQualityUpperAut.csv) that has been measured at different locations in Upper Austria.

The data includes the following **variables**

* **time** of measurement
* ID of the measuring **station**
* measured meteorological [**component**](data/MetadataAirqual.xlsx)
* [**meantype**](data/MetadataAirqual2.xlsx)
* **unit** of measurement
* measurement **value**

The following code renders the first five lines of the dataset in a knitr table:

library(tidyverse)  
library(knitr)  
  
#read csv data, Note: Semicolon seperated CSVs can be loaded by function 'read\_delim()'  
airquality <- read\_delim("data/AirQualityUpperAut.csv", delim = ";")  
  
airquality %>%  
 dplyr::slice\_head(n = 5) %>%  
 knitr::kable()

| time | station | component | meantype | unit | value |
| --- | --- | --- | --- | --- | --- |
| 21.10.2021 13:30 | C001 | BOE | HMW | m/s | 14.1 |
| 21.10.2021 14:00 | C001 | BOE | HMW | m/s | 12.0 |
| 21.10.2021 14:30 | C001 | BOE | HMW | m/s | 10.1 |
| 21.10.2021 15:00 | C001 | BOE | HMW | m/s | 7.9 |
| 21.10.2021 15:30 | C001 | BOE | HMW | m/s | 9.2 |

The code below filters the airquality dataset by measurement component and temporal resolution. Then the data subset is passed as a first argument to function ggplot(). In the second argument, we map the variable value onto the x-axis with the aesthetics argument aes(). geom\_histogram() specifies the geometry of the plot and theme\_bw() is used to add a [background theme](https://ggplot2.tidyverse.org/reference/ggtheme.html).

#filter NO2 measurements with temporal resolution 30min (HMW)  
airquality %>%  
 dplyr::filter(component == "NO2" & meantype == "HMW") %>%  
  
 #create plot   
 ggplot2::ggplot(., #the dot '.' represents the piped value   
 aes(   
 x = value #map variable 'value' onto x-axis  
 )  
 ) +  
 ggplot2::geom\_histogram() + #define geometry  
 ggplot2::theme\_bw() #define theme



If we aim to distinguish between measurements of respective measurement stations, we can map the variable ‘station’ onto visual variable color:

airquality %>%  
 dplyr::filter(component == "NO2" & meantype == "HMW") %>%  
 dplyr::filter(station == "S125" | station == "S431" | station == "S270") %>% #select 3 stations  
  
 ggplot2::ggplot(.,   
 aes(   
 x = value,   
 fill = station  
   
 )  
 ) +  
   
 ggplot2::xlab("NO2 [mg/m^3]") +  
 ggplot2::ylab("Count") +  
 scale\_fill\_manual(name = "Measurement stations", values = c("grey20", "grey50", "grey80")) +  
 ggplot2::geom\_histogram() +   
 ggplot2::theme\_bw()



This is implemented by adding an attribute fill = station to the aesthetics element (aes()). ggplot2 offers a number of [functions](https://ggplot2.tidyverse.org/reference/scale_manual.html) to specify your own set of mappings from levels in the data to aesthetic values. In the example above the function scale\_fill\_manual() is used to map the three levels S125, S270 and S431 to the fill [colors](http://sape.inf.usi.ch/quick-reference/ggplot2/colour) grey20, grey50 and grey80. Instead of ggplot colors, you can also use [hex color codes](https://htmlcolorcodes.com/).

Note that plot components are [added](https://ggplot2.tidyverse.org/reference/gg-add.html) by means of a plus ‘+’ sign. It allows you to start simple, and then get more and more complex.

So far, we have added two axis labels. Create a new R-Script, download the [input data](data/AirQualityUpperAut.csv), recreate the histogram and insert one additional line of code to add a plot title (see [documentation](https://ggplot2.tidyverse.org/reference/labs.html)).

**See solution!**

Insert title:

ggplot2::ggtitle(“Nitrogen dioxide concentration”)

## 11.3 Boxplots

The same basic syntax is used to create other types of plots like bar plots (use geometry [geom\_bar()](https://ggplot2.tidyverse.org/reference/geom_bar.html) or [geom\_col()](https://ggplot2.tidyverse.org/reference/geom_bar.html), line plots (use geometry [geom\_line()](https://ggplot2.tidyverse.org/reference/geom_path.html)) and many [others](https://ggplot2.tidyverse.org/reference/index.html).

For instance, if we replace geom\_histogram() by geom\_boxplot(), the value distribution of NO2 measurements is visualized by means of a box plot:

#filter NO2 measurements with temporal resolution 30min (HMW)  
airquality %>%  
 dplyr::filter(component == "NO2" & meantype == "HMW") %>%  
  
 #create plot   
 ggplot2::ggplot(., #the dot '.' represents the piped value   
 aes(   
 x = value #map variable 'value' onto x-axis  
 )  
 ) +  
 ggplot2::xlab("NO2 [mg/m^3]") +  
 ggplot2::geom\_boxplot() + #define geometry  
 ggplot2::theme(  
 axis.text.y = element\_blank(), #remove text and ticks from y axis  
 axis.ticks.y = element\_blank()  
 )



Note that the last two lines remove text and tick marks from the y-axis of the plot.

Just as histograms, box plots are used to inspect distributions in data. The [interpretation](https://ggplot2.tidyverse.org/reference/geom_boxplot.html), however, does require some additional information.

The lower and upper edge of the box (the so-called lower and upper **hinges**) correspond to the first and third [quartiles](https://www.statisticshowto.com/probability-and-statistics/statistics-definitions/what-are-quartiles/). The vertical line that separates the box indicates the **median** value (second quartile).

The upper **whisker** extends from the hinge to the largest value no further than 1.5 \* IQR from the hinge (where IQR is the inter-quartile range, or distance between the first and third quartiles). The lower whisker extends from the hinge to the smallest value at most 1.5 \* IQR of the hinge. Data beyond the end of the whiskers are called “outlying” points and are plotted individually.

In our histogram examples, we have mapped the variable ‘station’ onto visual variable color to separately visualize measurements of different stations. Try to apply the same approach to render measurements of stations S125, S270 and S431 separately in a box plot.

See [my solution](Solution_Exercise_boxplots.R)!

## 11.4 Scatterplots

While boxplots and histograms reveal distributions in data, scatterplots are used to illustrate relationships between variables.

In the following example, air temperature (TEMP) and relative humidity (RF) measured in a 30min interval by station ’S108‘ are filtered from data table ’airquality‘. Then the two tables are joined by their common field ’time‘. The joined table is used as data input to render a scatterplot with temperature on the x-axis and relative humidity on the y-axis.

#half-hourly temperature measurement of station S108 to data frame  
temp\_tab <- airquality %>%  
 dplyr::filter(component == "TEMP" & meantype == "HMW" & station == "S108")   
  
#half-hourly relative humidity measurement of station S108 to data frame  
humidity\_tab <- airquality %>%  
 dplyr::filter(component == "RF" & meantype == "HMW" & station == "S108")   
  
#join humidity and temperature tables by common field 'time'  
temp\_tab %>%  
 dplyr::inner\_join(  
 # right table  
 humidity\_tab,  
 # columns to match  
 by = c("time" = "time")  
 ) %>%  
   
 dplyr::select(time, value.x, value.y) %>% #select relevant columns from joined table  
  
 #create plot   
 ggplot2::ggplot(.,   
 aes(   
 x = value.x,   
 y = value.y  
 )  
 ) +  
 ggplot2::xlab("air temperature [°C]") +  
 ggplot2::ylab("relative humidity [%]") +  
 ggplot2::geom\_point(color="blue") + #define geometry scatterplot, with point color blue  
 ggplot2::geom\_smooth(method=lm , color="red", fill="#69b3a2", se=TRUE) + #with linear trend and confidence interval  
 ggplot2::theme\_minimal()



The plot reveals a trend between the two variables temperature and humidity. Relative humidity tends to increase as temperature decreases and vice versa. In [this video](https://www.youtube.com/watch?v=saojnBSBUuk) you can find an explanation for the inverse proportional relationship between relative humidity and air temperature.

Obviously, due to other predictors such as windspeed, evaporation etc., this relationship is not perfectly linear, however, it can be closely approximated by means of a linear regression line. Deviations from the linear model are indicated by a 95% confidence interval.

Copy and run the code example from above in a new R-Script. Note that the air quality data as well as the tidyverse library must be loaded to run the code in a standalone R-script file.

[Complete Script!](Solution_Scatterplot.R)

Go through the code example line by line and answer the following questions:

1. How many measurements (records) are included in the scatterplot?
2. What is value.x and value.y?
3. We have used the function geom\_smooth() to fit a linear regression model (method = lm). What is the purpose of argument se?

**See answers!**

1. Measurements between 21.10.2021 14:00 and 22.10.2021 12:00, half-hourly interval -> 45 records (see environment tab in RStudio)
2. Temperature and humidity values in the data frame tables humidity\_tab and temp\_tab are both denoted value. In order to avoid ambiguities, the join function renames columns.
3. The argument defines whether confidence bounds are displayed (se is TRUE by default).

## 11.5 Map visualization

In the previous lesson you have already learned how to read vector data and create simple map layouts by means of the plot() function. In this concluding section, we will use the ggplot() library to create more complex map layouts.

In order to replicate the code examples below, you will have to install and load the libraries sf() (remember: sf stands for simple features and is used to read and write vector data) and ggplot(). Also download the [North Carolina](data/nc.zip) and [US States](data/us-states.zip) sample datasets.

First, let us start with creating a single-layer base map:

library(sf)  
library("ggplot2")  
  
nc <- sf::st\_read("data/nc.shp")  
  
ggplot(data = nc) +  
 geom\_sf() +  
 xlab("Longitude") + ylab("Latitude") +  
 ggtitle("North Carolina", subtitle = paste0("(", length(unique(nc$NAME)), " counties)"))



In the code above, we first load the North Carolina shapefile as an sf() object and then assign the data to the ggplot() graph. The geom\_sf function adds a geometry stored in a sf object. Other map components such as title and axis labels are added by means of a plus sign. Note that length(unique(nc$NAME)) returns the count of table rows, which corresponds to the number of geometries/counties. Geometry count and string “counties” are concatenated by function paste0().

The geometry element geom\_sf provides a number of arguments to customize the appearance of vector features:

ggplot(data = nc) +   
 geom\_sf(color = "black", fill = "lightgreen")



Data can also be mapped onto visual variables in the same way as with diagram plots. In the example below, the variable AREA is mapped onto visual variable fill color:

ggplot(data = nc) +  
 geom\_sf(aes(fill = AREA)) +  
 scale\_fill\_viridis\_c(option = "plasma", trans = "sqrt")



The function coord\_sf() allows to deal with the coordinate system, which includes both projection and extent of the map. By default, the map will use the coordinate system of the first layer or if the layer has no coordinate system, fall back on the geographic coordinate system WGS84. Using the argument crs, it is possible to override this setting, and project on the fly to any projection that has an [EPSG code](https://epsg.io/). For instance, we may change the coordinate system to EPSG 32618, which corresponds to WGS 84 / UTM zone 18N:

ggplot(data = nc) +  
 geom\_sf() +  
 coord\_sf(crs = st\_crs(32618))



The extent of the map can also be set in coord\_sf, in practice allowing to “zoom” in the area of interest, provided by limits on the x-axis (xlim), and on the y-axis (ylim). The limits are automatically expanded by a fraction to ensure that data and axes do not overlap; it can also be turned off to exactly match the limits provided with expand = FALSE:

library("ggspatial")  
ggplot(data = nc) +  
 geom\_sf() +  
 coord\_sf(xlim = c(-78.9, -75.5), ylim = c(34, 34.85), expand = FALSE) +  
 annotation\_scale(location = "br", width\_hint = 0.5) +  
 annotation\_north\_arrow(location = "bl", which\_north = "true",   
 pad\_x = unit(14.5, "cm"), pad\_y = unit(0.8, "cm"),  
 style = north\_arrow\_fancy\_orienteering)



Note that scale bar and north arrow are available with package ggspatial.

In the following example, we will assign labels to vector features. The function geom\_text() can be used to add a layer of text to a map using geographic coordinates. The North Carolina dataset contains county names as column (column: NAME). In order to define label positions, we take the centroids of the county polygons (function st\_centroid()), derive X and Y coordinates from centroids (function st\_coordinates()), merge the new X and Y columns with the columns of nc and assign the output to a new variable identifier nc\_points:

nc\_points <- cbind(nc, st\_coordinates(st\_centroid(nc$geometry)))

I have used a standard syntax to create variabe nc\_points. Convert the code to pipe operator syntax.

By the way, pipe operators are available with library magrittr, which is part of tidyverse. So make sure to load tidyverse in your script.

**See solution!**

st\_centroid(nc$geometry) %>% st\_coordinates() %>% cbind(nc, .)

Note that the reading direction of pipe syntax code is from left to right (more intuitive), whereas standard syntax (nested functions) is read from right to left.

After deriving centroid coordinates from nc geometries, we call the new variable nc\_points in function geom\_text and map X and Y columns (centroid coordinates) onto visual variables x and y (position in graph) and also map column NAME onto visual variable label. Moreover, we can insert individual text annotations manually by means of function annotate():

ggplot(data = nc) +  
geom\_sf() +  
geom\_text(data= nc\_points,aes(x=X, y=Y, label=NAME),  
 color = "darkblue", fontface = "bold", check\_overlap = FALSE, size = 3) +  
annotate(geom = "text", x = -76.5, y = 34.3, label = "Atlantic Ocean",   
 fontface = "italic", color = "grey22", size = 5) +  
coord\_sf(xlim = c(-78.9, -75.5), ylim = c(34, 34.85), expand = FALSE)



In a final example, the methods introduced so far are combined to create a more comprehensive map visualization:

us\_states <- sf::st\_read("data/us-states.shp")  
us\_states\_points <- st\_centroid(us\_states)  
us\_states\_points <- cbind(us\_states, st\_coordinates(st\_centroid(us\_states$geometry)))  
  
ggplot(data = nc) +  
 geom\_sf(data = us\_states, fill= "antiquewhite1") +  
 geom\_sf(aes(fill = AREA)) +  
 geom\_label(data= us\_states\_points,aes(x=X, y=Y, label=NAME),  
 color = "black", fontface = "bold", check\_overlap = FALSE, size = 3, nudge\_x = 0.5) +  
 annotation\_scale(location = "br", width\_hint = 0.5) +  
 annotation\_north\_arrow(location = "bl", which\_north = "true",   
 pad\_x = unit(11, "cm"), pad\_y = unit(0.8, "cm"),  
 style = north\_arrow\_fancy\_orienteering) +   
 scale\_fill\_viridis\_c(trans = "sqrt", alpha = .4) +  
 coord\_sf(xlim = c(-84.9, -70), ylim = c(24.5, 37), expand = FALSE) +  
 xlab("Longitude") + ylab("Latitude") +  
 ggtitle("US Southeast", subtitle = "(Detail: North Carolina)") +  
 annotate(geom = "text", x = -76.5, y = 30.5, label = "Atlantic Ocean",   
 fontface = "italic", color = "grey22", size = 6) +  
 theme(panel.grid.major = element\_line(color = gray(0.5), linetype = "dashed",   
 size = 0.1), panel.background = element\_rect(fill = "aliceblue"))



The function geom\_sf() is used to add US state polygons as an additional layer. The function geom\_label() is used as an alternative to function geom\_text(). Note that nudge\_x of function geom\_label() is introduced to offset labels horizontally.

Eventually, we save a PDF version of the map, which keeps the best quality, and a PNG version of it for web purposes:

ggsave("data/map.pdf")  
ggsave("data/map\_web.png", width = 10, height = 10, dpi = "screen")

Take a look at the [ggplot Cheatsheet](https://www.rstudio.com/resources/cheatsheets/) which shows the most important ggplot operations at a glance.

## 11.6 Interactive Maps

The [Leaflet library for R](https://cran.r-project.org/web/packages/leaflet/index.html) makes it easy to create interactive web maps. Leaflet is one of the most popular open-source JavaScript libraries used by a number of websites such as [The New York Times](https://www.nytimes.com/projects/elections/2013/nyc-primary/mayor/map.html), [Flickr](https://www.flickr.com/map) or [OpenStreetMap](https://www.openstreetmap.org/#map=14/53.7727/20.5526).

The first step in creating a leaflet map is to initialize an empty map widget:

library(leaflet)  
m <- leaflet()

The map widget can be supplemented with additional layers such as a basemap or clickable markers:

m %>%  
 addTiles() %>%  
 addMarkers(lng=174.768, lat=-36.852, popup="The birthplace of R")

You may have recognized that layers can be simply appended by means of the pipe operator (%>%). This is because most functions in the leaflet package have an argument “map” as their first argument.

The function addTiles() per default adds OpenStreetMap map tiles. You may use the function addProviderTiles() to add other map tiles. Leaflet supports a large number of [basemap layers](http://leaflet-extras.github.io/leaflet-providers/preview/).

The same pipe-syntax can be used to add [Markers](https://rstudio.github.io/leaflet/markers.html) and HTML [Labels](https://rstudio.github.io/leaflet/popups.html) or [Popups](https://rstudio.github.io/leaflet/popups.html). In the following example, an HTML Popup locates a restaurant:

library(leaflet)  
  
content <- paste(sep = "<br/>",  
 "<b><a href='https://www.techno-z.at/standort-und-service/gastronomie/'>Bistro im Techno\_Z</a></b>",  
 "Schillerstrasse 30",  
 "5020 Salzburg",  
 "This is where I had lunch today!"  
)  
  
leaflet() %>%  
 setView(lng = 13.040030, lat = 47.823112, zoom = 18) %>%  
 addProviderTiles("OpenStreetMap.Mapnik") %>%  
 addPopups(13.040030, 47.823112, content,  
 options = popupOptions(closeButton = TRUE))

Moreover, leaflet offers numerous methods and functions for [manipulating the map widget](https://rstudio.github.io/leaflet/map_widget.html) and integrating [lines and shapes](https://rstudio.github.io/leaflet/shapes.html), [GeoJSON](https://rstudio.github.io/leaflet/json.html) and [Raster Images](https://rstudio.github.io/leaflet/raster.html). To get more information on creating interactive maps with R and leaflet, turn to the [Documentation](https://rstudio.github.io/leaflet/).

# 12 R Markdown

This lesson is dedicated to the library rmarkdown. In fact, however, R Markdown is **more than only a library**. [RMarkdown](https://rmarkdown.rstudio.com/) belongs to a set of tools that are designed to improve the reproducibility of your work. Other tools and platforms such as [GitHub](https://github.com/), [Jupyter](https://jupyter.org/), [Docker](https://www.docker.com/) or preprint servers such as [ArXiv](https://arxiv.org/) or [bioRxiv](https://www.biorxiv.org/) facilitate reproducibility in many different ways. In this module, we will not cover the paradigm of [reproducible research](https://esajournals.onlinelibrary.wiley.com/doi/full/10.1002/bes2.1801) in detail. Instead, we will focus on those aspects that will helping you **make your analyses and reports** more appealing, interactive and **efficient** through the use of RMarkdown.

In this lesson, we will weave together code and text in professionally rendered R Markdown documents. And we will use [GitHub](https://github.com/) to safely store, share and administer our results.

## 12.1 Set up your work environment

Before you will create your first R Markdown document, we have to **setup** the **GitHub** environment. GitHub was originally founded as a platform for software developers. The architecture of GitHub is designed to manage changes that are made to computer software in the process of software development. The same architecture, however, can also be used to control the versioning of documents or any other collection of information.

Version control is particularly important when working in teams to sync working steps among project participants. However, even when working individually, GitHub is a trustworthy and open online data dump that tracks changes and simplifies the documentation and sharing of your work and others.

To setup your personal GitHub environment, follow these steps:

1. Read through the [Hello-World-Section](https://docs.github.com/en/get-started/quickstart/hello-world) of GitHub’s Quickstart Documentation. For now, reading is fine. You do not necessarily have to work through the tutorial.
2. [Create a GitHub account](https://github.com/).
3. [Download and install Git](https://git-scm.com/downloads). [Git](https://docs.github.com/en/get-started/quickstart/set-up-git) it is a distributed VCS (version control system), in which the codebase aswell as it’s full history is mirrored on every computer. GitHub therefore, is a web-based interface that uses Git very well. If you are interested in its theory and practice, this [video](https://www.youtube.com/watch?v=uR6G2v_WsRA) gives a very nice explanation of core concepts in Git.
4. Open R Studio -> Tools -> Global Options -> Git / SVN: check “enable version control” and set the path to the git.exe (e.g. C:/Program Files/Git/bin/git.exe). Restart RStudio.
5. [Create a repository](https://docs.github.com/en/get-started/quickstart/create-a-repo) on GitHub. In the tutorial, Skip section ‘Commit your first changes’.
6. By default, your repository has one branch named main. Create an additional branch off the main and call this new branch dev. To do so, follow the instructions in the [Hello-World-Tutorial](https://docs.github.com/en/get-started/quickstart/hello-world).

As next step, you will have to install the rmarkdown library as well as tinytex in RStudio as described in the [R Markdown Guide](https://bookdown.org/yihui/rmarkdown/installation.html).

In order to correctly **install**tinytex, make sure to execute both lines in RStudio:

install.packages('tinytex')  
tinytex::install\_tinytex() # install TinyTeX

If R Studio prompts you to install any dependencies, follow the instructions and install the necessary libraries. If you encounter any technical issues, please turn to the discussion forum for help.

## 12.2 Create a local clone

To work on your repository locally, you will need to create a local clone of your online GitHub repository. Here’s how:

In RStudio: File -> New Project… -> Version Control -> Git.

Copy and paste the repository URL (you can find this by going to your online repository), browse to the local directory and click: Create Project (see Fig. 12.1).



Figure 12.1: Clone GitHub Repository

Once you have cloned the online repository, the file contents of the repository as well as a new tab called Git appears in RStudio (see Fig. 12.2).



Figure 12.2: New features in RStudio

Per default the repository contains three files:

1. [gitignore](https://git-scm.com/docs/gitignore)
2. An [RStudio Project File](https://support.rstudio.com/hc/en-us/articles/200526207-Using-RStudio-Projects) (.Rproj)
3. A [ReadMe File](https://docs.github.com/en/repositories/managing-your-repositorys-settings-and-features/customizing-your-repository/about-readmes) (.md = pure markdown)

[gitignore](https://git-scm.com/docs/gitignore) and the [RStudio Project File](https://support.rstudio.com/hc/en-us/articles/200526207-Using-RStudio-Projects) were created on project initialization, meaning that these files are new and not yet available in the online repository. Changes made to the original repository are listed in the Git tab (see Fig. 12.3).



Figure 12.3: Changes in Git tab

Before we further modify our repository, switch to the dev branch (see Fig. 12.4). At the moment dev branch and main branch **are identical**.



Figure 12.4: Switch branch

It his **highly recommended** to keep work in progress separate from the main branch by working in a separate developer branch. Later, branches may be unified by merging from dev into the main branch (see [Opening a pull request](https://docs.github.com/en/get-started/quickstart/hello-world#opening-a-pull-request)).

## 12.3 Create a first RMarkdown document

Now that the environment is set up, we can create a first simple R Markdown document.

In RStudio: File -> New File -> R Markdown….

Type in a title, keep the default settings and confirm with OK. As a result you get a minimal R Markdown sample file, with the file extension .Rmd.

As apparent from the sample file, R Markdown documents are composed of the three basic components **metadata**, **text** and **code** (see Fig. 12.5).



Figure 12.5: R Markdown sample file

The syntax for the metadata is [YAML](https://en.wikipedia.org/wiki/YAML). In our document, the metadata section specifies the title of the document, the output format, and the date of creation. Many other document properties can be specified in this section. [Here](https://cran.r-project.org/web/packages/ymlthis/vignettes/yaml-overview.html) you can find an **overview** of the basic YAML syntax.

You can also update the date automatically to reflect the last time the report was updated by adding this line to the metadata section: date: "```r format(Sys.time(), '%d %B, %Y')```". This will output the current date in a human-readable format based on your systems time zone.

After the metadata section an **R inline code block** starts and ends with three backticks (see Fig. 12.5). The three **parameters** in curly brackets identify the code as R code. The r specifies the programming language, which is the default for R. Alternatively, you can also use parameter {py} to insert Python code into your markdown document (see Fig. 12.6).



Figure 12.6: Python Example

The setup parameter specifies the name of the code block and (as we will see later) include=FALSE prevents the code and code results from being displayed in the compiled HTML output. Nevertheless, RMarkdown still runs the code in this block, which sets echo=TRUE as the default option for all code blocks in the RMarkdown document. This means that, by default, the code of all code blocks in the document will be displayed in the output file unless otherwise indicated.

To learn more about code block options you may turn to the [knitr documentation](https://yihui.org/knitr/options/).

The other code blocks in the RMarkdown sample file produce a summary output (see line 17-19) or create a simple scatterplot (see line 25-27). To see how the compiled HTML output looks like, click Knit (see Fig. 12.7).



Figure 12.7: Knit HTML Output

You can also use the dropdown arrow next to the **Knit** button to compile PDF or .docx outputs, or choose from the [other](https://rmarkdown.rstudio.com/lesson-9.html) supported formats.

When you **“knit”** an R Markdown document, the .Rmd file is processed by the knitr package, which executes the code chunks and creates a new markdown file containing the code and its output. This markdown file is then processed by pandoc, which creates the final output document in the specified format. This two-step process allows for a wide range of output options and enables you to easily create professional-quality documents from your R Markdown code.

## 12.4 Synchronize with GitHub

It’s good practice to **synchronize** changes made to the project **regularly** with the online repository. First, you need to pull any changes that someone else may have made.

Click the Pull button in the Git tab (see Fig. 12.8). A message appears indicating that no other changes have been made (Already up to date).



Figure 12.8: Make Pull

Even if you’re working on your own, it’s a good idea to routinely start the sync process with a Pull.

Before you can push your own changes to the online repository, you have to Commit changes. Committing is like in-process saving. The Commit takes a snapshot of your changes and combines it with a user-defined commit message.

Save all documents in R Studio, then click the “Commit” button in the Git tab. The files that have been affected by changes will be listed in the commit window. You can click on one of the files to see the changes (green represents new content, red represents deleted content).

Manually select all the files by clicking the corresponding checkboxes. To make sure you have not missed a file, you may execute git add -A in the terminal window to add all files to the commit (see [this](https://github.com/joshnh/Git-Commands) list of popular Git Commands.) Once this is done, enter a meaningful commit message that describes your revision and click the Commit button.

See Fig. 12.9



To finish the sync process, click the Push button in the Git tab (see Fig. 12.10).

Figure 12.10: Make Push

Your online repository on GitHub should now be updated (switch to dev branch in your repository) (see Fig. 12.11).



Figure 12.11: Commit with message ‘describe sync process in github’ was pushed to the developer branch a minute ago

## 12.5 Basic RMarkdown syntax

In R Markdown documents, you can use double asterisks (\*\*Text\*\*) to **bold** words and phrases, or single asterisks (\*Text\*) to *italicize* them.

You can also create **headings** using hash signs (#). The number of hash signs you use corresponds to the heading level:

# Heading level 1  
  
## Heading level 2  
  
### Heading level 3

**Tables** are created by using the symbols | and -. Do you remember in the first chapter, the [numeric operators table](#numop)? Figure 12.12 illustrates the RMarkdown syntax being used to create this table.



Figure 12.12: How Tables are made in Markdown

To create an **ordered list**, use numbers followed by a period. The first item should start with the number 1:

Code - Ordered List:  
1. item 1  
4. item 2  
3. Item 3  
 + Item 3a  
 + Item 3b

1. Item 1
2. Item 2
3. Item 3
   * Item 3a
   * Item 3b

To create an **unordered list**, use \*, -, or +:

Code - Unordered List:  
\* item 1  
\* item 2  
 \* Item 3.1  
 - Item 3.2

* Item 1
* Item 2
  + Item 2a
  + Item 2b

The following syntax example illustrate how to create **hyperlinks** in RMarkdown documents:

[Github](https://github.com/){target="\_blank"}

Result looks like… [Github](https://github.com/)

By setting parameter target="\_blank, the link is opened in a new browser tab.

**Blockquotes** are written after a greater-than sign **(>)** and can be indented by adding a second greater-than sign **(>>)** like…

>"Everything is related to everything else, but near things are more related than distant things".  
>  
>>The phenomenon external to an area of interest affects what goes on inside.

The first law of geography is: “Everything is related to everything else, but near things are more related than distant things”

The phenomenon external to an area of interest affects what goes on inside.

Meanwhile, you know a number of characters that have a special meaning in RMarkdown syntax (like # or >). If you want these characters verbatim, you have to escape them. The way to **escape a special character** is to add a backslash before. For instance, \# will not translate into a heading, but will return #.

RMarkdown supports a large number of **mathematical notations**, which are surrounded by dollar signs $.

**Math. notation example 1:**

$x = y$

Result looks like…

**Math. notation example 2:**

$\frac{\partial f}{\partial x}$

Result looks like…

See [“Mathematics in R Markdown”](https://rpruim.github.io/s341/S19/from-class/MathinRmd.html) for more examples.

### 12.5.1 References in RMarkdown

R Markdown also comes with a time-saving method to insert **citations** and to build a **bibliography**. References are collected in a .bib-file. Create a new document in a text editor such as Windows Editor. Save the file with a **.bib** extension (e.g. references.bib) in your RStudio project folder.

You can use the RStudio project that you have cloned, modified and synchronized in previous exercises.

Now you can **add references** to the .bib-file. References are encoded in BibTeX format. The easiest way to get the BibTeX description of a reference is to export from Google Scholar. To enable BibTeX export, change setting in Google Scholar (see Fig. 12.13).



Figure 12.13: Enable BibTeX in Firefox 106.0.1

You may use a different browser or version of Firefox. In case you need support, please turn to the discussion forum!

Once you have enabled BibTeX export in your browser, a new link Import into BibTeX appears in Google Scholar (see Fig. 12.14).



Figure 12.14: BibTeX Link in Firefox 106.0.1

Click the link in Google Scholar and copy & paste the BibTeX code into your .bib-file. As a next step, you have to specify the location of the .bib-file in the YAML metadata (YAML parameter bibliography:<.bib file>). Now you can use the BibTeX reference sources from .bib in your R Markdown file. Insert @ and the first BibTeX parameter of a BibTeX reference source to add citations to your document (see Fig. 12.15).



Figure 12.15: Integrate BibTeX reference in RMarkdown document

To compile your R Markdown document, knit as HTML, PDF or Word. The rmarkdown library renders references as indirect citations (*reference without squared brackets in .Rmd file*) and direct citations (*reference with squared brackets in .Rmd file*) and inserts a bibliography (see Fig. 12.16).



Figure 12.16: Knit R Markdown as PDF

[Here](data/ReferencesInRMarkdown.zip) you can find a worked example for download. Unzip the folder and open the .Rproj file in RStudio.

Other helpful R Markdown syntax examples can be looked up in the [RMarkdown Cheatsheet](https://www.rstudio.com/wp-content/uploads/2015/02/rmarkdown-cheatsheet.pdf).

## 12.6 Speed up your workflows

Many of the methods that were covered in this module **are intended to make repetitive workflows more efficient** and less time-consuming. In this section, an example is introduced that illustrates the great potentials of R Markdown in terms of automation and efficiency improvement.

Imagine you have a client who is interested in a specific set of spatial economic indicators that are updated daily. Rather than creating a data report from scratch every day, R Markdown allows you to create a data report with charts that are automatically generated on report compilation. This can save you a significant amount of time and effort, and allow you to focus on other tasks.

Data can be retrieved in real-time by means of [Alpha Ventage](https://www.alphavantage.co/). Alpha Ventage provides financial market data through the [Alpha Ventage Rest API](https://www.alphavantage.co/documentation/). To access the API in R, we can use the R library [alphaventager](https://cran.r-project.org/web/packages/alphavantager/index.html).

In this exercise you are provided with a first draft version of the finance data report. [Download](data/RMarkdownFinanceData.zip) the draft version. Unzip the folder and open the .Rproj file in RStudio.

The project includes a .bib file that contains a BibTeX reference, a .csv file (see folder data) with more than 400 country names, national currencies and currency codes and an .Rmd file with inline R code that renders real-time currency exchange in a map.

Carefully read through the .Rmd file **before you compile an HTML output**.

Important: The .Rmd file includes an interactive leaflet map, i.e. other outputs than HTML are not supported.

Once you have understood the structure of this document, try to supplement the finance data report with an additional spatial indicator (e.g. map of exchange rates from national currencies to Euro).

## 12.7 Self-study

The functionality of R Markdown is comprehensive and **cannot be covered** in one lesson. It is highly recommended to consult the online book [R Markdown: The Definitive Guide](https://bookdown.org/yihui/rmarkdown/) to leverage the full potential of R Markdown.

Other R Markdown compile formats that are described in this book are Notebooks and Presentations. RMarkdown also supports other languages such as Python, C++ and SQL. It can also be customized and used to create more complex documents with extensions like: [BookDown](https://bookdown.org/) or [ThesisDown](https://github.com/ismayc/thesisdown). It is **recommended to practice and experiment** with these different features to become proficient in using R Markdown.

By the way, this module is written in BookDown.