Automated Data Processing with R

Christian Neuwirth & Maximilian Elixhauser

18 Juli, 2024

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. We welcome everybody to explore the contents and to work through the exercises. A *certificate of completion* and *online support by a tutor* can exclusively be claimed by those who are signed up for this UNIGIS module.

**Important Note for Learners:**

To fully engage with the exercises and assignments in this module, it’s essential to obtain an API key from Open Weather. The API key is required for accessing weather data, a crucial component of our practical exercises, particularly in Lesson 8. You can register and obtain your API key [here](https://openweathermap.org/appid). Please ensure you have this key ready before starting the relevant sections. This is especially important for UNIGIS students who will need it for the module assignments.

For more information, please get in contact with the [UNIGIS Office](https://unigis.at/en/contact/) at the University of Salzburg.

# 1 Introduction to R

## 1.1 About this module

This module will equip you with fundamental R programming skills, beginning with core programming concepts common to most languages. These include **Datatypes**, **Operators**, **Variables**, **Functions**, **Control Structures**, and **Libraries**. You will gain a solid understanding of these basics, forming a strong foundation for advanced programming.

We then progress to more complex data types like **Data Frames** and **Tibbles**, and explore how to **Read and Write** both spatial and non-spatial datasets. Special emphasis is placed on techniques to manipulate data, enabling you to adeptly manage and analyze datasets. This will be particularly useful for preparing and refining data for in-depth analysis.

The module also focuses on **data visualization**, where you’ll learn to create informative and compelling visual representations, such as box plots, scatterplots, line plots, and maps. These skills are critical for data exploration and presenting your findings in an accessible manner.

Additionally, the course introduces the essentials of working with **spatial data**. This includes handling **spatial data structures**, performing **spatial data manipulation**, and understanding **spatial relationships**.

Upon completing this module, you’ll possess foundational R programming skills, preparing you for more advanced topics such as “Geospatial Data Analysis”, as covered in the “Spatial Statistics” module of the MSc program.

This module partly draws from [granolarr](https://github.com/sdesabbata/granolarr), developed by [Stefano de Sabbata](https://sdesabbata.github.io/) at [University of Leicester](https://le.ac.uk/). For further exploration, refer to the Webbook [R for Geographic Data Science](https://sdesabbata.github.io/r-for-geographic-data-science/). We particularly recommend its chapters on **Statistical Analysis** and **Machine Learning** for those interested in advanced R applications.

## 1.2 R programming language

R is versatile in data science and analytics, with applications including:

* Data wrangling
* Statistical analysis
* Machine learning
* Data visualization and mapping
* Spatial data processing
* Geographic information analysis

Why R stands out:

* Free and open-source
* Offers extensive functionality surpassing most proprietary tools
* Available across Windows, MacOS, and Linux
* Primarily a domain-specific language with a focus on statistics and data analysis, it’s also versatile enough for general-purpose programming, making it ideal for automating analyses and creating custom functions.
* Large, supportive community, facilitating problem-solving and knowledge sharing

R, a **high-level programming** or **scripting language**, relies on an **interpreter** instead of a compiler. This interpreter directly executes written instructions, requiring adherence to the programming language’s grammar or **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. Install the latest version, choosing ‘base’ and the appropriate bit-version for your OS.

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

This video offers a concise RStudio overview:



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

Encounter technical difficulties? Please consult the discussion forum!

## 1.4 Interpreting Values

With RStudio and R installed, let’s dive into coding. The **Console Window** in RStudio (Fig. 1.2) is where the interpreter outputs results based on your input.



Figure 1.2: Console Window in RStudio

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

What happens when you input a text value (e.g., ‘test’)?

**See solution!**

The interpreter returns an error when unquoted text is entered, as it’s not recognized as a string. In R, text or strings must be enclosed in quotes (either single 'test' or double "test") to be understood as character data. Text is commonly referred to as **String** or **String of Characters**.

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

R’s simple data types, essential for encoding information, include:

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

These TRUE or FALSE values are typically the result of evaluating logical expressions. 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 |
| %% | Modulo | 5 %% 2 | 1 |
| ^ | Power | 5^2 | 25 |

Whereas mathematical operators are self-explanatory, the operators **Modulo** 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 ‘Modulo’ operator.

**See solution!**

The “Modulo” 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 |
| --- | --- | --- | --- |
| == | Equality | 5 == 2 | FALSE |
| != | Inequality | 5 != 2 | TRUE |
| > (>=) | Greater (or equal) | 5 > 2 | TRUE |
| < (<=) | Less (or equal) | 5 <= 2 | FALSE |
| ! | Negation | !TRUE | FALSE |
| & | Logical AND | TRUE & FALSE | FALSE |
| | | Logical 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 these expressions in the RStudio console to validate your assumptions.

## 1.8 References

Apart from [Stefano de Sabbata’s](https://sdesabbata.github.io/) teaching materials, this module draws from various sources, most of which are 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.* [See 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/).
* *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/).
* *Discovering Statistics Using R* by Andy Field, Jeremy Miles and Zoë Field, SAGE Publications Ltd, 2012. [See Book Webpage](https://us.sagepub.com/en-us/nam/discovering-statistics-using-r/book236067).
* *The RStudio Cheatsheets - A collection of cheatsheets for various R functions and packages.* [View Collection on RStudio Website](https://www.rstudio.com/resources/cheatsheets/).
* *The terra package - A comprehensive guide to the ‘terra’ package for spatial analysis in R.* [View Online Documentation](https://rspatial.org/pkg/index.html)
* *Sf - Simple Features for R - Documentation on using the ‘sf’ package for handling spatial data in R.* [View Online Documentation](https://r-spatial.github.io/sf/)

# 2 Core Concepts

In this lesson, we’ll explore three fundamental concepts in programming:

1. Variables
2. Functions
3. Libraries

## 2.1 Variables

Variables are essential for **storing data**. To define a variable, use an *identifier* (e.g., a\_variable) on the left of an *assignment operator* <-, followed by the value you wish to assign (e.g., 1):

a\_variable <- 1

To retrieve the value of the variable, simply use its **identifier**:

a\_variable

## [1] 1

To save and manage your code efficiently, create an **R Script** in RStudio (File > New File > R Script). Select the code in the R Script Window and click ‘Run’ to execute it. Scripts offer the advantage of running multiple lines of code at once and keeping a record of your work.

Variables allow you to store computational results and later access them for further analysis. For instance:

a\_variable <- 1  
a\_variable <- a\_variable + 10  
another\_variable <- a\_variable

Why use variables instead of direct input? They make your code **reusable, scalable, and time-efficient**, especially with complex data analyses or larger datasets.

**Let us consider the following example:**

Meteorologists track water temperature gradients to forecast weather patterns. If location A’s temperature is 22°C and B’s is 26°C, calculating the difference as 26 - 22 in the RStudio console is straightforward. However, for ongoing real-time measurements, an algorithm using variables for each location’s temperature can significantly speed up the process. Such an algorithm is adaptable to various data inputs, making it a robust tool for complex calculations.

1. In RStudio, create a new R script (File > New File > R Script).
2. Declare two variables (temp\_A and temp\_B) with temperature values.
3. Declare a third variable (diff) to store their difference.
4. Run your script and observe the changes in the Environment Panel.

**See solution!**

temp\_A <- 24 temp\_B <- 28 diff <- temp\_A - temp\_B

After executing the code in RStudio, you should notice changes in the Environment Panel, located in the top right corner. The **Environment Panel** will display three memory slots with identifiers diff, temp\_A, and temp\_B, having values of -4, 24, and 28, respectively. Invoking the name of an identifier in the code (e.g., typing diff and running it) will return the value stored in that memory slot.

To clear your workspace memory, click the “broom icon” in the Environment Panel’s menu.

## 2.2 Algorithms and Functions

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

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

The definition of an algorithm (and thus a program) can consist of one or more functions. Functions are sets of instructions that perform a task, structuring code into functional, reusable units. Some functions receive values as inputs, while others return output values.

Programming languages typically provide pre-defined functions that implement common algorithms (e.g., finding the square root of a number or calculating a linear regression).

For example, the pre-defined function sqrt() calculates the square root of an input value. **Like all functions** in R, sqrt() is invoked by specifying the *function name* and *arguments* (input values) between parentheses:

sqrt(2)

## [1] 1.414214

Each input value corresponds to a parameter defined in the function.

**Important:** In programming, the terms ‘parameter’ and ‘argument’ are often used interchangeably, but there is a subtle distinction. A **parameter** is the variable listed inside the parentheses in the function declaration, while an **argument** is the actual value passed to the function. For example, in the function definition square(number), number is a parameter. When you call square(4), the value 4 is the argument passed to the number parameter. Understanding this difference helps in comprehending how functions receive and process information.

round() is another predefined function in R:

round(1.414214, digits = 2)

## [1] 1.41

Note that the name of the second parameter (digits) needs to be specified. The digits parameter indicates the number of digits to keep after the decimal point.

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

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

## [1] 1.414214

Here, the output value is stored in a memory slot with the identifier sqrt\_of\_two. We can use the identifier sqrt\_of\_two as an argument in other functions:

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 named sqrt\_of\_two. The second line rounds the value stored in sqrt\_of\_two to three decimal places.

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 within other functions. For instance, we can use sqrt() as the first argument in 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 is skipped.

While using functions as arguments within other functions (nested functions) can sometimes reduce readability, they are a common and powerful practice in R. This approach can make your code more concise and expressive. However, it’s essential to balance conciseness with clarity!

Moreover, to enhance the readability of R code, it is recommended to follow naming conventions for variables and functions:

* R is a **case-sensitive** language, meaning UPPER and lower case are treated as distinct.
  + a\_variable is not the same as a\_VARIABLE.
* Valid names can include:
  + Alphanumeric characters, . and \_.
* Names must start with a letter, not a number or a symbol.

## 2.3 Libraries

Related, reusable functions in R can be grouped and stored in **libraries**, also known as *packages*.

As of now, there are more than 10,000 R libraries available. These can be downloaded and installed using the install.packages() function. Once a library is installed, the library() function is used to make it accessible in a script.

Libraries can vary greatly in size and complexity. For example:

* base: This includes basic R functions, such as the sqrt function discussed earlier.
* sf: A package providing [simple feature access](https://en.wikipedia.org/wiki/Simple_Features).

The stringr library illustrates the use of libraries in R. It offers a consistent and well-defined set of functions for string manipulation. Assuming the library is already installed on your computer, you can load it as follows:

library(stringr)

If not yet installed, you can download and install it with:

install.packages('stringr') # Note: the function takes a string argument ('')

Alternatively, libraries (a.k.a. packages) can be installed through RStudio’s ‘Install Packages’ menu (Tools > Install Packages…). You can choose to install from CRAN or a package archive file. The majority of libraries are available through [CRAN - Comprehensive R Archive Network](https://cran.r-project.org/), a vast collection of R resources and libraries.

* When using install.packages(), the package name must be a string, hence the quotes around ‘stringr’.
* To explore the functions provided by a library, like stringr, you can use ls() to list them. This command shows you what’s available to use once a library is loaded into your R session.

Once installed and loaded, the library provides a new set of functions in your environment. For instance, str\_length in the stringr library returns the number of characters in a string:

str\_length("UNIGIS")

## [1] 6

The str\_detect() function returns TRUE if the first argument (a string) contains the second argument (a character string). Otherwise, it returns FALSE:

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

## [1] TRUE

str\_replace\_all replaces all instances of a specified character in a string with another character:

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

## [1] "UNXGXS"

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

ls("package:stringr")

## [1] "%>%" "boundary" "coll"   
## [4] "fixed" "fruit" "invert\_match"   
## [7] "regex" "sentences" "str\_c"   
## [10] "str\_conv" "str\_count" "str\_detect"   
## [13] "str\_dup" "str\_ends" "str\_equal"   
## [16] "str\_escape" "str\_extract" "str\_extract\_all"   
## [19] "str\_flatten" "str\_flatten\_comma" "str\_glue"   
## [22] "str\_glue\_data" "str\_interp" "str\_length"   
## [25] "str\_like" "str\_locate" "str\_locate\_all"   
## [28] "str\_match" "str\_match\_all" "str\_order"   
## [31] "str\_pad" "str\_rank" "str\_remove"   
## [34] "str\_remove\_all" "str\_replace" "str\_replace\_all"   
## [37] "str\_replace\_na" "str\_sort" "str\_split"   
## [40] "str\_split\_1" "str\_split\_fixed" "str\_split\_i"   
## [43] "str\_squish" "str\_starts" "str\_sub"   
## [46] "str\_sub\_all" "str\_sub<-" "str\_subset"   
## [49] "str\_to\_lower" "str\_to\_sentence" "str\_to\_title"   
## [52] "str\_to\_upper" "str\_trim" "str\_trunc"   
## [55] "str\_unique" "str\_view" "str\_view\_all"   
## [58] "str\_which" "str\_width" "str\_wrap"   
## [61] "word" "words"

# 3 Data Structures

In this lesson, we expand upon the simple data types (**numeric**, **character** and **logical**) discussed in Lesson [1](#intro) by introducing more complex data structures.

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 of the following simple types:

* Numeric
* Character
* Logical

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

Defining a **vector variable** is similar to declaring a simple type variable, but the vector is created using the function c(), which 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 it begins with the fifth element of the vector.

Other functions for creating vectors include seq() and rep():

# Create a vector of real numbers with an interval of 0.5 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

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

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

Numeric vectors can also be created using a simple syntax:

# Create a vector of integer numbers from 1 to 10  
a\_vector <- 1:10  
a\_vector

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

### 3.1.1 Vector Element Selection

You can access individual elements of a vector by specifying the **index** of the element between square brackets, following the vector’s identifier. Remember, in R, the **first element** of a vector has an **index of 1**. For example, to retrieve the third element of a vector named a\_vector:

a\_vector <- 3:8  
a\_vector[3] # Retrieves the third element

## [1] 5

To retrieve multiple elements, use a vector of indices:

a\_vector <- 3:8  
a\_vector[c(2, 4)] # Retrieves the second and fourth elements

## [1] 4 6

In this case, the values 4 and 6 are returned, corresponding to indices 2 and 4 in a\_vector. Note that the vector of indices (c(2, 4)) is created on the fly.

Try creating and selecting elements from a vector yourself. Follow these steps:

1. Create a vector named east\_midlands\_cities containing the cities: Derby, Leicester, Lincoln, Nottingham.
2. Select the last three cities and assign them to a new vector named selected\_cities.

**See solution!**

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

my\_indexes <- 2:4

selected\_cities <- east\_midlands\_cities[my\_indexes]

### 3.1.2 Using the range() Function with Vectors

The range() function in R, is used to find the minimum and maximum values within a vector. This can be particularly helpful when analyzing the spread of data in a vector. For example:

# Create a numeric vector  
numeric\_vector <- c(2, 8, 4, 16, 6)  
  
# Apply the range() function  
vector\_range <- range(numeric\_vector)  
vector\_range # Displays the minimum and maximum values

## [1] 2 16

### 3.1.3 Applying Functions to Vectors

In R, functions can be applied to vectors just like they are with individual variables. When a function is applied to a vector, it typically processes each element of the vector, resulting in a new vector of the same length as the input.

For instance, adding a value (like 10) to a numeric vector will add that value to each element of the vector:

numeric\_vector <- 1:5  
numeric\_vector <- numeric\_vector + 10 # Adds 10 to each element  
numeric\_vector

## [1] 11 12 13 14 15

Similarly, applying a function like sqrt() to a numeric vector will compute the square root of each element:

numeric\_vector <- 1:5  
numeric\_vector <- sqrt(numeric\_vector)  
numeric\_vector # Displays the square roots

## [1] 1.000000 1.414214 1.732051 2.000000 2.236068

A logical condition applied to a vector will return a logical vector indicating whether each element meets the condition:

numeric\_vector <- 1:5  
logical\_vector <- numeric\_vector >= 3  
logical\_vector # Shows TRUE or FALSE for each element

## [1] FALSE FALSE TRUE TRUE TRUE

Moreover, functions like any() and all() provide overall evaluations of a vector based on a condition. any() returns TRUE if any elements satisfy the condition, while all() returns TRUE only if all elements satisfy the condition:

numeric\_vector <- 1:5  
any(numeric\_vector >= 3) # Checks if any element is >= 3

## [1] TRUE

all(numeric\_vector >= 3) # Checks if all elements are >= 3

## [1] FALSE

Also, when creating vectors in R, it’s important to understand the concept of **type coercion**. R is designed to be user-friendly, and when you combine different data types in a vector (e.g., mixing numbers and characters), R will automatically convert all elements to the same type. This process is known as type coercion. For example, if you combine numeric and character data in a vector, all elements will become characters.

mixed\_vector <- c(1, "text", TRUE)  
print(mixed\_vector) # Notice how all elements are coerced to the same type

## [1] "1" "text" "TRUE"

**Factors** are a special data type in R, similar to vectors but limited to predefined values called **levels**. Factors are not covered in this module, but you can learn more about them in the [Programming with R](https://swcarpentry.github.io/r-novice-inflammation/12-supp-factors.html) tutorial.

## 3.2 Multi-dimensional Data Types

### 3.2.1 Matrices

Matrices in R are two-dimensional data structures, where data is organized in rows and columns. They are particularly useful for performing a variety of mathematical operations.

To create a matrix, use the matrix() function, providing a vector of values and the desired dimensions:

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

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

R supports numerous operators and functions for matrix algebra. For example, basic arithmetic operations can be performed on matrices:

x <- matrix(c(3, 5, 7, 4, 3, 1), nrow=3, ncol=2)  
y <- matrix(c(1, 2, 3, 4, 5, 6), nrow=3, ncol=2)  
z <- x \* y # Element-wise multiplication  
z

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

When working with matrices, range() can help you quickly identify the lowest and highest values within a particular row or column. However, it’s not typically used for selecting rows or columns. Instead, you’d use direct indexing or other functions for selection. Here’s how to correctly utilize range() with matrices::

# Creating a matrix with numeric values  
matrix\_data <- matrix(1:9, nrow=3)  
  
# Finding the range of values in the first column  
first\_column\_range <- range(matrix\_data[,1])  
print(first\_column\_range) # Displays the minimum and maximum values of the first column

## [1] 1 3

In the context of matrix selection, while range() is not used for selecting specific rows or columns, understanding the spread of data within a matrix can be crucial for informed data manipulation and analysis. Here’s an example of how you might use this information:

# Assuming you want to know if the first column contains values within a specific range  
is\_in\_range <- first\_column\_range[1] >= 2 && first\_column\_range[2] <= 8  
print(is\_in\_range) # Checks if the range of the first column is between 2 and 8

## [1] FALSE

Or you can exclude specific columns or rows from a matrix using negative indexing. This is particularly useful for analysis or visualization when you want to focus on specific parts of the matrix:

# Creating a matrix  
matrix\_data <- matrix(1:9, nrow=3)  
  
# Excluding the first column from the matrix  
matrix\_without\_first\_column <- matrix\_data[, -1] # Excludes the first column  
print(matrix\_without\_first\_column)

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

For a detailed overview of matrix operations, refer to [Quick-R](https://www.statmethods.net/advstats/matrix.html).

### 3.2.2 Arrays

Arrays in R are like higher-dimensional matrices, capable of storing data in multiple dimensions. Creating an array requires specifying the values and the dimensions for each axis:

a3dim\_array <- array(1:24, dim=c(4, 3, 2)) # Creates a 3-dimensional array  
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: An array can have a single dimension, resembling a vector. However, arrays have additional attributes like dim and offer different functionalities.

### 3.2.3 Selection in Multi-Dimensional Data Types

Selecting elements from matrices and arrays in R is similar to vector selection, but requires specifying an index for each dimension.

For matrices:

# Example matrix  
a\_matrix <- matrix(c(3, 5, 7, 4, 3, 1), nrow=3, ncol=2)  
a\_matrix

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

# Selecting the second row, first and second columns  
a\_matrix[2, c(1, 2)]

## [1] 5 3

For arrays with multiple dimensions:

# Example 3-dimensional array  
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

# Selecting elements with specific indices  
an\_array[2, c(1, 2), 2]

## [1] 8 11

Create a 3-dimensional array, extract 2 elements to form a vector, and 4 elements to form a matrix.

**See solution!**

Array creation:

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

Extracting elements:

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

### 3.2.4 Lists

Lists in R are incredibly versatile and can hold elements of different types, including vectors, matrices, other lists, and even functions. This makes lists a powerful tool for organizing and storing complex, heterogeneous collections of data.

Elements in lists are selected using double square brackets.

Basic list:

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

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

# Selecting the first element  
employee[[1]]

## [1] "Christian"

Named lists allow selection using the $ symbol:

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

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

# Selecting by name  
employee$employee\_name

## [1] "Christian"

Applying range() to lists requires consideration of the list’s diverse elements. For numeric elements, range() can be applied either individually or to the entire list converted into a numeric vector:

# List with numeric and character vectors  
list\_data <- list(num\_vector = 1:5, char\_vector = c("a", "b", "c"))  
# Applying range() to numeric elements  
numeric\_ranges <- lapply(list\_data, function(x) if(is.numeric(x)) range(x))  
numeric\_ranges

## $num\_vector  
## [1] 1 5  
##   
## $char\_vector  
## NULL

### 3.2.5 Data Frame

Data frames are essential in R for representing tables of data. Each data frame is structured similarly to a named list with each element being a vector of equal length. Below is an example of creating a data frame:

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

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

Data frames are similar to tables in that each column represents a variable, and each row represents an observation.

Can elements of different types be mixed within a single vector or data frame column?

**See solution!**

Vector elements (and by extension, data frame columns) must be of the same type (character, logical, or numeric). For example, EmployeeName contains characters, while Age contains numerics.

Elements in each column of a data frame correspond to a row. The first element in EmployeeName represents the name of the first employee, and similarly for other columns.

To rename columns, use the ‘names()’ function: **names(data frame)[column index] = “new name”**

Selecting data from a data frame is analogous to vector and list selection, but with a focus on the data frame’s two-dimensional structure. You typically need two indices to extract data.

Example of selecting the first element in the first column:

employees[1, 1]

## [1] "Maria"

Selecting whole rows:

employees[1, ]

## EmployeeName Age Role  
## 1 Maria 47 Professor

Selecting whole columns:

employees[, 1]

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

Columns can also be selected using dollar signs and column names:

employees$Age

## [1] 47 34 32

employees$Age[1] # Selecting the first element in the 'Age' column

## [1] 47

Modifying a data frame:

* Changing an element (e.g., updating Pete’s age):

employees$Age[2] <- 33

* Adding a new column:

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

Perform operations on data frame columns as you would on vectors. Create a variable to represent the current year and use it to calculate and add a new column for each employee’s year of birth.

**See solution!**

Creating a data frame employees:

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

Calculating the current year:

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

Calculating year of birth:

employeesAge employees

# 4 Spatial Data Structures

In the realm of geoinformatics, spatial data is a cornerstone, offering a lens through which we can view, analyze, and interpret the world around us. At this point in your studies, 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 multidimensional 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 the [sf](https://r-spatial.github.io/sf/) package.
2. Raster data structures as provided by the [terra](https://rspatial.org/pkg/) package.

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 will use the sf library to work with vector data structures. The name sf (which stands for **simple features**) 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 setups, for example, importing from and exporting to spatial databases.

Simple features have spatial, geometric attributes as well as non-spatial attributes. The most common geometry types are points, lines, and polygons and their respective “multi” versions (see [Simple feature geometry types](https://r-spatial.github.io/sf/articles/sf1.html#simple-feature-geometry-types)).

**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-objects 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 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), (...

To extract or set CRS information, use the st\_crs function:

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

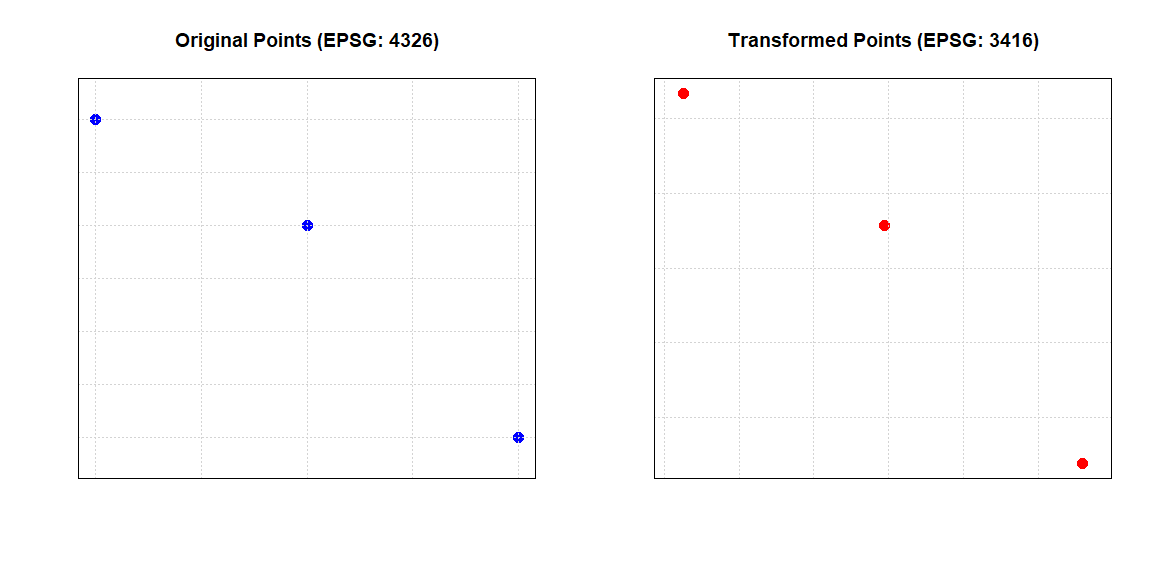
EPSG-Codes of other reference systems can be found [here](https://spatialreference.org/ref/epsg/?search=4326&srtext=Search).

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

# Assign Austrian Lambert Projection 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:

# Plotting the original and transformed points side by side  
par(mfrow = c(1, 2)) # Arrange plots in 1 row, 2 columns  
  
# Plot original points with a coordinate grid and axes  
plot(sf::st\_geometry(point\_sf\_multi), main = "Original Points (EPSG: 4326)", pch = 19, col = "blue", cex = 1.5)  
grid()  
box()  
  
# Plot transformed points  
plot(sf::st\_geometry(point\_sf\_multi\_transform), main = "Transformed Points (EPSG: 3416)", pch = 19, col = "red", cex = 1.5)  
grid()  
box()



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 the 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 lesson 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 the 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 (CRS84) (OGC:CRS84)

By 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 32N 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)

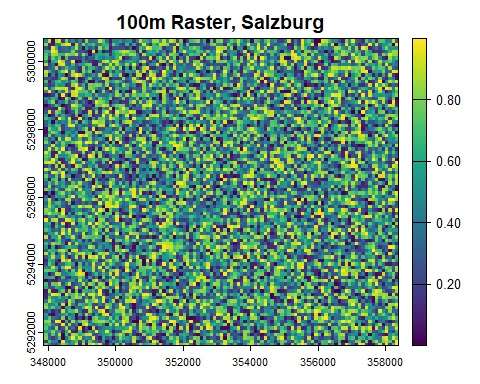
## [1] FALSE

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] 0.47483037 0.91352366 0.02084114 0.88092062 0.93635539

#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.

The syntax to select values from SpatRaster Objects is the same that we have used to select elements of matrix and array data structures:

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

Where is the origin of the grid? In which 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 [8](#readwrite).

# 5 Control Structures

In this lesson, you will learn about control structures, a significant elements in coding that allows for dynamic behavior based on variable values.

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 repeating one or more instructions multiple times. Loops are commonly used to apply the same operation to a series of values 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.

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") is 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 check 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.

What result do you expect when you remove the negative sign in the code block above (a\_value <- 7)? To evaluate your expectation, run the code in RStudio.

**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") is 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 the statements are enclosed within curly brackets for clarity. While indentation doesn’t affect code execution in R, it’s essential for readability and maintaining code structure.

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\n") prints 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 loop 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 (or ‘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 iteration number", i, ":\n")  
 cat(" See you later!\n")  
}

## This is iteration number 1 :  
## See you later!  
## This is iteration number 2 :  
## See you later!  
## This is iteration 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

Having explored 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.

### 5.5.1 Exercise: loops with conditional statements

As a last exercise in this lesson, you will implement code that iterates over a two-dimensional SpatRaster Object and counts values in the raster grid that exceed a certain threshold.

1. Create a SpatRaster Object r with 20 rows and 20 columns and assign random values in a range between 0 and 1. Use the function runif as random value generator. It is recommended to make use of code snippets from lesson Spatial Data Structures.
2. Define a variable named cnt and assign a value of 0 to it.
3. Iterate over the SpatRaster Object r by means of a nested for-loop:

for(row in c(1:nrow(r))) {   
 for(col in 1:ncol(r)) {   
   
 print(r[row, col])   
   
 }  
}

1. Add a conditional statement within the for-loops that increases the value of cnt by one, given that r[row, col] is > 0.5. Eventually, variable cnt will hold the number of raster grid values that exceed a threshold of 0.5.

**Task 1:**

Try to find out, in what order the nested-for-loop-structure iterates over the SpatRaster Object r.

**Task 2:**

r contains randomly generated values in a range between 0 and 1. Change the threshold value in your conditional statement a couple of times to investigate the distribution of random values. Does the algorithm in function runif draw values from a normal or from a uniform distribution?

**Solution!**

See [our code](Solution_Exercise_ForLoop.R).

**Answer 1:**

The *nested for-loop-structure* iterates through the SpatRaster Object in a **row-major order**. It starts its iteration in the leftmost cell of row 1, moves to the right through columns (e.g., col1, col2) until it reaches the last column in row 1. Then, it moves down to row 2 and repeats the process for each row in a similar manner. This row-major order ensures that it covers each element of the SpatRaster Object row by row.

**Answer 2:**

A threshold value of 0.9 returns a count of about 40, which indicates that about 10% of values are greater than 0.9. It seems like random values are uniformly distributed in a range between 0 and 1. Other R functions like [rnorm()](https://www.rdocumentation.org/packages/compositions/versions/2.0-6/topics/rnorm) generate numbers from a normal distribution.

As an alternative, the same functionality may be implemented without loop. Instead, operations can be vectorized:

library(terra)  
  
r <- terra::rast(ncol=20, nrow=20)  
terra::values(r) <- stats::runif(terra::ncell(r),0,1)  
  
# Get values of SpatRaster Object `r` as matrix  
rm <- terra::values(r)  
# Create logic vector by condition  
log\_vec <- rm[,1] > 0.5  
# Get length of TRUE values (grid value > 0.5) in logic vectors  
length(log\_vec[log\_vec== TRUE])

## [1] 215

It is apparent from the code example above that avoiding loops in R code makes code shorter and increases performance. Increase the size of your raster grid (e.g. ncol=100, nrow=100) and test respective code realizations to investigate the difference in code performance. More information on vectorization and parallelization of operations in R can be found [here](https://tmieno2.github.io/R-as-GIS-for-Economists/repetitive-processes-and-looping.html#do-you-really-need-to-loop).

# 6 Functions

In previous lessons, we’ve used various functions (like cat(), print()) without delving deeply into their mechanics. This lesson aims to demystify functions by guiding you through the creation and understanding of custom functions. We’ll also explore the concepts of global and local variable scopes.

## 6.1 Defining Functions

Defining a function in R follows a syntax similar to variable assignments or conditional statements. We start with an identifier (e.g., add\_one) on the left of an assignment operator (<-).

The body of the function follows, beginning with the keyword function, the **parameter(s)** enclosed in parentheses (e.g., input\_value), and the function’s body within curly braces. The value of the last executed statement is automatically returned:

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

add\_one(3)

## [1] 4

Call this function by specifying its **identifier** and the necessary parameter(s). For instance, in our example add\_one(3) returns 4.

### 6.1.1 Understanding the return() Function

In R, functions automatically return the value of the last expression evaluated. The decision to use the return() function explicitly or rely on implicit return is often guided by the specific context of your work, especially in fields like GIS.

**Implicit Return:**

In R, if return() is not explicitly used, the function returns the result of the last line of code executed. This is often sufficient, especially in simpler functions. For example:

add\_one <- function (input\_value) {  
 input\_value + 1  
}  
add\_one(3) # Returns 4

## [1] 4

**Explicit Return:**

Using return(), you can explicitly specify what the function should return. This approach is particularly useful when the function has complex logic, or when you need to return a value before reaching the end of the function. For example:

add\_one <- function (input\_value) {  
 result <- input\_value + 1  
 return(result)  
}  
add\_one(3) # Also returns 4

## [1] 4

In both examples, calling add\_one(3) returns 4. The explicit use of return() in the second example provides clarity about the intended output of the function.

The choice between implicit and explicit return in R functions can depend on various factors, including the complexity of the function and the coding standards of your team or organization. In GIS and similar fields, it’s important to understand both approaches and adapt to the coding practices of your workplace. Always consider the context and ask, “What is the standard here?”

**Early Return Example:**

Sometimes, it’s necessary to exit a function early, for instance in error handling:

safe\_divide <- function (numerator, denominator) {  
 if (denominator == 0) {  
 return("Error: Division by zero")  
 }  
 numerator / denominator  
}  
safe\_divide(10, 0) # Returns "Error: Division by zero"

## [1] "Error: Division by zero"

So, in GIS and data analysis, adapting to the specific requirements of your project and following established coding standards are key to effective and collaborative work.

## 6.2 More Parameters

Functions can have multiple parameters, separated by commas.

A function expects as many input values as there are parameters specified in its definition; otherwise, an error is triggered.

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

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

## [1] 6

Default parameters can enhance a function’s flexibility. For instance, you can modify the area\_rectangle function parameters as (height, width = 3). Now, with only one input, the function should return YOUR INPUT \* 3. Specifying both parameters overrides the default width. Here’s how it looks:

area\_rectangle <- function (height, width = 3) {  
 return(height \* width)  
}  
  
# Calling with one parameter uses the default width of 3  
area\_rectangle(3) # Returns 9

## [1] 9

# Calling with two parameters uses the specified width  
area\_rectangle(3, 2) # Returns 6

## [1] 6

This approach allows the function to be more flexible and adaptable to different use cases.

## 6.3 Returning Multiple Values

To return multiple values from a function, encapsulate them in a list. For example, rectangle\_metrics calculates and returns both the area and perimeter of a rectangle:

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

Retrieve these values using list indices [[1]] and [[2]], or by simply using their names:

metrics <- rectangle\_metrics(3, 2)  
  
# Using list indices  
cat("Area (list index): ", metrics[[1]], "\n")

## Area (list index): 6

cat("Perimeter (list index): ", metrics[[2]], "\n")

## Perimeter (list index): 10

# Using names  
cat("Area (name): ", metrics$area, "\n")

## Area (name): 6

cat("Perimeter (name): ", metrics$perimeter, "\n")

## Perimeter (name): 10

Upon defining a function in R, it appears in the Environment Window of RStudio, similar to a variable. When invoked, the R interpreter executes the function stored in memory.

## 6.4 Functions and Control Structures

Functions can contain loops and conditional statements. Here’s a function that uses a loop to calculate the factorial of a 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  
 }  
 return(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 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.

Though technically possible, defining a function within loops or conditional statements is generally avoided.

## 6.5 Scope

Function parameters are internal variables acting as a bridge between the function and its environment. They are ‘visible’ only within the function’s scope.

The scope of a variable refers to the code region where the variable is accessible.

Understanding scope is crucial for several reasons:

* **Avoiding Variable Conflicts:** Variables with the same name can exist in different scopes without conflicting with each other. This prevents unexpected behavior caused by variable name overlaps.
* **Predictable Behavior:** Knowing whether a variable is global or local helps predict its behavior and influence within your script. It clarifies where and how variables can be accessed or modified.
* **Resource Management:** Local variables are often cleared from memory once the function execution is complete, aiding in efficient resource management in larger scripts or applications.

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

* **Global Variables:** Defined outside of any function and accessible anywhere in the script, including within functions.
* **Local Variables:** Defined within a function and accessible only in that function’s scope. This includes function parameters, which act as internal variables bridging the function and its environment.

In the following example, x\_value is a global variable, while new\_value and input\_value are local to the times\_x function. Accessing new\_value or input\_value outside times\_x would result in an error, but x\_value can be referred to within the function:

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

## [1] 20

Understanding these distinctions is essential for writing robust and error-free R code, especially in more complex data analysis or GIS projects.

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) { return(input\_value \* x\_value) }

These lessons have introduced fundamental R programming concepts. The [Base R Cheatsheet](https://github.com/rstudio/cheatsheets/blob/main/base-r.pdf) offers a concise summary of key operations.

# 7 Data Manipulation

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

In this lesson, you will learn about the fundamental data wrangling operations using the dplyr library from the Tidyverse suite. You’ll also be introduced to [tibbles](https://tibble.tidyverse.org/), a modern take on data frames in R. Tibbles are lightweight and work seamlessly within the Tidyverse ecosystem. For more on tibbles, see [Tibbles in R for Data Science](https://r4ds.had.co.nz/tibbles.html).

## 7.1 Preparation

If not already installed on your machine, install both the tidyverse and nycflights13 libraries. (see “libraries” in lesson [2](#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 avoid ambiguities when functions from different loaded libraries have the same 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 dplyr library provides a number of functions to investigate the 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 this function. The returned row count n is represented in a table.

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

| n |
| --- |
| 336776 |

In the example above, we use the pipe operator %>%, a key feature of magrittr and tidyverse syntax. The operator links a sequence of analysis steps, passing flights\_from\_nyc through count() and then kable(). This makes the code simpler and more readable. For a deeper understanding, check out [this video](https://www.youtube.com/watch?v=sohARFx6aTo).

The same result could be achieved with a traditional approach using an intermediate variable or by nesting functions. However, these methods can make the code less readable compared to using the pipe operator:

**Intermediate Variable:**

cnt <- dplyr::count(flights\_from\_nyc)  
knitr::kable(cnt)

**Nested Function:**

knitr::kable(dplyr::count(flights\_from\_nyc))

The result is the same. However, nested structures are harder to read. Accordingly, 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 |

Notice how the code becomes more readable with each operation on a new line after %>%. Remember, the pipe operator passes the left-hand object as the first argument to the right-hand function. To change the argument placement, use . within the function call.

For example, you could explicitly pass variable flights\_from\_nyc as a first argument by dplyr::count(., origin). Passing the variable as a second argument dplyr::count(origin, .) will return an error as the data frame is expected as a first input to function count.

### 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 mean() 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 |

The results show that the average distance covered by flights starting from JFK is significantly higher that flights departing from LGA or EWR.

### 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 with dplyr::transmute. What happens to your results?

**See solution!**

The transmute function creates a new data frame containing only column air\_time\_hours. The function drops other table columns.

### 7.2.4 Arrange

The function arrange() sorts a `tibble or data frame in ascending order based on 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)!

**Note on Encoding:** When working with datasets from various sources, you may encounter encoding errors. If such an issue arises, a simple approach is to specify the encoding in the read.csv() function. For instance, you can use read.csv(..., fileEncoding = "latin1"). However, the best case would be, to check the original encoding of the data beforehand and match it in your read.csv() command. Encoding can be tricky, and while it’s outside the scope of this exercise, being aware of it is crucial in data analysis.

## 7.3 Join

A join operation combines two tables into one by matching rows that have the same values in a 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 created below (data frame city\_telephone\_prexix) contains the telephone prefixes of three cities.

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 |

That information can be combined with the data present in a second table city\_info\_wide (see below) through the join operation on the columns containing the city names.

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 necessary because the kable() function requires tibbles as input.

Joins between two tables can be executed by different join operations. The dplyr library offers join operations, which correspond to SQL joins illustrated in the image below.



Figure 7.2: Join types

Please take your time to understand the examples below that show different realizations of joins between table city\_telephone\_prexix and table city\_info\_wide.

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 |

Compare the results of respective full, left, right and inner join operations. Turn to the discussion forum in case you need further explanations.

### 7.3.1 Exercise: join

In the previous exercise, we 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](https://ggplot2.tidyverse.org/reference/map_data.html) to convert the built in sample dataset world (comes with library [maps](https://cran.r-project.org/web/packages/maps/index.html)) 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.

[Our exercise solution](Solution_Exercise_DataWrangling2.R) creates a simple output map. Don’t worry in [Lesson 10](#data-viz) 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 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, spatial vector and raster file formats. Moreover, we will retrieve online data by means of a data API.

## 8.1 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 write\_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.

## 8.2 Read and write vector data

The library sf makes it easy to read and write vector datasets such as shapefiles.

Remember: In lesson [4](#spds), we used the sf library to build vector geometries from scratch. Investigate their properties and assign and transform coordinate reference systems. The name sf stands for **simple features**, meaning that [sf-objects conform to the simple feature standard](https://r-spatial.github.io/sf/articles/sf1.html). Now we use the same library to load sf-objects from file.

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 (attribute) of the sf-object.

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



As you have learned in lesson [4](#spds), sf represents features as records in data-frame-like structures with an additional geometry list-column. The example below renders three features (rows) of variable nc including the geometry column of type MULTIPOLYGON (see [Simple feature geometry types](https://r-spatial.github.io/sf/articles/sf1.html#simple-feature-geometry-types)) 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 [our solution](Solution_Miniexercise_plot.R)!

## 8.3 Read and write raster data

The terra library provides functions to read and write raster data. In lesson [4](#spds), we used the terra library to build SpatRaster Objects from scratch, investigated their properties and assigned and transformed coordinate reference systems.

It is more common, however, to create a SpatRaster Object from a file. Supported file formats for reading are GeoTIFF, ESRI, ENVI, and ERDAS. Most formats supported for reading can also be written to (see [Creating SpatRaster objects](https://rspatial.org/pkg/3-objects.html)).

The following code reads a GeoTIFF (download [venice-Sentinel.tiff 712KB](data/venice-Sentinel.tiff)) into memory and assigns it to a variable named r:

library(terra)  
  
r <- terra::rast("data/venice-Sentinel.tiff")

The function hasValues returns the logic value TRUE, which indicates that in-memory layers have cell values:

terra::hasValues(r)

## [1] TRUE

Executing the variable name r returns additional metadata information, which among others reveals that the image has 4 layers:

r

## class : SpatRaster   
## dimensions : 503, 692, 4 (nrow, ncol, nlyr)  
## resolution : 7.025638, 6.675899 (x, y)  
## extent : 288674.2, 293535.9, 5033073, 5036431 (xmin, xmax, ymin, ymax)  
## coord. ref. : WGS 84 / UTM zone 33N (EPSG:32633)   
## source : venice-Sentinel.tiff   
## names : venice-Sentinel\_1, venice-Sentinel\_2, venice-Sentinel\_3, venice-Sentinel\_4

Layers 1-3 correspond to color bands red, green, and blue. In order to plot the true color sentinel imagery, we can use terra function plotRGB and assign the numbers of respective color bands to attributes r (red), g (greeb), b (blue):

terra::plotRGB(r, r=1, g=2, b=3, main="Sentinel-2 image of Venice")



The following line converts the original CRS (which is WGS 84 / UTM zone 33N) to the Italian national CRS [Italy Zone 1:](https://spatialreference.org/ref/epsg/3003/)

r\_z1 <- terra::project(r, "+proj=tmerc +lat\_0=0 +lon\_0=9 +k=0.9996 +x\_0=1500000 +y\_0=0 +ellps=intl +units=m +no\_defs")

Eventually, the modified SpatRaster Object is written to file by means of terra function writeRaster:

terra::writeRaster(r\_z1, "data/venice\_zone1.tif", overwrite=TRUE)

## 8.4 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.

Figure 8.1: Video (3:13 min): REST API concepts and examples.

The most important takeaway 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 contains 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), an API key to get access, and parameters to filter the desired data subset (lat and lon).

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.

**Security Note:** In general, when working with APIs, it’s crucial to handle API keys securely. API keys are sensitive tokens that authenticate your requests to an API service. Exposing these keys publicly can lead to unauthorized access and potential misuse. Always keep your API keys confidential. Avoid hardcoding them directly into scripts, especially if the scripts are shared or stored in public repositories. Instead, consider using environment variables or secure vaults to store your API keys. For sharing scripts, you can instruct users to enter their own keys, ensuring that each user’s key remains private.

First we will have to create an URL request. The function sprintf() assembles the URL request composed of base URL, key and lat / lon:

**Note**: The sprintf() function in R is used for string formatting. It allows you to combine, format, and interpolate variables into strings in a flexible way. For example, sprintf("%s?lat=%s&lon=%s&appid=%s", base, lat, lon, apiKey) constructs a URL string by inserting the values of base, lat, lon, and apiKey into the placeholders (%s). Each %s is replaced by the corresponding variable’s value in the order they are listed. This function is particularly useful in scenarios where you need to dynamically generate strings or URLs with variable data.

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"

As a next step, the URL is passed to the request() function to create a request object, and req\_perform() is used to execute the HTTP Get method:

# Create the request and perform it  
req <- httr2::request(full\_url)  
resp <- httr2::req\_perform(req)

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():

# Check the status code of the response  
httr2::resp\_status(resp)

## [1] 200

# View the content structure of the response in JSON  
str(httr2::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 8  
## ..$ temp : num 302  
## ..$ feels\_like: num 309  
## ..$ temp\_min : num 300  
## ..$ temp\_max : num 304  
## ..$ pressure : int 1020  
## ..$ humidity : int 99  
## ..$ sea\_level : int 1020  
## ..$ grnd\_level: int 947  
## $ visibility: int 10000  
## $ wind :List of 3  
## ..$ speed: num 2.43  
## ..$ deg : int 7  
## ..$ gust : num 1.77  
## $ clouds :List of 1  
## ..$ all: int 30  
## $ dt : int 1721307666  
## $ sys :List of 5  
## ..$ type : int 2  
## ..$ id : int 2000815  
## ..$ country: chr "AT"  
## ..$ sunrise: int 1721273285  
## ..$ sunset : int 1721329185  
## $ 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:

library(knitr)  
  
# Using resp\_body\_json to get the JSON content of the response  
response\_content <- httr2::resp\_body\_json(resp)  
response\_df <- as.data.frame(response\_content)  
  
# Select columns 1:5 of response data frame   
# and render data frame as html table  
response\_df %>%  
 dplyr::select(1:5) %>%  
 knitr::kable(., format="html")

coord.lon

coord.lat

weather.id

weather.main

weather.description

13.0329

47.8077

802

Clouds

scattered clouds

For purposes of presentation, the returned data frame was shortened to show only columns 1 to 5. The returned values contain a short description of current weather conditions at location E: 13.03, N: 47.81.

Copy the code snippets above to a new R-Script and make an API request. **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.

# 9 Spatial Data Manipulation

In the previous lesson, you learned how to access data on a server via a REST API. The Open Geospatial Consortium (OGC) has adapted the REST API paradigm for geospatial applications. A variety of OGC API standards have been implemented to provide and utilize geospatial data on the web. An overview of the current implementation status can be found [here](https://ogcapi.ogc.org/#standards).

In this lesson, we will utilize online data resources in workflows that involve data cleaning, spatial queries, and analyses. Additionally, you will learn to carry out basic raster manipulation operations on a sample raster dataset.

## 9.1 Data Acquisition

We will work with vector data to correctly identify agricultural land parcels in European Union countries. The Austrian Agricultural Agency ([AMA](https://www.ama.at/intro)) provides access to Austrian agricultural parcels through an [OGC Rest API - Feature](https://ogcapi.ogc.org/) interface. The OGC API Features standard is the successor to the OGC Web Feature Service (WFS) specification.

The R syntax used to interact with OGC APIs is the same as described in the “Data API” section (see Lesson [8](#readwrite)).

Before loading the data into an R script, examine the API’s contents. Visit the web service’s [landing page](https://gis.lfrz.gv.at/ogcapi009501/ogc/features/) and proceed to the [collection page](https://gis.lfrz.gv.at/ogcapi009501/ogc/features/collections?f=text%2Fhtml). There you will find an overview of the available layers.

Enter the following URL into your browser:

https://gis.lfrz.gv.at/ogcapi009501/ogc/features/collections/ogcapi009501:INVEKOS\_feldstuecke\_aktuell\_polygon/items?f=json&limit=10

This request returns the first ten features of the ogcapi009501:INVEKOS\_feldstuecke\_aktuell\_polygon layer (agricultural land parcels as polygons) in GeoJSON format.

GeoJSON is a JSON-based standard for representing simple geographic features, along with their non-spatial attributes.

Upon inspecting the GeoJSON, you will find the coordinate vertices of the polygon features and attributes such as fs\_flaeche\_ha (parcel area in hectares) and fnar\_bezeichnung (land use).

Use the bbox parameter to filter resources within a specific area:

https://gis.lfrz.gv.at/ogcapi009501/ogc/features/collections/ogcapi009501:INVEKOS\_feldstuecke\_aktuell\_polygon/items?f=json&bbox=14,48,14.02,48.02

For a visual representation of the bounding box, enter the coordinates 14,48,14.02,48.02 into [linestrings.com](https://linestrings.com/bbox/) and click “Display box”.

To execute the request in R, use the following script:

library(httr2)  
library(geojsonsf)  
library(sf)  
  
full\_url <- "https://gis.lfrz.gv.at/ogcapi009501/ogc/features/collections/ogcapi009501:INVEKOS\_feldstuecke\_aktuell\_polygon/items?f=json&bbox=14,48,14.02,48.02"  
  
# Make the request and convert the response to an sf object  
invekos <- httr2::request(full\_url) %>% # Create request  
 httr2::req\_perform() %>% # Execute request  
 httr2::resp\_body\_string() %>% # Extract JSON body as string  
 geojsonsf::geojson\_sf() # Convert JSON string to sf object

The above code retrieves 100 polygon features and stores them in an object named invekos.

## 9.2 Data Cleaning

Once the data is loaded into R, we can more closely examine its structure. The dplyr function glimpse, for instance, allows us to preview each column’s name and type:

library(dplyr)  
dplyr::glimpse(invekos)

## Rows: 100  
## Columns: 15  
## $ fart\_id <dbl> 1783, 1783, 1783, 1783, 1783, 1783, 1783, 1783, 1783…  
## $ gml\_geom <chr> "[B@7202d363", "[B@2ab8edc1", "[B@5d565dad", "[B@6ae…  
## $ gml\_length <dbl> 786, 581, 550, 780, 754, 1160, 1090, 382, 448, 1535,…  
## $ geom\_date\_created <chr> "2022-10-01T04:48:30+02:00", "2022-10-01T05:16:44+02…  
## $ log\_pkey <dbl> 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3…  
## $ fnar\_code <chr> "A", "G", "A", "A", "A", "A", "A", "G", "A", "A", "A…  
## $ gml\_identifier <chr> "https://data.inspire.gv.at/0095/5f147bb9-7fe5-41ba-…  
## $ fs\_kennung <dbl> 103931764, 103931719, 103931731, 103931769, 10384647…  
## $ geo\_part\_key <dbl> 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, 14, …  
## $ gml\_id <chr> "AT.0095.5f147bb9-7fe5-41ba-a91e-0a3595c8335e.elu.Ex…  
## $ inspire\_id <chr> "https://data.inspire.gv.at/0095/5f147bb9-7fe5-41ba-…  
## $ fnar\_bezeichnung <chr> "ACKERLAND", "GRÜNLAND", "ACKERLAND", "ACKERLAND", "…  
## $ fs\_flaeche\_ha <dbl> 2.54829187, 0.12070263, 0.52804037, 1.39019474, 1.26…  
## $ geo\_id <dbl> 12267790, 12286832, 12286835, 12267794, 12315820, 12…  
## $ geometry <POLYGON [°]> POLYGON ((14.01765 48.01724..., POLYGON ((14…

The abbreviation dbl stands for double, a data type that stores numbers with decimal points.

To create a subset of the invekos sf-object, use the following code:

invekos %>%  
 head(2) %>%  
 dplyr::select(fnar\_bezeichnung, fs\_flaeche\_ha, geo\_id, geometry) %>%  
 knitr::kable(., format="html")

fnar\_bezeichnung

fs\_flaeche\_ha

geo\_id

geometry

ACKERLAND

2.5482919

12267790

POLYGON ((14.01765 48.01724…

GRÜNLAND

0.1207026

12286832

POLYGON ((14.01353 48.02075…

Field names and entries are in German. To rename them, use the base R function colnames():

# Subset of invekos object  
invekos.sub <- invekos %>%  
 dplyr::select(fnar\_bezeichnung, fs\_flaeche\_ha, geo\_id, geometry)  
  
# Renaming fields  
colnames(invekos.sub)[1] <- "land\_use"  
colnames(invekos.sub)[2] <- "area\_ha"

Entries are renamed using dplyr’s mutate and case\_when functions:

unique(invekos.sub$land\_use) # Unique entries in 'land\_use'

## [1] "ACKERLAND" "GRÜNLAND"

# Renaming entries 'ACKERLAND' and 'GRÜNLAND'  
invekos.sub <- dplyr::mutate(invekos.sub, land\_use =   
 case\_when(land\_use == 'ACKERLAND' ~ 'arable land',  
 land\_use == 'GRÜNLAND' ~ 'grassland', TRUE ~ 'Other'))  
  
# Display result  
invekos.sub %>%  
 head(2) %>%  
 knitr::kable(., format="html")

land\_use

area\_ha

geo\_id

geometry

arable land

2.5482919

12267790

POLYGON ((14.01765 48.01724…

grassland

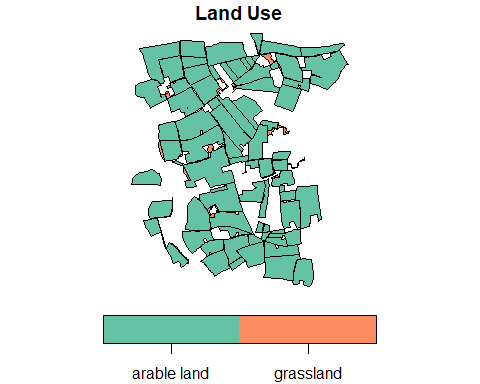
0.1207026

12286832

POLYGON ((14.01353 48.02075…

For an initial visual impression, plot the sf-object using the base R plot() function:

plot(invekos.sub[1], main="Land Use", key.pos = 1, key.width = lcm(1.8))



By default, the plot() function generates a multi-panel plot, with one sub-plot for each field of the object. However, specifying invekos.sub[1] restricts the output to a single plot, specifically showcasing the land\_use field. The function parameter key.pos = 1 aligns the legend below the map (1=below, 2=left, 3=above and 4=right). key.width defines the width of the legend.

Next, we can validate the geometry of sf polygons using st\_is\_valid:

sf::st\_is\_valid(invekos) %>%  
 summary()

## Mode TRUE   
## logical 100

The st\_is\_valid function checks the validity of each geometry, returning a logical vector. The summary confirms that all geometries in our dataset are valid. If any invalid geometries are detected, they can be corrected using the st\_make\_valid function.

Let’s closely investigate a polygon geometry with the code below, which creates a single polygon feature. The validity check flags an invalid geometry. Can you identify the problem?

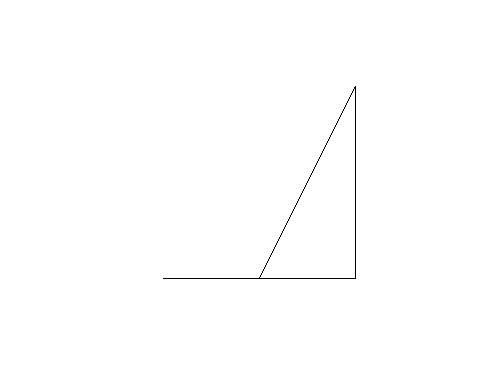
# Coordinates for the polygon  
coords <- matrix(c(-1, -1, 1, -1, 1, 1, 0, -1, -1, -1), ncol = 2, byrow = TRUE)  
  
# List of coordinate matrices  
list\_of\_coords <- list(coords)  
  
# Constructing the polygon  
polygon <- st\_polygon(list\_of\_coords)  
  
# Converting to an sf object  
error\_sf <- st\_sf(geometry = st\_sfc(polygon))  
  
# Validity check  
sf::st\_is\_valid(error\_sf)

## [1] FALSE

**See solution!**

When plotted, the polygon reveals a [sliver polygon](https://en.wikipedia.org/wiki/Sliver_polygon), which is invalid due to its shape.

plot(error\_sf)



Another common cause of invalid geometries is self-intersection of lines.

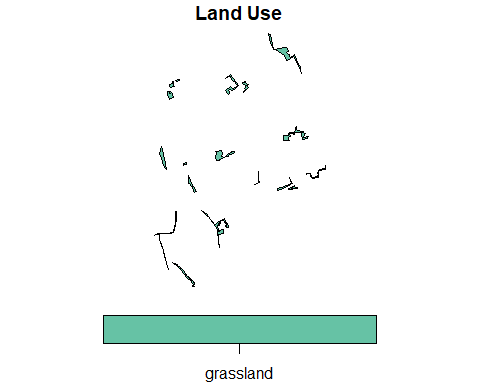
Note: A valid polygon requires at least four coordinate points with the first and last points being identical. A polygon with only three points is considered to have an invalid geometry.

## 9.3 Vector operations

The structure of an sf object is similar to that of data frames, enabling attribute-based filtering operations using dplyr functions, as described in Lesson [7](#datman).

For example, to extract grassland parcels, the filter() function is used:

invekos.sub %>%  
 dplyr::filter(land\_use=='grassland') %>%  
 {plot(.[1], main="Land Use", key.pos = 1, key.width = lcm(1.8))}



Note the use of curly brackets in the plot() function. This is required because the dot notation (.) cannot directly index a piped value. For a deeper explanation, see this [Stack Overflow discussion](https://stackoverflow.com/questions/42385010/using-the-pipe-and-dot-notation).

### 9.3.1 Geometrical operations

sf provides a range of geometric predicates such as st\_within, st\_contains, or st\_crosses, with a complete list available [here](https://r-spatial.github.io/sf/reference/geos_binary_pred.html). While these are often used between pairs of simple feature geometry sets, they can also operate on a single sf object:

sf::st\_intersects(invekos.sub, sparse = TRUE)

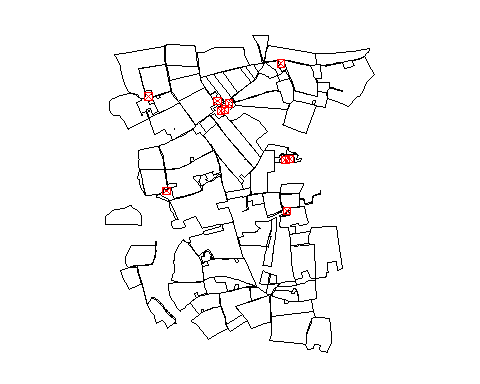
## Sparse geometry binary predicate list of length 100, where the  
## predicate was `intersects'  
## first 10 elements:  
## 1: 1, 17, 62  
## 2: 2, 4, 16  
## 3: 3, 64, 83  
## 4: 2, 4  
## 5: 5  
## 6: 6, 36  
## 7: 7, 8, 66, 99  
## 8: 7, 8  
## 9: 9, 28, 30, 54  
## 10: 10, 74

This returns a sparse matrix revealing intersections within features of the same sf object invekos.sub, indicating possible topology errors. For correcting these errors, GIS software like [QGIS](https://www.qgis.org/de/site/) is recommended as R’s capabilities for handling topology errors are less advanced.

### 9.3.2 Binary operations

Binary operations are essential for analyzing spatial relationships between different datasets. Here, we demonstrate this by creating an sf object representing farmsteads using data from the [AMA Rest API service](https://gis.lfrz.gv.at/ogcapi009501/ogc/features/collections?f=text%2Fhtml):

full\_url <- "https://gis.lfrz.gv.at/ogcapi009501/ogc/features/collections/ogcapi009501:INVEKOS\_hofstellen\_aktuell\_point/items?f=json&bbox=14,48,14.02,48.02"  
  
farms <- httr2::request(full\_url) %>% # Create request  
 httr2::req\_perform() %>% # Execute request  
 httr2::resp\_body\_string() %>% # Extract JSON body as string  
 geojsonsf::geojson\_sf() # JSON string to sf object  
  
plot(invekos.sub[1], main=NULL, key.pos=NULL, reset = FALSE, col\_graticule = "grey")  
plot(farms[1], main=NULL, key.pos=NULL, pch = 7, col='red', add = TRUE, cex=1)



Note: The plot() function is used sequentially to overlay farm points on top of the land parcel polygons. For advanced plotting techniques, refer to the [plot() documentation](https://r-spatial.github.io/sf/reference/plot.html).

We proceed to calculate the distances between farms and land parcels to determine proximity:

dist\_m <- sf::st\_distance(farms, invekos.sub)

The st\_distance function calculates the shortest distance matrix between the geometries of the two sf objects, with distances returned in meters.

Distances are computed in meters, which may seem unexpected since the reference system’s units are in degrees. To understand the underlying calculations, examine the coordinate reference system of the sf objects farms and invekos using sf::st\_crs(). This will reveal that objects use geographic coordinates (WGS 84).

In this short exercise, we will take a closer look at the algorithm that is implemented in function st\_distance.

Open the [documentation of st\_distance](https://r-spatial.github.io/sf/reference/geos_measures.html) to find out how metric distances were derived from geographic coordinates.

**See solution!**

According to the documentation, greater circle distances are computed for geodetic coordinates. Greater circle distance calculations use by default spherical distances. Alternatively, distances can be computed based on an ellipsoidal model. See [Algorithms for geodesics, Journal of Geodesy](https://link.springer.com/article/10.1007/s00190-012-0578-z) for more information.

When plotting the complete distance matrix dist\_m, we see that column 1 contains the distances between the first feature (parcel 1) in sf-object invekos and the 10 farm features of sf-object farms. Accordingly, the matrix has 100 columns (one column for every parcel) and 10 rows (one row for every farm).

The following line returns the first column of the distance matrix as vector:

dist\_m[, 1]

## Units: [m]  
## [1] 255.9825 642.4693 547.6458 621.1653 1326.6061 1230.1068 510.2196  
## [8] 587.8220 926.4913 492.0585

To identify the farm that is located closest to parcel 1, we need to query the index of the minimum value in this vector:

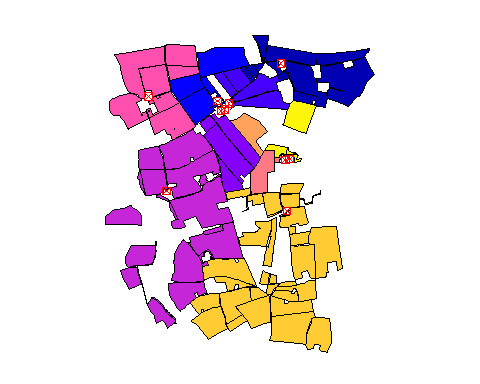
which(dist\_m[, 1] == min(dist\_m[, 1]))

## [1] 1

The which function, a base R utility, identifies the indices of elements that satisfy a given condition. In the example, it returns the index of the smallest value within the vector dist\_m[, 1]. Consequently, it indicates that farm 1 is the closest to parcel 1.

The demonstrated procedure for detecting closest farms can be executed for ever parcel in a for-loop. The number of the closest farm is appended to a vector named closest. This vector is in turn appended as a new column to sf-object invekos.sub. And invekos.sub is plotted together with farms:

closest <- c()  
  
for (i in 1:100){  
   
 out <- which(dist\_m[, i] == min(dist\_m[, i]))  
   
 closest <- c(closest, out)  
   
}  
  
cbind(invekos.sub, closest) %>%  
 {plot(.[4], main=NULL, key.pos=NULL, reset = FALSE)}  
  
plot(farms[1], main=NULL, key.pos=NULL, pch = 7, col='red', add = TRUE, cex=1)



The output map visualizes the closest farm to each agricultural parcel, highlighting the practical application of sf Geometrical operations.

This chapter focuses on logical matrix outputs from geometric operations. For operations generating new geometries, such as st\_union, st\_buffer, and st\_centroid, see [operations returning a geometry](https://r-spatial.github.io/sf/articles/sf3.html#operations-returning-a-geometry). For network operations on sf objects, consider using sfnetworks or the igraph package.

## 9.4 Raster operations

Having previously discussed the structure of SpatRaster Objects in Lesson [4](#spds) and the reading and writing of such objects in Lesson [8](#readwrite), we now turn our attention to raster manipulation operations like *resampling* and *cropping*.

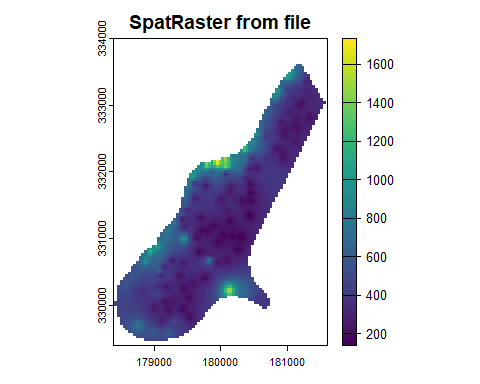
For the following examples, we will utilize a sample dataset from the terra package. You can download the sample data [here](data/terra_sample.tif)).

### 9.4.1 Resampling

Resampling is crucial for working with raster datasets. It alters the spatial resolution, allowing you to align multiple rasters with different resolutions. Additionally, it adjusts the level of detail necessary for your analysis.

Let’s demonstrate resampling with the terra sample dataset:

library(terra)  
  
r <- terra::rast("data/terra\_sample.tif") # path may be different on your machine  
plot(r, main='SpatRaster from file')



Before we change the raster resolution of SpatRaster Object, it is important to know the original resolution of the raster. You can use the res() function to check the original resolution:

res(r)

## [1] 40 40

For resampling, we’ll create a target raster by copying the original and setting a new resolution:

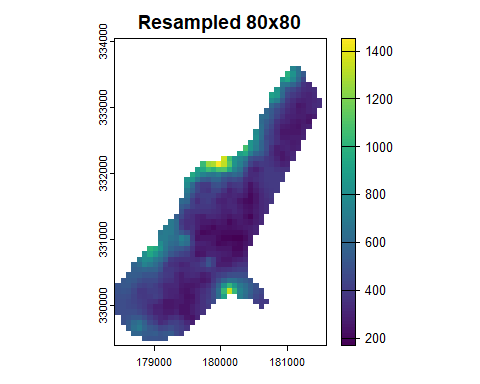
r2 <- r  
terra::res(r2) <- 80 # Assigning a new resolution of 80x80

Although this operation clears the raster values, r2 can still be used as a target for resampling:

r\_resampled <- terra::resample(r, r2, method="bilinear")  
  
res(r2) # Verify the new resolution

## [1] 80 80

plot(r\_resampled, main='Resampled 80x80')



The bilinear method is used for interpolating new values in r\_resampled, with alternatives such as nearest or cubic. The choice of interpolation method for resampling raster data is a crucial consideration, discussed in further detail [here](https://gis.stackexchange.com/questions/2587/deciding-what-interpolation-method-to-use-for-resampling-raster-data).

### 9.4.2 Crop raster

Raster cropping allows you to select a specific area from a larger raster dataset for targeted analysis. Both SpatRaster and SpatExtent objects can be utilized for cropping operations.

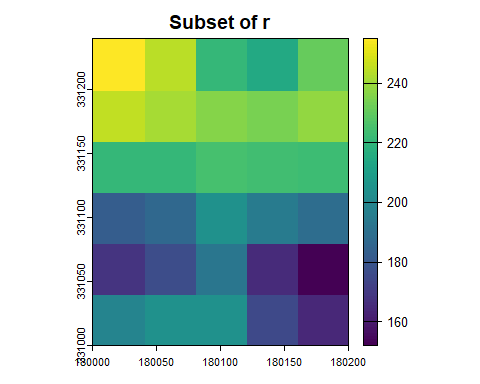
To derive an extent from a SpatRaster, the ext function is used:

terra::ext(r)

## SpatExtent : 178400, 181600, 329400, 334000 (xmin, xmax, ymin, ymax)

The output format is <xmin, xmax, ymin, ymax>. Knowing this, we can define a cropping extent and apply it using the crop function:

crop\_ext <- terra::ext(180000, 180200, 331000, 331250)  
subset\_r <- terra::crop(r, crop\_ext)  
plot(subset\_r, main='Subset of r')



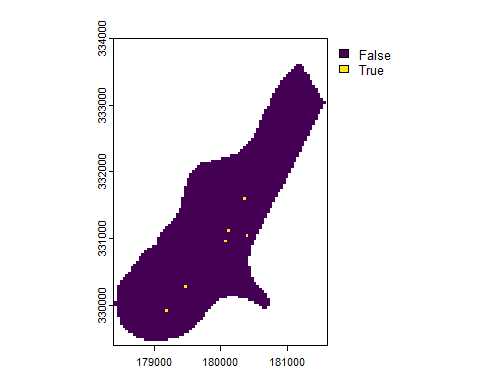
If the resulting cropped area doesn’t match expectations due to cell alignment, adjust the cropping extent or raster resolution as necessary.

### 9.4.3 Raster algebra

Terra provides functionality for algebraic operations on rasters, supporting standard arithmetic and logical operators, and functions like abs and round. For a full list, visit the [terra algebra documentation](https://rspatial.org/pkg/4-algebra.html).

Here’s how you can perform some simple operations:

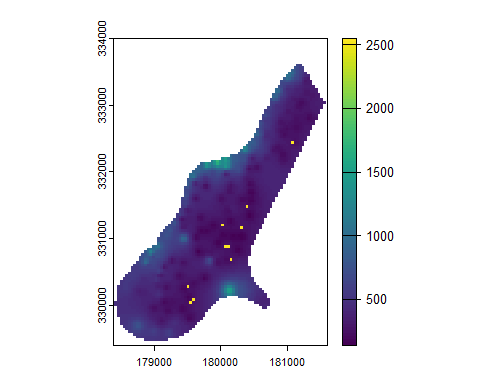
# Add a constant value to each cell  
s <- r + 10  
# Calculate the square root of cell values  
s <- sqrt(r)  
# Generate a logical raster based on cell value equality  
s1 <- s == 15  
  
plot(s1)



Operators can be applied between SpatRaster objects with matching resolutions and origins. The result covers their spatial intersection.

You can also perform in-place replacements:

r[r == 255] <- 2550  
plot(r)



Here, all cells with the value 255 are replaced with 2550.

A broader range of vector and raster operations can be found in [Chapter 5 - Geometry Operations, in the Book Geocomputation with R](https://bookdown.org/robinlovelace/geocompr/geometric-operations.html).

**Note:** As of writing this module, “Geocomputation with R” uses the raster package, which is expected to be superseded by terra.

# 10 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/).

## 10.1 The Grammar of Graphics

The [ggplot2](https://ggplot2.tidyverse.org/) library, a part of the Tidyverse suite, is renowned for its comprehensive and intuitive approach to data visualization in R. Rooted in the principles of “The Grammar of Graphics”, conceptualized by [Leland Wilkinson](https://en.wikipedia.org/wiki/Leland_Wilkinson), ggplot2 enables users to construct graphics through a layered approach, incorporating seven distinct elements:

“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 core dataset to be visualized.
2. **Aesthetics:** Mappings of variables to visual scales, like color or size.
3. **Geometries:** The visual representation of data, such as points, lines, or bars.
4. **Facets:** Creating subsets of data to generate similar graphs for each subset.
5. **Statistics:** Applying statistical transformations to data (mean, median, etc.).
6. **Coordinates:** Managing axes and spatial transformation.
7. **Themes:** Customizing the graphical backdrop for enhanced visual appeal.

In essence, these layers enable a structured and flexible approach to crafting visual narratives from data. For instance, the visual variables such as size, shape, and color offer nuanced ways to represent and differentiate data points.

In the example below, ‘Gdp per capita’ and ‘Life Expectancy’ align with the x and y axes, respectively, and ‘national population’ and ‘world regions’ are differentiated by size and color.



Figure 10.1: Visual variables color and size in a ggplot2 graph

Upcoming sections will delve into practical examples, demonstrating the power and versatility of ggplot2 in visualizing complex datasets.

**Further Reading:** For a foundational understanding of “The Grammar of Graphics” as implemented in ggplot2, refer to Hadley Wickham’s insightful [article](https://byrneslab.net/classes/biol607/readings/wickham_layered-grammar.pdf).

## 10.2 Understanding Wide and Long Data Formats in R

When working with data in R, especially for visualization purposes, it’s crucial to understand the structure of your dataset. Typically, datasets can be categorized into two main formats: **Wide** and **Long** (also known as **Tidy**) formats. The format of your data can significantly impact how you manipulate and visualize it.

### 10.2.1 Wide format

* **Characteristics:** In a wide-format dataset, each subject or entity (such as a city) is represented once, with multiple columns for different variables or time periods.
* **Example:** Consider a dataset representing annual rainfall measurements across different cities, measured in millimeters (mm). In this wide format, each row corresponds to a city, and the columns represent rainfall measurements for different years.

# Example of a wide format dataset  
wide\_data <- data.frame(  
 City = c("CityA", "CityB", "CityC", "CityD", "CityE"),  
 Rainfall\_2015 = c(600, 500, 550, 450, 400),  
 Rainfall\_2016 = c(650, 550, 600, 500, 450)  
)  
  
knitr::kable(wide\_data)

| City | Rainfall\_2015 | Rainfall\_2016 |
| --- | --- | --- |
| CityA | 600 | 650 |
| CityB | 500 | 550 |
| CityC | 550 | 600 |
| CityD | 450 | 500 |
| CityE | 400 | 450 |

### 10.2.2 Long format:

* **Characteristics:** In a long-format dataset, each row is a single observation for a single variable, often requiring multiple rows per subject or unit
* **Example:** Using the same rainfall dataset, a long format would list each year’s rainfall for each city as a separate row.

# Example of long-format data:  
long\_data <- tidyr::pivot\_longer(   
 wide\_data,  
 cols = -City,   
 names\_to = "Year",   
 values\_to = "Rainfall"  
)  
knitr::kable(long\_data)

| City | Year | Rainfall |
| --- | --- | --- |
| CityA | Rainfall\_2015 | 600 |
| CityA | Rainfall\_2016 | 650 |
| CityB | Rainfall\_2015 | 500 |
| CityB | Rainfall\_2016 | 550 |
| CityC | Rainfall\_2015 | 550 |
| CityC | Rainfall\_2016 | 600 |
| CityD | Rainfall\_2015 | 450 |
| CityD | Rainfall\_2016 | 500 |
| CityE | Rainfall\_2015 | 400 |
| CityE | Rainfall\_2016 | 450 |

Understanding these formats is crucial for effective data manipulation and visualization in R, especially when using libraries like ggplot2 and tidyverse. Certain types of visualizations and statistical analyses are more straightforward with data in a specific format.

**Note**: Transforming data between wide and long formats is often achieved using functions like pivot\_longer() and pivot\_wider() from the [tidyr](https://tidyr.tidyverse.org/reference/index.html) package, which is in included in the Tidyverse.

## 10.3 Visualization of distributions

As already mentioned above, functions in the ggplot2 library are structured according to “The Grammar of Graphics”. When creating graphs with ggplot2, we start by setting up data and aesthetics (aes()), then defining the type of plot (geometry) like geom\_point, and finally enhancing the plot with additional transformations and themes.

We begin 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, one of the nine states in Austria.

**Data Overview:**

* **Time:** Timestamp of each measurement.
* **Station:** Identifier for the measurement station.
* [**Component**](data/MetadataAirqual.xlsx)**:** The air quality component measured.
* [**Meantype**](data/MetadataAirqual2.xlsx)**:** The temporal resolution of measurements.
* **Unit:** The unit of measurement.
* **Value:** The measured value of air quality.

Let’s start by displaying the first few rows of this dataset:

library(tidyverse)  
library(knitr)  
  
# Read the dataset (Note: Semi-colon separated)  
airquality <- read\_delim("data/AirQualityUpperAut.csv", delim = ";")  
  
# Display the first five rows  
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



To differentiate between measurements from various stations, we can map the station variable to the color attribute:

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]") + # add x-axis label  
 ggplot2::ylab("Count") + # add y-axis label  
 scale\_fill\_manual(name = "Measurement stations", values = c("grey20", "grey50", "grey80")) + # add legend  
 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 the solution!**

Insert a title by adding:

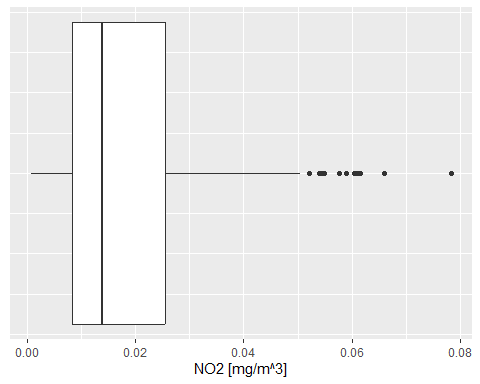
ggplot2::ggtitle(“Nitrogen dioxide concentration”)

## 10.4 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.

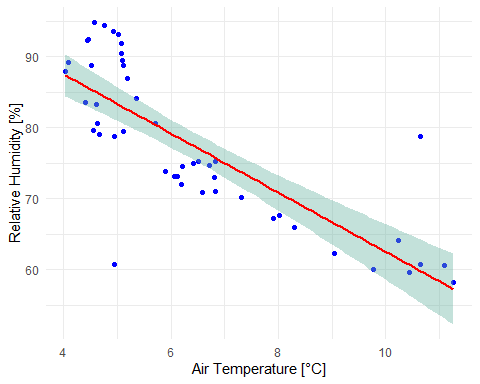
See [our solution](Solution_Exercise_boxplots.R)!

## 10.5 Scatterplots

Scatterplots serve as a pivotal tool in statistical analysis, particularly when it comes to exploring the interplay between two variables. Their strength lies in visually capturing the nature and strength of relationships.

Consider the scenario where we seek to understand the relationship between air temperature (TEMP) and relative humidity (RF). For this purpose, we delve into the airquality dataset, focusing on half-hourly readings from station S108. Our objective is to juxtapose the temperature and humidity readings and analyze their correlation.

# Extract half-hourly temperature readings from station S108  
temp\_tab <- airquality %>%  
 dplyr::filter(component == "TEMP", meantype == "HMW", station == "S108")   
  
# Similarly, extract half-hourly humidity readings  
humidity\_tab <- airquality %>%  
 dplyr::filter(component == "RF", meantype == "HMW", station == "S108")   
  
# Join the two datasets on 'time'  
temp\_humidity\_joined <- temp\_tab %>%  
 dplyr::inner\_join(humidity\_tab, by = "time") %>%  
 dplyr::select(time, value.x, value.y)   
  
# Create the scatterplot  
ggplot(temp\_humidity\_joined, aes(x = value.x, y = value.y)) +  
 xlab("Air Temperature [°C]") +  
 ylab("Relative Humidity [%]") +  
 geom\_point(color = "blue") +  
 geom\_smooth(method = lm, color = "red", fill = "#69b3a2", se = TRUE) +  
 theme\_minimal()



The generated scatterplot vividly illustrates the relationship between air temperature and relative humidity. Typically, as the temperature drops, the relative humidity tends to rise, and vice versa. This inverse relationship, although not strictly linear, can be approximated using a linear regression model. Deviations from this model are captured within a 95% confidence interval, providing a more nuanced understanding of the data dynamics.

For a comprehensive explanation of the inverse relationship between relative humidity and temperature, refer to [this video](https://www.youtube.com/watch?v=saojnBSBUuk)

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)

While exploring, consider the following:

1. How many individual measurements are visualized in the scatterplot?
2. What do value.x and value.y represent in this context?
3. The geom\_smooth() function, with method = lm, fits a linear model. What is the significance of the se argument in this function?

**Solution and Insights**

1. The scatterplot encompasses 45 records, corresponding to half-hourly measurements over approximately 24 hours.
2. In the joined dataset, value.x and value.y represent the temperature and humidity values, respectively, after renaming to avoid duplication.
3. The se argument controls the display of confidence bounds around the regression line, which are shown by default when se is set to TRUE.

## 10.6 Map Visualization

In our previous exploration, we learned how to employ the plot() function for basic map layouts. This section, however, elevates our cartographic journey by harnessing the versatility of the ggplot2 library for more intricate map visualizations.

Before diving into the examples, ensure you have the sf and ggplot2 libraries installed and loaded. Additionally, download the [North Carolina](data/nc.zip) and [US States](data/us-states.zip) datasets for practical implementation.

Our first endeavor is to construct a foundational map of North Carolina:

library(sf)  
library(ggplot2)  
  
# Load North Carolina shapefile  
nc <- sf::st\_read("data/nc.shp")  
  
# Basic map of North Carolina  
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:

# Map with custom colors  
ggplot(data = nc) +   
 geom\_sf(color = "black", fill = "lightgreen")



Data attributes can be visually represented as well. In this example, the AREA variable influences the fill color:

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



The function coord\_sf() allows you to work with the coordinate system, which includes both the 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:

# Map with changed coordinate system  
ggplot(data = nc) +  
 geom\_sf() +  
 coord\_sf(crs = st\_crs(32618)) # Projecting to WGS 84 / UTM zone 18N



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)))

we 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 it 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.

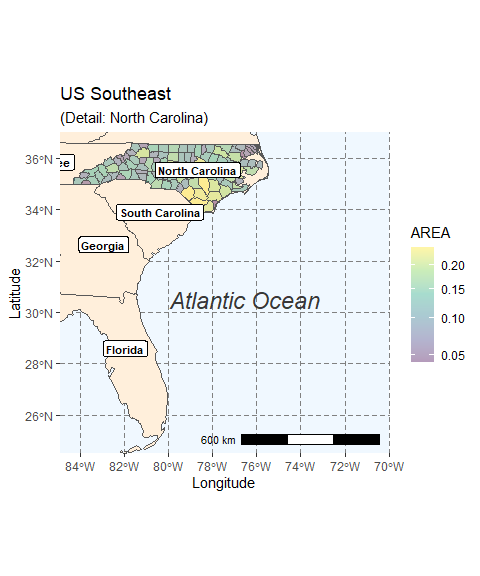
After deriving centroid coordinates from nc geometries, we call the new variable nc\_points in function geom\_text and map X and Y columns (centroid coordinates) onto visual variables x and y (position in graph) and also map column NAME onto visual variable label. Moreover, we can insert individual text 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 our final example, we combine previously introduced methods to craft a detailed map visualization. This example layers data sources, customizes feature appearances, and adds informative annotations:

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"))



Here, geom\_sf() adds layers of US state polygons. geom\_label() offers an alternative to geom\_text() for feature labeling, with nudge\_x providing horizontal label offset. The map is saved in both PDF for high-quality prints and PNG for web use. Remember to consult the [ggplot2 Cheatsheet](https://www.rstudio.com/resources/cheatsheets/) for an overview of key ggplot2 operations.

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

## 10.7 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).

Start by initializing a Leaflet map widget and augment it with base layers and interactive markers:

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

The pipe operator (%>%) conveniently adds layers, thanks to Leaflet’s function design.

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/).

# 11 R Markdown

This lesson is dedicated to the rmarkdown library. However, R Markdown is **more than just a library**, [RMarkdown](https://rmarkdown.rstudio.com/) is part of a set of tools designed to enhance the reproducibility of your work. Other tools and platforms such as [GitHub](https://github.com/), [Jupyter](https://jupyter.org/), [Docker](https://www.docker.com/), [ArXiv](https://arxiv.org/), and [bioRxiv](https://www.biorxiv.org/) can facilitate reproducibility in various ways. In this module, we won’t explore the paradigm of [reproducible research](https://esajournals.onlinelibrary.wiley.com/doi/full/10.1002/bes2.1801) in detail. Instead, our focus will be on how to use RMarkdown to make your analyses and reports more appealing, interactive, and **efficient**.

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

## 11.1 Set up your work environment

Before creating your first R Markdown document, we need to **set up** the **GitHub** environment. Originally founded as a platform for software developers, GitHub’s architecture is designed to manage changes made during software development. This architecture is also beneficial for version control of documents or any information collection.

Version control is especially important when working in teams, as it helps synchronize efforts among project participants. However, GitHub is also a reliable and open online platform for individual work, providing change tracking, documentation, and sharing features.

To set up your personal GitHub environment, follow these steps:

1. Review the [Hello-World Section](https://docs.github.com/en/get-started/quickstart/hello-world) in GitHub’s Quickstart Documentation. Initially, reading it is sufficient—no need to complete the tutorial yet.
2. [Create a GitHub account](https://github.com/).
3. [Download and install Git](https://git-scm.com/downloads). Git is a distributed **VCS (version control system)** that mirrors the codebase and its full history on every computer. GitHub is a web-based interface that integrates seamlessly with Git. For a clear explanation of Git’s core concepts, watch this [video](https://www.youtube.com/watch?v=uR6G2v_WsRA).
4. In RStudio (under Tools > Global Options > Git / SVN), check “enable version control” and set the path to git.exe (e.g., C:/Program Files/Git/bin/git.exe). Restart RStudio afterward.
5. [Create a repository](https://docs.github.com/en/get-started/quickstart/create-a-repo) on GitHub. In the tutorial, skip the section ‘Commit your first changes’.
6. By default, your repository will have one branch named main. Create an additional branch called dev off the main. Follow the instructions in the [Hello-World Tutorial](https://docs.github.com/en/get-started/quickstart/hello-world) for guidance.

Next, install the rmarkdown library and tinytex in RStudio as described in the [R Markdown Guide](https://bookdown.org/yihui/rmarkdown/installation.html).

To properly **install**tinytex, execute both lines in RStudio:

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

Follow RStudio’s prompts to install any dependencies. For technical issues, please consult the discussion forum.

## 11.2 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, go to (File > New Project > Version Control > Git).

Enter the URL of your online repository (find this URL in your GitHub repository) and select a local directory for the clone. Then click “Create Project” (refer to Fig. 11.1).



Figure 11.1: 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. 11.2).



Figure 11.2: New features in RStudio

By default, the repository includes three files:

1. [.gitignore](https://git-scm.com/docs/gitignore): Specifies intentionally untracked files to ignore.
2. [RStudio Project File](https://support.rstudio.com/hc/en-us/articles/200526207-Using-RStudio-Projects) (.Rproj): Contains metadata for the RStudio project.
3. [ReadMe File](https://docs.github.com/en/repositories/managing-your-repositorys-settings-and-features/customizing-your-repository/about-readmes) (.md): A markdown file with information about the repository.

The [gitignore](https://git-scm.com/docs/gitignore) and [.Rproj](https://support.rstudio.com/hc/en-us/articles/200526207-Using-RStudio-Projects) files are created during project initialization and are not yet in the online repository. Modifications appear in the “Git” tab (Fig. 11.3).



Figure 11.3: Changes in Git tab

Before making further changes, switch to the dev branch (see Fig. 11.4). At this point, the dev branch mirrors the main branch.



Figure 11.4: Switch branch

It is **highly recommended** to work in progress on a separate developer branch, like dev, and keep the main branch for stable versions. You can later merge changes from dev to main through a pull request (see [Opening a Pull Request](https://docs.github.com/en/get-started/quickstart/hello-world#opening-a-pull-request)).

## 11.3 Creating Your First R Markdown Document

Now that the environment is set up, let’s create our first R Markdown document.

In RStudio: Navigate to (File > New File > R Markdown). Enter a title for your document, accept the default settings, and click “OK”. You’ll receive a sample R Markdown file with the extension .Rmd.

An R Markdown document comprises three core components: **metadata**, **text** and **code** (see Fig. 11.5).



Figure 11.5: R Markdown sample file

The metadata, written in [YAML](https://en.wikipedia.org/wiki/YAML) syntax, defines document properties like title, output format, and creation date. Explore more about YAML syntax and document properties [here](https://cran.r-project.org/web/packages/ymlthis/vignettes/yaml-overview.html).

To automatically update the date in your document, insert date: "```r format(Sys.time(), '%d %B, %Y')```" in the metadata section. This outputs the current date based on your system’s time zone in a human-readable format.

After the metadata section an **R inline code block** starts and ends with three backticks (see Fig. 11.5). 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. 11.6).



Figure 11.6: 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.

Explore more about code block options in 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. 11.7).



Figure 11.7: Knit HTML Output

Use the dropdown next to the “Knit” button to compile into formats such as PDF or .docx, among [others](https://rmarkdown.rstudio.com/lesson-9.html).

Knitting an R Markdown document involves a two-step process. First, the .Rmd file is processed by the knitr package, which executes the code chunks and generates a new markdown file with the code and its output. Then, pandoc converts this markdown file into the final output document in the chosen format, allowing a wide range of output options for creating professional-quality documents.

## 11.4 Synchronizing with GitHub

Regular synchronization of your local changes with the online repository is a key practice in version control. Start by pulling any updates from the repository.

In the RStudio Git tab, click the “Pull” button (see Fig. 11.8). A notification should indicate whether any new changes are available (e.g., Already up to date).



Figure 11.8: Make Pull

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

Next, commit your changes. Think of committing as taking a snapshot of your progress, accompanied by a descriptive message.

First, save all documents in RStudio. Then, hit the “Commit” button in the Git tab. The commit window will display a list of modified files. Green highlights indicate new content; red highlights show deleted content.

Check the boxes next to each file to include them in the commit. Alternatively, run git add -A in the terminal to add all files at once (see this [list of popular Git commands](https://github.com/joshnh/Git-Commands)). After selecting files, enter a meaningful commit message and click “Commit”.

See Fig. 11.9.



Finally, push your committed changes to the online repository.

Figure 11.10: Make Push

Your online repository on GitHub should now be updated (switch to dev branch in your repository) (see Fig. 11.11).



Figure 11.11: Commit with message ‘describe sync process in GitHub’ was pushed to the developer branch a minute ago

## 11.5 Basic R Markdown Syntax

In R Markdown, you can apply text formatting using simple markers:

**Bold:** Double asterisks \*\*Text\*\* turn text **bold**. **Italicize:** Single asterisks \*Text\* create *italicized* text. **Headings:** Use hash signs # for headings. The number of hashes denotes the heading level:

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

**Tables** are created by using the symbols | and -. Recall the [numeric operators table](#numop) from the first lesson. Figure 11.12 shows the RMarkdown syntax used for that table:



Figure 11.12: 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

**Will result in:**

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

**Which will result in:**

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

**Hyperlinks** are created with the format [Text](URL), for example, [GitHub](https://github.com/){target="\_blank"} becomes [GitHub](https://github.com/). The target="\_blank" parameter opens the link in a new tab, which is a good practice when linking to external websites.

**Blockquotes** are indicated by > and can be nested:

>"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.

**Will result in:**

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** using 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.

### 11.5.1 References in RMarkdown

R Markdown facilitates an efficient method for inserting **citations** and building a **bibliography**. References are organized in a .bib file.

To begin, create a new document in a text editor, such as Windows Editor, and save it with a .bib extension (e.g., references.bib) in your RStudio project folder.

Consider using the RStudio project that you previously cloned, modified, and synchronized.

1. **Enable BibTeX Export:** Modify your settings in Google Scholar to enable BibTeX export (see Fig. 11.13).



Figure 11.13: Enable BibTeX in Firefox 106.0.1

Browser versions may vary. For assistance, refer to the discussion forum if needed.

1. **Export BibTeX Entries:** After enabling BibTeX export, a new link “Import into BibTeX” will appear in Google Scholar (see Fig. 11.14).



Figure 11.14: BibTeX Link in Firefox 106.0.1

Click the link and copy the BibTeX code into your .bib file.

1. **Integrate References in RMarkdown:** Specify the location of your .bib file in the YAML metadata of your RMarkdown document (bibliography: <.bib file>). Insert @ followed by the BibTeX key to add citations (see Fig. 11.15).



Figure 11.15: Integrate BibTeX reference in RMarkdown document

1. **Compile the Document:** Knit the R Markdown file as HTML, PDF, or Word. The rmarkdown package processes both indirect (without square brackets) and direct citations (with square brackets) and includes a bibliography (see Fig. 11.16).



Figure 11.16: Knit R Markdown as PDF

For a practical demonstration, download and explore this [RMarkdown reference example](data/ReferencesInRMarkdown.zip). Unzip the folder and open the .Rproj file in RStudio.

Explore more RMarkdown syntax examples in the [RMarkdown Cheatsheet](https://www.rstudio.com/wp-content/uploads/2015/02/rmarkdown-cheatsheet.pdf).

## 11.6 Speed up your workflows

R Markdown significantly enhances the efficiency of repetitive workflows. For instance, consider a scenario where a client requires daily updates on specific spatial economic indicators. Instead of manually generating a new report each day, R Markdown can automate this process, creating data reports with charts that update automatically upon compilation. This approach can save substantial time and effort.

Real-time data retrieval is possible through [Alpha Ventage](https://www.alphavantage.co/), which provides financial market data via the [Alpha Ventage Rest API](https://www.alphavantage.co/documentation/). The R library [alphaventager](https://cran.r-project.org/web/packages/alphavantager/index.html) facilitates API access within R. The use of alphaventager enables the extraction of various types of financial data, including real-time stock prices, FX rates, and technical indicators, directly into R. This allows for efficient data processing and visualization, making it a good tool for finance-related reports and analyses in R Markdown.

Explore a practical example by downloading this [draft finance data report](data/RMarkdownFinanceData.zip). Unzip the folder and open the .Rproj file in RStudio.

The project includes:

* A .bib file with a BibTeX reference.
* A .csv file in the data folder, listing over 400 country names, national currencies, and currency codes.
* An .Rmd file with inline R code that renders real-time currency exchange rates in a map.

Review the .Rmd file thoroughly before compiling an HTML output. Note that it includes an interactive Leaflet map, making HTML the only supported output format.

Try enhancing the report with an additional spatial indicator, such as a map displaying exchange rates from national currencies to the Euro.

## 11.7 Self-study

The vast functionalities of R Markdown extend beyond the scope of a single lesson. To fully exploit its capabilities, refer to the comprehensive online book [R Markdown: The Definitive Guide](https://bookdown.org/yihui/rmarkdown/).

This guide covers additional topics such as Notebooks, Presentations, support for languages like Python, C++, SQL, and more complex document creation with extensions like [BookDown](https://bookdown.org/) or [ThesisDown](https://github.com/ismayc/thesisdown). While R Markdown predominantly supports R, its flexibility extends to integrating other programming languages like Python, C++, and SQL. This integration is facilitated through specific settings in the code chunks of the R Markdown document. For instance, by specifying the python engine in a code chunk (e.g., {python}), you can seamlessly run Python code within your document. Similar approaches are used for C++ (using the cpp engine) and SQL (using the sql engine). These capabilities are enabled by the knitr package, which supports various languages, allowing for a multi-language analytical workflow within a single R Markdown document. Experimenting with these various features is key to mastering R Markdown.

**Fun Fact:** This module was written using BookDown.