

APPLICATION DEVELOPMENT WITH R

Christian Neuwirth

17 Juni, 2022



Table of Contents

1	Introduction to R	4
1.1	About this module	4
1.2	R programming language	5
1.3	Installation and Setup	6
1.4	Interpreting values	6
1.5	Simple data types	8
1.6	Numeric operators.....	9
1.7	Logical operators	10
1.8	References	10
2	Core concepts.....	12
2.1	Variables	12
2.2	Algorithms and functions	14
2.3	Libraries	16
3	Complex Data Types.....	19
3.1	Vectors.....	19
3.2	Multi-dimensional data types.....	23
4	Control structures	31
4.1	If	31
4.2	Else.....	32

4.3	Code blocks.....	33
4.4	Loops.....	34
4.5	Loops with conditional statements	37
5	Functions	39
5.1	Defining functions.....	39
5.2	More parameters.....	40
5.3	More return values	40
5.4	Functions and control structures	41
5.5	Scope.....	42
6	Data wrangling	45
6.1	Preparation	45
6.2	Data manipulation	46
6.3	Join.....	52
7	Read and write data	60
7.1	Read and write tabular data	60
7.2	Read and write vector data	61
7.3	Data API	65
8	Data visualization	69
8.1	The Grammar of Graphics	69
8.2	Visualization of distributions	72
8.3	Boxplots	76
8.4	Scatterplots.....	78
8.5	Map visualization	80

8.6 Interactive Maps.....87

Projects (select one out of two projects, project should cover all relevant concepts: control structures, data types, functions, selection and filtering, read and write, visualization): 1) total pandemic excess mortality, data: <https://www.mortality.org/>, select 10 countries, create map 2) Climate Change, Open Weather API, temp. difference two periods, 100 locations, create map, Important: cover APIs in lecture 'read and write'

1 Introduction to R

1.1 About this module

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

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

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

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

This module is partly based on the teaching materials [granolarr](#) worked out by [Stefano de Sabbata](#) at the [University of Leicester](#). For more information take a look at the [Granolarr Lecture Pages](#) or at the more comprehensive [Bookdown Version](#). The chapters on **Statistical Analysis** and **Machine Learning** are recommended as a follow up read for those who are willing to delve into more advanced applications of R.

1.2 R programming language

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

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

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

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

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

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

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

1.3 Installation and Setup

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

1. Download R from [R Archive Network \(CRAN.\)](#)
2. Follow the instructions and install the most up to date version on your machine (choose 'base' as well as 32-bit or 64-bit dependent on the bit-version of your operating system).

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

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

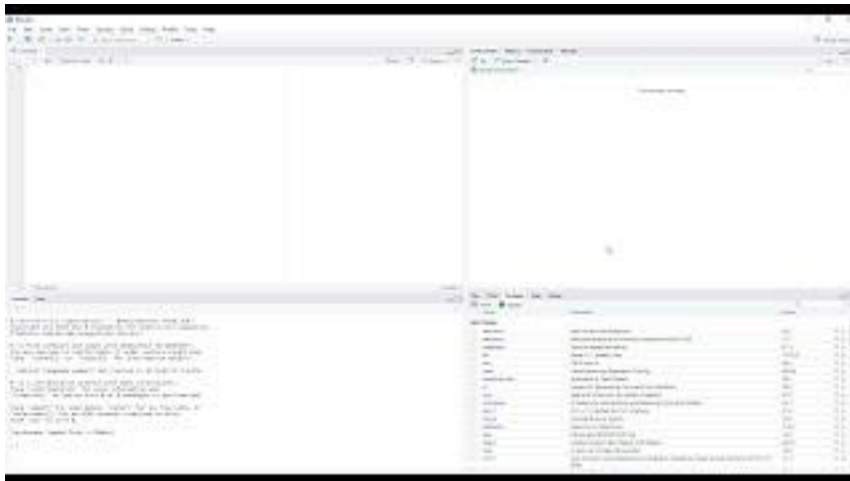


Figure 1.1: Video (6:09 min): RStudio for the Total Beginner.

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

1.4 Interpreting values

Now that you have installed RStudio and R on your machine, it is time to run some code in RStudio. When you open RStudio, you will find the **Console Window** (see Fig. 1.2). When values

and operations are inputted in the *Console*, the interpreter returns the results of its interpretation of the expression.

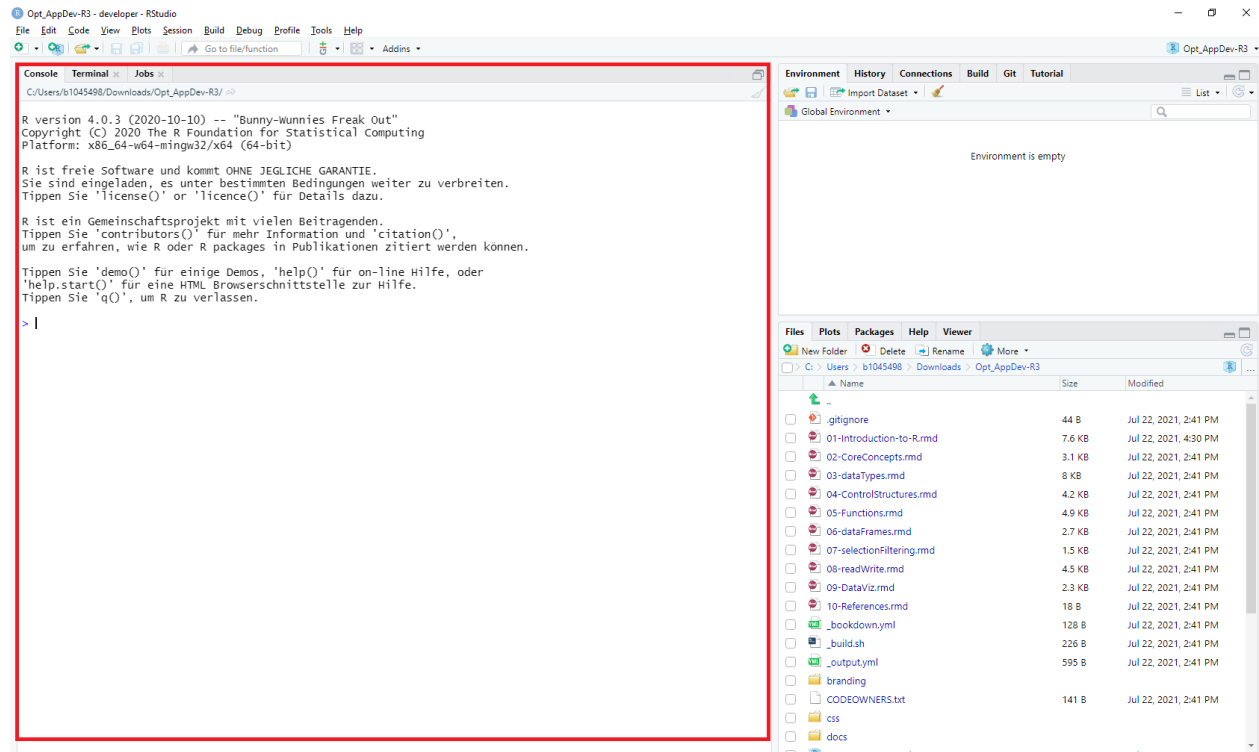


Figure 1.2: Console Window in RStudio

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

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

See solution!

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

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

1.5 Simple data types

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

R provides three core data types

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

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

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

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

See solution!

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

1.6 Numeric operators

R provides a series of basic numeric operators.

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

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

```
5 %/% 2
```

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

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

Execute 5 %% 2 to test the 'Module' operator.

See solution!

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

1.7 Logical operators

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

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

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

What logical values are returned by the following expressions:

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

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

1.8 References

Apart from [Stefano de Sabbata's](#) teaching materials, a number of other sources are use in this module. Most of them are freely available online:

- Granolarr - A geographic data science reproducible teaching resource in R by Stefano de Sabbata. see [Online Book](#)
- R for Geographic Data Science by Stefano de Sabbata. see [Online Book](#)
- The Grammar Of Graphics – All You Need to Know About ggplot2 and Pokemons by Pascal Schmidt. see [Online Tutorial](#)
- ggplot2 - Overview. see [Online Documentation](#)

- Getting started with httr - httr quickstart guide. [Online Tutorial](#)
- *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](#) and [repository](#).
- *R for Data Science* by Garrett Grolemund and Hadley Wickham, O'Reilly Media, 2016. See [online book](#).
- *Discovering Statistics Using R* by Andy Field, Jeremy Miles and Zoë Field, SAGE Publications Ltd, 2012. See book [webpage](#).
- *Machine Learning with R: Expert techniques for predictive modeling* by Brett Lantz, Packt Publishing, 2019. See book [webpage](#).
- *The Art of R Programming: A Tour of Statistical Software Design* by Norman Matloff, No Starch Press, 2011. See book [webpage](#)
- *An Introduction to R for Spatial Analysis and Mapping* by Chris Brunsdon and Lex Comber, Sage, 2015. See book [webpage](#)
- *Geocomputation with R* by Robin Lovelace, Jakub Nowosad, Jannes Muenchow, CRC Press, 2019. See [online book](#).
- *The RStudio Cheatsheets* - [Collection on R Studio Website](#).

2 Core concepts

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

- 1) Variables
- 2) Functions
- 3) Libraries

2.1 Variables

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

```
a_variable <- 1
```

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

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

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

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

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

```
a_variable <- 1
```

manipulate the value of the variable as

```
a_variable <- a_variable + 10
```

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

```
another_variable <- a_variable
```

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

Let us consider the following example:

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

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

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

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

See solution!

```
temp_A <- 24
```

```
temp_B <- 28
```

```
diff <- temp_A - temp_B
```

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

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

2.2 Algorithms and functions

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

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

The definition of an algorithm (and thus a program) can consist of one or more **functions**.

Functions are a set of instructions that perform a task, i.e. functions help structuring code into functional units. These functional units are reusable in the code. Some of them receive values as inputs, some return output values.

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

For instance, the pre-defined function `'sqrt()'` calculates the square root of an input value.

`'sqrt()'` (as every function in R) is invoked by specifying the *function name* and the *arguments* (input values) between simple brackets:

```
sqrt(2)
## [1] 1.414214
```

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

'round()' is another function that is predefined in R:

```
round(1.414214, digits = 2)
## [1] 1.41
```

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

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

```
sqrt_of_two <- sqrt(2)
sqrt_of_two
## [1] 1.414214
```

The output value is stored in the memory slot with the identifier 'sqrt_of_two'. We can use the identifier 'sqrt_of_two' as an argument in other functions as

```
sqrt_of_two <- sqrt(2)
round(sqrt_of_two, digits = 3)
## [1] 1.414
```

The first line calculates the square root of '2' and stores it in a variable with identifier 'sqrt_of_two'. The second line rounds the value stored in 'sqrt_of_two' to three digits after the dot.

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

See solution!

```
sqrt_of_two <- sqrt(2)
rounded_sqrt_of_two <- round(sqrt_of_two, digits = 3)
```


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

```
round(sqrt(2), digits = 3)
## [1] 1.414
```

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

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

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

- R is a **case sensitive** language
 - UPPER and lower case are not the same
 - `a_variable` is different from `a_VARIABLE`
- names can include
 - alphanumeric symbols
 - `.` and `_`
- names must start with
 - a letter

2.3 Libraries

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

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

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

- `base`: base R functions, including the `sqrt` function above
- `rgdal`: implementation of the [GDAL \(Geospatial Data Abstraction Library\)](#) functionalities.

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

```
library(stringr)
```

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

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

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

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

```
str_length("UNIGIS")
```

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

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

```
str_detect("UNIGIS", "I")
```

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

The function ‘`str_replace_all`’ replaces all the instances of the first argument that are identical with the second argument by a third argument:

```
str_replace_all("UNIGIS", "I", 'X')
```

```
## [1] "UNXGXS"
```

You may list all the functions available with library 'stringr' using the built in function 'ls()':

```
ls("package:stringr")
```

3 Complex Data Types

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

In this lesson, you will get to know the following complex data types:

- 1) Vectors
- 2) Matrices and Arrays
- 3) Lists
- 4) Data Frames

3.1 Vectors

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

- numeric
- character
- logic

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

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

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

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

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

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

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

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

or 'rep()':

```
#create vector with 4 identical character string values
a_vector <- rep("Ciao", 4)
a_vector

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

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

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

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

3.1.1 Vector element selection

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

```
a_vector <- (3:8)
a_vector[3]

## [1] 5
```

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

```
a_vector <- (3:8)
a_vector[c(2, 4)]
```

```
## [1] 4 6
```

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

Now try by yourself. Create a vector that looks like

```
east_midlands_cities <- c("Derby", "Leicester", "Lincoln", "Nottingham")
```

, select the last three cities out of the four cities in 'east_midlands_cities' and assign the returned values to a new vector named 'selected_cities'.

See solution!

```
east_midlands_cities <- c("Derby", "Leicester", "Lincoln", "Nottingham")
```

```
my_indexes <- 2:4
```

```
selected_cities <- c(east_midlands_cities[my_indexes])
```

3.1.2 Functions on vectors

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

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

```
a_numeric_vector <- 1:5
a_numeric_vector <- a_numeric_vector + 10
a_numeric_vector
```

```
## [1] 11 12 13 14 15
```

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

```
a_numeric_vector <- 1:5  
a_numeric_vector <- sqrt(a_numeric_vector)  
a_numeric_vector  
  
## [1] 1.000000 1.414214 1.732051 2.000000 2.236068
```

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

```
a_numeric_vector <- 1:5  
a_numeric_vector <- a_numeric_vector >= 3  
a_numeric_vector  
  
## [1] FALSE FALSE TRUE TRUE TRUE
```

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

```
a_numeric_vector <- 1:5  
any(a_numeric_vector >= 3)  
  
## [1] TRUE
```

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

```
a_numeric_vector <- 1:5  
all(a_numeric_vector >= 3)  
  
## [1] FALSE
```

A **factor** is a data type similar to a vector. However, the values contained in a factor can only be selected from a set of **levels**. Factors will not be covered in the module. For more information on this data type turn to the online tutorial [Programming with R](https://UNIGIS.at/en)

3.2 Multi-dimensional data types

3.2.1 Matrices

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

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

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

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

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

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

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

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

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

z <- x*y
z
```



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

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

A more comprehensive list of matrix algebra operations is provided by [Quick-R](#).

3.2.2 Arrays

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

```
a3dim_array <- array(1:24, dim=c(4, 3, 2))
a3dim_array

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

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

3.2.3 Selection

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

In the example, below we are subsetting the second row and the first and second column of `a_matrix`:

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

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

a_matrix[2, c(1, 2)]

## [1] 5 3
```

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

```
an_array <- array(1:12, dim=c(3, 2, 2))
an_array

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

an_array[2, c(1, 2), 2]
```

```
## [1] 8 11
```

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

See solution!

```
a3dim_array <- array(1:24, dim=c(4, 3, 2))  
  
a3dim_array  
  
a_vector <- a3dim_array[3, c(1, 2), 2]  
  
a_vector  
  
a_matrix <- a3dim_array[c(3, 4), c(1, 2), 2]  
  
a_matrix
```

3.2.4 List

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

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

```
employee <- list("Christian", 2017)  
employee  
  
## [[1]]  
## [1] "Christian"  
##  
## [[2]]  
## [1] 2017
```

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

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

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

```
employee <- list(employee_name = "Christian", start_year = 2017)
employee

## $employee_name
## [1] "Christian"
##
## $start_year
## [1] 2017

employee$employee_name
## [1] "Christian"
```

3.2.5 Data Frame

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

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

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

```
employees

## EmployeeName Age    Role
## 1      Maria 47 Professor
```

```
## 2    Pete 34 Researcher  
## 3    Sarah 32 Researcher
```

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

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

See solution!

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

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

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

names(name of data frame)[column index as number] = "new column name"

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

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

```
employees[1, 1]  
## [1] "Maria"
```

We can also retrieve entire rows...

```
employees[1, ]
```

```
## EmployeeName Age   Role  
## 1      Maria 47 Professor
```

...and columns:

```
employees[,1]
```

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

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

```
employees$Age
```

```
## [1] 47 34 32
```

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

```
employees$Age[1]
```

```
## [1] 47
```

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

```
employees$Age[2] <- 33
```

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

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

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

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

```
current_year <- as.integer(format(Sys.Date(), "%Y"))
```

...use the column 'Age' in data frame 'employees' to calculate the year of birth for every employee...

```
current_year - employees$Age
```

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

See solution!

```
#Instantiate data frame employees
```

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

```
#Use Sys.Date to retrieve the current year
```

```
current_year <- as.integer(format(Sys.Date(), "%Y"))
```

```
#Calculate employee year of birth
```

```
employees$Year_of_birth <- current_year - employees$Age
```

```
employees
```

4 Control structures

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

We distinguish between two types of control structures:

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

4.1 If

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

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

```
a_value <- -7
if (a_value < 0) {
  cat("Negative")
}

## Negative
```

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

The function `cat()` concatenates and prints string inputs (“Negative” in the example above). Alternatively, you can use the function `print()` to write variable values to the console window.

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

Note that every conditional statement (e.g. `'a_value < 0'`) returns a logical value that is either TRUE or FALSE.

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

See solution!

The condition yields FALSE. The statement is not executed.

4.2 Else

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

```
a_value <- -7
if (a_value < 0){
  cat("Negative")
} else {
  cat("Positive")
}
```

Negative

In the example above, the **condition** `'a_value < 0'` is TRUE, **statement 1** `'cat("Negative")'` gets executed and **statement 2** `'cat("Positive")'` is ignored. If you change `'a_value'` to a positive value, the interpreter will ignore statement 1 and execute statement 2.

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

4.3 Code blocks

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

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

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

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

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

4.4 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`

4.4.1 While and repeat

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

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

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

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

See solution!

1. The variable 'current_value' takes on a value of zero.
2. The condition of the while-loop returns `TRUE`.

3. The 'cat()' function is executed and prints a text as well as 'current_value'.
4. The variable 'current_value' is incremented by +1.
5. The condition of the while-loop returns TRUE (current_value = 1), the code block is executed again (see 3 and 4).
6. current_value = 2, the code block is executed again (see 3 and 4).
7. current_value = 3, the condition returns FALSE, the loops ends.

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

```
current_value <- 0
repeat {
  cat("Current value is", current_value, "\n")
  current_value = current_value + 1
  if (current_value == 3){           #if (variable == 10)...
    break                          #the loop will break!
  }
}

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

The break statement is executed and stops (breaks) the repeat loop (also applicable to while or for loops) once the variable `current_value` is equal to ten.

4.4.2 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")
}

## Do you live in Derby ?
## Do you live in Leicester ?
```

```
## Do you live in Lincoln ?  
## Do you live in Nottingham ?
```

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

The code block below illustrates another example.

```
cities <- c("Derby", "Leicester", "Lincoln", "Nottingham")  
letter_cnt <- c()  
for (city in cities) {  
  letter_cnt <- c(letter_cnt, nchar(city))  
}  
print(letter_cnt)  
## [1] 5 9 7 10
```

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

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

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

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

```
for (i in 1:3) {  
  cat("This is exectuion number", i, ":\n")  
  cat("    See you later!\n")  
}  
## This is exectuion number 1 :  
##    See you later!  
## This is exectuion number 2 :  
##    See you later!
```

```
## This is execution 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.

4.5 Loops with conditional statements

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

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

```
#Example: countdown!  
for (i in 3:0) {  
  if (i == 0) {  
    cat("Go!\n")  
  } else {  
    cat(i, "\n")  
  }  
}  
  
## 3  
## 2  
## 1  
## Go!
```

The deterministic loop runs 4 times 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](#)).

‘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 Functions

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

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

5.1 Defining functions

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

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

```
add_one <- function (input_value) {  
  output_value <- input_value + 1  
  output_value  
}
```

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

```
add_one(3)
```



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

5.2 More parameters

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

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

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

```
area_rectangle <- function (height, width) {  
  area <- height * width  
  area  
}  
area_rectangle(3, 2)  
## [1] 6
```

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

5.3 More return values

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

```
rectangle_metrics <- function (height, width) {  
  area <- height * width  
  perimeter <- 2*height + 2*width  
  
  return_vals <- list(area, perimeter)  
  
  return_vals  
}
```

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

```
cat("This is the first return value - area: ", rectangle_metrics(3, 2)[[1]])  
## This is the first return value - area: 6  
  
cat("This is the second return value - perimeter: ", rectangle_metrics(3,  
2)[[2]])  
## This is the second return value - perimeter: 10
```

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

5.4 Functions and control structures

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

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

```
factorial <- function (input_value) {  
  result <- 1  
  for (i in 1:input_value) {  
    cat("current:", result, " | i:", i, "\n")
```

```
    result <- result * i
  }
  result
}
factorial(3)

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

## [1] 6
```

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

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

5.5 Scope

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

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

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

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

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

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

Let us take an example. In the case below, `x_value` is **global** to the function `times_x`. `new_value` and `input_value` are **local** to the function `times_x`. Referring to `new_value` or `input_value` from outside the definition of `times_x` would result in an error. However, we can refer to `x_value` from inside function `times_x`:

```
x_value <- 10
times_x <- function (input_value) {
  new_value <- input_value * x_value
  new_value
}
times_x(2)

## [1] 20
```

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

See solution!

```
times_x <- function (input_value, x_value = 10) {

  new_value <- input_value * x_value

  new_value

}
```

The lessons so far have covered some fundamental concepts of R programming. The [Base R Cheatsheet](#) contains a concise summary of most important operations at a glance.

6 Data wrangling

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

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

You will also get to know [tibbles](#) as a new complex data type. Tibbles are basically a lightweight version of data frames (see [Tibbles in R for Data Science](#)).

6.1 Preparation

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

The code below, loads a sample dataset (a [tibble](#)) from the library `nycflights13` into the variable `flights_from_nyc`. We will use this sample data in this lesson.

```
library(nycflights13)
flights_from_nyc <- nycflights13::flights
```

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

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

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

Alternatively, you may inspect `flights_from_nyc` by writing it to the console.

6.2 Data manipulation

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

```
library(tidyverse)
library(knitr)

flights_from_nyc %>%
  dplyr::count() %>%
  knitr::kable()

```

	n
	336776

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

The pipe operator allows us to link a sequence of analysis steps. In the example above, `flights_from_nyc` is passed into function `count()` and the output is passed into function `kable()` to render the result_df

The pipe operator is a powerful tool to simplify your code. See [this video](#) to learn more about it.

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

In the example below, the column name `origin` is provided as an argument to the function `count()`, so rows representing flights from the same origin are counted together – EWR is the

Newark Liberty International Airport, JFK is the John F. Kennedy International Airport, and LGA is LaGuardia Airport.

```
flights_from_nyc %>%  
  dplyr::count(origin) %>%  
  knitr::kable()
```

origin	n
EWR	120835
JFK	111279
LGA	104662

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

6.2.1 Summarise

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

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

```
flights_from_nyc %>%  
  dplyr::group_by(origin) %>%  
  dplyr::summarise(  
    mean_distance_traveled_from = mean(distance)  
  ) %>%  
  knitr::kable()
```

origin	mean_distance_traveled_from
EWR	1056.7428
JFK	1266.2491
LGA	779.8357

6.2.2 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.

6.2.3 Mutate

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

```
flights_from_nyc %>%  
  dplyr::select(flight, origin, dest, air_time) %>%  
  dplyr::mutate(  
    air_time_hours = air_time / 60  
  ) %>%  
  dplyr::slice_head(n = 5) %>%  
  knitr::kable()
```

flight	origin	dest	air_time	air_time_hours
1545	EWR	IAH	227	3.783333
1714	LGA	IAH	227	3.783333
1141	JFK	MIA	160	2.666667
725	JFK	BQN	183	3.050000
461	LGA	ATL	116	1.933333

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

See solution!

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

6.2.4 Arrange

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

```
flights_from_nyc %>%  
  dplyr::select(origin, dest, air_time) %>%  
  dplyr::arrange(-air_time) %>%  
  dplyr::slice_head(n = 5) %>%  
  knitr::kable()
```

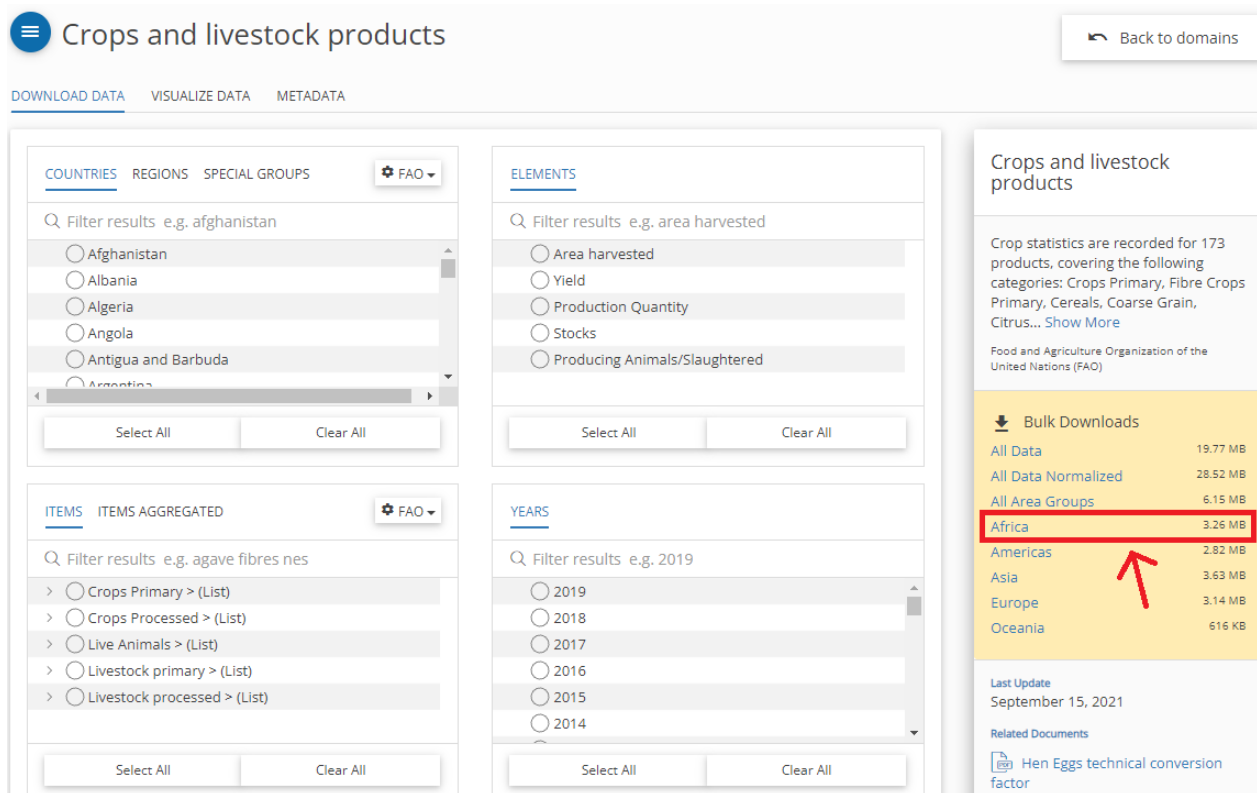
origin	dest	air_time
EWR	HNL	695
JFK	HNL	691
JFK	HNL	686
JFK	HNL	686
JFK	HNL	683

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

6.2.5 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](#) 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](#) data as CSV:



Crops and livestock products

Back to domains

DOWNLOAD DATA | VISUALIZE DATA | METADATA

COUNTRIES | REGIONS | SPECIAL GROUPS |  **FAO**

Filter results e.g. afghanistan

- ☐ Afghanistan
- ☐ Albania
- ☐ Algeria
- ☐ Angola
- ☐ Antigua and Barbuda
- ☐ Argentina


Select All | Clear All

ELEMENTS

Filter results e.g. area harvested

- ☐ Area harvested
- ☐ Yield
- ☐ Production Quantity
- ☐ Stocks
- ☐ Producing Animals/Slaughtered

Select All | Clear All

ITEMS | ITEMS AGGREGATED |  **FAO**

Filter results e.g. agave fibres nes

- ☐ Crops Primary > (List)
- ☐ Crops Processed > (List)
- ☐ Live Animals > (List)
- ☐ Livestock primary > (List)
- ☐ Livestock processed > (List)

Select All | Clear All

YEARS

Filter results e.g. 2019

- ☐ 2019
- ☐ 2018
- ☐ 2017
- ☐ 2016
- ☐ 2015
- ☐ 2014

Select All | Clear All

Crops and livestock products

Crop statistics are recorded for 173 products, covering the following categories: Crops Primary, Fibre Crops Primary, Cereals, Coarse Grain, Citrus... [Show More](#)

Food and Agriculture Organization of the United Nations (FAO)

Bulk Downloads

All Data	19.77 MB
All Data Normalized	28.52 MB
All Area Groups	6.15 MB
Africa	3.26 MB
Americas	2.82 MB
Asia	3.63 MB
Europe	3.14 MB
Oceania	616 KB

Last Update
September 15, 2021

Related Documents


 [Hen Eggs technical conversion factor](#)

Figure 6.1: FAO Data Download

- 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 = ",")
```

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

See [my solution!](#)

6.3 Join

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

```
city_telephone_prefix <- data.frame(
  city = c("Leicester", "Birmingham", "London"),
  telephone_prefix = c("0116", "0121", "0171")
) %>%
  tibble::as_tibble()
```

```
city_telephone_prefix %>%
  knitr::kable()
```

city	telephon_prefix
Leicester	0116
Birmingham	0121
London	0171

```
city_info_wide <- data.frame(
  city = c("Leicester", "Nottingham"),
  population = c(329839, 321500),
  area = c(73.3, 74.6),
  density = c(4500, 4412)
) %>%
  tibble::as_tibble()
```

```
city_info_wide %>%
  knitr::kable()
```

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

Note that data frames in the code above are converted to [tibbles](#). This step is needed as the function `kable()` takes tibbles as an input.

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

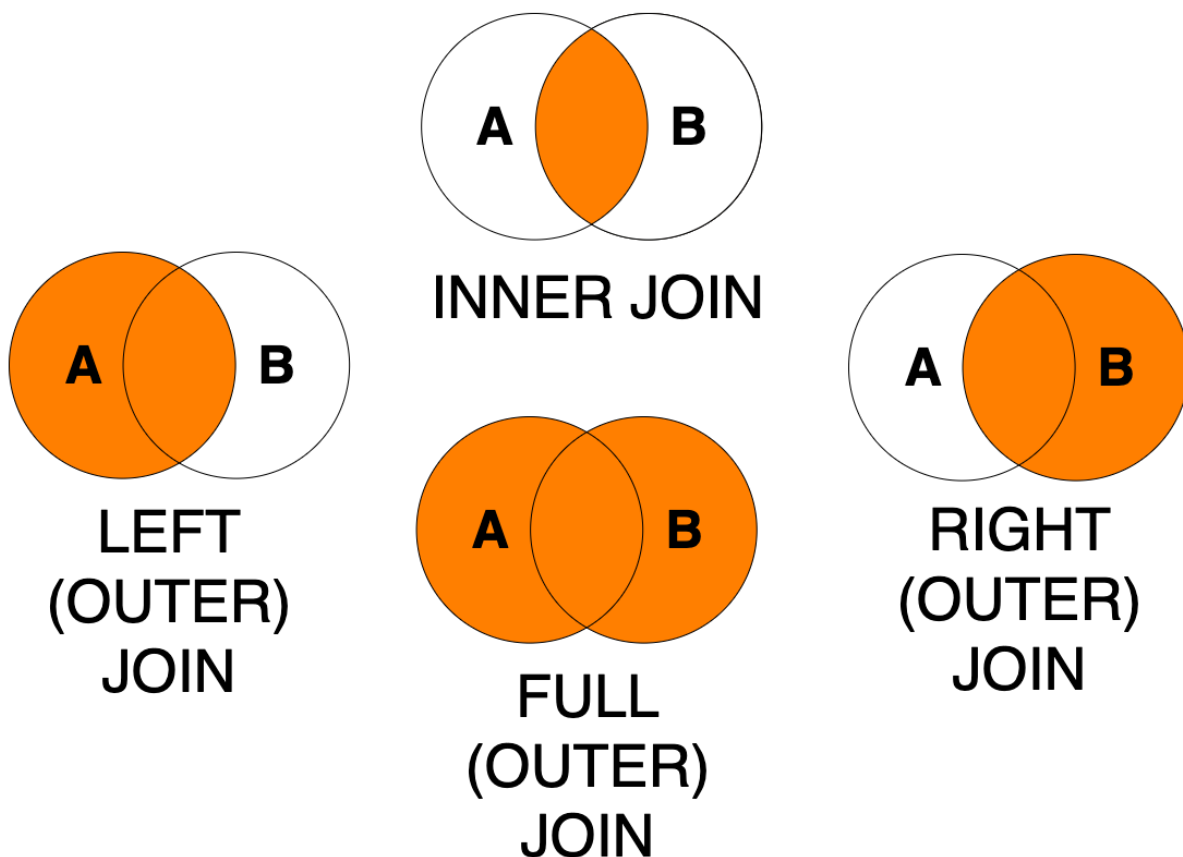


Figure 6.2: Join types

Please take your time to understand the examples below and check out the [related dplyr help pages](#) before continuing. The first four examples execute the exact same *full join* operation using three different syntaxes: with or without using the pipe operator and specifying the `by` argument or not. Note that all those approaches to writing the join are valid and produce the same result. The choice about which approach to use will depend on the code you are writing. In particular, you might find it useful to use the syntax that uses the pipe operator when the join operation is itself only one stem in a series of data manipulation steps. Using the `by` argument is

usually advisable unless you are certain that you aim to join two tables with all and exactly the column that have the same names in the two table.

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

Option 1: without using the pipe operator

full join verb

```
dplyr::full_join(
  # left table
  city_info_wide,
  # right table
  city_telephone_prexix,
  # columns to match
  by = c("city" = "city")
) %>%
knitr::kable()
```

city	population	area	density	telephon_prefix
Leicester	329839	73.3	4500	0116
Nottingham	321500	74.6	4412	NA
Birmingham	NA	NA	NA	0121
London	NA	NA	NA	0171

Option 2: without using the pipe operator

and without using the argument "by"

as columns have the same name

in the two tables.

Same result as Option 1

full join verb

```
dplyr::full_join(
  # left table
  city_info_wide,
  # right table
  city_telephone_prexix
) %>%
knitr::kable()
```

city	population	area	density	telephon_prefix
Leicester	329839	73.3	4500	0116
Nottingham	321500	74.6	4412	NA
Birmingham	NA	NA	NA	0121
London	NA	NA	NA	0171


```

# Option 3: using the pipe operator
#   and without using the argument "by"
#   as columns have the same name
#   in the two tables.
# Same result as Option 1 and 2
# Left table
city_info_wide %>%
  # full join verb
  dplyr::full_join(
    # right table
    city_telephone_prexix
  ) %>%
  knitr::kable()

```

city	population	area	density	telephon_prefix
Leicester	329839	73.3	4500	0116
Nottingham	321500	74.6	4412	NA
Birmingham	NA	NA	NA	0121
London	NA	NA	NA	0171

```

# Option 4: using the pipe operator
#   and using the argument "by".
# Same result as Option 1, 2 and 3
# Left table
city_info_wide %>%
  # full join verb
  dplyr::full_join(
    # right table
    city_telephone_prexix,
    # columns to match
    by = c("city" = "city")
  ) %>%
  knitr::kable()

```

city	population	area	density	telephon_prefix
Leicester	329839	73.3	4500	0116
Nottingham	321500	74.6	4412	NA
Birmingham	NA	NA	NA	0121
London	NA	NA	NA	0171

```

# Left join
# Using syntax similar to Option 1 above
# left join
dplyr::left_join(
  # left table
  city_info_wide,
  # right table
  city_telephone_prexix,
  # columns to match
  by = c("city" = "city")
) %>%
knitr::kable()

```

city	population	area	density	telephon_prefix
Leicester	329839	73.3	4500	0116
Nottingham	321500	74.6	4412	NA

```

# Right join
# Using syntax similar to Option 2 above
# right join verb
dplyr::right_join(
  # left table
  city_info_wide,
  # right table
  city_telephone_prexix
) %>%
knitr::kable()

```

city	population	area	density	telephon_prefix
Leicester	329839	73.3	4500	0116
Birmingham	NA	NA	NA	0121
London	NA	NA	NA	0171

```
# Inner join
# Using syntax similar to Option 3 above
# left table
city_info_wide %>%
  # inner join
  dplyr::inner_join(
    # right table
    city_telephone_prefix
  ) %>%
  knitr::kable()
```

city	population	area	density	telephon_prefix
Leicester	329839	73.3	4500	0116

6.3.1 Exercise: join

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

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

```
world_ctype <- map_data("world")
```

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

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

Take a look at the [dplyr Cheatsheet](#) which shows the most important dplyr operations at a glance.

7 Read and write data

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

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

7.1 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 (also part of `Tidyverse`) 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](#) and queries Norwegian entries. The function `writes_csv` writes these entries to a new CSV file.

```
library(tidyverse)

fishery_data <- readr::read_csv("data/capture-fisheries-vs-aquaculture.csv")
#print(fishery_data)
```

```
#print(typeof(fishery_data$))

fishery_data %>%
  dplyr::filter(Entity == "Norway") %>%
  readr::write_csv("data/capture-fisheries-vs-aquaculture-noraway.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](#). Then copy and run the code in a new R-script.

Other important packages for reading tabular data are [readxl](#) for Excel (.xls and .xlsx) and [haven](#) for SPSS, Stata and SAS data.

7.2 Read and write vector data

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

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

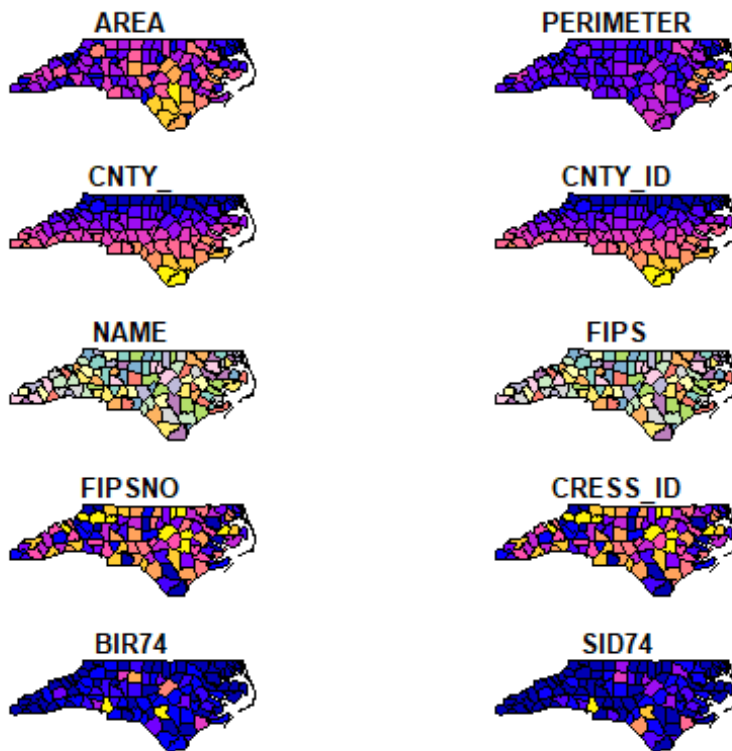
In order to load vector data in an R-Script, we can use the function `st_read()`. In the code block below, a [shapefile \(North Carolina sample data\)](#) is loaded and assigned to a variable `nc`.

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

```
library(sf)

nc <- sf::st_read("data/nc.shp")

plot(nc)
```



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

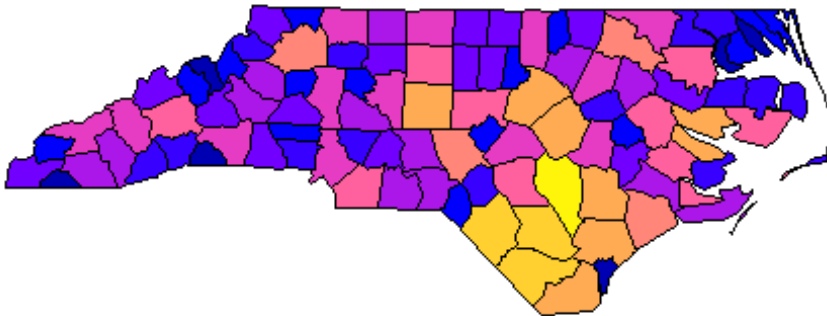
```
nc %>%
  dplyr::select(AREA, NAME, geometry) %>%
  dplyr::slice_head(n = 3) %>%
  knitr::kable()
```

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

`sf` also includes a number of operations to manipulate the geometry of features such as `st_simplify`:

```
sf::st_simplify(nc) %>%  
plot(., max.plot = 1)
```

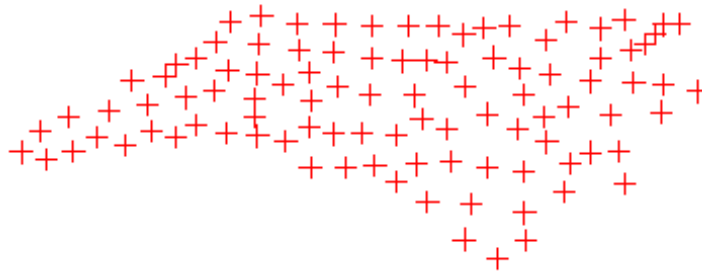
AREA



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](#) offers a more comprehensive explanation of available geometric, attribute and spatial data operations. For a quick overview, you may turn to the [sf cheatsheets](#).

In order to test the code on your machine, [download](#) 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](#) for more information.

See [my solution](#)!

Similar R functions are also available for raster data (see package [raster: Geographic Data Analysis and Modeling](#))

7.3 Data API

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

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

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

Figure 7.1: Video (3:13 min): REST API concepts and examples.

The most important take away messages are:

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

The [library httptr](#) (also part of [Tidyverse](#)) 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](#) offered by [OpenWeather Map](#) that contain historical and real-time weather data for retrieval. In the upcoming example we will make a call to the [Current Weather API](#), which is one out of currently [10 free APIs](#) provided by OpenWeather Map.

For accessing the data, we need to create a URL that is composed of a reference to the data source (base) and parameters to filter the desired data subset (lat and lon). The parameters are concatenated by means of a '&' symbol, a question mark '?' needs to be placed between reference URL and parameters:

```
library(httptr)
```

```
base <- "http://api.openweathermap.org/data/2.5/weather"  
lat <- "47.81"  
lon <- "13.03"  
apiKey <- "3f87141421b32590d50416aae5ca780c"
```

```
call <- paste(base, "?lat=", lat, "&lon=" , lon, "&appid=", apiKey, sep="")
print(call)

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

get <- GET(call)
```

In the code above, the function `paste()` assembles base URL and parameters. The URL (of type string) is then passed as an argument to function `GET()` that executes the HTTP Get method.

Note: The OpenWeather API requires an API Key to be passed as a parameter in the call. Get your [personal Key](#) 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](#), it may take a couple of hours until your key is active.

Per default, the `GET()` function returns a response object. Printing a response object gives you some useful information: the actual url used (after any redirects), the http status, the file (content) type, the size, and if it's a text file, the first few lines of output.

You can pull out important parts of the response with various helper functions such as `status_code()` and `content()`:

```
status_code(get)

## [1] 200

str(content(get))

## List of 13
## $ coord   :List of 2
## ..$ lon: num 13
## ..$ lat: num 47.8
## $ weather :List of 1
## ..$ :List of 4
## ...$ id    : int 801
## ...$ main   : chr "Clouds"
```

```
## ..$ description: chr "few clouds"
## ..$ icon      : chr "02d"
## $ base       : chr "stations"
## $ main       :List of 6
## ..$ temp      : num 299
## ..$ feels_like: num 299
## ..$ temp_min   : num 299
## ..$ temp_max   : num 301
## ..$ pressure   :int 1024
## ..$ humidity   :int 36
## $ visibility: int 10000
## $ wind        :List of 2
## ..$ speed: num 5.66
## ..$ deg  :int 310
## $ clouds      :List of 1
## ..$ all: int 20
## $ dt          :int 1655474391
## $ sys         :List of 5
## ..$ type  :int 1
## ..$ id   :int 6877
## ..$ country: chr "AT"
## ..$ sunrise: int 1655435320
## ..$ sunset : int 1655492933
## $ 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](#) 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).

In order to facilitate subsequent analyses and data visualization, we can convert the content of the return object to a data frame by means of function `fromJSON()` that is part of the library `library(jsonlite)`:

```
library(jsonlite)

get_text <- content(get, "text")           #retrieve contents of
request as character vector (library httr)
get_json <- fromJSON(get_text, flatten = TRUE) #convert from JSON to R
Object (library jsonlite)
get_df <- as.data.frame(get_json)          #convert R Object to Data
Frame
```

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

[Tidyverse](#) also provides other packages for reading data such as [DBI](#) for relational databases [jsonlite](#) for JSON and [xml2](#) for XML.

8 Data visualization

R has a very rich set of graphical functions. The [R Graph Gallery](#) provides a large number of examples (including code).

In this lesson you will get to know the [ggplot2 library](#), 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](#).

8.1 The Grammar of Graphics

The [ggplot2 library](#) is part of [Tidyverse](#) and offers a series of functions for creating graphics declaratively, based on the concepts outlined in the Grammar of Graphics by [Leland Wilkinson](#).

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

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

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

Visual variables are:

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

For instance, in Figure 8.1 variables ‘Gdp per capita’ and ‘Life Expectancy’ are mapped onto the x and y axes (visual variable **position**), variables ‘national population’ and ‘world regions’ are mapped onto visual variables **size** and **color**.

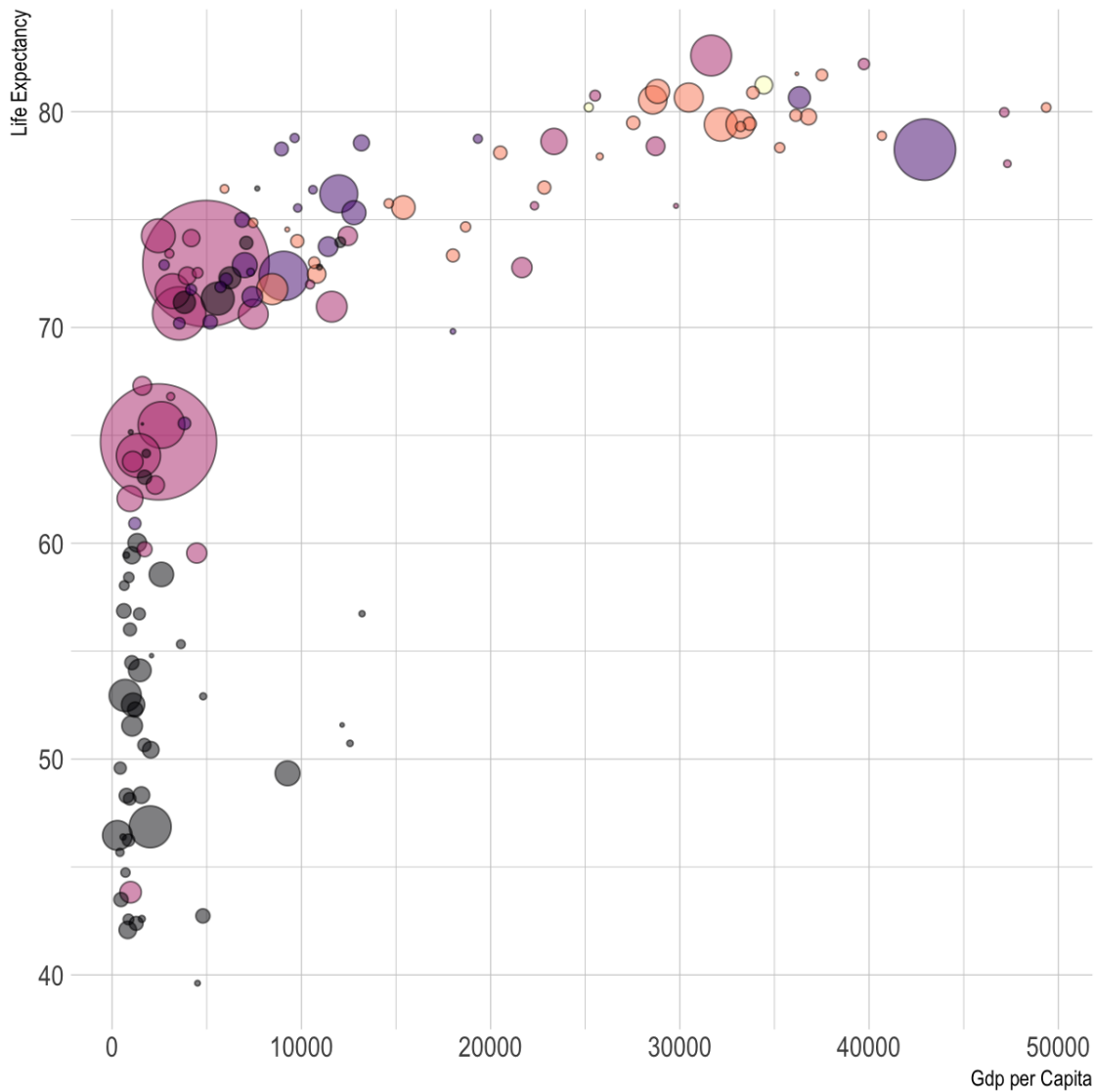


Figure 8.1: Visual variables color and size

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

The basic concept behind the grammar of graphics is described in an [article](#) by Hadley Wickham.

8.2 Visualization of distributions

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

We start the analysis with a simple histogram, to explore the distribution of air quality `data` that has been measured at different locations in Upper Austria.

The data includes the following **variables**

- **time** of measurement
- ID of the measuring **station**
- measured meteorological **component**
- **meantype**
- **unit** of measurement
- measurement **value**

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

```
library(tidyverse)
library(knitr)

#read csv data, Note: Semicolon seperated CSVs can be loaded by function
'read_delim()'
airquality <- read_delim("data/AirQualityUpperAut.csv", delim = ";")

airquality %>%
  dplyr::slice_head(n = 5) %>%
  knitr::kable()
```

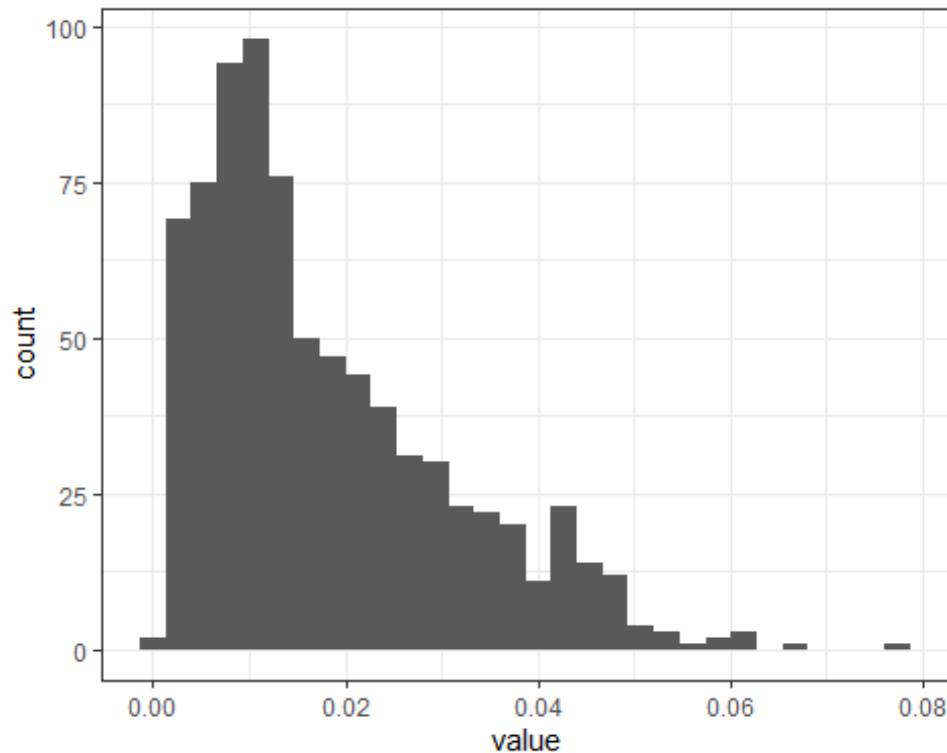
time	station	component	meantype	unit	value
21.10.2021 13:30	C001	BOE	HMW	m/s	14.1

21.10.2021 14:00	C001	BOE	HMW	m/s	12.0
21.10.2021 14:30	C001	BOE	HMW	m/s	10.1
21.10.2021 15:00	C001	BOE	HMW	m/s	7.9
21.10.2021 15:30	C001	BOE	HMW	m/s	9.2

The code below filters the airquality dataset by measurement component and temporal resolution. Then the data subset is passed as a first argument to function `ggplot()`. In the second argument, we map the variable `value` onto the x-axis with the aesthetics argument `aes()`. `geom_histogram()` specifies the geometry of the plot and `theme_bw()` is used to add a [background theme](#).

```
#filter NO2 measurements with temporal resolution 30min (HMW)
airquality %>%
  dplyr::filter(component == "NO2" & meantype == "HMW") %>%

  #create plot
  ggplot2::ggplot(.,      #the dot '.' represents the piped value
    aes(
      x = value           #map variable 'value' onto x-axis
    )
  ) +
  ggplot2::geom_histogram() + #define geometry
  ggplot2::theme_bw()        #define theme
```

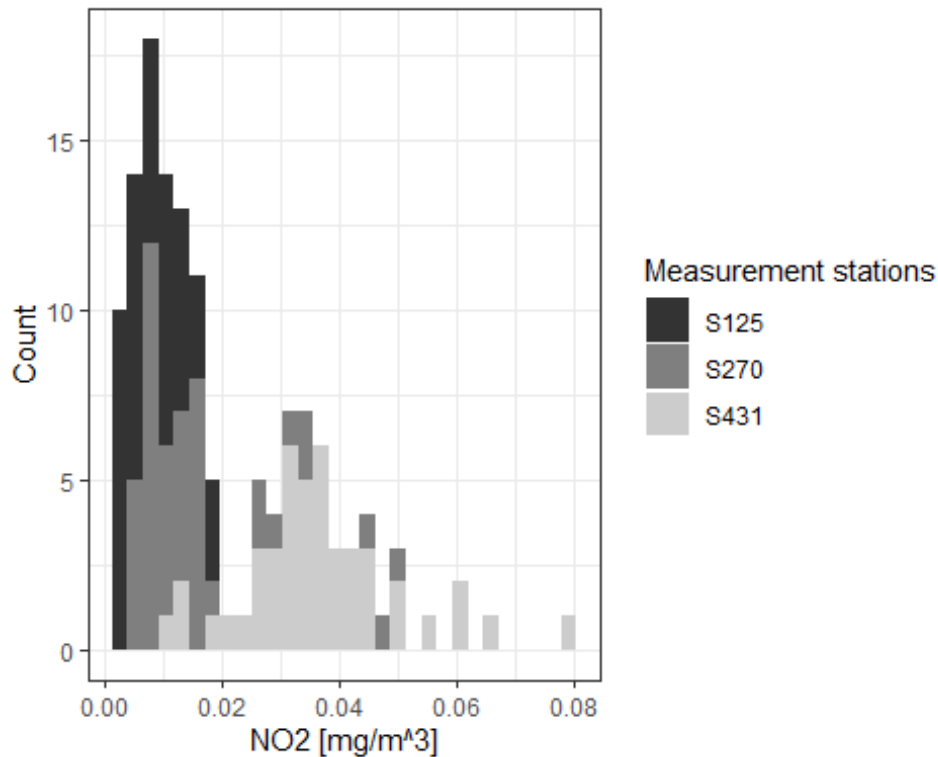


If we aim to distinguish between measurements of respective measurement stations, we can map the variable 'station' onto visual variable color:

```
airquality %>%
  dplyr::filter(component == "NO2" & meantype == "HMW") %>%
  dplyr::filter(station == "S125" | station == "S431" | station == "S270")
%>% #select 3 stations

ggplot2::ggplot(.,
  aes(
    x = value,
    fill = station
  )
) +

ggplot2::xlab("NO2 [mg/m^3]") +
ggplot2::ylab("Count") +
scale_fill_manual(name = "Measurement stations", values = c("grey20",
"grey50", "grey80")) +
ggplot2::geom_histogram() +
ggplot2::theme_bw()
```



This is implemented by adding an attribute `fill = station` to the aesthetics element (`aes()`). `ggplot2` offers a number of [functions](#) 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](#) `grey20`, `grey50` and `grey80`. Instead of `ggplot` colors, you can also use [hex color codes](#).

Note that plot components are [added](#) 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](#), recreate the histogram and insert one additional line of code to add a plot title (see [documentation](#)).

See solution!

Insert title:

```
ggplot2::ggtitle("Nitrogen dioxide concentration")
```

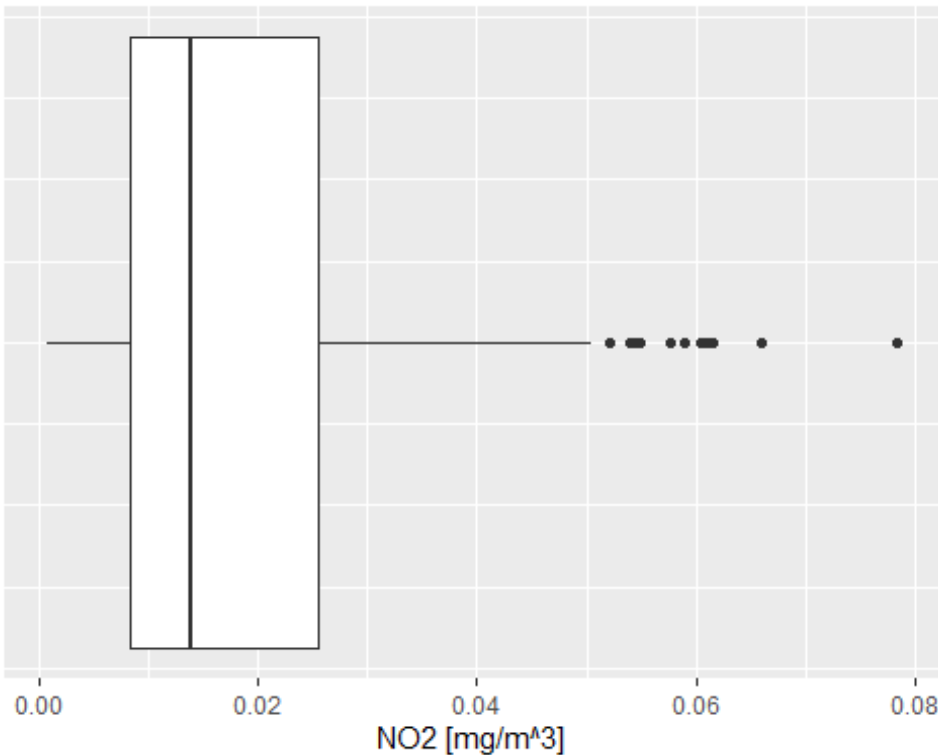
8.3 Boxplots

The same basic syntax is used to create other types of plots like bar plots (use geometry `geom_bar()` or `geom_col()`), line plots (use geometry `geom_line()`) and many others.

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](#), 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](#). 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 $1.5 \cdot \text{IQR}$ 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 $1.5 \cdot \text{IQR}$ of the hinge. Data beyond the end of the whiskers are called “outlying” points and are plotted individually.

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

See [my solution!](#)

8.4 Scatterplots

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

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

```
#half-hourly temperature measurement of station S108 to data frame
temp_tab <- airquality %>%
  dplyr::filter(component == "TEMP" & meantype == "HMW" & station == "S108")

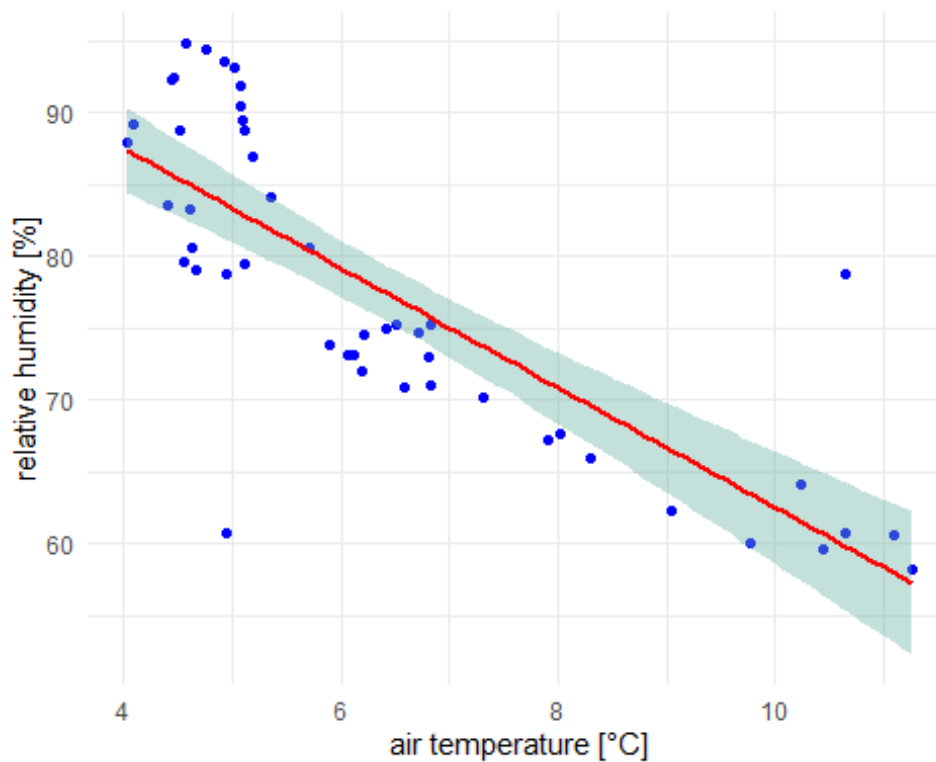
#half-hourly relative humidity measurement of station S108 to data frame
humidity_tab <- airquality %>%
  dplyr::filter(component == "RF" & meantype == "HMW" & station == "S108")

#join humidity and temperature tables by common field 'time'
temp_tab %>%
  dplyr::inner_join(
    # right table
    humidity_tab,
    # columns to match
    by = c("time" = "time")
  ) %>%

  dplyr::select(time, value.x, value.y) %>% #select relevant columns from
joined table

#create plot
ggplot2::ggplot(.,
  aes(
    x = value.x,
```

```
y = value.y
)
) +
ggplot2::xlab("air temperature [°C]") +
ggplot2::ylab("relative humidity [%]") +
ggplot2::geom_point(color="blue") +
#define geometry scatterplot, with point color blue
ggplot2::geom_smooth(method=lm , color="red", fill="#69b3a2", se=TRUE) +
#with linear trend and confidence interval
ggplot2::theme_minimal()
```



The plot reveals a trend between the two variables temperature and humidity. Relative humidity tends to increase as temperature decreases and vice versa. In [this video](#) you can find an explanation for the inverse proportional relationship between relative humidity and air temperature.

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

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

Complete Script!

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

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

See answers!

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

8.5 Map visualization

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

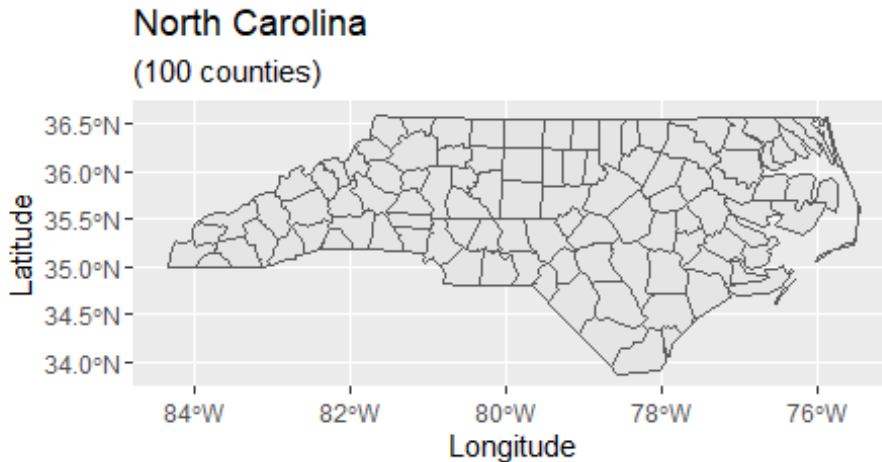
In order to replicate the code examples below, you will have to install and load the libraries `sf` (remember: `sf` stands for simple features and is used to read and write vector data) and `ggplot()`. Also download the [North Carolina](#) and [US States](#) sample datasets.

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

```
library(sf)
library("ggplot2")

nc <- sf::st_read("data/nc.shp")
```

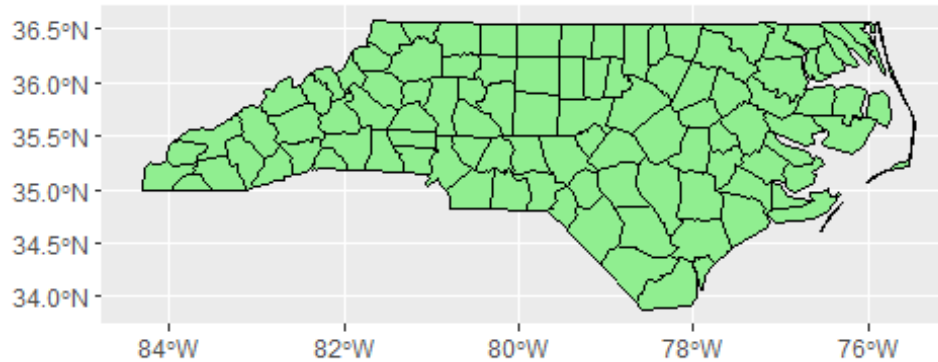
```
ggplot(data = nc)+
  geom_sf() +
  xlab("Longitude") + ylab("Latitude") +
  ggtitle("North Carolina", subtitle = paste0("(", length(unique(nc$NAME)),
" counties")))
```



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

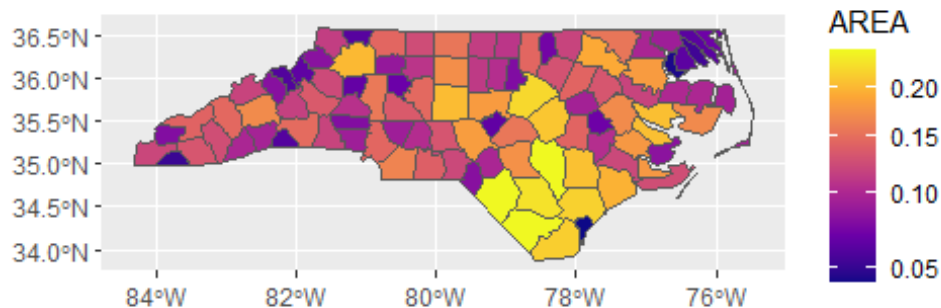
The geometry element `geom_sf` provides a number of arguments to customize the appearance of vector features:

```
ggplot(data = nc) +
  geom_sf(color = "black", fill = "lightgreen")
```



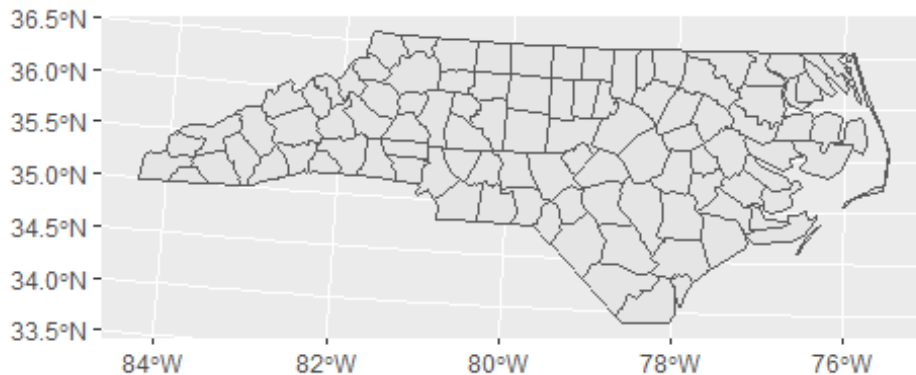
Data can also be mapped onto visual variables in the same way as with diagram plots. In the example below, the variable AREA is mapped onto visual variable fill color:

```
ggplot(data = nc) +  
  geom_sf(aes(fill = AREA)) +  
  scale_fill_viridis_c(option = "plasma", trans = "sqrt")
```



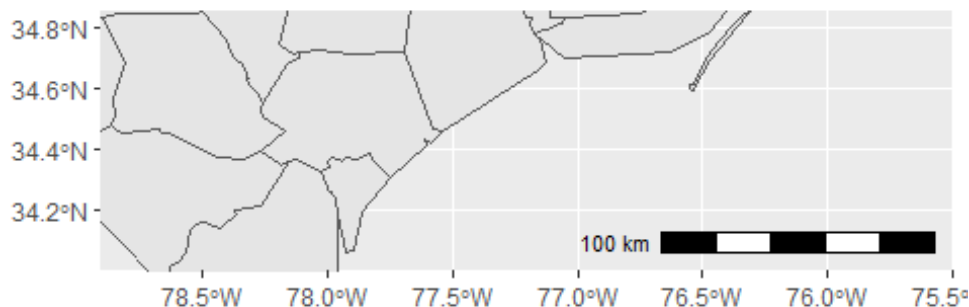
The function `coord_sf()` allows to deal with the coordinate system, which includes both projection and extent of the map. By default, the map will use the coordinate system of the first layer or if the layer has no coordinate system, fall back on the geographic coordinate system WGS84. Using the argument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

```
ggplot(data = nc) +  
  geom_sf() +  
  coord_sf(crs = st_crs(32618))
```



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

```
library("ggspatial")
ggplot(data = nc) +
  geom_sf() +
  coord_sf(xlim = c(-78.9, -75.5), ylim = c(34, 34.85), expand = FALSE) +
  annotation_scale(location = "br", width_hint = 0.5) +
  annotation_north_arrow(location = "bl", which_north = "true",
    pad_x = unit(14.5, "cm"), pad_y = unit(0.8, "cm"),
    style = north_arrow_fancy_orienteering)
```



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

In the following example, we will assign labels to vector features. The function `geom_text()` can be used to add a layer of text to a map using geographic coordinates. The North Carolina dataset contains county names as column (column: `NAME`). In order to define label positions, we take the centroids of the county polygons (function `st_centroid()`), derive X and Y

coordinates from centroids (function `st_coordinates()`), merge the new X and Y columns with the columns of `nc` and assign the output to a new variable identifier `nc_points`:

```
nc_points <- cbind(nc, st_coordinates(st_centroid(nc$geometry)))
```

I have used a standard syntax to create variable `nc_points`. Convert the code to pipe operator syntax.

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

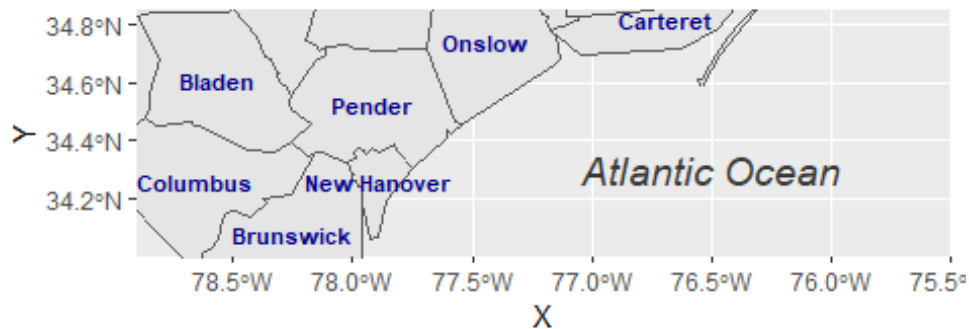
See solution!

```
st_centroid(nc$geometry) %>% st_coordinates() %>% cbind(nc, .)
```

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

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

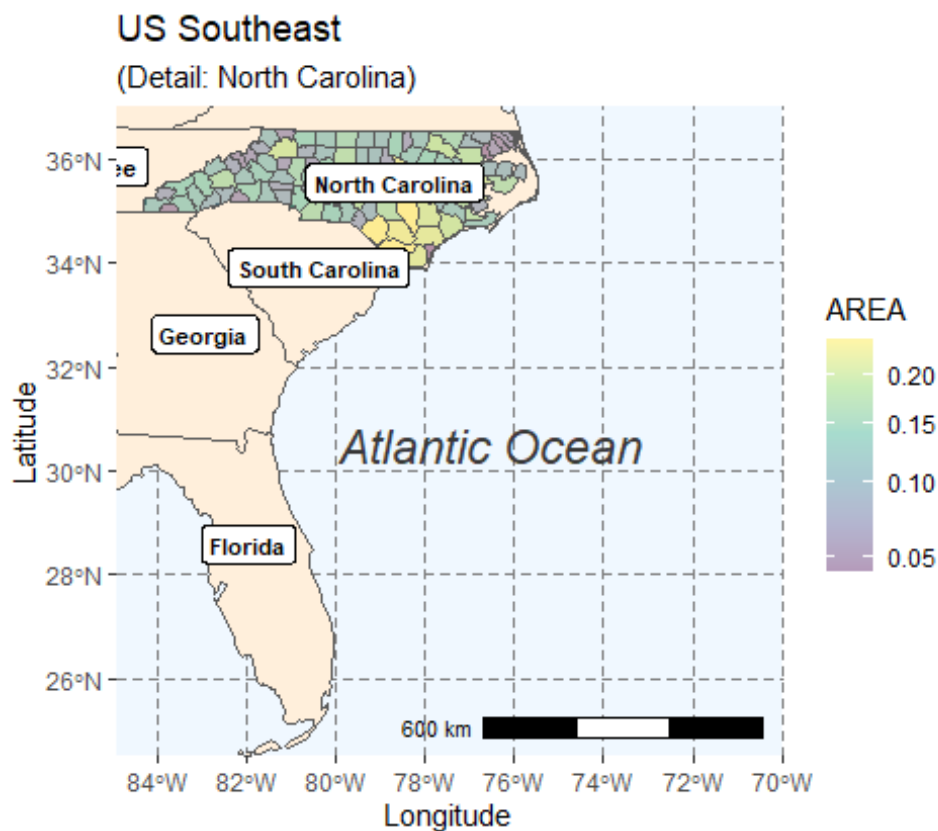
```
ggplot(data = nc) +
  geom_sf() +
  geom_text(data= nc_points, aes(x=X, y=Y, label=NAME),
    color = "darkblue", fontface = "bold", check_overlap = FALSE, size = 3) +
  annotate(geom = "text", x = -76.5, y = 34.3, label = "Atlantic Ocean",
    fontface = "italic", color = "grey22", size = 5) +
  coord_sf(xlim = c(-78.9, -75.5), ylim = c(34, 34.85), expand = FALSE)
```



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

```
us_states <- sf::st_read("data/us-states.shp")
us_states_points <- st_centroid(us_states)
us_states_points <- cbind(us_states,
  st_coordinates(st_centroid(us_states$geometry)))

ggplot(data = nc) +
  geom_sf(data = us_states, fill= "antiquewhite1") +
  geom_sf(aes(fill = AREA)) +
  geom_label(data= us_states_points, aes(x=X, y=Y, label=NAME),
    color = "black", fontface = "bold", check_overlap = FALSE, size =
3, nudge_x = 0.5) +
  annotation_scale(location = "br", width_hint = 0.5) +
  annotation_north_arrow(location = "bl", which_north = "true",
    pad_x = unit(11, "cm"), pad_y = unit(0.8, "cm"),
    style = north_arrow_fancy_orienteering) +
  scale_fill_viridis_c(trans = "sqrt", alpha = .4) +
  coord_sf(xlim = c(-84.9, -70), ylim = c(24.5, 37), expand = FALSE) +
  xlab("Longitude") + ylab("Latitude") +
  ggtitle("US Southeast", subtitle = "(Detail: North Carolina)") +
  annotate(geom = "text", x = -76.5, y = 30.5, label = "Atlantic Ocean",
    fontface = "italic", color = "grey22", size = 6) +
  theme(panel.grid.major = element_line(color = gray(0.5), linetype =
"dashed",
    size = 0.1), panel.background = element_rect(fill = "aliceblue"))
```



The function `geom_sf()` is used to add US state polygons as an additional layer. The function `geom_label()` is used as an alternative to function `geom_text()`. Note that `nudge_x` of function `geom_label()` is introduced to offset labels horizontally.

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

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

Take a look at the [ggplot Cheatsheet](#) which shows the most important ggplot operations at a glance.

8.6 Interactive Maps

The [Leaflet library for R](#) 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](#), [Flickr](#) or [OpenStreetMap](#).

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

```
library(leaflet)
m <- leaflet()
```

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

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

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

The function `addTiles()` per default adds OpenStreetMap map tiles. You may use the function `addProviderTiles()` to add other map tiles. Leaflet supports a large number of [basemap layers](#).

The same pipe-syntax can be used to add [Markers](#) and HTML [Labels](#) or [Popups](#). 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](#) and integrating [lines and shapes](#), [GeoJSON](#) and [Raster Images](#). To get more information on creating interactive maps with R and leaflet, turn to the [Documentation](#).