Application Development with R

Christian Neuwirth

27 August, 2021

Table of Contents

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 learn to perform **statistical analyses** (exploratory data analysis, comparing data, regression models) and how to **visualize results** 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 based on the teaching materials [granolarr](https://sdesabbata.github.io/granolarr/) worked out by [Stefano de Sabbata](https://stefanodesabbata.com/) at the [University of Leicester](https://le.ac.uk/). For more information take a look at the [Granolarr Lecture Pages](https://sdesabbata.github.io/granolarr/lectures_100/). The chapter **Machine Learning** is 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 avaialble for Windows, May 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 lession we will focus on some key principles of the R syntax and logic.

## 1.3 Installation and Setup

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

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

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

If you need help, 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. 1.1). When values and operations are inputted in the *Console*, the interpreter returns the results of its interpretation of the expression.



Figure 1.1: Console Window in RStudio

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

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

**See solution!**

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

If you start your input with a hash symbol (#) the interpreter will consider that line as a comment. For instance, if you type in *# comments are ignored*, 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 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

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

**See solution!**

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

## 1.7 Logical operators

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

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

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

What logical values are returned by the following expressions:

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

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

## 1.8 Reference books

Suggested reading

* *Programming Skills for Data Science: Start Writing Code to Wrangle, Analyze, and Visualize Data with R* by Michael Freeman and Joel Ross, Addison-Wesley, 2019. See book [webpage](https://www.pearson.com/us/higher-education/program/Freeman-Programming-Skills-for-Data-Science-Start-Writing-Code-to-Wrangle-Analyze-and-Visualize-Data-with-R/PGM2047488.html) and [repository](https://programming-for-data-science.github.io/).
* *R for Data Science* by Garrett Grolemund and Hadley Wickham, O’Reilly Media, 2016. See [online book](https://r4ds.had.co.nz/).
* *Discovering Statistics Using R* by Andy Field, Jeremy Miles and Zoë Field, SAGE Publications Ltd, 2012. See book [webpage](https://www.discoveringstatistics.com/books/discovering-statistics-using-r/).
* *Machine Learning with R: Expert techniques for predictive modeling* by Brett Lantz, Packt Publishing, 2019. See book [webpage](https://subscription.packtpub.com/book/big_data_and_business_intelligence/9781788295864).

Further reading

* *The Art of R Programming: A Tour of Statistical Software Design* by Norman Matloff, No Starch Press, 2011. See book [webpage](https://nostarch.com/artofr.htm)
* *An Introduction to R for Spatial Analysis and Mapping* by Chris Brunsdon and Lex Comber, Sage, 2015. See book [webpage](https://uk.sagepub.com/en-gb/eur/an-introduction-to-r-for-spatial-analysis-and-mapping/book241031)
* *Geocomputation with R* by Robin Lovelace, Jakub Nowosad, Jannes Muenchow, CRC Press, 2019. See [online book](https://bookdown.org/robinlovelace/geocompr/).

# 2 Core concepts

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

1. Variables
2. Functions
3. Libraries

## 2.1 Variables

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

a\_variable <- 1

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

a\_variable

## [1] 1

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

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

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

a\_variable <- 1

manipulate the value of the variable as

a\_variable <- a\_variable + 10

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

another\_variable <- a\_variable

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

Let us consider the following example:

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

To speed up the process we could write code that does the calculation (temperature at location A - temperature at location B). This piece of code takes to 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.

**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.

## 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 **function**s. Functions are a set of instructions that preform 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

By calling the functions, we instruct

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.

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)](https://gdal.org/) functionalities.

R provides some basic functions to manipulate strings, but the stringr library provides a more consistent and well-defined set of functions. Assuming that the library has already been installed on your computer, you can load the library as

library(stringr)

## Warning: Paket 'stringr' wurde unter R Version 4.0.5 erstellt

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

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

## Warning: package 'stringr' is in use and will not be installed

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

Once the library is installed and loaded, a new series of functions is available within your environment. For instance, ‘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"

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

**See solution!**

ls("package:stringr")

# 3 Complex Data Types

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

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

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

Complex data types constrain the structure that a container (such as a variable) might take.

## 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 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 selected 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 Stefano de Sabbatas video on [‘Data Types (see min 9:30)’.](https://sdesabbata.github.io/granolarr/lectures_110/)

## 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

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

### 3.2.2 Arrays

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

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

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

### 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 the 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

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 (charater, 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.

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!**

#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

## 4.1 Recap

**Prev**: Data types

* Vectors
* Factors
* Matrices and arrays
* Lists

**Now**: Control structures

* Conditional statements
* Loops

## 4.2 If

Format: **if** (*condition*) *statement*

* *condition*: expression returning a logic value (TRUE or FALSE)
* *statement*: any valid R statement
* *statement* only executed if *condition* is TRUE

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

## Negative

a\_value <- 8  
if (a\_value < 0) cat("Negative")

## 4.3 Else

Format: **if** (*condition*) *statement1* **else** *statement2*

* *condition*: expression returning a logic value (TRUE or FALSE)
* *statement1* and *statement2*: any valid R statements
* *statement1* executed if *condition* is TRUE
* *statement2* executed if *condition* is FALSE

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

## Negative

a\_value <- 8  
if (a\_value < 0) cat("Negative") else cat("Positive")

## Positive

## 4.4 Code blocks

**Code blocks** allow to encapsulate **several** statements in a single group

* { and } contain code blocks
* the statements are execute together

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

## 4.5 Loops

Loops are a fundamental component of (procedural) programming.

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

## 4.6 While

The *while* construct can be defined using the while reserved word, followed by the conditional statement 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 conditional statement 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

## 4.7 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 ?

## 4.8 For

It is common practice to create a vector of integers on the spot in order to execute a certain sequence of steps a pre-defined number of times.

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

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

## 4.9 Loops with conditional statements

3:0

## [1] 3 2 1 0

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

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

## 4.10 Summary

Control structures

* Conditional statements
* Loops

**Next**: Functions

* Defining functions
* Scope of a variable

# 5 Functions

## 5.1 Summary

**Prev**:Control structures

* Conditional statements
* Loops

**Now**: Functions

* Defining functions
* Scope of a variable

## 5.2 Defining functions

A function can be defined

* using an **identifier** (e.g., add\_one)
* on the left of an **assignment operator** <-
* followed by the corpus of the function

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

## 5.3 Defining functions

The corpus

* starts with the reserved word function
* followed by the **parameter(s)** (e.g., input\_value) 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  
 }

## 5.4 Defining functions

After being defined

* a function can be invoked by specifying
  + the **identifier**
  + the necessary **parameter(s)**

add\_one(3)

## [1] 4

add\_one(1024)

## [1] 1025

## 5.5 More parameters

* A function can be defined as having two or more **parameters**
  + by specifying more than one parameter name (separated by **commas**) in the function definition
* A function always take as input as many values as the number of parameters specified in the definition
  + otherwise an error is generated

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

## [1] 6

## 5.6 Functions and control structures

Functions can contain both loops and conditional statements

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

## 5.7 Scope

The **scope of a variable** is the part of code in which the variable is `visible'' In R, variables have a \*\*hierarchical\*\* scope: - a variable defined in a script can be used referred to from within a definition of a function in the same script - a variable defined within a definition of a function will \*\*not\*\* be referable from outside the definition - scope does \*\*not\*\* apply toifor loop constructs ## Example In the case below -x\_valueis \*\*global\*\* to the functiontimes\_x-new\_valueandinput\_valueare \*\*local\*\* to the functiontimes\_x- referring tonew\_valueorinput\_valuefrom outside the definition oftimes\_x` would result in an error

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

## [1] 20

## 5.8 Summary

Functions

* Defining functions
* Scope of a variable

**Next**: Practical session

* Conditional statements
* Loops
  + While
  + For
* Functions
  + Loading functions from scripts
* Debugging

# 6 Selection and filtering

all R code parts are removed, because of an unsolved error message

## 6.1 Recap

**Prev**: Data Frames

* Data Frames
* Tibbles

**Now**: Data selection and filtering

* dplyr
* dplyr::select
* dplyr::filter

## 6.2 dplyr

The dplyr (pronounced *dee-ply-er*) library is part of tidyverse and it offers a grammar for data manipulation

* select: select specific columns
* filter: select specific rows
* arrange: arrange rows in a particular order
* summarise: calculate aggregated values (e.g., mean, max, etc)
* group\_by: group data based on common column values
* mutate: add columns
* join: merge tables (tibbles or data.frames)

## 6.3 Example dataset

## 6.4 Selecting table columns

## 6.5 dplyr::select

select can be used to specify which columns to retain

## 6.6 dplyr::select

… or whichones to drop, using - in front of the column name

## 6.7 Logical filtering

Conditional statements can be used to filter a vector

* i.e. to retain only certain values
* where the specified value is TRUE

## 6.8 Conditional filtering

As a conditional expression results in a logic vector…

… conditional expressions can be used for filtering

## 6.9 Filtering data frames

The same approach can be applied to **data frames** and **tibbles**

## 6.10 dplyr::filter

## 6.11 Select and filter

## 6.12 Summary

Data selection and filtering

* dplyr
* dplyr::select
* dplyr::filter

# 7 Read and write data

all R code parts are removed, because of an unsolved error message

## 7.1 Summary

Tidy-up your data

* Wide and long data
* Re-shape data
* Handle missing values

**Next**: Read and write data

* file formats
* read
* write

## 7.2 Text file formats

A series of formats 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

## 7.3 Comma Separated Values

The file 2011\_OAC\_supgrp\_Leicester.csv contains

* one row for each [Output Area (OA)](https://www.ons.gov.uk/methodology/geography/ukgeographies/censusgeography) in Leicester
* [Lower-Super Output Area (LSOA)](https://www.ons.gov.uk/methodology/geography/ukgeographies/censusgeography) containing the OA
* code and name of the supergroup assigned to the OA by the [2011 Output Area Classification](http://geogale.github.io/2011OAC/)
* total population of the OA

Extract showing only the first few rows

OA11CD,LSOA11CD,supgrpcode,supgrpname,Total\_Population  
E00069517,E01013785,6,Suburbanites,313  
E00069514,E01013784,2,Cosmopolitans,323  
E00169516,E01013713,4,Multicultural Metropolitans,341  
E00169048,E01032862,4,Multicultural Metropolitans,345

## 7.4 readr

The [readr](https://readr.tidyverse.org/) (pronounced *read-er*) library is part of [tidyverse](https://www.tidyverse.org/)

Provides functions to read and write text files

* readr::read\_csv: comma-separated files .csv
* readr::read\_csv2: semi-colon-separated files .csv
* readr::read\_tsv: tab-separated files .tsv
* readr::read\_fwf: fix-width files .fwf
* readr::read\_delim: files using a custom delimiter

and their *write* counterpart, such as

* readr::write\_csv: comma-separated files .csv

## 7.5 readr::read\_csv

The readr::read\_csv function of the [readr](https://readr.tidyverse.org/index.html) library reads a *csv* file from the path provided as the first argument

## 7.6 Read options

Read functions provide options about how to interpret a file contents

* For instance, readr::read\_csv
  + col\_names:
    - TRUE or FALSE whether top row is column names
    - or a vector of column names
  + col\_types:
    - a cols() specification or a string
  + skip: lines to skip before reading data
  + n\_max: max number of record to read

## 7.7 Column specifications

* col\_logical() or l as logic values
* col\_integer() or i as integer
* col\_double() or d as numeric (double)
* col\_character() or c as character
* col\_factor(levels, ordered) or f as factor
* col\_date(format = "") or D as data type
* col\_time(format = "") or t as time type
* col\_datetime(format = "") or T as datetime
* col\_number() or n as numeric (dropping marks)
* col\_skip() or \_ or - don’t import
* col\_guess() or ? use best type based on the input

## 7.8 readr::read\_csv

Using readr::read\_csv as in the previous example with no further options will generate the following warning

## 7.9 readr::read\_csv

## 7.10 readr::read\_csv

## 7.11 readr::write\_csv

The function write\_csv can be used to save a dataset to csv

Example:

1. **read** the 2011 OAC dataset
2. **select** a few columns
3. **filter** only those OA in the supergroup *Suburbanites* (code 6)
4. **write** the results to a file named *2011\_OAC\_supgrp\_Leicester\_supgrp6.csv*

## 7.12 readr::write\_tsv

## 7.13 Other data imports

[Tidyverse](https://www.tidyverse.org/) also imports [other packages for reading data](https://www.tidyverse.org/packages/#import)

* Tabular formats
  + [readxl](https://readxl.tidyverse.org/) for Excel (.xls and .xlsx)
  + [haven](https://haven.tidyverse.org/) for SPSS, Stata, and SAS data.
* Databases
  + [DBI](https://cran.r-project.org/web/packages/DBI/) for relational databases
* NoSQL
  + [jsonlite](https://cran.r-project.org/web/packages/jsonlite/) for JSON
  + [xml2](https://cran.r-project.org/web/packages/xml2/) for XML
* Web
  + [httr](https://cran.r-project.org/web/packages/httr/) for web APIs

## 7.14 Summary

Read and write data

* file formats
* read
* write

**Next**: Practical session

* Read and write data
* Tidy data
* Join operations

# 8 Data visualisation

all R code parts are removed, because of an unsolved error message

## 8.1 Recap

**Prev**: Reproducibility

* 221 Reproducibility
* 222 R and Markdown
* 223 Git
* 224 Practical session

**Now**: Data visualisation

* Grammar of graphics
* ggplot2

## 8.2 Visual variables

A **visual variable** is an aspect of a **mark** that can be controlled to change its appearance.

Visual variables include:

* Size
* Shape
* Orientation
* Colour (hue)
* Colour value (brightness)
* Texture
* Position (2 dimensions)

## 8.3 Grammar of graphics

Grammars provide rules for languages

*“The grammar of graphics takes us beyond a limited set of charts (words) to an almost unlimited world of graphical forms (statements)”* (Wilkinson, 2005)

Statistical graphic specifications are expressed in six statements:

1. **Data** manipulation
2. **Variable** transformations (e.g., rank),
3. **Scale** transformations (e.g., log),
4. **Coordinate system** transformations (e.g., polar),
5. **Element**: mark (e.g., points) and visual variables (e.g., color)
6. **Guides** (axes, legends, etc.).

## 8.4 ggplot2

The ggplot2 library offers a series of functions for creating graphics **declaratively**, based on the Grammar of Graphics.

To create a graph in ggplot2:

* provide the data
* specify elements
  + which visual variables (aes)
  + which marks (e.g., geom\_point)
* apply transformations
* guides

## 8.5 Boxplots

* x categorical variable
* y variable to plot
* geom\_boxplot

## 8.6 Jittered points

* x categorical variable
* y variable to plot
* geom\_jitter

## 8.7 Violin plot

* x categorical variable
* y variable to plot
* geom\_violin

## 8.8 Violin plot

## 8.9 Lines

* x e.g., a temporal variable
* y variable to plot
* geom\_line

## 8.10 Lines

## 8.11 Scatterplots

* x and y variable to plot
* geom\_point

## 8.12 Overlapping points

* x and y variable to plot
* geom\_count counts overlapping points and maps the count to size

## 8.13 Overlapping points

## 8.14 Bin counts

* x and y variable to plot
* geom\_bin2d

## 8.15 Bin counts

## 8.16 Summary

Data visualisation

* Grammar of graphics
* ggplot2

**Next**: Descriptive statistics

* stat.desc
* dplyr::across

# References