Application Development with R

Christian Neuwirth

01 Oktober, 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.

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!**

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

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

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

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

**See solution!**

The condition yields FALSE. The statement is not executed.

## 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 has 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 a 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

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.

### 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 exectuion number 3 :  
## See you later!

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

**See solution!**

The for-loop is still executed 3 times.

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

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

**See another example!**

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

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

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

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

# 5 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 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** 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

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

}

# 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 it offers a grammar for data manipulation.

## 6.1 Preparation

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

The code below, loads the data frame flights from the library nycflights13 into the variable flights\_from\_nyc. We will use this data frame throughout this lesson.

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

The operator :: is used to indicate that the data frame flights is situated within the library nycflights13. You can use ::, if you know exactly which library contains the object or function desired. This helps avoiding ambiguities in the case functions from different loaded libraries have identical names.

Add both lines above as well as the code snippets provided in the upcoming examples to a new R script file.

## 6.2 Data manipulation

Firstly, inspect the data frame flights\_from\_nyc by writing it to the console. 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. The code below provides 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 the data frame flights\_from\_nyc is passed into function count() and the output is pased into function kable() to render the result\_df

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

The function count() can also be used to count the number rows of a table that have 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. 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 distance\_traveled\_from (the total 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 some **columns** to output. For instance in the code below, the function select() is used to select the columns origin, dest, and dep\_delay, in combination with the function slice\_head, which can be used to include only the first n rows (5 in the example below) to 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 |

### 6.2.4 Arrange

The function arrange() can be used to sort a tibble by ascending order of the values in the specified column. If the operator - 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 (in the examples 5) rows in a table, based on the existing order.

### 6.2.5 Exercise: data manipulation

## 6.3 Join

### 6.3.1 Table join

### 6.3.2 Spatial join

### 6.3.3 Exercise: join

# 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