Data manipulation and visualization with R

2.1 Importing data

2.1. Importing data

Introduction

Now we know the basics of R and can apply our knowledge to work with data.

All data we used so far was already pre-installed in R. In this chapter we will learn how to import external data from files. The most commonly used file formats for statistical data are .xlsx used by Microsoft Excel and the comma separated values format (csv).

We will look at both formats and how to import data into RStudio through the script. For this purpose we will also look at directories and how to locate data on the computer.

File formats

.xlsx files

The .xlsx format is used by Microsoft Excel.

The format can only be opened with Excel, it is therefore called a proprietary file format. One can open an Excel file with a text editor, but the data will not be presented in a readable way

Each piece of data is stored in an individual cell. The cell not only contains data but also properties like fonts and text sizes, background and text colors. The displayed value can also be the result of an equation that is only shown when the user selects the cell via mouse cursor or keyboard. Therefore, Excel files need to store a lot of information besides the actual data, which usually requires much larger files.

number	pokemon	hp	attack	defense	spAttack
001	Bulbasaur	45	49	49	65
002	Ivysaur	60	62	63	80
003	Venusaur	80	82	83	100
003M	Venusaur	80	100	123	122
004	Charmander	39	52	43	60
005	Charmeleon	58	64	58	80
006	Charizard	78	84	78	109
006MX	Charizard	78	130	111	130
006MY	Charizard	78	104	78	159
007	Squirtle	44	48	65	50
008	Wartortle	59	63	80	65
009	Blastoise	79	83	100	85
009M	Blastoise	79	103	120	135

File formats

.csv files

Text files can be formatted as .csv files (comma separated values) to store information in a simple way that requires little storage capacity. csv is an open-source format and can be accessed and edited by many different programs, like a text editor or RStudio.

As the name implies, values are separated by commas to indicate individual pieces of information. Naturally, commas must not be used within the data itself to avoid messing with the data format. Character-based data like “Bond, James” require quotation marks to indicate that included commas are part of a character string. Commas within quotation marks will not separate the data.

German-speaking countries have a different way of writing decimals: Here, a comma is used instead of a period. Therefore, data files from German sources may use a “;” to separate data and a “,” for decimal numbers. Some files even use tabs (“ “) as separators. The formatting style should always be checked before the data import to ensure the right type of separators and decimals are used by the code.

```
# International formatting using “,” as separator
"index","number","pokemon","hp"
"1","001","Bulbasaur",45.0
"2","002","Ivysaur",60.0
"3","003","Venusaur",80.0
"4","003M","Mega Venusaur",80.0


# German formatting using “;” as separator
"Nr","Nummer";"Pokemon";"TP"
"1";"001";"Bisasam";45,0
"2";"002";"Bisaknosp";60,0
"3";"003";"Bisaflor";80,0
"4";"003M";"Mega Bisaflor";80,0
```

File formats

Comparison: .xlsx and .csv

Both file formats have their advantages and disadvantages.

Excel files are easy to edit within Excel and can be formatted in a presentable fashion. They, however, require a Microsoft licence and the files are often several MB large, which might impede sending them via mail.

CSV files are small in size and can easily be sent via mail. Their lack of formatting makes them unpresentable and difficult to read. They are mostly used to exchange data, especially among different programs or users.

Both formats can be converted into each other.

.xlsx	.csv
Contains data, as well as information about formatting and computation formulas	Contains only data as plain text
Can be formatted to be visually pleasing and accessible	Can be difficult to read due to lack of formatting
File is proprietary and can only be accessed by Microsoft Office software	File is open-source and can be accessed by many different programs
The files can easily become large in size (several MB) due to additional information stored within the file	Files remain relatively small in size even with larger data sets (several KB)
Easy to edit files directly in Excel	Usually require software (like RStudio) to effectively edit data
Can be exported to .csv	Can be imported by Excel

File formats

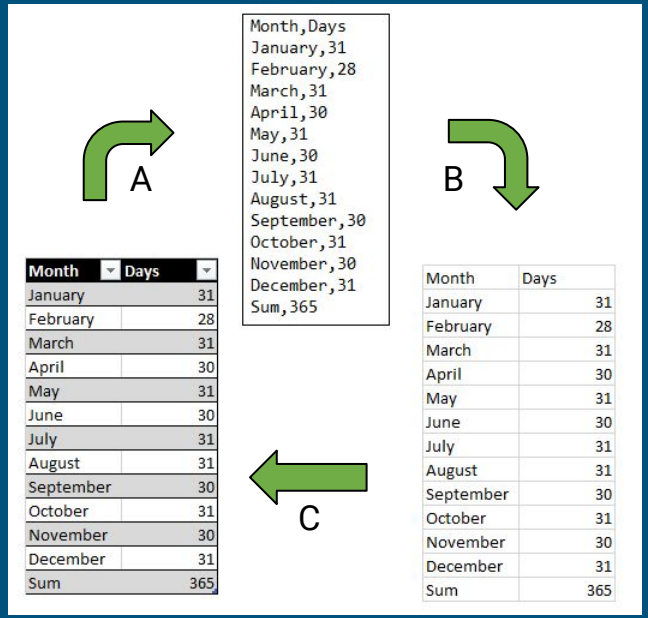
Conversion between .xlsx and .csv

Both formats can be converted into each other.

A: The formatted Excel table can be “saved as” a .csv file when the file format is selected and the suffix is added to the file’s name. It loses all formatting and formulas. In this example the “sum” is no longer an equation, but only a plain integer.

B: The .csv file can be imported back to Excel. There, under the tab “Data”, the option “From Text/CSV” (the exact wording might differ depending on the program version) can be selected. A popup window will display a few options, e.g. the manual selection of the separator. The imported data lacks formatting and formulas.

C: The formatting has to be added manually if desired. The user should therefore only format the data when no further conversions are necessary.



Month	Days
January	31
February	28
March	31
April	30
May	31
June	30
July	31
August	31
September	30
October	31
November	30
December	31
Sum	365

Month	Days
January	31
February	28
March	31
April	30
May	31
June	30
July	31
August	31
September	30
October	31
November	30
December	31
Sum	365

Locating files

Directories: Where is the data?

Before we can import data, we need to let the program know where the data is located.

R can find all (non-restricted) files as long as the complete path to the file is specified. As the path can be rather long, it makes sense to determine a folder as a working directory (wd) to declare this as the default place to look for files. The wd is also the place where exported files, like images, are stored.

A wd is declared with setwd(path), where path is a string with a folder path like "C:/Users/Scully/Xfiles". Please note that the path needs to use "/" to separate folders and not "\". The path must lead to a folder, and not a specific file itself. Only one folder can be the current working directory at a time! getwd( ) shows the current path on screen.

The dir( ) function, short for "directory", lists all files stored in the current working directory.

```
# set path to "C:/..."
path <- "C:/Users/Scully/Xfiles"


# set working directory to "C:/..."
setwd(path)


# show path of current working directory
getwd()
# output: "C:/Users/Scully/Xfiles"

# check content of current working directory
dir()
# output: "aliens_in_the_kitchen.mp4"
"i_want_to_believe.png"
```

Locating files

Directories: Grep everything you need!

Found the files you were looking for? Now grep them! “grep” is short for “global search for a regular expression and print out matched lines”. The function grep( ) uses regular expressions to select elements from a vector that fulfill a specific condition. (We will not go into detail with “regular expression”, but it is worth looking them up.)

This is useful when the wd contains more than one file. In this example we have two .csv files and an .xlsx file. grep(pattern, vector) requires first the selection condition (e.g. “*.xlsx” selects all files ending in “.xlsx”) and then the vector containing all files. The function grep( ) returns the index of matches, while grepl( ) returns a logical.

Applying the return of either grep( ) or grepl( ) to the original vector of files returns a selection of files that fulfill the condition. The details of data selection will be explained more thoroughly in the following chapter “2.2. Selecting data”.

```
# show all files in directory
dir()
# output: "aliens.csv" "area51.csv"
"unsolved_mysteries.xlsx"

# Select specific files
grep("*.xlsx", dir())
# output (index): 3

grepl("*.xlsx", dir())
# output (logical): FALSE FALSE TRUE

# Select all xlsx
dir()[grepl("*.xlsx", dir())] # "aliens.csv" "area51.csv"

# Select all csv
dir()[grepl("*.csv", dir())] # "unsovled_mistories.xlsx"
```

Reading files

Reading Excel files

To read Excel files, we first need to install the “readxl” package and add it to the library. The installation may warn that the package “Rtools” needs to be installed as well.

The function read_excel( ) reads the first entire table sheet within the Excel file. The only mandatory input is, of course, the filename (or its complete path if not stored within the current working directory).

The argument “skip=n” ignores the first n rows. This is useful when the head of the Excel table differs, e.g. by only having a username, source or date in the first rows. The range argument reads only selected cells, while the sheet argument reads only selected table sheets within the file. The range can also be specified as “Sheet!Range”, as shown in the last example.

The “population_1950.xlsx” file is provided for this course!
(source: https://service.destatis.de/bevoelkerungspyramide/#!y=1950&a=32,56&v=2)

```
# install “readxl” package and load it.
install.packages("readxl")
library("readxl")

#read excel file
data <- read_excel("Population_1950.xlsx")

#skip rows or select range
columns <- c("Index", "Year", "Sex", "Age", "Count")

data <- read_xlsx("Population_1950.xlsx", skip=2,
          col_names=columns)

data <- read_xlsx("Population_1950.xlsx", range="A1:E100",
          sheet="Table1", col_names=columns)

data <- read_xlsx("Population_1950.xlsx", range="Table1!A3:E100",
          col_names=columns)
```

Reading files

Reading .csv files

The read_csv( ) function is a standard function and does not require any further packages.

Only the input of a file name is mandatory, the function will use default settings for the other arguments. By default the separator (sep) will be set to "," and the decimal sign (dec) to ".". As mentioned earlier, some sources may use ";" or " " as separators and "." as decimal signs. If "sep" or "dec" are set incorrectly, the data is messed up.

The "header" argument is set to TRUE, i.e. the function expects the csv file's first row to contain the column names. This is usually the case in all csv files obtained from reliable sources like statistical agencies. If the first row contains regular data, it is imperative to set the "header" argument to FALSE. We can then use the names( ) function to manually specify column names, once the data is imported and stored.

The "population_1950.csv" file is provided for this course!
(source: https://service.destatis.de/bevoelkerungspyramide/#!y=1950&a=32,56&v=2)

```
# read csv file
data <- read.csv("population_1950.csv")


# specify separator and decimal sign
data <- read.csv("population_1950.csv", sep=",", dec=".")

# specify separator and decimal sign (incorrectly)
data <- read.csv("population_1950.csv", sep=";", dec=",")


# specify header
data <- read.csv("population_1950.csv", sep=",", dec=".",
header=FALSE)

names(data) <- c("Index", "Year", "Sex", "Age", "Count")
```

Reading files

Reading multiple .csv files and writing new files

With the help of a for-loop, it is possible to read multiple .csv files in a row and combine them to one dataframe.

First, we use the previously described grep( ) or grepl( ) functions to select all .csv files in our working directory. We create an empty dataframe called “data”. (Please note: We use the name “data” for the sake of simplicity and space efficiency. A name like “population_1950_2018” would be more appropriate.)

The for-loop goes through each element in the list, conventionally represented by the variable “i”, reads the file and combines it row-wise (rbind( )) to our data. Each iteration adds to the same dataframe. This requires the files to have a consistent way of naming their columns.

To omit this step for the next session, we use write.csv(data, “outputfile.csv”) to store the combined data in one file within the current directory.

```
# select all .csv files
csv_files <- dir()[grepl("*.csv", dir())]

# create empty dataframe
data <- data.frame()

# loop through all files
for(i in csv_files){
   # read csv and add all rows to data
   data <- rbind(data, read.csv(i))
}

# display data
data

# create new .csv file
write.csv(data, "data.csv")
```