

Data Preparation with R

Dr. Bernd Fellinghauer & Dr. Carolina Fellinghauer

PhD Courses in Health Sciences
University of Lucerne

December 2nd, 2019

Table of Contents

- ▶ A Tour of R for Data Preparation
- ▶ Preparing a WHO Tuberculosis Data Set

A Tour of R for Data Preparation

The RStudio Editor

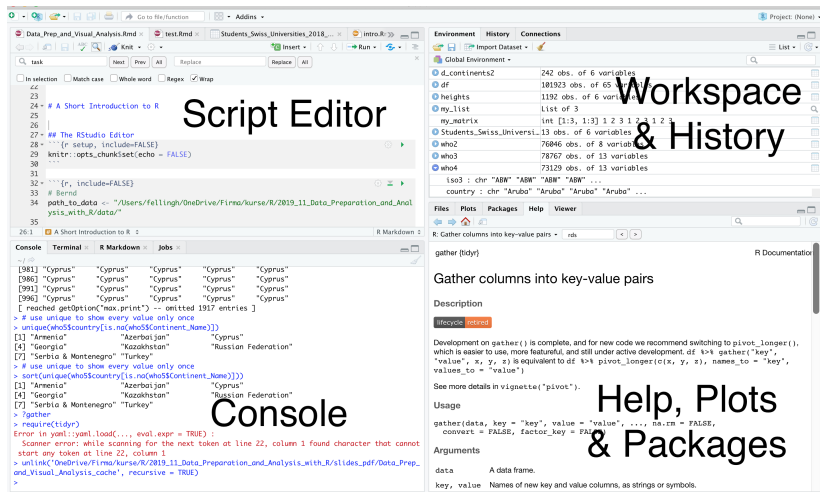


Figure 1: The RStudio 4-Pane Layout

RStudio - Useful Shortcuts

- ▶ **Send from Script Editor pane to Console:** Cntrl/Cmd + Enter or “Run” button
- ▶ **Auto-completion:** Tabulator
- ▶ **Recall last command in console:** up/down arrow (or history tab at the top right)
- ▶ **Stop button:** only visible when running code
- ▶ **Assignments:** Alt and - (insert <-)

Using the Console

Console

R code is executed on the **console**.

R is vector-based. The returned output **[1] 3** hence indicates that 3 is the first element of the returned vector.

```
1 + 2
```

```
## [1] 3
```

Multi-Line

A command can **span multiple lines** (as long as a line by itself is not complete)

```
1 +  
  2
```

```
## [1] 3
```

Getting Help

R has a built in help search. Type **?command_name**

```
?mean
```

You can also search through the help in the tab at the bottom right.

For some functions, you can also find vignettes, which provide an in-depth description. Type **vignette()** to see a complete list of all available vignettes (this depends on the installed packages), e.g.

```
vignette("readr")
```

Name Conventions & Assignments

Name Conventions

- ▶ A variable name cannot contain spaces, special characters or start with a number
- ▶ Tip: Using "_" or "." for longer names improves readability

Assignments

The operators = or <- can be used for variable assignment

```
my_variable = 1 + 2  
my_variable <- 1 + 2 # default for "ALT and -"
```

Comments

Comments start with #

```
3 + 4 # this is a comment and not executed  
## [1] 7
```


Reading Excel-Data into R

A Typical Excel File

Excel is one of the most wide-spread formats for data collection in research. However, it does not necessary follow a tidy data structure.

	A	B	C	D	E	F
1	Lots of people					
2	simply cannot resist writing					some notes
3	at	the	top		of	their spreadsheets
4	or	merging				cells
5	Name ▾	Profession ▾	Age ▾	Has kids ▾	Date of birth ▾	Date of death ▾
6	David Bowie	musician	69	TRUE	08.01.47	10.01.16
7	Carrie Fisher	actor	60	TRUE	21.10.56	27.12.16
8	Chuck Berry	musician	90	TRUE	18.10.26	18.03.17
9	Bill Paxton	actor	61	TRUE	17.05.55	25.02.17
10	Prince	musician	57	TRUE	07.06.58	21.04.16
11	Alan Rickman	actor	69	FALSE	21.02.46	14.01.16
12	Florence Henderson	actor	82	TRUE	14.02.34	24.11.16
13	Harper Lee	author	89	FALSE	28.04.26	19.02.16
14	Zsa Zsa Gábor	actor	99	TRUE	06.02.17	18.12.16
15	George Michael	musician	53	FALSE	25.06.63	25.12.16
16	Some					
17		also like to write stuff				
18			at the	bottom,		
19						too!

The readxl Package

- ▶ The **readxl**-package can read both **.xls** and **.xlsx** format data.
- ▶ Unlike some of the other R packages for reading Excel data (e.g. xlsx package), it has no external dependencies (such as a Java runtime environment), which makes setting it up (much) easier.
- ▶ Also, it is an official part of tidyverse - but it needs to be explicitly loaded with **require(readxl)**.

Example Excel File: deaths.xlsx

```
require(readxl)
```

```
## Loading required package: readxl
```

```
# list of example data sets
```

```
readxl_example()[1:3]
```

```
## [1] "clippy.xls"      "clippy.xlsx"    "datasets.xls"
```

```
readxl_example()[4:6]
```

```
## [1] "datasets.xlsx" "deaths.xls"     "deaths.xlsx"
```

```
# get the path to the deaths-data set
```

```
path_xlsx <- readxl_example("deaths.xlsx")
```

File Paths

Windows and Mac/Linux file paths differ in their usage of the folder-separator (/ vs. \\)

Tip: A Windows filepath can also use the Mac/Linux notation.

Mac or Linux:

```
"/path/to/file.xlsx"
```

on Windows:

```
"C:/path/to/file.xlsx" # or  
"C:\\\\path\\\\to\\\\file.xlsx"
```

Specifying Folder Paths

Tip: It is often helpful to separate the data folder and file name

```
data_folder <- "/Users/fellingh/.../readxl/extdata/"  
file_deaths <- "deaths.xlsx"  
full_path <- paste0(data_folder,file_deaths)  
full_path
```

```
## [1] "/Users/fellingh/.../readxl/extdata/deaths.xlsx"
```

Extract Sheet Names from Excel File

See all sheets in an Excel file

```
excel_sheets(path_xlsx)
```

```
## [1] "arts"  "other"
```

The read_excel Function

The `read_excel`'s key arguments are

- ▶ **path**: location of Excel file
- ▶ **sheet**: select a sheet to read in
- ▶ **range**: optionally limit the reading range to a subset of
 - ▶ rows (`cell_rows`)
 - ▶ columns (`cell_cols`)
 - ▶ a (named) Excel range (**B5:D16** or **exampleRange!B1:D5**)
- ▶ **na**: specify how missing values are coded

```
d <- read_excel(path=path_xlsx,  
                sheet = "arts",  
                range = cell_rows(5:11),  
                na=c("8888", "9999"))
```


Taking a First Look at a New Data Set

The **View**-function allows to view data sets in a format similar to Excel, i.e.

- ▶ Full display of cell contents
- ▶ Scrolling up/down and left/right
- ▶ Filter rows based on a particular value using the Filter icon at the top
- ▶ The free text field at the right allows to also search across all columns at the same time

```
View(d)
```

Exercise: Select only the actors

Data Structures in R

Main Data Structures in R

Now that we have some data to work with, we will take a look at the four main data structures in R:

- ▶ **vector**: 1-dimensional set of numeric, sting or logical elements
- ▶ **matrix**: 2-dimensional set of numeric, sting or logical elements
- ▶ **data.frame**: like a matrix, but can combine different element types - typically your data will be a data.frame
- ▶ **list**: combines any of the above element types in a list. A list element can itself again contain a list. Functions often collect all returned objects into a list.

Vectors

Defining a Vector

The **c**-function (short for **c**ombine) is useful to define a new vector

```
treatment <- c("Old", "New", "New", "Placebo")
```

```
days_since_injury <- c(1, 2, 1, 3)
```

```
complications <- c(TRUE, FALSE, FALSE, FALSE)
```

Tip: you can use the **c**-function also to append additional values to an existing vector

Select a Vector from a Data Frame / Tibble

Each column of a data frame is itself a vector and can be selected in one of two ways:

```
d$Age
```

```
## [1] 69 60 90 61 57 69
```

Select a Vector from a Data Frame / Tibble

The tidyverse has a data frame equivalent called “tibble”. They behave very similar, but the below is one noticable exception:

```
d[1:3, "Age"] # tibble - this does not return a vector!
```

```
## # A tibble: 3 x 1
```

```
##   Age
```

```
##   <dbl>
```

```
## 1    69
```

```
## 2    60
```

```
## 3    90
```

```
as.data.frame(d)[1:3, "Age"]
```

```
## [1] 69 60 90
```

Tibble vs. Data Frame - Any Other Surprises?

Beyond the subsetting behavior, there are two more main differences

- ▶ a tibble prints differently
- ▶ a tibble does not automatically convert characters to factors

Factors

Unordered Factors

Unordered factor

A special case of character variables, which fixes the set of allowed levels

```
(v_factor_unord <- factor(c("B", "A", "C"),  
                           levels = c("A", "B", "C")))
```

```
## [1] B A C
```

```
## Levels: A B C
```

Exercise

Try the two assignments below. What behavior do you observe?

```
v_factor_unord[4] <- "C"
```

```
v_factor_unord[5] <- "D"
```

Ordered Factors

Ordered factor

Fixes the order of levels, too. This is required for some statistical models (e.g. ordered logistic regression models)

```
(v_factor_ord <- factor(c("L", "H", "M"),  
                        ordered = TRUE,  
                        levels = c("L","M","H")))
```

```
## [1] L H M
```

```
## Levels: L < M < H
```

Assess the Structure of Factors

The **str** function shows details about the structure of objects.

Exercise

Run the two commands below and compare the output for unordered vs. ordered factors

```
str(v_factor_unord)
str(v_factor_ord)
```

Factors - Handle with Care

Why Factors?

- ▶ Helpful to enforce correct and consistent coding of levels
- ▶ Error for non-defined levels (e.g. a mis-spelling)

Warning: Numeric Vectors as Factors

- ▶ A factor can also consist of numbers
- ▶ Such factors gives unexpected results when converting back to numbers!

Converting a Factor of Numbers back to Numeric

Exercise

- ▶ Look at the below code prior to executing it. What output do you expect?
- ▶ What output do you get when running it?

```
v_factor_num <- factor(c("4","3","8"),  
                      levels=c("3", "4", "8"))  
as.numeric(v_factor_num)
```

Converting a Factor of Numbers back to Numeric - The Right Way

To extract the actual numbers and not the index use

```
as.numeric(as.character(v_factor_num)) # true values
```

Tip

- ▶ From personal experience, this is one of the most common problems when preparing data in R
- ▶ The **read.csv** often converts characters to factors by default) - the argument **stringsAsFactors = FALSE** prevents this behavior
- ▶ The complexity around factors is the main reason why the tidyverse's functions for reading data do not automatically convert character variables to factors

Sequences

Creating Sequences

A **sequence** indicates a vector with known from and to values. R extrapolates the intermediate values based on the chosen parameter settings.

As increments of 1 / -1 are very typical, the colon offers a shortcut

```
3:6
```

```
-5:-8
```


The seq Function

The **by** argument allows to customize the increment

```
seq(from=1, to=5, by=2)
```

Alternatively, **length.out** specifies the required sequence length

```
seq(from=1, to=4, length.out=7)
```

Repetitions

Repeat a number (or a vector) a specified nbr of **times**

```
rep(5, times = 4)
```

```
rep(c(2,3,4), times=2)
```

Repeat a pattern until the sequence has the specified **length**

```
rep(c(2,3,4), length.out=7)
```

Repeat **each** element a certain number of times

```
rep(c(2,3,4), each=2)
```

Exercise: `c`, `rep` and `seq`

- ▶ Create a vector with values A, B, C, and D using `c`
- ▶ Create -1, -2, -3, -4 using `seq`
- ▶ Create 6 6 6 7 7 7 8 8 8 using `rep`

Solution: c, rep and seq

```
c("A", "B", "C", "D")
```

```
## [1] "A" "B" "C" "D"
```

```
seq(-1, -4, -1)
```

```
## [1] -1 -2 -3 -4
```

```
rep(c(6,7,8), each = 3)
```

```
## [1] 6 6 6 7 7 7 8 8 8
```

Calculate Descriptive Statistics

Assess a Vector in More Detail

A couple of helpful functions

- ▶ **sum(v)**: sum up all elements
- ▶ **mean(v)**, **median**, **mode**: measures of central tendency
- ▶ **quantile(v)** to calculate percentile values of v
- ▶ **var(v)**, **sd(v)**: variance and standard deviation of v
- ▶ **min(v)**, **max(v)**, **range(v)**: basic summary functions
- ▶ **summary(df)**: return statistics each col of a data frame

Exercise: Calculate Descriptive Statistics

```
set.seed(123) # reproducible random number generation  
v_norm <- rnorm(100000) # sample 100'000 vars
```

- ▶ identify the **mean**, **min**, and **max** of v_norm using the respective function
- ▶ calculate the **default quantiles** of v_norm
- ▶ calculate the **97.5% quantile** of v_norm (tip: see **?quantile** and the **probs** argument)

Solution: Calculate Descriptive Statistics

```
mean(v_norm)
```

```
## [1] 0.0009767488
```

```
min(v_norm)
```

```
## [1] -4.13209
```

```
max(v_norm)
```

```
## [1] 4.322815
```

```
round(quantile(v_norm),2)
```

```
##      0%    25%    50%    75%   100%
```

```
## -4.13 -0.67  0.00  0.68  4.32
```

```
quantile(v_norm, probs = c(0.975))
```

```
##      97.5%
```

```
## 1.956446
```


Strings

Working with Strings

Get the length of a vector

```
length(d$Name) # same as nrow(d), dim(d)
```

Get length of each element in a string vector

```
nchar(d$Name)
```

Extract a substring with fixed positions

```
substring(d$Name, 1, 5)
```

Working with Strings II

Paste strings together

```
paste("path_to_file/", "file_name.txt", sep = "")
```

Change capitalization with **tolower** / **toupper**

```
tolower(d$Name)
```

```
toupper(d$Name)
```

Tip: Changing capitalization is particularly important when building a filter condition

Working with Strings III

Define Subsetting-Conditions: %in%

Check if a text occurs in a string vector

```
"Chuck Berry" %in% d$Name
```

Define Subsetting-Conditions: ==

Direct comparison and **which** to get the index of all matches

```
which("Chuck Berry" == d$Name)
```

The stringr Package

The stringr package from the tidyverse offers a lot more functions to work with strings

Extract words from a string vector

```
require(stringr)
first_name <- word(d$Name, 1)
last_name <- word(d$Name, 2)
```

Perform partial matches for a text string

```
which(str_detect(d$Name, "Berry")) # or grep in base R
```

Selecting Elements

Selecting from a Data Frame

Index

```
d[1:3, 1:3]
```

By Name

```
d[, c("Name", "Profession", "Age")]
```

Tip: Use `row.names` / `col.names` to get / set column names

subset Function - base R

```
subset(d, Age > 60)
```

filter Function - tidyverse

```
require(dplyr)  
filter(d, Age > 60)
```

Exercise: Selecting from a Data Frame

Clean up the non-standard column name for Has kids

```
d$has_kids <- d$`Has kids`
```

- ▶ Select all individuals with kids (tip: Use has_kids to avoid problems)
- ▶ Select all musicians
- ▶ Select all musicians over 70 with kids (tip: use & to combine to conditions)

Solution: Selecting from a Data Frame

```
subset(d, has_kids == TRUE)
filter(d, Profession == "musician")
subset(d, has_kids == TRUE
        & Profession == "musician"
        & Age > 70)
```

Negating Conditions: !

The !-operator negates a condition, i.e. to NOT select “musician”

```
filter(d, Profession != "musician")
```

To negate a multi-element condition use !(...)

```
subset(d, !(  
    has_kids == TRUE  
    & Profession == "musician"  
    & Age > 70)  
)
```

Tip:

- ▶ Use & for AND comparisons
- ▶ Use | for OR comparisons

Dates

Date Formatting


PUBLIC SERVICE ANNOUNCEMENT:

OUR DIFFERENT WAYS OF WRITING DATES AS NUMBERS CAN LEAD TO ONLINE CONFUSION. THAT'S WHY IN 1988 ISO SET A GLOBAL STANDARD NUMERIC DATE FORMAT.

THIS IS *THE* CORRECT WAY TO WRITE NUMERIC DATES:

2013-02-27

THE FOLLOWING FORMATS ARE THEREFORE DISCOURAGED:

02/27/2013 02/27/13 27/02/2013 27/02/13
20130227 2013.02.27 27.02.13 27-02-13
27.2.13 2013. II. 27. $27\frac{1}{2}$ -13 2013.158904109
MMXIII-II-XXVII MMXIII $\frac{\text{LVII}}{\text{CCCLXV}}$ 1330300800
 $((3+3) \times (111+1) - 1) \times 3 / 3 - 1 / 3^3$ 2013 Missss
10/11011/1101 02/27/20/13 $\begin{matrix} 2 & 3 & 1 & 4 \\ 0 & 1 & 2 & 3 & 7 \\ & 5 & 6 & 7 & 8 \end{matrix}$ 

Date Formatting: `as.Date`

Dates need to be properly coded for R to recognize them as such. Ideally, your dates are in a standard format such as YYYY-MM-DD

Standard (ISO)

```
course_day1 <- as.Date("2019-12-02")
```

Swiss-style dates

```
course_day2 <- as.Date("13.01.2020", "%d.%m.%Y")
```

Working with Dates

Calculate nbr days between course day 2 and 1

```
course_day2 - course_day1
```

Extract the year / year-month from a date

```
year_day2 <- format(course_day2, "%Y")  
year_month_day2 <- format(course_day2, "%Y-%m")
```

Today's date

```
Sys.Date()
```

Outlook: Dates and Times

```
as.POSIXct(Sys.time())  
as.POSIXlt(Sys.time())
```

- ▶ Fields containing both date and time information can be handled using either **as.POSIXct** and **as.POSIXlt** (they mainly differ in the way they are internally coded)
- ▶ See **?DateTimeClasses** for details on these classes
- ▶ See <https://www.r-bloggers.com/date-formats-in-r/> or **?strptime** for a list of format options for both dates and datetimes

Data Preparation

Typical Data Preparation Steps

- ▶ Data collection (typically outside of R)
- ▶ Read data into R
- ▶ Assess data
- ▶ Assign types
- ▶ Transform data
- ▶ Join data
- ▶ Handle missing values
- ▶ Aggregate data
- ▶ Outlook: More Aggregation Approaches

Read Data into R

Excel vs. CSV

- ▶ Whilst you can read Excel-files directly, its **extended formatting** abilities or an **unsupported set of characters** may cause trouble when reading data.
- ▶ Generally, format-independent data such as **comma separated values (CSV)** files are easier and faster to read and their respective R functions offer more capabilities to handle non-standard data (e.g. number formatting, dates).
- ▶ Also, you can export a CSV file from an Excel file using “Save as”. Note that different CSV-file formats might be available. They will differ in the used character set / “locale” (e.g. Latin1 and UTF-8).

read.csv - base R

Try reading either of the two data sets. What do you observe?

```
path_UTF8 <- paste0(path_to_data,  
  "Students_Swiss_Universities_2018_2019_UFT8.csv")
```

```
path_LATIN1 <- paste0(path_to_data,  
  "Students_Swiss_Universities_2018_2019_LATIN1.csv")
```

```
d2_utf8 <- read.csv(path_UTF8)  
d2_latin1 <- read.csv(path_LATIN1)
```

Example of a Text File with “Wrong” Encoding

```
ï»¿Regions-ID,Regionsname,Anzahl Studierende,Schweizer,Ausl nde  
,Schweiz,155'448,104'597,7'737,37'934  
261-1,Universit t Z rich,26'557,21'243,1'343,3'971  
261-2,ETH Z rich,20'807,12'852,887,7'068  
351,Universit t Bern,17'222,13'720,520,2'982  
1061,Universit t Luzern,3'007,2'529,124,354  
2196,Universit t Freiburg,10'366,8'378,494,1'494  
2701,Universit t Basel,13'151,9'515,711,2'925  
3203,Universit t St. Gallen,9'183,6'010,496,2'677  
5192,Universit  della Svizzera italiana,2'811,902,176,1'733  
5586-1,Universit t Lausanne,15'325,11'360,1'118,2'847  
5586-2,ETH Lausanne,10'785,4'476,546,5'763  
6458,Universit t Neuenburg,4'066,3'062,252,752  
6621,Universit t Gen ,16'988,10'550,1'070,5'368
```

Figure 3: Latin1 not showing umlauts correctly

Use UTF-8 as CSV Encoding in Excel

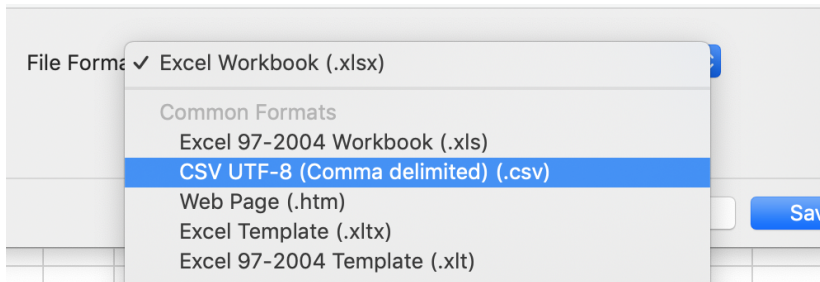


Figure 4: CSV UTF-8 Export in Excel

Outlook: `read_csv` - tidyverse

- ▶ The **readr** package's **`read_csv`** function is one of the tidyverse's cornerstones.
- ▶ It offers a lot of functions to read data in typical formats and allows a wide range of customizations.
- ▶ With `readr`, you will be able to provide a column specification, select a locale for the character set and define custom delimiters.

Tip: Rather than fixing formatting issues in R, it might often be much easier to avoid producing them in Excel in the first place

R and SAS, SPSS, Stata - The haven package

The haven package (<https://haven.tidyverse.org>) offers functionality to read in data sets from SAS, SPSS and Stata.

Example: SAS has a proprietary data format ".sas7bdat"

```
require(haven)
# you can try this with sas_example.sas7bdat
d_sas <- read_sas(data_file = path_sas)
```

A word of warning

There is a good reason why these formats are proprietary. They offer additional functionality beyond CSV, e.g.

- ▶ column names and column labels
- ▶ custom formats for data values

which may render your data mostly useless outside of SAS (or SPSS / STATA) and an export from there to CSV may be preferable.

The foreign Package - SPSS (and more)

The foreign package is another option to read data from other statistical packages.

From the manual: “Read Data Stored by ‘Minitab’, ‘S’, ‘SAS’, ‘SPSS’, ‘Stata’, ‘Systat’, ‘Weka’, ‘dBase’, ...”

Example: SPSS

```
require(foreign)
path_spss <- system.file("files",
                        "electric.sav", package = "foreign")
d_spss <- read.spss(file=path_spss)
```

Case Study: WHO Tuberculosis Data

The WHO Tuberculosis Data Set

The World Health Organization collects yearly information on the number of Tuberculosis cases

<https://www.who.int/tb/country/data/download/en/>

The data has already been pre-processed from a wide into a long-format (more on the gather function for wide to long transformation later).

Tuberculosis

Tuberculosis (TB) is a disease caused by bacteria called Mycobacterium tuberculosis. The bacteria usually attack the lungs, but they can also damage other parts of the body.

Variables in the Tuberculosis Data

- ▶ **country**: country name
- ▶ **iso3**: ISO3 country code
- ▶ **year**: year of data collection
- ▶ **var**: variable describing the severity of tuberculosis
 - ▶ sn = negative pulmonary smear (less infectious)
 - ▶ sp = positive pulmonary smear (more infectious)
- ▶ **sex**: gender group
- ▶ **age group**: (0-14, 15-24, 25-34, 35-44, 45-54, 55-64, 65+)
- ▶ **value**: number of TB cases

Reading the TB Data Set

The data is available as an .rds (R data source). Use **readRDS** to read who.rds into R

```
d_who <- readRDS(paste0(path_to_data, "who.rds"))
```

Assess Data

Assess Basic Data Characteristics

When reading in data there is always a chance something goes wrong. Hence, a couple of basic checks are recommended

Exercise: Check that

- ▶ the data set has 59'162 rows
- ▶ the data set has the 7 columns described earlier
- ▶ the columns have plausible data types (tip: `?str`)

Solution: Assess Basic Characteristics

The data set has 59'162 rows on 7 variables

```
dim(d_who) #nrow(d_who) ncol(d_who)
```

```
## [1] 59162      7
```

The data set contains the columns described earlier

```
colnames(d_who)[1:4]
```

```
## [1] "country" "iso3"    "year"    "var"
```

```
colnames(d_who)[5:7]
```

```
## [1] "sex"    "age"    "value"
```

```
str(d_who)
```

Summary & View

The **summary** function give a summary of each variable. This allowy to quickly check ranges, means etc. for plausibility.

Different column types show different summary metrics.

```
summary(d_who)
```

Also, recall the **View** function to display an interactive view of the data set

```
View(d_who)
```

Assessing Top / Bottom Rows

The **head** and **tail** functions allow to assess the first / last n rows

```
head(d_who) # first 6 rows
```

```
##      country iso3 year var sex age value
## 1 Afghanistan AFG 1997  sp   m  014     0
## 2 Afghanistan AFG 1998  sp   m  014    30
## 3 Afghanistan AFG 1999  sp   m  014     8
## 4 Afghanistan AFG 2000  sp   m  014    52
## 5 Afghanistan AFG 2001  sp   m  014   129
## 6 Afghanistan AFG 2002  sp   m  014    90
```

```
# tail(d_who, n = 10) # last 10 rows
```

Assess Individual Variables

Data can sometimes be messy and impact your analysis.

Tip: Always ask yourself what are the main characteristics your variables should fulfill?

- ▶ Is the **range** of numeric variables plausible?
- ▶ How are numeric variables distributed (use **quantile** or plotting functions such as **hist** for histograms)

```
hist(d_who$year)
```

Assess Individual Variables II

- ▶ How many levels has a char variable / factor? Is there a need to transform the variable and collapse some levels (e.g. “male”, “female”, “m”, “fem”)?
- ▶ How are missing values coded? Is there a need for transformation? Are there too many missing values for a variable to be useful?
- ▶ Which additional variables should be derived from the available columns to support the analysis of research questions?

Assign Types

Assign the Correct Variable Type

Numbers are often not recognized as expected (e.g. because they have separators 1'000 or a character value results in the whole column being read as char).

Use **as.numeric** to transform a char vector into numeric

```
as.numeric(c("2", "3"))
```

Also, dates and times as well as special characters may require additional care.

```
str(d_who)
```

Transform data

Exercise: Assign new Variable Names

Rename

- ▶ var to tb_severity
- ▶ value to tb_cnt

Tip: **colnames** + new assignment at the required sub-position

Solution: Assign new Variable Names

Change Column Names

```
colnames(d_who)[5] <- "tb_severity"  
colnames(d_who)[7] <- "tb_cnt"
```

Absolute Frequencies of Variable Levels

Display for a variable all levels and their counts with **table**

```
table(d_who$age)
```

```
##
```

```
##  014 1524 2534 3544 4554 5564    65
```

```
## 8432 8455 8444 8463 8472 8451 8445
```

Relative Frequencies of Variable Levels

- ▶ Divide the table through the **length** of the vector
- ▶ Multiply by 100 for percent
- ▶ Apply **round** to round values to two digits

```
round(100 * table(d_who$age)/length(d_who$age), 2)  
# same result, but using R's prop.table function  
round(100 * prop.table(table(d_who$age)), 2)
```

Note that WHO used stratified sampling, i.e. they aimed to systematically balance the sub-group sizes.

Recode data

Renaming the Levels of Variable age

The current levels for age are brief, but not very readable. We can use the **ifelse** function for re-coding levels

```
ifelse(vector, if_true_then, if_false_then)
```

```
d_who$age_grp <-  
  ifelse(d_who$age == "014", "0 - 14",  
  ifelse(d_who$age == "1524", "15 - 24",  
  ifelse(d_who$age == "2534", "25 - 34",  
  ifelse(d_who$age == "3544", "35 - 44",  
  ifelse(d_who$age == "4554", "45 - 54",  
  ifelse(d_who$age == "5564", "55 - 64",  
  ifelse(d_who$age == "65", "65 and older",  
  NA # else  
  )))))))
```

Note: This function can be used for any if/else-type decisions

The recode Function

A more compact alternative is the **recode** function

```
require(dplyr)
recode(vector, a = "new_name_for_a",
        b = "new_name_for_b")
```

```
d_who$age_grp2 <- recode(d_who$age,
                        `014` = "0 - 14",
                        `1524` = "15 - 24",
                        `2534` = "25 - 34",
                        `3544` = "35 - 44",
                        `4554` = "45 - 54",
                        `5564` = "55 - 64",
                        `65` = "65 and older"
                        )
```

Comparing Columns

To assess if **age_grp** and **age_grp2** are identical the following code is helpful

```
table(d_who$age_grp == d_who$age_grp2, useNA = "ifany")
```

```
##
```

```
## TRUE
```

```
## 59162
```


Collapsing Levels

The recode function can also be used to collapse levels

```
d_who$age_grp3 <- recode(d_who$age,  
  `014` = "0 - 14",  
  `1524` = "15 - 44",  
  `2534` = "15 - 44",  
  `3544` = "15 - 44",  
  `4554` = "45 and older",  
  `5564` = "45 and older",  
  `65` = "45 and older"  
)
```

The resulting variable has only 3 levels

```
d_who$age_grp3
```

Selecting Records with Differences I

Change observation nbr 17 to a new value

```
d_who$age_grp2[17]
```

```
## [1] "0 - 14"
```

```
d_who$age_grp2[17] <- "65 and older"
```

Selecting Records with Differences II

The **which** function identifies the position of differences !=

```
which(d_who$age_grp != d_who$age_grp2)
```

```
## [1] 17
```

The logical comparison can also directly be used to select the deviating observations

```
d_who[d_who$age_grp != d_who$age_grp2,  
      c("age_grp", "age_grp2")]
```

```
##      age_grp      age_grp2
```

```
## 17  0 - 14 65 and older
```

Joining Data

Joining Data

- ▶ Data may be spread across more than 1 table and needs to be merged. Here, we will extend the WHO data by a second data set which groups the countries into continents.
- ▶ Joining on the country name is difficult, as many different spellings are possible (e.g. Ireland / Republic of Ireland / Ireland, Republic of). Hence, whenever possible, it is advised to join on standardized codes (e.g. ISO country code, patient ID).
- ▶ Time-wise this step may overlap with transforming data (once additional columns are available, more transformations may be necessary).

Reading Continents data

The continents data is stored as a .rds file

```
d_continents <- readRDS(paste0(path_to_data,  
"country_continent.rds"))
```

The data set contains continent names and ISO3-codes

```
head(d_continents[c("Continent_Name",  
"Three_Letter_Country_Code")])
```

	Continent_Name	Three_Letter_Country_Code
## 1	Asia	AFG
## 2	Europe	ALB
## 3	Antarctica	ATA
## 4	Africa	DZA
## 5	Oceania	ASM
## 6	Europe	AND

The merge Function for Joins

The **merge** functions has 4 key arguments to specify which two data sets to join and which of their columns to use for row identification

```
merge(x=data1, y=data2,  
      by.x=vec_of_cols, by.y=vec_of_cols)
```

Here, we join on the respective ISO3-code columns

```
d_who2 <- merge(d_who, d_continents,  
               by.x = "iso3",  
               by.y = "Three_Letter_Country_Code")
```

Assessing Joins

Joining data needs special care to not introduce errors

```
nrow(d_who)
```

```
## [1] 59162
```

```
nrow(d_who2)
```

```
## [1] 61153
```

The count of **d_who2** is larger - hence some kind of duplication happened!

Question: Why might this occur?

De-Duplicate the Continents Data

Check if the continents ISO3-codes are unique

```
(iso3_freq <-  
  table(d_continents$Three_Letter_Country_Code))
```

##

```
##      ABW  AFG  AGO  AIA  ALA  ALB  AND  ANT  ARE  ARG  ARM  ASM  ATA  
##    4    1    1    1    1    1    1    1    1    1    2    1    1  
##  AZE  BDI  BEL  BEN  BES  BFA  BGD  BGR  BHR  BHS  BIH  BLM  BLR  BLZ  
##    2    1    1    1    1    1    1    1    1    1    1    1    1  
##  BRN  BTN  BVT  BWA  CAF  CAN  CCK  CHE  CHL  CHN  CIV  CMR  COD  COG  
##    1    1    1    1    1    1    1    1    1    1    1    1    1  
##  CRI  CUB  CUW  CXR  CYM  CYP  CZE  DEU  DJI  DMA  DNK  DOM  DZA  ECU  
... 
```

De-Duplicate the Continents Data II

Keep only ISO3-codes with frequency 1

```
iso3_freq_1 <- iso3_freq[iso3_freq == 1]
```

Get the matching codes

```
unique_iso3_codes <- names(iso3_freq_1)
```

Keep only continents data rows with unique ISO3-codes

```
d_continents2 <- d_continents[  
  d_continents$Three_Letter_Country_Code  
    %in% unique_iso3_codes,]
```

De-Duplicate the Continents Data III

The resulting continents data set has fewer rows

```
nrow(d_continents)
```

```
## [1] 262
```

```
nrow(d_continents2)
```

```
## [1] 242
```

De-Duplicate the Continents Data IV

Repeat the join with the reduced continents data

```
d_who3 <- merge(d_who, d_continents2,  
               by.x = "iso3",  
               by.y = "Three_Letter_Country_Code")
```

Re-assess the row counts

```
nrow(d_who)
```

```
## [1] 59162
```

```
nrow(d_who3)
```

```
## [1] 56975
```

Now we are missing some records!

Question: Why might this occur?

Recovering the Lost Records

By default the merge function performs an inner join. Thus only ISO3-codes which are present on both sides are kept. The merge function has two options **all.x** and **all.y** to keep also the non-matching rows on either side (left / right / full join).

Repeat the join with **all.x = TRUE** as we want to keep all WHO records even if we do not find a matching continent

```
d_who4 <- merge(d_who, d_continents2,  
               by.x = "iso3",  
               by.y = "Three_Letter_Country_Code",  
               all.x = TRUE)
```

Finally, the row counts match

```
nrow(d_who) == nrow(d_who4)
```

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

Handle missing values

Missing Values: NA

R codes missing values using the special expression **NA**.

Every researcher codes missings in a different way (e.g. 8888 or 9999), hence it is very important to re-code missing values to NA for R to handle them correctly.

Example

```
my_data$my_var[my_data$my_var %in% c(8888,9999)] <- NA
```

Missing Continents in the Merged Data Set

```
table(d_who4$Continent_Name, useNA = "ifany")
```

- ▶ Not all ISO3 codes from d_who could be matched.
- ▶ Also, since Continent_Name is a factor, the level Antarctica is displayed, but no tb cases were record in any country matching this geographic region

Exercise: Count Missings

Use the **sum** function to verify that the number of missing continent observations is 2'187

Tip: use **is.na()**

Solution: Count Missings

```
## R cannot compare NA values using ==  
sum(d_who4$Continent_Name == NA)
```

```
## [1] NA
```

```
## Use is.na() instead  
sum(is.na(d_who4$Continent_Name))
```

```
## [1] 2187
```

Impact of NA Values

- ▶ For some types of analyses in R (e.g. regressions with `lm` (linear model)), **observations with an NA** value on one of the included variables will be **dropped** from the analysis.
- ▶ In other cases, the **NA** records of character variables **may be considered a valid response**, e.g. “did not answer”.
- ▶ With **numeric variables** this is more difficult, as setting a value of 8888 might severely break your model. If you do not have many missing values, setting missing values to **0** or to the **mean** may be a feasible option - **with many missing values this is discouraged**.
- ▶ Techniques for more accurate **missing value imputation** are available in R, but are beyond the scope of the course.

Aggregate Data

Calculate Aggregated Values

Say, we want to report the total number of tb cases per country.

For a single country the below code can be used

```
sum(d_who[d_who$country=="Switzerland", "tb_cnt"])
```

```
## [1] 4369
```

However, using the **unique** function, we see that there are 219 countries in the data set

```
length(unique(d_who4$country))
```

```
## [1] 219
```

We will use **loops** and **functions** to automate the analysis

Calling Functions

When calling arguments by **name**, their **order** can be changed

```
function_name(argument2 = value2, argument1 = value1)
```

When calling a function **without argument name** each value will be matched by **position**

```
function_name(value2, value1)  
# matches value2 to argument1  
#       and value1 to argument2
```

Defining a Function

A **function** in R can be defined as follows

```
function_name <- function(argument1 = value1,  
                           argument2 = value2)  
{  
  ... code_to_execute ...  
}
```

A Function for TB Counts per Country

Function definition

```
tb_cnt_country <- function(country) {  
  sum(d_who4[d_who4$country==country, "tb_cnt"])  
}
```

Function call using the country argument

```
tb_cnt_country(country = "Switzerland")  
  
## [1] 4369
```


Exercise: A Function for TB Counts per Continent

Define a function `tb_cnt_continent` to calculate tb cases per continent

Tips:

- ▶ replace argument `country` by `Continent_Name`
- ▶ be sure to use `d_who4`
- ▶ use `sum(..., na.rm = TRUE)` to ignore rows with missing continents (otherwise the sum will return NA)
- ▶ try the function for “Europe” and “Africa”

Solution: A Function for TB Counts per Continent

Function definition

```
tb_cnt_continent <- function(Continent_Name) {  
  sum(d_who4[d_who4$Continent_Name == Continent_Name,  
           "tb_cnt"], na.rm = TRUE)  
}
```

Function calls

```
tb_cnt_continent("Europe")
```

```
## [1] 1002767
```

```
tb_cnt_continent("Africa")
```

```
## [1] 9442457
```

For Loops

R has 3 different loop types: **for**, **while** and **repeat**

Here, we will use a **for** loop to iterate through the countries. A **for** loop has the following syntax

```
for (i in some_vector) {  
  ... execute code...  
}
```

Example

```
for (i in 1:5)  
  {print(i)}
```

Calculate the TB Count for each Country

Define two helper vectors

```
v_countries <- sort(unique(d_who4$country))  
res <- vector()
```

And calculate the tb count for each country

```
for (i in v_countries) {  
  res[i] <- tb_cnt_country(i)  
}
```

TB Count res Vector

The results are stored in vector res

```
head(res, 3)
```

```
## Afghanistan      Albania      Algeria  
##      136771      3854      120995
```

```
tail(res, 3)
```

```
##      Yemen      Zambia      Zimbabwe  
##      70349      204118      285797
```

```
res["Switzerland"]
```

```
## Switzerland  
##      4369
```

Outlook: More Aggregation Approaches

The aggregate Function

```
res_aggr <- aggregate(d_who$tb_cnt,  
                      by = list(Country = d_who$country),  
                      FUN = sum)
```

```
res_aggr[res_aggr$Country == "Switzerland",]
```

The tapply Function

```
res_tapply <- tapply(d_who$tb_cnt,  
                     d_who$country, FUN=sum)  
  
res_tapply["Switzerland"]
```


The dplyr Package

```
require(dplyr)

# count for Switzerland only
d_who %>%
  filter(country=="Switzerland") %>%
  summarize(tb_cnt_CH = sum(tb_cnt))

# count for all countries
res_dplyr <- d_who %>%
  group_by(country) %>%
  summarize(Frequency = sum(tb_cnt))

res_dplyr[res_dplyr$country == "Switzerland", ]
```

The `%>` operator applies the operation on the right hand side to the data on the left hand side. Hence, multiple operations can be chained into a single expression

The dplyr Package for Data Transformation

The **dplyr** package is the tidyverse's core utility for data transformation and offers 5 core functions:

- ▶ **select**: choose variables to work on
- ▶ **filter**: select observations matching a condition
- ▶ **mutate**: create new variables from existing variables
- ▶ **summarize**: aggregate values, can be combined with **group by** to calculate summary measures for each level of the grouping variable(s)
- ▶ **arrange**: order rows

Transform a Wide Data Set into a Long Data Set

- ▶ WHO collected the d_who originally in a wide format (e.g. every year, month and illness characteristic in a separate column).
- ▶ In the book “R for Data Science”, dplyr together with tidyr’s **gather** and **separate** functions are used to transform the data into d_who.
- ▶ For details, see the case study here:
<https://r4ds.had.co.nz/tidy-data.html#case-study>

A List of Tidyverse Packages

Data preparation and shaping

- ▶ **readr**: reading data from flat files
- ▶ **readxl**: reading data from Excel files
- ▶ **tibble**: a tidy enhancement of R's data.frame class
- ▶ **tidyr**: methods to turn “messy” data into “tidy” data
- ▶ **dplyr**: methods for data transformation
- ▶ **stringr**: working with strings
- ▶ **forcats**: working with factors (categorical variables)
- ▶ **purrr**: tidy functional programming
- ▶ **haven**: reading data from other statistical software

Data visualization

- ▶ **ggplot2**: tidy plotting routines

Further Reading

- ▶ “R for Data Science” provides a lot of information on the Tidyverse. Freely available here: <https://r4ds.had.co.nz>
- ▶ An Introduction to Statistical Learning: with Applications in R (textbook on statistical modelling with a lot of examples)
- ▶ <https://www.r-bloggers.com> - A source for various tutorials on a wide variety of topics