Data Handling: Import, Cleaning and Visualisation

Lecture 4:
Data Storage and Data Structures

Prof. Dr. Ulrich Matter, updated by Dr. Aurélien Sallin

Recap

Computers and text

How can a computer understand text if it only understands `0` s and `1` s?

- *Standards* define how `0` s and `1` s correspond to specific letters/characters of different human languages.
- These standards are usually called *character encodings*.
- Coded character sets that map unique numbers (in the end in binary coded values) to each character in the set.
- For example, ASCII (American Standard Code for Information Interchange).

ASCII logo. (public domain).

Computer Code and Data Storage

Computer code

- Instructions to a computer, in a language it understands… (R).
- Code is written to *text files*.
- Text is 'translated' into 0s and 1s which the CPU can process.

Data storage

- Data usually stored in *text files*
    - Read data from text files: data import.
    - Write data to text files: data export.

Unstructured data in text files

- Store `Hello World!` in `helloworld.txt` .
    - Allocation of a block of computer memory containing `Hello World!` .
    - Simply a sequence of `0` s and `1` s…
    - `.txt` indicates to the operating system which program to use when opening this file.
- Encoding and format tell the computer how to interpret the `0` s and `1` s.

Inspect a text file

You can use the terminal in RStudio (on Nuvolos) to inspect a text file. Via `cat` you can display the content of a text file interpreted as text (using the system's standard encoding).

```
cat helloworld.txt; echo
```

```
## Hello World!
```

Or, from the R-console:

```
system("cat helloworld.txt")
```

With another application called `xxd`, you can directly look at the `0` s and `1` s stored in a file.

```
xxd -b helloworld.txt
```

```
00000000: 4865 6c6c 6f20 576f 726c 6421            Hello World!
```

```
00000000: 01001000 01100101 01101100 01101100 01101111 00100000  Hello
00000006: 01010111 01101111 01110010 01101100 01100100 00100001  World!
```

Encoding issues

If there is a mismatch between the encoding in which a file was created and the encoding used to look at the file, you will likely encounter some unexpected symbols in the middle of the text. Consider the following example:

```
cat hastamanana.txt; echo
```

```
## Hasta Ma?ana!
```

Likely, this text file has not been created with the same character encoding as the one used by your system to interpret the underlying 0s and 1s.

```
file -b hastamanana.txt
```

Use the correct encoding

Read the file again, this time with the correct encoding

```
iconv -f iso-8859-1 -t utf-8 hastamanana.txt | cat
```

```
## Hasta Mañana!
```

UTF encodings

- 'Universal' standards.
- Contain broad variety of symbols (various languages).
- Less problems with newer data sources…

Take-away message

- *Recognize an encoding issue when it occurs!*
- Problem occurs right at the beginning of the *data pipeline*!
    - Rest of pipeline affected…
    - … cleaning of data fails …
    - … analysis suffers.

From text to data structure

Structured Data Formats

- Still text files, but with standardized *structure*.
- *Special characters* define the structure.
- More complex *syntax*, more complex structures can be represented…

Table-like formats

Example `ch_gdp.csv`.

```
year,gdp_chfb
1980,184
1985,244
1990,331
1995,374
2000,422
2005,464
```

What is the structure?

Table-like formats

- What is the reocurring pattern?
  - Special character `,`
  - New lines
- Table is visible from structure in raw text file…

How can we instruct a computer to read this text as a table?

A simple parser algorithm

1. Start with an empty table consisting of one cell (1 row/column).
2. While the end of the input file is not yet reached, do the following:
   - Read characters from the input file, and add them one-by-one to the current cell.
     - If you encounter the character '`,`', ignore it, create a new field, and jump to the new field.
   - If you encounter the end of the line, create a new row and jump to the new row.

CSVs and fixed-width format

- *'Comma-Separated Values'* (therefore `.csv`)
  - commas separate values
  - new lines separate rows/observations
  - (many related formats with other separators)
- Instructions of how to read a `.csv`-file: *CSV parser*.

CSVs and fixed-width format

- Common format to store and transfer data.
  - Very common in a data analysis context.
- Natural format/structure when the dataset can be thought of as a table.

CSVs and fixed-width format

How does the computer know that the end of a line is reached?

End-of-line characters

```
xxd ch_gdp.csv
```

```
00000000: efbb bf79 6561 722c 6764 705f 6368 6662  ...year,gdp_chfb
00000010: 0d31 3938 302c 3138 340d 3139 3835 2c32  .1980,184.1985,2
00000020: 3434 0d31 3939 302c 3333 310d 3139 3935  44.1990,331.1995
00000030: 2c33 3734 0d32 3030 302c 3432 320d 3230  ,374.2000,422.20
00000040: 3035 2c34 3634                           05,464
```

- `,` (`2c`): indicates comma for new column.
- `.` (`0d`): indicates end of line!

Related formats

- Other delimiters (`;` , tabs, etc.)
- Fixed (column) width

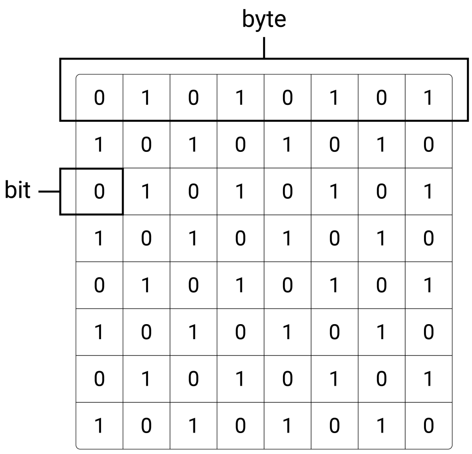
More complex formats

- N-dimensional data
- Nested data
- *XML*, *JSON*, *YAML*, etc.
    - Often encountered online!
    - (Next lecture!)

Units of Information/Data Storage

Bit, Byte

- Smallest unit (a `0` or a `1`): *bit* (from *bi*nary dig*it*; abbrev. ‘b’).
- *Byte* (1 byte = 8 bits; abbrev. ‘B’)
    - For example, `10001011` (`139`)



Bigger units for storage capacity

- $1 \text{ kilobyte (KB)} = 1000^1 \text{ bytes}$
- $1 \text{ megabyte (MB)} = 1000^2 \text{ bytes}$
- $1 \text{ gigabyte (GB)} = 1000^3 \text{ bytes}$

Data Structures and Data Types in R

Structures to work with…

- Data structures for storage on hard drive (e.g., csv).

- Representation of data in RAM (e.g. as an R-object)?
    - What is the representation of the ‘structure’ once the data is parsed (read into RAM)?

Structures to work with (in R)

We distinguish two basic characteristics:

1. Data types:
    - *integers*;
    - *real numbers* (‘numeric values’, ‘doubles’, floating point numbers);
    - *characters* (‘string’, ‘character values’);
    - (*booleans*)
2. Basic **data structures** in RAM:
    - *Vectors*
    - *Factors*
    - *Arrays/Matrices*
    - *Lists*
    - *Data frames* (very `R` -specific)

Describe data

The `type` and the `class` of an object can be used to describe an object.

- *type*: technical and low-level description of the actual storage mode or physical representation of an object.
    - It tells *how the object is stored in memory*.
- *class*: attribute about the nature of an `R` object.
    - It tells you *how to treat the object in a broad sense*.

Data types: numeric and integers

Data types: numeric and integers

R interprets these bytes of data as type `double` (‘numeric’) or type `integer` :

```
a <- 1.5
b <- 3
c <- 3L
```

```
a <- 1.5
typeof(a); class(a)
```

```
## [1] "double"
```

```
## [1] "numeric"
```

```
c <- 3L
typeof(c); class(c)
```

```
## [1] "integer"
```

```
## [1] "integer"
```

Given that these bytes of data are interpreted as numeric, we can use operators (here: math operators) that can work with such functions:

```
a + b
```

```
## [1] 4.5
```

Data types: character

```
a <- "1.5"
b <- "3"
```

```
typeof(a)
```

```
## [1] "character"
```

```
class(a)
```

```
## [1] "character"
```

With data type character, R interprets the values as text. Consequently, the same line of code as above will result in an error:

```
a + b
```

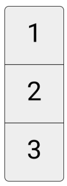
```
## Error in a + b: nicht-numerisches Argument für binären Operator
```

Data types: special values

- `NA` : “Not available”, i.e. missing value for any type
- `NaN` : “Not a number”: special case of NA for numeric
- `Inf` : specific to numeric
- `NULL` : absence of value

Data structures: vectors

- Collections of value of same type



Examples in R: initiate a character vector and an integer vector with three elements (vectors of “length” 3).

```
persons <- c("Andy", "Brian", "Claire")
persons
```

```
## [1] "Andy"   "Brian"  "Claire"
```

```
ages <- c(24, 50, 30)
ages
```

```
## [1] 24 50 30
```

What happens when you create a vector out of `persons` and `ages` ?

```
c(persons, ages)
```

The two types are not identical. Thus, R will automatically convert one type to fit the other. In this case, R converts the ages from numeric to factor.

Data structures: factors

- Factors are sets of categories.
- The values come from a fixed set of possible values.

Male
Male
Female

Example in R: initiate a factor variable based on a character vector.

```
gender <- factor(c("Male", "Male", "Female"))
gender
```

```
## [1] Male   Male   Female
## Levels: Female Male
```

What is the difference between this definition as a factor and the simple definition as a character vector? Factors are typically used to work with categorical variables. Under the hood, R stores the labels of each category (here: `"Male"` or `"Female"`), and then refers to factor elements via category ids. This makes the storage and filtering/access of categorical variables much more efficient (in particular, if there are very long/complex category names/labels contained in the data). Many of the high-level functions you will work with later in the course (e.g. to visualize data) expect you to define categorical variables in a dataset explicitly with factors.

In other words, for `R` , factors are “disguised” integers…

```
typeof(gender)
```

```
## [1] "integer"
```

… and contain two components:

- the integer (or “levels”);
- the labels.

```
levels(gender)
```

```
## [1] "Female" "Male"
```

Data structures: matrices

Matrices are two-dimensional collections of values of the same type

1	4	7
2	5	8
3	6	9

Example in R: initialize a matrix with three rows and two columns.

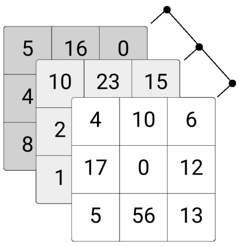
```
my_matrix <- matrix(c(1,2,3,4,5,6), nrow = 3)
my_matrix
```

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

From the code example we see that R considers matrices essentially as two-dimensional vectors. We first initiate a vector and then define in how many rows/columns this vector of values should be organized. Importantly, like vectors, matrices can only contain values of the same data type.

Data structures: arrays

Arrays are higherdimensional collections of values of the same type



Example in R: initialize a three-dimensional array.

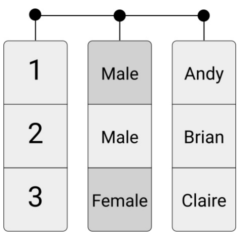
```
my_array <- array(c(1,2,3,4,5,6,7,8), dim = c(2,2,2))
my_array
```

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

Arrays in R are very similar to matrices. Essentially, they just allow us to store data in more than two dimensions (via the `dim` -parameter). Importantly, like vectors/matrices, arrays can only contain values of the same data type.

Data frames, tibbles, and data tables

- Each column contains a vector of a given data type (or factor), but all columns need to be of identical length.
- `data.frame` , `tibble` , `data.table`



Example in R: initiate a data frame with the three columns “person”, “age”, and “gender”.

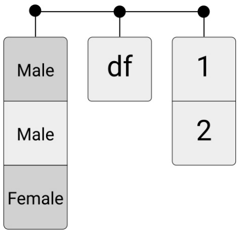
```
df <- data.frame(person = persons, age = ages, gender = gender)
df
```

```
##   person age gender
## 1   Andy  24   Male
## 2  Brian  50   Male
## 3 Claire  30 Female
```

Data frames/tibbles are the typical object class to store entire datasets for data analytics purposes in R. Columns of a data frames can contain vectors of different data types. However, each of the columns must contain the same number of elements. In a data analytics context, we often refer to the columns as the “variables” in a dataset and the rows as the “observations” of a data set. Variables describe the characteristics of the observations.

Data structures: lists

Lists can contain different data types in each element, or even different data structures of different dimensions.



Example in R: initiate a list containing an array, a matrix, and a data frame.

```
my_list <- list(my_array, my_matrix, df)
my_list
```

```
## [[1]]
## , , 1
##
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
##
## , , 2
##
##      [,1] [,2]
## [1,]    5    7
## [2,]    6    8
##
##
## [[2]]
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
##
## [[3]]
##   person age gender
## 1   Andy  24   Male
## 2  Brian  50   Male
## 3 Claire  30 Female
```

Lists are the most flexible general object class in R to store data. Each element of a list can contain different types of objects/data, and the length of each element can differ.