Data Handling: Import, Cleaning and Visualisation

Lecture 4:

Data Storage and Data Structures

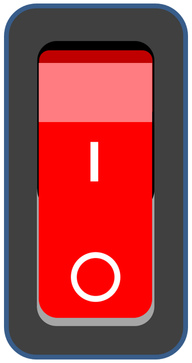
Prof. Dr. Ulrich Matter

The binary system

Microprocessors can only represent two signs (states):

'Off' = 0

'On' = 1



The binary counting frame

Only two signs: `0`, `1`.

Base 2.

Columns: $2^0 = 1$, $2^1 = 2$, $2^2 = 4$, and so forth.

Decimal numbers in a computer

Number	128	64	32	16	8	4	2	1

Decimal numbers in a computer

Number	128	64	32	16	8	4	2	1
0 =	0	0	0	0	0	0	0	0
1 =	0	0	0	0	0	0	0	1
2 =	0	0	0	0	0	0	1	0
3 =	0	0	0	0	0	0	1	1
...								
139 =	1	0	0	0	1	0	1	1

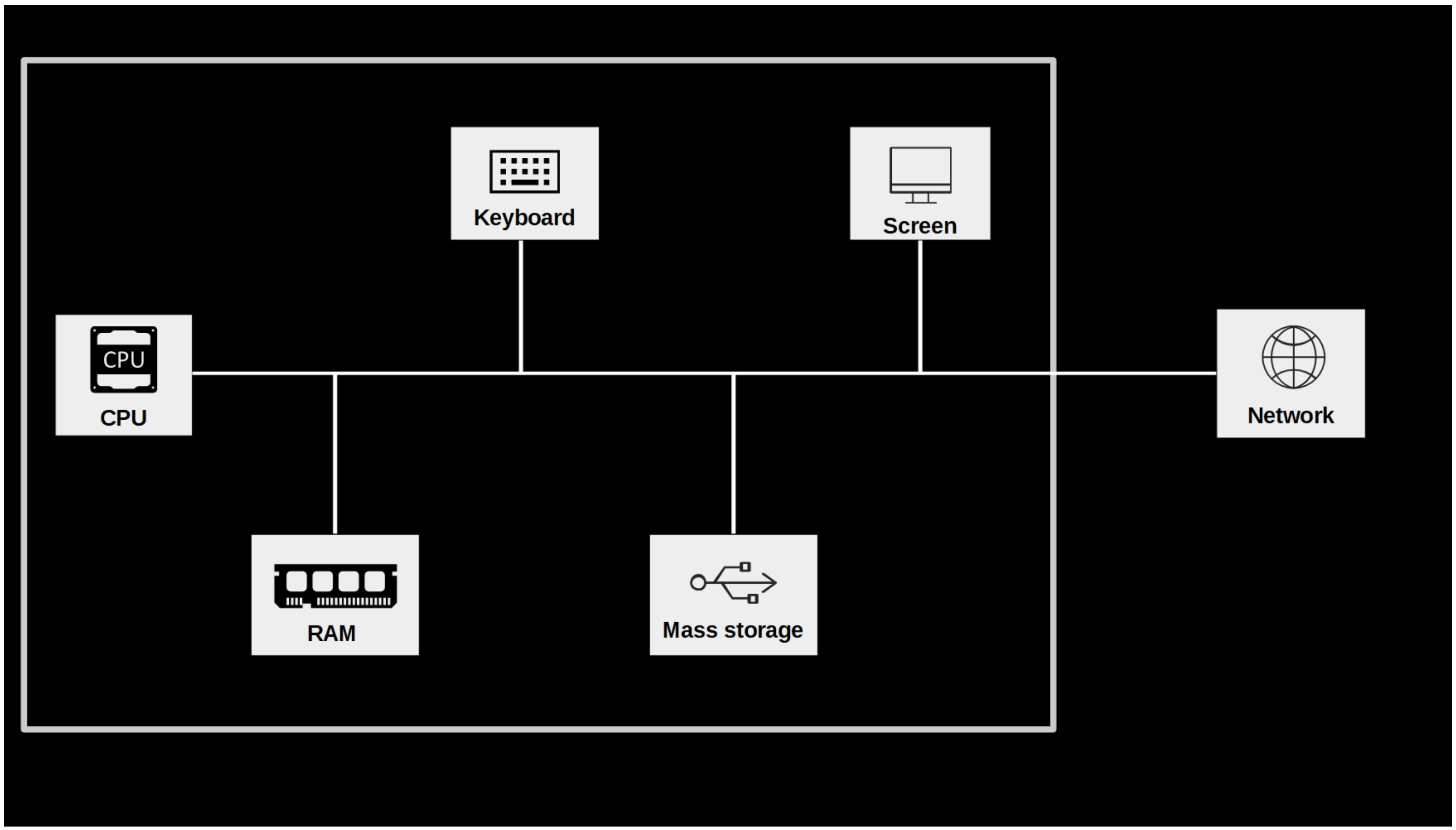
Computers and text

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).

Digital data processing



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

Interpreting `0`s and `1`s as text…

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

```
## Hello World!
```

Or, from the R-console:

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

Inspect a text file

Directly looking at the `0`s and `1`s…

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

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

Inspect a text file

Similarly we can display the content in hexadecimal values:

```
xxd  data/helloworld.txt
```

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

Encoding issues

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

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

What is the problem?

Encoding issues

Inspect the encoding

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

```
## ISO-8859 text
```

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.

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

CSVs and fixed-width format

'Comma-Separated Values' (therefore `.csv`)

- commas separate values
- other delimiters (`;`, tabs, etc.) possible
- 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.

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

–

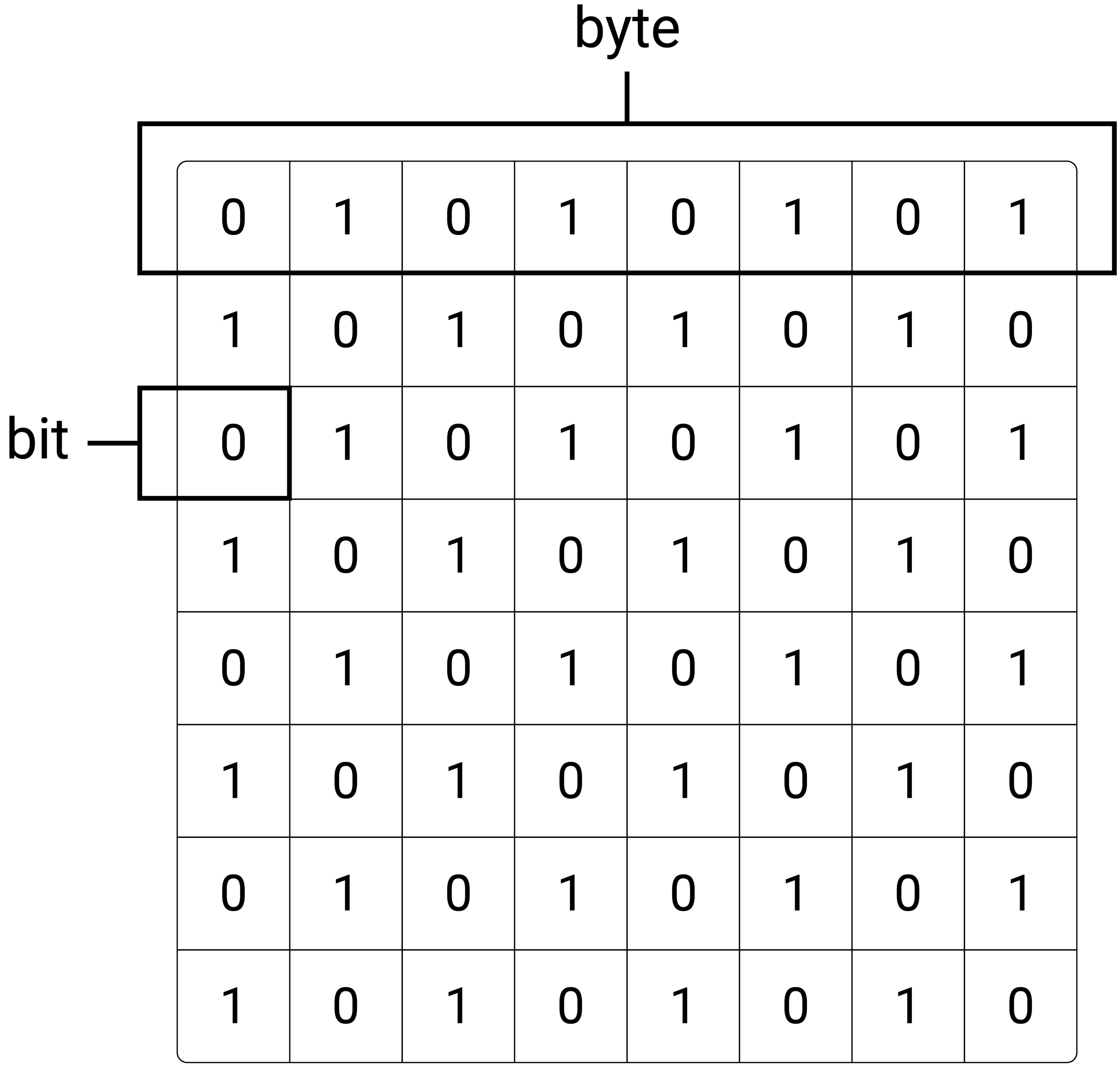
More complex formats

N-dimensional data

Nested data

XML, **JSON**, **YAML**, etc.

- Often encountered online!
- (Next lecture!)


byte
bit
0 1 0 1 0 1 0 1
1 0 1 0 1 0 1 0
0 1 0 1 0 1 0 1
1 0 1 0 1 0 1 0
0 1 0 1 0 1 0 1
1 0 1 0 1 0 1 0
0 1 0 1 0 1 0 1
1 0 1 0 1 0 1 0

Bigger units for storage capacity

1 kilobyte (KB) = 1000^1 bytes

1 megabyte (MB) = 1000^2 bytes

1 gigabyte (GB) = 1000^3 bytes

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’, floating point numbers); text (‘string’, ‘character values’).

Structures to work with (in R)

We distinguish two basic characteristics:

1. Data types: integers; real numbers (‘numeric values’, floating point numbers); text (‘string’, ‘character values’).
2. Basic data structures in RAM:

    Vectors

    Factors

    Arrays/Matrices

    Lists

    Data frames (very `R`-specific)

Data types: numeric

```r
a <- 1.5
b <- 3
```

R interprets this data as type `double` (class ‘numeric’):

```r
typeof(a)
```

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

```r
class(a)
```

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

Data types: numeric

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

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

```r
typeof(a)
```

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

```r
class(a)
```

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

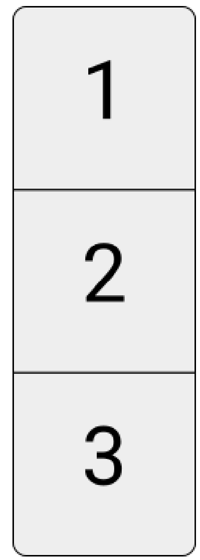
Data types: character

Now the same line of code as above will result in an error:

```
a + b
```

```
## Error in a + b: non-numeric argument to binary operator
```

Data structures: vectors



Data structures: vectors

Example:

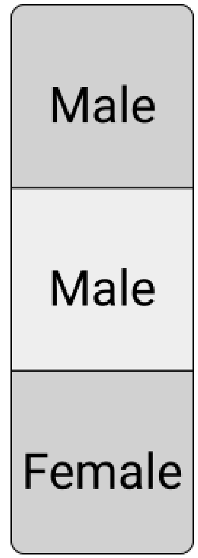
```
persons <- c("Andy", "Brian", "Claire")
persons
```

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

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

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

Data structures: factors



Data structures: factors

Example:

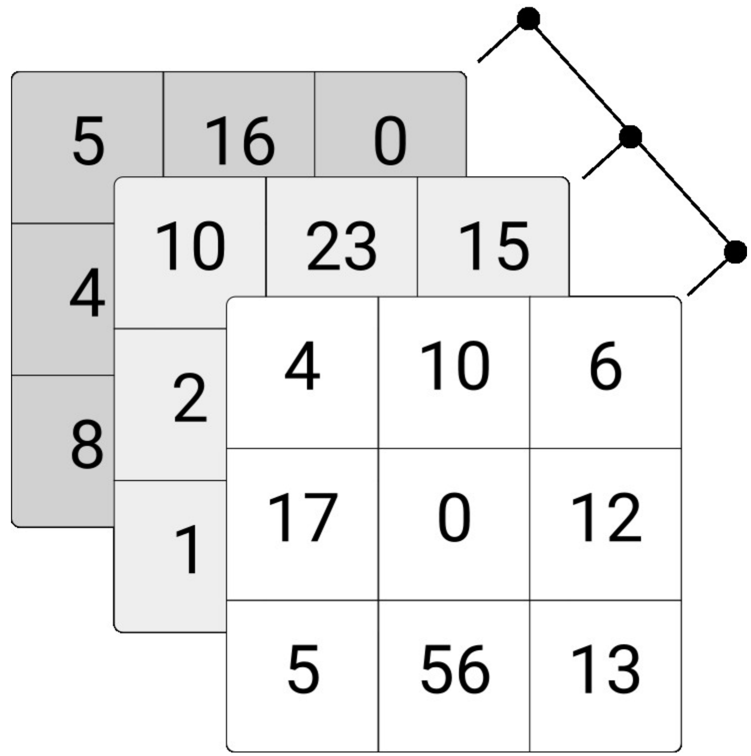
```
gender <- factor(c("Male", "Male", "Female"))
gender
```

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

Data structures: matrices

1	4	7
2	5	8
3	6	9

Data structures: arrays



Data structures: matrices/arrays

Example:

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

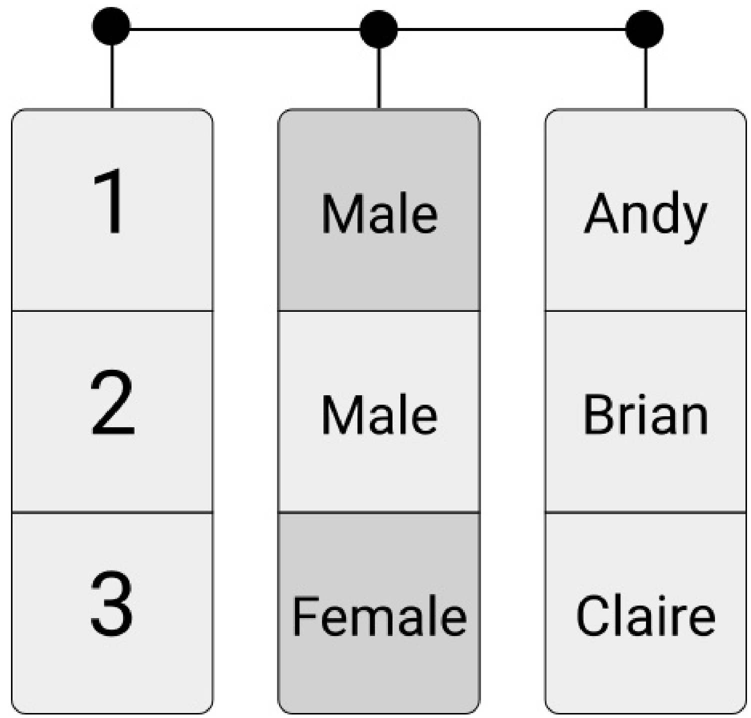
```
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
```

Data structures: matrices/arrays

```
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
```

Data frames, tibbles, and data tables



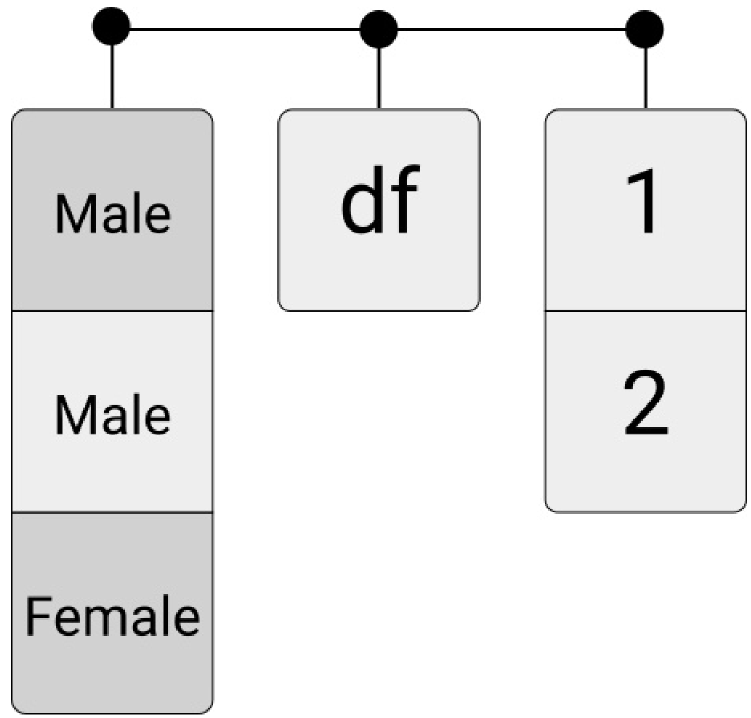
Data frames, tibbles, and data tables

Example:

```
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 structures: lists



Data structures: lists

Example:

```
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]]
```

References