Data Handling: Import, Cleaning and Visualisation

Lecture 3:

Data Storage and Data Structures

Prof. Dr. Ulrich Matter
01/10/2020

Updates

Next week: online only lecture!

- The lecture on **8 October, 2020** will be recorded and made available via Canvas (with additional information).
- There will be **no lecture in 02-001 on that day**.
- On **15 October** we will be back in 02-001.

See update of syllabus/schedule

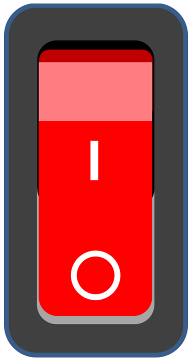
Date	Topic
17.09.20	Introduction: Big Data/Data Science, course overview
24.09.20	An introduction to data and data processing
24.09.20	Exercises/Workshop 1: Tools, working with text files
01.10.20	Data storage and data structures
08.10.20	Research Insights
08.10.20	Exercises/Workshop 2: Computer code and data storage
15.10.20	Big Data from the Web

Recap

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

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

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

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

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

The hexadecimal system

- Binary numbers can become quite long rather quickly.
- Computer Science: refer to binary numbers with the **hexadecimal** system.
- 16 symbols:
    - `0`-`9` (used like in the decimal system)…
    - and `A`-`F` (for the numbers 10 to 15).
- 16 symbols: base 16: each digit represents an increasing power of 16 (16^0, 16^1, etc.).

Floating point numbers

- Why would $1/3$ be an infinate string of ’0’s and ’1’s?
- Similar to manually writing our $1/3 = 0.3333....$
- Floating point numbers: formulaic representation of real numbers (approximatively).
  - –> example in R
  - Short YouTube video

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 Table

Binary	Hexadecimal	Decimal	Character
0011 1111	3F	63	?
0100 0001	41	65	A
0110 0010	62	98	b

Putting the pieces together…

Two core themes of this course:

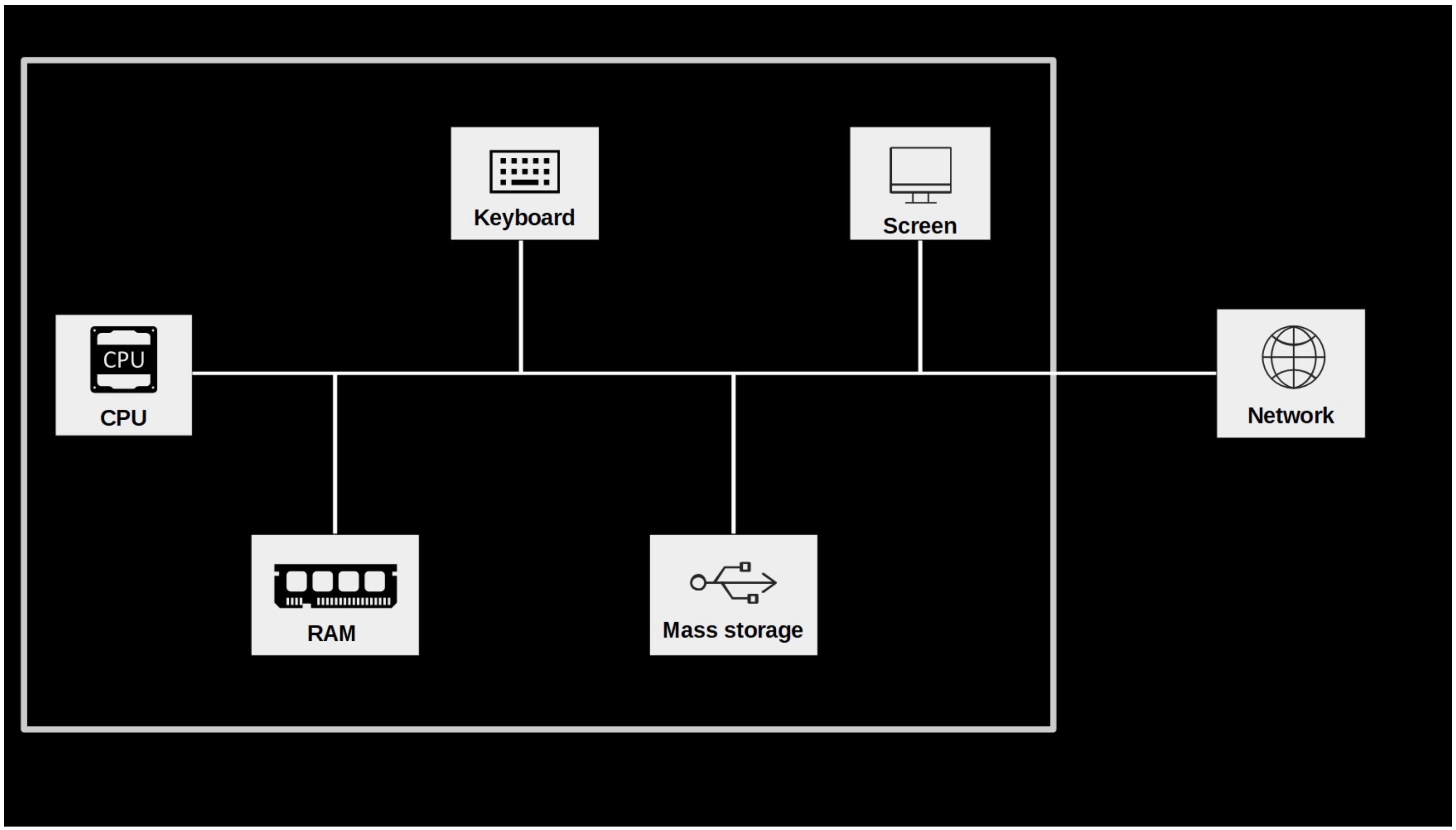
1. How can **data** be **stored** digitally and be **read** by/imported to a computer?
2. How can we give instructions to a computer by writing **computer code**?

In both of these domains we mainly work with one simple type of document: **text files**.

Text-files

- A **collection of characters** stored in a designated part of the computer memory/hard drive.
- A easy to read representation of the underlying information (`0`s and `1`s)!
- Common device to store data:
    - Structured data (tables)
    - Semi-structured data (websites)
    - Unstructured data (plain text)
- Typical device to store computer code.

Digital data processing



Putting the pieces together…

Recall the initial example (survey) of this course.

1. Access a website (over the Internet), use keyboard to enter data into a website (a Google sheet in that case).

2. R program accesses the data of the Google sheet (again over the Internet), download the data, and load it into RAM.

3. Data processing: produce output (in the form of statistics/plots), output on screen.

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

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 variaty 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

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:

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.

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.

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
```

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
```

- `.` (`0d`): indicates end of line!

Related formats

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

More complex formats

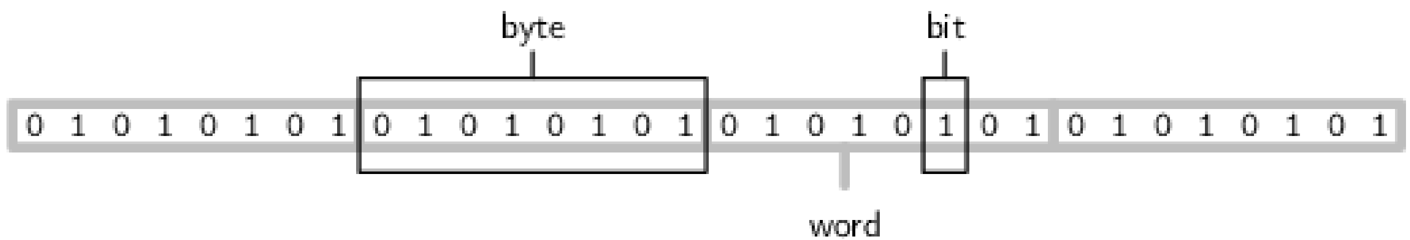
- N-dimensional data
- Nested data
- XML, JSON, YAML, etc.
    - Often encountered online!
    - (Next lecture!)

Units of Information/Data Storage

Bit, Byte, Word

- 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`)
- 4 bytes (or 32 bits) are called a **word**.

Bit, Byte, Word



Bigger units for storage capacity

- 1 kilobyte (KB) = 1000^1 bytes
- 1 megabyte (MB) = 1000^2 bytes
- 1 gigabyte (GB) = 1000^3 bytes

Common units for data transfer (over a network)

- 1 kilobit per second (kbit/s) = 1000^1 bit/s
- 1 megabit per second (mbit/s) = 1000^2 bit/s
- 1 gigabit per second (gbit/s) = 1000^3 bit/s

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’, 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

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

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

```
typeof(a)
```

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

```
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

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

```
typeof(a)
```

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

```
class(a)
```

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

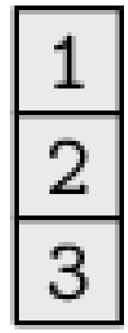
Data types: character

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:

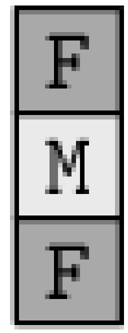
```
persons <- c("Andy", "Brian", "Claire")
persons
```

```
## [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/arrays

1	4	7
2	5	8
3	6	9

Data structures: matrices/arrays

Example:

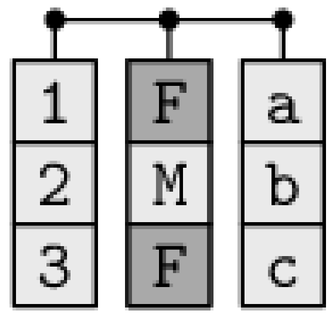
```
my_matrix <- matrix(c(1,2,3,4,5,6), nrow = 3)
my_matrix
```

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

```
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



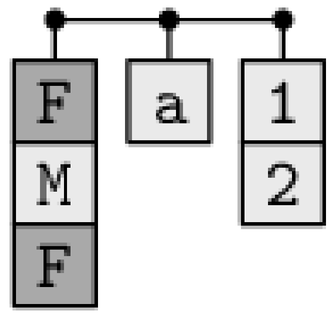
Data frames, tibbles, and data tables

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]]
```

Q&A

References

Murrell, Paul. 2009. Introduction to Data Technologies. London, UK: CRC Press.