Data Handling: Import, Cleaning and Visualisation

Lecture 6:
Non-rectangular data

Dr. Aurélien Sallin

2024-10-24

Recap

Goals of last lecture

- Understand how we import rectangular data into R
- Be familiar with the way csv parsers work
- Exercise: import, read, and manipulate financial data from a text file

Recap: parsing CSVs in R

- `readr`, `read_csv()`, `read_delim()` to parse csvs
  - Return a `tibble`.
- parsing, encoding, and guessing the right type: `guess_parser()`

```
read_csv('A,B
         12:00, 12:00
         14:30, midnight
         20:01, noon')
```

```
# A tibble: 3 × 2
  A      B
  <time> <chr>
1 12:00  12:00
2 14:30  midnight
3 20:01  noon
```

Recap: tibbles and data frames

- tibbles in `tidyverse()`
- basic manipulation of data frames and tibbles
- built-in datasets

Recap: exercise

Read financial data with another separator and with encoding problems.

```
financial_data <- read.csv("financial_data.txt", sep = ":")
financial_data <- read_delim("financial_data.txt", delim = ":")
```

- Error with `read.csv()`: “invalid multibyte string at ”.
- No error with `read_delim()`: more robust to encoding issues!

Recap: exercise

With `iconv()`, we found the right encoding for `\xF6`. We could then import the data using

```
iconv("\xF6", from = "ISO-8859-1", to = "UTF-8")
financial_data <- read.csv("financial_data.txt",
                           sep = ":",
                           fileEncoding = "ISO-8859-1")
```

However, because of the special character, the column was read as a character. We replaced the value and coerced the whole column to numeric.

```
financial_data[10, "Revenue"] <- 1933
financial_data[, "Revenue"] <- as.numeric(financial_data[, "Revenue"])
```

Coercion with comparison operators (but not with mathematical operators)

Last time, we saw that...

```
2 == "2"
```

```
[1] TRUE
```

Using:

```
?"=="
```

“If the two arguments are atomic vectors of different types, one is coerced to the type of the other, the (decreasing) order of precedence being character, complex, numeric, integer, logical and raw.”

... which is for me a good reminder that...

Warm-up

Coercion of boolean values

This question checks your understanding of the coercion of boolean values.

What is the output of the following code?

```
my_vector <- c(1, 0, 3, -1)
as.numeric(my_vector > 0)
```

- `"TRUE" "FALSE" "TRUE" "FALSE"`
- `TRUE FALSE TRUE FALSE`
- `1 0 1 0`
- `0 1 0 1`

Data frames operations

Consider the data frame

```
dataCHframe <- data.frame(
  "City" = c("St.Gallen", "Lausanne", "Zürich"),
  "PartyLeft" = c(35, 45, 55),
  "PartyRight" = c(40, 35, 30)
)
```

Which of these statements are TRUE?

- `dataCHframe$PartyCenter <- c(25, 20, 15)` creates a new variable called “PartyCenter”
- `dim(dataCHframe[, dataCHframe$PartyLeft > 40])` returns the same as `dim(dataCHframe[, c(2,3)])`
- `dim(dataCHframe[dataCHframe$PartyLeft > 40 | dataCHframe$PartyLeft < 40, ])` returns `c(3,3)`
- `dataCHframe` is a data.frame, which is a list consisting of one named character vector and two named integer vectors.

Working with csv files

You want to import a file using `read_delim()`. Describe what `read_delim()` does under the hood. What should be added to this command in order for it to work?

Tibbles

Consider the following code

```
df <- data.frame(a = c(1,2,3,4),
                 b = c("au", "de", "ch", "li"))
```

Are these statements TRUE or FALSE?

- `mean(df$a) == 2.5`
- `typeof(as.matrix(df)[,1])` is `numeric` (or `double`)
- `as_tibble(df)[1:2, 1]` contains the same information as `df[1:2, 1]` (FALSE, because tibbles do not simplify when subsetting. Not exam relevant.)

Today

Goals of today's lecture

- Understand non-rectangular data: xml, json, and html (exercises and next time)
- Be familiar with the way we parse these data into R

- "Non-Rectangular Data in Economic Research" with Minna Heim

Updates

- Exam for exchange students 🎁: **19.12.2024** at 16:15 in room 01-207.
- The mock exam is available on Canvas
- The reading list will be updated in the break

Non-rectangular data structures

Non-rectangular data

- Hierarchical data (xml, html, json)
  - XML and JSON (useful for complex/high-dimensional data sets).
  - HTML (a markup language to define the structure and layout of webpages).
- Unstructured text data
- Images/Pictures data

A rectangular data set

father	mother	name	age	gender
		John	33	male
		Julia	32	female
John	Julia	Jack	6	male
John	Julia	Jill	4	female
John	Julia	John jnr	2	male
		David	45	male
		Debbie	42	female
David	Debbie	Donald	16	male
David	Debbie	Dianne	12	female

👓 What is the data about?

A rectangular data set

father	mother	name	age	gender
		John	33	male
		Julia	32	female
John	Julia	Jack	6	male
John	Julia	Jill	4	female
John	Julia	John jnr	2	male
		David	45	male
		Debbie	42	female
David	Debbie	Donald	16	male
David	Debbie	Dianne	12	female

Which observations belong together?

A rectangular data set

```
father mother  name      age  gender
               John       33  male
               Julia      32  female
John   Julia   Jack        6  male
John   Julia   Jill        4  female
John   Julia   John jnr    2  male
               David      45  male
               Debbie     42  female
David  Debbie  Donald     16  male
David  Debbie  Dianne     12  female
```

Can a parser understand which observations belong together?

Deciphering XML

XML - eXtensible Markup Language

- Since 1998
- Real life examples:
  - Microsoft Office uses the XML-based file formats, such as .docx, .xlsx, and .pptx. See a XML representation of a Word document on the Wikipedia page on Microsoft Office XML format.
  - Store application configuration data
  - Store geospatial data in Geographic Information Systems (GIS)
  - ...

COVID-19 data in XML format

What features does the format have? What is its logic/syntax?

```xml
<?xml version="1.0" encoding="UTF-8"?>
<data>
  <row>
    <unique_id>216498</unique_id>
    <indicator_id>386</indicator_id>
    <name>Ozone (O3)</name>
    <measure>Mean</measure>
    <measure_info>ppb</measure_info>
    <geo_type_name>CD</geo_type_name>
    <geo_join_id>313</geo_join_id>
    <geo_place_name>Coney Island (CD13)</geo_place_name>
    <time_period>Summer 2013</time_period>
    <start_date>2013-06-01T00:00:00</start_date>
    <data_value>34.64</data_value>
  </row>

  <row>
    <unique_id>216499</unique_id>
    <indicator_id>386</indicator_id>
    ...
  </row>
</data>
```

XML declaration

An XML document begins with some information about XML itself. For example, it might mention the XML version that it follows. This opening is called an XML declaration.

```
<?xml version="1.0" encoding="UTF-8"?>
```

XML has a nested syntax

The “row-content” is nested between the ‘`row`’-tags:

```
<row>
...
</row>
```

XML has a nested syntax… similar to a tree structure

```
<root>
  <child>
    <subchild>.....</subchild>
  </child>
</root>
```

XML syntax: Temperature Data example

There are two principal ways to link variable names to values.

```
<?xml version="1.0" encoding="UTF-8"?>
<dataset>
  <variable>Monthly Surface Clear-sky Temperature (ISCCP) (Celsius)</variable>
  <filename>ISCCPMonthly_avg.nc</filename>
  <filepath>/usr/local/fer_data/data/</filepath<|meta_end|>filepath>
  <badflag>-1.E+34</badflag>
  <subset>48 points (TIME)</subset>
  <longitude>123.8W(-123.8)</longitude>
  <latitude>48.8S</latitude>
  <case date="16-JAN-1994" temperature="9.200012" />
  <case date="16-FEB-1994" temperature="10.70001" />
  <case date="16-MAR-1994" temperature="7.5" />
  <case date="16-APR-1994" temperature="8.100006" />
</dataset>
```

XML syntax

1. Define opening and closing XML-tags with the variable name and surround the value with them, such as in `<filename>ISCCPMonthly_avg.nc</filename>`.
2. Encapsulate the values within one tag by defining tag-attributes such as in `<case date="16-JAN-1994" temperature="9.200012" />`.

XML syntax

Attributes-based:

```
<case date="16-JAN-1994" temperature="9.200012" />
<case date="16-FEB-1994" temperature="10.70001" />
<case date="16-MAR-1994" temperature="7.5" />
<case date="16-APR-1994" temperature="8.100006" />
```

XML syntax

Tag-based:

```
<cases>
  <case>
    <date>16-JAN-1994<date/>
    <temperature>9.200012<temperature/>
  <case/>
  <case>
    <date>16-FEB-1994<date/>
    <temperature>10.70001<temperature/>
  <case/>
  <case>
    <date>16-MAR-1994<date/>
    <temperature>7.5<temperature/>
  <case/>
  <case>
    <date>16-APR-1994<date/>
    <temperature>8.100006<temperature/>
  <case/>
<cases/>
```

Insights: CSV vs. XML

- Represent much more *complex (multi-dimensional)* data in XML-files than what is possible in CSVs.
- Self-explanatory syntax: *machine-readable and human-readable*.
- Tags are part of the syntax, give both structure and name variables.

Insights: CSV vs. XML

Potential drawback of XML: **inefficient storage**

- Tags are part of the syntax, thus, part of the actual file.
  - Tags (variable labels) are *repeated* again and again!
  - CSV: variable labels are mentioned once.
  - Potential solution: data compression (e.g., zip).
- If the data is actually two dimensional, a CSV is more practical.

Parsing XML

XML in R with *The Office*

```
<?xml version="1.0" encoding="UTF-8"?>
<company_dundermifflin>
  <person id="1">
    <name>Michael Scott</name>
    <position>Regional Manager</position>
    <location branch="Scranton"/>
  </person>
  <person id="2">
    <name>Dwight Schrutte</name>
    <position>Assistant (to the) Regional Manager</position>
    <location branch="Scranton"/>
    <orders>
      <sales>
        <units>10</units>
        <product>paper A4</product>
      </sales>
    </orders>
  </person>
  <person id="3">
    <name>Jim Halpert</name>
    <position>Sales Representative</position>
    <location branch="Scranton"/>
    <orders>
```

Load a XML file in R

```
# load packages
library(xml2) # updated and faster than library `XML`

# parse XML, represent XML document as R object
xml_doc <- read_xml("company_dundermifflin.xml")
xml_doc
```

```
{xml_document}
<company_dundermifflin>
[1] <person id="1">\n  <name>Michael Scott</name>\n  <position>Regional Manager</position>\n  <lo ...
[2] <person id="2">\n  <name>Dwight Schrutte</name>\n  <position>Assistant (to the) Regional Mana ...
[3] <person id="3">\n  <name>Jim Halpert</name>\n  <position>Sales Representative</position>\n  < ...
```

```
# length of the html
xml_length(xml_doc)
```

```
[1] 3
```

Identify the root-node and the children

‘company_dundermifflin’ is the root-node, ‘persons’ are its children:

```
# navigate downwards
persons <- xml_children(xml_doc) # returns only elements
persons
```

```
{xml_nodeset (3)}
[1] <person id="1">\n  <name>Michael Scott</name>\n  <position>Regional Manager</position>\n  <lo ...
[2] <person id="2">\n  <name>Dwight Schrutte</name>\n  <position>Assistant (to the) Regional Mana ...
[3] <person id="3">\n  <name>Jim Halpert</name>\n  <position>Sales Representative</position>\n  < ...
```

```
xml_child(xml_children(xml_doc), 1)
```

```
{xml_node}
<name>
```

Navigate sidewards

```
# navigate sidewards
persons[1]
```

```
{xml_nodeset (1)}
[1] <person id="1">\n  <name>Michael Scott</name>\n  <position>Regional Manager</position>\n  <lo ...
```

```
xml_siblings(persons[1])
```

```
{xml_nodeset (2)}
[1] <person id="2">\n  <name>Dwight Schrutte</name>\n  <position>Assistant (to the) Regional Mana ...
[2] <person id="3">\n  <name>Jim Halpert</name>\n  <position>Sales Representative</position>\n  < ...
```

Navigate upwards

```
# navigate upwards (all the levels)
xml_parents(persons)
```

```
{xml_nodeset (1)}
[1] <company_dundermifflin>\n  <person id="1">\n    <name>Michael Scott</name>\n    <position>Reg ...
```

```
# navigate upwards (one level)
xml_parent(persons)
```

```
{xml_nodeset (1)}
[1] <company_dundermifflin>\n  <person id="1">\n    <name>Michael Scott</name>\n    <position>Reg ...
```

Extract specific parts of the data

```
# find data via XPath
sales <- xml_find_all(xml_doc, xpath = ".//sales")
sales
```

```
{xml_nodeset (3)}
[1] <sales>\n  <units>10</units>\n  <product>paper A4</product>\n</sales>
[2] <sales>\n  <units>20</units>\n  <product>paper A4</product>\n</sales>
[3] <sales>\n  <units>5</units>\n  <product>paper A3</product>\n</sales>
```

```
xml_text(sales) # extract the data as text
```

```
[1] "10paper A4" "20paper A4" "5paper A3"
```

```
position <- xml_find_all(xml_doc, xpath = ".//position")
position
```

```
{xml_nodeset (3)}
[1] <position>Regional Manager</position>
[2] <position>Assistant (to the) Regional Manager</position>
[3] <position>Sales Representative</position>
```

Create a data frame with sales

Try to extract the sales data from the xml file at home. What is the difference between // and / in the xpath?

▶ Show the code

```
             name units  product
1            <NA>    NA     <NA>
2 Dwight Schrutte    10 paper A4
3     Jim Halpert    20 paper A4
4     Jim Halpert     5 paper A3
```

Deciphering JSON

JavaScript Object Notation (JSON)

- Text-based format used for representing structured and ased on JavaScript object syntax.
- Used for data exchange between a server and a web application

JSON syntax

- Key difference to XML: no tags, but *attribute-value pairs*.
- A substitute for XML (often encountered in similar usage domains).

XML:

```xml
<person>
  <firstName>John</firstName>
  <lastName>Smith</lastName>
  <age>25</age>
  <address>
    <streetAddress>21 2nd Street</streetAddress>
    <city>New York</city>
    <state>NY</state>
    <postalCode>10021</postalCode>
  </address>
  <phoneNumber>
    <type>home</type>
    <number>212 555-1234</number>
  </phoneNumber>
  <phoneNumber>
    <type>fax</type>
    <number>646 555-4567</number>
  </phoneNumber>
  <gender>
    <type>male</type>
  </gender>
</person>
```

JSON:

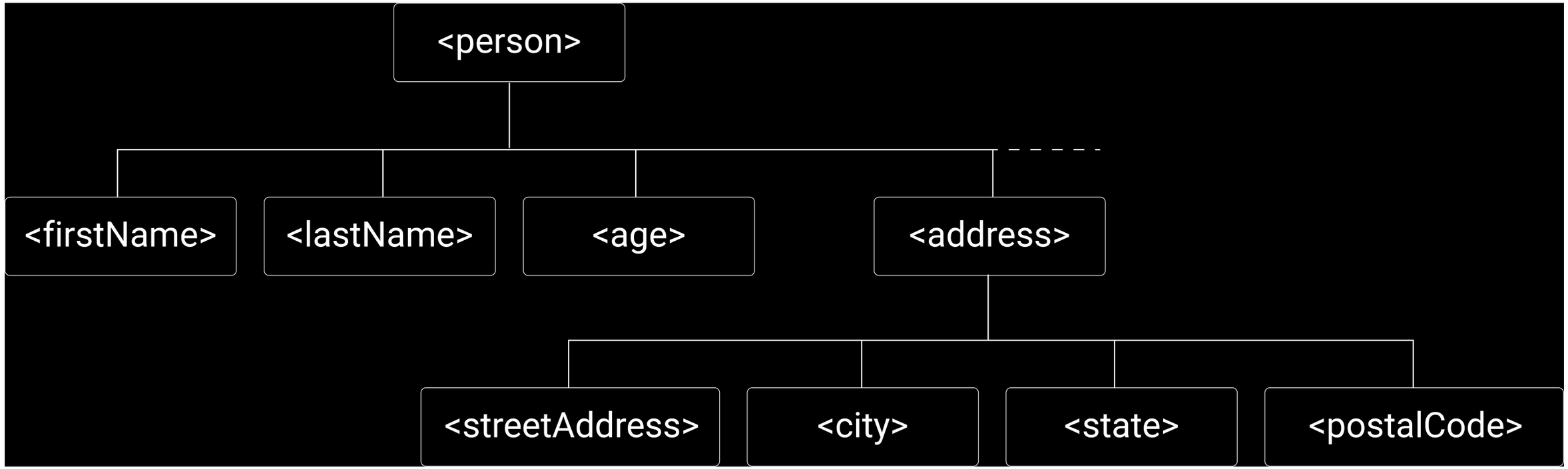
```json
{"firstName": "John",
  "lastName": "Smith",
  "age": 25,
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": "10021"
  },
  "phoneNumber": [
    {
      "type": "home",
      "number": "212 555-1234"
    },
    {
      "type": "fax",
      "number": "646 555-4567"
    }
  ],
  "gender": {
    "type": "male"
  }
}
```

XML:

```xml
<person>
  <firstName>John</firstName>
  <lastName>Smith</lastName>
</person>
```

JSON:

```json
{
  "firstName": "John",
  "lastName": "Smith",
}
```


<person>
<firstName>
<lastName>
<age>
<address>
<streetAddress>
<city>
<state>
<postalCode>

Parsing JSON

JSON in R

```r
# load packages
library(jsonlite)

# parse the JSON-document shown in the example above
json_doc <- fromJSON("data/person.json")

# look at the structure of the document
str(json_doc)
```

```
List of 6
 $ firstName  : chr "John"
 $ lastName   : chr "Smith"
 $ age        : int 25
 $ address    :List of 4
  ..$ streetAddress: chr "21 2nd Street"
  ..$ city         : chr "New York"
  ..$ state        : chr "NY"
  ..$ postalCode   : chr "10021"
 $ phoneNumber:'data.frame':    2 obs. of  2 variables:
  ..$ type  : chr [1:2] "home" "fax"
  ..$ number: chr [1:2] "212 555-1234" "646 555-4567"
 $ gender     :List of 1
  ..$ type: chr "male"
```

JSON in R

The nesting structure is represented as a *nested list*:

```
# navigate the nested lists, extract data
# extract the address part
json_doc$address
```

```
$streetAddress
[1] "21 2nd Street"

$city
[1] "New York"

$state
[1] "NY"

$postalCode
[1] "10021"
```

```
# extract the gender (type)
json_doc$gender$type
```

```
[1] "male"
```

Happy break!!!