TEXT AS DATA: WEEK 2

MATTHIAS HABER

15 SEPTEMBER 2021

GOALS FOR TODAY

GOALS

- Learn how to read data into R from different sources
- Learn easy ways to read in text data and to create a text corpus
- If we have time: Basic introduction to collecting data from the web

IMPORTING DATA FROM FLAT FILES

READ.CSV

`read.csv()` sets *sep=","* and *header=TRUE*

```r
potatoes <- read.csv("data/potatoes.csv")
head(potatoes)
```

```
##   area temp size storage method texture flavor moistness
## 1    1    1    1       1      1     2.9    3.2       3.0
## 2    1    1    1       1      2     2.3    2.5       2.6
## 3    1    1    1       1      3     2.5    2.8       2.8
## 4    1    1    1       1      4     2.1    2.9       2.4
## 5    1    1    1       1      5     1.9    2.8       2.2
## 6    1    1    1       2      1     1.8    3.0       1.7
```

SOME MORE IMPORTANT PARAMETERS

- *quote*: tell R whether there are any quoted values, quote="" means no quotes.
- *na.strings*: set the character that represents a missing value.
- *nrows*: how many rows to read of the file.
- *skip*: number of lines to skip before starting to read

REASONS TO USE `READR` INSTEAD

- `~`10x faster than base `read.table()` functions
- Long running jobs have a progress bar
- leave strings as is by default, and automatically parse common date/time formats.
- all functions work exactly the same way regardless of the current locale.

READR

- `read_csv()`: comma delimited files
- `read_csv2()`: semicolon separated files
- `read_tsv()`: tab delimited files
- `read_delim()`: files with any delimiter
- `read_fwf()`: fixed width files (`fwf_widths()` or `fwf_positions()`
- `read_table()`: files where columns are separated by white space
- `read_log()` reads Apache style log files

READ_CSV()

`read_csv()` uses the first line of the data for the column names

```
potatoes <- read_csv("data/potatoes.csv")
```

READ_CSV()

You can use `skip = n` to skip the first n lines; or use `comment = "#"` to drop all lines that start with (e.g.) #:

```
potatoes <- read_csv("data/potatoes.csv", skip = 2)
```

READ_CSV()

You can use `col_names = FALSE` to not treat the first row as headings, and instead label them sequentially from `X1` to `Xn:

```
read_csv("1,2,3\n4,5,6", col_names = FALSE)
```

```
## # A tibble: 2 × 3
##      X1    X2    X3
##   <dbl> <dbl> <dbl>
## 1     1     2     3
## 2     4     5     6
```

READ_CSV()

Alternatively you can supply your own column names with `col_names`:

```
read_csv("1,2,3\n4,5,6", col_names = c("x", "y", "z"))
```

```
## # A tibble: 2 × 3
##       x     y     z
##   <dbl> <dbl> <dbl>
## 1     1     2     3
## 2     4     5     6
```

READ_CSV()

Other arguments to `read_csv()`:

- `locale`: determine encoding and decimal mark.
- `na`, `quoted_na`: control which strings are treated as missing values
- `trim_ws`: trims whitespace before and after cells
- `n_max`: sets how many rows to read
- `guess_max`: sets how many rows to use when guessing the column type
- `progress`: determines whether a progress bar is shown.

READ_DELIM

`read_delim` is the main readr function and takes two mandatory arguments, *file* and *delim*.

```
properties <- c("area", "temp", "size", "storage",
                "method", "texture", "flavor",
                "moistness")
potatoes <- read_delim("data/potatoes.txt", delim = "\t",
                       col_names = properties)
```

`READR` PARSER

To figure out the type of each column `readr` reads the first 1000 rows and uses some heuristics to figure out the type of each column. You can try it out with `guess_parser()`, which returns `readr`'s best guess:

READR PARSER

```r
guess_parser("2010-10-01")
```

```
## [1] "date"
```

```r
guess_parser("15:01")
```

```
## [1] "time"
```

```r
guess_parser(c("TRUE", "FALSE"))
```

```
## [1] "logical"
```

READR( ) COL_TYPES

You can use `col_types` to specify which types the columns in your imported data frame should have. You can manually set the types with a string, where each character denotes the class of the column: `c`haracter, `d`ouble, `i`nteger and `l`ogical. `_` skips the column as a whole.

```
potatoes <- read_tsv("data/potatoes.txt",
                     col_types = "cccccccc",
                     col_names = properties)
```

IMPORT EXCEL FILES

READXL

The `readxl` package makes it easy to get data out of Excel and into R. `readxl` supports both .xls format and the modern xml-based .xlsx format.

You can use the `excel_sheets()` function to find out which sheets are available in the workbook.

```r
excel_sheets(path = "data/urbanpop.xlsx")
```

```
## [1] "1960-1966" "1967-1974" "1975-2011"
```

READXL

Use `read_excel()` to read in Excel files. You can pass a number (or string) to the `sheet` argument to import a specific sheet..

```
pop1 <- read_excel("data/urbanpop.xlsx", sheet = 1)
pop2 <- read_excel("data/urbanpop.xlsx", sheet = 2)
pop3 <- read_excel("data/urbanpop.xlsx", sheet = 3)

pop <- lapply(excel_sheets(path = "data/urbanpop.xlsx"),
              read_excel, path = "data/urbanpop.xlsx")
```

READXL

You can use `skip` to control which cells are read and `col_names` to set the column names.

```
pop <- read_excel(path = "data/urbanpop.xlsx", sheet=2,
                  skip=21, col_names=FALSE)
```

IMPORTING DATA FROM DATABASES

`DBI`

To import data from a database you first have to create a connection to it. You need different packages depending on the database you want to connect. `dbConnect()` creates a connection between your R session and a SQL database. The first argument has to be a `DBIdriver` object, that specifies how connections are made and how data is mapped between R and the database. If the SQL database is a remote database hosted on a server, you'll also have to specify the following arguments in dbConnect(): `dbname`, `host`, `port`, `user` and `password`.

ESTABLISH A CONNECTION

```r
host <- "courses.csrrinzqubik.us-east-1.rds.amazonaws.com"
con <- dbConnect(RMySQL::MySQL(),
                 dbname = "tweater",
                 host = host,
                 port = 3306,
                 user = "student",
                 password = "datacamp")
```

LIST THE DATABASE TABLES

After you've successfully connected to a remote database. you can use `dbListTables()` to see what tables the database contains:

```
tables <- dbListTables(con)
tables
```

```
## [1] "comments" "tweats"   "users"
```

IMPORT DATA FROM TABLES

You can use the `dbReadTable()` function to import data from the database tables.

```r
users <- dbReadTable(con, "users")
users
```

```
##   id        name     login
## 1  1 elisabeth  elismith
## 2  2        mike     mikey
## 3  3        thea   teatime
## 4  4      thomas tomatotom
## 5  5      oliver olivander
## 6  6        kate  katebenn
## 7  7      anjali    lianja
```

IMPORT DATA FROM TABLES

Again, you can use lapply to import all tables:

```r
tableNames <- dbListTables(con)
tables <- lapply(tableNames, dbReadTable, conn = con)
```

EXERCISE

The `tweats` table contains a column `user_id`, which refer to the users that have posted the tweat. The `comments` table contain both a `user_id` and a `tweat_id` column. Who posted the tweat on which somebody commented "awesome! thanks!" (comment 1012)? Be polite and disconnect from the database afterwards. You do this with the `dbDisconnect()` function.

EXERCISE SOLUTION

```
dbDisconnect(con)
```

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

IMPORTING DATA FROM THE WEB

IMPORT FILES DIRECTLY FROM THE WEB

You can use `read_csv` to directly import csv files from the web.

```
url <- paste0("https://raw.githubusercontent.com/",
"mhaber/AppliedDataScience/master/",
"slides/week2/data/potatoes.csv")
potatoes <- read_csv(url)
```

DOWNLOAD FILES

`read_excel()` does not yet support importing excel files directly from the web so you have to download the file first with `download.file()`:

```r
url <- paste0("https://github.com/",
"mhaber/AppliedDataScience/blob/master/",
"slides/week2/data/urbanpop.xlsx?raw=true")
download.file(url, "data/urbanpop.xlsx", mode = "wb")
urbanpop <- read_excel("data/urbanpop.xlsx")
```

HTTR

The `httr` package provides a convenient function `GET()` to download files. The result is a response object, that provides easy access to the content-type and the actual content. You can extract the content from the request using the `content()` function

```r
url <- "http://www.example.com/"
resp <- GET(url)
content <- content(resp, as = "raw")
head(content)
```

```
## [1] 3c 21 64 6f 63 74
```

JSON

- Javascript Object Notation
- Lightweight data storage
- Common format for data from application programming interfaces (APIs)
- Similar structure to XML but different syntax
- Data stored as
    - Numbers (double)
    - Strings (double quoted)
    - Boolean (*true* or *false*)
    - Array (ordered, comma separated enclosed in square brackets *[]*)
    - Object (unorderd, comma separated collection of key:value pairs in curley brackets *{}*)

EXAMPLE JSON FILE

```
[{
  "_id": {
    "$oid": "5968dd23fc13ae04d9000001"
  },
  "product_name": "sildenafil citrate",
  "supplier": "Wisozk Inc",
  "quantity": 261,
  "unit_cost": "$10.47"
}, {
  "_id": {
    "$oid": "5968dd23fc13ae04d9000002"
  },
  "product_name": "Mountain Juniperus ashei",
  "supplier": "Keebler-Hilpert",
  "quantity": 292,
  "unit_cost": "$8.74"
}, {
  "_id": {
    "$oid": "5968dd23fc13ae04d9000003"
  },
  "product_name": "Dextromathorphan HBr",
  "supplier": "Schmitt-Weissnat",
  "quantity": 211,
  "unit_cost": "$20.53"
}]
```

READING DATA FROM JSON (WITH `JSONLITE`)

```
url <- paste0("http://mysafeinfo.com/api/",
"data?list=englishmonarchs&format=json")
jsonData <- fromJSON(url)
str(jsonData)
```

```
## 'data.frame':    5 obs. of  5 variables:
##  $ ID     : int  1 2 3 4 5
##  $ Name   : chr  "Edward the Elder" "Athelstan" "Edmund" "Edred" ...
##  $ Country: chr  "United Kingdom" "United Kingdom" "United Kingdom" "U
##  $ House  : chr  "House of Wessex" "House of Wessex" "House of Wessex"
##  $ Reign  : chr  "899-925" "925-940" "940-946" "946-955" ...
```

READING TXT, PDF, HTML, DOCX, ...

READTEXT PACKAGE

A convenient way to read in document files into R is with the `readtext` package. It reads files containing text, along with any associated document-level metadata from forms such as .csv, .tab, .xml, and .json files.

`readtext` allows file masking, so you can specify patterns to load multiple texts at once. `readtext` returns a data.frame object with all text characters stored in a “text” field, as well as additional columns to store other document-level information.

INSTALLING READTEXT

To install `readtext`, you will need to use the `readtext` package, and then issue this command:

```r
# devtools package required to install readtext from Github
devtools::install_github("kbenoit/readtext")
```

IMPORTING TEXT WITH READTEXT

We can use the ‘glob’ operator ‘*’ to indicate that we want to load multiple files from a single directory.

```
library(readtext)
inaugural <- readtext("data/inaugural/*.txt")
sotu <- readtext("data/sotu/*.txt")
```

READING IN METADATA FROM TEXT

We can use the `docvarsfrom` argument to read in metadata encoded in the names of the files. For example, the inaugural addresses contain the year and the president's name:

```
inaugural_meta <- readtext("data/inaugural/*.txt",
                           docvarsfrom = "filenames", dvsep = "-",
                           docvarnames = c("Year", "President"))
head(inaugural_meta)
```

```
## readtext object consisting of 6 documents and 2 docvars.
## # Description: df [6 × 4]
##   doc_id               text                 Year President
##   <chr>                <chr>               <int> <chr>
## 1 1789-Washington.txt "\"Fellow-Cit\"..."  1789 Washington
## 2 1793-Washington.txt "\"Fellow cit\"..."  1793 Washington
## 3 1797-Adams.txt      "\"When it wa\"..."  1797 Adams
## 4 1801-Jefferson.txt  "\"Friends an\"..."  1801 Jefferson
## 5 1805-Jefferson.txt  "\"Proceeding\"..."  1805 Jefferson
## 6 1809-Madison.txt    "\"Unwilling \"..."  1809 Madison
```

READING TEXTS FROM STRUCTURED FILES

We can also read in texts and document variables from structured files such as csv, excel, or json by spefifying the name of the field containing the texts.

```
inaug <- readtext("data/inaugTexts.csv", textfield = "inaugSpeech")
head(inaug)
```

```
## readtext object consisting of 6 documents and 2 docvars.
## # Description: df [6 × 4]
##   doc_id           text                  Year President
##   <chr>            <chr>                <int> <chr>
## 1 inaugTexts.csv.1 "\"Fellow-Cit\"..."   1789 Washington
## 2 inaugTexts.csv.2 "\"Fellow cit\"..."   1793 Washington
## 3 inaugTexts.csv.3 "\"When it wa\"..."   1797 Adams
## 4 inaugTexts.csv.4 "\"Friends an\"..."   1801 Jefferson
## 5 inaugTexts.csv.5 "\"Proceeding\"..."   1805 Jefferson
## 6 inaugTexts.csv.6 "\"Unwilling \"..."   1809 Madison
```

READ IN PDF DATA

We can also use `readtext` to read in text from pdf files.

```r
inaugural_pdf <- readtext("data/*.pdf",
                          docvarsfrom = "filenames", dvsep = "-",
                          docvarnames = c("Year", "President"))
head(inaugural_pdf)
```

```
## readtext object consisting of 1 document and 2 docvars.
## # Description: df [1 × 4]
##   doc_id              text                  Year President
## * <chr>               <chr>                <int> <chr>
## 1 1789-Washington.pdf "\"Fellow-Cit\"..."  1789 Washington
```

CREATING A CORPUS OBJECT

Instead of loading our texts into a data.frame, we can transfer the text into a corpus object. A corpus is an extension of R list objects so we can use `[[]]` brackets to access single list elements, i.e. documents, within a corpus. We can also print the text of the first element of the corpus using the `as.character()` function

```
sotu_corpus <- corpus(readtext("data/sotu/*.txt"))
summary(sotu_corpus, n = 3)
```

```
## Corpus consisting of 229 documents, showing 3 documents:
##
##         Text Types Tokens Sentences
##   su1790.txt   591   1501        38
##  su1790b.txt   460   1167        24
##   su1791.txt   814   2473        58
```

```
#as.character(sotu_corpus[1])
```

ADDING ADDITIONAL DOCUMENT VARIABLES TO A CORPUS

We can easily add additional document variables to our data.frame, as long as the data frame containing them is of the same length as the texts:

```r
sotu_docvars <- read.csv("data/SOTU_metadata.csv", stringsAsFactors =
        FALSE)
sotu_docvars$date <- as.Date(sotu_docvars$Date, "%B %d, %Y")
sotu_docvars$delivery <- as.factor(sotu_docvars$delivery)
sotu_docvars$type <- as.factor(sotu_docvars$type)

sotu_corpus <- corpus(readtext("data/sotu/*.txt", encoding = "UTF-8-
        BOM"))
docvars(sotu_corpus) <- sotu_docvars
```

EXCERCISE

Take 5 minutes and try importing some text of your own.

PART II: BASIC INTRODUCTION TO COLLECTING DATA FROM THE WEB

BROWSING VS. SCRAPING

- **Browsing**
  - you click on something
  - browser sends request to server that hosts website
  - server returns resource (often an HTML document)
  - browser interprets HTML and renders it in a nice fashion

BROWSING VS. SCRAPING

- **Scraping with R**
  - you manually specify a resource
  - R sends request to server that hosts website
  - server returns resource
  - R parses HTML (i.e., interprets the structure), but does not render it in a nice fashion
  - it's up to you to tell R which parts of the structure to focus on and what content to extract

ONLINE TEXT DATA SOURCES

- **web pages** (e.g. http://example.com)
- **web formats** (XML, HTML, JSON, …)
- **web frameworks** (HTTP, URL, APIs, …)
- **social media** (Twitter, Facebook, LinkedIn, Snapchat, Tumbler, …)
- **data in the web** (speeches, laws, policy reports, news, …)
- **web data** (page views, page ranks, IP-addresses, …)

BEFORE SCRAPING, DO SOME GOOGLING!

- If the resource is well-known, someone else has probably built a tool which solves the problem for you.
- ropensci has a ton of R packages providing easy-to-use interfaces to open data.
- The Web Technologies and Services CRAN Task View is a great overview of various tools for working with data that lives on the web in R.

EXTRACTING DATA FROM HTML

For web scraping, we need to:

1. identify the elements of a website which contain our information of interest;
2. extract the information from these elements;

Both steps require some basic understanding of HTML and CSS. More advanced scraping techniques require an understanding of *XPath* and *regular expressions*.

WHAT'S HTML?

HyperText Markup Language

- markup language = plain text + markups
- standard for the construction of websites
- relevance for web scraping: web architecture is important because it determines where and how information is stored

INSPECT THE SOURCE CODE IN YOUR BROWSER

Firefox 1. right click on page 2. select “View Page Source”

Chrome 1. right click on page 2. select “View page source”

Safari 1. click on “Safari” 2. select “Preferences” 3. go to “Advanced” 4. check “Show Develop menu in menu bar” 5. click on “Develop” 6. select “Show Page Source.”

CSS?

Cascading Style Sheets

- style sheet language to give browsers information of how to render HTML documents
- CSS code can be stored within an HTML document or in an external CSS file
- selectors, i.e. patterns used to specify which elements to format in a certain way, can be used to address the elements we want to extract information from
- works via tag name (e.g.,`<h2>`,`<p>`) or element attributes `id` and `class`

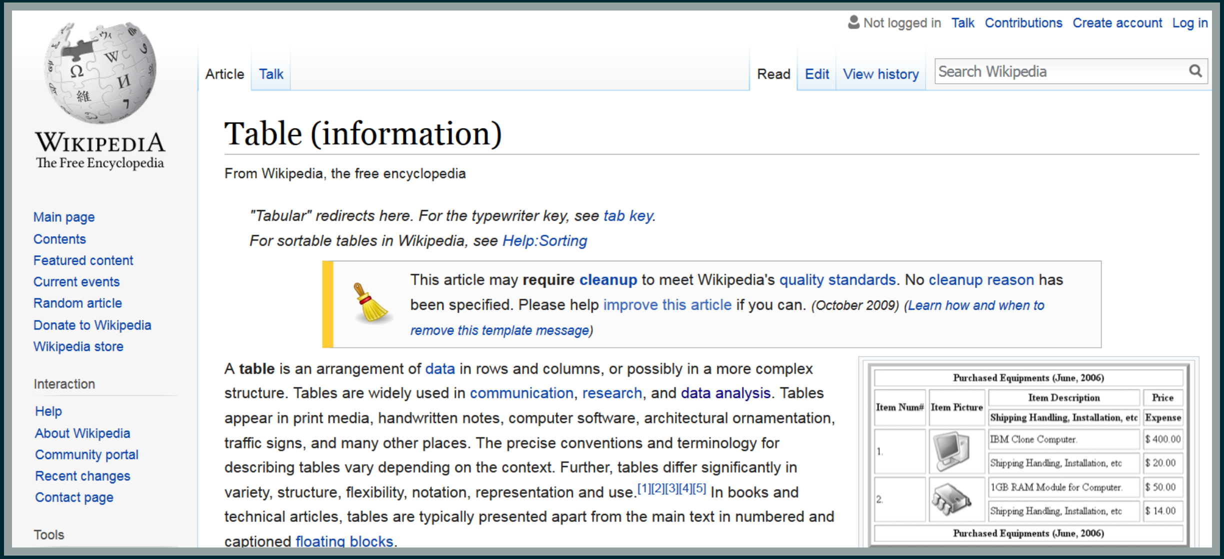
EXERCISE

1. Complete the first 5 levels on CSS Diner

XPATH

- XPath is a query language for selecting nodes from an XML-style document (including HTML)
- provides just another way of extracting data from static webpages
- you can also use XPath with R, it can be more powerful than CSS selectors

EXAMPLE



Not logged in Talk Contributions Create account Log in

WIKIPEDIA
The Free Encyclopedia

Article Talk | Read Edit View history | Search Wikipedia

Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikipedia store

Interaction

Help
About Wikipedia
Community portal
Recent changes
Contact page

Tools

Table (information)

From Wikipedia, the free encyclopedia

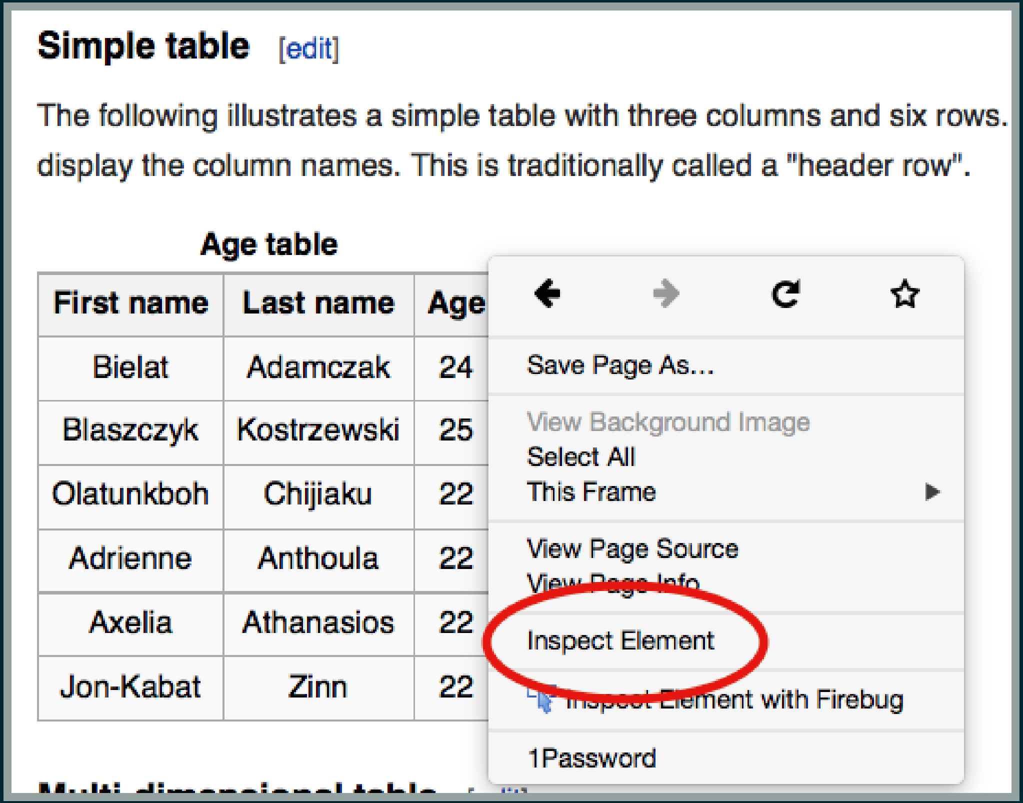
"Tabular" redirects here. For the typewriter key, see tab key.
For sortable tables in Wikipedia, see Help:Sorting

This article may **require cleanup** to meet Wikipedia's quality standards. No cleanup reason has been specified. Please help improve this article if you can. *(October 2009) (Learn how and when to remove this template message)*

A **table** is an arrangement of data in rows and columns, or possibly in a more complex structure. Tables are widely used in communication, research, and data analysis. Tables appear in print media, handwritten notes, computer software, architectural ornamentation, traffic signs, and many other places. The precise conventions and terminology for describing tables vary depending on the context. Further, tables differ significantly in variety, structure, flexibility, notation, representation and use.[1][2][3][4][5] In books and technical articles, tables are typically presented apart from the main text in numbered and captioned floating blocks.

Purchased Equipments (June, 2006)			
Item Num#	Item Picture	Item Description	Price
		Shipping Handling, Installation, etc	Expense
1.		IBM Clone Computer.	$ 400.00
		Shipping Handling, Installation, etc	$ 20.00
2.		1GB RAM Module for Computer.	$ 50.00
		Shipping Handling, Installation, etc	$ 14.00
Purchased Equipments (June, 2006)			

INSPECTING ELEMENTS



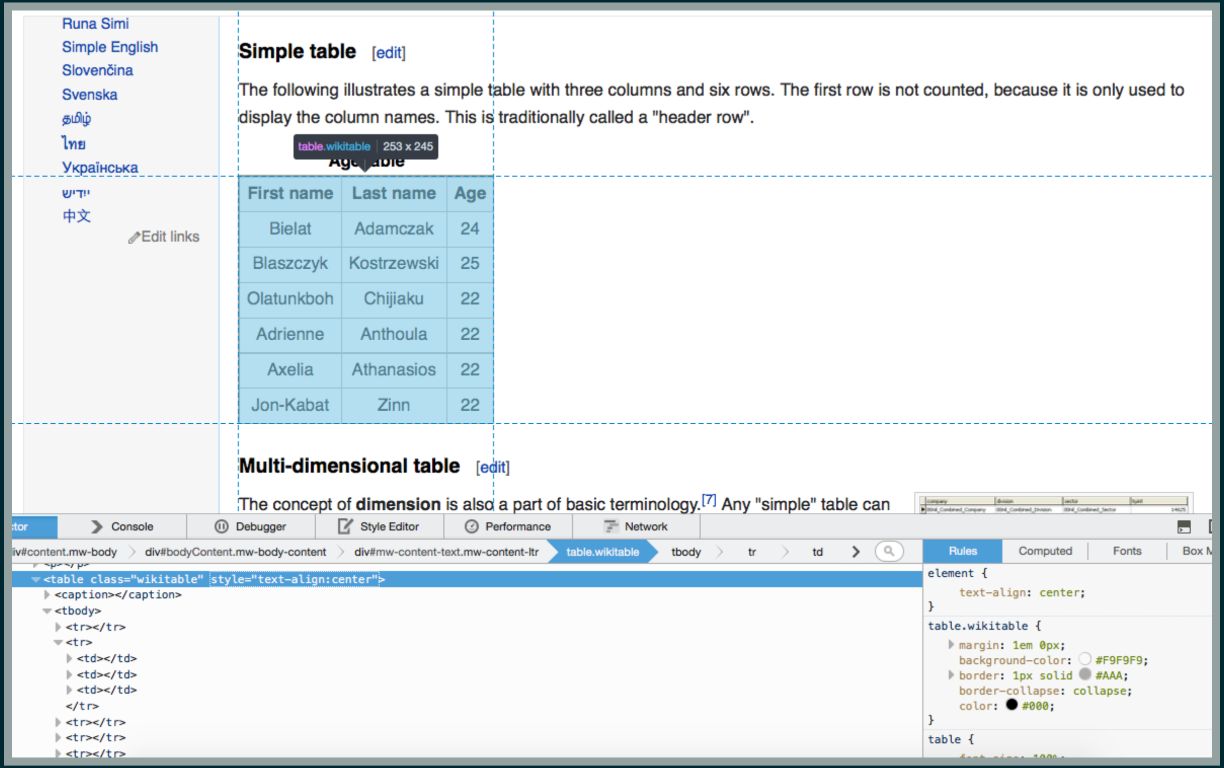
Simple table [edit]

The following illustrates a simple table with three columns and six rows. display the column names. This is traditionally called a "header row".

Age table

First name	Last name	Age
Bielat	Adamczak	24
Blaszczyk	Kostrzewski	25
Olatunkboh	Chijiaku	22
Adrienne	Anthoula	22
Axelia	Athanasios	22
Jon-Kabat	Zinn	22

HOVER TO FIND DESIRED ELEMENTS



Simple table [edit]

The following illustrates a simple table with three columns and six rows. The first row is not counted, because it is only used to display the column names. This is traditionally called a "header row".

First name	Last name	Age
Bielat	Adamczak	24
Blaszczyk	Kostrzewski	25
Olatunkboh	Chijiaku	22
Adrienne	Anthoula	22
Axelia	Athanasios	22
Jon-Kabat	Zinn	22

Multi-dimensional table [edit]

The concept of **dimension** is also a part of basic terminology.[7] Any "simple" table can

```
<table class="wikitable" style="text-align:center">
  <caption></caption>
  <tbody>
    <tr></tr>
    <tr>
      <td></td>
      <td></td>
      <td></td>
    </tr>
    <tr></tr>
    <tr></tr>
    <tr></tr>
```

```
element {
    text-align: center;
}
table.wikitable {
    margin: 1em 0px;
    background-color: #F9F9F9;
    border: 1px solid #AAA;
    border-collapse: collapse;
    color: #000;
}
table {
```

RVEST

rvest is a nice R package for web-scraping by (you guessed it) Hadley Wickham.

- see also: https://github.com/hadley/rvest
- convenient package to scrape information from web pages
- builds on other packages, such as xml2 and httr
- provides very intuitive functions to import and process webpages

BASIC WORKFLOW OF SCRAPING WITH RVEST

```r
# 1. specify URL
"http://en.wikipedia.org/wiki/Table_(information)" %>%

# 2. download static HTML behind the URL and parse it into an XML file
read_html() %>%

# 3. extract specific nodes with CSS (or XPath)
html_node(".wikitable") %>%

# 4. extract content from nodes
html_table(fill = TRUE)
```

```
## # A tibble: 9 × 3
##   `First name` `Last name`   Age
##   <chr>        <chr>       <int>
## 1 Tinu         Elejogun       14
## 2 Javier       Zapata         28
## 3 Lily         McGarrett      18
## 4 Olatunkbo    Chijiaku       22
## 5 Adrienne     Anthoula       22
## 6 Axelia       Athanasios     22
## 7 Jon-Kabat    Zinn           22
## 8 Thabang      Mosoa          15
## 9 Kgaogelo     Mosoa          11
```

SELECTORGADGET

- Selectorgadget is a Chrome browser extension for quickly extracting desired parts of an HTML page.
- to learn about it, use vignette(“selectorgadget”)
- to install it, visit http://selectorgadget.com/

SELECTORGADGET

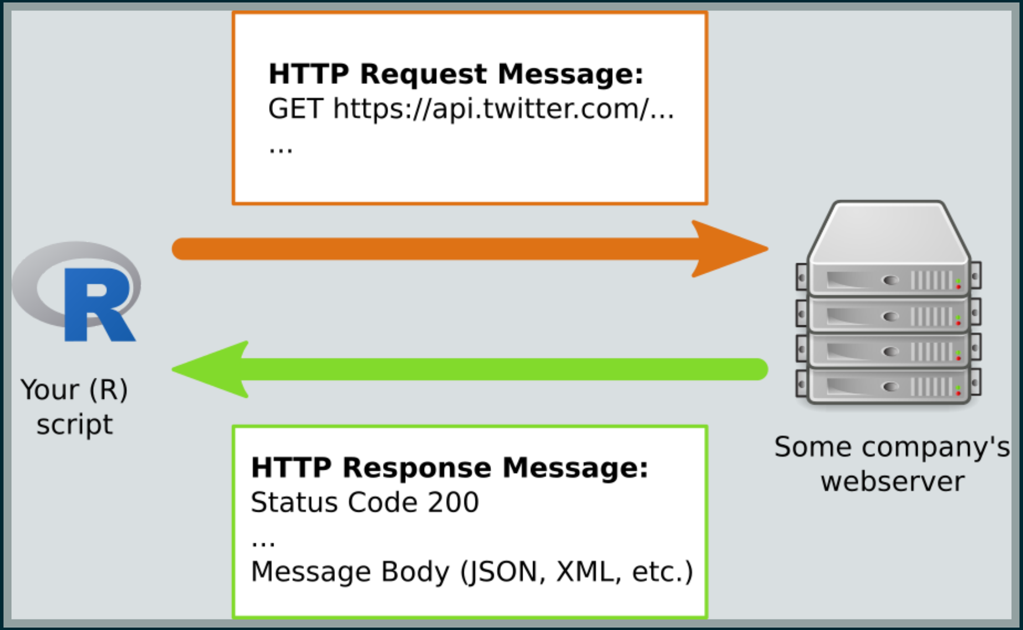
```r
url <- "http://spiegel.de/schlagzeilen"
css <- ".schlagzeilen-headline"
url_parsed <- read_html(url)
html_nodes(url_parsed, css = css) %>% html_text
```

APIS

- API stands for *Application Programming Interface*
- defined interface for communication between software components
- *Web* API: provides an interface to structured data from a web service
- APIs should be used whenever you need to automatically collect mass data from the web
- it should definitely be preferred over web scraping

FUNCTIONALITY OF APIS

Web APIs usually employ a **client-server model**. The client – that is you. The server provides the *API endpoints* as URLs.


HTTP Request Message:
GET https://api.twitter.com/...
...
Your (R) script
Some company's webserver
HTTP Response Message:
Status Code 200
...
Message Body (JSON, XML, etc.)

FUNCTIONALITY OF APIS

Communication is done with request and response messages over *Hypertext transfer protocol (HTTP)*.

Each HTTP message contains a *header* (message meta data) and a *message body* (the actual content). The three-digit HTTP status code plays an important role:

- 2xx: Success
- 4xx: Client error (incl. the popular *404: Not found* or *403: Forbidden*)
- 5xx: Server error

The message body contains the requested data in a specific format, often JSON or XML.

EXAMPLES OF POPULAR APIS

Social media:

- Twitter
- Facebook Graph API (restricted to own account and public pages)
- YouTube (Google)
- LinkedIn

For more, see programmableweb.com.

API WRAPPER PACKAGES

- Working with a web API involves:
  - constructing request messages
  - parsing result messages
  - handling errors
- For popular web services there are already “API wrapper packages” in R:
  - implement communication with the server
  - provide direct access to the data via R functions
  - examples: *rtweet*, *ggmap* (geocoding via Google Maps), *wikipediR*, *genderizeR*

TWITTER

Twitter has two types of APIs

- REST APIs –> reading/writing/following/etc.
- Streaming APIs –> low latency access to 1% of global stream - public, user and site streams
- authentication via OAuth
- documentation at https://dev.twitter.com/overview/documentation

ACCESSING THE TWITTER APIS

- To access the REST and streaming APIs, all you need is a Twitter account and you can be up in running in minutes!
- Simply send a request to Twitter's API (with a function like `search_tweets()`) during an interactive session of R, authorize the embedded `rstats2twitter` app (approve the browser popup), and your token will be created and saved/stored (for future sessions) for you!
- You can obtain a developer account to get more stability and permissions.

TWITTER API SUBSCRIPTIONS

Twitter provides three subscription levels:

- Standard (free)
  - search historical data for up to 7 days
  - get sampled live tweets
- Premium ($150 to $2500 / month)
  - search historical data for up to 30 days
  - get full historical data per user
- Enterprise (special contract with Twitter)
  - full live tweets

The *rate limiting* also differs per subscription level (number of requests per month).

USE TWITTER IN R

```r
library(rtweet)

## search for 1000 tweets using the #niewiedercdu hashtag
tweets <- search_tweets(
  "#iacaucus", n = 1000, include_rts = FALSE
)
```

Find out what else you can do with the `rtweet` package:
https://github.com/mkearney/rtweet

EXERCISE

1. We want to get data from abgeordnetenwatch.de. The site has an API, so technically there is no need to scrape anything. Load the package `jsonlite` into library.
2. Go to https://www.abgeordnetenwatch.de/api and figure our the syntax of their APIs.
3. Use the `fromJSON` function in `jsonlite` to load a JSON file via the API into R. Save the input as `aw_data`.

EXERCISE SOLUTION

WRAPPING UP

QUESTIONS?

OUTLOOK FOR OUR NEXT SESSION

- Next week we will learn how to clean and transform text
- We will meet online on MS Teams for the next eight sessions

THAT'S IT FOR TODAY

Thanks for your attention!