# Data Management With R: Data Import

Matthias Haber

02 October 2017

# Prerequisites

# Packages

```
# install.packages("tidyverse")
library(tidyverse)
```

```
library(readxl)
library(DBI)
library(RMySQL) # install.packages("RMySQL")
library(httr)
library(jsonlite)
library(haven)
```

# Data

Today we'll work with a number of datasets. They are located in *slides/week5/data* and on moodle.

- *potatoes*: impact of storage and cooking on potatoes' flavor
- *urbanpop*: worldwide urban population metrics over time
- *sales*: data on the age, gender, income, and purchase level
- *sugar*: data on yearly import and export numbers of sugar
- *personality*: data on Big Five personality traits for 434 persons

# Importing data from flat files

# `read.table()`

- Main function for reading data into R
- Flexible and robust but requires more parameters
- Reads the data into RAM - big data can cause problems
- Important parameters *file*, *header*, *sep*, *row.names*, *nrows*
- Related: `read.csv()`, `read.csv2()`

```
potatoes <- read.table(file = "data/potatoes.csv",
                       sep = ",", header = TRUE)
```

# read.csv

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

```
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 (use the `fread()` from `data.table` if you want even more speed)
- 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")
```

```
## Parsed with column specification:
## cols(
##   area = col_integer(),
##   temp = col_integer(),
##   size = col_integer(),
##   storage = col_integer(),
##   method = col_integer(),
##   texture = col_double(),
##   flavor = col_double(),
##   moistness = col_double()
## )
```

# 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)
```

```
## Parsed with column specification:
## cols(
##   `1` = col_integer(),
##   `1_1` = col_integer(),
##   `1_2` = col_integer(),
##   `1_3` = col_integer(),
##   `2` = col_integer(),
##   `2.3` = col_double(),
##   `2.5` = col_double(),
##   `2.6` = col_double()
## )
```

# 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
##   <int> <int> <int>
## 1     1     2     3
## 2     4     5     6
```

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"))
```

# `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

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

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

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

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

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

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

```
guess_parser(c("1", "5", "9"))
```

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

# `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: character, double, `integer` and `logical`. `_` skips the column as a whole.

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

# Exercises

1. What is wrong with each of the following inline CSV files.

```
read_csv("a,b\n1,2,3\n4,5,6")
read_csv("a,b,c\n1,2\n1,2,3,4")
read_csv("a;b\n1;3")
```

# Exercises solutions

```r
# Only two columns are specified in the
# header "a" and "b"
read_csv("a,b\n1,2,3\n4,5,6")
# There are only two values in row 1,
# so column c is set to NA
read_csv("a,b,c\n1,2\n1,2,3,4")
# The values are separated by ";"
# not ",". Use read_csv2 instead
read_csv("a;b\n1;3")
```

# 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.

```
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")
```

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

```
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.

```
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:

```
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

The user with `user_id` 5: Oliver.

```
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/HertieDataScience/master/",
"slides/week5/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()`:

```
url <- paste0("https://github.com/",
"mhaber/HertieDataScience/blob/master/",
"slides/week5/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

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

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

# JSON

## 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 $\{\}$)

`http://en.wikipedia.org/wiki/JSON`

# 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':    57 obs. of  4 variables:
##  $ nm : chr  "Edmund Ironside" "Cnut" "Harold I Harefoot
##  $ cty: chr  "United Kingdom" "United Kingdom" "United K
##  $ hse: chr  "House of Wessex" "House of Denmark" "House
##  $ yrs: chr  "1016" "1016-1035" "1035-1040" "1040-1042"
```

# Writing data frames to JSON

You can use `toJSON()` to convert R data to a JSON object. JSONs can come in mini or pretty format with indentation, whitespace and new lines.

```
# Mini
{"a":1,"b":2,"c":{"x":5,"y":6}}
# Pretty
{
  "a": 1,
  "b": 2,
  "c": {
    "x": 5,
    "y": 6
  }
}
```

# Convert back to JSON

```
myJson <- toJSON(iris)
iris2 <- fromJSON(myJson)
head(iris2)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Spec
## 1          5.1         3.5          1.4         0.2  set
## 2          4.9         3.0          1.4         0.2  set
## 3          4.7         3.2          1.3         0.2  set
## 4          4.6         3.1          1.5         0.2  set
## 5          5.0         3.6          1.4         0.2  set
## 6          5.4         3.9          1.7         0.4  set
```

# Exercise

1. Create an object called `json1` that contains a vector with the numbers 1 up to 6, in ascending order.
2. Create an object `json2` that contains a named list with two elements: `a`, containing the numbers 1, 2 and 3 and `b`, containing the numbers 4, 5 and 6.
3. Call `fromJSON()` on both `json1` and `json2`

# Exercise solution

```
json1 <- '[1, 2, 3, 4, 5, 6]'
json2 <- '{"a": [1, 2, 3], "b": [4, 5, 6]}'
fromJSON(json1)
```

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

```
fromJSON(json2)
```

```
## $a
## [1] 1 2 3
##
## $b
## [1] 4 5 6
```

# Importing data from other statistical software

# Importing data from other statistical software

We can use `haven()` to read data from other statistical software packages such as SAS, STATA and SPSS.

- *SAS*: `read_sas()`
- *STATA*: `read_dta()`
- *SPSS*: `read_sav()` or `read_por()`, depending on the file type.

All of these functions take the path to your local (or online) file.

# read_sas()

```
sales <- read_sas("data/sales.sas7bdat")
str(sales)
```

```
## Classes 'tbl_df', 'tbl' and 'data.frame':    431 obs. of
##  $ purchase: num  0 0 1 1 0 0 0 0 0 0 ...
##  $ age     : num  41 47 41 39 32 32 33 45 43 40 ...
##  $ gender  : chr  "Female" "Female" "Female" "Female" ...
##  $ income  : chr  "Low" "Low" "Low" "Low" ...
##  - attr(*, "label")= chr "SALES"
```

# read_dta()

With Stata data files, it can also happen that some of the variables you import have the labelled class. This is done to keep all the labelling information that was originally present in the `.dta`.It's advised to change these variables to factors or other standard R classes.

```
sugar <- read_dta("data/sugar.dta")
sugar$Date <- as.Date(as_factor(sugar$Date))
```

# read_sav()

```
personality <- read_sav("data/personality.sav")
```

# Importing data from other sources

# Images

- jpeg - http://cran.r-project.org/web/packages/jpeg/index.html
- readbitmap - http://cran.r-project.org/web/packages/readbitmap/index.html
- png - http://cran.r-project.org/web/packages/png/index.html
- EBImage (Bioconductor) - http://www.bioconductor.org/packages/2.13/bioc/html/EBImage.html

# Geospatial data

- rgdal - http://cran.r-project.org/web/packages/rgdal/index.html
- rgeos - http://cran.r-project.org/web/packages/rgeos/index.html
- raster - http://cran.r-project.org/web/packages/raster/index.html

# Music data

- tuneR - http://cran.r-project.org/web/packages/tuneR/
- seewave - http://rug.mnhn.fr/seewave/

# Homework Exercises

# Homework Exercises

There will be no homework assignments this week.

**That's it for today. Questions?**