# Data I/O?

The R Bootcamp
Twitter: @therbootcamp

September 2017

# Importing and Exporting Data

In this introduction you will learn...

1. How to import data data from **delimeter separated files** (e.g., .csv)?
2. How to import data data from **proprietory file formats** (e.g., .sav)?
3. How to save/export data to various formats, including **R's own files types**?
4. About a new data format called **tibble**.
5. How to use **file connections** to read data in its rawest possible way?

# readr

`readr` is a `tidyverse` package that provides convenient functions to **read in** *flat* (non-nested) data files into data frames (`tibbles` to be precise):

```
# Functions for import
read_csv()  # for comma-delimited files
read_csv2() # for semicolon-delimeted file
read_delim  # for any delimited files

# Functions for parsing
parse_logical() # parse logical vectors
parse_integer() # parse integers
parse_double(), parse_number()  # parse num
parse_character() # parse character aka str
parse_date()    # parse date -> lubridate p
```

# An example

Assume we have a *flat* data set with variables `id`, `var_1`, and `var_2` and cases as rows. Such data can be conveniently read in using `read_csv()`. Moreover, `read_csv()` will automatically identify **(a)** columns and rows, **(b)** column names, **(c)** the type of the columns and finally return a `tibble` (more on that later).

```
# define simple data set
data <-
   "id, var_1, var_2\n
  DCDL,  .287,  .048\n
  FEFK,  .894,  .383\n
  ZEWE, 1.374,  .623\n
  OJEE,  .631,  .826"
```

```
# read in data
require(readr)
read_csv(data)
```

```
## Loading required package: readr
```

```
## # A tibble: 4 x 3
##      id var_1 var_2
##   <chr> <dbl> <dbl>
## 1  DCDL 0.287 0.048
## 2  FEFK 0.894 0.383
## 3  ZEWE 1.374 0.623
## 4  OJEE 0.631 0.826
```

# Parsing

Behind the magic of `readr` functions such as `read_csv` are a set of really flexible parsing functions `parse_*()` that **transform the input into the apprpriate format and type**.

## Examples

```
# parsing a logical
parse_logical(c("TRUE","FALSE","NA"))
```

```
## [1]  TRUE FALSE    NA
```

```
# parsing an integer + errors
parse_integer(c("123", "345", "abc", "123.4
```

```
## [1] 123 345  NA  NA
## attr(,"problems")
## # A tibble: 2 x 4
##     row   col               expected actual
##   <int> <int>                  <chr>  <chr>
## 1     3    NA             an integer    abc
## 2     4    NA no trailing characters    .45
```

```
# parse
parse_character("hellow","world")
```

```
## [1] "hellow"
```

```
# parsing a numeric
parse_number(c("1.23","$123.209"))
```

```
## [1]   1.23 123.21
```

# Locale

How the text input is parsed can largely be controlled using **arguments** of the respective functions. Otherwise the parsing behavior will be **controlled by settings `locale()`**. To change, e.g., the `decimal_mark` use `locale("en", decimal_mark = ",")`.

```
str(locale())
```

```
## List of 7
##  $ date_names   :List of 5
##   ..$ mon   : chr [1:12] "January" "February" "March" "April" ...
##   ..$ mon_ab: chr [1:12] "Jan" "Feb" "Mar" "Apr" ...
##   ..$ day   : chr [1:7] "Sunday" "Monday" "Tuesday" "Wednesday" ...
##   ..$ day_ab: chr [1:7] "Sun" "Mon" "Tue" "Wed" ...
##   ..$ am_pm : chr [1:2] "AM" "PM"
##   ..- attr(*, "class")= chr "date_names"
##  $ date_format  : chr "%AD"
##  $ time_format  : chr "%AT"
##  $ decimal_mark : chr "."
##  $ grouping_mark: chr ","
##  $ tz           : chr "UTC"
##  $ encoding     : chr "UTF-8"
##  - attr(*, "class")= chr "locale"
```

# Error handling

When reading a file using, e.g., `read_csv`, with no specific arguments provided, `readr` **infers the type of each column** using a heuristic process base on the **first 1,000 rows** (see `guess_parser`). However, that may not always work. In that case consider ...

```
# to inspect the problems
problems()

# to set the types
read_csv(..., col_types = cols(...))

# read in as character and convert later
read_csv(..., col_types = cols(.default = col_character()))
type_convert()

# use more basic read-in functions (see later)
```

# Other read in options

The `tidyverse` together with other package offers a **variety of built-in, automated read functions** for almost any data format. For an overview **see rio**.

## `readr`


```
# read fixed width files (can be fast)
read_fwf()

# read Apache style log files
read_log()
```

## `haven`


```
# read SAS's .sas7bat and sas7bcat files
read_sas()

# read SPSS's .sav files
read_sav()

# etc
```

## `readxl`


```
# read Excel's .xls and xlsx files
read_excel()
```

## Other packages

```
# from package R.matlab: read .mat
readMat()

# from package XML: read and wrangle .xml a
xmlParseParse()

# from package jsonlite: read .json files
read_json()
```

# Writing data

Most read-in functions have **complementary write** functions to save the specific data format to disk. They usually require as **arguments** the **data frame** and a **file path** on the disk.

## readr


```
# read fixed width files (can be fast)
write_csv(my_data_frame,"my_data.csv")
write_delim(...)
```

## haven


```
# read SAS's .sas7bat and sas7bcat files
write_sas()

# read SPSS's .sav files
write_sav()

# etc
```

## Other packages

```
# from package R.matlab: read .mat
writeMat()

# from package XML: read and wrangle .xml a
saveXML()

# from package jsonlite: read .json files
write_json()
```

# R's data formats

R also has a couple of own data files that can be used to store and retrieve data. The benefits of these data files are that you can store **data as R objects**, as well as substantial **compression** (depending on repetitions/patterns in the data up to about 1% of the original size).

## `.RData`

- Bundles several R objects.
- Loads objects directly to workspace.
- Relatively slow.

```
# save data as .RData
save(object_1, object_2,..., file = "my_dat

# load data from .RData
load("my_data.RData")
```

## `.RDS`

- Stores individual R objects.
- Import can is newly assigned.
- Relatively fast.

```
# save data as .RDS
saveRDS(my_data, file = "my_data.RData")

# load data from .RDS
my_data <- readRDS("my_data.RData")
```

# File connections

Under the hood, practically all reading and writing functions rely on R's **file connection** architecture, which is very similar to most other programming languages. To the experienced programmer file connections represent a flexible option to access and manipulate files on **any accessible location** (harddrive, server, www).

## Benefits of file connections

- Define access mode (read, write, append)
- Read/write as binary or raw
- Handle encodings directly
- Read and write compressed files
- Access to www and servers

## Functions

```
# access a file
file("my_data.csv",'r')

# access url
url(...)

# access compressed files
gzfile(...)

# to handle connections
readLines() # to read contect
close() # close connection
```

# tibbles

The **output** from most `tidyverse` read functions such as `read_csv` and the preferred data format for many (but not all) analyses is a `tibble`. `tibbles` are a **modern, leaner version of data.frames**.

## tibbles ...

- never change the input's type
- never add row names
- never change column names
- look better in `print`
- are accessed more consistently

## Functions

```
# create tibble
my_data <- tibble(id, var_1, var_2)

# convert to and from data.frame
as_tibble(my_data_frame)
as.data.frame(my_tibble)
```

```
## # A tibble: 4 x 3
##      id var_1 var_2
##   <chr> <dbl> <dbl>
## 1  DCDL 0.287 0.048
## 2  FEFK 0.894 0.383
## 3  ZEWE 1.374 0.623
## 4  OJEE 0.631 0.826
```

# Practical

**Link to practical**