Importing Data

Daniel Anderson

Agenda

- Discuss reading in data
    - The *rio* package, and when you may want some additional flexibility
    - Other packages for reading in data
- Lab
    - Use R Markdown
    - Read in data from two different sources
    - Conduct some basic manipulations

rio

- Super nice package - most of the time, it just works, regardless of the source file type.
- (If this isn't astounding to you, you obviously haven't struggled for hours to read in data properly)

Example: these all work! Try it out and verify for yourself!

```r
library(rio)
exam1 <- import("./data/exam1.csv")
eclsk <- import("./data/ecls-k_samp.sav")
fatality <- import("./data/Fatality.txt")
```

Read in text files directly from the web!

```r
reads <- import("https://data.jacksonms.gov/api/views/97iy-g8hk/rows.csv")
head(reads)
```

```
##                Test Year Test Type Test Site Student ID Pre-Test Score
## 1 06/01/2016 12:00:00 AM  YEAR END    VIRDEN   Virden 1             43
## 2 06/01/2016 12:00:00 AM  YEAR END    VIRDEN   Virden 2             46
## 3 06/01/2016 12:00:00 AM  YEAR END    VIRDEN   Virden 3             39
## 4 06/01/2016 12:00:00 AM  YEAR END    VIRDEN   Virden 4             35
## 5 06/01/2016 12:00:00 AM  YEAR END    VIRDEN   Virden 5             46
## 6 06/01/2016 12:00:00 AM  YEAR END    VIRDEN   Virden 6             35
##   Pre-Test (%) Post-Test Score Post-Test (%) Percentage Change
## 1          29%              92           61%               32%
## 2          31%             104           69%               38%
## 3          26%              75           50%               24%
## 4          23%             115           77%               54%
## 5          31%              85           57%               26%
## 6          23%              91           61%               38%
##   Unit 1 Score Unit 1 (%) Unit 2 Score Unit 2 (%) Unit 3 Score Unit 3 (%)
## 1            3        12%            4        16%            6        24%
## 2            5        20%            5        20%            6        24%
## 3            4        16%            4        16%            6        24%
```

You can export just as easily!

Try it out!

```r
library(janitor)
reads <- clean_names(reads)
export(reads, "project_reads.sav")
export(reads, "project_reads.txt")
export(reads, "project_reads.dta")
```

Note. The `clean_names` function was neccessary because spaces aren't valid for SPSS or Stata variable names. If you don't run `clean_names()` first the stata export will fail, while Running `clean_names` first fixes the issue.

Export any data frame in any format

```r
library(tidyverse)
mtcars %>%
  group_by(cyl) %>%
  summarize(mean_mpg_by_cyl = mean(mpg))
```

```
## # A tibble: 3 x 2
##     cyl mean_mpg_by_cyl
##   <dbl>           <dbl>
## 1     4        26.66364
## 2     6        19.74286
## 3     8        15.10000
```

```r
mtcars %>%
  group_by(cyl) %>%
  summarize(mean_mpg_by_cyl = mean(mpg)) %>%
  export("mpg_mean_by_cyl.sav")
```

`convert()`

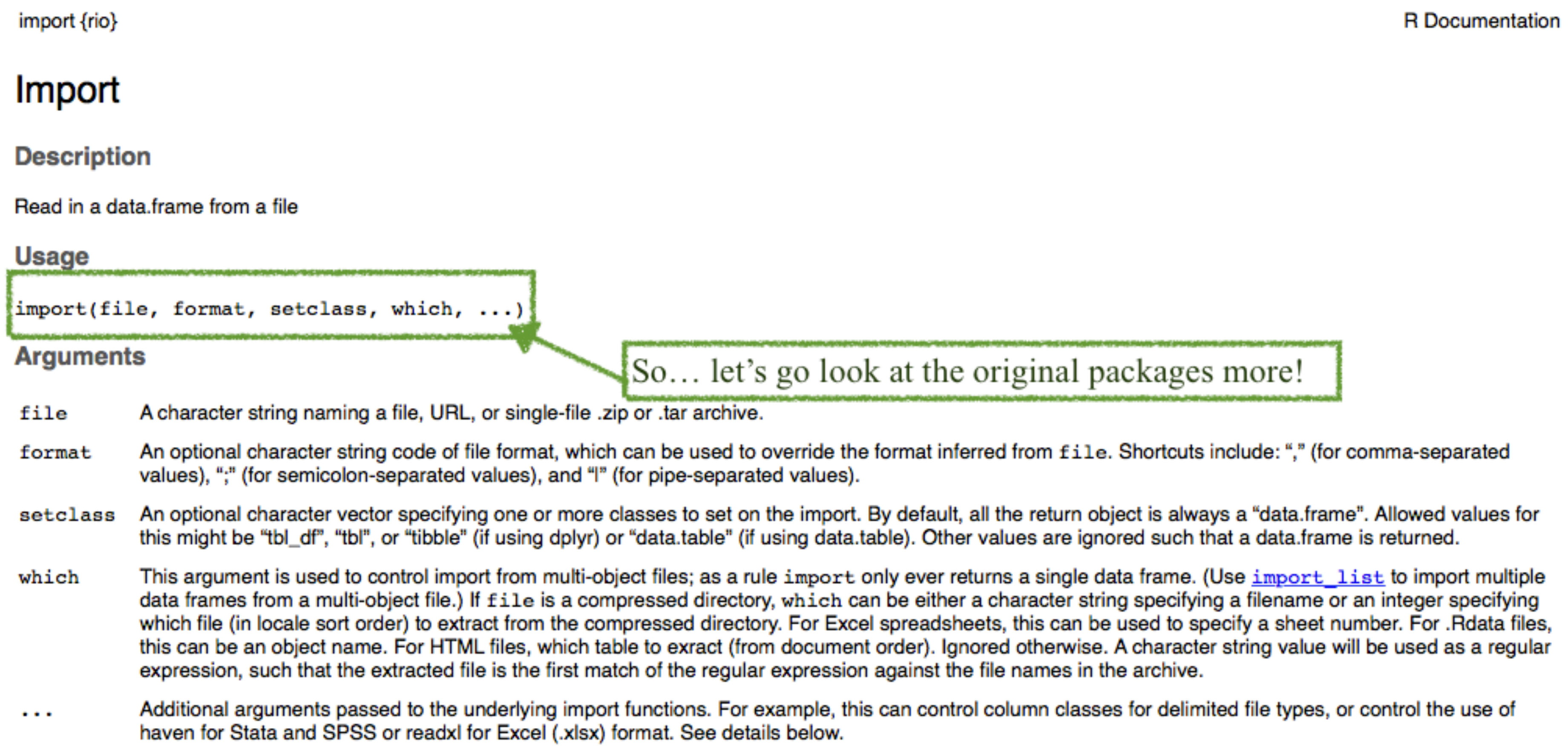
- Another really useful feature is `convert()`, which just takes a file of one type and converts it to another.
- Say your advisor uses SPSS, but her/his colleague uses Stata. They might use some proprietary (and expensive) software like SAS/CONNECT. Instead, just run this one line of code and voila!

```
convert("./data/ecls-k_samp.sav", "./data/ecls-k_samp.dta")
```

How is this all working?

Looking at the import documentation

```
?import
```



import {rio} R Documentation

Import

Description

Read in a data.frame from a file

Usage

```
import(file, format, setclass, which, ...)
```

So… let's go look at the original packages more!

Arguments

`file`	A character string naming a file, URL, or single-file .zip or .tar archive.
`format`	An optional character string code of file format, which can be used to override the format inferred from `file`. Shortcuts include: "," (for comma-separated values), ";" (for semicolon-separated values), and "|" (for pipe-separated values).
`setclass`	An optional character vector specifying one or more classes to set on the import. By default, all the return object is always a "data.frame". Allowed values for this might be "tbl_df", "tbl", or "tibble" (if using dplyr) or "data.table" (if using data.table). Other values are ignored such that a data.frame is returned.
`which`	This argument is used to control import from multi-object files; as a rule `import` only ever returns a single data frame. (Use `import_list` to import multiple data frames from a multi-object file.) If `file` is a compressed directory, `which` can be either a character string specifying a filename or an integer specifying which file (in locale sort order) to extract from the compressed directory. For Excel spreadsheets, this can be used to specify a sheet number. For .Rdata files, this can be an object name. For HTML files, which table to exract (from document order). Ignored otherwise. A character string value will be used as a regular expression, such that the extracted file is the first match of the regular expression against the file names in the archive.
`...`	Additional arguments passed to the underlying import functions. For example, this can control column classes for delimited file types, or control the use of haven for Stata and SPSS or readxl for Excel (.xlsx) format. See details below.

Tidyverse packages

- `readr`: Designed for quick and efficent reading/writing of plain text files (csv, tsv, txt, etc)
  - not used by *rio*, but if you're having any trouble with csv's, this is the method I'd recommend moving toward.
- `haven`: Designed to read/write files from SPSS, SAS, and Stata files
  - Used by *rio* but with some differences in how the data are actually read in.

readr

- Primary function is `read_csv`
- Used equivalently to `rio::import`, but only works for csv files
- Note the messages produced, below

```r
library(tidyverse)
exam1 <- read_csv("./data/exam1.csv")
```

```
## Parsed with column specification:
## cols(
##   .default = col_integer(),
##   stu_name = col_character(),
##   gender = col_character()
## )
```

```
## See spec(...) for full column specifications.
```

Use it to read in text from a string

```
tmp <- read_csv("Jane, Mary, Bob
                 1, 5, 8
                 4, 2, 6
                 3, 5, 1")
tmp
```

```
## # A tibble: 3 x 3
##    Jane  Mary   Bob
##   <int> <int> <int>
## 1     1     5     8
## 2     4     2     6
## 3     3     5     1
```

Skipping lines

(also works with `rio::import`)

If there's notes or blank lines to begin with, you can skip over them.

```
read_csv("Here's a line of garbage
Here's another with some note that you can see in excel but not here
That's a silly way to store data
Next line has the actual data.
Jane, Mary, Bob
1, 5, 8
4, 2, 6
3, 5, 1",
  skip = 4)
```

```
## # A tibble: 3 x 3
##    Jane  Mary   Bob
##   <int> <int> <int>
## 1     1     5     8
## 2     4     2     6
## 3     3     5     1
```

Column names

If there are no column names, they can be supplied with *col_names*.

```
read_csv("1, 5, 8
          4, 2, 6
          3, 5, 1",
    col_names = c("Jane", "Mary", "Bob"))
```

```
## # A tibble: 3 x 3
##     Jane  Mary   Bob
##    <int> <int> <int>
## 1      1     5     8
## 2      4     2     6
## 3      3     5     1
```

Missing values

(also works with `rio::import`) Specify your own missing values.

```
read_csv("1, 5, 8
          4, 2, 999
          999, 5, 1",
    col_names = c("Jane", "Mary", "Bob"),
    na = "999")
```

```
## # A tibble: 3 x 3
##    Jane  Mary   Bob
##   <int> <int> <int>
## 1     1     5     8
## 2     4     2    NA
## 3    NA     5     1
```

Other separators

What if the data are separated by tabs or something like "|"?

- Use `read_delim()` and specify the delimiter.
- Alternatively, specify the format as "|" with rio. Ex: `import(file.txt, format = "|")`

```
read_delim("1|5|8
          4|2|999
          999|5|1",
    delim = "|",
    col_names = c("Jane", "Mary", "Bob"),
    na = "999")
```

```
## # A tibble: 3 x 3
##            Jane  Mary   Bob
##           <chr> <int> <int>
## 1             1     5     8
## 2             4     2    NA
## 3           999     5     1
```

Specify column type

By default, these are all read in as integer. Let's change it.

```
read_delim("1|5|8
          4|2|999
          999|5|1",
    delim = "|",
    col_names = c("Jane", "Mary", "Bob"),
    na = "999",
    col_types = cols(
                  col_character(),
                  col_double(),
                  col_integer()))
```

```
## # A tibble: 3 x 3
##             Jane  Mary   Bob
##            <chr> <dbl> <int>
## 1              1     5     8
## 2              4     2    NA
## 3            999     5     1
```

Important things to think about

- When importing data, how many rows and columns do you expect? See if it matches using `dim(ob)` where ob is the data object.
- Be careful of missing data (how are they coded in the original file?)
- Always do some double checking to make sure everything read in correctly
  - e.g., `head()` `tail()`, `summary()`, `str()`

Importing data from other sources

The *haven* package

- Really powerful package - much better than the *foreign* package that comes pre-installed.
- Can read *and write* SPSS, SAS, and Stata files.
- By default, user-defined missing data will be read in as missing.
- Used by *rio* so all arguments should be passed directly

Example

Load a sample of the ECLS-K dataset

```
library(haven) # part of tidyverse so should already be installed
eclsk_haven <- read_sav("./data/ecls-k_samp.sav")
eclsk_haven
```

```
## # A tibble: 984 x 33
##    child_id teacher_id school_id    k_type school_type       sex    ethnic
##       <chr>      <chr>     <chr> <dbl+lbl>   <dbl+lbl> <dbl+lbl> <dbl+lbl>
##  1 0842021C    0842T02      0842         1           0         0         2
##  2 0905002C    0905T01      0905         1           1         0         5
##  3 0150012C    0150T01      0150         1           1         1         2
##  4 0556009C    0556T01      0556         1           1         1         4
##  5 0089013C    0089T04      0089         1           0         0         1
##  6 1217001C    1217T13      1217         0           0         1         6
##  7 1092008C    1092T01      1092         0           0         1         4
##  8 0083007C    0083T16      0083         1           0         0         1
##  9 1091005C    1091T02      1091         0           1         0         1
## 10 2006006C    2006T01      2006         1           1         0         1
## # ... with 974 more rows, and 26 more variables: famtype <dbl+lbl>,
## #   numsibs <dbl>, SES_cont <dbl>, SES_cat <dbl+lbl>, age <dbl>,
## #   T1RSCALE <dbl>, T1MSCALE <dbl>, T1GSCALE <dbl>, T2RSCALE <dbl>,
```

labels

- haven tries to maintain the attributes a variable had when it was in SPSS, SAS, or STATA.
- To do this, it provides a new *labelled* class
- This way, no information is lost, and you can decide what to do with them
    - remove labels
    - coerce to factor
- This is slightly different than how `rio::haven` handles the data on import

labelled class

```
##                    haven       rio
## child_id    character character
## teacher_id  character character
## school_id   character character
## k_type       labelled   numeric
## school_type  labelled   numeric
## sex          labelled   numeric
## ethnic       labelled   numeric
## famtype      labelled   numeric
## numsibs       numeric   numeric
## SES_cont      numeric   numeric
## SES_cat      labelled   numeric
## age           numeric   numeric
## T1RSCALE      numeric   numeric
## T1MSCALE      numeric   numeric
## T1GSCALE      numeric   numeric
## T2RSCALE      numeric   numeric
## T2MSCALE      numeric   numeric
## T2GSCALE      numeric   numeric
## IRTreadgain   numeric   numeric
## IRTmathgain   numeric   numeric
## IRTgkgain     numeric   numeric
```

Make them what you want

The main difference between the packages:

(note - these all work on a full data frame as well as individual columns)

If you want a variable to be numeric

- rio - nothing
- `haven::zap_labels()`

If you want a variable to be a factor (we'll talk more about these later)

- `rio::factorize()`
- `haven::as_factor()`

If you want a variable to be a character (more complicated with haven)

- `rio::characterize()`

Final notes for importing data

- Generally reading in data is not a big deal. Occasionally tricky formats can come up.
- `rio::import` should work most of the time - particularly in this class - but you may want need to pass additional arguments at times.
    - One of the drawbacks (but makes things simpler) is that you may not know which variables had labels attached and which did not.