UPPSALA UNIVERSITET

Uppsala Child & Baby Lab

Data Wrangling - Importing and preparing data for analyses

R for Psychology Research

Marcus Lindskog, Docent
2019-09-23

Overview

1. Keep data in a `tibble`
2. Import data with `readr`.
3. Data transformations with `dplyr`
4. Data tranformation with `tidyr`

What is data wrangling?

- Data wrangling is the process of getting your raw data into a format that can be used for analyses.
- Today we will work with a collection of packages from the `tidyverse` to do that.
- It is important to acknowledge that this is not the only way to do things.
- However, using the `tidyverse` gives you an integrated framework that helps you solve these tasks quite easy.

Keep your data in a `tibble`

What is a `tibble`?

- The `tibble` packages introduces a new data structure into R, the **tibble**.
- A tibble is a modern take on the data frame.
- It makes it easier and more consistent to work with tidy data in R.

Creating a `tibble`

- A regular `data.frame` can be coerced to tibble with:

```
library(psych)
library(tibble)

as_tibble(bfi)
```

Create a `tibble` from individual vectors

- A `tibble` can be created from individual vectors with:

```
tibble(
  x = 1:5,
  y = 1,
  z = x^2 + y+1,
  w = letters[1:5]
  )
```

Subsetting a `tibble`

- A `tibble` can be subsetted by name or position. But it works a bit different than for `data.frame`

```
my_data <- tibble(
  x = 1:20,
  y = 1,
  z = x^2 + y+1,
  w = letters[1:20]
  )

# Extract by name (1)
my_data$w

# Extract by name (2)
my_data[["w"]]
```

Subsetting a `tibble`

```
# Extract by position

my_data[[4]]
```

Import data with readr

`readr`.

- `readr` is a packages that turns flat files with rectangular data into data frames (`tibble`).
- The same functionality can be found in base R, but `readr` functions generally:
  - Are much faster.
  - Produce `tibbles` and don't make to many assumptions about your data (e.g., turns strings into factors).
  - Are more reproducible.

Functions in `readr`

- `read_csv` - reads comma-delimited files
- `read_csv2()`: reads semicolon-separated files
- `read_tsv()`: reads tab-delimited files
- `read_delim()`: reads in files with any (user specified) delimiter.

Functions in `readr`

- **`read_csv` - reads comma-delimited files**
- `read_csv2()`: reads semicolon-separated file
- `read_tsv()`: reads tab-delimited files
- `read_delim()`: reads in files with any (user specified) delimiter.

A note on the working directory

- When reading a file you need to know where it is (i.e., provide a path)
- But it is hard to give a path if you don't know where you are.
- You are in you're **working directory**, which can be found with:

```
getwd()
```

```
## [1] "/Volumes/ipb-users-1/marli361/@Undervisning/@Kurser/FoU - R for Psychology
```

- You can change your working directory with:

```
setwd("path")
```

- Or you can click **More** under the **Files** tab in RStudio.

Reading that .csv-file!

```
# read_csv("path")

heights_data <- read_csv("heights_data.csv")

bfi_data <- read_csv("heights_data.csv")
```

A bit more specific parsing

- `read_csv()` assumes that the first row contains column names. It also reads all lines, if noting else is specified. We can change that behavior.

```
#To skip one or more lines

bfi_data <- read_csv("bfi_data.csv",
                     skip = 3)

# To skip lines that begin with a specific character (i.e., comments)
bfi_data <- read_csv("bfi_data.csv",
                     comment = "#")

#If the data don't have column names in the first row
bfi_data <- read_csv("bfi_data.csv",
                     col_names = FALSE)

#To provide your own colnames if they are missing
height_data <- read_csv("height_data.csv",
                     col_names = c("A", "B"))
```

It is guess work.

- When you read a file with `readr` it tries to guess what data types are in your columns.
- This is good because it makes the function fast, but it can sometimes be problematic.
- If you know what you have in your columns, you can specify that directly in the `read_`- functions.

```
#To skip one or more lines

bfi_data <- read_csv("heights_data.csv",
                     col_types = col(
                       height = col_integer(),
                       cubit = col_character()
                     )
```

Other file formats.

- There are of course a lot of different file formats you might want to get into R.
- We can't cover them all. However, have a look at the following packages to solve some of your importing needs.
- `haven`: reads SPSS, STATA, and SAS files.
- `readxl`: reads Excel files (both .xls and .xlsx).
- `DBI` along with a database-specific backend (`RMySQL`, `RSQLite`, `RPostgreSQL`) to run queries against database.

Data transformation with `dplyr`

Functions from `dplyr`:

- Select, filter and arrange your data:
  - `select()`: Select columns from your dataset
  - `filter()`: Filter out certain rows that meet your criteria(s)
  - `arrange()`: Arrange your column data
- Create new variables:
  - `mutate()`: Create new columns by preserving the existing variables
- Summarize that data:
  - `group_by()`: Group different observations together.
  - `summarise()`: Summarise any of the above functions
- Join data with other data frames
  - *`join()`: Perform left, right, full, and inner joins in R*

`filter()`

- The `filter()` function subsets a data frame based on a series of criterion

```
heights_data <- read_csv("heights_data.csv")

#filter(data_frame, expression_to_filter_1, expression_to_filter_2,...)

filter(heights_data, height == 71)

filter(heights_data, height == 68, height == 71)

filter(heights_data, !(height ==68 | height == 71))

filter(heights_data, height %in% c(68, 70, 71))
```

- `filter()` returns a tibble and the input is left unchanged.

Piping that filter

- If you are running a large number of manipulations on the same data frame it is clunky to save each intermediate step in a new variable.
- To help you overcome this problem we have the pipe operator `%>%` from the `` `magrittr ``

```
heights_data <- read_csv("heights_data.csv")

#filter(data_frame, expression_to_filter_1, expression_to_filter_2,...)

filtered_heights_data <- heights_data %>%
  filter(height == 71)
```

arrange()

- The `arrange()` function changes the order of the rows in a data frame

```
bfi_data <- read_csv("bfi_data.csv")

#arrange(data_frame, column_to_arrange_1, column_to_arrange_2,...)

arrange(bfi_data, A1, A2)

#Use descending order instead

arrange(bfi_data, desc(A1), A2)
```

- Of course, filter and arrange can be combined

```
bfi_data <- read_csv("bfi_data.csv")

new_filtered_arranged_data <- bfi_data %>%
  filter(A1 %in% c(1,2,3)) %>%
  arrange(desc(A1), A2)
```

`select()`

- `select()` helps select a subset of columns from a data frame.

```
bfi_data <- read_csv("bfi_data.csv")

#select(data_frame, column_1, column_2,...)

#select by name
select(bfi_data, A1, A2, C5)

#select an interval
select(bfi_data, A1:C5)

#select all but specified columns
select(bfi_data, -A1, -A2, -C5)
select(bfi_data, -(A1:C5))
```

`select()`'s little helpers

- `starts_with("arn")` matches columns begining with "abc".
- `ends_with("klm")` matches columns ending with "klm".
- `contains("una")` matches names containing "una".

```
bfi_data <- read_csv("bfi_data.csv")

select(bfi_data, starts_with("A"))

#select an interval
select(bfi_data, ends_with("4"))
```

Rename a variable

- `rename()` is a useful tool to rename columns.

```
bfi_data <- read_csv("bfi_data.csv")

rename(bfi_data, A_1 = A1)
```

Let's combine

```r
bfi_data <- read_csv("bfi_data.csv")

new_changed_data_frame <- bfi_data %>%
  select(starts_with("A")) %>%
  rename(A_1 = A1, A_4 = A4) %>%
  filter(A_4 %in% c(3,4,5)) %>%
  arrange(A_1, desc(A3))
```

mutate()

- It is **very** often the case that we need to create (add) new variables that are some combination of existing variables or add some new information.
- This is can be done smoothly with `mutate()` and `transmute()`

```
bfi_data <- read_csv("bfi_data.csv")

#mutate(data_frame, new_var_1, new_var_2,...)
# mutate adds a new variable, and keeps all the old ones.
mutate(bfi_data,
  A = A1+A2+A3+A4,
  C = (C1*C2)/(C3*C4)
)

# transmute only keeps the new variables
transmute(bfi_data,
  A = A1+A2+A3+A4,
  C = (C1*C2)/(C3*C4),
  D = as_factor(rep(c("A", "B", "C", "D"), each = n()/4))
)
```

`mutate()`'s little helpers.

- Read p. 56-58 for some functions that can be very helpful when using `mutate` and `transmute`.
- Other useful versions of `mutate` is `mutate_all`, `mutate_if`, and `mutate_at`. Google them to find out exctly how they work.

Let's combine

```
bfi_data <- read_csv("bfi_data.csv")

new_changed_data_frame <- bfi_data %>%
  select(starts_with("A")) %>%
  rename(A_1 = A1, A_4 = A4) %>%
  filter(A_4 %in% c(3,4,5)) %>%
  mutate(K = (A_1 + A_4)/A3) %>%
  arrange(K)
```

summarize() and group_by()

- We often want to create summaries of our data.
- `summarize()` collapses data into a single row.

```
bfi_data <- read_csv("bfi_data.csv")

#summarize(data_frame, summary)
summarize(bfi_data,
          a1_mean = mean(A1, na.rm = TRUE),
          count = n(),
          c1_na = sum(is.na(C1))
          )
```

Grouped summaries

- To get grouped summaries, first use `group_by()`
- `summarize()` collapses data into a single row.

```
bfi_data <- read_csv("bfi_data.csv")

group_by(bfi_data, A1) %>%
summarize(a2_mean = mean(A2, na.rm = TRUE),
          count = n(),
          c1_na = sum(is.na(C1))
          )
```

Let's combine

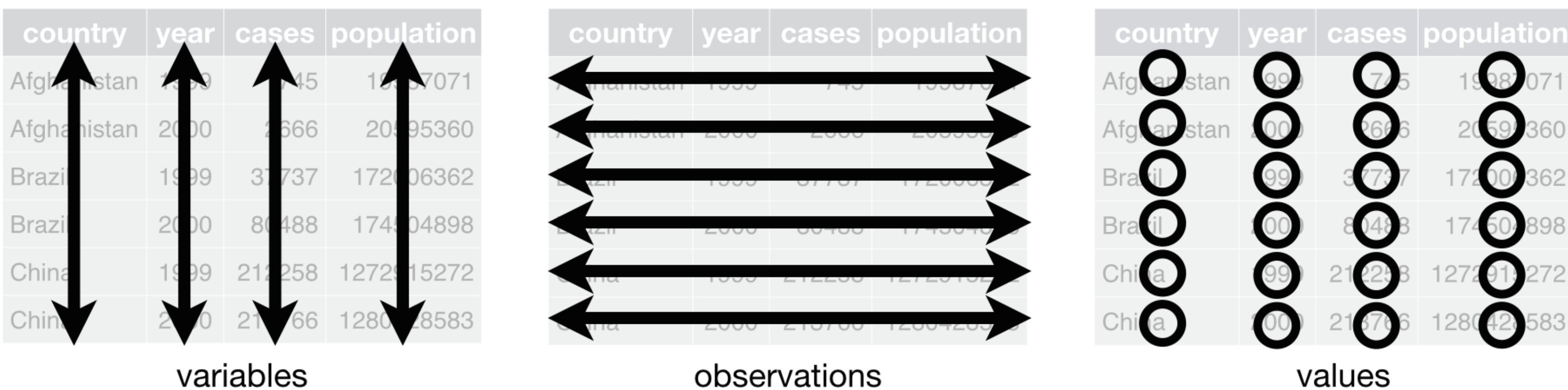
```
bfi_data <- read_csv("bfi_data.csv")

new_changed_data_frame <- bfi_data %>%
  select(starts_with("A")) %>%
  rename(A_1 = A1, A_4 = A4) %>%
  filter(A_4 %in% c(3,4,5)) %>%
  mutate(K = (A_1 + A_4)/A3) %>%
  arrange(K) %>%
  group_by(A2, A3) %>%
  summarize(mean_k = mean(K, na.rm = TRUE),
            count = n(),
            se_age = sd(age, na.rm = TRUE)/sqrt(sum(!is.na(K))))
```

Data tranformation with `tidyr`

`tidyr`

- `tidyr` has the functions we need to rearrange data into different formats.
- It was originally designed to get data into a **tidy** format, but you can of course use it to get your data into any format you need.


country
year
cases
population
variables
observations
values

Important functions in `tidyr`

- `gather()`: Gathers multiple columns and converts them into key-value pairs.
- `spread()`: Takes two columns and spreads them into multiple columns.
- `separate()`: Helps in separating or splitting a single column into numerous columns
- `unite()`: Works opposite to the separate() function. Combines two or more columns into one

Gathering

```
head(table4a,3)
```

```
## # A tibble: 3 x 3
##   country     `1999` `2000`
##   <chr>        <int>  <int>
## 1 Afghanistan    745   2666
## 2 Brazil       37737  80488
## 3 China       212258 213766
```

- Here we have a data frame with *values* (1999, 2000) in variable names.

gather()

- We can collect them into a variable by using `gather()`

```
table4a %>%
  gather(2:3, key = "year", value = "population")

table4a %>%
  gather(c("1999", "2000"), key = "year", value = "population")

table4a %>%
  gather("1999", "2000", key = "year", value = "population")
```

Spreading

- `spread()` does the opposite of `gather()`
- It takes observations that are scattered into multiple rows, and puts them in columns
- What is the problem with this data frame?

```
head(table2, 6)
```

```
## # A tibble: 6 x 4
##   country      year type            count
##   <chr>       <int> <chr>           <int>
## 1 Afghanistan  1999 cases             745
## 2 Afghanistan  1999 population   19987071
## 3 Afghanistan  2000 cases            2666
## 4 Afghanistan  2000 population   20595360
## 5 Brazil       1999 cases           37737
## 6 Brazil       1999 population 172006362
```

```
table2 %>%
  spread(key = type, value = count)
```

`separate()`

- `separate()` pulls apart one column into multiple columns.

```
table3
```

```
## # A tibble: 6 x 3
##   country      year rate
## * <chr>       <int> <chr>
## 1 Afghanistan  1999 745/19987071
## 2 Afghanistan  2000 2666/20595360
## 3 Brazil       1999 37737/172006362
## 4 Brazil       2000 80488/174504898
## 5 China        1999 212258/1272915272
## 6 China        2000 213766/1280428583
```

```r
table3 %>%
  separate(rate, c("cases", "population"), sep = "/")
```

```
## # A tibble: 6 x 4
##   country      year cases  population
##   <chr>       <int> <chr>  <chr>
## 1 Afghanistan  1999 745    19987071
## 2 Afghanistan  2000 2666   20595360
## 3 Brazil       1999 37737  172006362
## 4 Brazil       2000 80488  174504898
## 5 China        1999 212258 1272915272
## 6 China        2000 213766 1280428583
```

unite()

- unite() is the inverse of separate()

```
table3 %>%
  separate(rate, c("cases", "population"), sep = "/") %>%
  unite(rate, cases, population, sep = "-")
```

```
## # A tibble: 6 x 3
##   country      year rate
##   <chr>       <int> <chr>
## 1 Afghanistan  1999 745-19987071
## 2 Afghanistan  2000 2666-20595360
## 3 Brazil       1999 37737-172006362
## 4 Brazil       2000 80488-174504898
## 5 China        1999 212258-1272915272
## 6 China        2000 213766-1280428583
```

That's all folks!