AVED: Tidy data (tidyr)

Štěpán Mikula

Data formats

"It is often said that 80 % of data analysis is spent on the cleaning and preparing data."

"Tidy datasets are all alike but every messy dataset is messy in its own way."

– Hadley Wickham

Today we will deal with the bad, the ugly, and the messy.

Messy data

Data comes in many different formats. Typically we work with variables observed for cross-sectional unit (person, geographical unit,. . .) in time (1,. . . ,infinity).

Multiple rows contain one observation (state-year combination):

```
## # A tibble: 12 × 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
## 7       Brazil  2000      cases      80488
## 8       Brazil  2000 population  174504898
## 9        China  1999      cases     212258
## 10       China  1999 population 1272915272
## 11       China  2000      cases     213766
## 12       China  2000 population 1280428583
```

Real life example: CSV files exported from OECD (https://stats.oecd.org/), FAO or Barro-Lee data set

One variable is in many columns + header contain values:

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

Real life example: Files exported from http://data.worldbank.org/ or UN data on population from `wpp*` packages

. . . or other monstrosities (e.g. values aggregated into one "cell"):

```
## # A tibble: 6 × 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
```

Real life example: dates in CSV files from Google Trends or some tables from Eurostat

Unfortunately creativity is not forbidden nor punishable.

Tidy data

How to organize your data to get a format which is easy to work with?

Brief "tidy" format definition:

- Each variable forms a column
- Each observation forms a row
- Each type of observational unit forms a table

For full definition see Wickham, H. (2014): Tidy Data, https://www.jstatsoft.org/article/view/v059i10

Example of tidy data:

```
## # A tibble: 6 × 4
##       country  year  cases population
##         <chr> <int>  <int>      <int>
## 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
```

Real life example: Penn World Tables (excel file from http://www.rug.nl/research/ggdc/data/pwt/pwt-9.0) or Penn World Tables from `pwt8` package.

What the tibble?

A `tibble` is new implementation of a `data.frame`. Tibble is supposed to be a pronunciation of `tbl` which is an actual class of a `tibble` table.

```
tidyr::table2 %>% class
```

```
## [1] "tbl_df"     "tbl"        "data.frame"
```

As you can see `tibble` table still has class `data.frame` therefore it is accessible to methods designed for `data.frames`. There is only one major difference between `tibble` and `data.frame`. Tibble never coerces characters into factors (default behavior of `data.frames`). It also supports advanced printing methods and it is a bit faster in some cases.

Hack of the day: Methods designed for data.frames can handle tibbles. In most cases. If there is problem with a `tibble` (e.g. in `stargazer`) you can coerce tibble to ordinary data frame:

```
tidyr::table2 %>% as.data.frame %>% class
```

```
## [1] "data.frame"
```

Basic tools for data "tidying"

CRAN repository contains a lot of packages which contains tool useful for data tidying. For example:

- reshape (very obsolete),
- reshape2 (obsolete),
- **tidyr** (daisy fresh)

```
library(tidyr)
```

tidyr includes two basic functions `spread()` and `gather()` which can handle majority of tyding cases.

spread()

`spread()` is a function for spreading a key-value pair across multiple columns

Basic example:

```
table2
```

```
## # A tibble: 12 × 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
## 7       Brazil  2000      cases      80488
## 8       Brazil  2000 population  174504898
## 9        China  1999      cases     212258
## 10       China  1999 population 1272915272
## 11       China  2000      cases     213766
## 12       China  2000 population 1280428583
```

```
spread(data = table2, key = type, value = count)
```

```
## # A tibble: 6 × 4
##       country  year  cases population
## *       <chr> <int>  <int>      <int>
## 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
```

Arguments:

- `data` – an input data.frame
- `key` – bare name of the column whose values will be used as column headings
- `value` – bare name of the column whose values will populate the cells

> **Note (non-standard evaluation, NSE):** tidyr is among packages which use so-called non-standard evaluation (NSE, see `vignette("nse")`). The use of NSE has practical implications. With NSE the names of variables (columns) are called by bare (unquoted) names. There are standard evaluation (SE) versions of some functions ending with "_" (e.g. NSE `spread()` and SE `spread_()`)

Basic arguments are sufficient for uncorrupted data, but not all data sets are flawless. . .

```
table2[-2,] %>% spread(type, count)
```

```
## # A tibble: 6 × 4
##       country  year  cases population
## *       <chr> <int>  <int>      <int>
## 1 Afghanistan  1999    745         NA
## 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
```

`spread()` automatically add `NA` to empty cells. You can change this behavior using

- `fill` – If set, missing values will be replaced with this value.

Assume that we want (for some reason) add zeros to empty cells:

```
table2[-2,] %>% spread(type, count, fill = 0)
```

```
## # A tibble: 6 × 4
##       country  year  cases population
## *       <chr> <int>  <dbl>      <dbl>
## 1 Afghanistan  1999    745          0
## 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
```

What if we have no observation for Afghanistan in 1999?

```
table2[-c(1:2),] %>% spread(type, count)
```

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

`spread()` will not create row for Afghanistan in 1999. That might be a problem. (Perhaps we want to add the values from different source later.) Fortunately we can modify this behavior of `separate()` by:

- `drop` – If FALSE, will keep factor levels that don't appear in the data, filling in missing combinations with fill.

```
table2[-c(1:2),] %>% spread(type, count, drop = FALSE)
```

```
## # A tibble: 6 × 4
##       country  year  cases population
## *       <chr> <int>  <int>      <int>
## 1 Afghanistan  1999     NA         NA
## 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
```

```
table2[-c(1:2),] %>% spread(type, count, drop = FALSE, fill = 0)
```

```
## # A tibble: 6 × 4
##       country  year  cases population
## *       <chr> <int>  <dbl>      <dbl>
## 1 Afghanistan  1999      0          0
## 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
```

gather()

`gather()` takes multiple columns and collapses into key-value pairs, duplicating all other columns as needed. You use `gather()` when you notice that you have columns that are not variables.

Basic example:

```
table4a
```

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

```
gather(table4a, key = year, value = value, -country)
```

```
## # A tibble: 6 × 3
##       country  year  value
##         <chr> <chr>  <int>
## 1 Afghanistan  1999    745
## 2      Brazil  1999  37737
## 3       China  1999 212258
## 4 Afghanistan  2000   2666
## 5      Brazil  2000  80488
## 6       China  2000 213766
```

Arguments:

- `key`,`value` – Names of key and value columns to create in output. `key` is the bare name of an existing column. `value` is the name of the column being created.
- `...` – Specification of columns to gather. Use bare variable names. Select all variables between `x` and `z` with `x:z`, exclude `y` with `-y`.

So, it is possible to get the same result with the list of columns to gather and list of column not to gather:

```
gather(table4a, key = year, value = value, `1999`:`2000`)
```

```
## # A tibble: 6 × 3
##       country  year  value
##         <chr> <chr>  <int>
## 1 Afghanistan  1999    745
## 2      Brazil  1999  37737
## 3       China  1999 212258
## 4 Afghanistan  2000   2666
## 5      Brazil  2000  80488
## 6       China  2000 213766
```

Hack of the day: If you need to use ugly variable (column) name use "'" sign. Ugly is most often a name which is actually numeric or starts with numeric. Run "1_ahoj" and the same expression closed in the sign "'" and compare the results. (Ugly names can be quite often found in Stata data set converted into R – e.g. data sets from World Values Survey.)

Some tips... tidyr functions have recently adopted extended column-specifying options which are very useful for programming. Those options are identical with options available in `select()` function from dplyr package.

Some examples of standard evaluation (SE)

SE versions of tidyr functions provide the same functionality. The difference is in the way of specifying columns. See some examples:

spread()

```r
spread(data = table2, key = type, value = count)
```

```
## # A tibble: 6 × 4
##       country  year  cases population
## *       <chr> <int>  <int>      <int>
## 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
spread_(data = table2, key_col = "type", value_col = "count")
```

```
## # A tibble: 6 × 4
##       country  year  cases population
## *       <chr> <int>  <int>      <int>
## 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
```

gather()

```r
gather(table4a, key = year, value = value, `1999`:`2000`)
```

```
## # A tibble: 6 × 3
##       country  year  value
##         <chr> <chr>  <int>
## 1 Afghanistan  1999    745
## 2      Brazil  1999  37737
## 3       China  1999 212258
## 4 Afghanistan  2000   2666
## 5      Brazil  2000  80488
## 6       China  2000 213766
```

```r
gather_(table4a, key_col = "year", value_col = "value", gather_cols = c("1999","2000"))
```

```
## # A tibble: 6 × 3
##       country  year  value
##         <chr> <chr>  <int>
## 1 Afghanistan  1999    745
## 2      Brazil  1999  37737
## 3       China  1999 212258
## 4 Afghanistan  2000   2666
## 5      Brazil  2000  80488
## 6       China  2000 213766
```

Advanced stuff (more of tidyr)

Implicit missing observations in datasets: complete()

Observations can be missing explicitly or implicitly in your data set.

In the case of explicit missing values there is a combination of cross-sectional and time unit with `NA` value. See the example (population of Norway and Sweden from 1995 to 2000):

```
POP_explicit
```

```
## # A tibble: 12 × 3
##    country  year population
##      <chr> <int>      <int>
## 1   Norway  1995    4359788
## 2   Norway  1996         NA
## 3   Norway  1997    4412958
## 4   Norway  1998    4440109
## 5   Norway  1999         NA
## 6   Norway  2000    4491572
## 7   Sweden  1995         NA
## 8   Sweden  1996    8849420
## 9   Sweden  1997    8859106
## 10  Sweden  1998         NA
## 11  Sweden  1999         NA
## 12  Sweden  2000         NA
```

Missing observations (whole rows) can be omitted from the data set. In that case we deal with implicit missing values:

```
POP_implicit
```

```
## # A tibble: 6 × 3
##   country  year population
##     <chr> <int>      <int>
## 1  Norway  1995    4359788
## 2  Norway  1997    4412958
## 3  Norway  1998    4440109
## 4  Norway  2000    4491572
## 5  Sweden  1996    8849420
## 6  Sweden  1997    8859106
```

`tidyr` contains functions which can turn implicit to turn implicit missing values into explicit ones. `complete()` returns data set with implicit missing observations added:

```
complete(data = POP_implicit,country,year)
```

```
## # A tibble: 10 × 3
##    country  year population
##      <chr> <int>      <int>
## 1   Norway  1995    4359788
## 2   Norway  1996         NA
## 3   Norway  1997    4412958
## 4   Norway  1998    4440109
## 5   Norway  2000    4491572
## 6   Sweden  1995         NA
## 7   Sweden  1996    8849420
## 8   Sweden  1997    8859106
## 9   Sweden  1998         NA
## 10  Sweden  2000         NA
```

Arguments:

- `data` – An input data frame
- `...` – Specification of columns to expand. These columns form an unique ID of an observation (a country-year pair in this case).

Note, that if there is a value missing for both countries in a specific year than this implicit missing observation is missing even in the output of `complete()`. **`complete()` works only with values present in input dataset!**

To overcome this restriction it is possible to supply a vector with all possible values to `complete()`:

```
complete(data = POP_implicit,country,year=1995:2000)
```

```
## # A tibble: 12 × 3
##    country  year population
##      <chr> <int>      <int>
## 1   Norway  1995    4359788
## 2   Norway  1996         NA
## 3   Norway  1997    4412958
## 4   Norway  1998    4440109
## 5   Norway  1999         NA
## 6   Norway  2000    4491572
## 7   Sweden  1995         NA
## 8   Sweden  1996    8849420
## 9   Sweden  1997    8859106
## 10  Sweden  1998         NA
## 11  Sweden  1999         NA
## 12  Sweden  2000         NA
```

Resulting data frame is now identical with `POP_explicit`.

`complete()` is actually a wrapper around functions `expand()` and `replace_na()` from tidyr and `left_join()` from dplyr. `expand()` returns all combinations of values in ID columns. Newly created data frame is subsequently joined using `left_join()` with the original one.

For more related functions see help for `expand()` function.

Replacing missing values: fill()

Fills missing values in using the previous entry.

```
fill(data = POP_explicit, population, .direction = "down")
```

```
## # A tibble: 12 × 3
##    country  year population
##      <chr> <int>      <int>
## 1   Norway  1995    4359788
## 2   Norway  1996    4359788
## 3   Norway  1997    4412958
## 4   Norway  1998    4440109
## 5   Norway  1999    4440109
## 6   Norway  2000    4491572
## 7   Sweden  1995    4491572
## 8   Sweden  1996    8849420
## 9   Sweden  1997    8859106
## 10  Sweden  1998    8859106
## 11  Sweden  1999    8859106
## 12  Sweden  2000    8859106
```

Arguments:

- `data` – An input data frame
- `...` – Specification of columns to fill. Use bare variable names. Select all variables between `x` and `z` with `x:z`, exclude `y` with `-y`.
- `.direction` – Direction in which to fill missing values. Currently either "down" (the default) or "up".

You need to be extra careful while using `fill()` because it is sensitive to ordering. See example:

```
POP_explicit %>% arrange(year,country)
```

```
## # A tibble: 12 × 3
##    country  year population
##      <chr> <int>      <int>
## 1   Norway  1995    4359788
## 2   Sweden  1995         NA
## 3   Norway  1996         NA
## 4   Sweden  1996    8849420
## 5   Norway  1997    4412958
## 6   Sweden  1997    8859106
## 7   Norway  1998    4440109
## 8   Sweden  1998         NA
## 9   Norway  1999         NA
## 10  Sweden  1999         NA
## 11  Norway  2000    4491572
## 12  Sweden  2000         NA
```

```
POP_explicit %>% arrange(year,country) %>% fill(population)
```

```
## # A tibble: 12 × 3
##    country  year population
##      <chr> <int>      <int>
## 1   Norway  1995    4359788
## 2   Sweden  1995    4359788
## 3   Norway  1996    4359788
## 4   Sweden  1996    8849420
## 5   Norway  1997    4412958
## 6   Sweden  1997    8859106
## 7   Norway  1998    4440109
## 8   Sweden  1998    4440109
## 9   Norway  1999    4440109
## 10  Sweden  1999    4440109
## 11  Norway  2000    4491572
## 12  Sweden  2000    4491572
```

There might be a problem even if ordering is correct. In the first example of using `fill()` the value for the first year in Sweden was filled by the last value for Norway!

This problem can be easily solved using `group_by()` function from dplyr package (later alligator). See example:

```
POP_explicit %>% group_by(country) %>% fill(population)
```

```
## Source: local data frame [12 x 3]
## Groups: country [2]
##
##    country  year population
##      <chr> <int>      <int>
## 1   Norway  1995    4359788
## 2   Norway  1996    4359788
## 3   Norway  1997    4412958
## 4   Norway  1998    4440109
## 5   Norway  1999    4440109
## 6   Norway  2000    4491572
## 7   Sweden  1995         NA
## 8   Sweden  1996    8849420
## 9   Sweden  1997    8859106
## 10  Sweden  1998    8859106
## 11  Sweden  1999    8859106
## 12  Sweden  2000    8859106
```

Extracting cell content into multiple columns: separate(), unite(), and others

Sometimes cells contains data in strange forms – e.g. multiple data joined together (e.g. min-max/from-to format like `10.5-14.0`):

```
table3
```

```
## # A tibble: 6 × 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
```

Easiest way to separate values into two columns is using function `separate()`:

```
separate(table3, col = rate, into = c("X","Y"), sep="/", remove = TRUE, convert = TRUE)
```

```
## # A tibble: 6 × 4
##       country  year      X          Y
## *       <chr> <int>  <int>      <int>
## 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
```

Arguments:

- `data` – An input data frame.
- `col` – Bare name of the column to be separated.
- `into` – Names of new columns to create as character vector.
- `sep` – Separator between columns.
- `remove` – If `TRUE` (default) then original column is removed.
- `convert` – If `TRUE` (default is `FALSE`), will run `type.convert` with `as.is = TRUE` on new columns.

`separate()` divides input string into chunks separated by char given in `sep` argument. Each chunk is assigned to one newly created column. But what if the input structure is damaged:

```
table3a
```

```
## # A tibble: 6 × 3
##       country  year                      rate
## *       <chr> <int>                     <chr>
## 1 Afghanistan  1999
## 2 Afghanistan  2000          2666/20595360
## 3      Brazil  1999                172006362
## 4      Brazil  2000         80488/174504898
## 5       China  1999 212258/1272915272/3456888
## 6       China  2000        213766/1280428583
```

In this case the 1st row contains empty value (no chunks), 3rd only one chunk, and 5th 3 chunks. Running `separate()` in the previous setting will yield:

```
separate(table3a, col = rate, into = c("X","Y"), sep="/", remove = TRUE, convert = TRUE)
```

```
## Warning: Too many values at 1 locations: 5

## Warning: Too few values at 2 locations: 1, 3

## # A tibble: 6 × 4
##       country  year        X          Y
## *       <chr> <int>    <int>      <int>
## 1 Afghanistan  1999       NA         NA
## 2 Afghanistan  2000      2666   20595360
## 3      Brazil  1999 172006362         NA
## 4      Brazil  2000     80488  174504898
## 5       China  1999    212258 1272915272
## 6       China  2000    213766 1280428583
```

You can slightly modify this behavior using arguments `extra` and `fill`. `extra` controls what happens when there are too many pieces. There are three valid options:

- `"warn"` – (the default) emit a warning and drop extra values.
- `"drop"` – drop any extra values without a warning.
- `"merge"` – only splits at most length(into) times

`fill` controls what happens when there are not enough pieces. There are again three valid options:

- `"warn"` – (the default) emit a warning and fill from the right
- `"right"` – fill with missing values on the right
- `"left"` – fill with missing values on the left

```
separate(table3a, col = rate, into = c("X","Y"), sep="/", remove = TRUE, convert = TRUE, extra = "merge
```

```
## # A tibble: 6 × 4
##       country  year      X                  Y
## *       <chr> <int>  <int>              <chr>
## 1 Afghanistan  1999     NA
## 2 Afghanistan  2000   2666           20595360
## 3      Brazil  1999     NA          172006362
## 4      Brazil  2000  80488          174504898
## 5       China  1999 212258 1272915272/3456888
## 6       China  2000 213766         1280428583
```

Notice that with `extra = "merge"` it was unable to convert column `Y` to integer.

tidyr contains function `unite()` which is complementary to `separate()`. It joins multiple columns into one.

There are more functions in tidyr designed to help you to get data from cells in *strange* formats:

- `extract()` provides similar functionality as `separate()` with broader support for regular expressions.
- `extract_numeric()` returns only numeric part of input string. It is useful if you are dealing with data in a format such as `"12 years"`. `extract_numeric()` assumes English locale – compare results of `extract_numeric("12,1 years")` and `extract_numeric("12.1 years")`.

  Function `extract_numeric()` is deprecated from version 0.6.0 on. Suggested replacement is `readr::parse_number()`. `parse_numeric()` allows user to set locale and string for missing values. See `readr` for similar functions.

Extracting cell content into multiple rows: separate_rows()

Sometimes a cell contains delimited values of the same nature (e.g. sequence of temperatures recorded etc.). In this case we want to get only one column with all values recorded. It is easy with `separate_rows()` which works quite like `separate()`:

```r
table3
```

```
## # A tibble: 6 × 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
separate_rows(table3, rate, sep="/", convert = TRUE)
```

```
## # A tibble: 12 × 3
##        country  year       rate
##          <chr> <int>      <int>
## 1  Afghanistan  1999        745
## 2  Afghanistan  1999   19987071
## 3  Afghanistan  2000       2666
## 4  Afghanistan  2000   20595360
## 5       Brazil  1999      37737
## 6       Brazil  1999  172006362
## 7       Brazil  2000      80488
## 8       Brazil  2000  174504898
## 9        China  1999     212258
## 10       China  1999 1272915272
## 11       China  2000     213766
## 12       China  2000 1280428583
```

Arguments:

- `data` – An input data frame.
- `...` – Specification of columns to fill. Use bare variable names. Select all variables between `x` and `z` with `x:z`, exclude `y` with `-y`.
- `sep` – Separator between columns.
- `convert` – If `TRUE` (default is `FALSE`), will run `type.convert` with `as.is = TRUE` on new columns.

Exercises

OECD data

Get annual data on average unemployment duration from OECD database and make it tidy.

What you need to do:

- Load data set is available in the file `OECD_AVDDUR.Rdata`
- Convert `obsTime` from character into numeric
- There are values (`obsValue`) of unemployment duration for each country-year (`COUNTRY`,`obsTime`) pair which differs in sex (`SEX`) and age (`AGE`) of target group. Reorganize data to get one column for each sex-age combination.

Solution

You can download daisy fresh data from API of OECD database using following code_

```
library(OECD)
get_dataset("AVD_DUR") %>% as_data_frame() -> odata
```

. . . or use downloaded data from local file

```
load("data/OECD_AVDDUR.Rdata")
print(odata)
```

```
## # A tibble: 10,639 × 7
##    COUNTRY   SEX   AGE FREQUENCY TIME_FORMAT obsTime obsValue
## *    <chr> <chr> <chr>     <chr>       <chr>   <chr>    <dbl>
## 1      AUS    MW  1519         A         P1Y    1978 4.629423
## 2      AUS    MW  1519         A         P1Y    1979 5.420051
## 3      AUS    MW  1519         A         P1Y    1980 5.268784
## 4      AUS    MW  1519         A         P1Y    1981 5.081453
## 5      AUS    MW  1519         A         P1Y    1982 5.047878
## 6      AUS    MW  1519         A         P1Y    1983 6.498000
## 7      AUS    MW  1519         A         P1Y    1984 6.484891
## 8      AUS    MW  1519         A         P1Y    1985 6.387451
## 9      AUS    MW  1519         A         P1Y    1986 5.955381
## 10     AUS    MW  1519         A         P1Y    1987 6.026455
## # ... with 10,629 more rows
```

`obsTime` is a character. In order to convert it into integer we can use `readr::parse_number()` or regular expressions.

Remeber that it is useful to chceck all values before using brutal force methods:

```
odata$obsTime %>% unique
```

```
##  [1] "1978" "1979" "1980" "1981" "1982" "1983" "1984" "1985" "1986" "1987"
## [11] "1988" "1989" "1990" "1991" "1992" "1993" "1994" "1995" "1996" "1997"
## [21] "1998" "1999" "2000" "2001" "2002" "2003" "2004" "2005" "2006" "2007"
## [31] "2008" "2009" "2010" "2011" "2012" "2013" "2014" "1976" "1977" "2015"
## [41] "1971" "1972" "1973" "1974" "1975" "1968" "1969" "1970"
```

All values of obsTime are "integers in character" – no problem there – we can use `readr::parse_number()`

```
odata$obsTime %<>% readr::parse_number()
```

Let's transform the table. Each row is a unique combination of country, year, sex, and group. We want to have country-year pair in rows and rest in columns.

```
odata %>%
  # Create combination of SEX and AGE
  unite(group, SEX, AGE, sep = "_") %>%
  # spread values into many columns according to newlay crated key column group
  spread(group, obsValue)
```

```
## # A tibble: 630 × 22
##    COUNTRY FREQUENCY TIME_FORMAT obsTime MEN_1519 MEN_1524 MEN_2024
## *    <chr>     <chr>       <chr>   <dbl>    <dbl>    <dbl>    <dbl>
## 1      AUS         A         P1Y    1978    4.232 4.443016    4.752
## 2      AUS         A         P1Y    1979    5.168 5.491391    5.943
## 3      AUS         A         P1Y    1980    4.835 5.466771    6.318
## 4      AUS         A         P1Y    1981    4.625 5.479961    6.579
## 5      AUS         A         P1Y    1982    4.837 5.558256    6.386
## 6      AUS         A         P1Y    1983    6.498 7.442876    8.350
## 7      AUS         A         P1Y    1984    6.529 8.277356   10.136
## 8      AUS         A         P1Y    1985    6.451 8.467028   10.693
## 9      AUS         A         P1Y    1986    6.182 7.734859    9.547
## 10     AUS         A         P1Y    1987    5.923 8.089628   10.615
## # ... with 620 more rows, and 15 more variables: MEN_2554 <dbl>,
## #   MEN_5599 <dbl>, MEN_900000 <dbl>, MW_1519 <dbl>, MW_1524 <dbl>,
## #   MW_2024 <dbl>, MW_2554 <dbl>, MW_5599 <dbl>, MW_900000 <dbl>,
## #   WOMEN_1519 <dbl>, WOMEN_1524 <dbl>, WOMEN_2024 <dbl>,
## #   WOMEN_2554 <dbl>, WOMEN_5599 <dbl>, WOMEN_900000 <dbl>
```

World Population Prospects 2015

Data from UN World Population Prospects 2015 contains net migration (in thousands) for five years periods. Data are available in `wpp2015_migration.Rdata`. Problem is that data are messy as hell. Load it and make it tidy!

Solution:

You can get the table from `wpp2015` package:

```r
library(wpp2015)
data("migration")
```

or use data from local file:

```r
load("data/wpp2015_migration.Rdata")

# Convert data into data_frame
migration %<>% as_data_frame()

# Take a look
migration %>% print
```

```
## # A tibble: 241 × 32
##    country_code
##           <int>
## 1           900
## 2           901
## 3           902
## 4           941
## 5           934
## 6           948
## 7          1503
## 8          1517
## 9          1502
## 10         1501
## # ... with 231 more rows, and 31 more variables: name <fctr>,
## #   `1950-1955` <dbl>, `1955-1960` <dbl>, `1960-1965` <dbl>,
## #   `1965-1970` <dbl>, `1970-1975` <dbl>, `1975-1980` <dbl>,
## #   `1980-1985` <dbl>, `1985-1990` <dbl>, `1990-1995` <dbl>,
## #   `1995-2000` <dbl>, `2000-2005` <dbl>, `2005-2010` <dbl>,
## #   `2010-2015` <dbl>, `2015-2020` <dbl>, `2020-2025` <dbl>,
## #   `2025-2030` <dbl>, `2030-2035` <dbl>, `2035-2040` <dbl>,
## #   `2040-2045` <dbl>, `2045-2050` <dbl>, `2050-2055` <dbl>,
## #   `2055-2060` <dbl>, `2060-2065` <dbl>, `2065-2070` <dbl>,
## #   `2070-2075` <dbl>, `2075-2080` <dbl>, `2080-2085` <dbl>,
## #   `2085-2090` <dbl>, `2090-2095` <dbl>, `2095-2100` <dbl>
```

We make this data tidy in one step!

```r
migration %<>% gather(period,value,-country_code,-name)

migration %>% print
```

```
## # A tibble: 7,230 × 4
##    country_code
##           <int>
## 1           900
## 2           901
## 3           902
## 4           941
## 5           934
## 6           948
## 7          1503
## 8          1517
## 9          1502
## 10         1501
## # ... with 7,220 more rows, and 3 more variables: name <fctr>,
## #   period <chr>, value <dbl>
```

Data are tidy now in formal sense, but the period is quite useless in this form. It is better to describe the period by middle year.

In the first step we create two columns out of period containg first (year1) and last (year5) year. Notice, that newly created columns are converted into integer.

```r
migration %>%
  separate(period, c("year1","year5"), sep = "-", convert = TRUE) %>%
  # Using mutate() from dplyr we can easily create year2.5 variable:
  mutate(
    year2.5 = mean(c(year1,year5))
  )
```

```
## # A tibble: 7,230 × 6
##    country_code
##           <int>
## 1           900
## 2           901
## 3           902
## 4           941
## 5           934
## 6           948
## 7          1503
## 8          1517
## 9          1502
## 10         1501
## # ... with 7,220 more rows, and 5 more variables: name <fctr>,
## #   year1 <int>, year5 <int>, value <dbl>, year2.5 <dbl>
```

Google Forms

Google Forms is a free platform for creating simple questionnaires. Responses can be downloaded as csv (comma-separated values) file. Sample data was created using following questionnaire:

Responses are available in a file `GoogleForms_AVED.csv`. Load it and make it tidy!

Solution A

```r
library(readr)
library(stringr)

# readr::read_csv does not convert characters into factors (as ordinary read.csv does)
gdata <- read_csv("data/GoogleForms_AVED.csv")
```

Google forms returns really ugly names:

```r
names(gdata)
```

```
## [1] "Timestamp"
## [2] "Choose one of following (radiobutton):"
## [3] "Choose multiple answers (checkbox):"
## [4] "Radiobutton grid: [Row 1]"
## [5] "Radiobutton grid: [Row 2]"
## [6] "Radiobutton grid: [Row 3]"
```

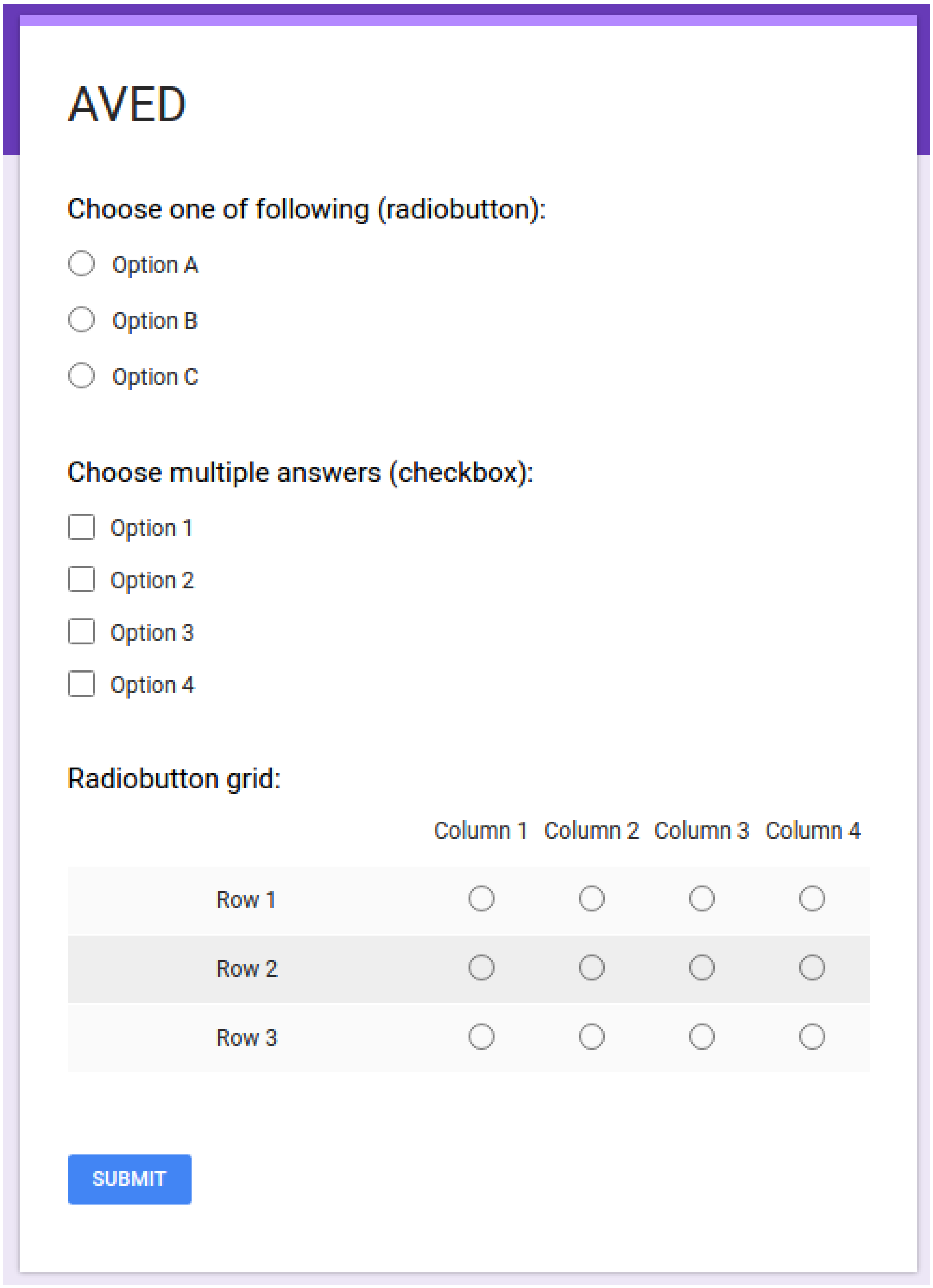
The first task is to rename columns:

```r
gdata %<>% set_names(c("time","Q1","Q2","Q3a","Q3b","Q3c"))
```

Now we can take a look at the data

```r
print(gdata)
```

```
## # A tibble: 5 × 6
##                         time       Q1                          Q2      Q3a
##                        <chr>    <chr>                       <chr>    <chr>
## 1 2016/07/04 8:58:36 am EET Option A          Option 1;Option 3 Column 2
```



AVED

Choose one of following (radiobutton):

- ◯ Option A
- ◯ Option B
- ◯ Option C

Choose multiple answers (checkbox):

- ☐ Option 1
- ☐ Option 2
- ☐ Option 3
- ☐ Option 4

Radiobutton grid:

	Column 1	Column 2	Column 3	Column 4
Row 1	◯	◯	◯	◯
Row 2	◯	◯	◯	◯
Row 3	◯	◯	◯	◯

Figure 1:

```
## 2 2016/07/04 8:58:56 am EET Option C                   Option 3 Column 4
## 3 2016/07/04 8:59:11 am EET Option B                   Option 2     <NA>
## 4 2016/07/04 8:59:35 am EET Option A          Option 1;Option 2 Column 2
## 5 2016/07/04 9:00:58 am EET Option C Option 1;Option 2;Option 3 Column 2
## # ... with 2 more variables: Q3b <chr>, Q3c <chr>
```

Google stores answers as (complete) strings – in the case of multiple-choice questions (Q2) are answers chosen separated by ";"

To get the data in useable tidy form we need to separate answers in Q2 into multiple columns. Each possible answer would get its own column with logical values according to anwers of the respondent.

The most straightforward solution is a combination of `separate_rows()`, `mutate()`, and `spread()`:

```r
gdata %>%
  # separate answers stored in Q2 into multiple rows
  separate_rows(Q2, sep=";") %>%
  # add new auxiliary column "aux" which is equal to TRUE for every row (mutate is a dplyr function)
  mutate(aux = TRUE) %>%
  # spread values from auxiliary variable using Q2 as a key
  spread(Q2, aux, fill = FALSE)
```

```
## # A tibble: 5 × 8
##                          time        Q1       Q3a       Q3b       Q3c `Option 1`
## *                       <chr>     <chr>     <chr>     <chr>     <chr>      <lgl>
## 1 2016/07/04 8:58:36 am EET Option A Column 2 Column 1 Column 4       TRUE
## 2 2016/07/04 8:58:56 am EET Option C Column 4 Column 3 Column 2      FALSE
## 3 2016/07/04 8:59:11 am EET Option B     <NA> Column 3 Column 2      FALSE
## 4 2016/07/04 8:59:35 am EET Option A Column 2 Column 3 Column 1       TRUE
## 5 2016/07/04 9:00:58 am EET Option C Column 2 Column 3     <NA>       TRUE
## # ... with 2 more variables: `Option 2` <lgl>, `Option 3` <lgl>
```

Solution B

However, solution A is far from ideal one:

- Names of newly created columns does not make much sense.
- Column for "Option 4" is missing (because nobody chose it – but we don't want to lose this information)

First step is the same as in the case of the previous solution – separate Q2 answers.

```r
gdata %>% separate_rows(Q2, sep=";") -> gdata_aux
```

In the second step the Q2 is converted into factor. It allows us to add missing level "Option 4" and create code (factor label) for each answer.

```r
Q2_options <- c(
  "opt1" = "Option 1",
  "opt2" = "Option 2",
  "opt3" = "Option 3",
  "opt4" = "Option 4"
)

gdata_aux$Q2 %<>% factor(., levels = Q2_options, labels = names(Q2_options))
```

```
gdata_aux %>%
  mutate(aux = TRUE) %>%
  # select() is a function from dplyr which selects variables
  select(time,Q2,aux) %>%
  # Here I use previously unused argument of spread "sep". If the argument is set then
  # newly created columns are named <key_name><sep><key_value>
  spread(key = Q2, value = aux, fill = FALSE, drop = FALSE, sep="_") %>%
  # Resulting data.frame is subsequently joined with original one
  left_join(gdata,.)
```

```
## Joining, by = "time"
```

```
## # A tibble: 5 × 10
##                         time       Q1                           Q2      Q3a
##                        <chr>    <chr>                        <chr>    <chr>
## 1 2016/07/04 8:58:36 am EET Option A          Option 1;Option 3 Column 2
## 2 2016/07/04 8:58:56 am EET Option C                   Option 3 Column 4
## 3 2016/07/04 8:59:11 am EET Option B                   Option 2     <NA>
## 4 2016/07/04 8:59:35 am EET Option A          Option 1;Option 2 Column 2
## 5 2016/07/04 9:00:58 am EET Option C Option 1;Option 2;Option 3 Column 2
## # ... with 6 more variables: Q3b <chr>, Q3c <chr>, Q2_opt1 <lgl>,
## #   Q2_opt2 <lgl>, Q2_opt3 <lgl>, Q2_opt4 <lgl>
```

Index of African Governance

The first messy data comes from data set Index of African Governance available at http://www.nber.org/data/iag.html. Index of African Governance is a project of Harvard University's Kennedy School of Government's Program on Intrastate Conflict and Conflict Resolution and of the World Peace Foundation under the direction of Robert I. Rotberg and Rachel M. Gisselquist. Data for your homework are available in `HW_HumanDevelopment_Africa.tsv` and it is real life messy mess!

Tables in the same structure are provided e.g. by Czech Statistical Office.

Solution:

```
# Read data:
read.table("data/HumanDevelopment_Africa.tsv",
           header = FALSE,
           colClasses = "character",
           sep="\t",
           quote = "",
           na.strings = "") %>%
  as_data_frame() -> africa

print(africa)
```

```
## # A tibble: 55 × 111
##                          V1             V2    V3    V4    V5    V6
##                       <chr>          <chr> <chr> <chr> <chr> <chr>
## 1                 Indicator poverty_npline  <NA>  <NA>  <NA>  <NA>
## 2                      Year           2000  2002  2005  2006  2007
## 3                    Angola           68.0  68.0  68.0  68.0  68.0
## 4                     Benin           36.8  36.8  36.8  36.8  36.8
## 5                  Botswana           30.3  30.3  30.3  30.3  30.3
## 6              Burkina Faso           54.6  46.4  46.4  46.4  46.4
## 7                   Burundi           68.0  68.0  36.2  36.2  36.2
## 8                  Cameroon           40.2  40.2  39.9  39.9  39.9
## 9                Cape Verde           36.7  36.7  36.7  36.7  36.7
## 10 Central African Republic           67.2  67.2  67.2  67.2  67.2
## # ... with 45 more rows, and 105 more variables: V7 <chr>, V8 <chr>,
## #   V9 <chr>, V10 <chr>, V11 <chr>, V12 <chr>, V13 <chr>, V14 <chr>,
## #   V15 <chr>, V16 <chr>, V17 <chr>, V18 <chr>, V19 <chr>, V20 <chr>,
## #   V21 <chr>, V22 <chr>, V23 <chr>, V24 <chr>, V25 <chr>, V26 <chr>,
## #   V27 <chr>, V28 <chr>, V29 <chr>, V30 <chr>, V31 <chr>, V32 <chr>,
## #   V33 <chr>, V34 <chr>, V35 <chr>, V36 <chr>, V37 <chr>, V38 <chr>,
## #   V39 <chr>, V40 <chr>, V41 <chr>, V42 <chr>, V43 <chr>, V44 <chr>,
## #   V45 <chr>, V46 <chr>, V47 <chr>, V48 <chr>, V49 <chr>, V50 <chr>,
## #   V51 <chr>, V52 <chr>, V53 <chr>, V54 <chr>, V55 <chr>, V56 <chr>,
## #   V57 <chr>, V58 <chr>, V59 <chr>, V60 <chr>, V61 <chr>, V62 <chr>,
## #   V63 <chr>, V64 <chr>, V65 <chr>, V66 <chr>, V67 <chr>, V68 <chr>,
## #   V69 <chr>, V70 <chr>, V71 <chr>, V72 <chr>, V73 <chr>, V74 <chr>,
## #   V75 <chr>, V76 <chr>, V77 <chr>, V78 <chr>, V79 <chr>, V80 <chr>,
## #   V81 <chr>, V82 <chr>, V83 <chr>, V84 <chr>, V85 <chr>, V86 <chr>,
## #   V87 <chr>, V88 <chr>, V89 <chr>, V90 <chr>, V91 <chr>, V92 <chr>,
## #   V93 <chr>, V94 <chr>, V95 <chr>, V96 <chr>, V97 <chr>, V98 <chr>,
## #   V99 <chr>, V100 <chr>, V101 <chr>, V102 <chr>, V103 <chr>, V104 <chr>,
## #   V105 <chr>, V106 <chr>, ...
```

This is a real mess. You can find tables like that for example in historical statistics of Czech Statistical Office. There are no actual column names. First two rows define content of the column – i.e. a combination of indicator and year. Moreover indicator is not filled in in all cases – it is just in the forst column and its value applies for all subsequent columns with `NA`.

The first step would be to transpose the table:

```
africa %<>%
  gather(variable,value,-V1) %>%
  spread(V1,value)
```

See come columns of transposed table:

```
africa %>% select(variable,Indicator,Year,Angola,Benin) %>% print()
```

```
## # A tibble: 110 × 5
##    variable Indicator  Year Angola Benin
## *     <chr>     <chr> <chr>  <chr> <chr>
## 1       V10      <NA>  2006   54.3  47.3
## 2      V100      <NA>  2006   <NA>  71.3
## 3      V101      <NA>  2007   <NA>  71.3
## 4      V102  bg_ratio  2000   82.1  64.2
## 5      V103      <NA>  2002   82.1  67.0
## 6      V104      <NA>  2005   82.1  73.5
## 7      V105      <NA>  2006   82.1  73.5
## 8      V106      <NA>  2007   82.1  73.5
## 9      V107  pt_ratio  2000   41.8  52.6
## 10     V108      <NA>  2002   41.8  53.0
## # ... with 100 more rows
```

Rows in resulting data frame are not in the same order as columns were. But we can fix it. We will extract numeric values from `variable` and use them for rows ordering:

```
# For parsing numeric values we will use readr::parse_number(). You can use regular expressions as well
africa$variable %<>% readr::parse_number()
# arrange() from dplyr will order data frame by column variable (i.e. original column number)
africa %<>% arrange(variable)
```

See come columns…

```
africa %>% select(variable,Indicator,Year,Angola,Benin) %>% print()
```

```
## # A tibble: 110 × 5
##    variable        Indicator  Year Angola Benin
##       <dbl>            <chr> <chr>  <chr> <chr>
## 1         2   poverty_npline  2000   68.0  36.8
## 2         3             <NA>  2002   68.0  36.8
## 3         4             <NA>  2005   68.0  36.8
## 4         5             <NA>  2006   68.0  36.8
## 5         6             <NA>  2007   68.0  36.8
## 6         7 poverty_1.25aday  2000   54.3  47.3
## 7         8             <NA>  2002   54.3  47.3
## 8         9             <NA>  2005   54.3  47.3
## 9        10             <NA>  2006   54.3  47.3
## 10       11             <NA>  2007   54.3  47.3
## # ... with 100 more rows
```

Now we can fill missing indicator names using `fill()`

```
africa %<>%
  fill(Indicator) %>%
  gather(country,value,-Indicator,-Year,-variable) %>%
  # We don't have any use for column variable. We can drop it off using select() from dplyr.
  select(-variable)

print(africa)
```

```
## # A tibble: 5,830 × 4
##           Indicator  Year country value
##               <chr> <chr>   <chr> <chr>
## 1    poverty_npline  2000 Algeria  15.0
## 2    poverty_npline  2002 Algeria  15.0
## 3    poverty_npline  2005 Algeria  15.0
## 4    poverty_npline  2006 Algeria  15.0
## 5    poverty_npline  2007 Algeria  15.0
## 6  poverty_1.25aday  2000 Algeria   6.8
## 7  poverty_1.25aday  2002 Algeria   6.8
## 8  poverty_1.25aday  2005 Algeria   6.8
## 9  poverty_1.25aday  2006 Algeria   6.8
## 10 poverty_1.25aday  2007 Algeria   6.8
## # ... with 5,820 more rows
```

Columns Year and value are characters – but they by converted to numeric easily. Then we could spread the data frame into tidy form

```
africa$Year %<>% readr::parse_number()
africa$value %<>% readr::parse_number()

africa %>% spread(Indicator,value)
```

```
## # A tibble: 265 × 24
##     Year                  country bg_ratio drinking_water  gini
## *  <dbl>                    <chr>    <dbl>          <dbl> <dbl>
## 1   2000                  Algeria     97.6             89  35.3
## 2   2000                   Angola     82.1             44  58.6
## 3   2000                    Benin     64.2             64  38.6
## 4   2000                 Botswana    101.6             95  61.0
## 5   2000             Burkina Faso     70.0             56  46.9
## 6   2000                  Burundi     79.9             71  42.4
## 7   2000                 Cameroon     82.4             63  44.6
## 8   2000               Cape Verde     99.0             80  50.5
## 9   2000 Central African Republic       NA             63  43.6
## 10  2000                  Comoros     84.1             88  64.3
## # ... with 255 more rows, and 19 more variables: HIV_prevalence <dbl>,
## #   chmortality <dbl>, immunization_DPT <dbl>, immunization_measles <dbl>,
## #   le <dbl>, literacy_rate <dbl>, literacy_rate_female <dbl>, mmr <dbl>,
## #   Nursing_Personnel <dbl>, Physicians <dbl>, poverty_npline <dbl>,
## #   poverty_1.25aday <dbl>, primary_school <dbl>,
## #   primary_school_female <dbl>, pt_ratio <dbl>,
## #   Sanitation_Facilities <dbl>, sec_school <dbl>, TBC <dbl>,
## #   undernourishment <dbl>
```

Homework

The task is to load a table from Eurostat and make it (almost) tidy. Table `lfsa_urgacob` contains unemployment rates (%) of immigrants by sex, age and country of birth.

The table is in the file `lfsa_urgacob.tsv` (tab-separated table):

```
## # A tibble: 33,751 × 22
##    `unit,sex,age,c_birth,geo\\time` `2015` `2014` `2013` `2012` `2011`
##                               <chr>  <chr>  <chr>  <chr>  <chr>  <chr>
## 1           PC,F,Y15-19,EU15_FOR,AT    : u    : u    : u    : u    : u
## 2           PC,F,Y15-19,EU15_FOR,BE      :    : u    : u    : u   : bu
## 3           PC,F,Y15-19,EU15_FOR,CH    : u    : u    : u    : u    : u
## 4           PC,F,Y15-19,EU15_FOR,CY    : u    : u    : u      :    : u
## 5           PC,F,Y15-19,EU15_FOR,CZ      :      :      :      :    : b
## 6           PC,F,Y15-19,EU15_FOR,DK    : u    : u    : u    : u    : u
## 7         PC,F,Y15-19,EU15_FOR,EA17    : u    : u    : u    : u    : u
## 8         PC,F,Y15-19,EU15_FOR,EA18    : u    : u    : u    : u    : u
## 9         PC,F,Y15-19,EU15_FOR,EA19    : u    : u    : u    : u    : u
## 10           PC,F,Y15-19,EU15_FOR,EE      :      :      :      :      :
## # ... with 33,741 more rows, and 16 more variables: `2010` <chr>,
## #   `2009` <chr>, `2008` <chr>, `2007` <chr>, `2006` <chr>, `2005` <chr>,
## #   `2004` <chr>, `2003` <chr>, `2002` <chr>, `2001` <chr>, `2000` <chr>,
## #   `1999` <chr>, `1998` <chr>, `1997` <chr>, `1996` <chr>, `1995` <chr>
```

- `:` stays for missing observations
- in records in the form `41.6 u` or `: bu` letters denote notes

You are supposed to drop all notes and transform the table into following format:

```
## # A tibble: 24,003 × 35
##     unit   sex  c_birth   geo  year Y15_19 Y15_24 Y15_39 Y15_59 Y15_64
## *  <chr> <chr>    <chr> <chr> <int>  <dbl>  <dbl>  <dbl>  <dbl>  <dbl>
## 1     PC     F EU15_FOR    AT  1995     NA     NA     NA     NA     NA
## 2     PC     F EU15_FOR    AT  1996     NA     NA     NA     NA     NA
## 3     PC     F EU15_FOR    AT  1997     NA     NA     NA     NA     NA
## 4     PC     F EU15_FOR    AT  1998     NA     NA     NA     NA     NA
## 5     PC     F EU15_FOR    AT  1999     NA     NA     NA     NA     NA
## 6     PC     F EU15_FOR    AT  2000     NA     NA     NA     NA     NA
## 7     PC     F EU15_FOR    AT  2001     NA     NA     NA     NA     NA
## 8     PC     F EU15_FOR    AT  2002     NA     NA     NA     NA     NA
## 9     PC     F EU15_FOR    AT  2003     NA     NA     NA     NA     NA
## 10    PC     F EU15_FOR    AT  2004     NA     NA     NA     NA     NA
## # ... with 23,993 more rows, and 25 more variables: Y15_74 <dbl>,
## #   Y20_24 <dbl>, Y20_64 <dbl>, Y25_29 <dbl>, Y25_49 <dbl>, Y25_54 <dbl>,
## #   Y25_59 <dbl>, Y25_64 <dbl>, Y25_74 <dbl>, Y30_34 <dbl>, Y35_39 <dbl>,
## #   Y40_44 <dbl>, Y40_59 <dbl>, Y40_64 <dbl>, Y45_49 <dbl>, Y50_54 <dbl>,
## #   Y50_59 <dbl>, Y50_64 <dbl>, Y50_74 <dbl>, Y55_59 <dbl>, Y55_64 <dbl>,
## #   Y60_64 <dbl>, Y65_69 <dbl>, Y65_74 <dbl>, Y70_74 <dbl>
```

Assign transformed table into variable `eudata`. You can find example of the outcome table in `lfsa_urgacob_solution.Rdata`.

Hints:

- Pay attention to column names and classes!
- You may find helpful to use some regular expressions
- Age groups contains "-" in original table and "_" in the transformed one