Data wrangling: Lecture notes

Claudius Gräbner-Radkowitsch

2022-04-20

Contents

1 Packages and data used

```
library(dplyr)
library(tidyr)
library(data.table)
library(here)
```

The data sets used in these notes are available from the course webpage:

- `wrangling_slides.csv` (`data_raw`)

- `wrangling_slides.csv` (`data_raw_long`)
- `wrangling_slides_gini.csv` (`gini_red`)
- `wrangling_slides_final_expl.csv` (`data_final_expl`)
- `wrangling_slides_gini_grc.csv` (`swiid_join`)

The brackets show the names of the data sets used below

2 General remarks

- If you imported the data from a file, make sure that the import went as expected
- Start with a data set that is of the type `tibble` (use `tibble::as_tibble()` if necessary)
- Before starting to wrangle, make a note to yourself of how the final data set should look like;
  - Then think about the different steps you need to take to reach this goal;
  - Each step should only address one single wrangling challenge
- It is often useful to save the wrangling code in one script in which you import raw data in the beginning and save tidy data in the end
- Then you keep data wrangling, visualization, and modlling in separate files

3 Reshaping data from long to wide format

3.1 Wide and long format: definition

There is no strict definition for wide and long data. Rather, the two should be understood as *relative* descriptions of data, meaning that it is more straightforward to speak of a data set that is long*er* relative to another one, rather than *a long* data set per se.

Here is an example for a rather long data set:

```
##     country year variable    value
##  1: Germany 2017    unemp     3.75
##  2: Germany 2017      gdp 53071.46
##  3: Germany 2017     gini    29.40
##  4: Germany 2018    unemp     3.38
##  5: Germany 2018      gdp 53486.84
##  6: Germany 2018     gini    29.60
##  7:  Greece 2017    unemp    21.49
##  8:  Greece 2017      gdp 28604.86
##  9:  Greece 2017     gini    32.20
## 10:  Greece 2018    unemp    19.29
## 11:  Greece 2018      gdp 29141.17
## 12:  Greece 2018     gini    31.70
```

Here, we have one column identifying the variable, the value of which is stored in a separate column. This means that the data is relatively 'long' in the sense of having many rows. At the same time, it is relatively 'narrow' in the sense of not having too many columns since the variable identifier is kept in a single column.

Contrast this with an example for a rather wide data set, where each variable has its own column:

```
##    country year unemp      gdp gini
## 1: Germany 2017  3.75 53071.46 29.4
## 2: Germany 2018  3.38 53486.84 29.6
## 3:  Greece 2017 21.49 28604.86 32.2
## 4:  Greece 2018 19.29 29141.17 31.7
```

Here, we have more columns since the three variables, the unemployment rate and GDP, have their own columns. In effect, the data set has much more columns, but tends to be shorter in the sense of having fewer rows.

While the long format is often easier to read and preferable when communicating data to humans, making data tidy often involves the task of making data 'longer'.

3.2 Transforming long data into wide data

To make data wider we use the function `tidyr::pivor_wider()`.

Assume that we start with our long data set introduced above and that this data set is bound to the name `data_raw_long`.

```
dplyr::glimpse(data_raw_long)
```

```
## Rows: 12
## Columns: 4
## $ country  <chr> "Germany", "Germany", "Germany", "Germany", "Germany", "Germa~
## $ year     <int> 2017, 2017, 2017, 2018, 2018, 2018, 2017, 2017, 2017, 2018, 2~
## $ variable <chr> "unemp", "gdp", "gini", "unemp", "gdp", "gini", "unemp", "gdp~
## $ value    <dbl> 3.75, 53071.46, 29.40, 3.38, 53486.84, 29.60, 21.49, 28604.86~
```

We will now use `tidyr::pivor_wider()` to make this data set wider. The most important arguments of this function are as follows:[1]

- `data` is the first argument and refers to the name of the data set to be considered
- `names_from` denotes the column that includes the names of the new columns
- `values_from` denotes the column that includes the values to be allocated in the newly created cells

[1] The function allows for much more finetuning. You might read more about its argument in the help page of the function or the online documentation.

In the present case, the call would look like the following:

```
data_raw_wide <- tidyr::pivot_wider(
  data = data_raw_long,
  names_from = "variable",
  values_from = "value")
data_raw_wide
```

```
## # A tibble: 4 x 5
##   country  year unemp    gdp  gini
##   <chr>   <int> <dbl>  <dbl> <dbl>
## 1 Germany  2017  3.75 53071.  29.4
## 2 Germany  2018  3.38 53487.  29.6
## 3 Greece   2017 21.5  28605.  32.2
## 4 Greece   2018 19.3  29141.  31.7
```

3.3 Transforming wide data into long data

Assume we want to take the data set `data_raw_wide` and re-create the original long version. To achieve this we can use `tidyr::pivot_longer()`. Again, lets have a look at the most important arguments:[2]

- `data` is the first argument and refers to the name of the data set to be considered
- `cols` denotes the columns that should be transformed into the longer format
- `names_to` denotes the column that includes the names of the new columns
- `values_to` denotes the column that includes the values to be allocated in the newly created cells

The arguments `names_to` and `values_to` are not strictly necessary since they have useful default values, but its usually nicer to be explicit.

When specifying the argument `cols` you have several possibilities. The safest variant is to use the function `dplyr::all_of()` and pass a character vector with the column names. You can save a lot of writing by using so called selection helpers, a very useful tool we will learn about later.

In our case this amounts to:

```
data_raw_long <- tidyr::pivot_longer(
  data = data_raw_wide,
  cols = dplyr::all_of(c("unemp", "gdp", "gini")),
  names_to = "indicator",
  values_to = "values")
data_raw_long
```

[2] See the online documentation for a more complete description.

```
## # A tibble: 12 x 4
##    country  year indicator   values
##    <chr>   <int> <chr>        <dbl>
##  1 Germany  2017 unemp         3.75
##  2 Germany  2017 gdp       53071.
##  3 Germany  2017 gini         29.4
##  4 Germany  2018 unemp         3.38
##  5 Germany  2018 gdp       53487.
##  6 Germany  2018 gini         29.6
##  7 Greece   2017 unemp        21.5
##  8 Greece   2017 gdp       28605.
##  9 Greece   2017 gini         32.2
## 10 Greece   2018 unemp        19.3
## 11 Greece   2018 gdp       29141.
## 12 Greece   2018 gini         31.7
```

4 Chaining wrangling tasks using pipes

Pipes are provided via the package `magrittr`, which is loaded automatically if you attach packages such as `tidyr` or `dplyr`. Pipes are short keywords that facilitate the development of very readable and transparent data wrangling code.

While there are many different pipes, the one we will use extensively is `%>%`. It is always used at the end of a line, and it basically 'throws' the result of this line into the next line of code. In this line, you can refer to the intermediate result via `.`, or it is used implicitly as the first argument to the function you use.

In other words, `x %>% f(y)` (or `x %>% f(., y)`) is equivalent to `f(x, y)`.

But lets look at an example! Assume we start with this data set:

```
##    country year      gdp unemp
## 1: Germany 2017 53071.46  3.75
## 2: Germany 2018 53486.84  3.38
## 3:  Greece 2017 28604.86 21.49
## 4:  Greece 2018 29141.17 19.29
```

And what we want is this:

```
## # A tibble: 4 x 4
##   country name    `2017`   `2018`
##   <chr>   <chr>    <dbl>    <dbl>
## 1 Germany gdp   53071.   53487.
## 2 Germany unemp     3.75     3.38
## 3 Greece  gdp   28605.   29141.
## 4 Greece  unemp    21.5     19.3
```

We can do this by first making the data longer, and then wider. We could do this explicitly:

```
pipe_data_1 <- pivot_longer(
  data = pipe_data_raw,
  cols = all_of(c("gdp", "unemp")))

pipe_data_2 <- pivot_wider(
  data = pipe_data_1,
  names_from = "year",
  values_from = "value")
```

But we can also write this code more concisely using the pipe:

```
pipe_data_final <- pivot_longer(
  data = pipe_data_raw,
  cols = all_of(c("gdp", "unemp"))) %>%
  pivot_wider(
  data = .,
  names_from = "year",
  values_from = "value")
```

Or, since the pipe carries the intermediate result implicitly as the first argument to the function on the next line we can space the `data = .`:

```
pipe_data_final <- pivot_longer(
  data = pipe_data_raw,
  cols = all_of(c("gdp", "unemp"))) %>%
  pivot_wider(
  names_from = "year",
  values_from = "value")
```

The `%>%`-pipe allows you to write very readable code, so make sure you use it often. But for code development it might be nevertheless helpful to write the intermediate steps explicitly.

5 Creating of manipulating variables

The function `dplyr::mutate()` is used both for manipulating existing columns as well as creating new columns. In the first case the name of the column that the result of `dplyr::mutate()` is written into already exists, in the second case we just use a new name.

Consider the following data set with the unemployment rate as an example:

```
data_unemp
```

```
## # A tibble: 2 x 3
##    year Germany Greece
##   <int>   <dbl>  <dbl>
## 1  2017    3.75   21.5
## 2  2018    3.38   19.3
```

Assume we want to express the percentage values via decimal numbers and, to this end, divide the values in the columns `Germany` and `Greece` by 100. We can use `dplyr::mutate()` to achieve this:

```
data_unemp %>%
  dplyr::mutate(
    Germany = Germany/100,
    Greece = Greece/100
  )
```

```
## # A tibble: 2 x 3
##    year Germany Greece
##   <int>   <dbl>  <dbl>
## 1  2017  0.0375  0.215
## 2  2018  0.0338  0.193
```

But we could use basically the same code to create a new column. Assume, for instance, we want a new column containing the difference between the unemployment rates:

```
data_unemp %>%
  dplyr::mutate(
    Difference = Greece - Germany
  )
```

```
## # A tibble: 2 x 4
##    year Germany Greece Difference
##   <int>   <dbl>  <dbl>      <dbl>
## 1  2017    3.75   21.5       17.7
## 2  2018    3.38   19.3       15.9
```

The only difference here was that the left-hand-side name of the column to be manipulated did not exist before!

6 Filtering rows

The function `dplyr::filter()` can be used to filter rows according to certain conditions. The conditions must evaluate for each cell entry to either `TRUE` or `FALSE`, and only those rows for which they evaluate to `TRUE` remain in the data set. Often, the conditions are specified via logical operators, which were already covered in the tutorial on vector types.

As always, the first argument to `dplyr::filter()` is `data`, i.e. the data set on which you want to operate. Then follow an arbitrary number of logical conditions on the different columns of the data set on question.

Assume we want to take the previously defined data set `data_raw_long`

```
data_raw_long
```

```
## # A tibble: 12 x 4
##    country  year indicator   values
##    <chr>   <int> <chr>        <dbl>
##  1 Germany  2017 unemp         3.75
##  2 Germany  2017 gdp       53071.
##  3 Germany  2017 gini         29.4
##  4 Germany  2018 unemp         3.38
##  5 Germany  2018 gdp       53487.
##  6 Germany  2018 gini         29.6
##  7 Greece   2017 unemp        21.5
##  8 Greece   2017 gdp       28605.
##  9 Greece   2017 gini         32.2
## 10 Greece   2018 unemp        19.3
## 11 Greece   2018 gdp       29141.
## 12 Greece   2018 gini         31.7
```

and only want to keep data on GDP:

```
data_raw_long %>%
  dplyr::filter(indicator=="gdp")
```

```
## # A tibble: 4 x 4
##   country  year indicator values
##   <chr>   <int> <chr>      <dbl>
## 1 Germany  2017 gdp       53071.
## 2 Germany  2018 gdp       53487.
## 3 Greece   2017 gdp       28605.
## 4 Greece   2018 gdp       29141.
```

You may also combine more than one condition in one call to `dplyr::filter()`. If you also want to filter by values and only keep those rows where the value is below 50.000:

```
data_raw_long %>%
  dplyr::filter(
    indicator=="gdp",
    values < 50000)
```

```
## # A tibble: 2 x 4
##   country  year indicator values
##   <chr>   <int> <chr>      <dbl>
## 1 Greece   2017 gdp       28605.
## 2 Greece   2018 gdp       29141.
```

7 Selecting columns

When you only want to keep certain *columns* we speak of selecting (rather than filtering) columns. This is done - surprise - via the function ´dplyr::select()‘.

There are different ways for selecting columns. In any case, the first argument is, again, `data`, i.e. the data set considered. In the present case, we will refer to `data_raw`:

```
data_raw
```

```
##    country year unemp      gdp gini
## 1: Germany 2017  3.75 53071.46 29.4
## 2: Germany 2018  3.38 53486.84 29.6
## 3:  Greece 2017 21.49 28604.86 32.2
## 4:  Greece 2018 19.29 29141.17 31.7
```

Then we can now select columns using one of the following two options. First, you may refer to columns via their *name*:

```
data_raw %>%
  dplyr::select(country, year, unemp)
```

```
##    country year unemp
## 1: Germany 2017  3.75
## 2: Germany 2018  3.38
## 3:  Greece 2017 21.49
## 4:  Greece 2018 19.29
```

But this is often error-prone. Thus, it is usually better to refer to the columns via selection helpers, which is also the most flexible version. While we will learn about more selection helpers later, here we will only use `dplyr::all_of()`, which accepts a character vector of column names:

```
data_raw %>%
  dplyr::select(dplyr::all_of(c("country", "year", "gini")))
```

```
##    country year gini
## 1: Germany 2017 29.4
## 2: Germany 2018 29.6
## 3:  Greece 2017 32.2
## 4:  Greece 2018 31.7
```

Caution: Do not forget the `c()`! Otherwise:

```
data_raw %>%
  dplyr::select(dplyr::all_of("country", "year", "gini"))
```

```
## Error in `dplyr::select()`:
## ! unused arguments ("year", "gini")
```

It is also possible to define the column vector first:

```
cols2keep <- c("country", "year", "gini")
data_raw %>%
  dplyr::select(dplyr::all_of(cols2keep))
```

```
##    country year gini
## 1: Germany 2017 29.4
## 2: Germany 2018 29.6
## 3:  Greece 2017 32.2
## 4:  Greece 2018 31.7
```

In any case, you can also specify the columns you want to *drop*. To this end, just add a `-` in front of the selection command:

```
data_raw %>%
  dplyr::select(-unemp, -gdp)
```

```
##    country year gini
## 1: Germany 2017 29.4
## 2: Germany 2018 29.6
## 3:  Greece 2017 32.2
## 4:  Greece 2018 31.7
```

8 Merging data sets

Often you need to obtain data from different sources. To merge all your data in one single data set, you need to use one of the `*_join()` functions of the `dplyr`-package. These functions all merge two data sets, but the way they do it is different. Below we illustrate the most common joins (so called mutating joins).[3]

As a guiding example we use the following two data sets:

First, data on income inequality from the SWIID data base:

```
swiid_join
```

```
##    country year gini
## 1:  Greece 2015 33.1
## 2:  Greece 2017 32.2
```

Second, data on GDP per capita from the World Bank:

```
gdp_join
```

```
##    country year      gdp
## 1: Germany 2017 53071.46
## 2: Germany 2018 53486.84
## 3:  Greece 2017 28604.86
## 4:  Greece 2018 29141.17
```

We will consider the behavior of the following four functions:

- `dplyr::left_join()`
- `dplyr::right_join()`
- `dplyr::full_join()`
- `dplyr::inner_join()`

All of them accept the following arguments:

- `x` and `y`: the two data sets to be merged
- `by`: a vector or a named vector indicating on which columns the data sets should be merged

Its easier to understand their behavior if you contrast them directly with each other. First, `dplyr::left_join()` joins the data sets on those columns mentioned in `by`, but only keeps those rows for which `x` contains an observation:

[3] The other join types are filtering joins and nest joins. You find more information in the web, and more details on the underlying theory in chapter 13 of R4DS.

```
dplyr::left_join(x = gdp_join, y = swiid_join, by = c("country", "year"))
```

```
##    country year      gdp gini
## 1: Germany 2017 53071.46   NA
## 2: Germany 2018 53486.84   NA
## 3:  Greece 2017 28604.86 32.2
## 4:  Greece 2018 29141.17   NA
```

This might introduce `NA`s into the columns of `y`, but not of `x`. It is the other way around for `dplyr::right_join()`: it only keeps those rows for which `y` contains an observation:

```
dplyr::right_join(x = gdp_join, y = swiid_join, by = c("country", "year"))
```

```
##    country year      gdp gini
## 1:  Greece 2017 28604.86 32.2
## 2:  Greece 2015       NA 33.1
```

`dplyr::inner_join()` is the most restrictive option, keeping only those rows for which both `x` and `y` contain an observation (i.e. it never introduces `NA`s):

```
dplyr::inner_join(x = gdp_join, y = swiid_join, by = c("country", "year"))
```

```
##    country year      gdp gini
## 1:  Greece 2017 28604.86 32.2
```

Finally, `dplyr::full_join()` contains all rows that occur at least in `x` *or* `y`, i.e. it might introduce `NA`s in both the columns of `x` and `y`:

```
dplyr::full_join(x = gdp_join, y = swiid_join, by = c("country", "year"))
```

```
##    country year      gdp gini
## 1: Germany 2017 53071.46   NA
## 2: Germany 2018 53486.84   NA
## 3:  Greece 2017 28604.86 32.2
## 4:  Greece 2018 29141.17   NA
## 5:  Greece 2015       NA 33.1
```

Two final remarks: first, the types of the columns on which you merge the data sets must be equal, otherwise R throws an error:

```
swiid_join <- dplyr::mutate(swiid_join, year=as.character(year))
dplyr::left_join(x = gdp_join, y = swiid_join, by = c("country", "year"))
```

```
## Error in `dplyr::left_join()`:
## ! Can't join on `x$year` x `y$year` because of incompatible types.
## i `x$year` is of type <integer>>.
## i `y$year` is of type <character>>.
```

Just enforce the correct data type before merging:

```
swiid_join %>%
  dplyr::mutate(year=as.integer(year)) %>%
  dplyr::left_join(x = gdp_join, y = ., by = c("country", "year"))
```

```
##    country year      gdp gini
## 1: Germany 2017 53071.46   NA
## 2: Germany 2018 53486.84   NA
## 3:  Greece 2017 28604.86 32.2
## 4:  Greece 2018 29141.17   NA
```

Second, you can also merge on columns with different names by passing named vectors to `by`:

```
swiid_join <- swiid_join %>%
  mutate(Year=as.double(year)) %>%
  select(-year)
swiid_join
```

```
##    country gini Year
## 1:  Greece 33.1 2015
## 2:  Greece 32.2 2017
```

Then this does not work any more:

```
dplyr::left_join(
  x = gdp_join, y = swiid_join,
  by = c("country", "year"))
```

```
## Error in `dplyr::left_join()`:
## ! Join columns must be present in data.
## x Problem with `year`.
```

But the named vector fixes it:

```
dplyr::left_join(
  x = gdp_join, y = swiid_join,
  by = c("country", "year"="Year"))
```

```
##    country year      gdp gini
## 1: Germany 2017 53071.46   NA
## 2: Germany 2018 53486.84   NA
## 3:  Greece 2017 28604.86 32.2
## 4:  Greece 2018 29141.17   NA
```

9 Grouping and summarising data

The final challenge we consider involves the application of two functions (at least in most cases): `dplyr::group_by()` and `dplyr::summarize()`.

`dplyr::group_by()` is usually used within pipes and groups a data set according to an arbitrary number of variables, each of which must refer to one (and only one) column. It produces a grouped data set:

```
data_raw_grouped <- data_raw %>%
  dplyr::group_by(country)
data_raw_grouped
```

```
## # A tibble: 4 x 5
## # Groups:   country [2]
##   country  year unemp    gdp  gini
##   <chr>   <int> <dbl>  <dbl> <dbl>
## 1 Germany  2017  3.75 53071.  29.4
## 2 Germany  2018  3.38 53487.  29.6
## 3 Greece   2017 21.5  28605.  32.2
## 4 Greece   2018 19.3  29141.  31.7
```

As you can see, the data set is now grouped by the variable `country`. We can specify the grouping variables the same way we selected columns in the context of `dplyr::select()` (see above).

Grouped data sets are usually not interesting in itself. You can ungroup them via `dplyr::ungroup()`:

```
data_raw_grouped %>%
  dplyr::ungroup()
```

```
## # A tibble: 4 x 5
##   country  year unemp    gdp  gini
##   <chr>   <int> <dbl>  <dbl> <dbl>
## 1 Germany  2017  3.75 53071.  29.4
## 2 Germany  2018  3.38 53487.  29.6
## 3 Greece   2017 21.5  28605.  32.2
## 4 Greece   2018 19.3  29141.  31.7
```

They are most useful if used in conjunction with `dplyr::summarise()`, which summarizes variables. While it can be used without `dplyr::group_by()`, it is most useful if it is applied to grouped data sets: then it computes summary statistics for each group.

```
data_raw %>%
  summarise(
    avg_gdp=mean(gdp)
  )
```

```
##    avg_gdp
## 1 41076.08
```

```
data_raw_grouped %>%
  summarise(
    avg_gdp=mean(gdp)
  )
```

```
## # A tibble: 2 x 2
##   country avg_gdp
##   <chr>     <dbl>
## 1 Germany  53279.
## 2 Greece   28873.
```

You can also summarized more than one column:

```
data_raw_grouped %>%
  summarise(
    avg_gdp=mean(gdp),
    median_unemp=median(unemp)
  )
```

```
## # A tibble: 2 x 3
##   country avg_gdp median_unemp
##   <chr>     <dbl>        <dbl>
## 1 Germany  53279.         3.57
## 2 Greece   28873.        20.4
```

Note that `dplyr::summarise()` drops all columns that it is not asked to compute summary statistics for, except potential grouping variables.

10 A final example

Thanks to the pipes it is easy to chain the many different wrangling steps into one function call. But in practice it is very important that you (1) inspect your raw data very clearly, (2) write down the desired end product, and then (3) think about the single steps required to reach the desired outcome. Each step should address one (and only one) wrangling challenge.

To illustrate this, assume we start start with this raw data:

```
str(data_final_expl)
```

```
## Classes 'data.table' and 'data.frame':   84 obs. of  4 variables:
##  $ country: chr  "Austria" "Austria" "Austria" "Austria" ...
##  $ year   : int  2000 2001 2002 2003 2004 2005 2006 2007 2008 2009 ...
##  $ unemp  : num  4.69 4.01 4.85 4.78 5.83 ...
##  $ gdp    : num  46470 46879 47419 47633 48633 ...
##  - attr(*, ".internal.selfref")=<externalptr>
```

What we want to do is to compute the difference in the country averages of the variables for the time periods 2005-2007 and 2010-2013. This would look like this:[4]

To achieve this, we need to chain a number of wrangling challenges introduced above:

```
data_final_expl %>%
  dplyr::mutate(
    period = ifelse(
      test = year %in% 2005:2007,
      yes = "Early",
      no =  ifelse(
        test = year %in% 2010:2013,
        yes = "Late",
        no = NA))
  ) %>%
  dplyr::filter(!is.na(period)) %>%
  group_by(country, period) %>%
  summarise(
    avg_unemp = mean(unemp),
    avg_gdp = mean(gdp),
    .groups = "drop"
  ) %>%
  tidyr::pivot_longer(
    cols = dplyr::all_of(c("avg_unemp", "avg_gdp")),
    names_to = "indicator",
    values_to = "values") %>%
```

[4] We have not yet covered the function `ifelse()`. It contains a logical test as a first argument, and then two further arguments: one return value for the case in which the test returns `TRUE`, and one for which the test returns `FALSE`.

```
  tidyr::pivot_wider(
    names_from = "period",
    values_from = "values") %>%
  dplyr::mutate(
    Difference = Late - Early
  ) %>%
  dplyr::select(-Early, -Late) %>%
  tidyr::pivot_wider(
    names_from = "indicator",
    values_from = "Difference")
```

```
## # A tibble: 4 x 3
##   country avg_unemp avg_gdp
##   <chr>       <dbl>   <dbl>
## 1 Austria    -0.348   1899.
## 2 Germany    -4.18    3570.
## 3 Greece     11.5    -6234.
## 4 Italy       3.02   -3014.
```

If you have trouble understanding the many steps, redo the computations yourself and always check what happens in the single steps. It is not a good idea to write such a long chain in one working step, but rather to make sure that you always understand what happens in any single step, and then expand the chain one by one.