Data preprocessing

Data visualization 2024/2025

Matteo Ceccarello

2025-01-15

These notes cover a set of basic tools to import data and manipulate it prior to visualization. Your might object that data processing is orthogonal to data visualization, and you would be right. However, data visualization skills without any data to visualize are rather useless. In this notes we will thus look at a few tools, tricks, and advice to handle and organize our data.

File reading

There are several formats in which you can find data, which can be broadly classified in two types:

- Textual
- Binary

Each comes with its set of strenghts and weaknesses. Textual data is:

- Human readable
- (possibly) Easy to parse
- Makes it easy to interoperate between different environments
- Slow
- May waste space (but compression helps)

On the other hand, binary data is:

- Fast
- Compact
- May require specialized software to be read
- More difficult to access in different environments
- Obviously non human readable

Textual format: CSV

One of the simplest textual formats is the CSV format, which stands for Comma Separated Values. Files in this format contain a record for each line, with the first line being usually a header, as shown in the following example.

```
state, year, energy
DE, 2015, 8.803
IT, 2015, 9.879
DE, 2016, 8.917
IT, 2016, 10.062
```

In the above example, the fields of each record are separated with commas (hence the name of the file format). To read the file using `R` you can use the `read_csv` function from the `tidyverse` library.[1] The following snippet provides an example.

[1] There is an older `read.csv` from the standard library. While it works, the newer `read_csv` function is faster and has better handling of corner cases.

```
library(tidyverse)

read_csv("data/example.csv")
```

The delimiter separating fields isn't always a comma, hence in the `tidyverse` library we also have functions such as `read_tsv` and `read_delim` to handle different types of files. There is a handy cheatsheet[2] listing all the available functions. The cheatsheet also lists functions to interact with files created by Excel.

[2] https://rstudio.github.io/cheatsheets/html/data-import.html

Handling missing values in the input

Most of the times we do not have all the information for all the records in our data. In CSV files these missing values are encoded by either omitting the field (like in the second line of the example below) or by using a special character (like in the third line, where the `:` character denotes a missing value).

```
state,year,energy
DE,2015,8.803
IT,2015,
DE,:,8.917
IT,2016,10.062
```

To handle these cases, the `read_csv` accepts the argument `na` that allows to specify which strings encode missing values in the data at hand. The strings to be interpreted as missing values are passed as a vector.

```
read_csv("data/example_na.csv", na=c("", ":"))
```

```
# A tibble: 4 x 3
  state  year energy
  <chr> <dbl>  <dbl>
1 DE     2015   8.80
2 IT     2015  NA
3 DE       NA   8.92
4 IT     2016  10.1
```

Note that the output contains has missing values encoded with `NA` in the appropriate places.

Specifying column names manually

Sometimes files are missing the header with the column names, like in the following example.

```
DE,2015,8.803
IT,2015,9.879
DE,2016,8.917
IT,2016,10.062
```

We can fix this situation easily by passing a vector of column names to the `read_csv` function by means of the `col_names` argument.

```
read_csv(
  "data/example_no_names.csv",
   col_names = c(
     "state",
     "year",
     "energy"
   )
)
```

```
# A tibble: 4 x 3
  state  year energy
  <chr> <dbl>  <dbl>
1 DE     2015   8.80
2 IT     2015   9.88
3 DE     2016   8.92
4 IT     2016  10.1
```

Data frames

A data frame is the most common way to represent tabular data in R. You can think of it as a table in a spreadsheet program like Excel. A data frame in fact has several columns with names and many rows.

One of the key points is that we usually don't manipulate single values directly. Rather, entire columns are processed all in one go.

The `tidiverse` library provides an enhanced data frame implementation, called a `tibble` that provides:

- better printing
- better type handling
- better defaults for building and subsetting
- possibility to use invalid identifiers as column names

As a running example in what follows we will use a table of all the flightrs leaving New York airports in 2013. This data is provided in the package `nycflights13` which you can install with:

```
install.packages("nycflights13")
```

This package contains 5 datasets:

```
nycflights13::airlines
nycflights13::airports
nycflights13::flights
nycflights13::planes
nycflights13::weather
```

Which you can easily inspect by typing in the console the following command:

```
View(nycflights13::flights)
```

Printing the `flighs` data frame provides some useful information, like the size (19 columns and 336,776 rows), the column names, and the data types of each column

```
# A tibble: 336,776 x 19
    year month   day dep_time sched_dep_time dep_delay arr_time sched_arr_time
   <int> <int> <int>    <int>          <int>     <dbl>    <int>          <int>
 1  2013     1     1      517            515         2      830            819
 2  2013     1     1      533            529         4      850            830
 3  2013     1     1      542            540         2      923            850
 4  2013     1     1      544            545        -1     1004           1022
 5  2013     1     1      554            600        -6      812            837
 6  2013     1     1      554            558        -4      740            728
 7  2013     1     1      555            600        -5      913            854
 8  2013     1     1      557            600        -3      709            723
 9  2013     1     1      557            600        -3      838            846
10  2013     1     1      558            600        -2      753            745
# i 336,766 more rows
# i 11 more variables: arr_delay <dbl>, carrier <chr>, flight <int>,
#   tailnum <chr>, origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>,
#   hour <dbl>, minute <dbl>, time_hour <dttm>
```

Data processing with `dplyr`

The `dplyr` library is part of the `tidyverse` and provides a consistent set of functions to solve most data manipulation problems[3]. When loading the `tidyverse` library with `library(tidyverse)`, all the functions from `dplyr` are loaded as well.

One of the main selling points of `dplyr` is the consistency of its API[4].

The call to most `dplyr` functions looks like the following:

```
function_name(data_frame, ...other arguments...)
```

The first argument is a table of data, and then there are other arguments that are specific to each function. In particular, in `...other arguments...` you can easily refer to column names of the data table in the first argument.

In what follows keep in mind this calling convention, as it will be key part of building data processing *pipelines*.

In the following we will see some of the most important functions in the package, using the `nycflights13` datasets as a working example. In particular, the following code snippets assume that `library(nycflights13)` has been called.

An important thing to know is that *all* the following functions return *a new table*, rather than modifying the input table in place.

[3] In many respects, the aim of `dplyr` is similar to that of `pandas` for Python.

[4] Application Programming Interface, i.e. the set of functions and calling conventions that a library exposes.

Filtering rows

One of the most basic operations is to select a subset of the rows (or observations) of a table according to some predicate on the values. This is accomplished with the `filter` function that takes as a first argument a table. The other arguments are *predicates* over columns. In the following example we select the rows where the `month` column takes value 1, and the `day` column takes values larger than or equal to 15.

Exercises

- How can we filter all the flights where the delay was less than 10?
- How can we filter all the flights with missing departure delay?

```
filter(flights, month == 1, day >= 15)
```

Note that the predicates are over *entire* columns, and that column names are referred to *unquoted*.

Sorting data

Another basic need is to arrange data according to the values in some column. We can use the `arrange` function for that. After the usual table as a first argument, it takes the columns to sort on. The order is increasing.

```
arrange(flights, dep_delay)
```

If we want to sort in decreasing (or `desc`ending) order we have to wrap the column name in a call to the `desc` function.

```
arrange(flights, desc(dep_delay))
```

Getting column names

To obtain a list of column names we can use the `names` function.

```
names(flights)
```

Selecting columns

While `filter` slices a table horizontally, the `select` function selects a subset of the `columns`[5].

[5] This is very similar to the `SELECT` statement in SQL.

```
select(flights, day, tailnum, distance)
```

As usual, the first argument is the table, the others are column names

A related function is `rename`, that returns a copy of the table with some column names changed.

```
rename(flights, tail_num = tailnum)
```

Creating new columns

We can also create a copy of the table with new columns that are the result of computations over already_existing columns.

Exercise

How can we add a column with the logarithm of the distance?

In the following example we create a new column `speed` that reports the average speed of each flight.

```
speed_data <- select(flights, distance, air_time)
mutate(speed_data, speed = distance / air_time * 60)
```

```
# A tibble: 336,776 x 3
   distance air_time speed
      <dbl>    <dbl> <dbl>
 1     1400      227  370.
 2     1416      227  374.
 3     1089      160  408.
 4     1576      183  517.
 5      762      116  394.
 6      719      150  288.
 7     1065      158  404.
 8      229       53  259.
 9      944      140  405.
10      733      138  319.
# i 336,766 more rows
```

Note that, despite the name, the `mutate` function returns a *new* data frame rather than modifying its input. Also, note that all arguments except for the first take the following form:

```
column_name = column_expression
```

where the column expression is any operation involving one or more columns. The operations are *vectorized*, i.e. they are applied to each individual value separately.

Conditional mutation

Sometimes we need to apply a function only to a subset of the rows, leaving the other untouched but still retaining them. In this case using `filter` would not cut it, as it discards some rows.

For this kind of situation there is a helper function `if_else` which takes three arguments: a condition and the values to be used when the condition is false or true.

```
if_else( condition, value_if_true, value_if_valse )
```

In the followin g example we are adding to the `flights` dataset a column with a label that reports whether the flight departed with some delay.

```
mutate(flights,
       delayed_str = if_else(dep_delay > 0, "delayed", "not_delayed"))
```

Counting

Use the `count` function to count how many rows are in the table.

```
count(flights)
```

A useful variant is to count the number of entries for each *group* defined by a given column (or combination of columns).

For instance, the following snippet counts how many rows we have for each month in the `flights` dataset.

```
count(flights, month)
```

Summarising and aggregating data

A more general variant of the above requires to *aggregate* data, for instance computing the mean or maximum of a column.

The function to achieve this is `summarise`, whose usage is exemplified in the following where we compute the average delay.

```
summarise(
  flights,
  avg_delay = mean(dep_delay, na.rm = TRUE)
)
```

Note that we are passing the `na.rm` argument to the function `mean`. This is because a single `NA` value in the `dep_delay` column would make the entire average `NA`, given that `NA`s propagate in computations. The `na.rm` parameter allows to instruct the `mean` function to ignore missing values in the computation. Other aggregation functions (like `median`, `max`, `min`, `quantile`) have the same parameter with the same goal.

```
# A tibble: 1 x 1
  avg_delay
      <dbl>
1      12.6
```

Note that the syntax is similar to that of `mutate`: there are column expressions whose results are assigned to new columns (in this example to the `avg_delay` column). The difference lies in the type of column expression: for `mutate` we need something that returns the same number of rows as the input, whereas for `summarise` we need to coalesce all values into one.

Doing summarization a the table-level is useful, but even more useful is to perform summarization by groups. To this end we can create groups – defined by data values – using the `group_by` function.

```
grouped_flights <- group_by(flights, year, month)
```

Exercises

- How can we get the maximum and minimum delay by year and month?
- How can you replicate the behavior of `count(flights, month)`? You can use `group_by`, `summarise` and `n()` (which returns the number of rows in a group).
- Come up with a way to compute the fraction of delayed flights per month

The `grouped_flights` variable now holds the same data as `flights` but partitioned by all possible combinations of `year` and `month` (i.e. `year==2013` and `month==1`, `year==2013` and `month==2`, and so on).

Calling summarize on this dataset results in the application of the summarization function *to each group separately*:

```
summarise(grouped_flights, avg_delay = mean(dep_delay, na.rm = TRUE))
```

```
# A tibble: 12 x 3
# Groups:   year [1]
    year month avg_delay
   <int> <int>     <dbl>
 1  2013     1     10.0
 2  2013     2     10.8
 3  2013     3     13.2
 4  2013     4     13.9
 5  2013     5     13.0
 6  2013     6     20.8
 7  2013     7     21.7
 8  2013     8     12.6
 9  2013     9      6.72
10  2013    10      6.24
11  2013    11      5.44
12  2013    12     16.6
```

Piping

Instead of repeatedly assigning the result of each function to a variable that will be used just once, you can use the `|>` operator.

This operator takes the *result* of the function on its left and makes it the *first argument* of the function on its right.

```
flights |>
  select(month, distance, air_time) |>
  mutate(speed = distance / air_time * 60) |>
  group_by(month) |>
  summarise(avg_speed = mean(speed, na.rm=TRUE))
```

The above code takes the `flights` dataset and pipes it into the `select` function to pick a subset of the columns, then pipes the result into `mutate` to add a new column, then groups by month and computes the monthly average speed.

In the rest of the course we will make heavy use of pipes to simplify the code.

Joining tables

Sometimes the information we need is in different data frames and we need to *join* them. This is a family of operations borrowed from the database world. The basic idea is that we use the values taken by some columns in one table to match rows in the other table.

There are several types of joins which all share a common idea: given two tables, we are interested in a subset of the cartesian product of the rows. The basic setup is displayed in Figure Figure 1. We have two tables `x` and `y`, both with two columns. The first column, with digits, will be used as a *key* column, the others will be *value* columns.

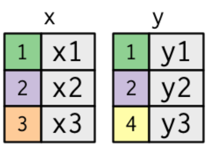

Figure 1: The tables we will use for the join examples.

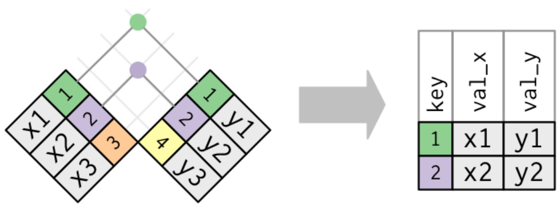

Figure 2: Inner join.

The most common join type we will use is the *inner join*. In the inner join, we keep only the pairs of rows that share the same *key* values. In Figure Figure 2 this means that only the rows corresponding to rows with keys 1 and 2 are part of the output.

Sometimes we are interest in keeping all the rows from either of the two tables of the join. These are called *outer* joins and there exist three variants:

- Left outer joins, keeping all rows from the left table (Figure Figure 3)
- Right outer joins, keeping all rows from the right table (Figure Figure 4)
- Full outer joins, keeping all rows from both tables (Figure Figure 5)

In all cases, rows from one table that do not have a matching row in the other table have the relevant entries filled with `NA` values.

The `tidyverse` package provides functions for all these use cases. In particular, if `key_column` is the name of the column by which we want to join then the following computes an inner join.

```
inner_join(x,y, by="key")
```

The following three functions, instead compute a left, a right, and a full outer join.

```
left_join(x,y, by="key")
```

```
right_join(x,y, by="key")
```

```
full_join(x,y, by="key")
```

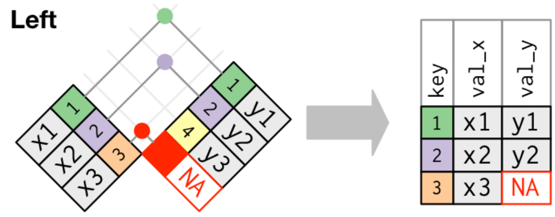

Figure 3: Left outer join.

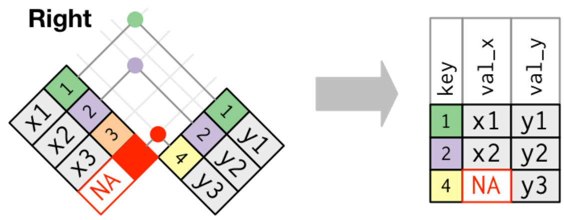

Figure 4: Right outer join.

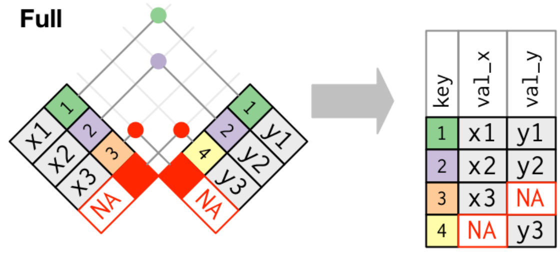

Figure 5: Full outer join.

As an example, the following snippet of code selects a subset of the columns from the `flights` table and from the `planes` tables (which contain information about the airplanes). Both tables share the `tailnum` of the airplane operating the flight. The last line joins the two datasets by the `tailnum` column.

```
flights2 <- select(flights, year, origin, dest, tailnum)
planes2 <- select(planes, tailnum, year, manufacturer, model)
inner_join(flights2, planes2, by="tailnum")
```

```
# A tibble: 284,170 x 7
   year.x origin dest  tailnum year.y manufacturer     model
    <int> <chr>  <chr> <chr>    <int> <chr>            <chr>
 1   2013 EWR    IAH   N14228    1999 BOEING           737-824
 2   2013 LGA    IAH   N24211    1998 BOEING           737-824
 3   2013 JFK    MIA   N619AA    1990 BOEING           757-223
 4   2013 JFK    BQN   N804JB    2012 AIRBUS           A320-232
 5   2013 LGA    ATL   N668DN    1991 BOEING           757-232
 6   2013 EWR    ORD   N39463    2012 BOEING           737-924ER
 7   2013 EWR    FLL   N516JB    2000 AIRBUS INDUSTRIE A320-232
 8   2013 LGA    IAD   N829AS    1998 CANADAIR         CL-600-2B19
 9   2013 JFK    MCO   N593JB    2004 AIRBUS           A320-232
10   2013 JFK    PBI   N793JB    2011 AIRBUS           A320-232
# i 284,160 more rows
```

Notice that the `year` column is present in both datasets, with different meanings. In `flights` is the year of the flights, in `planes` it is the year in which the plane was manufactured. To disabinguate, the `inner_join` function automatically renames the two columns in the output.

Further information is provided in the cheatsheet at https://rstudio.github.io/cheatsheets/html/data-transformation.html.

Tidy data

Data can come in many possible arrangements, but a particularily convenient one is *tidy data*. In simple terms, for data to be tidy:

- Each variable must have its own column;
- Each observation must have its own row;
- Each value must have its own cell.

Figure Figure 6 exemplifies such dataset. Tidy data is easier to process and visualize.

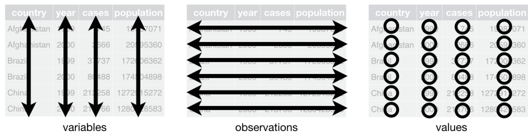

Figure 6: The characteristics of a tidy dataset.

There are a couple of alarm bells that help in finding out when data is not tidy:

- Column names that should be values
- Values that should be column names

Most datasets are not tidy, since:

- One variable might be spread across multiple columns;
- One observation might be scattered across multiple rows.

We mainly have two functions to fix these two problems:

- `pivot_longer`
- `pivot_wider`

For instance, the following dataset is not tidy because column names are actually values, in particular they are years.

```
table4b
```

```
# A tibble: 3 x 3
  country         `1999`     `2000`
  <chr>            <dbl>      <dbl>
1 Afghanistan   19987071   20595360
2 Brazil       172006362  174504898
3 China       1272915272 1280428583
```

To fix this situation we use the `pivot` longer function, which reshapes the table making the selected column names into values of a new column.

```r
pivot_longer(
  table4b,
  `1999`:`2000`, # selected column names
  names_to = "year", # name of the new column
  values_to = "population" # name of the column
                           # that will hold the values
                           # previously scattered
                           # across multiple columns.
)
```

```
# A tibble: 6 x 3
  country     year  population
  <chr>       <chr>      <dbl>
1 Afghanistan 1999    19987071
2 Afghanistan 2000    20595360
3 Brazil      1999   172006362
4 Brazil      2000   174504898
5 China       1999  1272915272
6 China       2000  1280428583
```

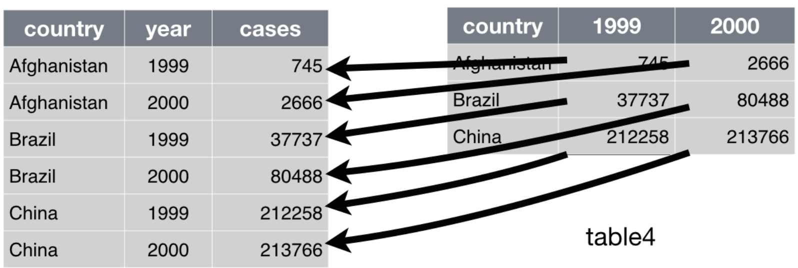

Figure 7: The effect of the `pivot_longer` operation.

This dataset is not tidy because the `type` column contains values that should actually be column names themselves.

```r
table2
```

```
# A tibble: 12 x 4
   country      year type            count
   <chr>       <dbl> <chr>           <dbl>
 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
```

To fix it we use the `pivot_wider` function, which takes a table as a first argument, and parameters to know from which columns it should take names and values.

```r
pivot_wider(table2, names_from=type, values_from=count)
```

```
# A tibble: 6 x 4
  country      year  cases population
  <chr>       <dbl>  <dbl>      <dbl>
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
```

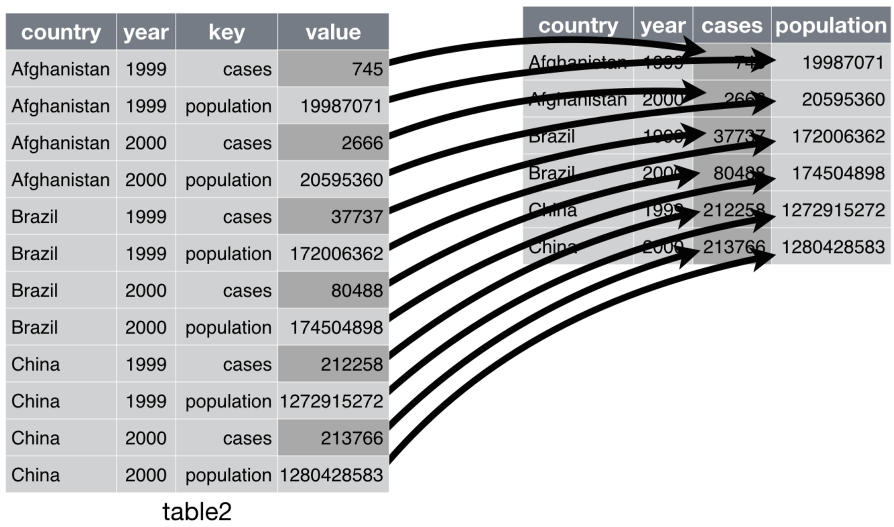

Figure 8: The effect of the `pivot_wider` operation.

Finally, the following table is not tidy because the values in the `rate` column are compound, they should actually be separated in two values.

```r
table3
```

```
# A tibble: 6 x 3
  country      year rate
  <chr>       <dbl> <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
```

The fix is provided by the `separate` function, which, along with the table to work on, takes the column to separate and the names of the resulting columns. The separation happens on any non alphanumeric character, or on the character provided by the `sep` argument. Finally, providing the `convert` argument allows to automatically convert the data.

```r
separate(table3, rate, into = c("cases", "population"), convert = TRUE, sep="/")
```

```
# A tibble: 6 x 4
  country      year  cases population
  <chr>       <dbl>  <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
```

We close with a small data cleaning exercise using the energy productivity data[6].

[6] Available from http://data.europa.eu/euodp/en/data/dataset/xWiT1

```
unit,geo\time   2000     2001     2002     2003     2004     2005     2006     2007     2008     2009     2010
EUR_KGOE,AT 8.688    8.340    8.429    8.083    8.151    8.060    8.265    8.721    8.779    8.899    8.489    9.0
EUR_KGOE,BA :    :    :    :    :    :    :    :    :    :    :    :    :    :    2.240    2.242    2.113    2.184    2.0
EUR_KGOE,BE 4.730    4.829    4.953    4.767    4.906    5.051    5.182    5.376    5.234    5.523    5.256    5.7
EUR_KGOE,BG 1.315    1.306    1.417    1.451    1.587    1.607    1.666    1.810    1.943    2.129    2.113    2.0
EUR_KGOE,CY 5.344    5.503    5.786    5.513    6.323    6.004    6.062    6.166    6.126    6.210    6.591    6.6
EUR_KGOE,CZ 2.793    2.805    2.811    2.788    2.850    3.068    3.203    3.389    3.545    3.598    3.473    3.6
EUR_KGOE,DE 6.833    6.754    6.889    6.820    6.850    6.930    6.981    7.548    7.528    7.591    7.520    8.2
```

This dataset has several issues:

- column names are years
- the first column has contains both the unit of measure and the country

we will thus use `pivot_longer` and `separate`:

```r
energy_productivity <- read_tsv(
  "data/energy_productivity.tsv.gz",
  na=c("", ":") # set how missing values are encoded
)

energy_productivity |>
  # make years into values
  pivot_longer(`2000`:`2019`, names_to='year', values_to='energy_productivity') |>
  # separate units and country codes. Note the sep argument.
  # Also note that we are wrapping the column name in backticks, since it is a non
  # valid identifier in R.
  separate(`unit,geo\\time`, into=c("unit", "state"), sep=',')
```

```
# A tibble: 1,560 x 4
   unit     state year  energy_productivity
   <chr>    <chr> <chr>               <dbl>
 1 EUR_KGOE AT    2000                 8.69
 2 EUR_KGOE AT    2001                 8.34
 3 EUR_KGOE AT    2002                 8.43
 4 EUR_KGOE AT    2003                 8.08
 5 EUR_KGOE AT    2004                 8.15
 6 EUR_KGOE AT    2005                 8.06
 7 EUR_KGOE AT    2006                 8.26
 8 EUR_KGOE AT    2007                 8.72
 9 EUR_KGOE AT    2008                 8.78
10 EUR_KGOE AT    2009                 8.90
# i 1,550 more rows
```