[**https://rafalab.github.io/dsbook/string-processing.html**](https://rafalab.github.io/dsbook/string-processing.html)

**Reshaping data**

As we have seen through the book, having data in *tidy* format is what makes the tidyverse flow. After the first step in the data analysis process, importing data, a common next step is to reshape the data into a form that facilitates the rest of the analysis. The **tidyr** package includes several functions that are useful for tidying data.

We will use the fertility wide format dataset described in Section [4.1](https://rafalab.github.io/dsbook/tidyverse.html#tidy-data) as an example in this section.

library(tidyverse)

library(dslabs)

path <- system.file("extdata", package="dslabs")

filename <- file.path(path, "fertility-two-countries-example.csv")

wide\_data <- read\_csv(filename)

## pivot\_longer

One of the most used functions in the **tidyr** package is pivot\_longer, which is useful for converting wide data into tidy data.

As with most tidyverse functions, the pivot\_longer function’s first argument is the data frame that will be converted. Here we want to reshape the wide\_data dataset so that each row represents a fertility observation, which implies we need three columns to store the year, country, and the observed value. In its current form, data from different years are in different columns with the year values stored in the column names. Through the names\_to and values\_to argument we will tell pivot\_longer the column names we want to assign to the columns containing the current column names and observations, respectively. The default names are name and value, which are often usable choices. In this case a better choice for these two arguments would be year and fertility. Note that nowhere in the data file does it tell us this is fertility data. Instead, we deciphered this from the file name. Through cols,the second argument we specify the columns containing observed values; these are the columns that will be pivoted. The default is to pivot all columns so, in most cases, we have to specify the columns. In our example we want columns 1960, 1961 up to 2015.

The code to pivot the fertility data therefore looks like this:

new\_tidy\_data <- pivot\_longer(wide\_data, `1960`:`2015`, names\_to = "year", values\_to = "fertility")

We can also use the pipe like this:

new\_tidy\_data <- wide\_data %>%

pivot\_longer(`1960`:`2015`, names\_to = "year", values\_to = "fertility")

We can see that the data have been converted to tidy format with columns year and fertility:

head(new\_tidy\_data)

*#> # A tibble: 6 x 3*

*#> country year fertility*

*#> <chr> <chr> <dbl>*

*#> 1 Germany 1960 2.41*

*#> 2 Germany 1961 2.44*

*#> 3 Germany 1962 2.47*

*#> 4 Germany 1963 2.49*

*#> 5 Germany 1964 2.49*

*#> # … with 1 more row*

and that each year resulted in two rows since we have two countries and this column was not pivoted. A somewhat quicker way to write this code is to specify which column will **not** include in the pivot, rather than all the columns that will be pivoted:

new\_tidy\_data <- wide\_data %>%

pivot\_longer(-country, names\_to = "year", values\_to = "fertility")

The new\_tidy\_data object looks like the original tidy\_data we defined this way

data("gapminder")

tidy\_data <- gapminder %>%

filter(country %in% c("South Korea", "Germany") & !is.na(fertility)) %>%

select(country, year, fertility)

with just one minor difference. Can you spot it? Look at the data type of the year column:

class(tidy\_data$year)

*#> [1] "integer"*

class(new\_tidy\_data$year)

*#> [1] "character"*

The pivot\_longer function assumes that column names are characters. So we need a bit more wrangling before we are ready to make a plot. We need to convert the year column to be numbers:

new\_tidy\_data <- wide\_data %>%

pivot\_longer(-country, names\_to = "year", values\_to = "fertility") %>%

mutate(year = as.integer(year))

Note that we could have also used the mutate and as.numeric.

Now that the data is tidy, we can use this relatively simple ggplot code:

new\_tidy\_data %>% ggplot(aes(year, fertility, color = country)) +

geom\_point()

## 21.2 pivot\_wider

As we will see in later examples, it is sometimes useful for data wrangling purposes to convert tidy data into wide data. We often use this as an intermediate step in tidying up data. The pivot\_wider function is basically the inverse of pivot\_longer. The first argument is for the data, but since we are using the pipe, we don’t show it. The names\_from argument tells pivot\_wider which variable will be used as the column names. The values\_from argument specifies which variable to use to fill out the cells.

new\_wide\_data <- new\_tidy\_data %>%

pivot\_wider(names\_from = year, values\_from = fertility)

select(new\_wide\_data, country, `1960`:`1967`)

*#> # A tibble: 2 x 9*

*#> country `1960` `1961` `1962` `1963` `1964` `1965` `1966` `1967`*

*#> <chr> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>*

*#> 1 Germany 2.41 2.44 2.47 2.49 2.49 2.48 2.44 2.37*

*#> 2 South Korea 6.16 5.99 5.79 5.57 5.36 5.16 4.99 4.85*

Similar to pivot\_wider, names\_from and values\_from default to name and value.

## 21.3 separate

The data wrangling shown above was simple compared to what is usually required. In our example spreadsheet files, we include an illustration that is slightly more complicated. It contains two variables: life expectancy and fertility. However, the way it is stored is not tidy and, as we will explain, not optimal.

path <- system.file("extdata", package = "dslabs")

filename <- "life-expectancy-and-fertility-two-countries-example.csv"

filename <- file.path(path, filename)

raw\_dat <- read\_csv(filename)

select(raw\_dat, 1:5)

*#> # A tibble: 2 x 5*

*#> country `1960\_fertility` `1960\_life\_expectancy` `1961\_fertility`*

*#> <chr> <dbl> <dbl> <dbl>*

*#> 1 Germany 2.41 69.3 2.44*

*#> 2 South Korea 6.16 53.0 5.99*

*#> # … with 1 more variable: 1961\_life\_expectancy <dbl>*

First, note that the data is in wide format. Second, notice that this table includes values for two variables, fertility and life expectancy, with the column name encoding which column represents which variable. Encoding information in the column names is not recommended but, unfortunately, it is quite common. We will put our wrangling skills to work to extract this information and store it in a tidy fashion.

We can start the data wrangling with the pivot\_longer function, but we should no longer use the column name year for the new column since it also contains the variable type. We will call it name, the default, for now:

dat <- raw\_dat %>% pivot\_longer(-country)

head(dat)

*#> # A tibble: 6 x 3*

*#> country name value*

*#> <chr> <chr> <dbl>*

*#> 1 Germany 1960\_fertility 2.41*

*#> 2 Germany 1960\_life\_expectancy 69.3*

*#> 3 Germany 1961\_fertility 2.44*

*#> 4 Germany 1961\_life\_expectancy 69.8*

*#> 5 Germany 1962\_fertility 2.47*

*#> # … with 1 more row*

The result is not exactly what we refer to as tidy since each observation is associated with two, not one, rows. We want to have the values from the two variables, fertility and life expectancy, in two separate columns. The first challenge to achieve this is to separate the name column into the year and the variable type. Notice that the entries in this column separate the year from the variable name with an underscore:

dat$name[1:5]

*#> [1] "1960\_fertility" "1960\_life\_expectancy" "1961\_fertility"*

*#> [4] "1961\_life\_expectancy" "1962\_fertility"*

Encoding multiple variables in a column name is such a common problem that the **readr** package includes a function to separate these columns into two or more. Apart from the data, the separate function takes three arguments: the name of the column to be separated, the names to be used for the new columns, and the character that separates the variables. So, a first attempt at this is:

dat %>% separate(name, c("year", "name"), "\_")

Because \_ is the default separator assumed by separate, we do not have to include it in the code:

dat %>% separate(name, c("year", "name"))

*#> Warning: Expected 2 pieces. Additional pieces discarded in 112 rows [2,*

*#> 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, 36, 38,*

*#> 40, ...].*

*#> # A tibble: 224 x 4*

*#> country year name value*

*#> <chr> <chr> <chr> <dbl>*

*#> 1 Germany 1960 fertility 2.41*

*#> 2 Germany 1960 life 69.3*

*#> 3 Germany 1961 fertility 2.44*

*#> 4 Germany 1961 life 69.8*

*#> 5 Germany 1962 fertility 2.47*

*#> # … with 219 more rows*

The function does separate the values, but we run into a new problem. We receive the warning Too many values at 112 locations: and that the life\_expectancy variable is truncated to life. This is because the \_ is used to separate life and expectancy, not just year and variable name! We could add a third column to catch this and let the separate function know which column to fill in with missing values, NA, when there is no third value. Here we tell it to fill the column on the right:

var\_names <- c("year", "first\_variable\_name", "second\_variable\_name")

dat %>% separate(name, var\_names, fill = "right")

*#> # A tibble: 224 x 5*

*#> country year first\_variable\_name second\_variable\_name value*

*#> <chr> <chr> <chr> <chr> <dbl>*

*#> 1 Germany 1960 fertility <NA> 2.41*

*#> 2 Germany 1960 life expectancy 69.3*

*#> 3 Germany 1961 fertility <NA> 2.44*

*#> 4 Germany 1961 life expectancy 69.8*

*#> 5 Germany 1962 fertility <NA> 2.47*

*#> # … with 219 more rows*

However, if we read the separate help file, we find that a better approach is to merge the last two variables when there is an extra separation:

dat %>% separate(name, c("year", "name"), extra = "merge")

*#> # A tibble: 224 x 4*

*#> country year name value*

*#> <chr> <chr> <chr> <dbl>*

*#> 1 Germany 1960 fertility 2.41*

*#> 2 Germany 1960 life\_expectancy 69.3*

*#> 3 Germany 1961 fertility 2.44*

*#> 4 Germany 1961 life\_expectancy 69.8*

*#> 5 Germany 1962 fertility 2.47*

*#> # … with 219 more rows*

This achieves the separation we wanted. However, we are not done yet. We need to create a column for each variable. As we learned, the pivot\_wider function can do this:

dat %>%

separate(name, c("year", "name"), extra = "merge") %>%

pivot\_wider()

*#> # A tibble: 112 x 4*

*#> country year fertility life\_expectancy*

*#> <chr> <chr> <dbl> <dbl>*

*#> 1 Germany 1960 2.41 69.3*

*#> 2 Germany 1961 2.44 69.8*

*#> 3 Germany 1962 2.47 70.0*

*#> 4 Germany 1963 2.49 70.1*

*#> 5 Germany 1964 2.49 70.7*

*#> # … with 107 more rows*

The data is now in tidy format with one row for each observation with three variables: year, fertility, and life expectancy.

## 21.4 unite

It is sometimes useful to do the inverse of separate, unite two columns into one. To demonstrate how to use unite, we show code that, although not the optimal approach, serves as an illustration. Suppose that we did not know about extra and used this command to separate:

var\_names <- c("year", "first\_variable\_name", "second\_variable\_name")

dat %>%

separate(name, var\_names, fill = "right")

*#> # A tibble: 224 x 5*

*#> country year first\_variable\_name second\_variable\_name value*

*#> <chr> <chr> <chr> <chr> <dbl>*

*#> 1 Germany 1960 fertility <NA> 2.41*

*#> 2 Germany 1960 life expectancy 69.3*

*#> 3 Germany 1961 fertility <NA> 2.44*

*#> 4 Germany 1961 life expectancy 69.8*

*#> 5 Germany 1962 fertility <NA> 2.47*

*#> # … with 219 more rows*

We can achieve the same final result by uniting the second and third columns, then pivoting the columns and renaming fertility\_NA to fertility:

dat %>%

separate(name, var\_names, fill = "right") %>%

unite(name, first\_variable\_name, second\_variable\_name) %>%

pivot\_wider() %>%

rename(fertility = fertility\_NA)

*#> # A tibble: 112 x 4*

*#> country year fertility life\_expectancy*

*#> <chr> <chr> <dbl> <dbl>*

*#> 1 Germany 1960 2.41 69.3*

*#> 2 Germany 1961 2.44 69.8*

*#> 3 Germany 1962 2.47 70.0*

*#> 4 Germany 1963 2.49 70.1*

*#> 5 Germany 1964 2.49 70.7*

*#> # … with 107 more rows*

## 21.5 Exercises

1. Run the following command to define the co2\_wide object:

co2\_wide <- data.frame(matrix(co2, ncol = 12, byrow = TRUE)) %>%

setNames(1:12) %>%

mutate(year = as.character(1959:1997))

Use the pivot\_longer function to wrangle this into a tidy dataset. Call the column with the CO2 measurements co2 and call the month column month. Call the resulting object co2\_tidy.

2. Plot CO2 versus month with a different curve for each year using this code:

co2\_tidy %>% ggplot(aes(month, co2, color = year)) + geom\_line()

If the expected plot is not made, it is probably because co2\_tidy$month is not numeric:

class(co2\_tidy$month)

Rewrite your code to make sure the month column is numeric. Then make the plot.

3. What do we learn from this plot?

1. CO2 measures increase monotonically from 1959 to 1997.
2. CO2 measures are higher in the summer and the yearly average increased from 1959 to 1997.
3. CO2 measures appear constant and random variability explains the differences.
4. CO2 measures do not have a seasonal trend.

4. Now load the admissions data set, which contains admission information for men and women across six majors and keep only the admitted percentage column:

load(admissions)

dat <- admissions %>% select(-applicants)

If we think of an observation as a major, and that each observation has two variables (men admitted percentage and women admitted percentage) then this is not tidy. Use the pivot\_wider function to wrangle into tidy shape: one row for each major.

5. Now we will try a more advanced wrangling challenge. We want to wrangle the admissions data so that for each major we have 4 observations: admitted\_men, admitted\_women, applicants\_men and applicants\_women. The trick we perform here is actually quite common: first use pivot\_longer to generate an intermediate data frame and then pivot\_wider to obtain the tidy data we want. We will go step by step in this and the next two exercises.

Use the pivot\_longer function to create a tmp data.frame with a column containing the type of observation admitted or applicants. Call the new columns name and value.

6. Now you have an object tmp with columns major, gender, name and value. Note that if you combine the name and gender, we get the column names we want: admitted\_men, admitted\_women, applicants\_men and applicants\_women. Use the function unite to create a new column called column\_name.

7. Now use the pivot\_wider function to generate the tidy data with four variables for each major.

8. Now use the pipe to write a line of code that turns admissions to the table produced in the previous exercise

# Basic pivoting

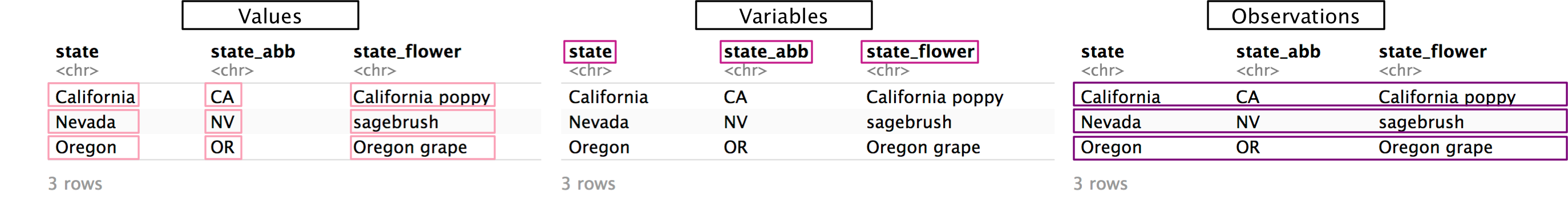
library(tidyverse)

library(dcldata)

Most of the data you’ll encounter won’t be tidy, and it will be your job to figure out how to make it tidy. In this chapter, you’ll learn about two of the most important tidying tools: pivot\_longer() and pivot\_wider().

First, recall the characteristics of tidy data:

* Each value has its own cell.
* Each variable has its own column.
* Each observation has its own row.



Non-tidy data will not fulfill one or more of these characteristics.

## 5.1 Longer

example\_eagle\_nests contains data on the number of [bald eagle nesting sites](https://www.fws.gov/migratorybirds/pdf/management/EagleRuleRevisions-StatusReport.pdf#page=19&zoom=100,0,700) across multiple regions and years.

*# Source: US Fish and Wildlife Service*

example\_eagle\_nests

*#> # A tibble: 3 × 3*

*#> region `2007` `2009`*

*#> <chr> <dbl> <dbl>*

*#> 1 Pacific 1039 2587*

*#> 2 Southwest 51 176*

*#> 3 Rocky Mountains and Plains 200 338*

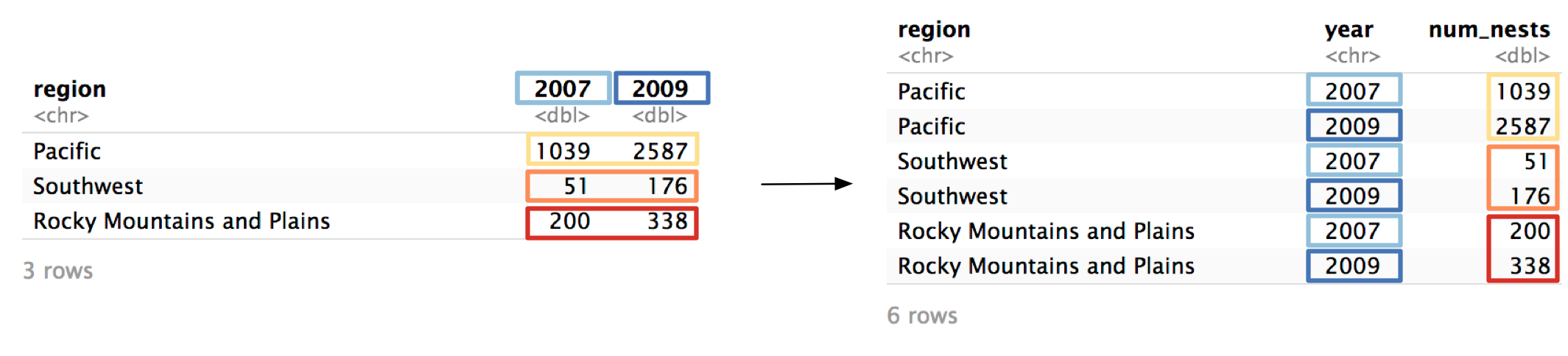
The data isn’t tidy. In the following steps, we’ll show you how to tidy example\_eagle\_nests using pivot\_longer().

**Step 1**: Identify the variables.

There are three variables in this dataset:

* region: The US region where the nests are located.
* year: The year the nests were found.
* num\_nests: The number of nests found.

Only one of these variables (region) is currently a column. Values of year are currently stored horizontally as column names, and values of num\_nests are stored as values of 2007 and 2009. In order for this data to be tidy, we’ll need to pivot 2007 and 2009 into a year column, and the values of 2007 and 2009 into a num\_nests column.



Now that we’ve identified the variables, we can start filling in our call to pivot\_longer(). We’ll need three arguments, which we’ll identify over the next three steps.

example\_eagle\_nests %>%

pivot\_longer(

*# Step 2*

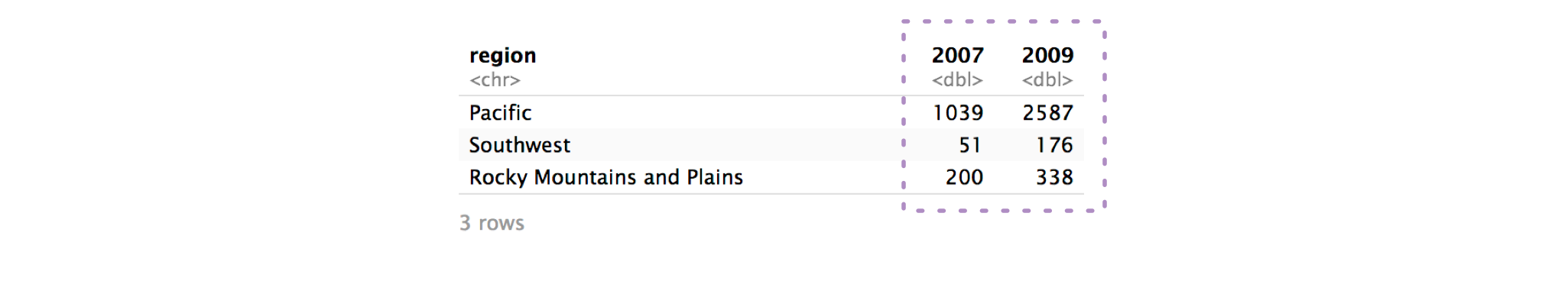
*# Step 3*

*# Step 4*

)

**Step 2**: Identify the columns to pivot.

To decide which columns to pivot, identify which columns are keeping the data from being tidy. In our example, those columns are 2007 and 2009.



2007 and 2009 are actually values of year, not variables themselves, and their values are actually values of num\_nests.

The cols argument controls which columns pivot\_longer() pivots.

example\_eagle\_nests %>%

pivot\_longer(

cols = c(`2007`, `2009`),

*# Step 3*

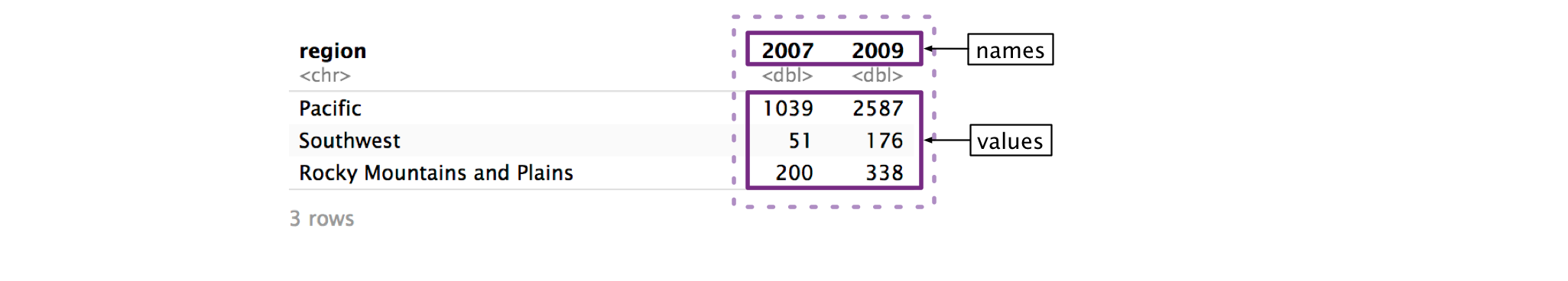
*# Step 4*

)

cols is similar to select(). You can specify columns by name, with contains(), starts\_with(), etc. Here, we have to wrap 2007 and 2009 in backticks (` `) because they start with numbers.

**Step 3**: Name the column that will store the values from the column names.

Now, we’re just going to focus on the columns we identified in cols. Ultimately, pivot\_longer() is going to move both the names of these columns and their values into new, separate columns.



First, we’ll focus on the column names: 2007 and 2009. pivot\_longer()’s names\_to argument controls the name of the column that will store the old column names. We want to name this new column "year". The argument is called names\_to because you’re specifying which column to move the column **names** to.

example\_eagle\_nests %>%

pivot\_longer(

cols = c(`2007`, `2009`),

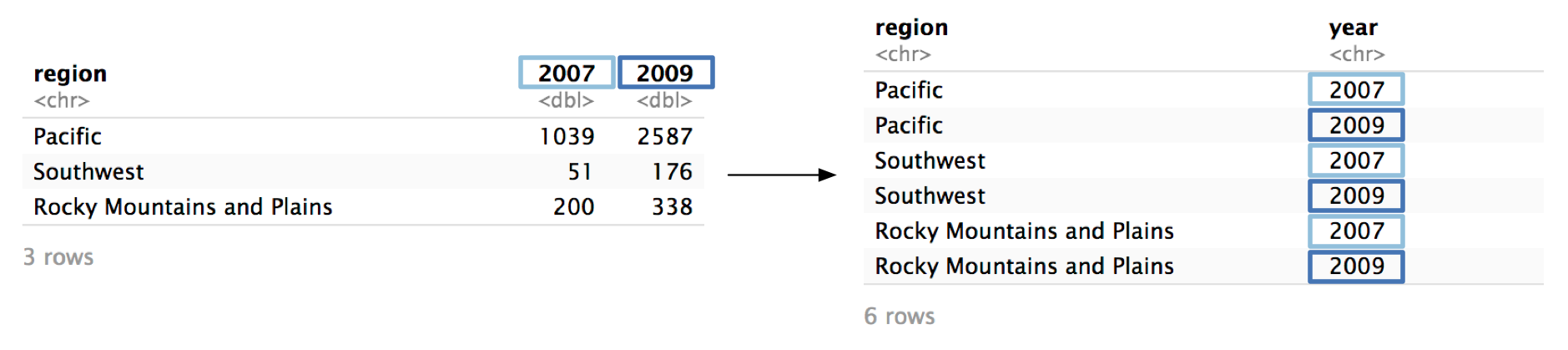
names\_to = "year",

*# Step 4*

)

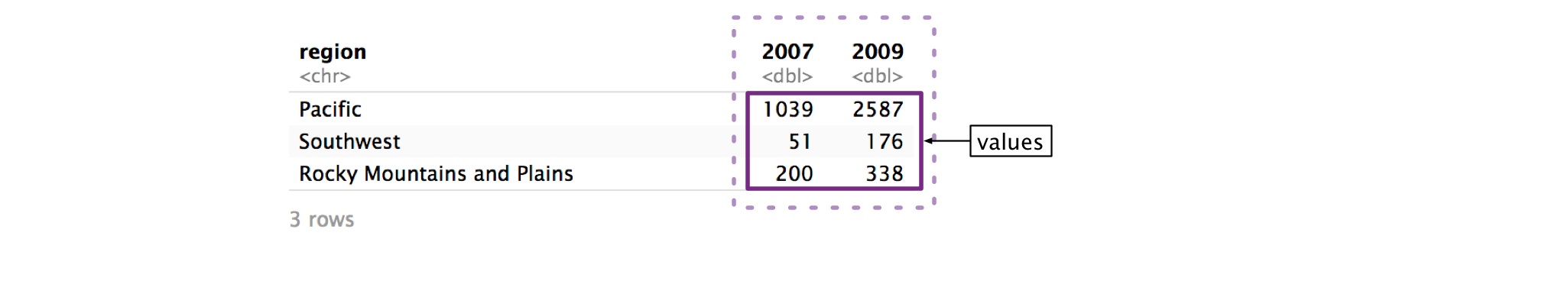
Note that the argument to names\_to has to be in quotes, while the arguments to cols do not. It’s easy to get confused about which pivot arguments need to be quoted. Here’s the general rule: if you’re identifying an existing column (e.g., 2007), do not quote. If you’re talking about a column that does not currently exist (e.g., year), quote it.

Now, pivot\_longer() will create a new column called year and fill it with the column names 2007 and 2009. Because we specified two columns in cols, we will get two values of year for each region.



**Step 4**: Name the column that will store the column values.

Now, we need to name the column that will store the values from 2007 and 2009.



Just as names\_to controls the name of the column for the names, values\_to controls the name of column for the values. In example\_eagle\_nests, the column values represent the number of nests, so we’ll name the new column "num\_nests".

example\_eagle\_nests %>%

pivot\_longer(

cols = c(`2007`, `2009`),

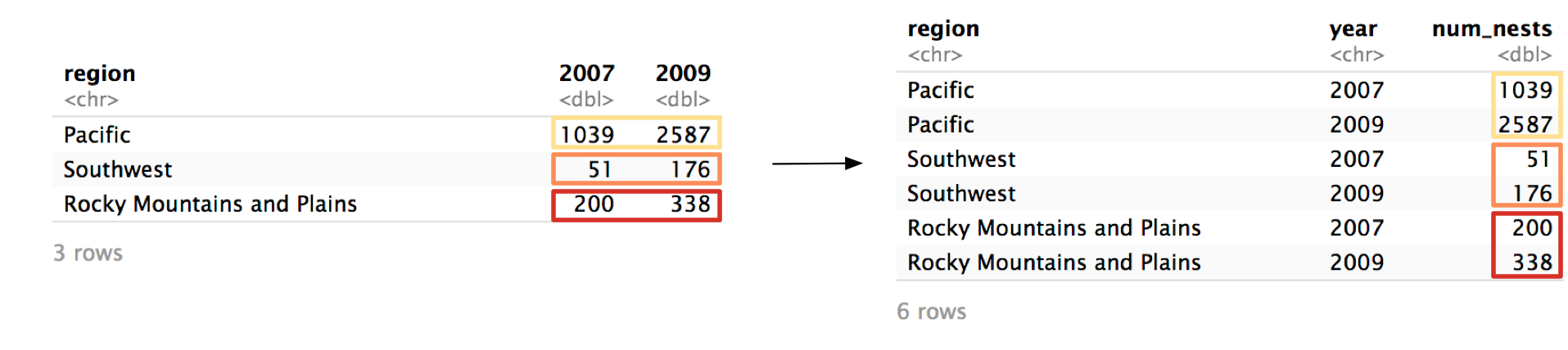
names\_to = "year",

values\_to = "num\_nests"

)

Again, notice that you have to quote any argument to values\_to because it references a column that does not exist.

pivot\_longer() will now move the values from 2007 and 2009 to a column called num\_nests.



Here’s the function call again with the results.

example\_eagle\_nests %>%

pivot\_longer(

cols = c(`2007`, `2009`),

names\_to = "year",

values\_to = "num\_nests"

)

*#> # A tibble: 6 × 3*

*#> region year num\_nests*

*#> <chr> <chr> <dbl>*

*#> 1 Pacific 2007 1039*

*#> 2 Pacific 2009 2587*

*#> 3 Southwest 2007 51*

*#> 4 Southwest 2009 176*

*#> 5 Rocky Mountains and Plains 2007 200*

*#> 6 Rocky Mountains and Plains 2009 338*

The data is now tidy! pivot\_longer() has many optional arguments, but cols, names\_to, and values\_to will cover most of your use-cases. The Missing values section below and the Advanced pivoting chapter cover some more specialized uses of pivot\_longer().

Here’s another eagle-related example. example\_eagle\_pairs contains data on the number of observed [bald eagle breeding pairs](https://www.fws.gov/midwest/eagle/NestingData/nos_state_tbl.html) across years and states.

example\_eagle\_pairs

*#> # A tibble: 48 × 12*

*#> state state\_abbr `1997` `1998` `1999` `2000` `2001` `2002` `2003` `2004`*

*#> <chr> <chr> <int> <int> <int> <int> <int> <int> <int> <int>*

*#> 1 Alabama AL 22 23 26 27 NA NA 47 NA*

*#> 2 Arizona AZ 34 36 38 37 37 43 43 NA*

*#> 3 Arkansas AR 24 29 34 36 NA NA 36 42*

*#> 4 California CA 142 148 151 NA NA NA 160 NA*

*#> 5 Colorado CO 29 27 29 42 45 NA NA NA*

*#> 6 Connecticut CT 2 2 2 4 6 8 8 NA*

*#> # … with 42 more rows, and 2 more variables: 2005 <int>, 2006 <int>*

Again, the data isn’t tidy because values are spread across column names. We need to pivot all the year columns (1997 through 2006), moving their names into a column named "year" and their values into a column named "num\_nests".

Here’s the full call to pivot\_longer():

example\_eagle\_pairs %>%

pivot\_longer(

cols = !starts\_with("state"),

names\_to = "year",

values\_to = "num\_pairs"

)

*#> # A tibble: 480 × 4*

*#> state state\_abbr year num\_pairs*

*#> <chr> <chr> <chr> <int>*

*#> 1 Alabama AL 1997 22*

*#> 2 Alabama AL 1998 23*

*#> 3 Alabama AL 1999 26*

*#> 4 Alabama AL 2000 27*

*#> 5 Alabama AL 2001 NA*

*#> 6 Alabama AL 2002 NA*

*#> # … with 474 more rows*

## 5.2 Wider

pivot\_wider() is the inverse of pivot\_longer(). pivot\_longer() moves data from column names to cell values, while pivot\_wider() pulls data from cell values into column names, creating a wider tibble.

You’ll likely use pivot\_longer() more often than pivot\_wider() when tidying. Often, you’ll actually use pivot\_wider() to un-tidy data. The non-tidy format may be more convenient for some tasks (e.g., creating a specific visualization).

To explain pivot\_wider(), we’ll turn the tidied example\_eagle\_nests back into its original form. Here’s the tidied version:

example\_eagle\_nests\_tidy

*#> # A tibble: 6 × 3*

*#> region year num\_nests*

*#> <chr> <chr> <dbl>*

*#> 1 Pacific 2007 1039*

*#> 2 Pacific 2009 2587*

*#> 3 Southwest 2007 51*

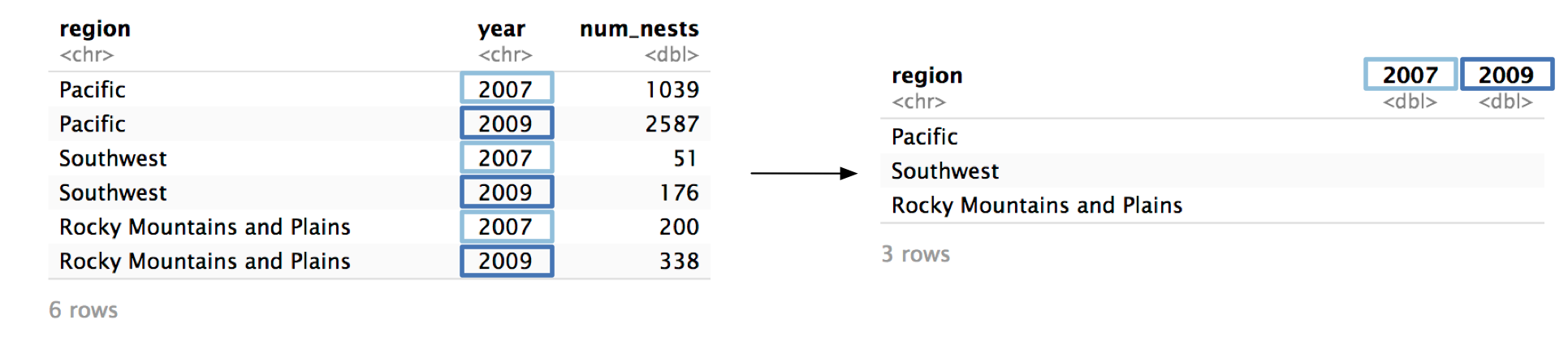
*#> 4 Southwest 2009 176*

*#> 5 Rocky Mountains and Plains 2007 200*

*#> 6 Rocky Mountains and Plains 2009 338*

**Step 1** Identify the column whose values will supply the column names.

pivot\_wider() turns the values from one column and turns them into column names. In our example, we want the unique values from year to become column names.



pivot\_wider()’s names\_from argument controls which column is pivoted into column names.

example\_eagle\_nests\_tidy %>%

pivot\_wider(

names\_from = year,

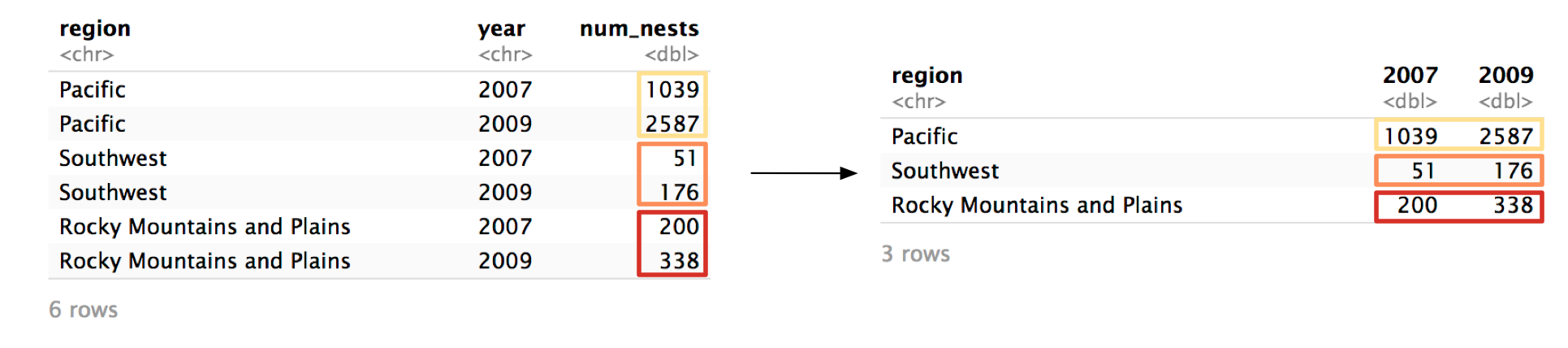
*# Step 2*

)

Notice that year is unquoted because, following the rule, year does exist in example\_eagle\_nests\_tidy.

**Step 2** Identify the column whose values will supply the column values.

Now, we need to identify the column that will supply the values of 2007 and 2009. In example\_eagle\_nests\_tidy, that’s num\_nests.



We specify num\_nests as the values\_from argument.

example\_eagle\_nests\_tidy %>%

pivot\_wider(

names\_from = year,

values\_from = num\_nests

)

*#> # A tibble: 3 × 3*

*#> region `2007` `2009`*

*#> <chr> <dbl> <dbl>*

*#> 1 Pacific 1039 2587*

*#> 2 Southwest 51 176*

*#> 3 Rocky Mountains and Plains 200 338*

Again, supply the name of the column unquoted.

We’re done! The tibble is now transformed back into its original form.

Let’s see an example of a tibble that actually does need pivot\_wider() to be tidy. example\_acs\_1 contains data from the 2013-2017 [American Community Survey](https://www.census.gov/programs-surveys/acs), obtained through the [tidycensus](https://walkerke.github.io/tidycensus/) package.

example\_acs\_1

*#> # A tibble: 156 × 4*

*#> geoid name variable estimate*

*#> <chr> <chr> <chr> <dbl>*

*#> 1 01 Alabama pop\_housed 4731852*

*#> 2 01 Alabama pop\_renter 1434765*

*#> 3 01 Alabama median\_rent 747*

*#> 4 02 Alaska pop\_housed 710743*

*#> 5 02 Alaska pop\_renter 241484*

*#> 6 02 Alaska median\_rent 1200*

*#> # … with 150 more rows*

variable and estimate are not really variables (if you see a variable named variable it’s a good sign you need pivot\_wider()). There are three distinct values in variable:

example\_acs\_1 %>%

distinct(variable)

*#> # A tibble: 3 × 1*

*#> variable*

*#> <chr>*

*#> 1 pop\_housed*

*#> 2 pop\_renter*

*#> 3 median\_rent*

Each of these values is actually a variable whose values are currently stored in estimate. To pivot, we’ll set names\_from to variable and values\_from to estimate.

example\_acs\_1 %>%

pivot\_wider(names\_from = variable, values\_from = estimate)

*#> # A tibble: 52 × 5*

*#> geoid name pop\_housed pop\_renter median\_rent*

*#> <chr> <chr> <dbl> <dbl> <dbl>*

*#> 1 01 Alabama 4731852 1434765 747*

*#> 2 02 Alaska 710743 241484 1200*

*#> 3 04 Arizona 6656124 2460534 972*

*#> 4 05 Arkansas 2894098 965690 709*

*#> 5 06 California 38168482 17066023 1358*

*#> 6 08 Colorado 5318396 1782975 1125*

*#> # … with 46 more rows*

## 5.3 Missing values

The United Nations compiles [data](https://www.un.org/en/development/desa/population/migration/data/estimates2/estimates19.asp) on the origin and destination countries of international migrants. example\_migration contains a subset of this data from 2017. The countries in the column names represent countries of origin, and the countries in dest represent destination countries.

example\_migration

*#> # A tibble: 3 × 6*

*#> dest Afghanistan Canada India Japan `South Africa`*

*#> <chr> <chr> <chr> <chr> <chr> <chr>*

*#> 1 Albania <NA> 913 <NA> <NA> <NA>*

*#> 2 Bulgaria 483 713 281 213 260*

*#> 3 Romania <NA> <NA> 102 <NA> <NA>*

Again, the data isn’t tidy. Afghanistan, Canada, etc. are values of a variable, not variables themselves. We can use pivot\_longer() to tidy the data.

example\_migration %>%

pivot\_longer(cols = !dest, names\_to = "origin", values\_to = "migrants")

*#> # A tibble: 15 × 3*

*#> dest origin migrants*

*#> <chr> <chr> <chr>*

*#> 1 Albania Afghanistan <NA>*

*#> 2 Albania Canada 913*

*#> 3 Albania India <NA>*

*#> 4 Albania Japan <NA>*

*#> 5 Albania South Africa <NA>*

*#> 6 Bulgaria Afghanistan 483*

*#> # … with 9 more rows*

There are a lot of NAs in the data. However, they don’t actually represent missing values. Someone didn’t forget to measure the number of migrants Afghanistan to Albania—there just weren’t any. It doesn’t really make sense to include these rows in our new, tidied dataset. We can use values\_drop\_na to exclude these rows.

example\_migration %>%

pivot\_longer(

cols = !dest,

names\_to = "origin",

values\_to = "migrants",

values\_drop\_na = TRUE

)

*#> # A tibble: 7 × 3*

*#> dest origin migrants*

*#> <chr> <chr> <chr>*

*#> 1 Albania Canada 913*

*#> 2 Bulgaria Afghanistan 483*

*#> 3 Bulgaria Canada 713*

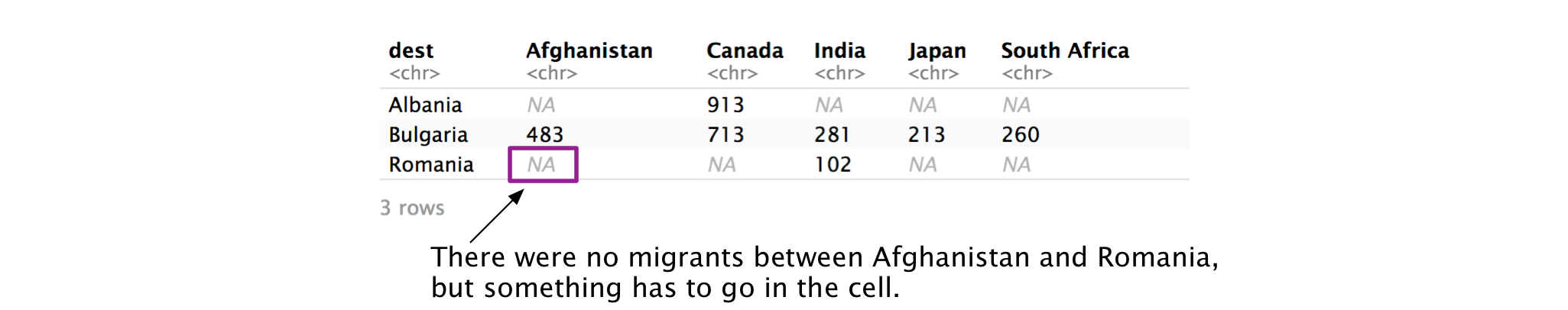
*#> 4 Bulgaria India 281*

*#> 5 Bulgaria Japan 213*

*#> 6 Bulgaria South Africa 260*

*#> # … with 1 more row*

When you use values\_drop\_na = TRUE in pivot\_longer(), you’re turning explicit missing values into implicit missing values. This is only a good idea if the NAs were in the non-tidy data for a purely structural reason, like in example\_migration.



In contrast, example\_eagle\_pairs’s NAs aren’t structural and represent actual missing data. Paired Alabamian eagles probably existed in 2001, but the data isn’t there.

example\_eagle\_pairs

*#> # A tibble: 48 × 12*

*#> state state\_abbr `1997` `1998` `1999` `2000` `2001` `2002` `2003` `2004`*

*#> <chr> <chr> <int> <int> <int> <int> <int> <int> <int> <int>*

*#> 1 Alabama AL 22 23 26 27 NA NA 47 NA*

*#> 2 Arizona AZ 34 36 38 37 37 43 43 NA*

*#> 3 Arkansas AR 24 29 34 36 NA NA 36 42*

*#> 4 California CA 142 148 151 NA NA NA 160 NA*

*#> 5 Colorado CO 29 27 29 42 45 NA NA NA*

*#> 6 Connecticut CT 2 2 2 4 6 8 8 NA*

*#> # … with 42 more rows, and 2 more variables: 2005 <int>, 2006 <int>*

If we used values\_drop\_na = TRUE when we pivoted example\_eagle\_pairs, we would turn all these explicit missing values implicit, which isn’t a good idea.

<https://dcl-wrangle.stanford.edu/pivot-advanced.html>

# What is data transformation: definition, benefits, and uses

Analyzing information requires structured and accessible data for best results. Data transformation enables organizations to alter the structure and format of raw data as needed. Learn how your enterprise can transform its data to perform analytics efficiently.

## What is data transformation?

Data transformation is the process of changing the format, structure, or values of data. For data analytics projects, data may be transformed at two stages of the data pipeline. Organizations that use on-premises data warehouses generally use an ETL ([**extract, transform, load**](https://www.stitchdata.com/resources/glossary/etl/)) process, in which [**data transformation is the middle step**](https://www.stitchdata.com/etldatabase/etl-transform/). Today, most organizations use cloud-based data warehouses, which can scale compute and storage resources with latency measured in seconds or minutes. The scalability of the cloud platform lets organizations skip preload transformations and load raw data into the data warehouse, then transform it at query time — a model called ELT ( [**extract, load, transform)**](https://www.stitchdata.com/resources/what-is-elt/).

Processes such as [**data integration**](https://www.stitchdata.com/resources/glossary/data-integration/), [**data migration**](https://www.stitchdata.com/resources/glossary/data-migration/), [**data warehousing**](https://www.stitchdata.com/resources/glossary/data-warehouse/), and [**data wrangling**](https://www.stitchdata.com/resources/glossary/data-wrangling/) all may involve data transformation.

Data transformation may be constructive (adding, copying, and replicating data), destructive (deleting fields and records), aesthetic (standardizing salutations or street names), or structural (renaming, moving, and combining columns in a database).

An enterprise can choose among a variety of [**ETL tools**](https://www.stitchdata.com/etldatabase/etl-tools/) that automate the process of data transformation. Data analysts, data engineers, and data scientists also transform data using [**scripting languages such as Python**](https://towardsdatascience.com/python-data-transformation-tools-for-etl-2cb20d76fcd0) or [**domain-specific languages like SQL**](https://towardsdatascience.com/python-vs-sql-comparison-for-data-pipelines-8ca727b34032).

## Benefits and challenges of data transformation

Transforming data yields several benefits:

* Data is transformed to make it better-organized. Transformed data may be easier for both humans and computers to use.
* Properly formatted and validated data improves data quality and protects applications from potential landmines such as null values, unexpected duplicates, incorrect indexing, and incompatible formats.
* Data transformation facilitates compatibility between applications, systems, and types of data. Data used for multiple purposes may need to be transformed in different ways..

However, there are challenges to transforming data effectively:

* Data transformation can be expensive. The cost is dependent on the specific infrastructure, software, and tools used to process data. Expenses may include those related to licensing, computing resources, and hiring necessary personnel.
* Data transformation processes can be resource-intensive. Performing transformations in an on-premises data warehouse after loading, or transforming data before feeding it into applications, can create a computational burden that slows down other operations. If you use a cloud-based data warehouse, you can do the transformations after loading because the platform can scale up to meet demand.
* Lack of expertise and carelessness can introduce problems during transformation. Data analysts without appropriate subject matter expertise are less likely to notice typos or incorrect data because they are less familiar with the range of accurate and permissible values. For example, someone working on medical data who is unfamiliar with relevant terms might fail to flag disease names that should be mapped to a singular value or notice misspellings.
* Enterprises can perform transformations that don’t suit their needs. A business might change information to a specific format for one application only to then revert the information back to its prior format for a different application.

## How to transform data

Data transformation can increase the efficiency of analytic and business processes and enable better data-driven decision-making. The first phase of data transformations should include things like data type conversion and flattening of hierarchical data. These operations shape data to increase compatibility with analytics systems. Data analysts and data scientists can implement further transformations additively as necessary as [**individual layers of processing**](https://www.stitchdata.com/blog/transformation-layer-data-modeling/). Each layer of processing should be designed to perform a specific set of tasks that meet a known business or technical requirement.

Data transformation serves many functions within the data analytics stack.

### Extraction and parsing

In the modern ELT process, data ingestion begins with extracting information from a data source, followed by copying the data to its destination. Initial transformations are focused on shaping the format and structure of data to ensure its compatibility with both the destination system and the data already there. Parsing fields out of comma-delimited log data for loading to a relational database is an example of this type of data transformation.

### Translation and mapping

Some of the most basic data transformations involve the mapping and translation of data. For example, a column containing integers representing error codes can be mapped to the relevant error descriptions, making that column easier to understand and more useful for display in a customer-facing application.

Translation converts data from formats used in one system to formats appropriate for a different system. Even after parsing, web data might arrive in the form of hierarchical JSON or XML files, but need to be translated into row and column data for inclusion in a relational database.

### Filtering, aggregation, and summarization

Data transformation is often concerned with whittling data down and making it more manageable. Data may be consolidated by filtering out unnecessary fields, columns, and records. Omitted data might include numerical indexes in data intended for graphs and dashboards or records from business regions that aren’t of interest in a particular study.

Data might also be aggregated or summarized. by, for instance, transforming a time series of customer transactions to hourly or daily sales counts.

BI tools can do this filtering and aggregation, but it can be more efficient to do the transformations before a reporting tool accesses the data.

### Enrichment and imputation

Data from different sources can be merged to create denormalized, enriched information. A customer’s transactions can be rolled up into a grand total and added into a customer information table for quicker reference or for use by customer analytics systems. Long or freeform fields may be split into multiple columns, and missing values can be imputed or corrupted data replaced as a result of these kinds of transformations.

### Indexing and ordering

Data can be transformed so that it’s ordered logically or to suit a data storage scheme. In relational database management systems, for example, creating indexes can improve performance or improve the management of relationships between different tables.

### Anonymization and encryption

Data containing personally identifiable information, or other information that could compromise privacy or security, should be anonymized before propagation. Encryption of private data is a requirement in many industries, and systems can perform encryption at multiple levels, from individual database cells to entire records or fields.

### Modeling, typecasting, formatting, and renaming

Finally, a whole set of transformations can reshape data without changing content. This includes casting and converting data types for compatibility, adjusting dates and times with offsets and format localization, and renaming schemas, tables, and columns for clarity.

## Refining the data transformation process

Before your enterprise can run analytics, and even before you transform the data, you must replicate it to a data warehouse architected for analytics. Most organizations today choose a cloud data warehouse, allowing them to take full advantage of ELT. Stitch can load all of your data to your [**preferred data warehouse**](https://www.stitchdata.com/integrations/destinations/) in a raw state, ready for transformation. [**Try Stitch**](https://www.stitchdata.com/signup/) for free.

* [A first look through your data](https://rstudio-pubs-static.s3.amazonaws.com/369629_d807323ee988436ea6b4d4bab5647047.html#a-first-look-through-your-data)
* [Strings and stringr](https://rstudio-pubs-static.s3.amazonaws.com/369629_d807323ee988436ea6b4d4bab5647047.html#strings-and-stringr)
* [Basic operations on tables](https://rstudio-pubs-static.s3.amazonaws.com/369629_d807323ee988436ea6b4d4bab5647047.html#basic-operations-on-tables)
  + [Selecting data](https://rstudio-pubs-static.s3.amazonaws.com/369629_d807323ee988436ea6b4d4bab5647047.html#selecting-data)
  + [Excluding data](https://rstudio-pubs-static.s3.amazonaws.com/369629_d807323ee988436ea6b4d4bab5647047.html#excluding-data)
  + [Adding data](https://rstudio-pubs-static.s3.amazonaws.com/369629_d807323ee988436ea6b4d4bab5647047.html#adding-data)
* [Tidyverse](https://rstudio-pubs-static.s3.amazonaws.com/369629_d807323ee988436ea6b4d4bab5647047.html#tidyverse)
  + [The wide-long Dillema](https://rstudio-pubs-static.s3.amazonaws.com/369629_d807323ee988436ea6b4d4bab5647047.html#the-wide-long-dillema)
  + [tidyr](https://rstudio-pubs-static.s3.amazonaws.com/369629_d807323ee988436ea6b4d4bab5647047.html#tidyr)
  + [dplyr](https://rstudio-pubs-static.s3.amazonaws.com/369629_d807323ee988436ea6b4d4bab5647047.html#dplyr)
    - [dplyr::joins](https://rstudio-pubs-static.s3.amazonaws.com/369629_d807323ee988436ea6b4d4bab5647047.html#dplyrjoins)
* [Apply family](https://rstudio-pubs-static.s3.amazonaws.com/369629_d807323ee988436ea6b4d4bab5647047.html#apply-family)
* [References](https://rstudio-pubs-static.s3.amazonaws.com/369629_d807323ee988436ea6b4d4bab5647047.html#references)

Data wrangling is a very time-consuming and sometimes, really hard task that you have to perform on your data to get the results you want. Data can come for different sources, in several formats, it is incomplete, it is unstructured… It is basically a mess and in order to extract some valuable information out of it, in order to construct knowledge out of it, you have to tidy it up make it uniform, according to the type of analysis that will follow.

# A first look through your data

A first look through your data will start with head(). It allows you to take a look at the first 6 rows of your data. In addition, tail() will give you the last 6 records. First, I will construct a dummy example for this purpose:

tab\_data <- **data.frame**("x"=**rnorm**(10, 60, 100), "variable1"=**seq**(1:10))

**names**(tab\_data) <- **c**("Latitude conventional", "Logitude separator")

tab\_data

## Latitude conventional Logitude separator

## 1 22.45927 1

## 2 -92.05847 2

## 3 74.58678 3

## 4 193.17415 4

## 5 59.32055 5

## 6 86.53750 6

## 7 96.25202 7

## 8 -242.93970 8

## 9 239.52399 9

## 10 136.72346 10

Now, take a look at its head and tail:

**head**(tab\_data)

## Latitude conventional Logitude separator

## 1 22.45927 1

## 2 -92.05847 2

## 3 74.58678 3

## 4 193.17415 4

## 5 59.32055 5

## 6 86.53750 6

**tail**(tab\_data)

## Latitude conventional Logitude separator

## 5 59.32055 5

## 6 86.53750 6

## 7 96.25202 7

## 8 -242.93970 8

## 9 239.52399 9

## 10 136.72346 10

Another function you can apply to your data is the function str(). This function will give you an idea of rows/columns, how many variables you have and their classification, and some data samples of each variable.

**str**(tab\_data)

## 'data.frame': 10 obs. of 2 variables:

## $ Latitude conventional: num 22.5 -92.1 74.6 193.2 59.3 ...

## $ Logitude separator : int 1 2 3 4 5 6 7 8 9 10

Another aspect to have into consideration are the names of the variables you are dealing with. Once you import data, variable names that contain special characters or spaces can completely drag you down with a million problems. They should be “cleaned” - no spaces or special characters.

The variable names on the table have a space that could give you trouble in your analysis. But R has function that will tidy them up, and prevent future possible problems - make.names()

**names**(tab\_data) <- **make.names**(**names**(tab\_data))

tab\_data[1,]

## Latitude.conventional Logitude.separator

## 1 22.45927 1

If you’re looking for a pattern, take a look at the function table(). It will count each factor for a variable.  
Let’s take as an example the mtcars dataset:

**table**(mtcars$cyl)

##

## 4 6 8

## 11 7 14

**table**(mtcars$gear)

##

## 3 4 5

## 15 12 5

table() can be used as a technique to find duplicates, when appropriate, but you can also use the funtion duplicated(). It returns a logical vector that will point TRUE whenever there is a duplicate. If you intend to remove duplicates directly, you can use the function unique().

# Strings and stringr

**require**(stringr)

stringr is a package in R that has “Simple, Consistent Wrappers for Common String Operations” (Hadley\_b Wickham 2017). It might seem that you won’t need it, but it is quite important, specially when the data you’re receiving has been inputed by different people, using different methods, and you have to uniformize it.

There are four main families of functions in stringr:

* Character Manipulation functions;
* Whitespaces manipulation functions;
* Locale sensitive functions and
* Pattern matching functions.

You can get the lenght of any string using str\_length().

a<-"Hello World"

**str\_length**(a)

## [1] 11

You can substitute characters in a string by something your want, or nada de nada. The first argument is your string, the second the starting position and the third is the last position.

**str\_sub**(a, 1, 5) *# to keep the string specified within your arguments*

## [1] "Hello"

**str\_sub**(a, 7, 11)<-"Pluto" *#substitute "world" by "pluto"*

a

## [1] "Hello Pluto"

a<-**c**("Hello World", "Goodbye World")

**str\_sub**(a, 1,6)

## [1] "Hello " "Goodby"

**str\_sub**(a, -5, -1)

## [1] "World" "World"

There is also a function from base that I really like which is gsub. It is quite usefull as well, in case that, for example, you need to supress whitespaces inside your variables and substitute them by “\_“.

a<-"I am Angela"

**gsub**(" ", "\_", a)

## [1] "I\_am\_Angela"

Regarding whitespaces, it is worth to mention str\_pad() that adds extra whitespaces either on the left, or right, or on both sides; str\_trim() will remove leading and trailing whitespaces. This can become quite usefull in our dataset, where in certain classes there is a leading whitespace at the beginning of each level. E.g.:

df<-**data.frame**(x=**c**(" FOO"," baR"), y= **c**("blUe ", "GREy "))

**str\_trim**(df$x)

## [1] "FOO" "baR"

Now, let’s apply this function to the rest of the table:

dftrim<-**as.data.frame**(**lapply**(df, str\_trim))

**levels**(dftrim$y)

## [1] "blUe" "GREy"

More on the lapply() function will be given further on.

To change cases, there are functions such as str\_to\_upper(), str\_to\_title() and str\_to\_lower(). You can also order strings, str\_order() or sort them with str\_sort().

For example, in our case data, I believe it would be better to lower all cases. Therefore,

dftrim<-**as.data.frame**(**lapply**(dftrim, str\_to\_lower))

dftrim

## x y

## 1 foo blue

## 2 bar grey

More information about string patterns can be found [here](https://cran.r-project.org/web/packages/stringr/vignettes/stringr.html).

# Basic operations on tables

One of the most basic table operations you can do is to transpose the table. You can do so by simply using the function t().

dftrimt<-**as.data.frame**(**t**(dftrim))

## Selecting data

Selection of data can be done in a couple of ways. Let’s say you want to select rows 7:9, or columns 1 and 2:

mtcars[7:9,]

## mpg cyl disp hp drat wt qsec vs am gear carb

## Duster 360 14.3 8 360.0 245 3.21 3.57 15.84 0 0 3 4

## Merc 240D 24.4 4 146.7 62 3.69 3.19 20.00 1 0 4 2

## Merc 230 22.8 4 140.8 95 3.92 3.15 22.90 1 0 4 2

mtcars[**c**(7,8,9),]

## mpg cyl disp hp drat wt qsec vs am gear carb

## Duster 360 14.3 8 360.0 245 3.21 3.57 15.84 0 0 3 4

## Merc 240D 24.4 4 146.7 62 3.69 3.19 20.00 1 0 4 2

## Merc 230 22.8 4 140.8 95 3.92 3.15 22.90 1 0 4 2

**head**(mtcars[,1:2])

## mpg cyl

## Mazda RX4 21.0 6

## Mazda RX4 Wag 21.0 6

## Datsun 710 22.8 4

## Hornet 4 Drive 21.4 6

## Hornet Sportabout 18.7 8

## Valiant 18.1 6

**head**(mtcars[, **c**(1,2)])

## mpg cyl

## Mazda RX4 21.0 6

## Mazda RX4 Wag 21.0 6

## Datsun 710 22.8 4

## Hornet 4 Drive 21.4 6

## Hornet Sportabout 18.7 8

## Valiant 18.1 6

You can also subset your data according to the values of your variables. E.g., let’s say you need to select all cars with “cyl” equal to 6. You have several ways to do it:

mtcars[**which**(mtcars$cyl == 6), ]

## mpg cyl disp hp drat wt qsec vs am gear carb

## Mazda RX4 21.0 6 160.0 110 3.90 2.620 16.46 0 1 4 4

## Mazda RX4 Wag 21.0 6 160.0 110 3.90 2.875 17.02 0 1 4 4

## Hornet 4 Drive 21.4 6 258.0 110 3.08 3.215 19.44 1 0 3 1

## Valiant 18.1 6 225.0 105 2.76 3.460 20.22 1 0 3 1

## Merc 280 19.2 6 167.6 123 3.92 3.440 18.30 1 0 4 4

## Merc 280C 17.8 6 167.6 123 3.92 3.440 18.90 1 0 4 4

## Ferrari Dino 19.7 6 145.0 175 3.62 2.770 15.50 0 1 5 6

**subset**(mtcars, mtcars$cyl ==6)

## mpg cyl disp hp drat wt qsec vs am gear carb

## Mazda RX4 21.0 6 160.0 110 3.90 2.620 16.46 0 1 4 4

## Mazda RX4 Wag 21.0 6 160.0 110 3.90 2.875 17.02 0 1 4 4

## Hornet 4 Drive 21.4 6 258.0 110 3.08 3.215 19.44 1 0 3 1

## Valiant 18.1 6 225.0 105 2.76 3.460 20.22 1 0 3 1

## Merc 280 19.2 6 167.6 123 3.92 3.440 18.30 1 0 4 4

## Merc 280C 17.8 6 167.6 123 3.92 3.440 18.90 1 0 4 4

## Ferrari Dino 19.7 6 145.0 175 3.62 2.770 15.50 0 1 5 6

Now, let’s a look at another example. Let’s select cyl=6 and carburator 4 or less. You can also select all cars with 6 cilinders plus all cars with carburator less than 4. Take into consideration that in the first case it is used & and the second |. They mean “AND” and “OR” respectively.

mtcars[**which**(mtcars$cyl== 6 & mtcars$carb <=4), ]

## mpg cyl disp hp drat wt qsec vs am gear carb

## Mazda RX4 21.0 6 160.0 110 3.90 2.620 16.46 0 1 4 4

## Mazda RX4 Wag 21.0 6 160.0 110 3.90 2.875 17.02 0 1 4 4

## Hornet 4 Drive 21.4 6 258.0 110 3.08 3.215 19.44 1 0 3 1

## Valiant 18.1 6 225.0 105 2.76 3.460 20.22 1 0 3 1

## Merc 280 19.2 6 167.6 123 3.92 3.440 18.30 1 0 4 4

## Merc 280C 17.8 6 167.6 123 3.92 3.440 18.90 1 0 4 4

**head**(mtcars[**which**(mtcars$cyl== 6 | mtcars$carb <=4), ])

## mpg cyl disp hp drat wt qsec vs am gear carb

## Mazda RX4 21.0 6 160 110 3.90 2.620 16.46 0 1 4 4

## Mazda RX4 Wag 21.0 6 160 110 3.90 2.875 17.02 0 1 4 4

## Datsun 710 22.8 4 108 93 3.85 2.320 18.61 1 1 4 1

## Hornet 4 Drive 21.4 6 258 110 3.08 3.215 19.44 1 0 3 1

## Hornet Sportabout 18.7 8 360 175 3.15 3.440 17.02 0 0 3 2

## Valiant 18.1 6 225 105 2.76 3.460 20.22 1 0 3 1

**subset**(mtcars, mtcars$cyl== 6 & mtcars$carb <=4)

## mpg cyl disp hp drat wt qsec vs am gear carb

## Mazda RX4 21.0 6 160.0 110 3.90 2.620 16.46 0 1 4 4

## Mazda RX4 Wag 21.0 6 160.0 110 3.90 2.875 17.02 0 1 4 4

## Hornet 4 Drive 21.4 6 258.0 110 3.08 3.215 19.44 1 0 3 1

## Valiant 18.1 6 225.0 105 2.76 3.460 20.22 1 0 3 1

## Merc 280 19.2 6 167.6 123 3.92 3.440 18.30 1 0 4 4

## Merc 280C 17.8 6 167.6 123 3.92 3.440 18.90 1 0 4 4

**head**(**subset**(mtcars, mtcars$cyl== 6 | mtcars$carb <=4))

## mpg cyl disp hp drat wt qsec vs am gear carb

## Mazda RX4 21.0 6 160 110 3.90 2.620 16.46 0 1 4 4

## Mazda RX4 Wag 21.0 6 160 110 3.90 2.875 17.02 0 1 4 4

## Datsun 710 22.8 4 108 93 3.85 2.320 18.61 1 1 4 1

## Hornet 4 Drive 21.4 6 258 110 3.08 3.215 19.44 1 0 3 1

## Hornet Sportabout 18.7 8 360 175 3.15 3.440 17.02 0 0 3 2

## Valiant 18.1 6 225 105 2.76 3.460 20.22 1 0 3 1

With the subset function, you can also select the columns you want on the output.

**subset**(mtcars, mtcars$cyl== 6 & mtcars$carb <=4, select = **c**("cyl", "wt"))

## cyl wt

## Mazda RX4 6 2.620

## Mazda RX4 Wag 6 2.875

## Hornet 4 Drive 6 3.215

## Valiant 6 3.460

## Merc 280 6 3.440

## Merc 280C 6 3.440

## Excluding data

This process is also referred to as dropping data.

**head**(mtcars[-1,]) *# exclude first row*

## mpg cyl disp hp drat wt qsec vs am gear carb

## Mazda RX4 Wag 21.0 6 160 110 3.90 2.875 17.02 0 1 4 4

## Datsun 710 22.8 4 108 93 3.85 2.320 18.61 1 1 4 1

## Hornet 4 Drive 21.4 6 258 110 3.08 3.215 19.44 1 0 3 1

## Hornet Sportabout 18.7 8 360 175 3.15 3.440 17.02 0 0 3 2

## Valiant 18.1 6 225 105 2.76 3.460 20.22 1 0 3 1

## Duster 360 14.3 8 360 245 3.21 3.570 15.84 0 0 3 4

**head**(mtcars[,-1]) *#exclude first column*

## cyl disp hp drat wt qsec vs am gear carb

## Mazda RX4 6 160 110 3.90 2.620 16.46 0 1 4 4

## Mazda RX4 Wag 6 160 110 3.90 2.875 17.02 0 1 4 4

## Datsun 710 4 108 93 3.85 2.320 18.61 1 1 4 1

## Hornet 4 Drive 6 258 110 3.08 3.215 19.44 1 0 3 1

## Hornet Sportabout 8 360 175 3.15 3.440 17.02 0 0 3 2

## Valiant 6 225 105 2.76 3.460 20.22 1 0 3 1

**head**(mtcars[ , -**c**(1:3)]) *#exclude columns 1:3*

## hp drat wt qsec vs am gear carb

## Mazda RX4 110 3.90 2.620 16.46 0 1 4 4

## Mazda RX4 Wag 110 3.90 2.875 17.02 0 1 4 4

## Datsun 710 93 3.85 2.320 18.61 1 1 4 1

## Hornet 4 Drive 110 3.08 3.215 19.44 1 0 3 1

## Hornet Sportabout 175 3.15 3.440 17.02 0 0 3 2

## Valiant 105 2.76 3.460 20.22 1 0 3 1

## Adding data

If you want to merge data horizontally, by key-value columns, check the package dplyr in the next sections. If you want to merge two datatables vertically, you can use rbind, as long as the tables have the same variables (columns).

x<-**data.frame**(x=1:5, y=2:6)

y<-**data.frame**(x=2:10, y=20)

xy=**rbind**(x,y)

# Tidyverse

Tidyverse (Wickham 2016) is a group of packages that share the same design philosophy, grammar and data structures. The packages that are part of this universe are dplyr (Wickham and Francois 2016), ggplot2 (Wickham and Chang 2016), readr (Wickham, Hester, and Francois 2016), tibble (Wickham, Francois, and Muller 2016), tidyr (Hadley\_a Wickham 2017) and purrr (Henry and Wickham 2017).

## The wide-long Dillema

A wide dataframe format is a dataframe that has a column for each variable, such as the table we’ve been using so far. A long format dataframe is a dataframe that has a column for all variables “stacked” into a column and another column has the values.

A wide dataframe looks like:

## SO4 Ca

## 1 0.1 0.5

## 2 0.2 0.6

## 3 0.3 0.7

The same dataframe on the long format looks like:

## variable value

## 1 SO4 0.1

## 2 SO4 0.2

## 3 SO4 0.3

## 4 Ca 0.5

## 5 Ca 0.6

## 6 Ca 0.7

Now this requires a little thinking about your data, and what are the “x’s” of your data and which are the “y’s”. Sometimes you’ll need long format, and other times, you’ll need wide format. It only depends on what you need to do with your data. ggplot2, plyr, and some modelling functions lm(), glm(), need the long format tables. However, you might find it easy to record your data in the wide-format.

The package reshape helps you to convert between these two formats, and all goes around with two main functions: melt() (from wide to long) and cast (from long to wide).

Let’s take into consideration the mtcars dataset:

**head**(mtcars)

## mpg cyl disp hp drat wt qsec vs am gear carb

## Mazda RX4 21.0 6 160 110 3.90 2.620 16.46 0 1 4 4

## Mazda RX4 Wag 21.0 6 160 110 3.90 2.875 17.02 0 1 4 4

## Datsun 710 22.8 4 108 93 3.85 2.320 18.61 1 1 4 1

## Hornet 4 Drive 21.4 6 258 110 3.08 3.215 19.44 1 0 3 1

## Hornet Sportabout 18.7 8 360 175 3.15 3.440 17.02 0 0 3 2

## Valiant 18.1 6 225 105 2.76 3.460 20.22 1 0 3 1

The “x” for this dataset is each car model, which is the name for each row. For that reason, I will transfer these names as a new column. Now, to transform a dataset from wide to long, you can use the function melt(). It’s always important that you think about the nature of your data. All datasets are different to a certain extent, and that largely depends on the results you’re trying to accomplish. A deep understanding of your data (what are the variables of interest; what are the questions you’re trying to answer what is the relationship between those questions and the variables you have) might be something that you don’t really want to think about, because you’re just eager to get the results. However, “wasting” this time at the beginning of your data analysis, is something that could avoid future bulk mistakes and misconceptions is your analysis. Therefore, whenever you’re melting your data, always use the argument “id.vars”, so you know exaclty what you’re doing with your data.

mtcars$model<-**gsub**(" ", "\_", **row.names**(mtcars))

longmtcars<-**melt**(mtcars, id.vars = "model")

**head**(longmtcars)

## model variable value

## 1 Mazda\_RX4 mpg 21.0

## 2 Mazda\_RX4\_Wag mpg 21.0

## 3 Datsun\_710 mpg 22.8

## 4 Hornet\_4\_Drive mpg 21.4

## 5 Hornet\_Sportabout mpg 18.7

## 6 Valiant mpg 18.1

**tail**(longmtcars)

## model variable value

## 347 Porsche\_914-2 carb 2

## 348 Lotus\_Europa carb 2

## 349 Ford\_Pantera\_L carb 4

## 350 Ferrari\_Dino carb 6

## 351 Maserati\_Bora carb 8

## 352 Volvo\_142E carb 2

The onbject we just created, longmtcars is now a “tidy” dataset according to (Wickham and others 2014). Now, you can convert it back again to wide format. You can use the function dcast() to have a table as an output or you can use acast to have an array or vector as an output. To achieve the conversion to wide format, you’ll need a formula of the type x1+x2∼y1+y2x1+x2∼y1+y2.

wide<-**dcast**(longmtcars, model ~ variable)

**head**(wide)

## model mpg cyl disp hp drat wt qsec vs am gear carb

## 1 AMC\_Javelin 15.2 8 304 150 3.15 3.435 17.30 0 0 3 2

## 2 Cadillac\_Fleetwood 10.4 8 472 205 2.93 5.250 17.98 0 0 3 4

## 3 Camaro\_Z28 13.3 8 350 245 3.73 3.840 15.41 0 0 3 4

## 4 Chrysler\_Imperial 14.7 8 440 230 3.23 5.345 17.42 0 0 3 4

## 5 Datsun\_710 22.8 4 108 93 3.85 2.320 18.61 1 1 4 1

## 6 Dodge\_Challenger 15.5 8 318 150 2.76 3.520 16.87 0 0 3 2

And there you go, your data is back as a wide table.

Examples of other types of tables you can have; note that the one to choose will always be directly related to the type of analysis you’re trying to do. You have to understand which variable is the cause and which is the effect.

**head**(**melt**(mtcars, id.vars = "hp"))

## hp variable value

## 1 110 mpg 21

## 2 110 mpg 21

## 3 93 mpg 22.8

## 4 110 mpg 21.4

## 5 175 mpg 18.7

## 6 105 mpg 18.1

**head**(**melt**(mtcars, id.vars = **c**("hp", "gear")))

## hp gear variable value

## 1 110 4 mpg 21

## 2 110 4 mpg 21

## 3 93 4 mpg 22.8

## 4 110 3 mpg 21.4

## 5 175 3 mpg 18.7

## 6 105 3 mpg 18.1

You can also use dcast() to perform aggregation. Let’s say you want to collect the mean of each variable for each gear. You can do so by:

df<-**melt**(mtcars, id.vars = **c**("model", "gear"))

**head**(df)

## model gear variable value

## 1 Mazda\_RX4 4 mpg 21.0

## 2 Mazda\_RX4\_Wag 4 mpg 21.0

## 3 Datsun\_710 4 mpg 22.8

## 4 Hornet\_4\_Drive 3 mpg 21.4

## 5 Hornet\_Sportabout 3 mpg 18.7

## 6 Valiant 3 mpg 18.1

**dcast**(df, gear ~ variable, mean)

## gear mpg cyl disp hp drat wt qsec

## 1 3 16.10667 7.466667 326.3000 176.1333 3.132667 3.892600 17.692

## 2 4 24.53333 4.666667 123.0167 89.5000 4.043333 2.616667 18.965

## 3 5 21.38000 6.000000 202.4800 195.6000 3.916000 2.632600 15.640

## vs am carb

## 1 0.2000000 0.0000000 2.666667

## 2 0.8333333 0.6666667 2.333333

## 3 0.2000000 1.0000000 4.400000

**dcast**(df, gear~variable) *#counts occurences*

## Aggregation function missing: defaulting to length

## gear mpg cyl disp hp drat wt qsec vs am carb

## 1 3 15 15 15 15 15 15 15 15 15 15

## 2 4 12 12 12 12 12 12 12 12 12 12

## 3 5 5 5 5 5 5 5 5 5 5 5

Note that it pops the message that the lenght is used as the aggregation function, because we did not define the function, which means that the output will give you counts.

However, I believe it is easier to do this operation (aggregation) with dplyr.

Now just as a curiosity, take a look on how to produce boxplots with base and with ggplot2 for the mtcars:

**require**(ggplot2, quietly = TRUE)

**boxplot**(mpg~gear, data=mtcars)

**ggplot**(mtcars, **aes**(x=**as.factor**(gear), y=mpg)) +

**geom\_boxplot**() + **xlab**("cyl") *# one variable against the other*

**ggplot**(longmtcars, **aes** (x=variable, y=value)) +

**geom\_boxplot**() *#all variables*

## tidyr

The tidyr package is a package that allows you to tidy your data easily (Hadley\_a Wickham 2017). It might seem very similar to reshape2, but tidyr is designed mainly to tidy data and not to reshape.

This package offers you four main functions:

* gather ()
* separate ()
* spread ()
* unite ()

Let’s take an example. You measured SO24SO42 levels, and collected that data along a river in three locations, from 2005 to 2008.

**require**(tidyr, quietly = TRUE)

riversulfate<-**data.frame**(Location= (1:3),

y2005 = **c**(0.1, 0.2, 0.1),

y2006= **c**(0.2, 0.1, 0.1),

y2007= **c**(0.5, 0.1, 0.2),

y2008 = **c**(0.2, 0.3, 0.1))

riversulfate

## Location y2005 y2006 y2007 y2008

## 1 1 0.1 0.2 0.5 0.2

## 2 2 0.2 0.1 0.1 0.3

## 3 3 0.1 0.1 0.2 0.1

If you take a look at the data, your rows are not a single observation for each location, but instead in each row you have four different observations for each location. To tidy these data, and keep each observation on a row, you use gather (), where the first argument is your table, second the name you want to call the variable that will “group” several columns into the same column (referred to as “key”), the third the name of the measurement itself, and lastly, the indication for columns to be gathered.

**gather**(riversulfate, year, value, y2005:y2008)

## Location year value

## 1 1 y2005 0.1

## 2 2 y2005 0.2

## 3 3 y2005 0.1

## 4 1 y2006 0.2

## 5 2 y2006 0.1

## 6 3 y2006 0.1

## 7 1 y2007 0.5

## 8 2 y2007 0.1

## 9 3 y2007 0.2

## 10 1 y2008 0.2

## 11 2 y2008 0.3

## 12 3 y2008 0.1

or more simplisticaly, with the pipe notation:

riversulfate %>%

**gather**(year, value, y2005:y2008)

## Location year value

## 1 1 y2005 0.1

## 2 2 y2005 0.2

## 3 3 y2005 0.1

## 4 1 y2006 0.2

## 5 2 y2006 0.1

## 6 3 y2006 0.1

## 7 1 y2007 0.5

## 8 2 y2007 0.1

## 9 3 y2007 0.2

## 10 1 y2008 0.2

## 11 2 y2008 0.3

## 12 3 y2008 0.1

Now, let’s take the same example, but let’s consider that for each location you measured both Na+Na+ and SO24SO42, and merged both variables into “Location”.

**set.seed**(2)

**options**(digits = 2)

riversulfate<-**data.frame**(Location= **c**("1;Na", "1;SO4", "2;Na",

"2;SO4", "3;Na", "3;SO4"),

y2005 = **rnorm**(6, 5),

y2006= **rnorm**(6, 0.2),

y2007= **rnorm**(6, 0.1),

y2008 = **rnorm**(6, 0.5))

**head**(riversulfate)

## Location y2005 y2006 y2007 y2008

## 1 1;Na 4.1 0.908 -0.29 1.51

## 2 1;SO4 5.2 -0.040 -0.94 0.93

## 3 2;Na 6.6 2.184 1.88 2.59

## 4 2;SO4 3.9 0.061 -2.21 -0.70

## 5 3;Na 4.9 0.618 0.98 2.09

## 6 3;SO4 5.1 1.182 0.14 2.45

To tidy this table, you first have to collapse the different years into one column using gather(), and then use separate() to separate the “location” column into two columns. As a result, you’ll have for each column a variable and for each row an observation.

tidy<-**gather**(riversulfate, year, mgL, y2005:y2008)

**head**(tidy)

## Location year mgL

## 1 1;Na y2005 4.1

## 2 1;SO4 y2005 5.2

## 3 2;Na y2005 6.6

## 4 2;SO4 y2005 3.9

## 5 3;Na y2005 4.9

## 6 3;SO4 y2005 5.1

river<-**separate**(tidy, Location, **c**("Location", "element"), sep="\\;")

**head**(river)

## Location element year mgL

## 1 1 Na y2005 4.1

## 2 1 SO4 y2005 5.2

## 3 2 Na y2005 6.6

## 4 2 SO4 y2005 3.9

## 5 3 Na y2005 4.9

## 6 3 SO4 y2005 5.1

If you desire to present an element as a column, you can use spread(). The first argument is your data, second the column you want to separate to several columns (the key), and the third name of the column that holds the values for that key.

tidier<-**spread**(river, element, mgL)

**head**(tidier)

## Location year Na SO4

## 1 1 y2005 4.10 5.185

## 2 1 y2006 0.91 -0.040

## 3 1 y2007 -0.29 -0.940

## 4 1 y2008 1.51 0.932

## 5 2 y2005 6.59 3.870

## 6 2 y2006 2.18 0.061

## dplyr

dplyr is “A fast, consistent tool for working with data frame like objects”" (Wickham and Francois 2016).

The main functions from this package are: \* filter() - selection based on an attribute value \* arrange() - reorder your variables by a variable order (ascending or descending) \* select() and rename() - selecting variables by name \* mutate() - works as a “field calculator” \* summarise() - condenses a variable into one value

Under this context, let’s use again the mtcars dataset. dplyr allows you to filter by attributes. In the following example, I want to extract all cars with gear 5:

**require**(dplyr)

**filter**(mtcars, gear==5)

## mpg cyl disp hp drat wt qsec vs am gear carb model

## 1 26 4 120 91 4.4 2.1 17 0 1 5 2 Porsche\_914-2

## 2 30 4 95 113 3.8 1.5 17 1 1 5 2 Lotus\_Europa

## 3 16 8 351 264 4.2 3.2 14 0 1 5 4 Ford\_Pantera\_L

## 4 20 6 145 175 3.6 2.8 16 0 1 5 6 Ferrari\_Dino

## 5 15 8 301 335 3.5 3.6 15 0 1 5 8 Maserati\_Bora

**filter**(mtcars, gear ==5,

carb!=2 & carb!=3) *#gear 5, carb isnt 2 and 3*

## mpg cyl disp hp drat wt qsec vs am gear carb model

## 1 16 8 351 264 4.2 3.2 14 0 1 5 4 Ford\_Pantera\_L

## 2 20 6 145 175 3.6 2.8 16 0 1 5 6 Ferrari\_Dino

## 3 15 8 301 335 3.5 3.6 15 0 1 5 8 Maserati\_Bora

**filter**(mtcars, gear>=5) *#gear equal or more than 5*

## mpg cyl disp hp drat wt qsec vs am gear carb model

## 1 26 4 120 91 4.4 2.1 17 0 1 5 2 Porsche\_914-2

## 2 30 4 95 113 3.8 1.5 17 1 1 5 2 Lotus\_Europa

## 3 16 8 351 264 4.2 3.2 14 0 1 5 4 Ford\_Pantera\_L

## 4 20 6 145 175 3.6 2.8 16 0 1 5 6 Ferrari\_Dino

## 5 15 8 301 335 3.5 3.6 15 0 1 5 8 Maserati\_Bora

With dplyr, you can also re-arrange your table, by order of a variable you want:

**head**(**arrange**(mtcars, gear))

## mpg cyl disp hp drat wt qsec vs am gear carb model

## 1 21 6 258 110 3.1 3.2 19 1 0 3 1 Hornet\_4\_Drive

## 2 19 8 360 175 3.1 3.4 17 0 0 3 2 Hornet\_Sportabout

## 3 18 6 225 105 2.8 3.5 20 1 0 3 1 Valiant

## 4 14 8 360 245 3.2 3.6 16 0 0 3 4 Duster\_360

## 5 16 8 276 180 3.1 4.1 17 0 0 3 3 Merc\_450SE

## 6 17 8 276 180 3.1 3.7 18 0 0 3 3 Merc\_450SL

**head**(**arrange**(mtcars, vs))

## mpg cyl disp hp drat wt qsec vs am gear carb model

## 1 21 6 160 110 3.9 2.6 16 0 1 4 4 Mazda\_RX4

## 2 21 6 160 110 3.9 2.9 17 0 1 4 4 Mazda\_RX4\_Wag

## 3 19 8 360 175 3.1 3.4 17 0 0 3 2 Hornet\_Sportabout

## 4 14 8 360 245 3.2 3.6 16 0 0 3 4 Duster\_360

## 5 16 8 276 180 3.1 4.1 17 0 0 3 3 Merc\_450SE

## 6 17 8 276 180 3.1 3.7 18 0 0 3 3 Merc\_450SL

**head**(**arrange**(mtcars, **desc**(wt))) *#descending order*

## mpg cyl disp hp drat wt qsec vs am gear carb model

## 1 10 8 460 215 3.0 5.4 18 0 0 3 4 Lincoln\_Continental

## 2 15 8 440 230 3.2 5.3 17 0 0 3 4 Chrysler\_Imperial

## 3 10 8 472 205 2.9 5.2 18 0 0 3 4 Cadillac\_Fleetwood

## 4 16 8 276 180 3.1 4.1 17 0 0 3 3 Merc\_450SE

## 5 19 8 400 175 3.1 3.8 17 0 0 3 2 Pontiac\_Firebird

## 6 13 8 350 245 3.7 3.8 15 0 0 3 4 Camaro\_Z28

You can also select only variables of your interest, e.g.

**head**(**select**(mtcars, model, cyl))

## model cyl

## Mazda RX4 Mazda\_RX4 6

## Mazda RX4 Wag Mazda\_RX4\_Wag 6

## Datsun 710 Datsun\_710 4

## Hornet 4 Drive Hornet\_4\_Drive 6

## Hornet Sportabout Hornet\_Sportabout 8

## Valiant Valiant 6

**head**(**select**(mtcars, -cyl, -wt, -model))

## mpg disp hp drat qsec vs am gear carb

## Mazda RX4 21 160 110 3.9 16 0 1 4 4

## Mazda RX4 Wag 21 160 110 3.9 17 0 1 4 4

## Datsun 710 23 108 93 3.8 19 1 1 4 1

## Hornet 4 Drive 21 258 110 3.1 19 1 0 3 1

## Hornet Sportabout 19 360 175 3.1 17 0 0 3 2

## Valiant 18 225 105 2.8 20 1 0 3 1

**head**(**select**(mtcars, -(disp:carb)))

## mpg cyl model

## Mazda RX4 21 6 Mazda\_RX4

## Mazda RX4 Wag 21 6 Mazda\_RX4\_Wag

## Datsun 710 23 4 Datsun\_710

## Hornet 4 Drive 21 6 Hornet\_4\_Drive

## Hornet Sportabout 19 8 Hornet\_Sportabout

## Valiant 18 6 Valiant

There is other syntaxes you can use (check ?select\_helpers ).

If you want to calculate a new field inside your table, you can use mutate or transmute. Transmute will drop variables (unless you use them as arguments), and mutate will keep them. I will use the Titanic dataset from now on.

Tita<-**as.data.frame**(Titanic)

**head**(Tita)

## Class Sex Age Survived Freq

## 1 1st Male Child No 0

## 2 2nd Male Child No 0

## 3 3rd Male Child No 35

## 4 Crew Male Child No 0

## 5 1st Female Child No 0

## 6 2nd Female Child No 0

Tita<-**mutate**(Tita, perc=Freq/**sum**(Freq)\*100)

**head**(Tita)

## Class Sex Age Survived Freq perc

## 1 1st Male Child No 0 0.0

## 2 2nd Male Child No 0 0.0

## 3 3rd Male Child No 35 1.6

## 4 Crew Male Child No 0 0.0

## 5 1st Female Child No 0 0.0

## 6 2nd Female Child No 0 0.0

**head**(**transmute**(Tita, Freq, perc=Freq/**sum**(Freq)\*100))

## Freq perc

## 1 0 0.0

## 2 0 0.0

## 3 35 1.6

## 4 0 0.0

## 5 0 0.0

## 6 0 0.0

The function summarise() helps you collapsing your variables into a single record, according to a given function - could be sum, or mean, or whatever you feel like.

**summarise**(Tita, Total=**sum**(Freq))*#How many people were sampled?*

## Total

## 1 2201

A very usefull operation is group\_by. With it, you can perform your operations and group your data.

people<-**group\_by**(Tita, Class)

**summarise**(people, **sum**(Freq))*#How many people were sampled by class?*

## # A tibble: 4 × 2

## Class `sum(Freq)`

## <fctr> <dbl>

## 1 1st 325

## 2 2nd 285

## 3 3rd 706

## 4 Crew 885

You can also use pipe notation (from magrittr (Bache and Wickham 2014)), and it simplifies your code:

Tita %>%

**group\_by**(Survived) %>%

**summarise**(**sum**(Freq)) *#How many survived and how many died?*

## # A tibble: 2 × 2

## Survived `sum(Freq)`

## <fctr> <dbl>

## 1 No 1490

## 2 Yes 711

Tita %>%

**group\_by**(Sex, Survived)%>%

**summarise**(**sum**(Freq)) *#How many women and men died?*

## Source: local data frame [4 x 3]

## Groups: Sex [?]

##

## Sex Survived `sum(Freq)`

## <fctr> <fctr> <dbl>

## 1 Male No 1364

## 2 Male Yes 367

## 3 Female No 126

## 4 Female Yes 344

Tita %>%

**group\_by**(Age, Survived) %>%

**summarise**(**sum**(Freq))*#How many children (or adults) died?*

## Source: local data frame [4 x 3]

## Groups: Age [?]

##

## Age Survived `sum(Freq)`

## <fctr> <fctr> <dbl>

## 1 Child No 52

## 2 Child Yes 57

## 3 Adult No 1438

## 4 Adult Yes 654

### dplyr::joins

Another very usefull feature of dplyr is the joins between tables. They work more or less like the SQL joins. After all, dplyr is a front end language that can be converted to SQL or spark. In order to perform a join between two tables, a key is needed to base that join on. In other words, a common variable between both tables is needed, on which the join will be based on.

So let’s make two dummy tables, with a common key (Name).

A=**data.frame**(Name = **c**("Maria", "Joana", "Paulo", "Abu", "Jacinta"),

age = **c**("cret", "jur", "mioc", "ord", "neog"),

refs = **c**(1:5))

B=**data.frame**(Name = **c**("Abu", "Joana", "Maria", "Gonzalo"),

comp = **c**(1:4),

sent= **c**(5:8))

**left\_join**(A, B)

## Name age refs comp sent

## 1 Maria cret 1 3 7

## 2 Joana jur 2 2 6

## 3 Paulo mioc 3 NA NA

## 4 Abu ord 4 1 5

## 5 Jacinta neog 5 NA NA

**right\_join**(A, B)

## Name age refs comp sent

## 1 Abu ord 4 1 5

## 2 Joana jur 2 2 6

## 3 Maria cret 1 3 7

## 4 Gonzalo <NA> NA 4 8

**inner\_join**(A, B)

## Name age refs comp sent

## 1 Maria cret 1 3 7

## 2 Joana jur 2 2 6

## 3 Abu ord 4 1 5

**semi\_join**(A, B)

## Name age refs

## 1 Abu ord 4

## 2 Joana jur 2

## 3 Maria cret 1

**semi\_join**(B, A)

## Name comp sent

## 1 Maria 3 7

## 2 Joana 2 6

## 3 Abu 1 5

**anti\_join**(A, B)

## Name age refs

## 1 Paulo mioc 3

## 2 Jacinta neog 5

**anti\_join**(B, A)

## Name comp sent

## 1 Gonzalo 4 8

**full\_join**(A, B)

## Name age refs comp sent

## 1 Maria cret 1 3 7

## 2 Joana jur 2 2 6

## 3 Paulo mioc 3 NA NA

## 4 Abu ord 4 1 5

## 5 Jacinta neog 5 NA NA

## 6 Gonzalo <NA> NA 4 8

**full\_join**(B, A)

## Name comp sent age refs

## 1 Abu 1 5 ord 4

## 2 Joana 2 6 jur 2

## 3 Maria 3 7 cret 1

## 4 Gonzalo 4 8 <NA> NA

## 5 Paulo NA NA mioc 3

## 6 Jacinta NA NA neog 5

The figure bellow is a resume on how the joins in dplyr work. Note that the difference of inner\_join() and semi\_join() is that the inner preserves all columns from both tables, whereas semi will only keep the “A” columns. In addition, if you’re looking for a join that will keep unmatching results from both tables, you have to use anti\_join() twice. For that, you can use the code:

**full\_join**(**anti\_join**(A, B), **anti\_join**(B, A))

## Name age refs comp sent

## 1 Paulo mioc 3 NA NA

## 2 Jacinta neog 5 NA NA

## 3 Gonzalo <NA> NA 4 8

knitr::**include\_graphics**("figure/12.jpg")

# Apply family

* apply will apply a function to a matrix
* lapply “l” stands for list. It returns a list of the same input as X, where each element is the result of applying a function to the corresponding element of X
* sapply “s” stands for simple. It is the same as lapply, but it returns a vector
* tapply “t” stands for table. This function will agregate your data into groups and apply the function over those groups.
* mapply m stands for multivariate. It is the same as lapply but instead of looping through a vector/list, it loops through each item of multiple lists, or vectors.

apply has three main arguments. First is the array (or matrix), second, the margin parameter which will determine if you want to perform the operation over a row or a column, and third indicates the function.

df<-**matrix**(1:20, 5)

df

## [,1] [,2] [,3] [,4]

## [1,] 1 6 11 16

## [2,] 2 7 12 17

## [3,] 3 8 13 18

## [4,] 4 9 14 19

## [5,] 5 10 15 20

**apply**(df, 2, sum)

## [1] 15 40 65 90

**apply**(df, 1, sum)

## [1] 34 38 42 46 50

Note that for the apply function you input an arrow, and what you get is a vector, or a matrix (if your margin argument is c(1,2)).

lapply

data <- **list**(foo=**c**(5:10), bar=**c**(1:50), baz=**c**(2000:2018))

**lapply**(data, FUN = sum)

## $foo

## [1] 45

##

## $bar

## [1] 1275

##

## $baz

## [1] 38171

**lapply**(data, FUN = mean)

## $foo

## [1] 7.5

##

## $bar

## [1] 26

##

## $baz

## [1] 2009

Note that for the lapply function you input a list, and you get as an output another list.

b <- **lapply**(data, FUN = sum)

**class**(b)

## [1] "list"

sapply

As stated before, sapply works exactly as lapply, but will return a vector.

a <- **sapply**(data, FUN = sum)

a

## foo bar baz

## 45 1275 38171

**class**(a)

## [1] "integer"

tapply To understand how this function works, let’s consider a data frame where I have a couple measurements on a plant characteristic - “foo” and “bar” and I measured those in 5 different ecosystems - A, B, C, D, and E. Now let’s say I want the mean over foo and bar for each ecosystem:

data<- **data.frame**("foo"=(1:100), "bar"=(201:400),

"ecosystem"=**rep**(**paste**(letters[1:5]), 40))

**tapply**(data$foo, data$ecosystem, mean)

## a b c d e

## 48 50 50 52 52

**tapply**(data$bar, data$ecosystem, mean)

## a b c d e

## 298 300 300 302 302

The first argument corresponds to the object to which you want to apply the function, second how should the data be grouped by, and third is the function to apply.

mapply As stated before, this function loops through each item of multiple lists, or vectors. Therefore, the objects should be the same lenght!!!

a<-**list**("foo"=**c**(0:10), "bar"= **c**(10:20))

**mapply**(sum, a$foo, a$bar)

## [1] 10 12 14 16 18 20 22 24 26 28 30

Note that first argument is the function, and the others are whatever objects you want.

**mapply**(sum, a$foo, **c**(50:60), a$foo, a$bar)

## [1] 60 64 68 72 76 80 84 88 92 96 100

**mapply**(mean, a$foo, a$bar)

## [1] 0 1 2 3 4 5 6 7 8 9 10

When writing regular expressions (regex) in Python language, we always start with the **letter r**. In this tutorial, we will understand the reason behind using it by answering the following questions:

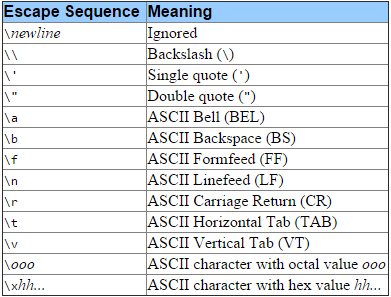
1. What are the escape sequences?
2. How Python interpreter interprets escape sequences with or without the **letter r**?
3. How regular expressions work in the Python language?
4. The importance of using the**letter r** in regular expressions

# 1. What are the escape sequences?

An escape sequence is a character set that does not represent itself when used in a text definition. It gets translated to some other character or character set that is otherwise difficult to present in a programming language. For example, in Python language, the **character set \n represents a new line, and \t represents a tab**.Both the character sets, \n, and \t are escape sequences.

The list of standard escape sequences understood by the Python interpreter and their associated meanings are as follows:

https://miro.medium.com/max/60/1*-LqNhcIKhafFCPAqStnMqQ.png?q=20



# 2. How Python interpreter interprets escape sequences with or without the letter r?

To understand its impact on escape sequences, let us have a look at the following example:

**#### Sample Text Definition**  
text\_1 = "My name is Ujjwal Dalmia.\nI love learning and teaching the Python language"  
print(text\_1)**#### Sample Output**  
My name is Ujjwal Dalmia.  
I love learning and teaching the Python language**#### Sample Text Definition**  
text\_2 = "My name is Ujjwal Dalmia.\sI love learning and teaching the Python language"  
print(text\_2)**#### Sample Output**  
My name is Ujjwal Dalmia.\sI love learning and teaching the Python language

In **text\_1** above, the example uses **\n** character set whereas **text\_2** uses **\s**. From the escape sequences table shared in section 1, we can see that **\n** is part of the standard escape sequence-set in Python language, whereas **\s** is not. Therefore, when we print both the variables, escape sequence **\n** is interpreted as a new line character by the Python interpreter, whereas **\s** is left as it is. Note that the **definition of both text\_1 and text\_2 does not include the letter r.**

Let us take a step further and **include the letter r** in the text definition.

**#### Sample Text Definition (with letter "r")**  
text\_1 = r"My name is Ujjwal Dalmia.\nI love learning and teaching the Python language"  
print(text\_1)**#### Sample Output**  
My name is Ujjwal Dalmia.\nI love learning and teaching the Python language**#### Sample Text Definition (with letter "r")**  
text\_2 = r"My name is Ujjwal Dalmia.\sI love learning and teaching the Python language"  
print(text\_2)**#### Sample Output**  
My name is Ujjwal Dalmia.\sI love learning and teaching the Python language

*The inclusion of the****letter r******had no impact on text\_2****because***\s***is not part of the standard escape sequence set in Python language. Surprisingly, for****text\_1***, ***the Python interpreter did not convert*\n*into the new line character****. It is because the presence of the****letter r****has transformed the text into a****raw-string. In simple terms, the letter r has instructed the Python interpreter to leave the escape sequence as it is****.*

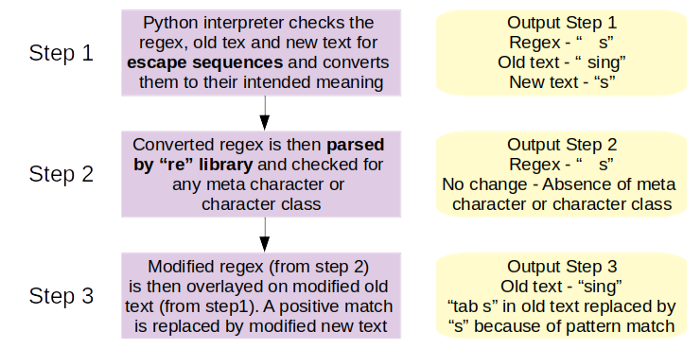
# 3.) How regular expression works in the Python language?

To understand how regular expressions work in Python language, we will use the **sub() function (re Python package) that substitutes the part of old text** with the **new text** based on the regular expression **driven pattern matching**. Let us understand this with an example:

**#### Importing the re package**  
import re**#### Using the sub function**  
re.sub("\ts","s", "\tsing")**#### Sample Output**  
'sing'

In this example, we are trying to **replace the letter s preceded by a tab**with the standalone**letter s**. One can see from the output that the text **\tsing** converts to **sing**. Let us refer to the below flow chart to understand how the **sub()** **function** produced the desired result. In the flow chart, we refer **to \ts** as**regex, letter s**as**new text,**and**\tsing**as**old text.**

https://miro.medium.com/max/60/1*8PZunh_2To7P7KBA7-S_iQ.png?q=20



Substitution using Standard Escape Sequence (Image by User)

## Explanation

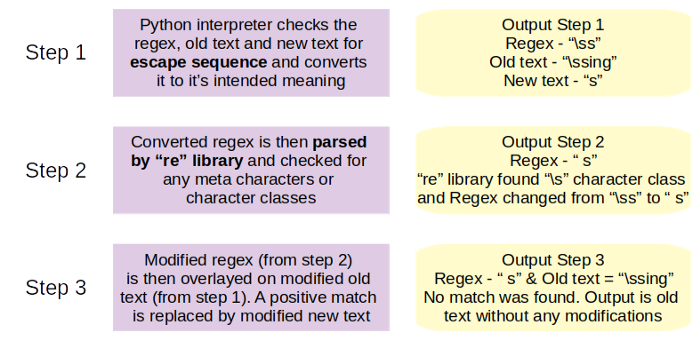
In the previous example, we have used the character set **\t,**which is part of the standard escape list in Python language. Therefore, in the first step, the Python interpreter replaced the escape sequence with the **tab** in both **regex text**and the **old text**. Since the regex pattern matched with the input text in the last step, the substitution took place.

In the next example, we will use a different character set, **\s,** that is not a part of the standard escape list in Python language.

**#### Importing the re package**  
import re**#### Using the sub function(this time with a non-standard escape sequence)**  
re.sub("\ss","s", "\ssing")**#### Sample Output**  
"\ssing"

In this example, we are trying to **replace any instance of the letter s preceded by \s with the standalone letter s**. It is evident that **there was no change in the input text,**and the output remained the same as the old text. Again, in the flow chart, we refer to **\ss**as **regex, s**as**the new text,**and**\ssing**as**the old.**Let us understand the reason behind this behavior from the below flowchart:

https://miro.medium.com/max/60/1*xsUYFEs_fX155TuQftMzeA.png?q=20



Substitution using Non-Standard Escape Sequence (Image by Author)

## Explanation

In step 1, since \s is **not a standard escape sequence**, the Python interpreter neither modified the regular expression nor the old text and left them as it is. In step 2, since \s is a **metacharacter** representing space, it gets **converted from \ss**to**space s**. Because in the old text, **space s** did not exist, there was no positive match, and hence the old-text remained the same.

The two learnings we can draw from this section are:

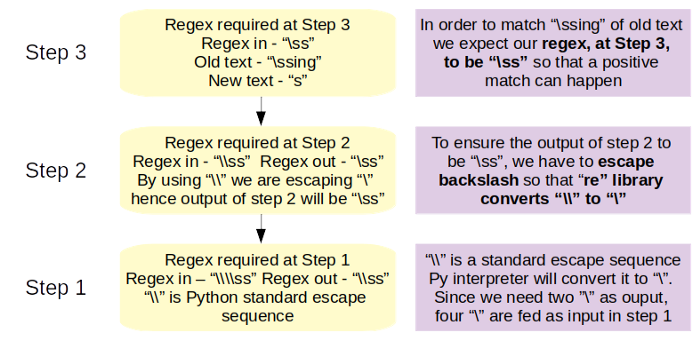
***The evaluation of old text and new text for escape sequences is done only by the Python interpreter. For the regular expression by the Python and the regex interpreter****. Therefore, for both old and new text, the outcome of step 1 is their final version, and for regex, it is step 2.*

***In a scenario where the texts and regex pattern contain only the standard escape sequence, which is part of Python language, we get our desired results. Whereas, when there are additional metacharacters, the results might not be as per expectation****.*

# 4.) The importance of using the letter r in regular expressions

From the 2nd example of the previous section, we saw that the regex failed to deliver the expected result. To find the right solution, let us work our way backward.

https://miro.medium.com/max/60/1*TCmP9bCIAJReRGcIpjSmWQ.png?q=20



Bottom-Up Approach to Solution (Image by Author)

## Explanation

*To substitute****\ss****from the old text with the***letter s*,****we expect the regex pattern at step 3 to match the text we want to replace.*

*To achieve this, we need the regex pattern to be****\\ss****by the end of step two. When the regex interpreter encounters this pattern, it will convert the metacharacters****double backslashes****to****single,****and the output of step 2 will be****\ss.***

*Finally, to ensure that regex at step 2 is****\\ss****, we pass****\\\\ss****at step 1. It is because****double backslashes****are a standard escape sequence of Python language and, as per the table in section 1****,****the Python interpreter will convert****double backslashes to single****. To get****\\ss****as the output of step 1, we supply****\\\\ss****as our first regular expression. The Python interpreter will convert the****\\\\ss****text pattern to****\\ss****.*

Therefore, the solution code to the problem mentioned above is as follows:

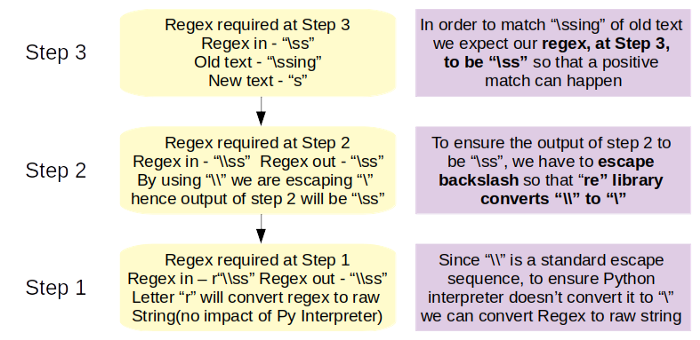
**#### Importing the re package**  
import re**#### Using the sub function with the modified regex**  
re.sub("\\\\ss","s", "\ssing")**#### Sample output**  
'sing'

We now have the solution to our problem, but the question which remains is, **where did we use the letter r**? The regex expression arrived at in the previous discussion is a candidate solution. In simple regex requirements, one can work with the above approach, but **consider a scenario where the regular expression dictates the use of multiple meta characters and** **standard escape sequences**. It would require us:

* To first differentiate between standard and non-standard escape sequences
* Then, appropriately place the right number of **backslashes** every time we encounter an escape sequence or metacharacters.

In such cumbersome scenarios, taking the below approach helps:

https://miro.medium.com/max/60/1*acW7YBPtXQhjNvzD5xv_8g.png?q=20



Replacing multiple escape characters by the letter r (Image by Author)

The only change we have made here is to replace **four backslashes**with **two**preceded by the **letter r**. It will ensure that in step 1, the Python interpreter considers the regular expression as the raw-string and leaves it as it is. Converting regex to a raw string will ensure the following:

* We are free from the worry of remembering the list of Python standard escape sequences.
* We do not have to worry about the right number of **backslashes** for the presence of standard escape sequences or any metacharacters.

Given above, our final and most appropriate solution will be as follows:

**#### Importing the re package**  
import re**#### Using the sub function with the modified regex**  
re.sub(**r**"\\ss","s", "\ssing")**#### Sample Output**  
'sing'

# Closing Note

Watch out for this **letter r** whenever writing your next regular expression. I hope that this tutorial gave you a good insight into the working of the regular expression.