# Table of Contents

# Data manipulation with dplyr and tidyr packages

In this section we focus on the use of `dplyr` and `tidyr`, two of the tidyverse packages. We will see how to group and summarise data in `dplyr`, and how to reformat long to wide and wide to long format of the data frame (or tibble, as it is called in tidyverse) using `tidyr`. We will use two set of real ecological datasets for this purpose.

## Piping data with %>% from magrittr

But first, let's have a reminder of the piping operator %>% from the package `margrittr`, which is used to pipe data into the function and also the output of one function to the other function. Here I will directly upload all tidyverse packages using `library (tidyverse)`, but for piping, `library (magrittr)` would be enough.

For example, instead of writing

```
result <- sum (log (sqrt (abs (-10:10))))
```

you can write the sequence using the pipe operator as

```
library (tidyverse)
result <- -10:10 %>% abs %>% sqrt %>% log1p %>% sum
```

Piping produces cleaner code, and allows you to create a logical *pipeline* of the functions, allowing to better imagine how the data flow through it (from left to right).

If not specified, the data get piped to the first argument of the function, so

```
cars %>% plot
```

will plot the scatterplot (try it!). I can include the rest of the arguments in the parenthesis of the function, while simply ignoring (skipping) the first one:

```
cars %>% plot (xlab = 'Speed [mph]', ylab = 'Distance [ft]')
```

Alternatively, it can be specified and replaced by a placeholder, a dot (.):

```
cars %>% plot (., xlab = 'Speed [mph]', ylab = 'Distance [ft]')
```

In this way, the data can be piped also to other arguments (here `xlab`):

```
'Speed [mph]' %>% plot (cars, xlab = ., ylab = 'Distance [ft]')
```

The original package from which pipes come (`magrittr`) contains also compound assignment operator %<>%, which pipes the result of the pipeline back to the original variable:

```
a <- 1:10
a %<>% sin %>% abs
a
```

```
 [1] 0.8414710 0.9092974 0.1411200 0.7568025 0.9589243 0.2794155 0.6569866
0.9893582 0.4121185
[10] 0.5440211
```

This piping operator is not imported to dplyr, so if you call dplyr directly (instead of tidyverse), you are not likely be able to use it.

Check magrittr reference for more details.

## Group and summarise data (group_by and summarise from dplyr)

The two functions, `group_by` and `summarise` (or `summarize` if you wish to use American English) are used to aggregate the data in the data frame (or a `tibble`, how is it called in tidyverse language) possibly by grouping variable.

A simple example using the `iris` data set. Let's turn `iris` into a `tibble` using `as_tibble` function:

```
head (iris)
```

```
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5          1.4         0.2  setosa
2          4.9         3.0          1.4         0.2  setosa
3          4.7         3.2          1.3         0.2  setosa
4          4.6         3.1          1.5         0.2  setosa
5          5.0         3.6          1.4         0.2  setosa
6          5.4         3.9          1.7         0.4  setosa
```

```
iris_tb <- as_tibble (iris)
iris_tb
```

```
# A tibble: 150 x 5
   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
          <dbl>       <dbl>        <dbl>       <dbl> <fct>
 1          5.1         3.5          1.4         0.2 setosa
 2          4.9         3            1.4         0.2 setosa
 3          4.7         3.2          1.3         0.2 setosa
 4          4.6         3.1          1.5         0.2 setosa
 5          5           3.6          1.4         0.2 setosa
 6          5.4         3.9          1.7         0.4 setosa
 7          4.6         3.4          1.4         0.3 setosa
 8          5           3.4          1.5         0.2 setosa
 9          4.4         2.9          1.4         0.2 setosa
10          4.9         3.1          1.5         0.1 setosa
```

```
# ... with 140 more rows
```

As you can see, the `tibble` format returns the data in more compact format, which gets always printed on the screen. The first row are column names, below is the row summarising the types of variables (`<dbl>` means double precise real number, `<fct>` is a factor) and below the shorted table is summarised what else the tibble contains (`# ... with 140 more rows`).

Note that `iris` (and `iris_tb`) contains a grouping variable `Species` - the whole dataset represents four measurements of sepal and petal lenghts and widths on 150 plants of *Iris* from three species, *Iris setosa*, *I. versicolor* and *I. virginica* (each with 50 individuals). We may calculate mean of each column using the `colMean` function, but this is not going to be too informative, since each species is likely to have different mean values for different parameters. Here come the `group_by` and `summarise` functions:

```
mean_iris_SL <- iris_tb %>% group_by (Species) %>% summarise (mean_SL = mean
(Sepal.Length))
mean_iris_SL
```

```
# A tibble: 3 x 2
  Species    mean_SL
  <fct>        <dbl>
1 setosa        5.01
2 versicolor    5.94
3 virginica     6.59
```

The `iris_tb` was piped into `group_by` function, which used the column `Species` to group the rows into three groups. Then, the function `summarise`, applied the `mean` function on `Sepal.Length`, and stored the result into variable `mean_SL`. Resulting tibble contains three rows - mean value of `Sepal.Length` for each species.

Similarly, we may include more than just a mean; here we take `mean`, standard deviation (`sd`) and the number of samples in each group (the function `n()`, which does not take arguments and simply counts numbers of rows in each group):

```
summary_iris_SL <- iris_tb %>% group_by (Species) %>%
   summarise (mean_SL = mean (Sepal.Length), sd_SL = sd (Sepal.Length), n_SL
= n ())
summary_iris_SL
```

```
# A tibble: 3 x 4
  Species    mean_SL sd_SL  n_SL
  <fct>        <dbl> <dbl> <int>
1 setosa        5.01 0.352    50
2 versicolor    5.94 0.516    50
3 virginica     6.59 0.636    50
```

Or, instead of applying more functions on the same variable, we can apply the same function on several variables:

```
mean_iris_SL_SW <- iris_tb %>% group_by (Species) %>%
  summarise (mean_SL = mean (Sepal.Length), mean_SW = mean (Sepal.Width))
```

```
mean_iris_SL_SW
```

```
# A tibble: 3 x 3
  Species    mean_SL mean_SW
  <fct>        <dbl>   <dbl>
1 setosa        5.01    3.43
2 versicolor    5.94    2.77
3 virginica     6.59    2.97
```

If we want to apply the same function on all columns (here mean), the function `summarise_all` is exactly for this:

```
mean_iris_all <- iris_tb %>% group_by (Species) %>% summarise_all (.funs =
mean)
mean_iris_all
```

```
# A tibble: 3 x 5
  Species    Sepal.Length Sepal.Width Petal.Length Petal.Width
  <fct>             <dbl>       <dbl>        <dbl>       <dbl>
1 setosa             5.01        3.43         1.46       0.246
2 versicolor         5.94        2.77         4.26       1.33
3 virginica          6.59        2.97         5.55       2.03
```

Indeed, the function `summarise` can be used without `group_by`, and then it works much like column-wise functions like `colMeans`, or generally as `apply` function with `MARGIN = 2`:

```
iris_tb %>% select (-Species) %>% summarise_all (mean)
```

```
# A tibble: 1 x 4
  Sepal.Length Sepal.Width Petal.Length Petal.Width
         <dbl>       <dbl>        <dbl>       <dbl>
1         5.84        3.06         3.76        1.20
```

```
iris_tb %>% select (-Species) %>% colMeans
```

```
Sepal.Length  Sepal.Width Petal.Length  Petal.Width
    5.843333     3.057333     3.758000     1.199333
```

```
iris_tb %>% select (-Species) %>% apply (2, mean)
# or explicitly as iris_tb[,-5] %>% apply (MARGIN = 2, FUN = function (x)
mean (x))
```

```
Sepal.Length  Sepal.Width Petal.Length  Petal.Width
    5.843333     3.057333     3.758000     1.199333
```

Note that we needed to remove the fifth column from the original data (since it contains factor whcih we cannot be averaged) - in `dplyr` there is function `select`, which can either select columns from the tibble (if you name them), or can remove it (if you name them and include minus sign in front). Also, note that the `colMeans` and `apply` solution returns standard data frame, not a tibble (these are not tidyverse functions).

# Long vs wide data format: transformation

Long format describes dataset in which the values of variables are gathered in two or three long columns, while wide format describes dataset in which the information is spread across many columns. Example of long format may be situation that we measure something (e.g. abundance) for each individual tree within several plots; then the data have one column for plot ID, one column for species name, and one column for the abundance. In this example, species name and abundance is so called key-value pair: key describes what is being measured, and value is the actually measured number. The same data can be shaped in wide format: each column is a single species name, each row is plot, and each cell of this data frame is the actual value of abundance of given species in given plot (in this case, the species abundance for species not occuring in given plot will be set to zero).

## Long to wide

As an example of long data, we use the subset of 10 most dominant species occurring in 3 different plots sampled within the cloud forest of Tamanshan (塔曼山, highest peak of Taipei basin), stored in the dataset Tamanshan.

Load data stored as gist on GitHub (we use function `read_delim` from tidyverse package `readr` for it):

```
long_format <- readr::read_delim
('https://gist.githubusercontent.com/zdealveindy/3078ac5f2852531604e703900c3
d05a2/raw/770529ff2cc746e6b02318f53089fc72e52ac1d7/long_format.txt', delim =
'\t')
```

```
long_format
```

```
# A tibble: 22 x 3
   Plot_ID Species_name     BA
   <chr>   <chr>         <dbl>
 1 L2L     EuryGlab       98.2
 2 L2L     PrunPhae      630.
 3 L2L     SycoSine     1295
 4 L2L     NeolAcum     2321.
 5 L2L     QuerSess     9430.
 6 L2R     QuerSten       11.3
 7 L2R     SympMacr       11.3
 8 L2R     PrunPhae       59.4
 9 L2R     EuryGlab       77.9
10 L2R     SycoSine      170.
# ... with 12 more rows
```

Use the function `spread` to spread the values in BA (the `value` argument) across multiple columns using `Species_name` as a key (argument `fill = 0` makes sure that combinations of `Species_name` and `Plot_ID` which would become NA will be replaced by given value, here 0):

```
wide <- long_format %>% spread (key = Species_name, value = BA, fill = 0)
```

```
wide
```

```
# A tibble: 3 x 11
  Plot_ID CarpRank DaphHima EuryGlab NeolAcum PrunPhae PrunTran QuerSess
QuerSten
  <chr>      <dbl>    <dbl>    <dbl>    <dbl>    <dbl>    <dbl>    <dbl>
<dbl>
1 L2L           0         0     98.2    2321.    630.        0     9430.
0
2 L2R           0         0     77.9    1700      59.4       0    30627.
11.3
3 L2W        1166.     1190    719.      350.      1.3    2382.    1090
1625.
# ... with 2 more variables: SycoSine <dbl>, SympMacr <dbl>
```

Data in this format can be used for further analysis (e.g. using multivariate methods in the package vegan).

## Wide to long

This is opposite situation - we have data spread across multiple columns, and wont to gather them into one key and one value column. We use the `iris` dataset here as an example:

```
iris_long <- as_tibble (iris) %>% gather (key = flower_feature, value =
measurement, Sepal.Length:Petal.Width)
```

```
# A tibble: 600 x 3
   Species flower_feature measurement
   <fct>   <chr>                <dbl>
 1 setosa  Sepal.Length           5.1
 2 setosa  Sepal.Length           4.9
 3 setosa  Sepal.Length           4.7
 4 setosa  Sepal.Length           4.6
 5 setosa  Sepal.Length           5
 6 setosa  Sepal.Length           5.4
 7 setosa  Sepal.Length           4.6
 8 setosa  Sepal.Length           5
 9 setosa  Sepal.Length           4.4
10 setosa  Sepal.Length           4.9
# ... with 590 more rows
```

Note that `as_tibble` function transforms the `iris` dataset into a tibble (although this may not be necessary here - if we didn't transform, result will be ordinary data frame). The sequence of variable names (`Sepal.Length:Petal.Width`) indicates which columns should be gathered into the key-value columns (if we did not specify, the function will use also the `Plot_ID` and gather it as a part of these two columns. Alternatively, we may code `as_tibble (iris) %>% gather (key = flower_feature, value = measurement, -Species)` (minus sign means to exclude this column from gathering).

Long format of data may be required by some analyses, and also may be more suitable e.g. for plotting (for example, try to use the formula version of boxplot function: `boxplot (measurement ~ flower_feature, iris_long)`, or alternative solution using the `lattice` package for plotting: `lattice::bwplot (measurement~Species|flower_feature, iris_long)`).