

# Data transformation with dplyr

## Introduction to R - Day 2

Instructor: Selina Baldauf

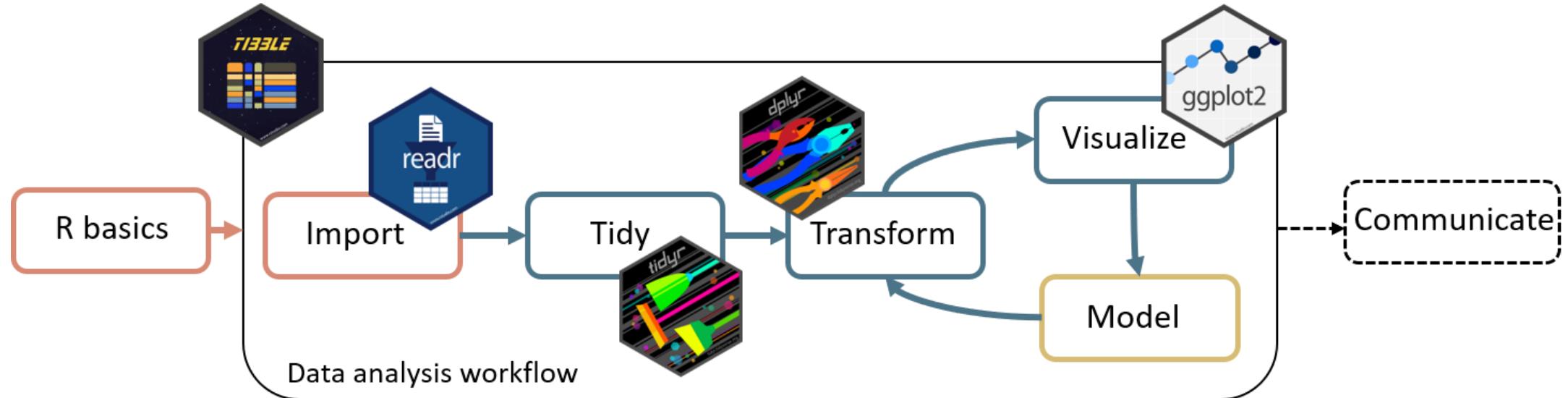
Freie Universität Berlin - Theoretical Ecology



2021-06-15 (updated: 2022-03-16)

# Data transformation

Data transformation is an important step in **understanding** the data and **preparing** it for further analysis.



We can use the tidyverse package `dplyr` for this.

# Data transformation

With `dplyr` we can (among other things)

- **Filter** data to analyse only a part of it
- **Reorder** observations or variables
- **Create** new variables
- **Summarize** data
- **Rename** variables

To get started load the package `dplyr`:

```
library(dplyr)
# or
library(tidyverse)
```

# Dplyr basic vocabulary

**dplyr** provides basic vocabulary for data manipulation:

- `filter()` picks observations (rows) based on their values
- `select()` picks variables (columns) based on their names
- `arrange()` changes order of observations (rows)
- `mutate()` adds new variables based on existing ones
- `summarize()` combines multiple values into a single summary value

Perform any of these operations by group with `group_by()`

# Dplyr basic vocabulary

All of the `dplyr` functions work similarly:

- **First argument** is the data (a tibble)
- **Other arguments** specify what to do exactly
- **Return** a tibble

# Example data

Soybean production for different use by year and country.

```
soybean_use
```

```
## # A tibble: 9,897 x 6
##   entity code    year human_food animal_feed processed
##   <chr>  <chr>  <dbl>      <dbl>      <dbl>
## 1 Africa <NA>    1961     33000      6000     14000
## 2 Africa <NA>    1962     43000      7000     17000
## 3 Africa <NA>    1963     31000      7000      5000
## 4 Africa <NA>    1964     43000      6000     14000
## 5 Africa <NA>    1965     34000      6000     12000
## # ... with 9,892 more rows
```

# filter()

picks observations (rows) based on their value

dplyr::filter()

KEEP ROWS THAT  
satisfy  
your CONDITIONS

keep rows from... this data... ONLY IF... type is "otter" AND site is "bay"

```
filter(df, type == "otter" & site == "bay")
```

type	food	site
otter	urchin	bay
Shark	seal	channel
otter	abalone	bay
otter	crab	wharf

@allison\_horst

# Useful `filter()` helpers

These functions and operators help you filter your observations:

- relational operators `<`, `>`, `==`, ...
- logical operators `&`, `|`, `!`
- `%in%` to filter multiple values
- `is.na()` to filter missing values
- `between()` to filter values that are between an upper and lower boundary
- `near()` to compare floating points (use instead of `==` for doubles)

# filter()

Filter rows that contain the values for Germany

```
filter(soybean_use, entity == "Germany")
```

```
## # A tibble: 53 x 6
##   entity code    year human_food animal_feed processed
##   <chr>   <chr> <dbl>      <dbl>      <dbl>      <dbl>
## 1 Germany DEU    1961        0       3000     1042000
## 2 Germany DEU    1962        0       3000      935000
## 3 Germany DEU    1963        0       3000     1092000
## 4 Germany DEU    1964        0       3000     1096000
## 5 Germany DEU    1965        0       3000     1435000
## # ... with 48 more rows
```

`filter()` goes through each row of the data and return only those rows where the value for `entity` is "Germany"

## filter() + %in%

Use the `%in%` operator to filter multiple countries

```
countries_select <- c("Germany", "Austria", "Switzerland")
filter(soybean_use, entity %in% countries_select)
```

```
## # A tibble: 159 x 6
##   entity code year human_food animal_feed processed
##   <chr>   <chr> <dbl>      <dbl>       <dbl>      <dbl>
## 1 Austria AUT    1961        0          0          0
## 2 Austria AUT    1962        0          0          0
## 3 Austria AUT    1963        0          0          0
## 4 Austria AUT    1964        0          0          0
## 5 Austria AUT    1965        0          0          0
## # ... with 154 more rows
```

## `filter() + is.na()`

Filter only rows that don't have a country code (i.e. the continents etc.)

```
filter(soybean_use, is.na(code))
```

```
## # A tibble: 1,734 x 6
##   entity code    year human_food animal_feed processed
##   <chr>  <chr>  <dbl>     <dbl>      <dbl>
## 1 Africa <NA>    1961     33000      6000     14000
## 2 Africa <NA>    1962     43000      7000     17000
## 3 Africa <NA>    1963     31000      7000      5000
## 4 Africa <NA>    1964     43000      6000     14000
## 5 Africa <NA>    1965     34000      6000     12000
## # ... with 1,729 more rows
```

Or the opposite: filter only the rows that have a country code with

```
filter(soybean_use, !is.na(code))
```

# filter() + between()

Combine different filters:

Select rows where

- the value for `years` is between 1970 and 1980
- the value for `entity` is Germany

```
filter(soybean_use, between(year, 1970, 1980) & entity == "Germany")
```

```
## # A tibble: 11 x 6
##   entity code    year human_food animal_feed processed
##   <chr>  <chr> <dbl>      <dbl>      <dbl>      <dbl>
## 1 Germany DEU     1970        0       3000    2118000
## 2 Germany DEU     1971        0       3000    2119000
## 3 Germany DEU     1972        0       5000    2271000
## 4 Germany DEU     1973        0       3000    2820000
## 5 Germany DEU     1974        0       3000    3704000
## # ... with 6 more rows
```

# `select()`

picks variables (columns) based on their names

# Useful `select()` helpers

- `starts_with()` and `ends_with()`: variable names that start/end with a specific string
- `contains()`: variable names that contain a specific string
- `matches()`: variable names that match a regular expression
- `any_of()` and `all_of()`: variables that are contained in a character vector

# select()

Select the variables entity, year and human food

```
select(soybean_use, entity, year, human_food)
```

```
## # A tibble: 9,897 x 3
##   entity    year human_food
##   <chr>     <dbl>      <dbl>
## 1 Africa    1961      33000
## 2 Africa    1962      43000
## # ... with 9,895 more rows
```

Remove variables using -

```
select(soybean_use, -entity, -year, -human_food)
```

```
## # A tibble: 9,897 x 3
##   code animal_feed processed
##   <chr>     <dbl>      <dbl>
## 1 <NA>       6000      14000
## 2 <NA>       7000      17000
## # ... with 9,895 more rows
```

# select() + ends\_with()

Select all columns that end with "d"

```
select(soybean_use, ends_with("d"))
```

```
## # A tibble: 9,897 x 3
##   human_food animal_feed processed
##       <dbl>      <dbl>     <dbl>
## 1     33000      6000    14000
## 2     43000      7000    17000
## 3     31000      7000     5000
## # ... with 9,894 more rows
```

You can use the same structure for starts\_with() and contains().

```
# this does not match any rows in the soy bean data set
# but combinations like this are helpful for research data
select(soybean_use, starts_with("sample_"))

select(soybean_use, contains("_id_"))
```

# `select() + any_of()/all_of()`

Use a character vector in conjunction with column selection

```
cols <- c("sample_", "year", "processed", "entity")
```

`any_of()` returns any columns that match an element in `cols`

```
select(soybean_use, any_of(cols))
```

```
## # A tibble: 9,897 x 3
##   year processed entity
##   <dbl>      <dbl> <chr>
## 1 1961        14000 Africa
## # ... with 9,896 more rows
```

`all_of()` tries to match all elements in `cols` and returns an error if an element does not exist

```
select(soybean_use, all_of(cols))
```

```
## Error: Can't subset columns that don't exist.
## x Column `sample_` doesn't exist.
```

## `select()` + `from:to`

Multiple consecutive columns can be selected using the `from:to` structure with either column id or variable name:

```
select(soybean_use, 1:3)
select(soybean_use, code:animal_feed)
```

```
## # A tibble: 9,897 x 4
##   code      year human_food animal_feed
##   <chr>     <dbl>     <dbl>       <dbl>
## 1 <NA>      1961     33000       6000
## 2 <NA>      1962     43000       7000
## 3 <NA>      1963     31000       7000
## # ... with 9,894 more rows
```

Be a bit careful with these commands: They are not robust if you e.g. change the order of your columns at some point.

# `arrange()`

change order of observations (rows)

# arrange()

Arrange the rows by **ascending** values of the processed variable:

```
arrange(soybean_use, processed)
```

```
## # A tibble: 9,897 x 6
##   entity code year human_food animal_feed processed
##   <chr>   <chr> <dbl>      <dbl>      <dbl>
## 1 Albania ALB     1961        0        NA        0
## 2 Albania ALB     1962        0        NA        0
## 3 Albania ALB     1963        0        NA        0
## # ... with 9,894 more rows
```

Arrange the rows by **descending** values of the processed variable:

```
arrange(soybean_use, desc(processed))
```

Arranging also works for character columns. They will be sorted **alphabetically**.

# arrange()

We can also sort rows by multiple variables

```
# sort first by year, then by entity
arrange(soybean_use, year, entity)

## # A tibble: 9,897 x 6
##   entity    code    year human_food animal_feed processed
##   <chr>     <chr>   <dbl>      <dbl>       <dbl>
## 1 Africa    <NA>    1961     33000      6000     14000
## 2 Albania   ALB     1961      0          NA        0
## 3 Algeria   DZA     1961      0          0         NA
## 4 Americas  <NA>    1961     7000      96000    11656000
## # ... with 9,893 more rows
```

# mutate()

adds new variables



Artwork by Allison Horst

# mutate()

New columns can be added based on values from other columns

```
mutate(soybean_use, sum_human_animal = human_food + animal_feed)
```

```
## # A tibble: 9,897 x 7
##   entity code    year human_food animal_feed processed sum_human_animal
##   <chr>  <chr>  <dbl>      <dbl>       <dbl>      <dbl>
## 1 Africa <NA>    1961      33000      6000      14000      39000
## 2 Africa <NA>    1962      43000      7000      17000      50000
## 3 Africa <NA>    1963      31000      7000      5000      38000
## # ... with 9,894 more rows
```

Add multiple new columns at once:

```
mutate(soybean_use,
       sum_human_animal = human_food + animal_feed,
       total = human_food + animal_feed + processed
     )
```

# `mutate() + case_when()`

Use `case_when` to add column values conditional on other columns.

`case_when()` can combine many cases into one.

```
mutate(soybean_use,
  legislation = case_when(
    year < 2000 & year >= 1980 ~ "legislation_1",      # case 1
    year >= 2000 ~ "legislation_2",                      # case 2
    TRUE ~ "no_legislation"                                # any other cases
  )
)
```

```
## # A tibble: 9,897 x 7
##   entity code    year human_food animal_feed processed legislation
##   <chr>  <chr> <dbl>       <dbl>       <dbl> <chr>
## 1 Africa <NA>    1961       33000       6000  14000  no_legislation
## 2 Africa <NA>    1962       43000       7000  17000  no_legislation
## 3 Africa <NA>    1963       31000       7000   5000  no_legislation
## # ... with 9,894 more rows
```

# transmute()

transmute() is like mutate() but only keeps the new columns

```
transmute(soybean_use,
          ratio_processed_animal = processed/animal_feed,
          ratio_human_animal = human_food/animal_feed)
```

```
## # A tibble: 9,897 x 2
##   ratio_processed_animal ratio_human_animal
##             <dbl>            <dbl>
## 1           2.33              5.5
## 2           2.43              6.14
## 3           0.714             4.43
## # ... with 9,894 more rows
```

# `summarize() + group_by()`

summarizes data by group

# summarize()

summarize will collapse the data to a single row

```
summarize(soybean_use,
          total_animal = sum(animal_feed, na.rm = TRUE),
          total_human = sum(human_food, na.rm = TRUE))
## # A tibble: 1 x 2
##   total_animal total_human
##       <dbl>      <dbl>
## 1     942503000  1589729000
```

# **summarize()** and **group\_by()**

`summarize` is much more useful in combination with `group_by()`.

If you group the data before summarizing it, the `summary` will be calculated **separately** for each group

```
# group the data by year
soybean_use_group <- group_by(soybean_use, year)

# summarize the grouped data
summarize(soybean_use_group,
          total_animal = sum(animal_feed, na.rm = TRUE),
          total_human = sum(human_food, na.rm = TRUE))

## # A tibble: 53 x 3
##   year total_animal total_human
##   <dbl>      <dbl>       <dbl>
## 1 1961        1503000     16994000
## 2 1962        1800000     17326000
## # ... with 51 more rows
```

To ungroup data that was grouped before, you can use `ungroup()`

# count()

Counts observations by group

```
# count rows grouped by year  
count(soybean_use, year)  
  
# or if the data is already grouped by year  
count(soybean_use_group)
```

```
## # A tibble: 53 x 2  
## # Groups:   year [53]  
##       year     n  
##   <dbl> <int>  
## 1 1961    178  
## 2 1962    178  
## 3 1963    178  
## 4 1964    178  
## # ... with 49 more rows
```

# The pipe %>%

Combine multiple data operations into one command

# The pipe %>%

Data transformation often requires **multiple operations** in sequence.

The pipe operator `%>%` helps to keep these operations clear and readable.

# The pipe %>%

Let's look at an example without pipe:

```
# 1: filter rows that actually represent a country
soybean_new <- filter(soybean_use, !is.na(code))

# 2: group the data by year
soybean_new <- group_by(soybean_new, year)

# 3: summarize mean values by year
soybean_new <- summarize(soybean_new,
  mean_processed = mean(processed, na.rm=TRUE),
  sd_processed = sd(processed, na.rm = TRUE))

# 4: reorder the observation with newest first
soybean_new <- arrange(soybean_new, desc(year))
```

How could we make this more efficient?

# The pipe %>%

We could do everything in one step without intermediate results by using one **nested function**

```
soybean_new <- arrange(  
  summarize(  
    group_by(  
      filter(soybean_use, !is.na(code)),  
      year  
    ),  
    mean_processed = mean(processed, na.rm = TRUE),  
    sd_processed = sd(processed, na.rm = TRUE)  
  ),  
  desc(year)  
)
```

But this gets complicated and error prone very quickly

# The pipe %>%

The pipe operator (included in the `tidyverse`) makes it very easy to combine multiple operations:

```
soybean_new <- soybean_use %>%
  filter(!is.na(code)) %>%
  group_by(year) %>%
  summarize(
    mean_processed = mean(processed, na.rm = TRUE),
    sd_processed = sd(processed, na.rm = TRUE)
  ) %>%
  arrange(desc(year))
```

You can read from top to bottom and interpret the `%>%` as an "and then do".

# The pipe %>%

But what is happening?

The pipe is "pushing" the result of one line into the first argument of the function from the next line.

```
soybean_use %>%
  count(year)

# instead of
count(soybean_use, year)
```

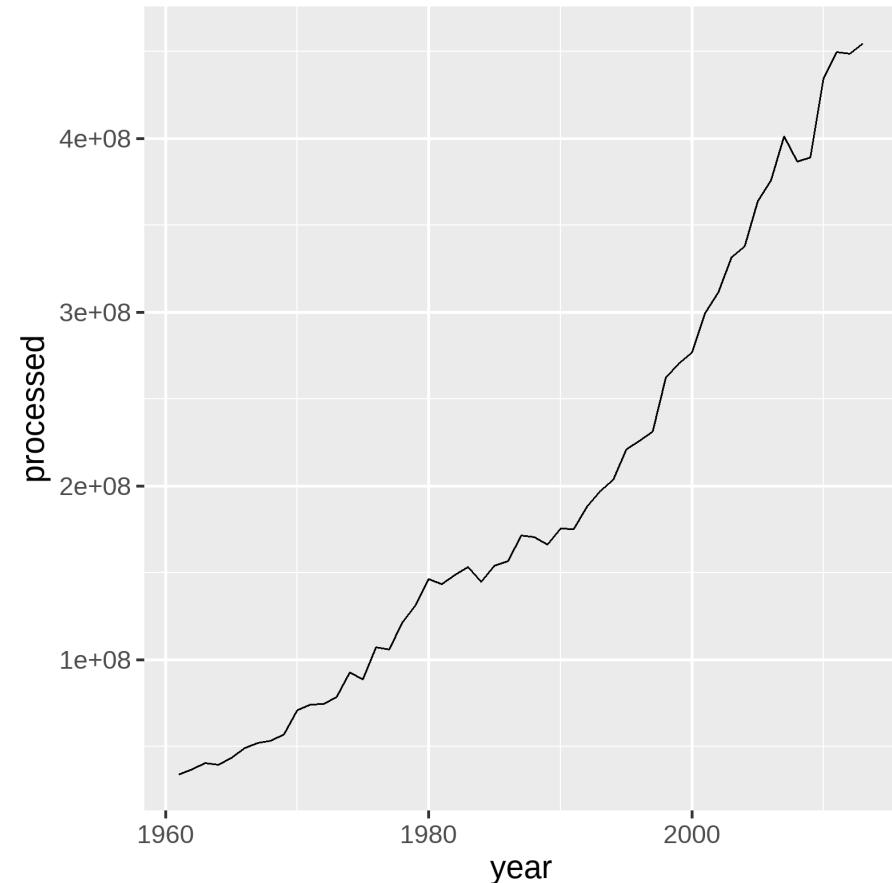
Piping works perfectly with the `tidyverse` functions because they are designed to return a tibble **and** take a tibble as first argument.

 Use the keyboard shortcut Ctrl/Cmd + Shift + M to insert %>%

# The pipe %>%

Piping also works well together with `ggplot`

```
soybean_use %>%
  filter(!is.na(code)) %>%
  select(year, processed) %>%
  group_by(year) %>%
  summarize(
    processed = sum(processed, na.rm = TRUE)
  ) %>%
  ggplot(aes(
    x = year,
    y = processed
  )) +
  geom_line()
```



# Summary I

All `dplyr` functions take a tibble as first argument and return a tibble.

## `filter()`

- **pick rows** with helpers
  - relational and logical operators
  - `%in%`
  - `is.na()`
  - `between()`
  - `near()`

## `select()`

- **pick columns** with helpers
  - `starts_with()`, `ends_with()`
  - `contains()`
  - `matches()`
  - `any_of()`, `all_of()`

# Summary II

## `arrange()`

- change order of rows (adscending)
  - or descending with `desc()`

## `mutate()`

- add columns but keep all columns
  - `case_when()` for conditional values

## `transmute()`

- add columns and drop old columns

# Summary III

## `summarize() + group_by()`

- collapse rows into one row by some summary
  - combine with `group_by()` to summarize by group
  - use `ungroup()` to ungroup grouped tibble

## `count`

- count rows based on a group
  - can be used in combination with `group_by()`

Now you

Task 2: Transforming the penguin data set (60 min)

Find the task description [here](#)