Data transformation with dplyr

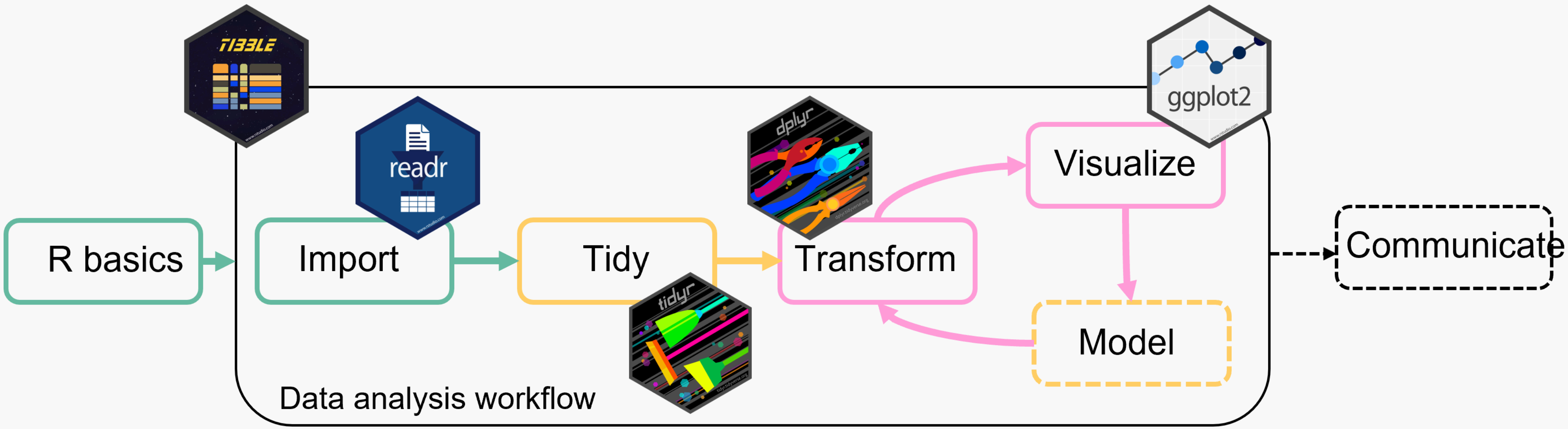
Day 2 - Introduction to Data Analysis with R

Selina Baldauf
Freie Universität Berlin - Theoretical Ecology

February 28, 2025

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
- **Create** new variables
- **Summarize** data
- **Combine** multiple tables
- **Rename** variables
- **Reorder** observations or variables

To get started load the package `dplyr`:

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

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

The data

Data set `and_vertebrates` with measurements of a trout and 2 salamander species in different forest sections.

- `year`: observation year
- `section`: CC (clear cut forest) or OG (old growth forest)
- `unittype`: channel classification (C = Cascade, P = Pool, …)
- `species`: Species measured
- `length_1_mm`: body length [mm]
- `weight_g`: body weight [g]

Coastal giant salamander (terrestrial form) Andrews Forest Program by Lina DiGregorio via CC-BY from https://andrewsforest.oregonstate.edu

References: Kaylor, M.J. and D.R. Warren. (2017) and Gregory, S.V. and I. Arismendi. (2020) as provided in the

The data

Data set `and_vertebrates` with measurements of a trout and 2 salamander species in different forest sections.

```
library(lterdatasampler)
#> Error in library(lterdatasampler): there is no package called 'lterdatasampler'
and_vertebrates
#> Error: object 'and_vertebrates' not found
```

`filter()`

> picks rows based on their value

`filter()`

Filter only the trout species:

```
filter(and_vertebrates, species == "Cutthroat trout")
#> Error: object 'and_vertebrates' not found
```

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

`filter()`

You can also combine filters using logical operators (`&`, `|`, `!`):

```r
filter(and_vertebrates, species == "Cutthroat trout" & year == 1987)
#> Error: object 'and_vertebrates' not found
```

filter() + %in%

Use the %in% operator to filter rows based on multiple values, e.g. unittypes

```
unittype_select <- c("R", "C", "S")
filter(and_vertebrates, unittype %in% unittype_select)
#> Error: object 'and_vertebrates' not found
```

filter() + is.na()

Filter only rows that don't have a value for the weight

```
filter(and_vertebrates, is.na(weight_g))
#> Error: object 'and_vertebrates' not found
```

Or the opposite: filter only the rows that have a value for the weight

```
filter(and_vertebrates, !is.na(weight_g))
```

`filter()` + `between()`

Combine different filters:

Filter rows where the value for `year` is between 2000 and 2005

```
filter(and_vertebrates, between(year, 2000, 2005))
#> Error: object 'and_vertebrates' not found
```

Or you could also do it like this:

```
filter(and_vertebrates, year >= 2000 & year <= 2005)
```

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)

`select()`

> picks columns based on their names

`select()`

Select the columns `species`, `length_1_mm`, and `year`

```
select(and_vertebrates, species, length_1_mm, year)
#> Error: object 'and_vertebrates' not found
```

Remove variables using `-`

```
select(and_vertebrates, -species, -length_1_mm, -year)
```

`select()` + `ends_with()`

Select all columns that start with `"s"`

```
select(and_vertebrates, starts_with("s"))
```

```
#> Error: object 'and_vertebrates' not found
```

You can use the same structure for `starts_with()` and `contains()`.

```
# this does not make sense for the example data
# but combinations like this are helpful for research data
select(and_vertebrates, starts_with("sample_"))

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

`select()` + `from:to`

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

```
select(and_vertebrates, 1:3)
select(and_vertebrates, year:unittype)
```

```
#> Error: object 'and_vertebrates' not found
```

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

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

`mutate()`

> Adds new columns to your data

`mutate()`

New columns can be added based on values from other columns

```
mutate(and_vertebrates, weight_kg = weight_g/1000)
```

```
#> Error: object 'and_vertebrates' not found
```

Add multiple new columns at once:

```
mutate(and_vertebrates,
       weight_kg = weight_g/1000,
       length_m = length_1_mm/1000)
```

mutate() + case_when()

Use case_when to add column values conditional on other columns.

case_when() can combine many cases into one.

```
mutate(and_vertebrates,
       type = case_when(
         species == "Cutthroat trout" ~ "Fish",                     # case 1
         species == "Coastal giant salamander" ~ "Amphibian", # case 2
         .default = NA                                              # all other
))
#> Error: object 'and_vertebrates' not found
```

`summarize()`

> summarizes data

summarize()

`summarize` will collapse the data to a single row

```
summarize(and_vertebrates,
          mean_length = mean(length_1_mm, na.rm = TRUE),
          mean_weight = mean(weight_g, na.rm = TRUE))
#> Error: object 'and_vertebrates' not found
```

`summarize()` by group

`summarize` is much more useful in combination with the grouping argument `.by`

- **summary** will be calculated **separately for each group**

```
# summarize the grouped data
summarize(and_vertebrates,
    mean_length = mean(length_1_mm, na.rm = TRUE),
    mean_weight = mean(weight_g, na.rm = TRUE),
    .by = species
  )
#> Error: object 'and_vertebrates' not found
```

count()

Counts observations by group

```
# count rows grouped by year
count(and_vertebrates, year)
#> Error: object 'and_vertebrates' not found
```

The pipe |>

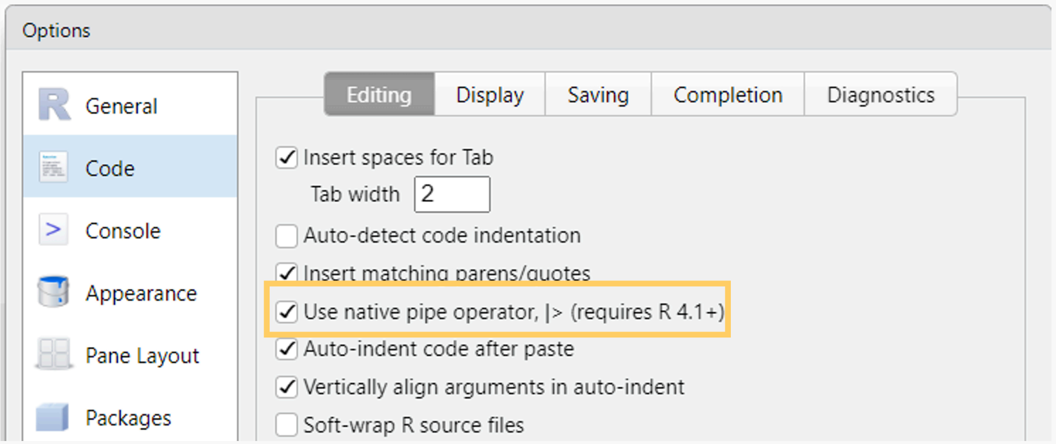
> Combine multiple data operations into one command

The pipe |>

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

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

- You may also see `%>%` from the `magrittr` package
- Turn on the native R pipe `|>` in **Tools -> Global Options -> Code**



The pipe |>

Let's look at an example without pipe:

```
# 1: filter rows that have don't have NA in the unittype column
and_vertebrates_new <- filter(and_vertebrates, !is.na(unittype))

# 2: summarize mean values by year
and_vertebrates_new <- count(and_vertebrates_new, year, species, section)
```

How could we make this more efficient?

Use one **nested function** without intermediate results:

```
and_vertebrates_new <- count(
  filter(and_vertebrates, !is.na(unittype)),
  year, species, section
)
```

But this gets complicated and error prone very quickly

The pipe |>

The pipe operator makes it very easy to combine multiple operations:

```
and_vertebrates_new <- and_vertebrates |>
  filter(!is.na(unittype)) |>
  count(year, species, section)

and_vertebrates_new
```

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.

```
and_vertebrates |>
  count(year)

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

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

> **Tip**
>
> Use the keyboard shortcut `Ctrl/Cmd + Shift + M` to insert `|>`

The pipe |>

Piping also works well together with `ggplot`

```
and_vertebrates |>
  filter(!is.na(unittype)) |>
  count(year, species, section) |>
  ggplot(aes(x = year, y = n, color = species)) +
  geom_line() +
  facet_wrap(~section)
#> Error: object 'and_vertebrates' not found
```

Combining mulitiple tables

Combine two tibbles by row `bind_rows`

Situation: Two (or more) `tibbles` with the same variables (column names)

```
tbl_a <- and_vertebrates[1:2, ] # first two rows
#> Error: object 'and_vertebrates' not found
tbl_b <- and_vertebrates[2:nrow(and_vertebrates), ] # the rest
#> Error: object 'and_vertebrates' not found
```

```
tbl_a
```

```
#> Error: object 'tbl_a' not found
```

```
tbl_b
```

```
#> Error: object 'tbl_b' not found
```

Combine two tibbles by row `bind_rows`

Bind the rows together with `bind_rows()`:

```
bind_rows(tbl_a, tbl_b)
```

```
#> Error: object 'tbl_a' not found
```

You can also add an ID-column to indicate which line belonged to which table:

```
bind_rows(a = tbl_a, b = tbl_b, .id = "id")
```

```
#> Error: object 'tbl_a' not found
```

You can use `bind_rows()` to bind as many tables as you want:

```
bind_rows(a = tbl_a, b= tbl_b, c = tbl_c, ..., .id = "id")
```

Join tibbles with `left_join()`

Situation: Two tables that share some but not all columns.

```
#> Error: object 'and_vertebrates' not found
```

```
and_vertebrates
```

```
#> Error: object 'and_vertebrates' not found
```

```
# table with more information on the species
species
#> Error: object 'species' not found
```

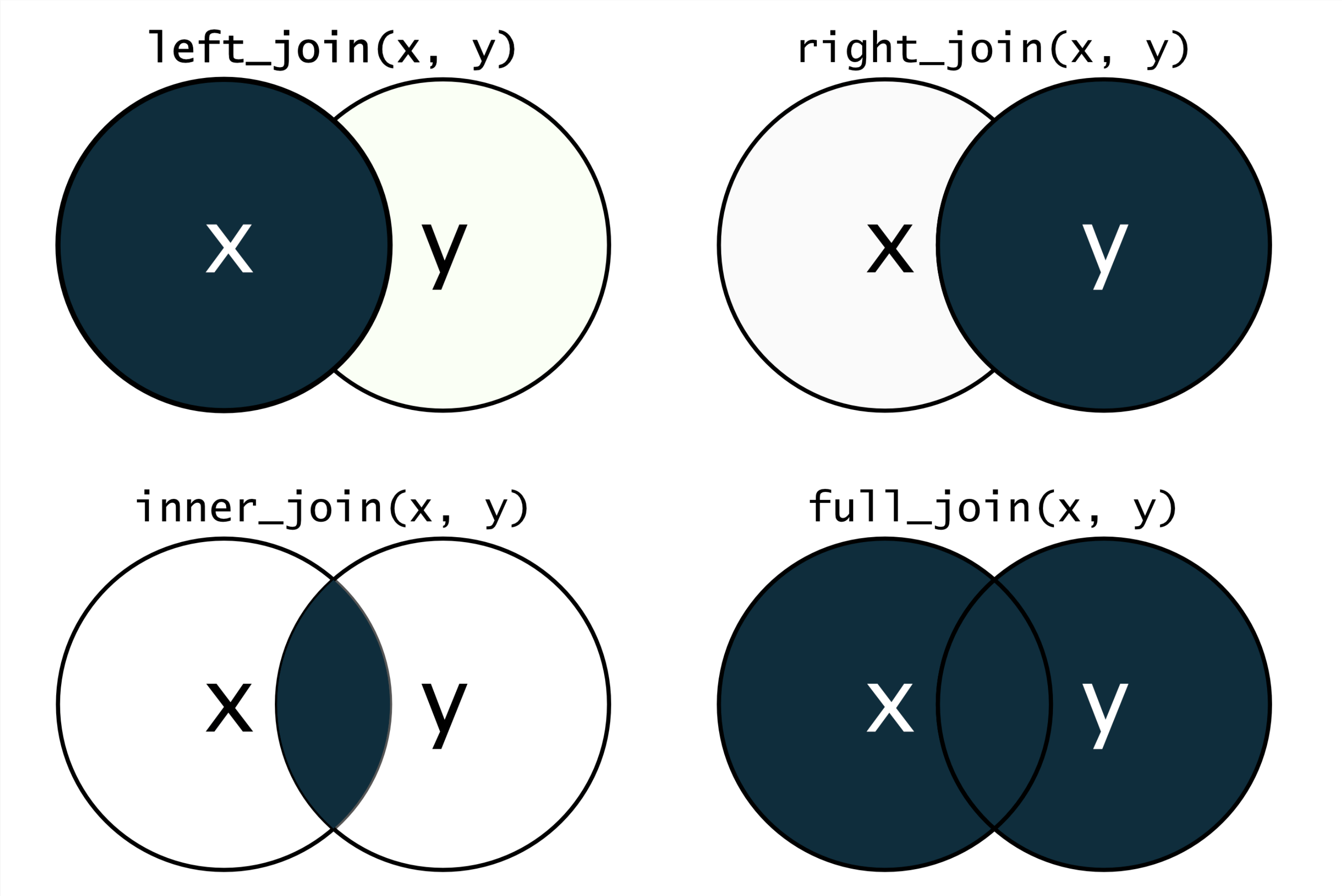
Join tibbles with `left_join()`

Join the two tables by the common column `species`

```
left_join(and_vertebrates, species, by = "species")
#> Error: object 'and_vertebrates' not found
```

`left_join()` means that the resulting tibble will contain all rows of `and_vertebrates`, but not necessarily all rows of `species` (in this case it does though).

Different `*_join()` functions



Summary

> Data transformation with dplyr

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()`

Summary II

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

`select()`

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

Summary III

`arrange()`

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

`mutate()`

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

Summary IV

`summarize()`

- **collapse rows** into one row by some summary
  - use `.by` argument to summarize by group

`count`

- **count rows** based on a group

Summary V

`bind_rows()`

- **combine rows** of multiple tibbles into one
  - the tibbles need to have the same columns
  - add an id column with the argument `.id = "id"`
  - function `bind_cols()` works similarly just for columns

`left_join()`

- **combine tables** based on common columns

Now you

Task (60 min)

Transform the penguin data set

Find the task description here