

Data transformation with dplyr

Day 2 - Introduction to Data Analysis with R

Selina Baldauf

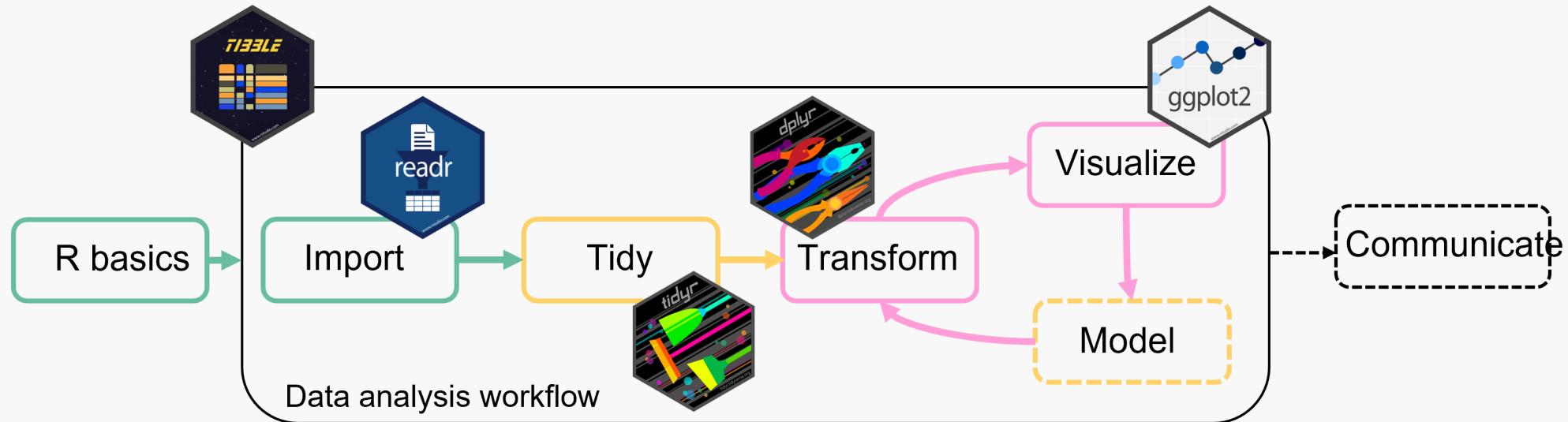
Freie Universität Berlin - Theoretical Ecology

October 2, 2023



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`:

```
1 library(dplyr)
2 # or
3 library(tidyverse)
```

Dplyr basic vocabulary for data manipulation

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

Perform any of these operations by group

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.

```
1 soybean_use <- readr::read_csv('https://raw.githubusercontent.com/rfordatascience/tidytuesday/master/
2 soybean_use
3 #> # A tibble: 9,897 × 6
4 #>   entity code    year human_food animal_feed processed
5 #>   <chr>  <chr> <dbl>      <dbl>      <dbl>
6 #> 1 Africa <NA>  1961       33000     6000     14000
7 #> 2 Africa <NA>  1962       43000     7000     17000
8 #> 3 Africa <NA>  1963       31000     7000      5000
9 #> 4 Africa <NA>  1964       43000     6000     14000
10 #> 5 Africa <NA>  1965       34000     6000     12000
11 #> 6 Africa <NA>  1966       41000     6000      2000
12 #> 7 Africa <NA>  1967       47000     6000      4000
13 #> 8 Africa <NA>  1968       50000     7000      3000
14 #> 9 Africa <NA>  1969       52000     6000      6000
15 #> 10 Africa <NA> 1970       52000     6000      8000
16 #> # i 9,887 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

Artwork by 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

```
1 filter(soybean_use, entity == "Germany")
2 #> # A tibble: 53 × 6
3 #>   entity code    year human_food animal_feed processed
4 #>   <chr>   <chr> <dbl>     <dbl>      <dbl>      <dbl>
5 #> 1 Germany DEU    1961      0     3000    1042000
6 #> 2 Germany DEU    1962      0     3000    935000
7 #> 3 Germany DEU    1963      0     3000    1092000
8 #> 4 Germany DEU    1964      0     3000    1096000
9 #> 5 Germany DEU    1965      0     3000    1435000
10 #> 6 Germany DEU   1966      0     3000    1588000
11 #> 7 Germany DEU   1967      0     3000    1646000
12 #> 8 Germany DEU   1968      0     3000    1480000
13 #> 9 Germany DEU   1969      0     3000    1423000
14 #> 10 Germany DEU  1970      0     3000    2118000
15 #> # i 43 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 rows based on multiple values, e.g. countries

```
1 countries_select <- c("Germany", "Austria", "Switzerland")
2 filter(soybean_use, entity %in% countries_select)
3 #> # A tibble: 159 × 6
4 #>   entity    code    year human_food animal_feed processed
5 #>   <chr>     <chr>   <dbl>      <dbl>      <dbl>
6 #>   1 Austria  AUT     1961       0          0          0
7 #>   2 Austria  AUT     1962       0          0          0
8 #>   3 Austria  AUT     1963       0          0          0
9 #>   4 Austria  AUT     1964       0          0          0
10 #>  5 Austria  AUT     1965       0          0          0
11 #>  6 Austria  AUT     1966       0          0          0
12 #>  7 Austria  AUT     1967       0          0          0
13 #>  8 Austria  AUT     1968       0          0          0
14 #>  9 Austria  AUT     1969       0          0          0
15 #> 10 Austria AUT     1970       0          0          0
16 #> # i 149 more rows
```

filter() + is.na()

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

```
1 filter(soybean_use, is.na(code))
2 #> # A tibble: 1,734 × 6
3 #>   entity code    year human_food animal_feed processed
4 #>   <chr>  <chr> <dbl>     <dbl>      <dbl>      <dbl>
5 #> 1 Africa <NA>  1961     33000     6000     14000
6 #> 2 Africa <NA>  1962     43000     7000     17000
7 #> 3 Africa <NA>  1963     31000     7000      5000
8 #> 4 Africa <NA>  1964     43000     6000     14000
9 #> 5 Africa <NA>  1965     34000     6000     12000
10 #> 6 Africa <NA>  1966     41000     6000      2000
11 #> 7 Africa <NA>  1967     47000     6000      4000
12 #> 8 Africa <NA>  1968     50000     7000      3000
13 #> 9 Africa <NA>  1969     52000     6000      6000
14 #> 10 Africa <NA> 1970     52000     6000      8000
15 #> # i 1,724 more rows
```

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

```
1 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

```
1 filter(soybean_use, between(year, 1970, 1980) & entity == "Germany")
2 #> # A tibble: 11 × 6
3 #>   entity code    year human_food animal_feed processed
4 #>   <chr>   <chr> <dbl>     <dbl>      <dbl>      <dbl>
5 #>   1 Germany DEU    1970        0       3000    2118000
6 #>   2 Germany DEU    1971        0       3000    2119000
7 #>   3 Germany DEU    1972        0       5000    2271000
8 #>   4 Germany DEU    1973        0       3000    2820000
9 #>   5 Germany DEU    1974        0       3000    3704000
10 #>  6 Germany DEU    1975        0       3000    3480000
11 #>  7 Germany DEU    1976        0       1000    3453000
12 #>  8 Germany DEU    1977        0       3000    3388000
13 #>  9 Germany DEU    1978        0       3000    3647000
14 #> 10 Germany DEU   1979        0       2000    3700000
15 #> 11 Germany DEU   1980        0       3000    3887000
```

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

```
1 select(soybean_use, entity, year, human_food)
2 #> # A tibble: 9,897 × 3
3 #>   entity    year  human_food
4 #>   <chr>    <dbl>
5 #>   1 Africa  1961     33000
6 #>   2 Africa  1962     43000
7 #>   3 Africa  1963     31000
8 #>   4 Africa  1964     43000
9 #>   5 Africa  1965     34000
10 #>  6 Africa  1966     41000
11 #>  7 Africa  1967     47000
12 #>  8 Africa  1968     50000
13 #>  9 Africa  1969     52000
14 #> 10 Africa  1970     52000
15 #> # i 9,887 more rows
```

Remove variables using -

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

select() + ends_with()

Select all columns that end with "d"

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

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

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

```
1 # this does not match any rows in the soy bean data set
2 # but combinations like this are helpful for research data
3 select(soybean_use, starts_with("sample_"))
4
5 select(soybean_use, contains("_id_"))
```

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

Use a character vector in conjunction with column selection

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

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

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

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

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

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

#> Error in `all_of()`:
#> ! Can't subset columns that don't exist.
#> ✘ 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:

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

#> # A tibble: 9,897 × 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
#> # i 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.

mutate()

Adds new variables



Artwork by Allison Horst

mutate()

New columns can be added based on values from other columns

```
1 mutate(soybean_use,
2       sum_human_animal = human_food + animal_feed
3     )

#> # A tibble: 9,897 × 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
#> # i 9,894 more rows
```

Add multiple new columns at once:

```
1 mutate(soybean_use,
2       sum_human_animal = human_food + animal_feed,
3       total = human_food + animal_feed + processed
4     )
```

mutate() + case_when()

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

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

```
1 mutate(soybean_use,
2   legislation = case_when(
3     between(year, 1980, 2000) ~ "legislation_1",    # case 1
4     year >= 2000 ~ "legislation_2",                # case 2
5     .default = "no_legislation"                     # all other cases
6   )
7 )
8 #> # A tibble: 9,897 × 7
9 #>   entity code    year human_food animal_feed processed legislation
10 #>   <chr>  <chr>  <dbl>      <dbl>       <dbl>    <dbl> <chr>
11 #>  1 Africa <NA>  1961      33000      6000    14000 no_legislation
12 #>  2 Africa <NA>  1962      43000      7000    17000 no_legislation
13 #>  3 Africa <NA>  1963      31000      7000     5000 no_legislation
14 #>  4 Africa <NA>  1964      43000      6000    14000 no_legislation
15 #>  5 Africa <NA>  1965      34000      6000    12000 no_legislation
16 #>  6 Africa <NA>  1966      41000      6000     2000 no_legislation
17 #>  7 Africa <NA>  1967      47000      6000     4000 no_legislation
18 #>  8 Africa <NA>  1968      50000      7000     3000 no_legislation
19 #>  9 Africa <NA>  1969      52000      6000     6000 no_legislation
20 #> 10 Africa <NA>  1970      52000      6000    8000 no_legislation
21 #> # i 9,887 more rows
```

summarize()

summarizes data

summarize()

summarize will collapse the data to a single row

```
1 summarize(soybean_use,
2   total_animal = sum(animal_feed, na.rm = TRUE),
3   total_human = sum(human_food, na.rm = TRUE)
4 )
5 #> # A tibble: 1 × 2
6 #>   total_animal total_human
7 #>       <dbl>      <dbl>
8 #> 1     942503000  1589729000
```

summarize() by group

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

- summary will be calculated separately for each group

```
1 # summarize the grouped data
2 summarize(soybean_use,
3   total_animal = sum(animal_feed, na.rm = TRUE),
4   total_human = sum(human_food, na.rm = TRUE),
5   .by = year
6 )
7 #> # A tibble: 53 × 3
8 #>   year total_animal total_human
9 #>   <dbl>      <dbl>       <dbl>
10 #>  1 1961      1503000    16994000
11 #>  2 1962      1800000    17326000
12 #>  3 1963      2060000    18667000
13 #>  4 1964      2002000    19639000
14 #>  5 1965      2162000    17796000
15 #>  6 1966      3096000    22179000
16 #>  7 1967      2818000    23282000
17 #>  8 1968      3361000    22747000
18 #>  9 1969      3084000    22212000
19 #> 10 1970      2496000    24119000
20 #> # i 43 more rows
```

count()

Counts observations by group

```
1 # count rows grouped by year
2 count(soybean_use, year)
3 #> # A tibble: 53 × 2
4 #>   year     n
5 #>   <dbl> <int>
6 #>   1     1961    178
7 #>   2     1962    178
8 #>   3     1963    178
9 #>   4     1964    178
10 #>  5     1965    178
11 #>  6     1966    178
12 #>  7     1967    178
13 #>  8     1968    178
14 #>  9     1969    178
15 #> 10    1970    178
16 #> # i 43 more rows
```

The pipe |>

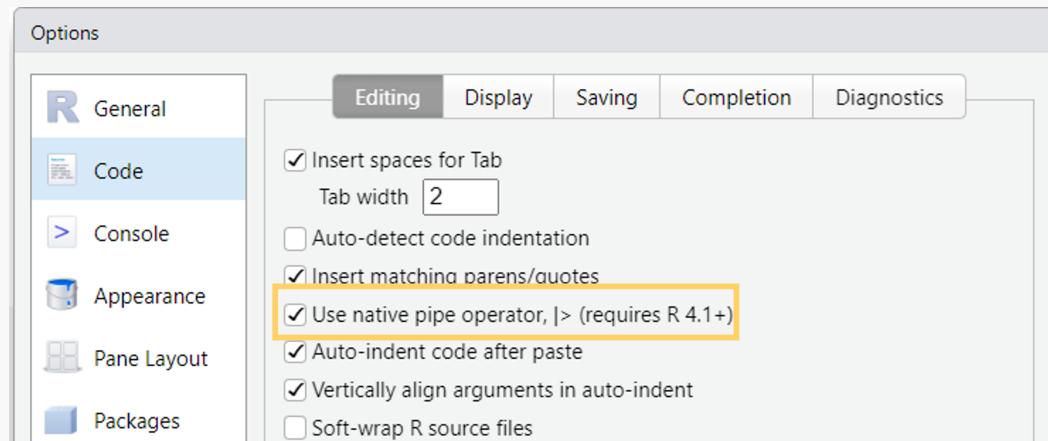
Combine multiple data operations into one command

The pipe |>

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

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

- 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 # 1: filter rows that actually represent a country
2 soybean_new <- filter(soybean_use, !is.na(code))
3
4 # 2: summarize mean values by year
5 soybean_new <- summarize(soybean_new,
6   mean_processed = mean(processed, na.rm = TRUE),
7   sd_processed = sd(processed, na.rm = TRUE),
8   .by = year
9 )
```

How could we make this more efficient?

The pipe |>

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

```
1 soybean_new <- summarize(  
2   filter(soybean_use, !is.na(code)),  
3   mean_processed = mean(processed, na.rm = TRUE),  
4   sd_processed = sd(processed, na.rm = TRUE),  
5   .by = year  
6 )
```

But this gets complicated and error prone very quickly

The pipe |>

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

```
1 soybean_new <- soybean_use |>
2   filter(!is.na(code)) |>
3   summarize(
4     mean_processed = mean(processed, na.rm = TRUE),
5     sd_processed = sd(processed, na.rm = TRUE),
6     .by = year
7   )
```

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.

```
1 soybean_use |>
2   count(year)
3
4 # instead of
5 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.



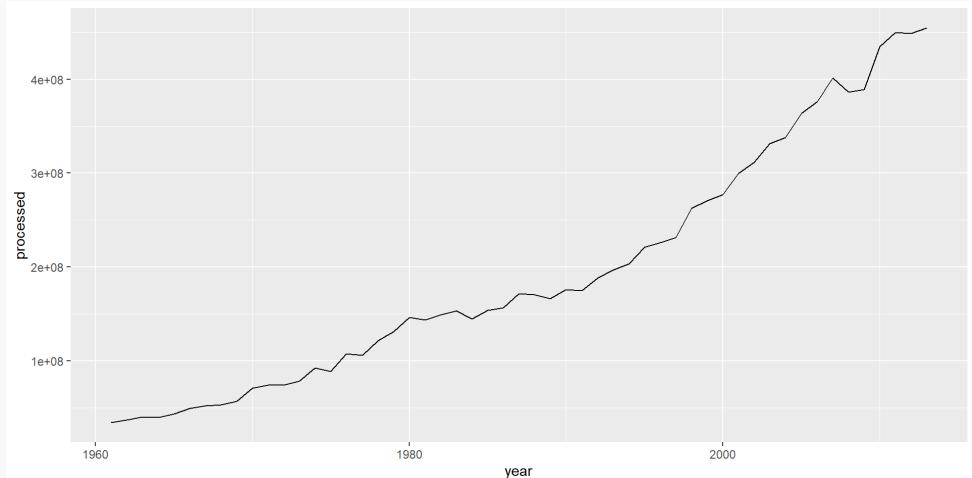
Tip

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

The pipe |>

Piping also works well together with `ggplot`

```
1 soybean_use |>
2   filter(!is.na(code)) |>
3   select(year, processed) |>
4   summarize(
5     processed = sum(processed,
6       na.rm = TRUE
7   ),
8   .by = year
9 ) |>
10 ggplot(aes(
11   x = year,
12   y = processed
13 )) +
14   geom_line()
```



Combining multiple tables

Combine two tibbles by row `bind_rows`

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

```
1 tbl_a <- soybean_use[1:2, ] # first two rows
2 tbl_b <- soybean_use[2:nrow(soybean_use), ] # the rest
```

```
1 tbl_a
```

```
#> # A tibble: 2 × 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
```

```
1 tbl_b
```

```
#> # A tibble: 9,896 × 6
#>   entity code    year human_food animal_feed processed
#>   <chr>  <chr> <dbl>      <dbl>      <dbl>
#> 1 Africa <NA>    1962     43000      7000     17000
#> 2 Africa <NA>    1963     31000      7000      5000
#> # i 9,894 more rows
```

Combine two tibbles by row `bind_rows`

Bind the rows together with `bind_rows()`:

```
1 bind_rows(tbl_a, tbl_b)

#> # A tibble: 9,898 × 6
#>   entity code    year human_food animal_feed processed
#>   <chr>  <chr>  <dbl>     <dbl>      <dbl>      <dbl>
#> 1 Africa <NA>    1961     33000     6000     14000
#> 2 Africa <NA>    1962     43000     7000     17000
#> # i 9,896 more rows
```

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

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

#> # A tibble: 9,898 × 7
#>   id    entity code    year human_food animal_feed processed
#>   <chr> <chr>  <chr>  <dbl>     <dbl>      <dbl>      <dbl>
#> 1 a     Africa <NA>    1961     33000     6000     14000
#> 2 a     Africa <NA>    1962     43000     7000     17000
#> 3 b     Africa <NA>    1962     43000     7000     17000
#> # i 9,895 more rows
```

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

```
1 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.

```
1 soybean_use

#> # A tibble: 9,897 × 6
#>   entity code    year human_food animal_feed processed
#>   <chr>  <chr> <dbl>      <dbl>       <dbl>      <dbl>
#> 1 Africa <NA>    1961     33000      6000     14000
#> 2 Africa <NA>    1962     43000      7000     17000
#> # i 9,895 more rows
```

```
1 # table with the gdp of the country/continent for each year
2 gdp

#> # A tibble: 9,897 × 3
#>   entity  year    gdp
#>   <chr>   <dbl>  <dbl>
#> 1 Africa   1961  4.73
#> 2 Africa   1962  4.73
#> # i 9,895 more rows
```

Join tibbles with `left_join()`

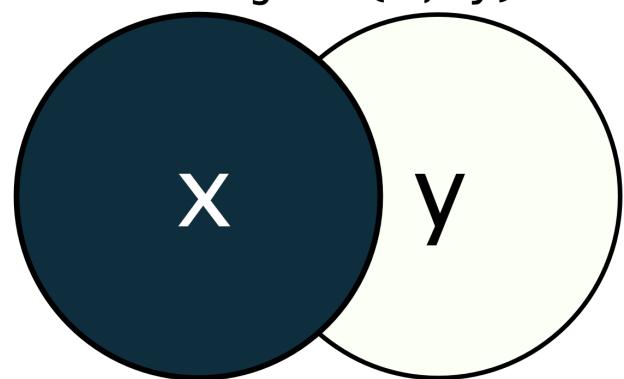
Join the two tables by the two common columns `entity` and `year`

```
1 left_join(soybean_use, gdp, by = c("entity", "year"))
2 #> # A tibble: 9,897 × 7
3 #>   entity code    year human_food animal_feed processed     gdp
4 #>   <chr>  <chr> <dbl>      <dbl>       <dbl>      <dbl> <dbl>
5 #> 1 Africa <NA>  1961      33000      6000      14000    4.73
6 #> 2 Africa <NA>  1962      43000      7000      17000    4.73
7 #> 3 Africa <NA>  1963      31000      7000      5000     4.73
8 #> 4 Africa <NA>  1964      43000      6000      14000    4.73
9 #> 5 Africa <NA>  1965      34000      6000      12000    4.73
10 #> 6 Africa <NA>  1966      41000      6000      2000     4.73
11 #> 7 Africa <NA>  1967      47000      6000      4000     4.73
12 #> 8 Africa <NA>  1968      50000      7000      3000     4.73
13 #> 9 Africa <NA>  1969      52000      6000      6000     4.73
14 #> 10 Africa <NA> 1970      52000      6000      8000     4.73
15 #> # i 9,887 more rows
```

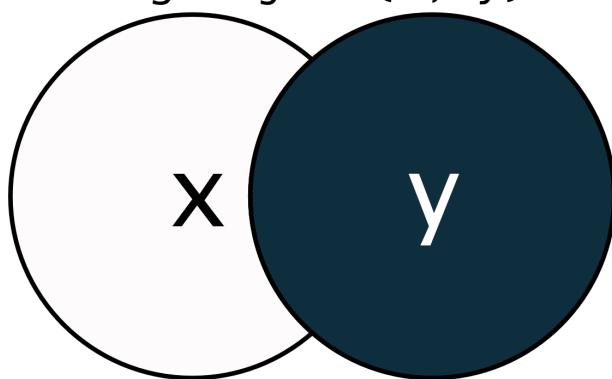
`left_join()` means that the resulting tibble will contain all rows of `soybean_use`, but not necessarily all rows of `gdp`

Different *_join() functions

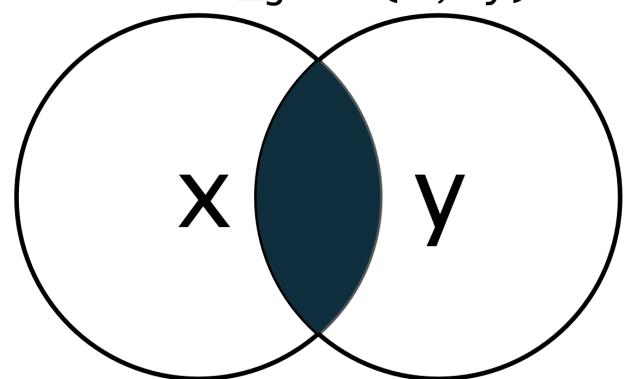
`left_join(x, y)`



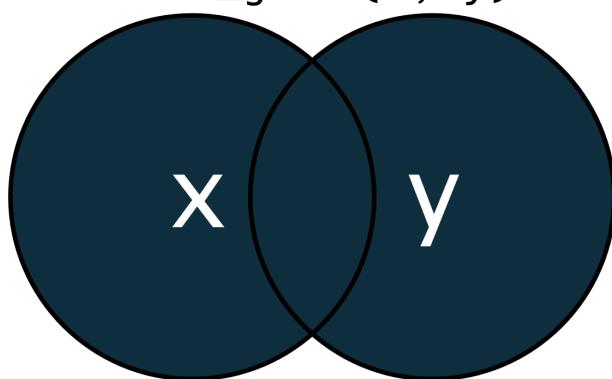
`right_join(x, y)`



`inner_join(x, y)`



`full_join(x, y)`



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 III

`summarize()`

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

`count`

- count rows based on a group

Summary IV

`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](#)

