

5. Data Wrangling

Self-Paced Workshop #5

Prof. Jack Reilly

Fall 2026

The Wrangling Verbs

Last week's workshop was about one command, `mutate()`, which changes the *values* in a dataset. This week's lecture was about changing its *structure*: keeping some rows and columns and dropping others, gluing two datasets together, collapsing many rows into a few summary numbers, weighting a survey, and reshaping a table from wide to long. That is a lot of commands, but they share one grammar — a dataset, a pipe, a verb — and each one on its own is small.

This page gives you the smallest possible example of each. Run every box and read every printout, and by the end the tidyverse cheatsheets will read like a menu instead of a foreign language.

Before you start

Do [Self-Paced Workshop #4](#) first if you haven't; this page assumes you know what `|>` and `mutate()` do. As before, the boxes share one R session — run them top to bottom, and if you see `object not found`, re-run the box above that creates it.

Step 1. Load the packages

Three packages this time. `dplyr` has the row, column, join, and summary verbs; `tidyr` has the reshaping verbs; `srvyr` handles survey weights.¹ Run this box and wait for it to finish — the browser has to fetch the packages the first time, which can take a moment.

```
library(dplyr)
library(tidyr)
library(srvyr)
```

¹In your own R, `library(tidyverse)` loads `dplyr` and `tidyr` (and several other packages) in one line. `srvyr` is not part of the tidyverse bundle, so it needs its own `install.packages("srvyr")` once and its own `library(srvyr)` each session. Any notes about “masked” objects are normal.

Step 2. A tiny dataset

Nine survey respondents across three election years, with whether they voted (turnout, 1 = yes) and their party (D, R, or I).

```
votes <- tibble(
  id      = 1:9,
  year    = c(2012, 2012, 2012, 2016, 2016, 2016, 2020, 2020, 2020),
  turnout = c(1, 0, 0, 1, 1, 0, 1, 1, 1),
  party   = c("D", "D", "R", "R", "I", "D", "I", "D", "R")
)
votes
```

Step 3. filter(): keep the rows you want

filter() keeps the rows for which a question is true and drops the rest. To keep only the 2020 respondents:

```
votes |>
  filter(year == 2020)
```

Three rows.² Remember: two equals signs == asks a question: whether things are equal. (One equals sign, as in mutate(x = ...), assigns something.) The opposite question is !=, “not equal to”:

```
votes |>
  filter(year != 2020)
```

Several questions separated by commas must *all* be true — “and.” Respondents who are not from 2020 *and* who voted:

```
votes |>
  filter(year != 2020, turnout == 1)
```

Try it yourself: keep only the Democrats who turned out to vote. Text values need quotes, “D”; numbers do not.³

```
votes |>
  filter(party == "D")    # add a second condition for turnout
```

filter() does not change votes itself — like mutate(), it shows you a result, and you keep it only by assigning with <-. We’ll leave votes whole for the rest of the page.

Step 4. select(): keep the columns you want

filter() picks rows; select() picks columns. List the ones to keep:

²Respondents 7, 8, and 9.

³You should end up with respondents 1 and 8.

```
votes |>
  select(id, year, turnout)
```

Or name the ones to drop, with a minus sign. This gives the same result:

```
votes |>
  select(-party)
```

Verbs chain. “Take votes, and then keep three columns, and then keep the 2020 rows”:

```
votes |>
  select(id, year, turnout) |>
  filter(year == 2020)
```

Try it yourself: in one pipeline, keep just id and party for the 2016 respondents.

```
votes |>
  select(id, party)   # and then filter to 2016
```

Step 5. Stacking datasets with `bind_rows()`

Suppose the 2024 wave arrives as its own file, with the same columns. To add more *rows* — the same variables, new cases — stack the two with `bind_rows()`:

```
votes_2024 <- tibble(
  id      = 10:12,
  year    = c(2024, 2024, 2024),
  turnout = c(1, 1, 0),
  party   = c("R", "I", "D")
)

all_votes <- bind_rows(votes, votes_2024)
all_votes
```

Twelve rows, all of them printed.⁴ `bind_rows()` matches columns by name, not by position, so the names have to agree exactly; a column that one dataset has and the other doesn’t gets filled with NA. A quick way to check the result is to count rows per year:

```
all_votes |>
  count(year)
```

Step 6. Joining datasets with `left_join()`

The other way to combine data is to add more *columns* for the same cases — a merge or, in tidyverse language, a **join**. Two datasets about the same people, each with an id, and each missing one person the other has:

⁴A tibble prints every row up to twenty. Beyond that it shows only the first ten, plus a note such as 2,990 more rows, so a real survey never floods your screen — the “concise data view” mentioned in lecture, and one reason to prefer tibbles to plain data frames.

```
ages <- tibble(id = 1:3, age = c(25, 30, 35))
incomes <- tibble(id = 2:4, income = c(40, 50, 60))
ages
incomes
```

A `left_join()` keeps every row of the *left* (first) dataset and attaches whatever matches from the right, lining rows up by the key you name in `by =`:

```
left_join(ages, incomes, by = "id")
```

Person 1 is kept, with NA for income (nothing to match). Person 4 is gone: they were never in `ages`. The other joins differ only in which rows survive:

```
inner_join(ages, incomes, by = "id") # only ids in both
full_join(ages, incomes, by = "id")  # every id from either
```

Try it yourself: change `left_join` to `right_join` in the box below, and predict the result before you run it.⁵

```
left_join(ages, incomes, by = "id")
```

Joins are where data quietly goes missing, so always look at what did *not* match. `anti_join()` shows the rows of the first dataset that have no partner in the second:

```
anti_join(ages, incomes, by = "id")
```

Joins also do not have to be one-to-one. Here five people live in two counties, and we want each person to carry their county's election result. Many people, one county — a left join handles it, repeating the county's value for everyone who lives there:

```
people <- tibble(person = 1:5,
                  county = c("A", "A", "B", "B", "B"))
results <- tibble(county = c("A", "B"),
                  dem_share = c(45, 55))

left_join(people, results, by = "county")
```

Step 7. Collapsing data with `group_by()` and `summarise()`

Often the question is not about your rows but about groups of them: not students but classrooms, not respondents but years. Seven students in three classes:

```
scores <- tibble(
  id    = 1:7,
  class = c("A", "A", "B", "B", "B", "C", "C"),
  score = c(80, 70, 90, 80, 70, 95, 85)
)
scores
```

⁵`right_join()` keeps every row of the *right* dataset, `incomes`: ids 2, 3, and 4, with NA for person 4's age.

`summarise()` boils a dataset down to one row of summary numbers. The mean score for everyone:

```
scores |>
  summarise(mean_score = mean(score))
```

Put a `group_by()` in front of it, and the same `summarise()` runs once per group instead — one row per class:

```
scores |>
  group_by(class) |>
  summarise(mean_score = mean(score))
```

Notice what happened to the data: it now has three rows, one per class, and no `id` column. The unit of analysis changed from the student to the class. You can ask for several summaries at once; `n()` counts the rows in each group:

```
scores |>
  group_by(class) |>
  summarise(mean_score = mean(score),
            sd_score    = sd(score),
            N           = n(),
            .groups     = "drop")
```

The last line, `.groups = "drop"`, tells R to forget the grouping once it has finished summarising — a good habit, since a still-grouped dataset behaves differently in later steps.⁶

One trap. If a value is missing, `mean()` gives up and returns `NA`:

```
scores_missing <- tibble(
  class = c("A", "A", "B", "B"),
  score = c(80, NA, 90, 80)
)

scores_missing |>
  group_by(class) |>
  summarise(mean_score = mean(score))
```

Tell it to remove the missing values first with `na.rm = TRUE`, and class A gets a mean again:

```
scores_missing |>
  group_by(class) |>
  summarise(mean_score = mean(score, na.rm = TRUE))
```

Try it yourself: back in `votes`, the mean of turnout is the turnout *rate*. Add a `turnout_rate = mean(turnout)` summary to the box below to get the rate for each year.⁷

⁶With one grouping variable, `summarise()` drops the grouping by itself; with two or more (`group_by(school, class)`) it keeps the outer ones and prints a message about it. Writing `.groups = "drop"` makes the behaviour the same either way.

⁷0.333 in 2012, 0.667 in 2016, and 1 in 2020.

```
votes |>
  group_by(year) |>
  summarise(N = n())
```

Step 8. Survey weights with `srvyr`

Real surveys are rarely perfectly representative, so they ship with a weight variable: a number per respondent saying how many people that respondent should stand for. Here is a small poll that over-recruited college graduates, with a weight `pwt` that corrects for it — the two high-school respondents count double, the three graduates count half.

```
poll <- tibble(
  id      = 1:6,
  educ    = c("HS", "HS", "SomeCollege", "BAplus", "BAplus", "BAplus"),
  age     = c(55, 48, 40, 30, 28, 27),
  turnout = c(0, 1, 0, 1, 1, 1),
  pwt     = c(2, 2, 1, 0.5, 0.5, 0.5)
)

poll |>
  summarise(mean_age = mean(age), turnout_rate = mean(turnout))
```

Those are the *unweighted* answers, and they lean young and high-turnout because of who was over-recruited. To use the weights, first tell R about the survey's design — which column is the weight — and store that as a new object:

```
poll_design <- as_survey_design(poll, weights = pwt)
```

Then summarise the *design* object with `survey_mean()` in place of `mean()`:

```
poll_design |>
  summarise(mean_age      = survey_mean(age),
            turnout_rate = survey_mean(turnout))
```

Older, and lower turnout, once the weights are applied.⁸ The extra `_se` columns are standard errors — how uncertain each weighted estimate is — which you can ignore for now.

Try it yourself: `group_by()` works on design objects too. Add `group_by(educ) |>` before the `summarise()` below to get a weighted mean age for each education level.

```
poll_design |>
  summarise(mean_age = survey_mean(age))
```

Step 9. Reshaping with `pivot_longer()` and `pivot_wider()`

Finally, the same data can be laid out in more than one shape. Two students' grades in two classes, in **wide** form — one row per student, one column per class:

⁸A mean age of about 44.4 rather than 38, and a turnout rate of about 0.54 rather than 0.67.

```
grades <- tibble(
  student = c("Alice", "Bob"),
  stat     = c(90, 80),
  policy   = c(85, 88)
)
grades
```

`pivot_longer()` turns it **long** — one row per student *per class*. You tell it which columns to fold up (`cols`), what to call the column that will hold their *names* (`names_to`), and what to call the column that will hold their *values* (`values_to`):

```
grades_long <- grades |>
  pivot_longer(cols = c(stat, policy),
               names_to = "subject",
               values_to = "score")
grades_long
```

Four rows: the two class names have become values in a subject column, and the four grades are stacked in a score column. This is the tidy shape for most of what you'll do with `ggplot2` next week. `pivot_wider()` is the exact reverse: say which column supplies the new column *names* and which supplies the *values*.

```
grades_long |>
  pivot_wider(names_from = subject,
              values_from = score)
```

Try it yourself: there is a third shape. Change `names_from = subject` to `names_from = student` and see what the rows and columns become. Which shape you want depends on what one row is supposed to *be*.

```
grades_long |>
  pivot_wider(names_from = subject,
              values_from = score)
```

Work on Your Own

That is every verb from this week's lecture, each in its smallest form. [Problem Set 6](#) uses them at scale on the General Social Survey: `filter()` to pick years, `mutate()` to recode, `group_by()` and `summarise()` to get a rate per year, and the survey weights to make those rates representative. Keep the [dplyr](#) and [tidyr](#) cheatsheets open while you work.