

4. Mutate & Recode

Self-Paced Workshop #4

Prof. Jack Reilly

Fall 2026

Making New Variables with `mutate()`

In lecture this week we met the tidyverse and one of its most-used commands: `mutate()`, which makes a new column in a dataset (or changes an existing one). Almost every data project starts with a round of this — recoding a survey question, turning ages into age groups, combining two columns into one. This workshop practices the two ways it shows up most: doing simple math on a column, and recoding a variable with `case_when()`.

Everything runs here in your browser, as in the first self-paced workshop, so you don't need to open R. The commands are exactly the ones you will type into RStudio or Positron for Problem Set 5.

Run the boxes in order

The code boxes on this page share one R session, just as the lines of a script do. Each box builds on the ones above it, so work top to bottom. If you ever see object 'survey' not found, scroll back up and re-run the box that created it.

Step 1. Load dplyr

`mutate()` lives in a package called `dplyr` (say it “dee-plier”), one of the core pieces of the tidyverse. Like any package, it has to be loaded before you can use it. Click “Run Code”:

```
library(dplyr)
```

You may see nothing at all, or a note that a few objects are “masked.”¹ Either way, the package is now loaded for the rest of the page. In your own R installation, `library(tidyverse)` loads `dplyr` along with all of its sibling packages at once; here we load only the piece we need, so the page starts faster. The commands are identical either way.

¹The note — something like The following objects are masked from 'package:stats': `filter`, `lag` — is normal: it just means `dplyr` has its own versions of a few commands, and R will now use those. Whether it shows up in the browser varies; in RStudio or Positron you will always see it.

Step 2. A tiny dataset

Let's build a small survey by hand. Ten respondents, each with an id, an age, and their answer to the party identification question from lecture: pid runs from 1 (strong Democrat) to 7 (strong Republican), with 8 meaning "don't want to answer" and 9 meaning "don't know."

```
survey <- tibble(  
  id = 1:10,  
  pid = c(1, 6, 3, 7, 8, 9, 2, 2, 4, 5),  
  age = c(34, 61, 22, 45, 29, 52, 38, 71, 26, 40)  
)  
survey
```

`tibble()` is the tidyverse's version of a data frame. Look at the printout. The first line tells you the size (10 × 3: ten rows, three columns), and under each column name is its type (<int> for whole numbers, <dbl> for numbers that can have decimals). You will read a lot of these headers this semester.

Step 3. `mutate()`: a new column from math

Here is the simplest possible `mutate()`. Run it, then we will take it apart.

```
survey |>  
  mutate(birth_year = 2026 - age)
```

You should see the same ten rows with a new fourth column, `birth_year`.² Piece by piece:

1. `survey` — *start with the survey data*
2. `|>` — *and then*. This is the **pipe**. It hands whatever is on its left to the command on its right. Read it aloud as "and then" and tidyverse code turns into sentences.
3. `mutate()` — *change the data by adding a column*
4. `birth_year =` — *call the new column birth_year*. Note the **single** equals sign: inside `mutate()`, `=` means "name this."
5. `2026 - age` — *and fill it with 2026 minus each respondent's age*. R does the subtraction once for every row.

Try it yourself. Change the code so the new column is called `age_next_year` and holds each respondent's age plus one.

```
survey |>  
  mutate(birth_year = 2026 - age)
```

You can make more than one column at a time; separate them with commas. Notice what the second new column here contains.³

²Its first value should be 1992: respondent 1 is 34, and $2026 - 34 = 1992$.

³TRUE or FALSE for each row (its type prints as <lgl>, for logical). `age > 40` is a question, and R answers it once per respondent. Keep this in mind for Step 5: recoding is built entirely out of questions like this one.

```
survey |>
  mutate(birth_year = 2026 - age,
         over_40 = age > 40)
```

Step 4. Saving your work

Run this box and look carefully at what prints:

```
survey
```

Still three columns. `mutate()` *showed* you a new version of the data but did not *keep* it — nothing in R is saved unless you store it in an object with `<-`. That is a feature: you can try a change, look at the result, and only commit to it once it looks right. To keep the new column, assign the result back to `survey`:

```
survey <- survey |>
  mutate(birth_year = 2026 - age)
survey
```

And then call the object `survey` again to see the change:

```
survey
```

Now `survey` has four columns, and will keep them for the rest of the page. The original `age` column is still there: `mutate()` adds; it never throws away what you started with.

Step 5. Recoding with `case_when()`

The real work of `mutate()` is recoding: taking a variable coded one way and making a new variable coded another. In lecture we collapsed the seven-point `pid` scale into three categories — Democrats, independents, Republicans — and set the “don’t know” codes to missing. Here it is:

```
survey |>
  mutate(pid3 = case_when(
    pid %in% 1:2 ~ -1,
    pid %in% 3:5 ~ 0,
    pid %in% 6:7 ~ 1,
    pid == 8 | pid == 9 ~ NA
  ))
```

Compare the `pid` and `pid3` columns row by row and check that each respondent landed where they should. Now the structure:

- `mutate(pid3 = ...)` — the same as before: make a new column called `pid3`.
- `case_when()` — *fill it according to the following rules.*
- Each rule has the form **question ~ answer**. The squiggle `~` (a tilde) separates the two: *when the question is true, write down the answer*. The rules are separated by commas.
- `pid %in% 1:2` asks *is pid one of 1, 2?* (`1:2` means “the numbers 1 through 2,” and `%in%` means “is one of.”)

- `pid == 8 | pid == 9` asks *is pid 8, or is it 9?* Two equals signs `==` ask whether things are equal; the vertical bar `|` means “or.” The answer for these respondents is `NA`, which is how R writes missing data.

A respondent for whom no rule is true gets `NA` automatically, so the last line is not strictly necessary — but writing it out makes your intentions plain to whoever reads the code next, including you.

This time, save the result:

```
survey <- survey |>
  mutate(pid3 = case_when(
    pid %in% 1:2 ~ -1,
    pid %in% 3:5 ~ 0,
    pid %in% 6:7 ~ 1,
    pid == 8 | pid == 9 ~ NA
  ))
```

And then call ‘survey’ back again to see the change:

```
survey
```

Step 6. Check your work

With ten rows you can check a recode by eye. With ten thousand you cannot, so get in the habit of asking for a table that lines the old variable up against the new one. `count()` does this: it lists every combination that occurs, and how often.

```
survey |>
  count(pid, pid3)
```

Read down the rows: every `pid` of 1 or 2 should sit next to a `pid3` of -1, and so on, with 8 and 9 next to `NA`. If any row surprises you, one of your rules is wrong.⁴

Step 7. Try it yourself

Words, not numbers. The answer to a `case_when()` rule doesn’t have to be a number. Finish the recode below so that `party` reads “Democrat”, “Independent”, or “Republican” (text goes in quotes), and check it with `count()`. Don’t forget the comma at the end of each rule but the last.

```
survey |>
  mutate(party = case_when(
    pid %in% 1:2 ~ "Democrat"
    # add the Independent and Republican rules here
  )) |>
  count(pid, party)
```

⁴Base R’s `table(survey$pid, survey$pid3)` gives the same information as a grid — the `$` pulls a single column out of a dataset. It silently leaves out the `NA` rows unless you ask: `table(survey$pid, survey$pid3, useNA = "ifany")`.

Rules are checked in order. `case_when()` works down the rules top to bottom and uses the *first* one that is true for a respondent. Run this:

```
survey |>
  mutate(age_group = case_when(
    age < 30 ~ "under 30",
    age < 50 ~ "30 to 49",
    age >= 50 ~ "50 and up"
  )) |>
  count(age_group)
```

A 26-year-old satisfies both `age < 30` and `age < 50`, but only the first matching rule counts, so they land in “under 30.” Now swap the first two rules and run it again. What happened to the under-30s, and why?⁵

Step 8. For comparison: the base R way

Before the tidyverse, recoding meant square brackets. This is the version from the first half of lecture, and you will still see it in the wild.

```
survey$pid3_base <- NA
survey$pid3_base[survey$pid <= 2] <- -1
survey$pid3_base[survey$pid >= 3 & survey$pid <= 5] <- 0
survey$pid3_base[survey$pid >= 6 & survey$pid <= 7] <- 1
survey |>
  count(pid3, pid3_base)
```

Read each line from the inside out. `survey$pid <= 2` is a question asked of every row; the brackets keep only the rows where the answer is TRUE; the arrow writes -1 into `pid3_base` for exactly those rows. (The `&` means “and.”) The `count()` at the end confirms that the two recodes agree.

Both approaches are correct. The `case_when()` version reads like a list of rules, is harder to get wrong — notice how easy it would be to forget the `<= 7` and sweep the 8s and 9s in with the Republicans — and scales up to the dozen-variable recodes that real projects need. That is why the tidyverse is what we’ll use from here on.

Work on Your Own

Now head to [Problem Set 5](#). Every recode it asks for — ideology into three categories, vote into two parties, education into four levels — is a `mutate()` with a `case_when()` inside, followed by a `count()` to check it. Keep the [dplyr cheatsheet](#) open while you work.

⁵With `age < 50` first, everyone under 50 — including the 22-, 26-, and 29-year-olds — matches that rule and is filed under “30 to 49.” Nobody is left for “under 30.” When rules overlap, order them from most specific to most general.