Lesson 9

by Lorraine Gaudio

Lesson generated on October 15, 2025

BOISE STATE UNIVERSITY

Contents

1. 🧹 Clean up on aisle R!

In lesson 8, we learned about about subsetting with dplyr (select() & filter()). This week, we will look at other functions in the dplyr package (mutate() and case_when()). Real-world data often needs new or cleaner variables before analysis. The dplyr verb mutate() lets you create or modify columns, while case_when() supplies the logic for quick recoding. Together they form a duo you will use in nearly every data project.

To begin Lesson 9, follow these steps:

1. Open your course project for RStudio
2. Create a new file. From the file types we have used so far, pick which file type you want to use. (File > New File > RMarkdown (.Rmd) | Quarto Document (.qmd)).
3. Type in the code provided in this document as you follow along with the video. Pause the video at anytime to answer assignment questions, dig deeper or add memo notes.

Lesson Overview

By the end of Lesson 9 you will be able to:

1. □ Remember – State the purposes of mutate() and case_when()
2. □ Understand – Explain how case_when() replaces values based on a test.
3. □ Apply – Overwrite or add columns using mutate() + case_when()
4. □ Analyze – Build multi□ level recodes with nested case_when().
5. □ Evaluate – Choose between overwriting vs. keeping the original data.

Keep these goals in mind as you move through each section.

2. 📦 Packages

Install once (if needed): install.packages("dplyr"); install.packages("dslabs")

Load the packages at the start of every session:

```
library(dslabs)  # Data science labs package
library(dplyr)   # Data manipulation package
```

3. Warm-Up

3.1 Logical Test Review

 Goal: To write a simple logical test for numbers one through five showing TRUE for when the numbers are larger than 3.

 Prediction: I expect 4 and 5 to be TRUE.

```
# Code: TRUE appears where the test is met.
c(1, 2, 3, 4, 5) > 3
```

 Check / Comment: Did the output met your expectation?

3.2 **sample()** Review

Recall sample() is a base-R function that randomly selects elements from a vector.

 Goal: Randomly select 5 numbers from 1 to 10.

```
# Code:
set.seed(99)  # Ensures reproducibility
sample(1:10, 5)  # Sample 5 numbers from 1 to
```

 Check / Comment: Confirm the output length is 5 and numbers are between 1 and 10.

4. sample_n()

□ The SYNTAX:sample_n(data, size, replace = FALSE, weight = NULL)

□ Goal: Randomly select 5 rows from the mtcars dataset.

```
# □ Code:
sample_n(mtcars, 5)  # Sample 5 rows from mtcars data frame
```

```
# □ Code:
mtcars2 <- mtcars %>% sample_n(5)  # Using with pipe
```

□ Check / □ Comment: Confirm the output has 5 rows and all original columns.

□ Reflect: Why are the selected row numbers different each time you run the code?

We'll come back to sample_n() later in this lesson.

5. ifelse() vs case_when()

5.1 ifelse() tests

 The SYNTAX: ifelse(test, yes, no)

- *test* – logical vector (TRUE/FALSE/NA)
- *yes* – value to insert where test is TRUE
- *no* – value to insert where test is FALSE or NA

 Goal: Label numbers > 3 as "Large" and others as "Small".

 Prediction: I expect to see a vector of "Small" and "Large" labels. Specifically, I expect only 4 and 5 to be "Large".

```
ifelse(test = c(1, 2, 3, 4, 5) > 3, yes = "Large", no = "Small")
```

 Check / Comment: Confirm the output positions for 1:3 are "Small" and 4:5 are "Large".

5.2 case_when() tests

dplyr::case_when() can do exactly what ifelse() is doing.

 The SYNTAX: case_when(condition1 ~ value1, condition2 ~ value2, TRUE ~ default_value)

The ~ symbol means "maps to" or "yields".

 Goal: Label numbers > 3 as "Large" and others as "Small".

 Prediction: I expect to see a vector of "Small" and "Large" labels. Specifically, I expect only 4 and 5 to be "Large".

```
case_when(c(1, 2, 3, 4, 5) > 3 ~ "Large", TRUE ~ "Small")
```

 Check / Comment: Confirm the output positions for 1:3 are "Small" and 4:5 are "Large"

So, what's the point? Why do we need both ifelse() and case_when()? case_when() is usually clearer once you have 3+ categories.

5.3 Multi‑Level Recoding

Nested ifelse()

 Goal: Re-code numbers 1 to 5 as “One”, “Two”, “Three”, “Four”, or “Five”.

 Prediction: I expect to see a vector of the words “One” through “Five” corresponding to the numbers 1 through 5.

```
# Code:
object2 <- c(1, 1, 2, 2, 3, 3, 4, 4, 5, 5)
length(object2) # Check the length of object2
```

```
# Code
ifelse(object2 == 1, "One",
       ifelse(object2 == 2, "Two",
              ifelse(object2 == 3, "Three",
                     ifelse(object2 == 4, "Four", "Five"))))
```

 Check / Comment: Confirm the output length is 10 and the labels match the numbers.

Nested case_when()

 Goal: Re-code numbers 1 to 5 as “One”, “Two”, “Three”, “Four”, or “Five”.

 Prediction: The case_when() output should be identical in length and order to the nested ifelse() version.

```
# Code
case_when(object2 == 1 ~ "One",
          object2 == 2 ~ "Two",
          object2 == 3 ~ "Three",
          object2 == 4 ~ "Four",
          object2 == 5 ~ "Five")
```

 Check / Comment: Confirm the output is identical to the ifelse() version.

 What I learned: case_when() is easier to read because the function name is not repeated at each line.

Apply this nested case_when() in Task 3 of the assignment.

6. mutate()

mutate() is a function that adds new columns or modifies existing ones

 The SYNTAX: mutate(data, new_col = value)

6.1 Overwrite Columns

6.1.1 Copy A Column

 Goal: Create a new column that is a copy of an existing column.

```
# Preview Column Names
names(mtcars) # Check the column names
```

 Prediction: In a new data set called mtcars2, a **new** column named new_column_copy is created that is identical to the am column.

```
#  Code:
mtcars2 <- mtcars %>%
  mutate(new_column_copy = am)
```

```
#  Verify:
names(mtcars2) # Check the column names again
```

 Comment: Did you see the new column in the output?

```
#  Verify: is am == to n_column_copy?
mtcars2$am == mtcars2$new_column_copy
```

```
#  Verify: is am identical to new_column_copy?
identical(mtcars2$am, mtcars2$new_column_copy)
```

 Comment: mutate(new_column_copy = am) created an identical column of am called new_column_copy

6.1.2 Overwrite Original Column Values

 Goal: Recode the am column in mtcars from 0/1 to "Automatic"/"Manual".

```
# Check the unique values in the am column BEFORE
unique(mtcars$am)
```

 Prediction: In a new data set called cars_overwrite, the am column is changed from 0/1 to "Automatic"/"Manual". In a second new data set called cars_keep, a **new** column named transmission is created that has "Automatic"/"Manual" values, while the original am column remains unchanged.

```
#  Code:
cars_overwrite <- mtcars %>%
  mutate(am = case_when(am == 0 ~ "Automatic", TRUE ~ "Manual"))
```

```
#  Verify:
unique(cars_overwrite$am)
```

6.1.3 New Column & New Values

 Goal: Create a new column transmission that recodes am from 0/1 to "Automatic"/"Manual", while keeping the original am column unchanged.

```
#  Code:
cars_keep <- mtcars %>%
  mutate(transmission = case_when(am == 0 ~ "Automatic", TRUE ~ "Manual"))
```

```
#  Verify:
cars_keep %>% select(am, transmission)
```

 Comment: What is the difference between the code that creates cars_overwrite and cars_keep?

 Metacognition: Why might you want to create a copy of a column when changing the values?

6.2 Factors and Levels

In Lesson 2, we learned about factors for vectors. A factor is a categorical variable that can take on a limited number of values, called levels. For example, in the vector tshirt_sizes_data, the possible values are "S", "M", "L", and "XL".

```
#  Creating a character vector
tshirt_sizes_data <- c("M", "L", "S", "M", "L", "XL", "S")
#  Changing the character vector to a factor class with levels
tshirt_sizes <- factor(tshirt_sizes_data, levels = c("S", "M", "L", "XL"))
```

```r
# □ Investigating your factor class with levels
unique(tshirt_sizes)
```

In the dataset iris, the Species column is a factor with three levels: "setosa", "versicolor", and "virginica".

```r
data(iris)
?iris
```

```r
# □ Investigating the Species column
unique(iris$Species)
class(iris$Species)
levels(iris$Species)
```

6.2.1 Overwrite with Factors

□ Goal: Take a 100-row sample from movielens, then overwrite the numeric rating column with an ordered factor: "Bad" if < 4, "Good" if >= 4, using case_when() inside mutate().

□ Prediction: Before: rating is numeric. After: rating is an ordered factor with levels Bad < Good. Row count stays 100; no NAs created.

□ SYNTAX reminder: mutate(data, new_col = case_when(cond1 ~ value1, cond2 ~ value2, TRUE ~ default))

name_data <- data %>% mutate(new_col = case_when(cond1 ~ value1, cond2 ~ value2, TRUE ~ default))

```r
data("movielens")
?movielens
```

```r
set.seed(99)

# Sample 100 movies from the movielens dataset
smaller_movies <- movielens %>%
  sample_n(100) %>%
  select(title, rating)

head(smaller_movies)
```

```r
# □ Checks (before)
names(smaller_movies)              # expect "title", "rating"
class(smaller_movies$rating)       # expect "numeric"
summary(smaller_movies$rating)     # quick distribution check
```

□ Goal: We'll apply mutate() to smaller_movies to recode the rating column to "Good" or "Bad".

```r
# □ Overwrite numeric rating with an ordered factor via case_when()
movies_overwrite <- smaller_movies %>%
  mutate(
    rating = factor(
      case_when(
        rating >= 4 ~ "Good",
        rating < 2 ~ "Bad",
        TRUE       ~ "Ok"
      ),
      levels = c("Bad", "Ok", "Good"),
      ordered = TRUE
    )
  )
```

```r
# □ Checks (after)
names(movies_overwrite)              # still "title", "rating"
```

```r
# □ Checks (after)
is.ordered(movies_overwrite$rating)  # expect TRUE
```

```r
# □ Checks (after)
levels(movies_overwrite$rating)      # expect "Bad", "Ok", "Good"
```

```r
# □ Checks (after)
table(movies_overwrite$rating)       # counts in each level
```

□ Comment: Overwriting keeps the column name but replaces its contents and type (numeric → ordered factor).

□ Metacognition: case_when() reads cleaner than nested ifelse() for multi-branch recodes and makes me specify a default. In the future, I could keep the original numeric ratings (e.g., rating_num) and add a new labeled column instead of overwriting, to preserve provenance.

```r
# □ Overwrite numeric rating with an ordered factor via case_when()
movies_rename <- smaller_movies %>%
  mutate(
    rating_words = factor(
      case_when(
        rating >= 4 ~ "Good",
        rating < 2 ~ "Bad",
        TRUE       ~ "Ok"
      ),
      levels = c("Bad", "Ok", "Good"),
      ordered = TRUE
    )
  )
```

6.2.2 Bonus Example

 Goal: Create a new column rating_label in smaller_movies using case_when() that labels ratings >= 4 as "Good" and the rest as "Bad". Keep the original numeric rating column.

 Prediction: After running, the data will still have 100 rows, keep both title and rating, and add rating_label (character or factor). No NAs should appear.

```
# Checks (before)
names(smaller_movies)              # expect "title", "rating"
class(smaller_movies$rating)       # expect "numeric"
```

```
# Add a NEW column using case_when()
smaller_movies %>%
  mutate(
    rating_label = case_when(
      rating >= 4 ~ "Good",
      TRUE        ~ "Bad"
    )
  ) -> movies_newcol
```

 NOTICE: What is happening here -> movies_newcol?

Break Things! How can you reorganize this script?

```
# Checks (after)
names(movies_newcol)               # expect "title", "rating", "rating_label"
class(movies_newcol$rating)        # expect "numeric"
class(movies_newcol$rating_label)  # expect "character" or "factor"
```

 The code used -> movies_newcol, which assigns right-to-left.

 Break Things! What happens with head(movies_newcol) -> movies_newcol?

 Metacognition: I'll stick with the usual <- to make my code easier to read

6.3 Calculations

 Using mtcars, let's create a column l_per_100km that converts from US units (miles per gallon) to metric units (liters per 100km). 235.215 is the conversion factor between these units

 Goal: Identify the most fuel-efficient car in L/100 km terms.

 Prediction: The new column l_per_100km will be added to mtcars_units, and the most fuel-efficient car will have the lowest value in this column.

```
# Code
mtcars_units <- mtcars %>%
  mutate(l_per_100km = round(235.215 / mpg, 1))
```

```
# □ Check
head(mtcars_units)
```

□ Check□ in: Which car is most fuel□ efficient in l/100 km terms?

```
# Find the most fuel-efficient (smallest L/100km)
best_idx <- which.min(mtcars_units$l_per_100km)
best_car <- rownames(mtcars_units)[best_idx]
best_value <- mtcars_units$l_per_100km[best_idx]
```

```
"best car"
best_car
```

```
"best value"
best_value
```

□ Metacognition: Unit conversions are easy to keep alongside original units with mutate() (don't overwrite).

7. ⛶ Practice Space

□ Practice: Recode the mpg column. Use mutate() and case_when()

Fill in the Blanks.

□ Goal: Create mpg_class with three labels using case_when(): “High MPG” if mpg > 25; “Low MPG” if mpg < 15; otherwise “Medium MPG”.

```
# □ Code
mtcars %>%
  ______(mpg_class = _____(
    ___ > 25 ~ "High MPG",
    mpg < 15 ~ "________",
    TRUE     ~ "Medium MPG"
  )) -> mpg_recode
```

```
# □ Checks
table(mpg_recode$____)  # expect 3 levels
```

8. ☒ Assignment

Replace each ____ placeholder with working code or a short written answer. Run each section; be sure the requested objects appear in the Environment. When finished, submit lesson notes with assignment in one script (.Rmd / .qmd) along with the .RData workspace.

Template (use around every code chunk):

□ /□ Goal. One sentence stating the outcome.

□ Prediction. (Optional) What you expect to see before you run code.

```
#□ Code attempts.

# your code here
```

□ Checks (before/after). Show what you're verifying (names, head(), table(), class(), etc.).

□ /□ What I learned / Next step. 1–2 sentences.

8.1 Task 1

□ Library it up!

Make sure there is script in your document that loads dplyr and dslabs packages so their functions / datasets load.

8.2 Task 2

□ Recode ranges (overwrite)

Using quakes, recode mag into THREE ordered levels of *your choice* (e.g., "Low", "Mid", "High") **overwriting** the original column. Use mutate() and case_when() chain and pipe (%>%). Store as **quake_recode**.

```
data("quakes")
?quakes
```

```
____ <- _____
```

```
# Quick check
table(quake_recode$mag)   # should show exactly 3 levels
```

8.3 Task 3

Recode categories (new col)

□ With stars, turn the one-letter *type* into a descriptive phrase of *your own* in a **new** column called full_type. Keep the original type. Store as **stars_recode**.

```
data("stars")
?dslabs::stars
```

```
____ <- _____
```

```
# Quick check
head(select(stars_recode, type, full_type))
```

8.4 Task 4

Unit conversion

Create international_friendly_heights by adding a height_cm column to heights (inches × 2.54). Leave original height.

```
data("heights")
?heights
```

```
____ <- _____
```

□ Reflect: Why add—not overwrite—the height column?

□ EXPLANATION: "___"

9. Save and Upload

1. You will be submitting **both** the R Markdown | Quarto Document and the workspace file. The workspace file saves all the objects in your environment that you created in this lesson. You can save the workspace by running the following command in a code chunk of the R Markdown | Quarto Document document:

```
save.image("Assignment9_Workspace.RData")
```

Or you can click the "Save Workspace" button in the Environment pane.

 Always save the R documents before closing.

2. Find the assignment in this week's module in Canvas and upload **both** the RMD and the workspace file.

10. Today you practiced:

- Practiced logical tests and the structure of case_when().
- Created binary and multi level recodes.
- Used mutate() to overwrite or create columns.
- Performed inline calculations for unit conversion.

 Continue experimenting. Recoding and mutation are the building blocks of clean, analysis ready data!