Use case_when() in dplyr

Authored by

Mohammed looti

November 4, 2025

The **case_when()** function stands out as a powerful utility within the **dplyr** package, a core component of the **R** Tidyverse. This function offers a dramatically improved, elegant, and concise method for performing conditional assignments and generating new variables based on a multitude of logical criteria. Traditional programming often relies on cumbersome nested `if-else` structures, which rapidly degrade code readability and maintainability as complexity increases. By contrast, `case_when()` provides a clean syntax that evaluates conditions sequentially, making complex **data manipulation** tasks significantly more manageable and efficient.

Mastering the Syntax and Logic of case_when()

The operational mechanism of `case_when()` is built upon evaluating pairs of conditions and their corresponding outcomes. Its primary purpose is typically realized when utilized inside the **mutate()** function, which is responsible for adding or modifying columns within a **data frame**. Understanding the fundamental structure is key to leveraging this function effectively for data classification and transformation.

The essential syntax employs a left-hand side (LHS) that specifies a **logical condition**, followed by the tilde operator (~), and finally, the right-hand side (RHS) which dictates the value to assign if the condition is met. A critical feature of `case_when()` is its sequential evaluation: the function processes the conditions in the order they are listed, immediately stopping and returning the result corresponding to the very first condition that evaluates to **TRUE**. This sequential logic necessitates careful ordering of conditions, a principle we will explore further in the examples.

To illustrate this fundamental structure, consider how one might classify a numerical variable into broad categories using sequential logic:

library(dplyr)

df %>%
mutate(new_var = case_when(var1 < 15 ~ 'low',
var2 < 25 ~ 'med',
TRUE ~ 'high'))

It is paramount to include the final condition: `TRUE ~ 'default_value'`. This statement serves as the necessary catch-all, functioning as the default or "else" clause. If every preceding condition fails to evaluate to **TRUE** for a given row, the value associated with the final **TRUE** condition is assigned. This practice is essential for robust code, ensuring that every observation in the **data frame** receives a classification, thereby preventing unwanted implicit assignments of **NA** (Not Available).

Establishing a Practical Data Frame for Demonstration

To concretely demonstrate the versatility and application of the `case_when()` function, we will utilize a small, fictional sample dataset. This **data frame** simulates athlete statistics, including columns for player identification, position, points scored, and assists made. Importantly, this dataset is intentionally structured to contain **NA** (Not Available) values, which represent **missing data**. Addressing these missing observations explicitly is a critical skill in real-world data analysis, and we will use later examples to demonstrate the correct methodology.

#create data frame
df <- data.frame(player = c('AJ', 'Bob', 'Chad', 'Dan', 'Eric', 'Frank'),
position = c('G', 'F', 'F', 'G', 'C', NA),
points = c(12, 15, 19, 22, 32, NA),
assists = c(5, 7, 7, 12, 11, NA))

#view data frame
df

player position points assists
1 AJ G 12 5
2 Bob F 15 7
3 Chad F 19 7
4 Dan G 22 12
5 Eric C 32 11
6 Frank NA NA NA

This structure allows us to proceed with several examples, moving from simple classification based on a single metric to complex logic involving multiple columns and the explicit handling of data integrity issues.

Example 1: Conditional Assignment Based on a Single Variable

Our initial objective is to classify each player's performance based exclusively on their **points** score. We aim to generate a new variable named `quality`, assigning labels of "high," "med," or "low" based on predefined scoring thresholds. This example highlights the fundamental principle that condition order is paramount when using `case_when()`.

Since the function evaluates conditions sequentially and stops at the first **TRUE** match, we must structure the logic from the most restrictive criteria to the least restrictive. If we were to place the broadest condition first (e.g., points > 10), it would incorrectly capture all observations that meet

that minimum threshold, preventing subsequent, more specific conditions (e.g., points > 20) from ever being evaluated. Therefore, we start with the highest threshold (points > 20) and work our way down:

```
df %>%
mutate(quality = case_when(points > 20 ~ 'high',
points > 15 ~ 'med',
TRUE ~ 'low' ))
```

player position points assists quality
1 AJ G 12 5 low
2 Bob F 15 7 low
3 Chad F 19 7 med
4 Dan G 22 12 high
5 Eric C 32 11 high
6 Frank NA NA NA low

The resulting classification demonstrates the strict sequential processing employed by the **case_when()** function. For instance, Player Chad has 19 points; he does not satisfy the first condition (points > 20), but he immediately satisfies the second condition (points > 15), resulting in the classification "med." If the `points` score is 15 or below, or if the score is **NA**, the observation falls through to the final, universal `TRUE ~ 'low'` condition, ensuring a classification is assigned.

Example 2: Applying Complex Logic Across Multiple Variables

One of the primary advantages of `case_when()` is its seamless integration with standard R logical operators, such as `&` (AND) and `|` (OR). This capability allows analysts to define sophisticated criteria by combining conditions from several columns within a single statement, moving beyond simple single-variable thresholds. This flexibility is essential when defining performance tiers that rely on multiple metrics, such as requiring high scores in both **points** and **assists**.

In this example, we redefine player **quality** based on a combined assessment of both statistics, demanding higher overall performance for the top tiers. Notice how the use of the `&` operator forces both conditions within a single line to be simultaneously **TRUE** for that classification to apply:

```
df %>%
mutate(quality = case_when(points > 15 & assists > 10 ~ 'great',
points > 15 & assists > 5 ~ 'good',
TRUE ~ 'average' ))
```

player position points assists quality

1 AJ G 12 5 average
2 Bob F 15 7 average
3 Chad F 19 7 good
4 Dan G 22 12 great
5 Eric C 32 11 great
6 Frank NA NA NA average

By reviewing the output, we can observe the impact of the combined logic. Player Chad (19 points, 7 assists) meets the requirements of the second condition (points > 15 AND assists > 5) and is correctly labeled "good." Conversely, Player Dan (22 points, 12 assists) satisfies the first, stricter condition (points > 15 AND assists > 10), thereby earning the "great" classification. This nested conditional complexity is handled cleanly and efficiently by `case_when()`, maintaining excellent code clarity within the **dplyr** pipeline.

Explicitly Handling Missing Data with is.na()

When working with real-world **data frames**, the presence of **missing values**, represented by **NA**, can significantly complicate conditional logic. If a condition involving an **NA** value is evaluated, the result of that condition itself is typically **NA**, meaning it neither evaluates to **TRUE** nor **FALSE**. By default, any row for which all preceding conditions result in **NA** will fall through to the final `TRUE ~ 'default_value'` statement.

While assigning the default value to missing data might sometimes be acceptable, it often masks genuine data quality issues. It is highly recommended to handle missingness explicitly to ensure transparency and accuracy in analysis. We achieve this by using the **is.na()** function as the very first condition in the `case_when()` sequence. This ensures that all missing observations are identified and categorized appropriately before any numerical or logical thresholds are evaluated.

df %>%
mutate(quality = case_when(is.na(points) ~ 'missing',
points > 15 & assists > 10 ~ 'great',
points > 15 & assists > 5 ~ 'good',
TRUE ~ 'average'))

player position points assists quality
1 AJ G 12 5 average
2 Bob F 15 7 average
3 Chad F 19 7 good
4 Dan G 22 12 great
5 Eric C 32 11 great

6 Frank NA NA NA missing

As demonstrated by the output, Player Frank, who possesses **NA** values for both points and assists, is now accurately and distinctly labeled "missing." In the previous examples, Frank was incorrectly categorized as "low" or "average" by the default catch-all `TRUE` condition. By prioritizing the **is.na()** check, we ensure that data analysts are immediately aware of observations where critical information is absent.

Summary of Best Practices for Robust Conditional Logic

The **case_when()** function is an indispensable, modern tool in the **dplyr** toolkit for performing conditional assignments within the **Tidyverse** framework. It offers superior performance, and crucially, significantly enhanced readability compared to nested `ifelse()` statements, particularly when managing numerous, complex conditions. Adhering to specific guidelines ensures the resulting code is not only clean but also logically sound and robust against unexpected data patterns.

To guarantee optimal and predictable results when implementing `case_when()` logic, data analysts should consistently follow these three core guidelines:

Order Matters: Due to the sequential evaluation process, always list the most specific, restrictive conditions before the more general or encompassing conditions. Evaluation stops the moment the first condition evaluates to **TRUE**.
Handle Defaults: Always conclude your `case_when()` statement with `TRUE ~ 'default_value'`. This universal condition acts as the final safety net, guaranteeing that all rows are assigned a value and preventing implicit **NA** assignment for any uncaught data points.
Address Missingness: If data quality or missingness is a concern, use the **is.na()** function explicitly as the starting condition. This segregates observations with missing data from valid observations, allowing you to categorize them distinctly before applying numerical thresholds.

Additional Resources for Tidyverse Mastery

For those seeking to deepen their understanding of data manipulation in **R** and the broader Tidyverse ecosystem, the following official resources provide authoritative documentation and comprehensive learning materials:

The official documentation for the **case_when()** function, offering detailed technical specifications and advanced examples.
Comprehensive resources for the **dplyr** package provided directly by the Tidyverse team, covering functions like `mutate()`, `filter()`, and `summarise()`.