How to Easily Create New Columns with case_when() in dplyr

Authored by

stats writer

December 4, 2025

The **case_when()** function represents one of the most powerful and versatile tools provided by the **dplyr** package within the **R** programming environment. It is specifically designed to handle conditional logic, allowing data analysts to efficiently create new variables based on a series of complex criteria applied to existing columns. Unlike traditional nested `if-else` structures, **case_when()** offers a cleaner, more readable, and highly vectorized approach to recoding data, making it the preferred method for complex conditional assignments in the Tidyverse ecosystem. This function evaluates conditions sequentially, assigning the corresponding value as soon as the first condition evaluates to **TRUE**, thus ensuring precise control over data transformation workflows.

The primary objective of using **case_when()** is data transformation and categorization. Imagine needing to classify numerical scores into descriptive categories like "High," "Medium," and "Low," or combining multiple binary flags into a single status indicator. This function excels in these scenarios, processing thousands of observations simultaneously while maintaining code clarity. The foundational concept relies on pairing a logical condition with the desired output value, following the structure: `condition ~ value`. By stringing together multiple condition-value pairs, users can define sophisticated categorization schemes tailored to the specific needs of their analysis. This deep dive will explore the syntax, practical applications, and best practices necessary to master conditional data assignment using this essential `dplyr` tool.

Deconstructing the `case_when()` Syntax and Mechanics

To effectively employ the **case_when()** function, it must be integrated into a data manipulation pipeline, typically utilized alongside the **mutate()** function. The **mutate()** function is responsible for adding new columns or modifying existing ones in a data frame, providing the necessary context for **case_when()** to operate. The overall structure involves piping the data frame (`df`) into **mutate()**, where a new variable name is defined and assigned the output of the **case_when()** call. This ensures that the conditional logic is applied element-wise across the rows of the specified data frame, resulting in the desired categorical output.

The syntax within **case_when()** is based on a series of two-sided formulas separated by commas. Each formula takes the form of `LHS ~ RHS`, where the Left Hand Side (LHS) is a logical expression (e.g., `var1 > 10` or `var2 == 'A'`), and the Right Hand Side (RHS) is the value assigned to the new column for observations where the LHS is **TRUE**. It is crucial to remember that evaluation is done sequentially from the first condition defined to the last. Once an observation meets a condition, the corresponding value is assigned, and subsequent conditions are ignored for that specific observation. This sequential nature mandates careful ordering of conditions, especially when dealing with overlapping ranges (e.g., defining "High" before "Medium").

A mandatory component for robust implementation is the inclusion of a final, catch-all condition.

This is achieved by setting the last LHS to **TRUE**, which acts as the equivalent of an "else" statement in procedural programming. If an observation fails to meet any of the preceding explicit conditions, the value defined by the **TRUE** condition is assigned. Neglecting this step can lead to unassigned values being filled with missing data (`NA`), which is often undesirable unless explicitly intended. The following foundational code block illustrates the typical setup required to define and execute complex conditional assignment using **case_when()**:

library(dplyr)

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

The core takeaway from this syntax is the elegance of the structure: the data is piped (`%>%`) into **mutate()**, a new column is defined (`new_var`), and the logic is concisely defined within **case_when()**, with the final **TRUE** statement ensuring completeness across the entire dataset.

Preparing the Sample Data Frame for Analysis

To demonstrate the functionality of **case_when()** in a realistic context, we will utilize a small data frame containing mock performance statistics for several players. This dataset includes various types of variables--character, numeric, and missing values--which allows us to showcase how the function handles different data types and conditions. Understanding the structure of this input data is essential for interpreting the subsequent transformation examples correctly.

The sample data frame, named `df`, contains four relevant columns: `player` (character identifier), `position` (categorical character data, including one **NA** value), `points` (numeric score), and `assists` (numeric score, also including **NA** values). The inclusion of missing values (represented by **NA**) is intentional, as it highlights a critical consideration when writing conditional logic in R: how to handle observations where the condition cannot be evaluated due to missing data. We will use the numeric columns, `points` and `assists`, as the basis for our conditional recoding exercises.

Before proceeding with transformations, it is vital to inspect the raw data frame structure. This step confirms data integrity and ensures that the conditional ranges defined in **case_when()** align logically with the values present in the columns. For instance, `points` range from 12 to 32, while `assists` range from 5 to 12. These ranges will be used to define the boundaries for our new categorical quality variable. The following code block shows the creation and structure of our sample dataset:

#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

Example 1: Creating a Variable Based on a Single Column

In the first example, we focus on generating a new categorical variable, `quality`, based solely on the values found in the `points` column. This is a common requirement in data analysis where continuous numerical data needs to be discretized into distinct bins or levels. We aim to categorize players into "high," "med" (medium), or "low" quality tiers based on their scoring performance. This task clearly demonstrates the fundamental application of **case_when()** for simple, single-variable recoding.

We establish three sequential conditions within **case_when()**. The critical aspect here is the ordering: we define the most restrictive or highest category first. Specifically, we check if `points` is greater than 20, assigning "high" if true. Then, we check if `points` is greater than 15, assigning "med." Because the conditions are evaluated sequentially, any player with points greater than 20 would have already been assigned "high" by the first condition; therefore, the second condition only evaluates players whose points are between 16 and 20 (inclusive). Finally, the catch-all `TRUE` condition assigns "low" to all remaining observations--this includes players scoring 15 or less, as well as the observation with **NA** in the `points` column.

This sequential execution is key to achieving accurate results when defining ranges. If the conditions were incorrectly ordered (e.g., checking for `points > 15` before `points > 20`), observations with very high scores would erroneously be categorized as "med" because the first satisfied condition (points > 15) would halt further evaluation. The following code implements this logic and displays the resulting data frame with the new `quality` column:

```
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

A detailed breakdown of how the **case_when()** function determined the value for each row in the new `quality` column confirms the sequential evaluation process and the role of the final **TRUE** condition:

If the value in the **points** column is greater than 20 (as for Dan and Eric), then the value in the **quality** column is "high."
Else, if the value in the **points** column is greater than 15 but not greater than 20 (as for Chad), then the value in the **quality** column is "med."
Else, if all previous conditions fail (including cases where **points** is 15 or less, or is a missing value like **NA**), then the value in the **quality** column is "low."

Example 2: Implementing Complex Logic with Multiple Variables

The true power of **case_when()** emerges when handling conditional assignments that depend on the interaction between multiple variables. This functionality is achieved by incorporating standard logical operators, such as the AND operator (`&`) and the OR operator (`|`), directly within the LHS condition of the **case_when()** formula. For this example, we aim to define `quality` based on a combination of both `points` and `assists`, requiring a player to excel in both metrics to achieve the highest categorization.

We define three distinct categories: "great," "good," and "average." The "great" category requires a player to score over 15 points AND have more than 10 assists. The "good" category is slightly less demanding, requiring over 15 points AND more than 5 assists. Any player who does not meet either of these specific criteria will be assigned the default "average" quality via the final **TRUE** statement. This structure demonstrates how easily compound conditions can be written and managed within the single, unified structure of **case_when()**, providing far greater clarity than deeply nested `if` statements.

Notice how players Dan (22 points, 12 assists) and Eric (32 points, 11 assists) satisfy the first, most restrictive condition (`points > 15 & assists > 10`) and are correctly labeled "great." Player Chad (19 points, 7 assists) fails the first condition but satisfies the second (`points > 15 & assists > 5`) and is therefore labeled "good." The remaining players (AJ, Bob, and Frank) do not meet either criteria and default to "average." This multi-variable approach is indispensable for creating complex segmentation variables essential for advanced analytical tasks.

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

Handling Missing Values Explicitly using `is.na()`

A common pitfall in data manipulation within R is the implicit handling of **NA** (missing) values. When a logical condition involves an **NA** value (e.g., `points > 15` where `points` is **NA**), the result of that condition is almost always **NA**, not **TRUE** or **FALSE**. In the context of **case_when()**, if an observation results in **NA** for all specified conditions, it will only be caught by the final `TRUE` condition, as seen in the previous examples where Frank's missing data defaulted to "low" or "average."

However, analysts often need to explicitly categorize missing data, perhaps labeling it "missing" or "unknown" instead of grouping it with the default category. To achieve this necessary differentiation, we must use the **is.na()** function as the first condition within **case_when()**. By placing `is.na(points)` or a combined check like `is.na(points) | is.na(assists)` at the very beginning of the conditional sequence, we intercept all missing observations before they reach the main logic flow.

This best practice ensures that missing values are handled predictably and separately from valid data points. In the revised code below, we prioritize the check for missing data in the `points` column. If `points` is **NA**, the player Frank is immediately assigned the category "missing." Only after this explicit missing value check do we proceed to evaluate the performance-based

conditions, ensuring clarity and accuracy across the entire dataset, regardless of data completeness.

```
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

Best Practices and Advanced Considerations

When writing complex conditional statements using **case_when()**, adopting certain best practices can significantly enhance code maintainability and prevent subtle logical errors. Foremost among these is the principle of ordering: always place the most specific, restrictive, or critical conditions at the beginning of the function call. Since the evaluation stops at the first true condition, poorly ordered conditions--such as placing a general range before a specific sub-range--will result in the specific sub-range never being reached, leading to incorrect categorization for those observations.

Secondly, ensure that the data types for the assigned values (the RHS of the formulas) are consistent across all conditions. For instance, if the first condition assigns a character string ("high"), all subsequent conditions, including the final `TRUE` condition, must also assign character strings. Mixing output types (e.g., assigning a character string in one condition and a numeric value in another) will result in R coercing all outputs to the most flexible type, which can lead to unexpected type conversions (like turning all numbers into strings). Always confirm that the output column maintains a predictable data type.

Finally, never underestimate the necessity of the final **TRUE** condition. This step guarantees that every single row in the data frame receives an assignment, preventing the creation of unwanted intermediate **NA** values that can complicate later analysis. Even if you believe all possible scenarios are covered by your explicit conditions, including the default assignment (e.g., `TRUE ~ 'Other'`) acts as a robust safeguard against unexpected data patterns or errors in logic. Adhering to these principles ensures that your data transformations are efficient, accurate, and easily

understandable by other members of an analysis team.