How to Write a Nested If Else Statement in R (With Examples)

Authored by

stats writer

December 11, 2025

Introduction to Conditional Logic in R

Conditional statements are fundamental building blocks in any R programming task, allowing the execution of different code blocks based on whether specific criteria are met. In R, handling element-wise conditional evaluation is often managed efficiently using the `ifelse` function, which is part of **base R**. Unlike standard `if...else` control flow structures, `ifelse` is fully vectorized, meaning it can process entire vectors or columns of a data frame simultaneously, resulting in significantly improved performance for data manipulation tasks.

The primary purpose of the `ifelse` function is to apply a rule to every element within a vector and return a corresponding output vector based on the test results. This capability is essential when conducting data transformations, feature engineering, or categorization based on numerical or categorical thresholds. Mastering this function, including its nested forms, is a cornerstone skill for efficient data analysis in R.

Syntax and Structure of the R `ifelse` Function

The `ifelse` function utilizes a concise and intuitive structure designed for maximum readability and speed when performing binary conditional checks. It requires three distinct arguments, providing a clear map for defining the logical test and the resulting outputs for both true and false outcomes. The simplicity of its signature makes it highly suitable for incorporation directly into column generation steps within a data frame.

The function adheres to the following strict syntax:

ifelse(test, yes, no)

where the arguments are defined as follows:

test: This argument requires a logical expression that evaluates to either `TRUE` or `FALSE` for each element in the input vector. Common examples include comparisons like `x > 10` or `y == 'category'`.
yes: This is the value or vector of values returned whenever the corresponding element in the **test** evaluates to `TRUE`. This result dictates the outcome for successful conditions.
no: This is the value or vector of values returned when the corresponding element in the **test** evaluates to `FALSE`. This argument becomes the crucial insertion point for further nested conditions when handling multi-criteria decisions.

This tutorial provides practical explanations and examples on how to effectively utilize the `ifelse` function for both simple binary decisions and complex multi-level nested conditional logic in R. We will use a consistent sample data frame throughout to illustrate these concepts in a practical

context.

Setting Up the Sample Data Frame

Before diving into the conditional statements, we must establish a realistic dataset upon which to perform our transformations. The following code snippet creates a sample R **data frame** named `df`. This dataset represents performance statistics for different teams, containing columns for team identifier, points scored, and rebounds achieved.

Understanding the structure of this input data is vital, as all subsequent `ifelse` operations will rely on these existing columns to generate new classifications or ratings. Note the varying levels of points and rebounds across the teams 'A', 'B', 'C', and 'D'.

#create data frame
df <- data.frame(team = c('A', 'A', 'B', 'B', 'B', 'C', 'D'),
points = c(4, 7, 8, 8, 8, 9, 12),
rebounds = c(3, 3, 4, 4, 6, 7, 7))

#view data frame
df

team points rebounds
1 A 4 3
2 A 7 3
3 B 8 4
4 B 8 4
5 B 8 6
6 C 9 7
7 D 12 7

This `df` provides the necessary context to demonstrate how the `ifelse` function vectorizes calculations, applying conditional logic row-by-row based on the values present in the `team` column.

Example 1: Implementing a Basic Binary If Else Statement

The most straightforward application of `ifelse` involves a simple binary decision: classifying rows based on a single condition. In this first example, we aim to categorize teams as 'great' if they are Team 'A' and 'bad' otherwise. This operation requires defining the comparison test (`df$team == 'A'`), the value for true outcomes ('great'), and the value for false outcomes ('bad'). The result is stored in a new column called `rating`.

The benefit of using `ifelse` here is its ability to perform this evaluation across all seven rows simultaneously without needing an explicit loop. The function returns a vector of seven elements which is then assigned directly to the new column in the data frame. This approach highlights the efficiency gained through R's vectorization capabilities.

#create new column in data frame
df$rating <- ifelse(df$team == 'A', 'great', 'bad')

#view data frame
df

team points rebounds rating
1 A 4 3 great
2 A 7 3 great
3 B 8 4 bad
4 B 8 4 bad
5 B 8 6 bad
6 C 9 7 bad
7 D 12 7 bad

Reviewing the resultant output confirms the successful execution of the conditional logic:

If the value in the `team` column is 'A', then the `rating` is assigned 'great.'
Otherwise (Else condition), the `rating` is assigned 'bad.'

This implementation demonstrates the simplest form of conditional assignment, providing a foundation for understanding how the function handles subsequent, more complex layering of conditions.

Example 2: Constructing a Nested If Else Statement (Two Levels)

When more than two outcomes are required, we transition from a simple binary `ifelse` to a **nested `ifelse`** structure. Nesting is achieved by placing a second `ifelse` function call within the 'no' argument of the preceding function. This effectively creates a sequence of conditional checks, where the flow only moves to the next check if all prior conditions have evaluated to `FALSE`.

In this example, we expand our categorization to include a third outcome: 'OK'. We first check if `df$team == 'A'`. If true, the output is 'great'. If false, R proceeds to the nested `ifelse` statement, which checks if `df$team == 'B'`. If this second test is true, the rating is 'OK'; otherwise, if both 'A' and 'B' are false, the final fallback rating is 'bad'.

While this structure is powerful, careful attention must be paid to parentheses placement and ensuring that the logical flow correctly handles sequential evaluation. For clarity, it is often helpful to format the code across multiple lines, aligning the nested functions as shown below.

```
#create new column in data frame
df$rating <- ifelse(df$team == 'A', 'great',
ifelse(df$team == 'B', 'OK', 'bad'))

#view data frame
df
```

team points rebounds rating
1 A 4 3 great
2 A 7 3 great
3 B 8 4 OK
4 B 8 4 OK
5 B 8 6 OK
6 C 9 7 bad
7 D 12 7 bad

The resulting structure clearly defines the hierarchical conditions:

If `team` is 'A', assign 'great'.
Else (move to inner `ifelse`): If `team` is 'B', assign 'OK'.
Else (final fallback), assign 'bad'.

As observed in the output, teams B are now correctly categorized as 'OK', while teams C and D fall into the final default 'bad' category.

Example 3: Writing Longer, Multi-Level Nested If Else Statements

The nesting pattern introduced in Example 2 can be extended indefinitely to accommodate any number of required conditions. This technique is crucial when categorizing data based on multiple mutually exclusive criteria, especially when using a fixed set of categories like those in our `team` column.

In this final example using the `df`, we introduce a third level of nesting to specifically account for Team 'C'. The structure now involves three distinct checks before defaulting to the final 'bad' rating. This demonstrates the full capacity of repeated `ifelse` calls to handle complex, ordered conditional logic within a single vector assignment.

```
#create new column in data frame
df$rating <- ifelse(df$team == 'A', 'great',
ifelse(df$team == 'B', 'OK',
ifelse(df$team == 'C', 'decent', 'bad')))

#view data frame
df
```

team points rebounds rating
1 A 4 3 great
2 A 7 3 great
3 B 8 4 OK
4 B 8 4 OK
5 B 8 6 OK
6 C 9 7 decent
7 D 12 7 bad

This extended nested `ifelse` structure successfully implements the multi-tiered categorization:

If the value in the team column is 'A' then give the player a rating of 'great.'
Else, if the value in the team column is 'B' then give the player a rating of 'OK.'
Else, if the value in the team column is 'C' then give the player a rating of 'decent.'
Else, give the player a rating of 'bad.'

Note that you can use this exact pattern to write nested `ifelse` statements as long as you require additional hierarchical distinctions. However, caution is advised, as complexity increases rapidly with each nested layer.

Alternatives and Best Practices for Complex Conditional Logic

While nested `ifelse` statements are powerful for vectorizing calculations, they quickly become unwieldy and difficult to debug when multiple layers are involved. For very complex or lengthy sequences of conditional checks, R developers often employ alternative functions that enhance readability and robustness.

One popular alternative is the `case_when()` function provided by the tidyverse package, specifically `dplyr`. `case_when()` allows users to define many pairs of conditions and corresponding outcomes in a much flatter, cleaner structure, eliminating the need for deeply nested parentheses. This is generally the recommended approach when dealing with three or more mutually exclusive conditions that apply to a data frame column.

Furthermore, for scenarios where conditional assignment is based on numerical ranges (e.g., points or rebounds), the `cut()` function in R provides a specialized, highly efficient mechanism for binning continuous variables into discrete categorical factors. For example, if we were rating performance based on 'points' rather than 'team' name, `cut()` would often be superior to a lengthy series of `ifelse` statements checking numerical thresholds.

Regardless of the method chosen, consistency and clear code documentation are essential. When utilizing nested conditional logic with `ifelse`, ensure proper indentation and line breaks, as shown in the examples above, to maintain clarity regarding which 'yes' and 'no' arguments belong to which logical test.

Conclusion: Leveraging Vectorization with Nested If Else

The `ifelse` function remains a core component of R for applying conditional logic across vectors and columns. Its primary advantage lies in its inherent vectorization, offering significant performance benefits over traditional iterative loops or standard scalar `if...else` control structures when working with large datasets.

By mastering the nesting technique, as demonstrated through the three examples, developers can construct sophisticated assignment rules within a single, vectorized command. While deeply nested structures should be approached with caution due to potential readability issues, they offer a powerful, built-in solution for multi-criteria conditional assignment without requiring external packages. This knowledge is indispensable for any user performing data preparation and feature creation in R.

You can find more R tutorials and advanced data transformation techniques in authoritative online resources.