Learning R: Using IF Statements with Multiple Conditions

Authored by

Mohammed looti

October 30, 2025

Mastering Conditional Logic for Data Transformation in R

Effective data manipulation is fundamental to success in R programming. A frequent requirement in data analysis involves deriving new features or columns based on complex rules applied to existing data. This process relies heavily on conditional statements, which govern the execution flow, allowing different outcomes based on whether specific criteria are met. When these criteria involve simultaneous or alternative requirements, mastering the application of if statements with multiple conditions becomes indispensable for robust scripting.

This comprehensive guide is designed to clarify the construction of new variables within an R data frame using advanced conditional logic. We will specifically concentrate on the highly efficient `ifelse()` function, which is optimally engineered for vectorized operations, ensuring high performance when processing entire columns. By combining logical operators--specifically the inclusive OR and the restrictive AND--you gain the capability to define precise and intricate rules for categorization and transformation.

Throughout this article, we will examine practical, step-by-step examples demonstrating how to execute these conditional assignments reliably. Developing proficiency in these techniques will significantly boost your efficiency and ability to handle sophisticated data transformations, moving beyond simple binary checks to truly nuanced data categorization.

The Power and Syntax of the `ifelse()` Function

The `ifelse()` function stands out in R for its versatility and efficiency in performing conditional operations across entire vectors simultaneously. Unlike traditional control structures like the standard `if-else` block, which is designed to handle single-value evaluations iteratively, `ifelse()` excels at vectorized processing. This makes it the ideal tool for generating new columns in a data frame by evaluating existing column values against a specified logical test.

The foundational syntax for utilizing `ifelse()` is straightforward: `ifelse(test, yes, no)`. Understanding the role of each argument is key to leveraging its full potential:

`test`: This required argument is a logical vector, meaning it must resolve to a sequence of `TRUE` or `FALSE` values for every element. It is typically constructed by comparing one or more columns within your data frame using relational operators (e.g., **>**, **<=**, **==**).

`yes`: This specifies the value or vector that the function returns for corresponding elements where the `test` argument evaluates to `TRUE`. This is the positive assignment outcome.

`no`: This specifies the value or vector returned for corresponding elements where the `test` argument evaluates to `FALSE`. This is the negative assignment outcome.

When working with multiple conditions, the true power of `ifelse()` is revealed. By seamlessly integrating R's logical operators within the `test` argument, you can combine numerous logical expressions. This capability enables sophisticated conditional assignments that remain remarkably concise and highly readable, facilitating efficient data transformation workflows.

Implementing Inclusive Logic with the OR Operator (`|`)

The need for flexible categorization often arises where a condition must be satisfied if **any** of several criteria are met. For this purpose, the OR operator is employed. In R, this logical disjunction is represented by the pipe symbol (`|`). This operator yields a result of `TRUE` if at least one of the conditions it connects is `TRUE`. It is exceptionally valuable when you aim to categorize data broadly, recognizing that multiple distinct factors can lead to the same desired outcome or label.

When structuring a new column assignment using `ifelse()` in conjunction with the OR operator, the general formula adheres to the following pattern:

df$new_var <- ifelse(df$var1>15 | df$var2>8, "value1", "value2")

In this structure, we are creating or modifying the column `df$new_var`. The logical test, `df$var1 > 15 | df$var2 > 8`, dictates that if the value in `df$var1` exceeds 15, **OR** the value in `df$var2` exceeds 8, the corresponding row in `df$new_var` will be assigned `"value1"`. Crucially, only if both conditions simultaneously evaluate to false will the entry be assigned the alternative `"value2"`. This inclusive approach provides the necessary flexibility for broad data inclusion based on satisfying minimal criteria.

Enforcing Strict Criteria with the AND Operator (`&`)

In contrast to the inclusive nature of OR, situations demanding strict filtering require that a condition is fulfilled only if **all** specified criteria are simultaneously true. For this rigorous requirement, we utilize the AND operator. In R, the logical conjunction is symbolized by the ampersand (`&`). This operator returns `TRUE` exclusively when every single condition it connects is also `TRUE`. It is an essential tool for setting highly precise categorization rules where no compromises on the criteria are acceptable.

To define a new column using `ifelse()` coupled with the AND operator, the standard structure is implemented as follows:

df$new_var <- ifelse(df$var1>15 & df$var2>8, "value1", "value2")

Here, the column `df$new_var` is being generated based on the compound logical test: `df$var1 > 15 & df$var2 > 8`. An entry in `df$new_var` will be assigned `"value1"` only if the value in `df$var1` is greater than 15 **AND** the value in `df$var2` is greater than 8 concurrently. If either or both of these conditions are false, the entry defaults to `"value2"`. This method is indispensable for precise classification, guaranteeing that only data points that satisfy the entire set of stringent criteria are isolated and identified.

Preparing and Examining the Example Data Set

To provide a clear, hands-on demonstration of how `ifelse()` works with multiple conditions, we will establish a straightforward yet realistic example data frame. This data set will simulate hypothetical player performance statistics, which we can then use to derive new categorical variables based on performance metrics like scores and assists. This environment allows us to observe the immediate consequences of applying different logical operators.

We will construct a data frame named `df` containing three columns: `team`, `points`, and `assists`. These columns will serve as the inputs for our conditional logic evaluations. The R code provided below executes the initialization of this data set, preparing it for subsequent manipulation:

#create data frame
df <- data.frame(team=c('A', 'A', 'A', 'A', 'B', 'B', 'B', 'B'),
points=c(8, 8, 10, 13, 17, 19, 22, 25),
assists=c(5, 10, 9, 6, 8, 10, 11, 12))

#view data frame
df

team points assists
1 A 8 5
2 A 8 10
3 A 10 9
4 A 13 6
5 B 17 8
6 B 19 10
7 B 22 11
8 B 25 12

Our resulting `df` comprises eight records, each detailing a player's statistics across two distinct teams. The numerical columns, `points` and `assists`, are the variables we will subject to conditional testing. This setup ensures a controlled and transparent demonstration of how R

handles complex Boolean expressions during data assignment.

Case Study 1: Flexible Categorization Using the OR Operator (|)

We now apply the inclusive OR operator to our prepared data frame. Our objective is to generate a new column, `rating`, which assigns a status of "good" or "bad" based on a flexible standard. A player achieves a "good" rating if their `points` total is greater than 15 **OR** if their `assists` count exceeds 8. If neither of these conditions is satisfied, the player is assigned the "bad" rating. This scenario perfectly showcases the utility of the OR operator for achieving broad inclusion based on alternative successful pathways.

The following R script demonstrates the implementation of this logic, utilizing the `ifelse()` function to perform the vectorized assignment:

#create new "rating" column using if statement with multiple conditions
df$rating <- ifelse(df$points>15 | df$assists>8, "good", "bad")

#view updated data frame
df

team points assists rating
1 A 8 5 bad
2 A 8 10 good
3 A 10 9 good
4 A 13 6 bad
5 B 17 8 good
6 B 19 10 good
7 B 22 11 good
8 B 25 12 good

Examination of the updated data frame reveals how the inclusive logic operates. For example, consider the second player: they scored only 8 points (not > 15), but their 10 assists satisfied the second condition (assists > 8), resulting in a "good" rating. Conversely, the fifth player achieved 17 points (satisfying the first condition) but only 8 assists (not > 8), yet still earned a "good" rating. This clearly confirms the inclusive nature of the OR operator: a row is only assigned "bad" if both underlying conditions are simultaneously false.

Case Study 2: Strict Filtering Using the AND Operator (&)

Next, we pivot to demonstrate the stringent requirements enforced by the AND operator. In this scenario, we redefine the `rating` column with a much stricter criterion: a player must be rated

"good" only if their `points` are greater than 15 **AND** their `assists` are also greater than 8. If these two conditions are not met simultaneously in any given row, the player receives the "bad" rating. This powerful demonstration highlights how the AND operator is used to enforce tight, multi-factor criteria.

The following R code applies this more restrictive conditional logic to our data frame:

#create new "rating" column using if statement with multiple conditions
df$rating <- ifelse(df$points>15 & df$assists>8, "good", "bad")

#view updated data frame
df

team points assists rating
1 A 8 5 bad
2 A 8 10 bad
3 A 10 9 bad
4 A 13 6 bad
5 B 17 8 bad
6 B 19 10 good
7 B 22 11 good
8 B 25 12 good

Comparing this output to the previous example, the difference in categorization is immediately apparent. Notice the second player (8 points, 10 assists), who was "good" previously, is now categorized as "bad" because they failed the points requirement (8 is not > 15), thus failing the combined AND test. Similarly, the fifth player (17 points, 8 assists) is also "bad" because their assists (8) are not strictly greater than 8. Only the players who excelled across both metrics--those with high points and high assists (rows 6, 7, and 8)--successfully pass the combined criteria and receive the "good" rating, solidifying the strict filtering capability of the AND operator.

Beyond Binary: Handling Multiple Outcomes with Advanced Conditionals

While the `ifelse()` function is perfectly suited for scenarios involving two possible outcomes (a binary assignment), real-world data analysis frequently demands categorization into three or more possible states. To handle these multi-outcome requirements, R users traditionally rely on two primary methods: nesting `ifelse()` statements or employing the superior clarity of the `dplyr::case_when()` function.

Nested `ifelse()` Statements: This technique involves embedding one `ifelse()` call within the final "no" argument of the preceding function call. For instance, creating categories like "excellent",

"good", and "average" based on different performance tiers would look like this:

df$performance <- ifelse(df$points > 20 & df$assists > 10, "excellent",
ifelse(df$points > 15 | df$assists > 8, "good", "average"))

Although functionally correct, deeply nested `ifelse()` statements introduce significant complexity. As the number of required conditions and resultant outcomes grows, debugging and maintaining this code structure quickly becomes challenging, leading to reduced code readability.

Leveraging `dplyr::case_when()`: A far more readable and maintainable solution for managing multiple, potentially complex, and mutually exclusive conditions is offered by the `dplyr` package's `case_when()` function. This function allows users to list sequential condition-value pairs, which drastically simplifies complex logic. Replicating the "excellent", "good", "average" categorization using `case_when()` demonstrates its efficiency:

 # Ensure dplyr is installed and loaded
 # install.packages("dplyr")
library(dplyr)

df$performance <- case_when(
df$points > 20 & df$assists > 10 ~ "excellent",
df$points > 15 | df$assists > 8 ~ "good",
TRUE ~ "average" # Default case if no other conditions are met
)

The `case_when()` function evaluates these conditions strictly in order, assigning the value associated with the first logical expression that returns `TRUE`. The final line, `TRUE ~ "average"`, serves as a crucial default or "catch-all" assignment, guaranteeing that every row in the data frame receives an appropriate value. This clarity and sequential evaluation make `case_when()` highly recommended for advanced data categorization.

Conclusion: Enhancing Data Analysis Capabilities

The ability to effectively combine if statements with multiple conditions is a cornerstone skill for any user engaging in data manipulation and feature engineering. The `ifelse()` function, when paired with the logical operators OR (`|`) and AND (`&`), provides a powerful and inherently vectorized method for generating new columns based on complex, multi-criteria rules. Whether your analysis requires the broad inclusivity of OR or the rigid precision of AND, these fundamental tools ensure accurate and robust data categorization.

For analysts dealing with classification scenarios that extend beyond simple binary results, we strongly recommend migrating from nested `ifelse()` structures to the `dplyr::case_when()` alternative. This function not only simplifies the expression of intricate logic but also significantly improves code clarity and maintainability, which is paramount in collaborative or large-scale data projects. Mastering these conditional techniques will solidify your foundation for high-quality data science workflows.

Further Resources for R Data Manipulation

To continue expanding your data analysis toolkit in R, explore these related tutorials and techniques:

Placeholder for future links: How to Reshape Data in R

Placeholder for future links: Performing Joins in R

Placeholder for future links: Grouping and Summarizing Data