Learning dplyr: Conditionally Mutating Columns Based on String Content

Authored by

Mohammed looti

October 27, 2025

Conditionally Mutating Variables in R with dplyr

In the realm of advanced data analysis and statistical computing, the ability to selectively transform columns within a data frame is not merely a convenience--it is a fundamental necessity. Often, analysts need to apply specific transformations, such as standardization, normalization, or complex arithmetic operations, only to variables that meet certain naming conventions or structural criteria. The **dplyr** package, which stands as a cornerstone of the tidyverse collection in R, provides an exceptionally powerful and expressive grammar for achieving this precise conditional data manipulation. This approach significantly enhances code readability and reduces the risk of unintended transformations on unrelated variables.

Achieving this level of precision requires harnessing **dplyr**'s specialized functions, particularly those designed for scoped operations. Specifically, the combination of the `mutate_at()` function with powerful selection helpers like `vars()` and `contains()` allows for dynamic targeting of variables based on substrings present within their column names. This mechanism provides a robust solution for scenarios where manual column selection is impractical due to the high volume or variability of the dataset. Mastering this technique is essential for efficient and scalable data wrangling workflows in modern statistical programming.

This comprehensive guide will detail the structure and application of these functions, demonstrating how to efficiently mutate variables only when their column names contain a specified string. We will explore the theoretical underpinnings of the scoped verbs and walk through practical examples that solidify understanding of this highly valuable data transformation strategy.

The Role of Scoped Verbs: Understanding `mutate_at()`

The `mutate_at()` function belongs to a family of "scoped" **dplyr** verbs, which were designed to simplify operations across multiple columns. These scoped verbs--including `mutate_all()`, `mutate_if()`, and `mutate_at()`--offer distinct ways to define the scope of a transformation. While `mutate_all()` applies a function uniformly across every column in the data frame, and `mutate_if()` conditionally applies a function based on a logical test of the column's content (e.g., if it is numeric), `mutate_at()` is solely focused on column selection criteria defined by names or indices.

The fundamental structure of `mutate_at()` requires three key components to execute a successful transformation. First, the function receives the data structure itself, typically passed via the piping operator (`%>%`). Second, it requires a precise definition of the columns to be targeted, which is where `vars()` and the selection helpers come into play. Finally, it needs the function or list of functions that must be applied to the selected subset of variables. This design provides maximum

control over which transformations occur where, ensuring that data integrity is maintained across columns that should remain untouched.

It is the integration of `vars()` that unlocks the potential for dynamic name-based selection. By wrapping various selection functions--such as `starts_with()`, `ends_with()`, or, in our case, `contains()`--inside `vars()`, we instruct `mutate_at()` exactly which columns to operate on. This separation of selection logic from the transformation logic makes the code highly modular and easier to debug, a significant advantage in complex analytical projects in R.

Precision Targeting: Leveraging `vars()` and `contains()`

The core mechanism for conditional mutation based on column names relies heavily on the specific interplay between the `vars()` function and the column selection helper `contains()`. The `vars()` function serves as a container or wrapper for specifying column names or selection criteria, translating human-readable logic into a format that `mutate_at()` can interpret for targeting. Without `vars()`, the selection helpers would not function correctly within the scoped verb framework.

The true power, however, resides in the `contains()` helper. This function allows the user to specify a literal string or pattern, and it efficiently searches through all column names in the provided data frame, returning a set of indices for every column that includes that specified substring. This is crucial because it enables highly flexible pattern matching, making it possible to target columns regardless of their position or the presence of other words, as long as the defining substring is included.

For instance, if a data frame contains columns named `sales_Q1`, `profit_Q1`, and `expense_Q1`, using `contains("Q1")` will successfully select all three columns, allowing for simultaneous manipulation. This method is far superior to manually listing column names, particularly when dealing with wide datasets where the structure is consistent but verbose. This precision targeting is what makes **dplyr** an indispensable tool for data preparation.

Implementing the Core Syntax for Conditional Transformation

Implementing conditional mutation requires a concise yet powerful syntax that leverages the strengths of the **dplyr** ecosystem. The general structure is elegant and follows the standard tidyverse philosophy: feed the data forward, define the action, and specify the target columns. We utilize an anonymous function syntax (starting with `~`) to clearly define the transformation operation that must be applied to each selected column.

The following example demonstrates the fundamental syntax for standardizing variables whose names contain the specific string 'starter'. This is a common requirement in statistical modeling to

ensure variables contribute equally to the analysis.

library(dplyr)

df %>% mutate_at(vars(contains('starter')), ~ (scale(.) %>% as.vector))

Let us meticulously dissect each component of this powerful command to understand its function and purpose:

`df %>%`: This indicates the initiation of the data pipeline. The piping operator (`%>%`) sends the data frame `df` as the primary input to the subsequent function, streamlining the code execution flow.
`mutate_at(...)`: This function is explicitly called upon to apply transformations to a defined subset of the columns, creating new or overwriting existing variables in the process.
`vars(contains('starter'))`: This is the selection engine. The `contains('starter')` function identifies all columns where the column header includes the exact substring 'starter'. This selection is then passed via `vars()` to `mutate_at()`.
`~ (scale(.) %>% as.vector)`: This defines the transformation. The `~` signals an anonymous function, where the `.` represents the current column being processed. Here, the `scale()` function is applied to standardize the column (setting the mean to zero and the standard deviation to one). Crucially, the result is piped into `as.vector()` because `scale()` often returns a specialized matrix object, which must be converted back into a simple numerical vector for seamless integration back into the data frame structure.

Practical Application: Scaling Specific Columns

To fully appreciate the efficiency of this method, let's examine a concrete use case involving sports statistics. Imagine we are analyzing performance data where we differentiate between 'starter' player statistics and 'bench' player statistics. For machine learning or comparative statistical models, we might only need to standardize the 'starter' variables to ensure they are on a comparable scale, leaving the raw 'bench' data for descriptive analysis.

We begin by setting up a representative sample data frame in R. This frame includes a mix of variables, some containing the target string 'starter' and others containing 'bench', simulating a typical messy dataset encountered in real-world data science projects.

#create data frame
df <- data.frame(team=c('A', 'B', 'C', 'D', 'E', 'F'),
starter_points=c(22, 26, 25, 13, 15, 22),
starter_assists=c(4, 5, 10, 14, 12, 10),
bench_points=c(7, 7, 9, 14, 13, 10),

bench_assists=c(2, 5, 5, 4, 9, 14))

#view data frame
df

team starter_points starter_assists bench_points bench_assists
1 A 22 4 7 2
2 B 26 5 7 5
3 C 25 10 9 5
4 D 13 14 14 4
5 E 15 12 13 9
6 F 22 10 10 14

Next, we execute the conditional mutation, applying the `scale()` function only to columns identified by `contains('starter')`. This single line of code handles the complex selection and transformation simultaneously, producing a clean, standardized output for the targeted variables.

library(dplyr)

#apply scale() function to each variable that contains 'starter' in the name
df %>% mutate_at(vars(contains('starter')), ~ (scale(.) %>% as.vector))

team starter_points starter_assists bench_points bench_assists
1 A 0.2819668 -1.3180158 7 2
2 B 1.0338784 -1.0629159 7 5
3 C 0.8459005 0.2125832 9 5
4 D -1.4098342 1.2329825 14 4
5 E -1.0338784 0.7227828 13 9
6 F 0.2819668 0.2125832 10 14

The resulting data frame clearly illustrates the effect of the conditional transformation. The numerical values in the `starter_points` and `starter_assists` columns are now standardized, having a mean of zero and a standard deviation of one. Conversely, the `bench_points` and `bench_assists` columns retain their original, raw values, confirming the precision and selectivity of the column selection using `contains()`.

Expanding Utility: Custom Transformations and Adaptability

The power of this technique is not limited to built-in R functions like `scale()`. The framework provided by `mutate_at()` in conjunction with `vars()` and `contains()` is highly flexible, supporting

the application of virtually any function, whether a complex custom function defined by the user or a simpler arithmetic operation. The core principle remains defining the desired transformation within the anonymous function (`~ . * operation`).

For example, instead of a statistical standardization, an analyst might need to apply a simple adjustment, such as multiplying all 'starter' related statistics by a constant factor (e.g., two) to adjust for scoring rules or weighting. This simple alteration to the anonymous function demonstrates the adaptability of the syntax to diverse data manipulation requirements, maintaining the same precise column targeting mechanism.

library(dplyr)

#multiply values by two for each variable that contains 'starter' in the name
df %>% mutate_at(vars(contains('starter')), ~ (. * 2))

team starter_points starter_assists bench_points bench_assists
1 A 44 8 7 2
2 B 52 10 7 5
3 C 50 20 9 5
4 D 26 28 14 4
5 E 30 24 13 9
6 F 44 20 10 14

The output confirms that only the columns `starter_points` and `starter_assists` have been modified, with their values doubled. The remaining columns, including those for the bench players, are left undisturbed. This illustrates how the conditional selection based on the column name string 'starter' provides absolute control over which variables are subject to the defined transformation, making this methodology ideal for large-scale, automated data processing tasks.

Conclusion: Mastering Targeted Data Wrangling

The strategic use of the `mutate_at()` function, expertly combined with selection helpers like `vars()` and `contains()`, represents an essential skill for any data professional working in the R environment. This approach offers an elegant, efficient, and highly readable solution for conditionally mutating variables based on specific patterns in their column names. By automating the selection process, analysts can write code that is less error-prone, easier to maintain, and readily adaptable to changes in data structure. Mastering this targeted data wrangling technique allows for greater precision and efficiency in diverse analytical workflows, from exploratory data analysis to complex predictive modeling.

Additional Resources

For those seeking to delve deeper into the capabilities of **dplyr** and other facets of data manipulation in R, the following resources are highly recommended. These tutorials and official documentation sources can help advance your understanding of scoped verbs and optimize your data preparation processes.