Learn How to Use String Variables as Column Names in dplyr

Authored by

Mohammed looti

May 1, 2026

When developing scalable and reusable scripts for data analysis in R, particularly when utilizing the industry-standard data manipulation package, dplyr, programmers frequently encounter a need for **dynamic column selection**. This scenario arises when the name of the column required for an operation--such as filtering, selecting, or mutating--is not hardcoded but is instead stored within a variable as a string. This capability is absolutely essential for writing flexible functions, automating repetitive analytical workflows, and integrating dplyr commands within larger applications where the input data structure may vary.

However, simply passing a string variable directly into a standard dplyr verb, such as `filter()` or `select()`, often fails to produce the desired result. This behavior is rooted in Non-Standard Evaluation (NSE), a design choice that prioritizes syntactic convenience for interactive data analysis but complicates programmatic access. This comprehensive guide details the technical reasons behind this challenge and, more importantly, provides two robust, authoritative methods--one using base R and one using the Tidyverse framework--to effectively use a string variable as a column reference within any dplyr operation.

The Core Conflict: Non-Standard Evaluation (NSE) in dplyr

The dplyr package, a cornerstone of the modern Tidyverse, gains much of its celebrated power and readability from its use of Non-Standard Evaluation (NSE). NSE allows developers to write code that is highly intuitive: instead of typing `df$column_name` or quoting the column name like `df`, dplyr functions permit the use of unquoted column names directly within their arguments. This abstraction significantly streamlines interactive data transformation tasks, making the code feel closer to natural language.

The challenge arises when shifting from interactive analysis to programmatic scripting. When a string containing a column name is stored in a variable (e.g., `my_col <- "points"`), and this variable is passed to a function like `filter()`, dplyr's NSE mechanism attempts to evaluate the variable itself, rather than the content it holds. Specifically, dplyr searches for a column in the data frame that is literally named `my_col`. If such a column does not exist--which is the case when `my_col` is meant to represent another column, such as 'points'--the operation fails or, more misleadingly, returns an empty result set.

Understanding this distinction is crucial: dplyr, by default, expects an expression that references a column name, not a variable containing that name as a string. To bridge this gap and enable dynamic execution, we must employ specific techniques to instruct R (or the underlying dplyr environment) to first evaluate the variable, retrieve the string value, and then use that resulting string as the column identifier. The following example demonstrates this common failure point when attempting to filter based on a dynamically stored column name.

library(dplyr)

```
#define variable
my_var <- 'team'

#attempt to filter for rows where team is equal to a variable
df %>% filter(my_var == 'A')

team points assists rebounds
<0 rows> (or 0-length row.names)
```

As clearly shown in the output, this straightforward application of the string variable `my_var` yields an empty result. This confirms that the `filter()` function was looking for a literal column named `my_var`, rather than accessing the column specified by the string value 'team'. To successfully perform dynamic column selection, we must adopt methods that explicitly handle Non-Standard Evaluation.

Setting the Stage: Example Data for Dynamic Filtering

To provide a tangible context for the solutions we will explore, we first need a well-defined, reproducible data frame. This example data set is intentionally small and simple, allowing us to clearly observe how both the base R and Tidyverse programmatic methods successfully handle dynamic column references during filtering operations within a dplyr pipeline. We will focus on dynamically referencing the 'team' column.

```
#create data frame
df <- data.frame(team=c('A', 'A', 'A', 'B', 'B'),
points=c(99, 90, 86, 88, 95),
assists=c(33, 28, 31, 39, 34),
rebounds=c(30, 28, 24, 24, 28))

#view data frame
df

team points assists rebounds
1 A 99 33 30
2 A 90 28 28
3 A 86 31 24
4 B 88 39 24
5 B 95 34 28
```

The data frame, named `df`, contains five rows of hypothetical sports statistics, including columns

for `team`, `points`, `assists`, and `rebounds`. Our objective is to write a flexible `filter()` call that uses a string variable, which holds the value 'team', to select only the rows associated with Team 'A'. This setup provides the perfect foundation to test our two primary solutions for dynamic variable manipulation in dplyr.

Method 1: Leveraging Base R with the `get()` Function

The first and most universally accessible solution involves incorporating the base R function, `get()`. This function is a powerful tool designed specifically to retrieve the value of an object whose name is provided as a string. When utilized inside a dplyr verb like `filter()`, `get()` acts as a temporary escape hatch from the standard Non-Standard Evaluation rules, allowing the column reference to be resolved dynamically before the filtering condition is evaluated.

The implementation is straightforward: instead of referencing the variable directly in the condition (e.g., `my_var == 'A'`), you wrap the variable holding the column name within the `get()` function (e.g., `get(my_var) == 'A'`). When dplyr processes the `filter()` argument, the `get()` function executes first. If the variable `my_var` contains the string 'team', then `get(my_var)` resolves to the actual vector data contained within the 'team' column of the data frame. This resolution happens seamlessly, making the comparison valid and producing the correct subset of data.

This technique is highly valued because it leverages a fundamental base R capability, meaning it is robust, well-documented, and does not require specific knowledge of the Tidyverse's internal evaluation mechanisms. For simple, ad-hoc programmatic tasks or when working in environments where relying solely on base R functions is preferred, `get()` provides an efficient and easily readable solution to dynamic column access.

library(dplyr)

#define variable
my_var <- 'team'

#filter for rows where team is equal to a variable
df %>% filter(get(my_var) == 'A')

team points assists rebounds
1 A 99 33 30
2 A 90 28 28
3 A 86 31 24

Method 2: The Tidyverse Approach via the `.data` Pronoun

For those firmly entrenched within the Tidyverse ecosystem, dplyr provides a specific and idiomatic solution designed to handle programmatic operations: the `.data` pronoun. This pronoun is an integral part of tidy evaluation, dplyr's framework for managing how expressions are interpreted. The `.data` pronoun explicitly refers to the data mask--the current data frame context within the pipeline--and allows column names to be accessed using standard list-like indexing with the `]`, we explicitly signal to dplyr that the value stored within the variable `my_var` (which is a string) should be used as the name to look up a column within the current data frame. This differs fundamentally from NSE, where the variable name itself is sought. The use of the double bracket `] == 'A')`

team points assists rebounds
1 A 99 33 30
2 A 90 28 28
3 A 86 31 24

The output confirms that using `.data]` within the `filter()` command successfully performs the dynamic column selection, demonstrating the power and intended use of tidy evaluation within dplyr.

Choosing the Right Tool: `get()` vs. `.data` in Practice

When faced with the need for dynamic column selection, developers must decide between the base R approach (`get()`) and the Tidyverse approach (`.data`). While both methods reliably solve the problem of passing a string as a column identifier, they belong to different programming philosophies and offer distinct advantages based on the context of the script or function being written.

The `get()` function provides **universality**. Since it is part of base R, its use is not limited to dplyr or the Tidyverse. It is a robust choice when integration across different packages is necessary or when readability is prioritized for programmers who may not be familiar with the nuances of tidy evaluation. It is particularly effective for small, self-contained scripts where complexity is minimal. However, in highly complex functional programming scenarios within dplyr, `get()` can occasionally introduce scope ambiguity if an object with the same name exists both in the data frame and in the global environment.

In contrast, the `.data` pronoun offers **scope clarity and integration**. It forces the column lookup to occur strictly within the context of the data frame being piped through the dplyr operation, eliminating the risk of accidental variable retrieval from the global environment. For developers building robust functions, packages, or intricate data pipelines relying heavily on the Tidyverse

framework, `.data` is generally the recommended standard. It adheres to the package's design principles, ensuring better compatibility and maintainability as the Tidyverse evolves.

Ultimately, the choice often boils down to the programming environment: if you are writing code that must operate outside the Tidyverse or requires maximum backward compatibility, `get()` is an excellent choice. If you are developing idiomatic dplyr code meant to scale within the Tidyverse framework, adopting the `.data` pronoun is the professional standard for tidy evaluation.

Summary and Best Practices

Mastering the technique of passing a string as a column name is a critical step in transitioning from interactive R scripting to powerful, production-ready programming. The necessity for these programmatic solutions stems directly from dplyr's reliance on Non-Standard Evaluation, which requires explicit instructions when referencing variables dynamically.

We have established two reliable methods for overcoming this hurdle. The base R function, `get()`, offers an immediate and universally applicable solution by forcing the evaluation of the string variable before dplyr processes the expression. Conversely, the `.data` pronoun, accessed via the double bracket operator (`.data[]`), provides the canonical Tidyverse solution, offering superior scope control and integration for advanced dplyr programming.

By integrating these dynamic programming patterns into your workflow, you ensure that your dplyr scripts are not only readable but also highly adaptable and reusable across diverse data analysis tasks. We highly recommend exploring the broader topic of tidy evaluation to further unlock the full potential of programmatic data manipulation in R.

Additional Resources

To further enhance your dplyr and R programming skills, explore these related tutorials:

Programming with dplyr
Advanced R by Hadley Wickham
The R Manuals