How to Group All But One Column in R with dplyr

Authored by

stats writer

November 21, 2025

Using the powerful dplyr package within the R programming environment allows users to manipulate and structure data efficiently. A common requirement in data analysis is to perform aggregations or transformations across a subset of columns, often grouping by every column except one specific variable. Instead of manually listing dozens of column names for the grouping operation, modern dplyr syntax offers an elegant and scalable solution. This method leverages the `across()` function in conjunction with negative column selection to create a concise and readable data pipeline. This approach ensures that your code remains clean and maintainable, especially when dealing with wide datasets, allowing you to group all columns together except the designated exclusion.

Mastering Efficient Column Selection in R

In the world of data wrangling using Tidyverse principles, efficiency and clarity are paramount. Traditional methods for grouping data required explicitly naming every variable intended for grouping. When a dataset contains numerous explanatory variables and only one variable needs to be excluded from the group definition--perhaps a measurement column like `points` or `value`--listing all other columns is tedious and error-prone. The advanced functionality of the dplyr library provides a streamlined way to handle this exclusion using positional or name-based selection helpers directly inside the group_by() verb.

The solution utilizes the `across()` selector coupled with the negation operator (`-`) inside the column specification. This combination signals to dplyr that the grouping operation should span 'across' all columns, 'except' for the one specified with the negative sign. This is particularly useful when preparing data for summary statistics where you want to calculate metrics (like maximum, minimum, or mean) based on every possible combination of categorical variables present in the data frame.

Understanding this specific syntax is crucial for leveraging the full potential of functional programming paradigms in R. This method ensures that if columns are added or removed from the source data frame later, the grouping logic automatically adapts without requiring manual code changes, significantly improving code resilience and future proofing your analysis scripts.

Understanding the Core Syntax for Exclusion Grouping

To group by all columns except a specific one in a data frame using the dplyr package in R, you apply the pipeline operator (`%>%`) followed by the group_by() function, which internally calls the selection helper across(). This approach makes the intention of the code immediately clear to anyone reading it.

You can use the following basic syntax to group by all columns but one in a data frame using the

package in R:

df %>%
group_by(across(c(-this_column)))

This particular example groups the data frame by all of the columns except the one called **this_column**.

The structure is designed to be declarative. The use of `c()` within across() is essential here, as it allows for column specification using tidy selection rules. The most critical component is the negative sign (`-`) placed directly before the column name you wish to omit.

Note that the negative sign (**-**) in the formula tells dplyr to exclude that particular column in the group_by() function.

This negative selection works robustly because the across() function, when used without a preceding argument, defaults to selecting 'all' columns in the data context, and the negative sign then removes the specified column from that 'all' set. The resulting vector of column names is then passed to the group_by() function, defining the unique groupings for subsequent operations like summarization or mutation.

Diving Deep into across() and Negative Selection

The function across() was introduced in recent versions of dplyr to facilitate applying the same transformation or selection logic across multiple columns simultaneously. While it is primarily known for applying functions (like `mutate(across(...))`), its use within group_by() is a powerful application of its column selection capabilities.

When across() is utilized inside group_by(), it fundamentally serves as a sophisticated column selector. The arguments passed inside `c()` follow standard tidy selection semantics. These semantics allow for various methods of column specification, including: by name (e.g., `team`), by position (e.g., `1:3`), by data type (e.g., `where(is.numeric)`), or, in our specific case, by exclusion using the negative sign (e.g., `-points`).

The ability to use the negative operator (`-`) simplifies the process immensely compared to older dplyr methods which might have required using `select()` beforehand to filter the grouping variables, or explicitly typing out all column names. By using the exclusion syntax, the code immediately communicates the desired outcome: group by the entire context of the data frame, minus this one column. This principle of exclusion is highly scalable and prevents mistakes associated with missed variables when manually listing columns.

Practical Application: Setting Up the Basketball Data Frame

To illustrate this efficient grouping mechanism, we will work with a sample dataset containing information about basketball players. This example demonstrates how to prepare the data and then apply the grouping logic to find aggregate statistics based on player attributes, excluding only the numeric score column.

The following example shows how to use this syntax in practice.

Example: Group by All But One Column in dplyr

Suppose we have the following data frame in R that contains information about various basketball players, including their team, position, starter status, and points scored:

```
#create data frame
df <- data.frame(team=c('A', 'A', 'A', 'A', 'B', 'B', 'B', 'B'),
position=c('G', 'G', 'F', 'F', 'G', 'G', 'F', 'F'),
starter=c('Y', 'Y', 'Y', 'N', 'Y', 'N', 'N', 'N'),
points=c(99, 104, 119, 113))
```

#view data frame
df

team position starter points
1 A G Y 99
2 A G Y 104
3 A F Y 119
4 A F N 113
5 B G Y 99
6 B G N 104
7 B F N 119
8 B F N 113

This data frame, `df`, contains four columns. Three columns (`team`, `position`, `starter`) are categorical or grouping variables, and one column (`points`) is the quantitative variable we intend to aggregate. Our goal is to calculate the maximum points achieved for every unique combination of team, position, and starter status without having to type out `team, position, starter` explicitly in the group_by() call.

The structure of this task perfectly aligns with the use case for exclusion grouping. By telling dplyr to group by everything *except* `points`, we achieve the desired grouping across all three descriptive

columns efficiently. This setup is highly realistic for many data analysis scenarios where descriptive metadata needs to be preserved for grouping while numeric metrics are analyzed.

Implementing Exclusion Grouping: Analyzing Player Data

Now suppose we would like to find the max value in the **points** column, grouped by every other column in the data frame. We will use the `mutate()` function after grouping to calculate the maximum points achieved within each defined group.
We can use the following syntax to do so, demonstrating the power and simplicity of the exclusion method:

library(dplyr)

#group by all columns except points column and find max points
df %>%
group_by(across(c(-points))) %>%
mutate(max_points = max(points))

A tibble: 8 x 5
Groups: team, position, starter
team position starter points max_points

1 A G Y 99 104
2 A G Y 104 104
3 A F Y 119 119
4 A F N 113 113
5 B G Y 99 99
6 B G N 104 104
7 B F N 119 119
8 B F N 113 119

The resulting output is an augmented data frame (specifically, a tibble) that retains the grouping structure. The new column, `max_points`, contains the maximum value of the `points` column found within the group defined by the remaining columns (`team`, `position`, and `starter`). Notice the output explicitly states the grouping variables: `Groups: team, position, starter` , confirming that the exclusion worked exactly as intended.

The use of `mutate()` here, rather than `summarise()`, is intentional. While `summarise()` would collapse the data into one row per group, `mutate()` calculates the maximum value and attaches it back to every row belonging to that specific group. This allows for direct comparison between individual player scores and the group maximum, which is often crucial for downstream statistical

modeling or visualization tasks.

Interpreting the Grouped Aggregation Results

Analyzing the output of the grouped mutation confirms how the exclusion criteria successfully created distinct groupings based on the combination of `team`, `position`, and `starter` status. The aggregation function, `max(points)`, then calculated the highest point value for each of these unique groups.

From the output we can see, for example:

The **max points** value for all players who had a **team** value of A, **position** value of G, and **starter** value of Y was **104**. This maximum is repeated for both rows (row 1 and row 2) that share these grouping characteristics.
The **max points** value for all players who had a **team** value of A, **position** value of F, and **starter** value of Y was **119**. Since this group only had one member in our sample, the maximum matches the individual score.
The **max points** value for all players who had a **team** value of A, **position** value of F, and **starter** value of N was **113**. This demonstrates the grouping working across all three categorical variables simultaneously.

And so on.

This method of deriving group-specific metrics and attaching them back to the original data frame is a cornerstone of advanced data manipulation in R, often referred to as window function operations or non-aggregating transformations. The efficiency gained by using `group_by(across(c(-points)))` minimizes the cognitive load and ensures that the grouping logic dynamically adapts to any changes in the underlying dataset schema.

Alternative Approaches to Column Exclusion

While the `across(c(-column_name))` method is the most modern and recommended approach for excluding a single column, it is helpful to understand the alternative, explicit method. This approach confirms that the concise syntax achieves the identical result as manually spelling out every grouping variable, thereby validating its accuracy and convenience.

Note that we could also get the same result if we typed out every individual column name except **points** in the group_by() function, although this is less scalable:

library(dplyr)

#group by all columns except points column and find max points

```
df %>%
group_by(team, position, starter) %>%
mutate(max_points = max(points))

# A tibble: 8 x 5
# Groups: team, position, starter
team position starter points max_points

1 A G Y 99 104
2 A G Y 104 104
3 A F Y 119 119
4 A F N 113 113
5 B G Y 99 99
6 B G N 104 104
7 B F N 119 119
8 B F N 113 119
```

This matches the result from the previous example, confirming the logical equivalence between the two methods.

For datasets containing a large number of descriptive variables (e.g., 20 columns), manually typing out 19 column names becomes prohibitively slow and increases the risk of typographical errors. Furthermore, if the source data structure changes--for instance, if a new column like `conference` is added--the explicit method requires mandatory modification of the R code. In contrast, the `across(c(-points))` approach automatically includes the new `conference` column in the grouping without any modification, demonstrating superior adaptability and robustness for complex data pipelines.

Why This Exclusion Method is Superior for Data Pipelines

The method of using `group_by(across(c(-column)))` encapsulates best practices in modern data analysis within R. It adheres to the Tidyverse philosophy by being expressive, readable, and highly functional. This technique is not just a syntax shortcut; it represents a more intelligent way to interact with data structure.

Firstly, it promotes **non-breaking changes**. In dynamic data environments where the number of variables might shift, relying on exclusion rather than inclusion ensures that the analytic code remains stable. If your analysis depends on grouping by all available categorical variables, this method guarantees that new variables are automatically included, preventing analytical omissions.

Secondly, it significantly **enhances code readability**. A user encountering `group_by(across(c(-`

`score)))` instantly understands the intent is to group by the entire context except the score column, regardless of how many columns are actually present. This is far clearer than seeing a long list of variable names, which requires careful manual verification.

In conclusion, while multiple paths exist to achieve the desired grouping result, the combination of group_by() and across() with negative selection is the definitive, professional standard for efficient and scalable data manipulation in R when the goal is to group by all variables except a specified subset.

Further Reading on dplyr Operations

For users looking to expand their knowledge of data manipulation in R, especially using the dplyr package, there are several foundational topics related to grouping, selection, and aggregation that are essential for building complex data workflows.

The following tutorials explain how to perform other common tasks using dplyr: