Learning to Extract First and Last Rows by Group with dplyr

Authored by

Mohammed looti

November 13, 2025

The Challenge of Grouped Slicing in R

Data analysis frequently requires us to work with subsets of data, particularly when dealing with structured or panel data where observations are nested within specific categories or groups. A common necessity is selecting the boundary observations--the very first and the very last row--within each of these defined groups. This operation is critical for tasks such as calculating duration, tracking initial and final states, or summarizing chronological data. While base R offers methods to handle this through complex indexing and looping, these approaches often result in code that is difficult to read, maintain, and debug.

The complexity arises because standard row indexing in a data frame applies globally, ignoring the internal group structure. To correctly identify the first row of "Group A" and the last row of "Group A," and then repeat that process for "Group B," we must first partition the data and then apply the indexing locally within those partitions. This necessity highlights the value of specialized libraries designed for data manipulation.

Fortunately, the dplyr package, a fundamental component of the Tidyverse ecosystem, provides elegant and highly readable solutions for these complex grouping and filtering tasks. By leveraging the package's powerful verbs, analysts can express sophisticated data manipulation logic in a concise, pipeline-friendly manner, significantly improving workflow efficiency and code clarity compared to traditional methods.

Introducing the dplyr Solution: Syntax and Logic

To efficiently select the first and last row within every group of a data frame, we combine three core dplyr functions: **group_by()**, the pipe operator (**%>%**), and **filter()**, along with the specialized helper functions **row_number()** and **n()**. This syntax forms a powerful and standard pattern for manipulating grouped data in R.

The general syntax for achieving this selection is remarkably straightforward. We first define the grouping variable (e.g., a column named **team**) using the group_by() function. Once the data frame is logically partitioned, the subsequent operations--such as filtering--are applied independently to each group. The filtering logic uses **row_number()**, which assigns a sequential index (starting at 1) to the rows *within* the current group being processed.

The complete syntax leverages the **filter()** function to select rows where the internal index matches either the first position (**1**) or the last position, which is determined dynamically by the helper function **n()**. The function **n()** returns the total number of rows present in the current group. Therefore, the condition **row_number() %in% c(1, n())** instructs filter() to retain only those rows that are ranked first or last within their respective partitions.

library(dplyr)

df %>%
group_by(team) %>%
filter(row_number() %in% c(1, n()))

Prerequisites and Package Installation

Before executing any of the advanced data manipulation commands provided by the Tidyverse, it is essential to ensure that the necessary package is installed and loaded into your current R session. The **dplyr** package is not part of the standard R installation and must be explicitly installed using the appropriate console command. This step is a one-time requirement per machine, though updating the package periodically is recommended.

To install the package, you can use the following command in the R console. It is important to note that this command will download the package and its dependencies from CRAN (Comprehensive R Archive Network) and make them available on your system.

install.package('dplyr')

Once the installation is complete, the package must be loaded into the current environment before its functions can be called. This is achieved using the **library()** function. Failure to load the library will result in errors when attempting to use functions like group_by() or filter(), as the R interpreter will not recognize them. After confirming the successful loading of the dplyr package, you are ready to apply its powerful capabilities to slice and summarize your data frames.

Practical Example: Setting Up the Data Frame

To illustrate the application of this slicing technique, let us construct a sample data frame containing hypothetical statistics for several basketball players across two different teams. This setup mimics common real-world datasets where multiple observations (rows) belong to distinct categorical groups (teams), requiring group-wise operations.

Our data frame, named **df**, includes four columns: **team** (the grouping variable), **points**, **assists**, and **rebounds**. The data contains four entries for Team A and four entries for Team B, providing a perfect scenario to test our ability to isolate the first and last observation for each team independently.

#create data frame
df <- data.frame(team=c('A', 'A', 'A', 'A', 'B', 'B', 'B', 'B'),

points=c(99, 68, 86, 88, 95, 74, 78, 93),
assists=c(22, 28, 45, 35, 34, 45, 28, 31),
rebounds=c(30, 28, 24, 24, 30, 36, 30, 29))

#view data frame
df

team points assists rebounds
1 A 99 22 30
2 A 68 28 28
3 A 86 45 24
4 A 88 35 24
5 B 95 34 30
6 B 74 45 36
7 B 78 28 30
8 B 93 31 29

Upon inspection of the created data frame, we clearly see the structure: rows 1 through 4 belong to Team A, and rows 5 through 8 belong to Team B. Our goal is to extract rows 1 and 4 for Team A, and rows 5 and 8 for Team B. This requires the filter() operation to be context-aware, which is precisely what group_by() enables.

Implementing the Solution: Slicing First and Last Rows

We now apply the core dplyr syntax to achieve the desired slicing. The code utilizes the pipe operator (**%>%**) to pass the data frame **df** sequentially through the grouping and filtering steps. This pipeline structure enhances readability by making the sequence of data transformations explicit and linear.

First, **df** is piped into group_by(team), which prepares the data for group-wise calculations. Then, the grouped data is passed to filter(row_number() %in% c(1, n())). When the filter processes Team A, **n()** equals 4, so it selects rows 1 and 4. When it processes Team B, **n()** also equals 4, and it selects the first (row 5 in the original data, but row 1 relative to the group) and the last (row 8 in the original data, but row 4 relative to the group).

library(dplyr)

#select first and last row for each team
df %>%
group_by(team) %>%
filter(row_number() %in% c(1, n()))

A tibble: 4 x 4
Groups: team
team points assists rebounds

1 A 99 22 30
2 A 88 35 24
3 B 95 34 30
4 B 93 31 29

The resulting output is a new data structure (a tibble, which is dplyr's modern take on a data frame) containing exactly four rows--the first and last observation for Team A, and the first and last observation for Team B. Notice that the output confirms the selection: for Team A, we kept the row with 99 points and the row with 88 points, and for Team B, we kept the row with 95 points and the row with 93 points, successfully isolating the boundary conditions for both groups.

Advanced Slicing Variations

The elegance of the filter() function combined with row_number() and **n()** is its flexibility. While selecting the first and last row is a common requirement, the analyst often needs to select only the starting observation or only the final observation within each group. The syntax allows for easy modification of the criteria supplied to the **%in%** operator to achieve these specific goals.

If the objective is to extract only the initial observation for every group--perhaps to establish a baseline measurement--we simply modify the criteria within the **filter()** function to include only the index **1**. This targets the first row relative to the group, discarding all others, including the final row identified by **n()**.

library(dplyr)

#select first row for each team
df %>%
group_by(team) %>%
filter(row_number() %in% c(1))

A tibble: 2 x 4
Groups: team
team points assists rebounds

1 A 99 22 30
2 B 95 34 30

Conversely, if the requirement is to retain only the final observation within each group--useful for determining terminal values or end-of-period summaries--the criteria must be updated to exclusively use **n()**. This instructs the filter() function to match the index of the last row in the currently processed group.

library(dplyr)

#select last row for each team
df %>%
group_by(team) %>%
filter(row_number() %in% c(n()))

A tibble: 2 x 4
Groups: team
team points assists rebounds

1 A 88 35 24
2 B 93 31 29

This modular approach ensures that analysts can easily customize the row selection logic. For example, if you needed the first three rows, you would use **row_number() %in% c(1, 2, 3)**. If you needed the second-to-last row, you would use **row_number() %in% c(n()-1)**. The combination of row_number() and **n()** grants precise control over group boundaries.

Conclusion and Further Exploration

The ability to efficiently slice the first and last rows of data within defined groups is a fundamental skill in modern data preprocessing using R. By leveraging the power of the dplyr package, particularly the synergy between group_by() and the conditional filter() logic, data analysts can perform complex subsetting operations with minimal, highly readable code. This method not only improves efficiency but also drastically reduces the potential for indexing errors common in base R alternatives.

It is also important to remember the flexibility of the group_by() function. If your analysis requires partitioning the data based on multiple criteria--for instance, grouping by **team** and then by **year**--you can simply include all relevant variables inside the function: **group_by(team, year)**. The subsequent use of **row_number() %in% c(1, n())** will then correctly identify the first and last observation for every unique combination of team and year.

Mastering these core dplyr functions provides a robust foundation for tackling almost any data manipulation challenge. Analysts are encouraged to experiment with different combinations of

indices specified after the **%in%** operator within the filter() function to meet diverse data extraction needs, moving beyond simple boundary selection to more generalized row slicing.

Additional Resources

For those interested in expanding their proficiency with R and the Tidyverse, the following resources provide further tutorials and explanations on performing common data manipulation tasks:

Tutorial on using **slice_head()** and **slice_tail()** for alternative slicing approaches.

Guide to leveraging the **mutate()** function within grouped operations.

Deep dive into the intricacies of the R pipe operator (**%>%**).