# How to Use n() Function in R (With Examples)

Authored by

**stats writer**

November 22, 2025

## Introduction to the n() Function in R

The **n()** function is a powerful and indispensable tool provided by the dplyr package in R. Its primary purpose is to count the number of observations or rows within a specific context, typically after data has been segregated into groups using group_by(). This function is designed to be used exclusively inside data manipulation verbs like summarise(), mutate(), and filter(), making it fundamental for performing group-wise aggregations and conditional filtering based on group size. Understanding how to integrate the n() function into your workflow is essential for effective data analysis and reporting in R.

While **n()** seems simple, its utility is vast, enabling sophisticated data transformations that would be complex to achieve with base R functions alone. We will explore the three most common and practical applications of this function, demonstrating how it can be used to count, append counts, and filter groups based on size.

The versatility of **n()** stems from its ability to dynamically recognize the current grouping context established by the preceding data pipeline. When chained with the pipe operator (**%>%**), it automatically calculates the count for the subset of data currently being processed, allowing for highly efficient, readable, and reproducible code.

## Three Essential Methods for Using n()

Depending on the desired outcome--whether you need a summary table, an appended column, or a filtered subset--the **n()** function is paired with different dplyr verbs. Mastery of these three core methods covers the vast majority of use cases for counting observations by group in R.

The following outlines the general syntax for each application. Note that in all cases, the data frame (**df**) must first be grouped by the relevant categorical variable (**group_variable**) to define the boundaries within which **n()** operates.

### Method 1: Using n() to Count Observations by Group (Summarization)

This method is used when the goal is to produce a concise summary table showing the count of observations for each unique level of the grouping variable. The summarise() function collapses the grouped data into a single row per group, providing an aggregate count.

```
df %>%
group_by(group_variable) %>%
summarise(count = n())
```

### Method 2: Using n() to Add Column that Shows Observations by Group (Mutation)

If the requirement is to retain all original rows of the data frame while simultaneously adding a new column that indicates the total count of the group that row belongs to, the mutate() function is employed. This is particularly useful for subsequent calculations or visualizations that depend on group size without losing row-level detail.

```
df %>%
group_by(group_variable) %>%
mutate(count = n())
```

### Method 3: Using n() to Filter Based on Observations by Group (Filtering)

The final core application involves using **n()** inside the filter() function. This allows analysts to subset the data based on group size, selecting only those groups that meet a certain numerical threshold (e.g., groups with more than 15 observations).

```
df %>%
group_by(group_variable) %>%
filter(n() > 15)
```

## Preparing the Sample Data Frame

To illustrate these methods practically, we will utilize a simple sample data frame containing hypothetical statistics for several basketball players across three different teams (A, B, and C). This data structure includes critical variables such as **team**, **points**, **assists**, and **rebounds**. The primary objective in the following examples will be to analyze counts based on the **team** variable.

The creation and structure of our example data frame are detailed below. It is important to load the data into the R environment exactly as shown to ensure the reproducibility of the upcoming examples.

```
#create data frame
df <- data.frame(team=c('A', 'A', 'A', 'B', 'B', 'C'),
points=c(22, 25, 25, 20, 29, 13),
assists=c(10, 12, 9, 4, 11, 10),
rebounds=c(9, 8, 5, 10, 14, 12))

#view data frame
df
```

team points assists rebounds
1 A 22 10 9
2 A 25 12 8
3 A 25 9 5
4 B 20 4 10
5 B 29 11 14
6 C 13 10 12

As visible in the output, the data frame contains six total observations. Team A is represented three times, Team B two times, and Team C only once. These internal counts are what the **n()** function will calculate when grouped by the **team** variable.

## Example 1: Counting Observations by Group using summarise()

Our first demonstration utilizes the combination of group_by() and summarise() alongside **n()**. This is the most common use case, intended for generating summary statistics, specifically the frequency of occurrence for each category within a variable. By grouping the data by **team** and then applying **n()** within summarise(), we obtain a precise count of players belonging to each team.

The syntax requires first loading the dplyr library, followed by piping the data frame through the grouping and summarization steps. The resulting output is a tidy table that clearly shows the distribution of observations across the three teams.

**library(dplyr)**

#count number of observations by team
df %>%
group_by(team) %>%
summarise(count = n())

# A tibble: 3 x 2
team count

1 A 3
2 B 2
3 C 1

Analyzing the output confirms the initial visual inspection of the raw data frame:

Team **A** contains 3 observations (players).
Team **B** contains 2 observations (players).

Team **C** contains 1 observation (player).

This succinct method is foundational for frequency analysis and quality checks, ensuring data integrity across categorical variables.

## Example 2: Appending Group Counts using mutate()

In contrast to summarization, there are many scenarios where the analyst needs to retain the complete, original detail of the data while enriching it with group-level metadata. This is achieved using the mutate() function. When **n()** is used within mutate() following a group_by() operation, it calculates the group size and adds that value as a new column to every row corresponding to that group.

The following code demonstrates how the group count is appended to the original data frame. Notice that the resulting data structure maintains all six original rows, but a new column, **count**, has been added.

**library(dplyr)**

#add new column that shows number of observations by team
df %>%
group_by(team) %>%
mutate(count = n())

# A tibble: 6 x 5
# Groups: team
team points assists rebounds count

1 A 22 10 9 3
2 A 25 12 8 3
3 A 25 9 5 3
4 B 20 4 10 2
5 B 29 11 14 2
6 C 13 10 12 1

The newly created **count** column reflects the team count for each respective row. For example, all rows belonging to Team A now show a count of 3, while the single row for Team C shows a count of 1. This method is particularly valuable when calculating proportions or performing weighted analyses where the size of the group is a necessary factor at the individual record level.

## Example 3: Filtering Based on Group Size using filter()

The final application explores conditional subsetting, leveraging **n()** within the filter() verb. This technique allows for the selective removal of entire groups based on whether their size meets a specified criterion. For instance, an analyst might wish to exclude groups that are too small to yield statistically reliable results.

In the following demonstration, we apply a filter to keep only those teams that have a player count greater than one. This will exclude Team C, which only has a single observation.

**library(dplyr)**

#filter rows where team count is greater than 1
df %>%
group_by(team) %>%
filter(n() > 1)

# A tibble: 5 x 4
# Groups: team
team points assists rebounds

1 A 22 10 9
2 A 25 12 8
3 A 25 9 5
4 B 20 4 10
5 B 29 11 14

Notice the structure of the resulting data frame: it contains only five rows, corresponding to Teams A and B. Team C has been entirely removed because its count (1) did not satisfy the condition (count > 1). This is a highly efficient way to clean data by removing sparsely populated groups without the need for manual calculation or complex joining operations.

## Conclusion and Best Practices

The **n()** function, when used within the dplyr framework, offers a precise and elegant solution for counting observations relative to defined groups. Whether the goal is aggregation (using summarise()), enrichment (using mutate()), or subsetting (using filter()), **n()** simplifies complex counting logic into a single, highly readable expression.

Key to utilizing **n()** effectively is always ensuring that the data has been appropriately grouped using group_by() beforehand, as **n()** operates relative to the current group context. Without

grouping, **n()** simply returns the total number of rows in the entire data frame, which may not be the intended result. Mastering these three distinct patterns allows R users to perform robust group-wise data manipulation with confidence and efficiency.