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

Authored by

**Mohammed looti**

March 20, 2026

In the dynamic field of R programming, especially when performing intensive data manipulation and essential statistical analysis, the ability to accurately count elements within structured subsets--or groups--is paramount. The **dplyr** package, a foundational component of the Tidyverse ecosystem, provides an exceptionally efficient and readable method for achieving this through the powerful **n()** function. This function is specifically designed to count the total number of observations--or rows--within each defined group in your data frame, streamlining what would otherwise require cumbersome base R code.

Mastering the effective utilization of **n()** is vital for anyone seeking to optimize their data analysis workflows in R. It offers a concise, contextual mechanism to gain immediate insights into the underlying distribution and composition of your grouped data, regardless of whether your goal is summarization, transformation, or selective filtering. The function's inherent power lies in its seamless integration within core **dplyr** verbs such as **summarise()**, **mutate()**, and **filter()**. This contextual behavior makes **dplyr** and its helper functions, like **n()**, indispensable tools for modern data scientists and analysts who rely on clarity and efficiency.

This comprehensive guide is dedicated to exploring the three primary and most practical applications of the **n()** function. We will provide detailed, practical examples tailored to illustrate its versatility and power within typical R projects. Our focus will span basic group counts, sophisticated enrichment by adding group counts as a new column to the dataset, and advanced techniques for filtering data subsets based on specific group size criteria. Understanding these methods will significantly elevate your capability in data manipulation and preparation.

## Setting Up the Environment and Example Data

Before diving into the practical applications of the **dplyr** package, it is necessary to establish a clear and reproducible environment. All the methods demonstrated rely heavily on the package's ability to handle grouped operations efficiently. The **dplyr** package, as part of the Tidyverse, must be loaded into the R session before its functions, including **n()**, can be called. A robust understanding of how to set up and inspect the data is fundamental to ensuring the subsequent analyses are accurate and meaningful.

To demonstrate these three powerful methods in a practical context, we will utilize a sample data frame created specifically for this guide. This structured dataset contains hypothetical information about several basketball players, capturing variables such as their team affiliation, points scored, assists recorded, and rebounds collected. This scenario provides an excellent analogy for real-world analytical problems where analysts frequently need to summarize or filter records based on characteristics of their respective groups, such as team size or market segment volume.

The following code snippet demonstrates the creation and immediate inspection of our example data frame, named `df`. This step ensures clarity and reproducibility across all subsequent

examples, establishing a consistent foundation. We can observe that the data comprises six unique observations, distributed across three distinct teams (A, B, and C), thereby providing sufficient variability to showcase the grouping capabilities of **n()**.

```
#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

## Method 1: Counting Observations by Group Using summarise()

The most fundamental and common application of the **n()** function involves summarizing the total count of observations associated with each distinct group within a larger data frame. This operation is the cornerstone of frequency analysis and is crucial for quickly understanding the size and structure of subgroups. In the **dplyr** framework, this summary is elegantly achieved by combining three core functions in sequence: **group_by()**, the pipeline operator (`%>%`), and finally, **summarise()** which contains the **n()** call.

The process begins with **group_by()**, which logically partitions the dataset based on the values of one or more specified categorical variables--in our case, the `team` variable. Once the data is grouped, the **summarise()** function executes a summary computation on each of these newly formed groups. When **n()** is invoked inside **summarise()**, it automatically inherits the group context established by **group_by()** and returns the count of rows present within that specific group. This concise method is exceptionally useful for generating simple frequency tables or deriving quick, group-wise counts, offering immediate visibility into the distribution of the data.

Utilizing our basketball player data frame, the following example demonstrates how to count the number of players (or observations) associated with each unique team. This straightforward code

snippet first loads the necessary R package and then chains the operations to produce a new, aggregated result set. This resulting output is a condensed table, unlike the original data, where each row represents a team and its corresponding size.

```
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
```

The resulting output confirms the counts for each team: Team **A** contains 3 players, Team **B** has 2 players, and Team **C** has 1 player. This summarized view is an excellent starting point for any statistical analysis, allowing analysts to quickly verify group sizes and ensure sufficient representation before proceeding with more complex calculations.

## Method 2: Enriching Data with Group Counts Using mutate()

While summarizing data is frequently necessary, there are many instances where the original row-level granularity must be preserved while simultaneously incorporating group-level metrics. This is often required when individual records need to be assessed against the context of their larger group. For example, a quality control measure might depend on knowing the total production volume of a batch (the group) for every single item (the row). In **dplyr**, this enrichment process is handled by combining **n()** with the **mutate()** function, following the grouping operation.

The key distinction between **summarise()** and **mutate()** is that the latter preserves all original rows and columns of the data frame, adding the computed results as new variables. When **n()** is executed within the scope of a **mutate()** call following **group_by()**, it calculates the group size just as before. However, instead of collapsing the data, **mutate()** repeats this calculated count value for every single row belonging to that specific group. This preserves the full dataset while enriching it with necessary contextual information.

This approach is invaluable for subsequent analytical tasks that require both individual metrics and group size context, without sacrificing the detail of the raw data. It allows for advanced

comparisons, such as calculating the proportion of points scored by a player relative to the team's total player count. The following code demonstrates how we can use this method to add a new column, `count`, to our basketball dataset, showing the number of players on each team next to every player's record.

**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 output clearly displays the newly added `count` column. Every player from Team A now correctly displays a count of 3, while Team B players show 2, and the single player from Team C shows 1. This successful application of **mutate()** with **n()** illustrates a core technique in effective data manipulation, enabling analysts to retain individual granularity while leveraging group-level summaries.

## Method 3: Conditional Filtering Based on Group Counts

One of the most powerful and practical uses of **n()** is its role in conditional filtering. In many analytical scenarios, it is necessary to exclude groups that are either too small (lacking statistical significance) or too large (potentially skewing results). The combination of **group_by()** and **filter()** allows analysts to define dynamic criteria based on the count of observations within each subgroup, ensuring that subsequent analyses are focused solely on relevant and robust data subsets.

The mechanism involves first grouping the data frame by the desired categorical variable (e.g., `team`). Subsequently, the **filter()** function evaluates a logical condition within the context of these groups. By embedding the **n()** function inside the **filter()** argument, we instruct **dplyr** to keep only

those groups where the group count satisfies the specified criterion. For instance, we might choose to retain only teams that have a minimum of two players or transactions exceeding a certain monthly volume.

In our example, we aim to exclude teams that have only one player, focusing our analysis exclusively on groups with two or more members. This is achieved by setting the condition `n() > 1` within **filter()**. This method is fundamentally different from row-wise filtering because the condition is applied to the group as a whole, meaning that if the group count fails the test, all observations belonging to that group are removed, ensuring precise data subsetting for advanced statistical analysis.

**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

The filtered output successfully retains only the rows corresponding to Team A and Team B, which meet the criteria of having more than one player. Team C, which had a count of only 1, is entirely excluded from the result set. This capability of conditional filtering based on group metrics is critical for robust data manipulation, allowing analysts to maintain control over the quality and relevance of the data used in downstream modeling or reporting.

## Comparative Summary of n() Applications

To fully appreciate the versatility of the **n()** function, it is helpful to contrast its behavior across the three primary **dplyr** verbs discussed. Although **n()** always calculates the number of rows in the current group context, the resulting output structure changes drastically depending on whether it is paired with **summarise()**, **mutate()**, or **filter()**. Understanding these differences is essential for choosing the correct method for a given analytical goal.

When used with **summarise()**, the objective is aggregation. The resulting output collapses the dataset, reducing the number of rows to match the number of unique groups. The new column created holds the group count, providing a concise frequency table. This is the optimal method for generating reports on group sizes or calculating summary statistics that require counts, fundamentally changing the dimensionality of the data frame.

In contrast, when **n()** is combined with **mutate()**, the goal is enrichment. The original structure of the data frame is maintained, meaning the number of rows remains unchanged. The group count is added as a new column, repeating the value for all observations within that group. This preservation of row-level data while adding group context is ideal for preparing datasets for complex modeling where individual records must retain their identity but also require group metrics.

Finally, using **n()** with **filter()** serves the purpose of selection and subsetting. The resulting output maintains the structure of the data frame (columns are preserved), but the number of rows is reduced based on a group-level condition. The entire group is either kept or discarded based on whether its count meets the filter criteria. This powerful feature enables analysts to clean datasets by removing underrepresented or insignificant groups, leading to more robust and focused analysis.

## Conclusion: Mastering the n() Function

The **n()** function, residing within the indispensable **dplyr** R package, stands out as a critical utility for anyone engaged in serious data manipulation and statistical analysis in the R environment. Its elegant simplicity belies its functional power, offering an accurate and highly readable means of counting observations within defined subgroups. This capability is central to streamlining countless analytical tasks, ranging from basic data profiling to setting up complex filtering rules.

By gaining proficiency in the three methods detailed in this guide--summarizing counts, enriching datasets with group metrics, and conditionally filtering groups--you are equipped to significantly enhance both the efficiency and the analytical depth of your work with grouped data. These techniques exemplify the core philosophy of the **dplyr** framework: providing efficient, expressive tools that simplify complex data wrangling challenges. The contextual power of **n()** ensures that your code remains clean, maintainable, and highly effective.

We strongly encourage practitioners to apply these examples using diverse datasets, recognizing that hands-on experimentation is the best way to internalize the power and flexibility of the **n()** function. Continuous exploration of the official **dplyr** documentation will further unlock advanced applications and clever combinations with other key functions, ultimately leading to more sophisticated and targeted data analysis outcomes. Harnessing this essential function will undoubtedly contribute to more robust and insightful analytical results in your ongoing R projects.