How to Easily Create Frequency Tables in R Using the Table Function

Authored by

stats writer

December 4, 2025

The ability to quickly summarize and understand data distributions is fundamental to effective statistical analysis. In the R programming language, the native function used for this precise purpose is **table()**. This function serves as an indispensable tool for generating clean, concise summaries, making it one of the most frequently used commands by data scientists and analysts alike for initial exploratory data analysis.

At its core, the **table()** function is designed to construct a frequency table. It accepts various inputs, including vectors, factors, or other related data objects. The output systematically tallies the number of observations for each unique category or value present within the provided input. This immediate visualization of occurrence counts allows researchers to rapidly assess variable distribution, identify potential outliers, and gain initial insights into the dataset's characteristics before proceeding to more complex modeling techniques.

Understanding how to leverage **table()** is crucial for data preparation and exploratory data analysis (EDA). Whether working with categorical variables like demographics or discreet numerical identifiers, the function provides a foundational overview. Throughout this guide, we will explore practical examples demonstrating how to use **table()** effectively, moving from simple single-variable counts to complex multi-dimensional summaries and calculations of proportions.

To illustrate the utility of the **table()** function, we will utilize a small, representative dataset detailing player performance metrics. This dataset is structured as a data frame named `df`, containing observations for six players across three distinct variables: `player` (a unique identifier), `position` (a categorical variable indicating assigned role), and `points` (a quantitative variable representing scores).

Setting up this initial data frame is the essential first step in any R analysis. The following code snippet demonstrates how to construct and display the data structure we will be analyzing throughout this guide. Note the mix of character and numeric data types within the structure, which are ideal candidates for frequency analysis.

The **table()** function in R will be applied to analyze the variables within the data frame presented below:

#create data frame
df <- data.frame(player = c('AJ', 'Bob', 'Chad', 'Dan', 'Eric', 'Frank'),
position = c('A', 'B', 'B', 'B', 'B', 'A'),
points = c(1, 2, 2, 1, 0, 0))

#view data frame
df

player position points

1 AJ A 1
2 Bob B 2
3 Chad B 2
4 Dan B 1
5 Eric B 0
6 Frank A 0

Frequency Table for a Single Categorical Variable

The simplest and most common application of **table()** involves analyzing the distribution of a single categorical variable. This process, often referred to as univariate frequency counting, provides an immediate count of how many times each unique category appears in the selected column. For our illustration, we will focus on the `position` variable, which classifies players into group 'A' or group 'B'.

To generate this count, we pass the specific column of the data frame (using the `df$position` notation) directly into the **table()** function. The resulting output clearly delineates the absolute frequencies, offering straightforward quantitative insights into the group sizes. This step is crucial for checking data balance or identifying dominant categories early in the analysis, ensuring that subsequent statistical models are not unduly biased toward overrepresented groups.

The following code snippet demonstrates the straightforward implementation required to obtain the frequency table for the `position` variable:

#calculate frequency table for *position* variable
table(df$position)

A B
2 4

Upon reviewing the output, the interpretation is unambiguous. We can definitively conclude that the data frame comprises 6 total observations, divided into two distinct groups. Specifically, we observe that 2 players belong to position 'A', while a greater number, 4 players, are assigned to position 'B'. This high-level summary immediately highlights that position 'B' is twice as prevalent as position 'A' in our sample dataset, suggesting a sampling bias or a structural imbalance in the player roles.

2 players in the data frame have a position of '**A**'.
4 players in the data frame have a position of '**B**'.

Calculating Relative Frequencies and Proportions

While absolute counts are useful, analysts often require relative frequencies, expressed as percentages or proportions, to better understand the internal composition of the data relative to the total population size. To transform the raw frequency counts generated by **table()** into relative frequencies, we utilize the dedicated R function **prop.table()**.

The key to using **prop.table()** effectively is recognizing that it must take the output of **table()** as its input argument. This nesting structure ensures that the absolute frequencies are normalized by the total sum of all observations (in this case, 6 players). The resulting values represent the decimal proportion of each category, where the sum of all resulting values must strictly equal 1. This normalization facilitates direct comparison across datasets of differing sizes.

Below is the command sequence for calculating the relative frequency table for the `position` variable. This provides a percentage-based breakdown of the distribution, which is often easier to communicate than raw counts, especially in presentation settings or academic papers where standardized metrics are preferred:

#calculate frequency table of proportions for *position* variable
prop.table(table(df$position))

A B
0.3333333 0.6666667

Interpreting these decimal proportions requires multiplication by 100 to convert them into percentages. The results clearly indicate a significant imbalance between the two positions. Specifically, Position 'A' accounts for approximately 33.33% of the player population, whereas Position 'B' dominates, representing 66.67% of the total sample. A fundamental property of the output from **prop.table()** is that the sum of all proportions across all categories will always sum up exactly to 1 (or 100% when expressed as percentages), serving as an internal consistency check.

33.33% of players in the data frame have a position of '**A**'.
66.67% of players in the data frame have a position of '**B**'.

Cross-Tabulation of Two Variables (Bivariate Counts)

The true power of the **table()** function emerges when performing bivariate analysis, often referred to as cross-tabulation or contingency table analysis. By supplying two input vectors or variables, R constructs a two-dimensional table, displaying the joint frequency distribution. This allows analysts to determine how one variable (e.g., `position`) relates to the other (e.g., `points`).

In this context, we are interested in counting how many players belong to a specific position AND achieved a specific score. The output format is a matrix where the categories of the first variable define the rows, and the categories of the second variable define the columns. Each cell within the matrix contains the absolute count corresponding to the co-occurrence of those two specific categories, offering joint frequency data.

To generate this cross-tabulation, we pass both column vectors--`df$position` and `df$points`--as separate arguments to the **table()** function, separated by a comma. This creates a powerful contingency table that helps visualize the relationship between the categorical position and the discrete points scored, forming the basis for tests like the Chi-squared test of independence.

#calculate frequency table for *position* and *points* variable
table(df$position, df$points)

0 1 2
A 1 1 0
B 1 1 2

Analyzing this two-way frequency table reveals nuanced details that univariate analysis would miss. For example, while we know Position 'A' has two players total, the table clarifies that one player scored 0 points and the other scored 1 point. Similarly, the four players in Position 'B' are distributed across all three possible point scores, with the highest concentration (two players) scoring 2 points. Summing the row totals (1+1+0=2 for A; 1+1+2=4 for B) confirms the total univariate frequencies observed earlier.

1 player in the data frame has a position of '**A**' and **0** points.
1 player in the data frame has a position of '**A**' and **1** point.
0 players in the data frame have a position of '**A**' and **2** points, indicating Position A players did not achieve the maximum score.
1 player in the data frame has a position of '**B**' and **0** points.
1 player in the data frame has a position of '**B**' and **1** point.
2 players in the data frame have a position of '**B**' and **2** points, highlighting better performance for Position B overall.

Bivariate Proportions and Controlling Decimal Precision

When analyzing cross-tabulations, it is often more insightful to view the joint distribution as proportions relative to the grand total, rather than raw counts. Just as in the univariate case, the **prop.table()** function is employed, but this time it takes the two-dimensional output of **table(df$position, df$points)** as its argument. This calculation yields the proportion of the entire

dataset that falls into each combination of position and points.

The resulting matrix, which sums to 1 across all cells, provides an immediate assessment of the overall contribution of each category intersection. For instance, if a specific cell has a high proportion value, it signifies that this combination of attributes is heavily represented in the total dataset. This is particularly useful for identifying dominant subgroups within the sample population and understanding resource allocation or performance concentration.

The code below generates the joint proportion table, showing how the total of 6 players is distributed percentage-wise across the six possible combinations of position and points scored:

#calculate frequency table of proportions for *position* and *points* variable
prop.table(table(df$position, df$points))

0 1 2
A 0.1666667 0.1666667 0.0000000
B 0.1666667 0.1666667 0.3333333

Interpretation of the results confirms that the highest single cell contribution comes from players in Position 'B' who scored 2 points (33.33%). The other three non-zero cells (A-0, A-1, B-0, B-1) each contribute equally, representing 16.67% of the total dataset. This confirms that one-third of our players are high-scorers in Position B, emphasizing the importance of this specific subgroup in driving the overall point distribution.

16.67% of players are in Position '**A**' and scored **0** points.
16.67% of players are in Position '**A**' and scored **1** point.
0% of players in the data frame are in Position '**A**' and scored **2** points.
16.67% of players are in Position '**B**' and scored **0** points.
16.67% of players are in Position '**B**' and scored **1** point.
33.3% of players are in Position '**B**' and scored **2** points.

When presenting these proportions, the default high decimal display can often be cumbersome and reduce readability. R provides the global **options()** function to manage the default number of displayed digits. By setting the `digits` option, we can format the output to show a cleaner, more readable representation, which is especially helpful for creating final reports or visualizations that prioritize clarity over extreme precision.

#only display two decimal places
options(digits=2)

#calculate frequency table of proportions for *position* and *points* variable

prop.table(table(df$position, df$points))

0 1 2
A 0.17 0.17 0.00
B 0.17 0.17 0.33

Deriving Marginal Frequencies and Proportions

While the joint frequency table shows the count for the intersection of two categories, analysts frequently need to view the marginal totals--that is, the row sums or column sums--to understand the distribution of one variable independent of the other, directly from the bivariate table. When using **prop.table()** on a two-dimensional table, we can specify a margin argument (`margin=1` for rows, `margin=2` for columns) to normalize the proportions based on these totals, giving us conditional probabilities.

When `margin=1` is specified, the proportions are calculated such that each row sums to 1. This is interpreted as the conditional distribution: "Given a certain row category (e.g., Position A), what is the proportion of observations across the column categories (Points 0, 1, 2)?" This conditional analysis is essential for comparing performance characteristics or outcomes across different defined groups or treatments.

If we calculate the row-wise proportions for our data, we are asking: for players in Position A, what percentage scored 0, 1, or 2 points? Similarly, for Position B, what were their scores? This normalization reveals the internal distribution of scores within each position group, disregarding the overall size difference between the groups.

 # Row-wise proportions (sums to 1 across rows)
prop.table(table(df$position, df$points), margin = 1)

0 1 2
A 0.50 0.50 0.00
B 0.25 0.25 0.50

The interpretation changes significantly here. For Position A, 50% of players scored 0 points and 50% scored 1 point (and 0% scored 2). For Position B, the distribution is 25% for 0 points, 25% for 1 point, and 50% for 2 points. This conditional analysis immediately highlights that Position B players are much more likely to hit the top score than Position A players, providing strong evidence of differential performance based on position.

Conversely, specifying `margin=2` calculates proportions such that each column sums to 1. This

answers the question: "Given a specific score (e.g., 2 points), what is the proportion of players from each position?" This perspective helps determine which categories contribute most heavily to a specific outcome.

 # Column-wise proportions (sums to 1 across columns)
prop.table(table(df$position, df$points), margin = 2)

0 1 2
A 0.50 0.50 0.00
B 0.50 0.50 1.00

Managing Missing Values (NAs) with table()

In real-world data analysis, missing values, typically represented as `NA` (Not Available) in R, are a common challenge. By default, the **table()** function silently excludes `NA` values from the frequency counts, essentially treating the data as if those observations never existed. However, ignoring missing data can lead to misleading summaries if the missingness itself holds analytical significance or if the goal is to count all records.

To explicitly include and count missing observations within the frequency summary, the `useNA` argument must be set within the **table()** function. This argument accepts three primary values: `"no"` (the default behavior), `"ifany"` (includes NAs only if they exist in the data), and `"always"` (ensures an NA category is displayed, even if empty). For routine reporting, `"ifany"` is often the most practical choice.

To demonstrate this, let us simulate a small alteration to our dataset by introducing a missing value in the `position` vector. We replace one 'B' position with `NA` to observe how **table()** handles this condition when the `useNA` parameter is active, ensuring we maintain the integrity of our total count.

 # Create a temporary data frame with a missing value
df_na <- data.frame(position = c('A', 'B', NA, 'B', 'B', 'A'))

# Default behavior (NAs are omitted)
table(df_na$position)

A B
2 3

# Including NAs using useNA="ifany"
table(df_na$position, useNA = "ifany")

A B
2 3 1

The comparison clearly shows the difference: without `useNA="ifany"`, the total count is 5 (2 A + 3 B), which is incorrect for a six-row dataset. By applying `useNA="ifany"`, a new category labeled `<NA>` is generated, accurately tallying the 1 missing observation. This feature is vital for preliminary data quality checks, ensuring all records are accounted for in the frequency table and preventing silent data loss during analysis.

If we apply **prop.table()** to a table that includes NAs, the resulting proportions will be calculated relative to the entire dataset, including the missing values. This ensures that the sum of all categories, including the NA proportion, equals 1, thus giving an accurate picture of the proportion of missing data.

 # Calculate proportions including NAs
prop.table(table(df_na$position, useNA = "ifany"))

A B
0.3333 0.5000 0.1667

Advanced Usage: Combining table() with Data Subsetting

While **table()** is typically applied to an entire data frame column, analysts frequently need to calculate frequencies only for a specific subset of the data. For instance, we might want to know the distribution of points only for players in Position 'B'. R's indexing capabilities allow us to seamlessly integrate subsetting commands directly into the **table()** function arguments, offering flexibility without creating intermediate data structures.

To subset a vector based on a logical condition, we use square bracket notation (). The condition inside the brackets specifies which rows should be included. For example, to find the points distribution exclusively for Position B players, we filter the `df$points` vector using the condition `df$position == 'B'`. This results in a new, shorter vector containing only the point values relevant to the specified position, which **table()** then processes.

This method provides extreme flexibility, enabling analysts to isolate specific populations or conditions quickly without having to create temporary data objects. Below is the code demonstrating how to calculate the frequency of points scored, conditional solely on belonging to Position 'B', using direct subsetting:

 # Calculate frequency table for points, but only for Position 'B' players

table(df$points)

0 1 2
1 1 2

The resulting frequency table now reflects only the four players in Position B. We see 1 player scored 0 points, 1 player scored 1 point, and 2 players scored 2 points. This level of granularity is crucial when analyzing group performance differences or conducting focused statistical tests on isolated segments of the data, as it ensures the analysis focuses only on the subgroup of interest.

Furthermore, this technique can be extended to subsetting on multiple variables simultaneously, allowing for highly complex conditional frequency counting. For instance, if we wanted to count the positions only for players who scored 1 point or less (i.e., points <= 1), we can apply a combined logical test to the rows of our data frame. This is achieved by applying the filter condition to the column we wish to count (`df$position`).

 # Calculate frequency of positions, but only for players scoring 1 or less
table(df$position)

A B
2 1

In this final conditional example, among the players who scored 1 point or less (AJ, Dan, Eric, Frank), two belonged to Position A and only one belonged to Position B. This confirms the utility of combining **table()** with advanced indexing for targeted exploratory data analysis, allowing for rapid hypothesis generation based on specific subpopulations.