How to Group & Summarize Data in R

Authored by

stats writer

December 23, 2025

The core of effective data analysis often involves simplifying vast amounts of raw information into actionable insights. This simplification process is known as data aggregation or summarization. Traditionally, grouping and summarizing data in R involved using the `aggregate()` function, which takes a data frame and splits it into defined subsets based on the levels of a factor variable.

The `aggregate()` function then applies one or more summary calculations to each of those subsets, producing a new data frame with the concise, summarized results. This procedure is incredibly useful for summarizing large datasets to visualize complex patterns or trends, or to significantly reduce the data size to a more manageable scale suitable for subsequent statistical modeling.

Why Choose the `dplyr` Package for Summarization?

Two of the most common and essential tasks that you will perform in any modern data pipeline are grouping and summarizing data. While base R functionalities exist, the modern standard utilizes the dplyr package, part of the Tidyverse ecosystem. This package provides a highly optimized and specialized grammar for data manipulation that makes complex operations intuitive, readable, and generally faster than traditional methods.

The efficiency and clarity offered by `dplyr`, particularly its use of the piping operator (`%>%`) to chain commands, have established it as the preferred tool for data professionals. This tutorial serves as a quick but comprehensive guide to getting started with `dplyr`, focusing specifically on the crucial functions used for data aggregation: `group_by()` and `summarize()`.

Setting Up Your R Environment: Installing `dplyr`

Before you can utilize the powerful functionality offered by the `dplyr` package, you must ensure it is properly installed and loaded into your current R session. If you are using R for the first time or setting up a new environment, the package must first be installed using the `install.packages()` command.

Once the package files are locally installed, you must use the `library(dplyr)` command to load the package into memory, making its functions available for use during your session. Remember that installation is a one-time process, but loading the library must be done every time you start a new R session.

 # Install dplyr (if not already installed)
install.packages('dplyr')

# Load dplyr into the R environment
library(dplyr)

Case Study Data: Exploring the `mtcars` Dataset

To effectively illustrate the various examples of how to use the functions in `dplyr` to group and summarize data, we will be utilizing the well-known, built-in R dataset called **mtcars**. This dataset contains 32 observations (rows) on 11 variables (columns) related to automobile specifications and performance.

A crucial first step in any data analysis project is inspecting the structure of your data. We can confirm the dimensions of the dataset and then view the first few rows to confirm the column names and data types, which helps us decide which variables are best suited for grouping (like `cyl`) and which are suitable for summarization (like `mpg`).

 # Obtain rows and columns of *mtcars*
dim(mtcars)

32 11

# View first six rows of *mtcars*
head(mtcars)

mpg cyl disp hp drat wt qsec vs am gear carb
Mazda RX4 21.0 6 160 110 3.90 2.620 16.46 0 1 4 4
Mazda RX4 Wag 21.0 6 160 110 3.90 2.875 17.02 0 1 4 4
Datsun 710 22.8 4 108 93 3.85 2.320 18.61 1 1 4 1
Hornet 4 Drive 21.4 6 258 110 3.08 3.215 19.44 1 0 3 1
Hornet Sportabout 18.7 8 360 175 3.15 3.440 17.02 0 0 3 2
Valiant 18.1 6 225 105 2.76 3.460 20.22 1 0 3 1

Understanding the Core `dplyr` Syntax: Grouping and Summarizing

The standard, concise syntax used in `dplyr` for grouping and summarizing data leverages the pipe operator (`%>%`) to sequentially pass the data frame through different manipulation steps. This structure significantly improves code readability and maintainability.

The process begins by using the `group_by()` function, which specifies the categorical column(s) that will form the basis of the aggregation. Following the grouping, the `summarize()` function is called to calculate the desired statistics within each defined group. You define a new column name (e.g., `average_value`) and set it equal to the summary function being applied (e.g., `mean(variable)`).

It is important to remember that the functions `summarize()` and `summarise()` are treated

identically by the package, offering users flexibility based on their regional spelling preference. The basic syntax template ensures consistency across all data aggregation tasks:

data %>%
group_by(col_name) %>%
summarize(summary_name = summary_function)

Calculating Central Tendency Measures (Mean and Median)

A primary objective when summarizing data is to calculate measures of central tendency, which help describe the typical value of a dataset. We will demonstrate how to calculate both the arithmetic mean and the median for the miles per gallon (`mpg`) variable, grouped by the number of cylinders (`cyl`).

To calculate the mean MPG by cylinder count, we pipe `mtcars` into `group_by(cyl)` and then use `summarize()` with the `mean()` function. Note the inclusion of the argument `na.rm = TRUE`, which is a standard precaution to handle any potential missing data points, ensuring the calculation proceeds smoothly.

The median is calculated similarly, substituting `mean()` with `median()`. Because the median is less sensitive to extreme outliers than the mean, comparing both statistics is often essential for a full understanding of the group's distribution.

 # Find mean mpg by cylinder
mtcars %>%
group_by(cyl) %>%
summarize(mean_mpg = mean(mpg, na.rm = TRUE))

A tibble: 3 x 2
cyl mean_mpg

1 4 26.7
2 6 19.7
3 8 15.1

Find median mpg by cylinder
mtcars %>%
group_by(cyl) %>%
summarize(median_mpg = median(mpg, na.rm = TRUE))

A tibble: 3 x 2

cyl median_mpg

1 4 26
2 6 19.7
3 8 15.2

Assessing Data Dispersion (Standard Deviation and IQR)

Understanding how data points scatter around the central value is provided by measures of spread, or dispersion. Calculating these metrics for each group is vital for comparing the consistency and reliability of performance across categories. Key measures include the standard deviation (SD), the Interquartile Range (IQR), and the Median Absolute Deviation (MAD).

The standard deviation quantifies the typical distance of data points from the mean. The IQR, conversely, captures the range of the middle 50% of the data, offering a robust perspective on spread that is less affected by extreme values. The MAD serves a similar purpose, providing another reliable measure of variability.

We can calculate all three measures simultaneously within a single `summarize()` block. This efficient approach yields a comprehensive picture of MPG variability across 4, 6, and 8-cylinder vehicles, highlighting where performance consistency is highest or lowest.

 # Find sd, IQR, and mad by cylinder
mtcars %>%
group_by(cyl) %>%
summarize(sd_mpg = sd(mpg, na.rm = TRUE),
iqr_mpg = IQR(mpg, na.rm = TRUE),
mad_mpg = mad(mpg, na.rm = TRUE))

A tibble: 3 x 4
cyl sd_mpg iqr_mpg mad_mpg

1 4 4.51 7.60 6.52
2 6 1.45 2.35 1.93
3 8 2.56 1.85 1.56

Counting Observations and Unique Values per Group

Determining the sample size and checking the uniqueness of values within each group are foundational steps in data validation and analysis. `dplyr` facilitates this using two specialized

counting functions: `n()` for calculating the total count of rows within a group, and `n_distinct()` for counting how many unique values exist within a specified column within that group.

In the example below, we use `n()` to confirm the number of cars available for calculating the summary statistics for each cylinder type. We also use `n_distinct(mpg)` to see how many different MPG values are present in that specific group, providing insight into the variety of fuel efficiency ratings.

```
# Find row count and unique row count by cylinder
mtcars %>%
group_by(cyl) %>%
summarize(count_mpg = n(),
u_count_mpg = n_distinct(mpg))
```

```
# A tibble: 3 x 3
cyl count_mpg u_count_mpg

1 4 11 9
2 6 7 6
3 8 14 12
```

Determining Grouped Percentiles (Quantiles)

Quantiles, or percentiles, allow analysts to determine the positional standing of a value within its distribution. This is essential for understanding where performance thresholds lie--for example, identifying the fuel efficiency rating that the top 10% of 8-cylinder cars surpass.

To calculate a percentile, we use the `quantile()` function within `summarize()`. This function requires the `probs` argument, where a decimal value (e.g., `.9` for the 90th percentile) defines the desired cutoff point.

The following code calculates the 90th percentile of `mpg` for each cylinder group, clearly showing the upper limit of efficiency for 90% of the vehicles in each category.

```
# Find 90th percentile of mpg for each cylinder group
mtcars %>%
group_by(cyl) %>%
summarize(quant90 = quantile(mpg, probs = .9))
```

```
# A tibble: 3 x 2
cyl quant90
```

1 4 32.4
2 6 21.2
3 8 18.3

Conclusion and Advanced `dplyr` Applications

Mastery of the `group_by()` and `summarize()` functions provides a foundational skill set for effective data analysis in R. By utilizing this powerful `dplyr` methodology, data scientists can quickly and accurately transform unwieldy data frames into concise, actionable reports summarizing key statistics like mean, median, standard deviation, and quantiles across specified subgroups.

The true utility of `summarize()` lies in its versatility, allowing almost any R function that returns a single value to be applied after grouping. For those interested in exploring further capabilities, the official documentation and comprehensive visual cheat sheets for the dplyr package are excellent resources for advanced learning.

In practice, these aggregation functions are rarely used in isolation. They are frequently combined with other essential data manipulation verbs. Highly useful companion functions used alongside `group_by()` and `summarize()` include tools for filtering data frame rows and arranging rows in certain orders, enabling the construction of sophisticated and robust data processing workflows.