Keep Certain Columns in R (With Examples)

Authored by

Mohammed looti

October 28, 2025

Welcome to this comprehensive guide on managing data structures within the R programming environment. A fundamental requirement in nearly all data analysis projects is the ability to efficiently filter, select, and manipulate the variables (columns) contained within a data frame. Whether you are aiming to streamline your analysis by removing redundant fields or focusing exclusively on a core set of variables, mastering column selection is vital for effective data wrangling. This tutorial focuses on utilizing the versatile built-in R function, `subset()`, to selectively retain or discard columns based on specific criteria, providing clear examples for both approaches.

The initial process of loading and inspecting data often reveals that not all columns are necessary for the task at hand. Excess columns can clutter your workspace, increase memory consumption, and potentially slow down computationally intensive operations. Therefore, learning how to cleanly define a new data frame containing only the essential information is a hallmark of efficient R programming. We will demonstrate two powerful applications of the `subset()` function: first, specifying exactly which columns to keep; and second, specifying which columns to drop, relying on R to retain the rest.

Understanding Column Selection Fundamentals

The `subset()` function in R is primarily designed to return subsets of vectors, matrices, or data frames that meet specified conditions. While it is frequently used for row filtering, it possesses a highly useful `select` argument specifically dedicated to column manipulation. When using `subset()`, the resulting object will always be a new data frame, leaving the original data structure untouched--a crucial feature for maintaining data integrity during exploration and transformation. Understanding how to correctly pass column names or indices to the `select` argument is key to achieving precise control over your output.

In the context of column selection, the `select` argument expects a vector of column names or numerical indices. If you provide positive inputs (names or positive indices), R interprets this as the list of columns to **keep**. Conversely, if you prefix the vector of names or indices with a negative sign (`-`), R interprets this as the list of columns to **drop**, retaining all others. This dual functionality makes `subset()` a highly flexible tool for preliminary data preparation.

Before diving into the specific methods, let us establish a working sample data frame. This structure represents typical observational data, allowing us to clearly illustrate the effects of keeping or dropping various columns. We will use a small dataset tracking basketball statistics for different teams.

Preparing the Sample Data Frame in R

To effectively demonstrate column selection, we first need to construct a robust sample data frame. This example uses four variables: `team` (character), `points` (numeric), `assists` (numeric),

and `rebounds` (numeric). This variety of data types ensures that the column selection process applies universally regardless of the underlying data structure within the column.

The following code snippet initializes the data frame, named `df`, using the `data.frame()` constructor. We encourage you to run this setup code in your own R environment to follow along with the subsequent filtering steps. Once created, viewing the data frame allows us to confirm its structure and content before manipulation begins.

```
#create data frame
df <- data.frame(team=c('A', 'A', 'A', 'B', 'B', 'B'),
points=c(19, 14, 14, 29, 25, 30),
assists=c(4, 5, 5, 4, 12, 10),
rebounds=c(9, 7, 7, 6, 10, 11))

#view data frame
df
```

```
team points assists rebounds
1 A 19 4 9
2 A 14 5 7
3 A 14 5 7
4 B 29 4 6
5 B 25 12 10
6 B 30 10 11
```

As shown in the output, the data frame `df` contains six rows of observations and four distinct columns. Our goal in the following sections will be to isolate specific combinations of these four columns, demonstrating the precision afforded by the `subset()` function in R. We will first look at how to explicitly define the columns required for the resulting data frame.

Method 1: Explicitly Selecting Columns to Keep

When working with a data frame that has numerous columns, but you only need a small, defined set of variables, the most direct approach is to explicitly list the columns you wish to retain. This method enhances code readability and ensures that only the required data is carried forward, minimizing the risk of including sensitive or irrelevant information in downstream analysis. We achieve this by providing a vector of unquoted column names to the `select` argument within the `subset()` function.

To illustrate, suppose our analysis requires us only to investigate the relationship between the `team` and the number of `assists` recorded, ignoring `points` and `rebounds` entirely. The following

code demonstrates how to define `new_df` such that it contains only these two specified columns. Notice that the column names are passed inside the `c()` vector constructor, which is then assigned to the `select` parameter.

#only keep columns 'col1' and 'col2'
new_df = subset(df, select = c(col1, col2))

Applying this structure to our specific sample data frame yields the following result. The resulting data frame, `new_df`, is a precise representation of the data, stripped of the unnecessary variables, making it optimized for focused analysis. This process is often the first step in creating specialized datasets for visualizations or statistical modeling.

#keep 'team' and 'assists' columns
new_df = subset(df, select = c(team, assists))

#view new data frame
new_df

team assists
1 A 4
2 A 5
3 A 5
4 B 4
5 B 12
6 B 10

Method 2: Excluding Columns to Drop

Conversely, there are scenarios where your data frame contains hundreds of columns, but you only need to eliminate a small handful of variables, such as unique identifiers or metadata fields that complicate quantitative analysis. In these situations, listing every column to keep would be cumbersome and prone to error. The more efficient approach is to specify the columns you wish to **drop**.

This is achieved by applying a negative sign (`-`) immediately before the vector of column names within the `select` argument of the `subset()` function. The negative sign signals to R that the listed columns should be excluded from the resulting data frame, while all other columns are automatically retained. This technique drastically simplifies the code when dealing with wide datasets.

To demonstrate, we will define a new data frame that systematically drops the `team` and `assists` columns from the original `df`. Notice the placement of the negative operator outside the `c()` function. This subtle yet critical difference dictates the entire outcome of the column selection process.

#drop columns 'col3' and 'col4'
new_df = subset(df, select = -c(col3, col4))

When applied to our sample data, the resulting `new_df` retains only the `points` and `rebounds` columns, successfully excluding the two specified variables. This method is highly recommended for preliminary cleaning steps where extraneous columns are easily identified and need to be quickly discarded without rewriting the entire list of desired variables.

#drop 'team' and 'assists' columns
new_df = subset(df, select = -c(team, assists))

#view new data frame
new_df

points rebounds
1 19 9
2 14 7
3 14 7
4 29 6
5 25 10
6 30 11

Alternative Approach: Base R Indexing

While `subset()` offers a clean, function-based approach, it is important for any R user to be comfortable with the fundamental method of column selection: using square bracket indexing (). Base R indexing provides direct, powerful control and is often faster for very large datasets, although perhaps less intuitive than `subset()` for beginners. When selecting columns using brackets, the syntax is `df`. Since we are interested in keeping all rows, we leave the first argument blank or use a comma, and focus entirely on the second argument (columns).

To keep specific columns using Base R, you can pass a vector of column names (quoted strings) or a vector of column indices (numbers). For instance, to select the first and third columns (`team` and `assists`), you would use `df`. To select them by name, you would use `df`. This method requires all column names to be enclosed in quotation marks, distinguishing it from the `subset()`

approach where names are unquoted.

Similarly, dropping columns is accomplished by passing negative indices. To drop the second and fourth columns (`points` and `rebounds`), the command would be `df`. Using negative indices is the standard way to exclude columns in Base R indexing, offering a comparable mechanism to the negative sign used in the `subset()` function, reinforcing the flexibility of the R environment for data wrangling tasks.

Advanced Column Management with the dplyr Package

For users who frequently engage in complex data manipulation, the `dplyr` package, part of the widely adopted Tidyverse ecosystem, provides an even more intuitive and powerful framework for column selection. The `dplyr` package introduces the `select()` function, which is optimized for these tasks and supports helpful syntax extensions not available in Base R or `subset()`.

Using `dplyr::select()`, the commands for keeping and dropping columns become extremely clear. To keep `team` and `assists`, the command is simply `df %>% select(team, assists)`. The pipe operator (`%>%`) enhances flow and readability. To drop columns, the syntax is `df %>% select(-points, -rebounds)`, which allows you to list unwanted columns directly with the leading negative sign without needing the `c()` vector constructor, further simplifying the code structure.

Furthermore, `dplyr::select()` offers powerful helper functions, which are invaluable for managing large, systematically named datasets. These functions include `starts_with("prefix")`, `ends_with("suffix")`, `contains("text")`, and `matches("regex")`. For instance, if you wanted to select all columns whose names start with the letter 'p', you could use `df %>% select(starts_with("p"))`. These advanced selection tools make `dplyr` the preferred choice for complex data preparation workflows, offering robust, expressive, and highly readable syntax for column management.

Conclusion and Best Practices for Data Wrangling in R

Effective column selection is a cornerstone of modern data analysis in R. As demonstrated, the R language provides several highly effective methods for achieving this goal, each suited to slightly different contexts. The `subset()` function, with its clear `select` argument, provides a robust, built-in solution for both keeping and dropping columns via positive or negative specification.

While `subset()` is excellent for quick tasks and is available in Base R without loading additional packages, mastering Base R indexing (using) is essential for fundamental understanding and speed optimization. For long-term projects and complex, iterative data cleaning, adopting the `dplyr::select()` function is highly recommended due to its superior syntax, readability, and

access to powerful helper functions that automate selection based on name patterns. Ultimately, the best practice is to choose the method that ensures the highest level of clarity and efficiency for the specific data manipulation task at hand.

By systematically applying these techniques, you ensure that your analytical workflow is focused, memory-efficient, and easily reproducible. This ability to precisely manage your data frame columns is a vital step toward producing accurate and insightful statistical results.

Additional Resources

The following tutorials explain how to perform other common tasks in R: