How to Easily Reshape Data in R with pivot_longer()

Authored by

stats writer

November 21, 2025

The process of transforming data structures is fundamental to effective data analysis and visualization. In the R programming environment, specifically within the Tidyverse ecosystem, the **`pivot_longer()`** function serves as a crucial tool for this transformation. It specializes in reshaping data by converting multiple columns (which represent measured variables) into two columns: one for variable names and one for their corresponding values. This conversion is essential for moving data from a **wide format**, often dictated by easy data entry, into a **long format**, which is far more conducive to statistical modeling and efficient plotting using packages like `ggplot2`.

Understanding when and how to reshape data is key to achieving the goals of tidy data. Tidy datasets follow specific principles: each variable must be a column, each observation must be a row, and each type of observational unit must form a table. When data is structured such that measurements are spread across numerous columns (the wide format), it violates these principles, making iterative computations or group-wise summaries challenging. The primary objective of **`pivot_longer()`** is to resolve this structural conflict, enabling analysts to work with consistent and predictable data structures.

This article specifically addresses a common requirement: applying **`pivot_longer()`** to reshape **all** columns within a data frame simultaneously. While typically analysts select a subset of columns to pivot, certain scenarios--such as dealing with pure measurement matrices or simplifying data frames without identifying variables--necessitate treating every single column as a measured value. This is where the powerful selection helper **`everything()`**, provided by the `dplyr` selection context within the tidyr package, becomes indispensable.

The Architecture of Data Reshaping with pivot_longer()

The **`pivot_longer()`** function is the successor to older reshaping tools, offering a highly expressive and robust interface for converting data from wide to long. At its core, it requires the user to define two main components: which columns are to be pivoted (the variable columns), and how the resulting key-value pairs should be named. The elegance of **`pivot_longer()`** lies in its flexibility to handle complex column selections and naming conventions, allowing for sophisticated data preparation without excessive manual coding.

The essential arguments for the function include `data` (the data frame being transformed), `cols` (specifying the columns to pivot), `names_to` (defining the name of the new column holding the original column names, or keys), and `values_to` (defining the name of the new column holding the observations, or values). For instance, if you have columns named 'Jan', 'Feb', and 'Mar', pivoting them would result in a new column (specified by `names_to`) containing 'Jan', 'Feb', and 'Mar' as entries, and an adjacent column (specified by `values_to`) containing the corresponding numerical measurements. This structure is known as the **long format**, where each row represents a single

observation for a single variable.

A crucial consideration when using **`pivot_longer()`** is correctly identifying the columns that need pivoting versus those that serve as unique identifiers (ID variables). If a data frame contains ID columns (e.g., `Subject_ID` or `Country`) alongside measurement columns, it is standard practice to exclude the ID columns from the `cols` argument. However, in certain specialized datasets, particularly aggregated summaries or simplified input files, there might be no ID columns whatsoever, meaning every column represents a measurement that needs to be stacked vertically. It is precisely for these situations that selecting all columns becomes necessary, ensuring a complete collapse of the data structure.

Selecting All Columns Using everything()

When the requirement is to pivot every single variable within a **data frame**, manually listing dozens or hundreds of column names is inefficient and prone to error. Furthermore, if the input data schema changes (e.g., new measurement columns are added), the static code would break or require manual updates. To overcome this, the Tidyverse provides powerful selection helpers designed to work dynamically within functions like **`pivot_longer()`** and `select()`.

The core solution for pivoting all columns is utilizing the selection helper **`everything()`** within the `cols` argument. The function **`everything()`** is designed to select all variables that have not been previously selected or explicitly excluded in the current selection context. When used alone in the `cols` argument of **`pivot_longer()`**, it instructs the function to treat every column in the provided data frame as a measurement variable to be collapsed into the long format. This approach provides maximum flexibility and resilience against changes in the input data structure, making the code much more robust.

The syntax for this comprehensive pivot operation is surprisingly concise and highly effective. By omitting specific column names and simply relying on **`everything()`**, the operation becomes universally applicable to any data frame where all columns are measurements intended for vertical stacking. This technique simplifies complex data preparation pipelines, particularly those involving iterative processing of many similarly structured data files, where consistent, automated column selection is paramount for efficient data analysis.

Core Syntax for Full Dataset Pivoting

To execute the pivot operation across the entire dataset, the required structure involves calling the **`pivot_longer()`** function from the **`tidyr`** package and passing the **`everything()`** helper to the `cols` parameter. This syntax represents the most efficient way to transform a purely wide dataset into its corresponding long form.

The following example demonstrates the fundamental code required, assuming your input data frame is named `df`. It is assumed that the `tidyr` library has been loaded prior to execution:

library(tidyr)

df_long <- pivot_longer(df, cols = everything(), names_to = "name", values_to = "value")

In this syntax, the **`cols`** argument explicitly specifies which columns are targeted for pivoting. By assigning it the value **`everything()`**, we instruct R to include every column currently present in the `df` data frame in the transformation. We have also explicitly included the default settings for `names_to` ("name") and `values_to` ("value") for clarity, although these are often the default behavior if not specified. This efficient command immediately reshapes the entire structure, paving the way for easier statistical computation and visualization tasks.

Demonstration: Basketball Scores Example

To illustrate the practical application of pivoting all columns, let us consider a typical scenario involving sports performance data. Suppose we have collected data showing the points scored by various anonymous players across three different basketball games. Since we are focused only on the scores and not on identifying individual players (as there is no explicit ID column), the data frame is currently structured such that each row represents a distinct observation (a player's performance set) and each column represents a measured variable (the score in a specific game).

This dataset, while concise, is in a **wide format**, making it challenging to calculate summary statistics, such as the overall mean score across all games, without iterating through each column individually. The requirement is to restructure this data so that all scores are contained within a single column, ready for immediate statistical manipulation. Below is the code used to generate the sample data frame in R:

#create data frame
df <- data.frame(game1=c(20, 30, 33, 19, 22, 24),
game2=c(12, 15, 19, 19, 20, 14),
game3=c(22, 29, 18, 12, 10, 11))

#view data frame
df

game1 game2 game3
1 20 12 22
2 30 15 29
3 33 19 18

4 19 19 12
5 22 20 10
6 24 14 11

As evident from the output, the initial **data frame** has 6 rows and 3 columns. Each row implicitly represents a player's performance across the three games. Since all three columns--`game1`, `game2`, and `game3`--are measurements of the same type (points scored), and there are no ID columns to preserve, we intend to pivot all three simultaneously. This preparation is a mandatory step before performing analysis that treats all 18 individual scores as a single continuous variable.

Executing the Full Pivot Operation

The goal is to transform the 6x3 wide data frame into a 18x2 long format structure. This transformation involves taking the three columns and collapsing them down into a pair of columns: one containing the source game name (the key) and one containing the recorded score (the value). We achieve this by loading the **`tidyr package`** and applying **`pivot_longer()`** with the `cols = everything()` argument.

By using **`everything()`**, the function automatically identifies and targets `game1`, `game2`, and `game3`. The transformation process then systematically stacks the values of these columns into the specified `values_to` column, replicating the original row index information where necessary to maintain the observation integrity. This dynamic selection ensures that if this process were applied to similar datasets containing `game4` or `game5`, the code would handle them correctly without modification.

The following code block demonstrates the complete operation and the resulting restructured data frame:

library(tidyr)

#pivot all columns into long data frame
df_long <- pivot_longer(df, cols = everything())

#view long data frame
df_long

A tibble: 18 x 2
name value

1 game1 20
2 game2 12

3 game3 22
4 game1 30
5 game2 15
6 game3 29
7 game1 33
8 game2 19
9 game3 18
10 game1 19
11 game2 19
12 game3 12
13 game1 22
14 game2 20
15 game3 10
16 game1 24
17 game2 14
18 game3 11

Analyzing the Transformed Data Structure

Upon execution, the output `df_long` is a new data frame that strictly adheres to the principles of tidy data. The original column names, `game1`, `game2`, and `game3`, have been repurposed and placed sequentially into the first new column, which defaults to the name **"name"**. This "name" column now acts as the indicator variable, specifying which game the corresponding score originated from. Simultaneously, all the numerical scores (the actual measurements) are aggregated into the second new column, which defaults to the name **"value"**.

Observe that the row count has increased dramatically, from 6 rows in the original wide format to 18 rows in the long format (3 columns multiplied by 6 original rows). Each of the original 18 score observations now occupies its own row, paired with the appropriate game indicator in the **name** column. For example, the first row of the original data (20, 12, 22) has been transformed into three distinct rows in the long format: (game1, 20), (game2, 12), and (game3, 22). This structure is ideal for operations like calculating the overall mean score for the entire dataset or filtering observations based on specific game criteria.

The successful use of **`cols = everything()`** confirms that every single variable in the original input was included in the pivot operation. This makes the resulting data frame fully prepared for subsequent Tidyverse operations, such as grouping, filtering, and summarizing, which rely heavily on variables being stored vertically rather than horizontally. The resulting structure facilitates powerful statistical methods and simplifies the overall data analysis workflow.

Refinements and Alternatives to everything()

While `cols = everything()` is perfect when every column needs to be pivoted, practical data often includes identifying variables (like `ID` or `Date`) that must be preserved as identifiers in the long format. In such cases, using `everything()` alone would incorrectly pivot the ID columns. The correct methodology involves selecting all columns **except** the ID columns, which can be achieved using a negative selection approach.

For example, if our basketball data had an explicit `Player_ID` column, we would modify the syntax to exclude it: `cols = -Player_ID`. This tells `pivot_longer()` to keep the `Player_ID` column untouched and only pivot all remaining columns (which are automatically selected using the minus sign combined with the column name). This negative selection achieves the same result as listing all measurement columns, but maintains the dynamic robustness offered by selection helpers.

Additionally, `pivot_longer()` offers arguments like `names_prefix` and `names_pattern` which are invaluable for cleaning up the resulting **name** column. In our example, the names 'game1', 'game2', and 'game3' share the prefix 'game'. We could use `names_prefix = "game"` to automatically strip 'game' from the resulting **name** column, leaving only the measurement number (1, 2, 3). This refinement further optimizes the data for final presentation and clarity, maintaining the high standards of tidy data.

Conclusion: The Efficiency of Dynamic Selection

The ability to use `pivot_longer()` effectively, especially on large datasets where manual column selection is impractical, is a cornerstone of efficient data wrangling in R. The `cols = everything()` argument provides an elegant, dynamic, and robust solution for transforming wide data matrices entirely into the necessary long format. This function is particularly valuable when dealing with raw measurement logs or highly aggregated tables that lack traditional identifying variables, ensuring that every observation is properly verticalized for immediate statistical consumption.

Mastering this dynamic selection technique ensures that your data preparation code is resilient and scalable. Whether you are dealing with three games of basketball scores or hundreds of quarterly financial metrics, the principle remains the same: transforming data structure from a **wide format** to a **long format** is often a prerequisite for advanced data analysis and high-quality visualization. By leveraging `everything()`, analysts can dramatically simplify their reshaping code while guaranteeing comprehensive coverage of the input data.

For those seeking deeper understanding and more advanced usage scenarios involving regular expressions or complex selection patterns, the complete documentation for the `pivot_longer()` function is an essential reference. Continuous learning of these core tidyr package functions will

significantly enhance productivity within the R programming environment.

The **pivot_longer()** function from the tidyr package in R can be used to pivot a data frame from a **wide format** to a **long format**.

If you'd like to use this function to pivot all of the columns in the data frame into a long format, you can use the following syntax:

library(tidyr)

df_long <- pivot_longer(df, cols = everything())

Note that the `cols` argument specifies which columns to pivot and `everything()` specifies that we want to pivot every column available in the data frame.

The following expanded example shows how to use this function in practice to prepare data for data analysis.

Example: Using pivot_longer() on All Columns in R

Suppose we have the following data frame in R that shows the number of points scored by various basketball players during three different games:

#create data frame
df <- data.frame(game1=c(20, 30, 33, 19, 22, 24),
game2=c(12, 15, 19, 19, 20, 14),
game3=c(22, 29, 18, 12, 10, 11))

#view data frame
df

game1 game2 game3
1 20 12 22
2 30 15 29
3 33 19 18
4 19 19 12
5 22 20 10
6 24 14 11

The data frame is currently in a **wide format**, with each measurement spread across its own column.

However, suppose we'd like to pivot the data frame to a long format by pivoting all three columns (since there are no identifier columns to preserve).

We can use the following syntax to do so, utilizing `everything()`:

library(tidyr)

#pivot all columns into long data frame
df_long <- pivot_longer(df, cols = everything())

#view long data frame
df_long

A tibble: 18 x 2
name value

1 game1 20
2 game2 12
3 game3 22
4 game1 30
5 game2 15
6 game3 29
7 game1 33
8 game2 19
9 game3 18
10 game1 19
11 game2 19
12 game3 12
13 game1 22
14 game2 20
15 game3 10
16 game1 24
17 game2 14
18 game3 11

Notice that the original column names `game1`, `game2` and `game3` are now used as values in a new column called **"name"**, and the values from these original columns are placed into one new column called **"value."**

The final result is a highly structured **long data frame**, ready for analysis.

Note: You can find the complete documentation for the `pivot_longer()` function in the official tidyr documentation.