How to Extract the First Row from Each Group in Pandas GroupBy

Authored by

stats writer

December 1, 2025

The process of data aggregation and analysis often requires filtering specific records once the data has been logically partitioned. When working with the pandas library in Python, grouping data is a fundamental step, typically achieved using the powerful **groupby()** function. A common and critically important requirement in data science workflows is isolating the very first observation within each of these newly formed groups. This task, while seemingly simple, necessitates a precise understanding of how the resulting GroupBy object behaves and which methods are optimized for retrieval rather than aggregation. Efficiently retrieving the first element--whether it represents the earliest time entry, the first recorded transaction, or simply the first row appearing in the group after sorting--is essential for summary reporting and subsequent analytical steps, preventing the unnecessary loading or processing of redundant data.

While aggregation methods like `mean()` or `sum()` are straightforward to apply to a GroupBy object, selecting a specific positional row (like the 0-th row) requires a specialized approach. Traditional methods like `.first()` or `.head(1)` often achieve the desired result, but they may carry subtle behavioral differences or performance drawbacks compared to the most direct solution. For instance, `.first()` strictly returns non-NaN values for the first entry, which might not align with the literal definition of the "first row" based on the original pandas DataFrame order. Therefore, mastering the appropriate technique--the **nth()** method--is paramount for maintaining code clarity and computational efficiency, especially when dealing with large datasets where performance optimization is a key concern.

In order to get the first row of each group in pandas group-by, the **groupby()** function can be used to group the data and the specialized **nth()** function can be used to select the first row of each group. The **groupby()** function will transform the pandas DataFrame into a GroupBy object, and the **nth()** function will precisely select the element at the specified positional index (0 for the first row) within each group, making it quick and easy to access the first relevant observation. This combination is the most idiomatic and reliable way to solve this common data manipulation challenge.

Understanding the GroupBy Operation in Pandas

The **groupby()** method is the cornerstone of split-apply-combine strategies in pandas. When applied to a pandas DataFrame, it does not immediately return a result; instead, it returns a **GroupBy object**. This object is a powerful, lazy representation of the grouped data, meaning that the actual computation (the "apply" step) is deferred until a specific method is called upon it. The primary function of the GroupBy object is to partition the rows of the original pandas DataFrame based on unique values in the specified grouping column(s), preparing them for individual processing.

This deferred execution model is highly efficient, as it avoids generating unnecessary intermediate

data structures. When we talk about finding the "first row," we are asking the GroupBy object to apply a selection operation to each of the internal partitions it has created. If we group a dataset by a categorical column like 'Region', the GroupBy object internally holds pointers to all rows belonging to 'North', 'South', 'East', and 'West'. The subsequent method called must be capable of extracting the row corresponding to the 0-th position relative to the starting point of that specific partition.

It is critical to note that the definition of the "first row" is dependent on the order of the original pandas DataFrame before the grouping occurred, unless the data is explicitly sorted prior to the **groupby()** operation. If the data is not pre-sorted by a timestamp or some other ordinal column, the "first row" is simply the one that appeared first chronologically within that group in the source dataset. For analyses that depend on specific ordering, such as time series data, ensuring the pandas DataFrame is correctly sorted using `df.sort_values()` before the grouping call is a necessary prerequisite to guarantee meaningful results.

The Primary Solution: Utilizing the nth() Method

The most robust and expressive way to select a row based on its position within a group is by utilizing the **nth()** method. This function allows users to retrieve the row located at a specific integer position, or a list of positions, within each group defined by the **groupby()** function. To select the very first row, we simply pass the integer 0 as the argument to **nth()**, corresponding to the zero-based index system used throughout Python and pandas.

You can use the following basic syntax to get the first row of each group in a pandas DataFrame:

df.groupby('column_name').nth(0)

The **nth()** method provides flexibility that other selection methods often lack. Unlike `.head(1)`, which sometimes behaves slightly differently depending on the data structure, **nth()** is explicitly designed for positional selection. Moreover, it handles missing data (NaN values) more predictably than methods like `.first()`, which might skip rows containing NaN values in some columns when determining the "first" non-null entry, potentially deviating from the strictly positional requirement.

Understanding that `0` represents the first element is key. If you needed the second element, you would pass `1`, and so forth. This method preserves all columns of the original pandas DataFrame for the selected row, meaning the resulting pandas DataFrame will have the same structure but contain only one row for every unique group. This makes the output immediately usable for further analysis or joining operations without requiring complex reshaping.

The following practical example demonstrates how to implement this syntax effectively using sample data.

Example: Get First Row of Each Group in Pandas

To illustrate the functionality of `groupby().nth(0)`, let us establish a sample pandas DataFrame representing team performance data, containing columns for 'team', 'points', and 'assists'. We will aim to extract the performance metrics corresponding to the first recorded entry for each unique team.

import pandas as pd

#create DataFrame
df = pd.DataFrame({'team': ,
'points': ,
'assists': })

#view DataFrame
df

team points assists
0 A 18 5
1 A 22 19
2 B 19 14
3 B 14 8
4 B 14 9
5 C 11 12
6 C 20 13
7 C 29 8

The resulting DataFrame contains eight records, with three distinct teams (A, B, C). Our objective is to retrieve rows 0, 2, and 5, as these represent the initial occurrences of Team A, Team B, and Team C, respectively, based on the default index order of the data.

We achieve this by applying the **groupby()** method on the 'team' column, followed immediately by the **nth(0)** selection. The grouping operation handles the partitioning, and `nth(0)` extracts the positional first element from each partition.

#get first row for each team
df.groupby('team').nth(0)

points assists
team
A 18 5

B 19 14
C 11 12

Managing the Output Index: Using as_index=False

A crucial aspect of working with the **groupby()** method is understanding how it handles the resulting index. By default, when a grouping operation is performed, the grouping column--in this case, 'team'--is promoted to become the new index of the resulting pandas DataFrame. While this is often desirable for data summary, it means the original positional index values (0, 2, 5 in our example) are discarded.

For many subsequent operations, such as merging the results back into a larger dataset or simply needing to reference the original row identifiers, it is necessary to prevent the grouping column from becoming the index. This is achieved by passing the optional parameter **as_index=False** within the **groupby()** function call. When `as_index=False` is utilized, the grouping column remains a regular data column, and the resulting pandas DataFrame maintains a default numerical index derived from the original positional indices of the selected rows.

We can specify **as_index=False** in conjunction with **nth(0)** to retain the original index values, effectively showing which row from the source data corresponded to the first entry of each team:

#get first row for each team, keep original index values
df.groupby('team', as_index=False).nth(0)

team points assists
0 A 18 5
2 B 19 14
5 C 11 12

Notice how the output now includes the 'team' column as a data column, and the row indices (0, 2, 5) match the original pandas DataFrame's indices for the selected entries. This output format is usually preferred when the goal is data filtering rather than pure aggregation reporting.

Expanding Selection: Retrieving Multiple Rows Per Group

The utility of the **nth()** method extends beyond simply retrieving the first row. It is highly versatile and allows for the selection of multiple rows simultaneously based on their positional index within the group. This is achieved by passing a list or tuple of integers to the **nth()** function, rather than a single integer.

When multiple positional indices are provided, **nth()** performs the selection for each index across all groups and returns a combined pandas DataFrame containing all selected rows. For instance, if the requirement is to obtain both the first and the second entry for every team, we would pass the tuple `(0, 1)` to the method. This operation efficiently scales the positional filtering across the entire dataset.

For example, the following code demonstrates how to get the first two rows (at positions 0 and 1) for each group, while still utilizing `as_index=False` to preserve clarity in the output structure:

#get first two rows for each team, keep original index values
df.groupby('team', as_index=False).nth((0, 1))

team points assists
0 A 18 5
1 A 22 19
2 B 19 14
3 B 14 8
5 C 11 12
6 C 20 13

It is important to observe that Team B only contributes two rows (index 2 and 3), while Team C contributes two rows (index 5 and 6), and Team A contributes two rows (index 0 and 1). Note that the original index values are maintained, providing a clear reference to the source data. This mechanism ensures that complex filtering criteria based on positional order can be executed with a single, highly optimized command.

Alternatives to nth(): Using head() and first()

While `groupby().nth(0)` is the definitive method for positional selection, pandas offers other functions that can achieve similar results, though with distinct operational nuances. The two most common alternatives are `.head(1)` and `.first()`.

The `.head(n)` method, when applied to a GroupBy object as `.head(1)`, also selects the first N rows from each group. For retrieving the first row, `.head(1)` functions identically to `.nth(0)` in terms of the resulting rows selected based on the original pandas DataFrame order. However, `.head()` is generally less flexible for selecting arbitrary, non-sequential positions (e.g., selecting only the 0th and 5th row) compared to the list functionality provided by **nth()**. For the specific task of retrieving only the first row, either method is technically viable, but `nth(0)` explicitly communicates the intent of positional zero-based selection.

Another alternative is the `.first()` method. This function is designed to return the first non-null

value within each group for every column. If a column's first row contains a missing value (NaN), `.first()` will search down the group until it finds a non-null entry for that specific column. This behavior means that `.first()` might return data aggregated from **different** rows within the group, potentially pulling 'points' from row 0 but 'assists' from row 1, if row 0 had a null 'assists' value. Therefore, `.first()` should only be used if the goal is to find the earliest non-null value per column, not if the goal is to retrieve the entire row corresponding to the original positional first entry.

Ensuring Order: Pre-Sorting Data for Meaningful Results

A critical consideration when performing any positional selection (like `.nth(0)`) after grouping is the underlying order of the source pandas DataFrame. If the data is inherently unordered, the definition of the "first row" is purely arbitrary, based on insertion order. In most analytical scenarios, the definition of "first" must be tied to a meaningful metric, such as time, score, or ID. If this order is not guaranteed, the grouping operation should be preceded by a call to `.sort_values()`.

To ensure that the first row of each group genuinely represents the minimum value of a specific column, or the earliest timestamp, the process should follow this structure:

Sort the pandas DataFrame by the desired metric (e.g., timestamp) within the grouping column.
Apply **groupby()** on the categorical column.
Apply **nth(0)** to select the row that is now positioned first after the sorting step.

For example, if we wanted the entry with the minimum 'points' for each team, we would sort the data ascendingly by 'points' before grouping. This careful pre-processing step elevates the result from arbitrary selection to meaningful data filtering.

nth() Documentation and Further Resources

For users looking to deepen their understanding of this powerful filtering technique, the official documentation for the **nth()** function provides exhaustive detail on all available parameters and edge case behaviors. It is an indispensable resource for mastering complex data manipulation tasks in pandas.

Note: You can find the complete documentation for the **nth()** function at the pandas official website.

Conclusion: Efficiency and Clarity in Grouped Selection

Selecting the first row of each group in a pandas DataFrame is a frequent requirement in analytical programming. By leveraging the combined power of the **groupby()** method and the specialized

`nth(0)` function, developers can achieve this goal with maximum efficiency and code clarity. The ability to control the resulting index structure using `as_index=False`, and the inherent flexibility of `nth()` to select multiple positions, solidify this approach as the industry standard for positional group filtering in pandas. Mastering this technique ensures reliable data preparation for large-scale data science projects.