How to Calculate the Mean with Groupby Including NaN Values in Pandas

Authored by

stats writer

November 20, 2025

When performing data aggregation using the Pandas library, a common requirement is calculating statistical measures like the mean across grouped data. However, the default behavior of Pandas is to automatically exclude **Not a Number (NaN)** values from these calculations, potentially skewing results if the presence of missing data itself needs to be reflected in the aggregated output. To precisely control this behavior and ensure that **NaN** values are not ignored when calculating the mean, you must explicitly set the parameter `skipna=False` within the aggregation function.

This technique involves utilizing the powerful groupby method in conjunction with a specialized aggregation function, specifically applying the customized mean calculation to override the default data cleaning process. For instance, the expression `df.groupby('column_name').mean(skipna=False)` is a straightforward way to achieve this when grouping by a single column and applying the calculation directly. However, for more robust scenarios involving multiple columns or complex aggregations, using the agg method provides superior flexibility, which we will demonstrate as the most effective solution.

By forcing the calculation engine to include **NaN** values, the resulting mean for any group containing at least one missing entry will itself be returned as **NaN**. This serves as a vital indicator that the group's summary statistic is compromised due to incomplete records, maintaining high data integrity and transparency, and providing an immediate visual cue regarding data quality issues within the aggregated results.

Understanding the Default Aggregation Behavior in Pandas

When analysts utilize the Pandas **groupby()** function to segment a **DataFrame** based on categorical data and compute the mean of a numerical column, the default setting is designed for efficiency and convenience: it automatically skips all instances of **Not a Number (NaN)** missing data. This is the standard procedure for most statistical packages, ensuring that the mean calculated reflects only the valid, observed data points available within the group.

This automatic exclusion, driven by the implicit use of `skipna=True`, means that if a group has 10 records and one is **NaN**, the mean is calculated using the sum of the remaining 9 records divided by 9. While mathematically sound for calculating the mean of observed data, this behavior can be undesirable when the existence of missing data itself must be factored into the aggregation result, signifying a flawed or incomplete group summary.

If your analytical requirement is to explicitly register the presence of missing values within the aggregated mean--meaning if any element in a group is **NaN**, the resulting group mean should also be **NaN**--we must actively override this default behavior. This strict approach is essential in scenarios where the completeness of the source data set is paramount to the validity of the resulting summary statistic, such as regulatory reporting or quality control monitoring.

Implementing the Solution using `agg()` and `skipna=False`

To achieve this strict adherence to data presence, we employ a combination of the **groupby** operation and the highly flexible **agg()** method. The `agg()` method allows us to pass a custom aggregation logic, typically using a Python `lambda` function, which explicitly defines the behavior of `skipna`.

df.groupby('team').agg({'points': lambda x: x.mean(skipna=False)})

This specific construction first partitions the **DataFrame** rows based on unique values in the **team** column. Subsequently, it computes the mean value for the **points** column within each partition. Crucially, the `lambda` function ensures that the calculation considers all elements, meaning the presence of a single **NaN** value forces the resulting group mean to be **NaN** itself. This methodology ensures that the aggregated output immediately flags incomplete data segments, providing a transparent measure of data validity.

The subsequent sections will provide a comprehensive demonstration of this method, illustrating how this syntax operates on a sample data set and contrasting its output with the results obtained from the default aggregation settings. Understanding this distinction is fundamental to accurate and context-aware data analysis using **Pandas**.

Practical Example: Constructing the Test DataFrame

To effectively illustrate the difference between the default **groupby** behavior and the strictly controlled aggregation using `skipna=False`, we will first construct a small sample **DataFrame**. This dataset contains hypothetical performance statistics for various basketball players, intentionally including a missing value to test our aggregation logic.

The data frame creation requires importing **Pandas** and **NumPy**, as **NumPy**'s representation of missing data (`np.nan`) is the standard method for representing nulls in **Pandas** numerical series. We define two teams, 'A' and 'B', ensuring Team A contains one record where the 'points' score is explicitly set to `np.nan`, simulating lost or unrecorded data.

import pandas as pd
import numpy as np

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

View DataFrame

```
print(df)
```

```
team points
0 A 15.0
1 A NaN
2 A 24.0
3 A 25.0
4 A 20.0
5 B 35.0
6 B 34.0
7 B 19.0
8 B 14.0
9 B 12.0
```

The resulting DataFrame structure clearly delineates the data imperfection for Team A. Our goal is to calculate the average points scored per team, but we need a method that recognizes that the missing data point for Team A renders its true average points score undeterminable, rather than calculating a mean based solely on the observed four scores.

Observing the Default Grouped Mean Calculation

Before implementing our solution, we must first execute the standard **groupby** mean calculation to observe **Pandas'** native handling of missing data. In this scenario, we group the data by the **team** column and calculate the mean of the **points** column directly using the `.mean()` method, which assumes `skipna=True`.

```
# Calculate mean of points, grouped by team (Default Behavior: skipna=True)
df.groupby('team').mean()
```

```
team
A 21.0
B 22.8
Name: points, dtype: float64
```

The output reveals that the mean value of **points** for Team A is returned as 21.0. This calculation derived the mean by summing the four valid point values (84 total) and dividing by the number of valid records (4). Notice that the presence of the **NaN** entry was completely bypassed, resulting in a clean numerical average, which might falsely suggest data completeness.

This result confirms that the default mechanism in **Pandas** prioritizes delivering a statistical

summary based on available data. While this behavior is convenient for many descriptive tasks, it fundamentally ignores the data quality issue represented by the missing value, requiring the implementation of the explicit `skipna=False` control mechanism to force an indication of data incompleteness.

Applying the Controlled Aggregation: `skipna=False`

Now we apply the specialized syntax using `.agg()` and the `lambda` function containing `skipna=False`. This instructs **Pandas** to treat any **NaN** value within the grouped segment as invalidating the entire mean calculation for that group.

 # Calculate mean points value grouped by team and don't ignore NaNs
df.groupby('team').agg({'points': lambda x: x.mean(skipna=False)})

points
team
A NaN
B 22.8

The output of this controlled aggregation demonstrates the desired outcome. For Team B, which had complete data, the mean remains 22.8. Crucially, for Team A, a **NaN** value is returned as the mean points value this time. This is because the argument **skipna=False** explicitly told Pandas not to ignore the **NaN** value when calculating the mean, resulting in an aggregated metric that reflects the uncertainty caused by the missing data point.

This technique is vital for maintaining transparency in data processing pipelines. It ensures that downstream analytical steps are immediately alerted to groups where the summary statistic is incomplete, preventing the use of potentially misleading averages derived from incomplete subsets of the original data. The use of the agg method combined with `lambda` provides the granular control necessary for complex data integrity requirements.

Summary of Techniques for Aggregation Control

Achieving precise control over grouped statistical calculations in **Pandas** requires mastery of the aggregation function parameters. The primary takeaway is the explicit use of `skipna=False` to override the default statistical behavior. This methodology ensures data transparency, signaling missing data directly in the output metric.

To summarize the effective methods for controlling **NaN** handling during mean calculation based on the required level of control:

Simple Direct Calculation (Default): Use `df.groupby().mean()`. This implicitly uses `skipna=True` and ignores all **NaN** values, resulting in a mean based only on observed data.

Direct Calculation (Controlled): Use `df.groupby().mean(skipna=False)`. This is the simplest way to enforce the **NaN** reflection rule if only the mean is needed for a single column.

Flexible Aggregation (Controlled): Use `df.agg({'col': lambda x: x.mean(skipna=False)})`. This syntax, demonstrated in the primary example, offers the best flexibility for applying controlled calculations alongside other potentially non-standard metrics within the same operation.

By consciously selecting the appropriate syntax and parameter setting, data scientists can ensure that their aggregated statistics accurately reflect the quality and completeness of the underlying data, moving beyond simple descriptive summaries to incorporate critical data integrity checks into their analytical workflows.