How to Easily Resample Time Series Data with Pandas groupby()

Authored by

stats writer

November 20, 2025

The `groupby()` and `resample()` methods only work with **Pandas DataFrame** objects, not with Pandas Series. If you want to use these methods, you need to convert the Series to a DataFrame first. Then you can use the `groupby()` and `resample()` methods on the DataFrame to get the desired output.

Understanding Time Series Resampling in Pandas

Time series data analysis frequently necessitates altering the frequency of observations. This critical procedure, referred to as **resampling**, enables analysts to aggregate data over new, uniform temporal intervals. Typical use cases involve downsampling, such as condensing minute-by-minute sensor readings into hourly summaries, or upsampling, which involves interpolating data to a finer granularity. In the robust **Pandas** library, resampling is a foundational operation for preparing data for statistical modeling, visualization, and reporting.

The complexity increases when the dataset is not homogeneous but contains distinct categorical groups--for example, tracking sales data across multiple regions, different product lines, or various retail locations. In such scenarios, a simple global resampling operation would incorrectly combine data from disparate groups. The challenge is to maintain the integrity of these categorical groups while accurately applying the temporal aggregation.

To solve this, data scientists must combine the strengths of the time-aware `resample()` method with the powerful splitting capabilities of the **groupby()** operator. This combination ensures that the resampling logic--defining the new time boundaries--is applied independently and correctly to each unique group defined by non-time columns, allowing for precise, group-specific temporal analysis.

Integrating Categorical Grouping with Time Frequency using pd.Grouper

While one might intuitively attempt to chain `groupby()` and `resample()` directly, this approach often fails to yield the desired result because `resample()` is primarily designed to operate directly on the DataFrame's time index. To effectively define groups based on both a categorical column and a desired time frequency, **Pandas** provides a specialized object: the **Grouper**.

The `pd.Grouper()` object acts as a powerful grouping key that can be passed directly into the standard `groupby()` function. It takes arguments specific to time series, most notably `freq` (the desired time interval) and optionally `key` (the column containing the datetime index, if it is not the main DataFrame index). By utilizing `pd.Grouper`, we instruct **Pandas** to establish temporal bins based on the frequency before proceeding with any other grouping criteria.

The core syntax involves supplying a list of grouping elements to `groupby()`: one or more categorical column names, and at least one `pd.Grouper` object specifying the time frequency. This

ensures a multi-level grouping structure where temporal aggregation is correctly nested within the categorical separation.

Essential Syntax for Combined Resampling

To **resample** time series data across multiple categorical groups, the implementation involves defining the specific grouping keys passed into the `groupby()` method. This approach elegantly handles the concurrent grouping by both time and category.

If you'd like to resample a time series in **Pandas** while applying the **groupby** operator, the following basic syntax outlines the creation of the grouper object and the subsequent aggregation steps:

grouper = df.groupby()

result = grouper.sum().unstack('store').fillna(0)

This sequence of operations first defines the grouping mechanism: rows are grouped by week (`freq='W'`) and then sub-grouped by the value in the **store** column. After grouping, the `sum()` method calculates the total sales for each unique combination of week and store. The subsequent steps--`unstack('store')` and `fillna(0)`--are standard practices used to pivot the resulting multi-index Series back into a wide **DataFrame** format, where stores become columns and any missing weekly data points are replaced with zero for clarity.

Selecting Appropriate Time Frequency Aliases

The integrity of the resampling process hinges on the correct specification of the frequency alias (`freq`). **Pandas** utilizes a system of offset aliases to define the precise time interval for aggregation. These aliases ensure that the new time index boundaries are established accurately, whether calculating daily totals, monthly averages, or quarterly statistics.

We can **resample** the time series data by a variety of standard time periods. The most common frequency aliases used with `pd.Grouper` include:

S: Seconds
Min: Minutes
H: Hours
D: Calendar Day
W: Week (Weekly frequency)
M: Month (Month end frequency)
Q: Quarter (Quarter end frequency)

A: Year (Year end frequency)

It is crucial to be aware that many aliases allow for adjustments to specify the anchor point, such as 'W-MON' for weekly resampling anchored to Monday, or 'MS' for month start instead of month end ('M'). Analysts must choose the alias that accurately reflects the temporal reporting requirements of their project.

Example: Preparing the Initial Daily Sales Data

To demonstrate the functionality of combined grouping and resampling, we will work with a sample dataset tracking daily sales. This initial step involves setting up a **Pandas DataFrame** that uses a datetime index, which is a prerequisite for any time series operation in Pandas.

The following code block creates a sample **DataFrame** where the index spans multiple days, and we track both the 'sales' metric and the 'store' identifier:

import pandas as pd

#create DataFrame
df = pd.DataFrame({'sales': ,
'store': },
index=pd.date_range('2023-01-06', '2023-01-16', freq='d'))

#view DataFrame
print(df)

sales store
2023-01-06 13 A
2023-01-07 14 A
2023-01-08 17 A
2023-01-09 17 A
2023-01-10 16 A
2023-01-11 22 B
2023-01-12 28 B
2023-01-13 10 B
2023-01-14 17 B
2023-01-15 10 B
2023-01-16 11 B

This dataset provides daily sales figures spanning eleven days, which naturally fall into three distinct weekly periods (given the standard Sunday week-ending convention often used by 'W'

frequency in Pandas), allowing us to clearly observe the effects of weekly **resampling**.

Executing the Grouped Weekly Resampling

Our primary goal is to group the rows first by **store**, and then apply a weekly aggregation, calculating the sum of the 'sales' column for each unique store-week combination. This requires the precise implementation of the **Grouper** object within the `groupby()` call.

We apply the previously defined syntax to perform the grouping, aggregation, and restructuring of the results. The output is a clean DataFrame showing weekly totals per store:

#group by store and resample time series by week
grouper = df.groupby()

#calculate sum of sales each week by store
result = grouper.sum().unstack('store').fillna(0)

#view results
print(result)

store A B
2023-01-08 14.0 0.0
2023-01-15 16.5 17.0
2023-01-22 0.0 11.0

This output successfully transforms the daily records into a weekly summary. The index now represents the week-ending date, and the sales figures reflect the total accumulated sales for that specific store within that time window.

Interpreting the Aggregated Output

The resulting data structure provides a clear, high-level view of performance across categories over time. Each row corresponds to a defined week, and the columns detail the aggregated metric (in this case, the sum of sales) for each store.

From the output shown above, we can draw specific conclusions about the aggregated weekly performance:

For the week ending 2023-01-08, the total sales at store A reached **14.0**. Since Store B's data points only started later, its recorded sales for this period are **0.0**.
In the subsequent period ending 2023-01-15, Store A recorded total sales of **16.5**, while Store B

accumulated **17.0** in sales.
The final period, ending 2023-01-22, captures the remainder of the data. Store B contributed **11.0** to its weekly total, whereas Store A had no data falling into this time window, resulting in **0.0** sales.

This output format is highly useful for comparative analysis, allowing business analysts to quickly compare weekly performance trends between different stores.

Flexibility in Aggregation Metrics

While the example above calculated the sum of sales, the combined **groupby** and resampling operation is not limited to summation. The true power of this methodology lies in the ability to select virtually any standard aggregation function provided by **Pandas**.

To change the calculation, simply replace `sum()` in the code with a different aggregation metric. For instance, to calculate the weekly average sales instead of the total sum, replace `.sum()` with `.mean()`. Other useful functions include:

count(): Determines the number of daily records that fell into each weekly bin.
median(): Calculates the 50th percentile sales value.
std() or **var()**: Measures the statistical dispersion or volatility of sales within the period.

Choosing the right aggregation function allows the analyst to extract diverse insights from the underlying **time series** data, tailoring the output to specific analytical requirements, such as performance consistency or outlier identification.

Further Reading on Pandas Operations

Mastering complex data manipulation often involves understanding how to chain multiple Pandas operations effectively. The `groupby()` and `resample()` combination is just one powerful method available for structured data analysis.

The following tutorials explain how to perform other common operations required for robust data analysis in Python: