How to Easily Count Unique Values in Pandas DataFrames with GroupBy

Authored by

stats writer

December 3, 2025

The ability to efficiently summarize and analyze large datasets is fundamental to modern data science. One of the most powerful tools available in the Python ecosystem for this task is the Pandas library, specifically its `groupby()` method. Utilizing the **Pandas GroupBy** mechanism, we can perform sophisticated aggregation functions, such as counting the number of **unique values** within specific columns of a Pandas DataFrame.

This process involves conceptually splitting the **DataFrame** based on defined criteria (the groups), applying a statistical function (counting unique elements), and then combining the results back into a meaningful structure. This methodology is incredibly useful for quickly determining the level of cardinality within subsets of your data, providing crucial insights into variable distribution and potentially identifying anomalies or redundancies. Mastering the combination of `groupby()` and the `nunique()` aggregation function is essential for effective data manipulation and subsequent **data analysis**.

Understanding the Power of Pandas GroupBy

The `groupby()` operation in Pandas is inspired by the SPLIT-APPLY-COMBINE strategy, a critical paradigm in data processing. When you invoke `groupby()`, you are essentially partitioning your **DataFrame** into distinct sub-DataFrames based on the unique values found in the specified grouping column(s). This initial split ensures that subsequent operations are applied independently to each subset of data, allowing for highly specific and targeted aggregations.

Once the data is split, the APPLY phase takes over. This is where we introduce the function designed to count unique elements: `nunique()`. Unlike standard count functions, `nunique()` specifically ignores duplicates and provides a count of only the distinct entries in the targeted column for the current group. This capability is paramount when exploring data diversity, such as determining how many different products were sold by each region or how many unique scores were achieved by each player.

The final phase, COMBINE, consolidates the results from all the independent groups back into a single Series or **DataFrame**, typically structured with the grouping key(s) forming the index. This streamlined approach makes complex hierarchical counts straightforward to execute and interpret, providing a clear, aggregated view of the data distribution which is often difficult to achieve using simple filtering or looping mechanisms. Furthermore, the efficiency of the vectorized operations within Pandas ensures that this grouping and counting process remains fast, even when dealing with millions of rows of data.

The Core Syntax: Counting Unique Values with nunique()

To count the number of unique values by group in a Pandas DataFrame, we must chain three

distinct operations: the grouping, the selection of the column to count, and the application of the unique counting method. The basic structure is highly intuitive and follows the logical flow of asking a question about the data: "For each category in X, how many unique items are there in Y?"

The syntax below illustrates this relationship. We start by calling the `groupby()` method on the **DataFrame** (`df`), specifying the `group_column` that defines the boundaries of our subgroups. We then use standard bracket notation to select the `count_column`--the variable we are interested in counting the unique occurrences of. Finally, we terminate the chain with `nunique()`, which performs the actual aggregation.

You can use the following basic syntax to count the number of unique values by group in a pandas DataFrame:

df.groupby('group_column').nunique()

It is important to understand that the result of this operation is a Pandas Series, where the index consists of the unique values from the grouping column(s), and the values represent the count of unique entries found in the target column within that specific group. This output structure is exceptionally convenient for immediate visualization or for integration into further **data analysis** pipelines.

Setting Up the Demonstration Dataset

To provide concrete, reproducible examples, we will utilize a small, simulated dataset representing team sports statistics. This Pandas DataFrame contains information across multiple categorical and numerical columns, allowing us to demonstrate both single and multiple column grouping strategies effectively. The data includes team identifiers, player positions, points scored, and rebounds collected.

We begin by importing the necessary Pandas library and then constructing the DataFrame using standard dictionary notation. Notice the repetition in values across the 'team', 'position', 'points', and 'rebounds' columns. This repetition is precisely what allows us to illustrate the difference between a total count (which would include duplicates) and a unique count (which filters them out).

The following examples show how to use this syntax with the following DataFrame:

import pandas as pd

#create DataFrame
df = pd.DataFrame({'team': ,
'position': ,

'points': ,
'rebounds': })

#view DataFrame
df

team position points rebounds
0 A G 5 11
1 A G 7 8
2 A G 7 10
3 A F 9 6
4 A F 12 6
5 B G 9 5
6 B G 9 9
7 B F 4 12
8 B F 7 13
9 B F 7 15

As observed in the output, Team A has 5 records and Team B also has 5 records, totaling 10 entries. However, notice how the 'points' column contains multiple instances of the values 7 and 9. Our objective in the following examples is to use `groupby()` to isolate the analysis to each team and then count how many distinct point totals exist within those subsets, effectively measuring the scoring diversity of each team.

Example 1: Grouping by a Single Column & Counting Unique Values

In the simplest application of this technique, we group the **DataFrame** by a single categorical variable. In our sports example, we want to assess the variety of scores recorded by each team. We will group by the `'team'` column and apply the `nunique()` function to the `'points'` column.

The code below executes this operation, grouping all rows belonging to Team A together and all rows belonging to Team B together. Within these two segregated groups, Pandas then checks every value in the 'points' column, discarding duplicates before providing the final count. This provides a clear metric of scoring performance heterogeneity between the two groups.

The following code shows how to count the number of unique values in the 'points' column for each team:

#count number of unique values in 'points' column grouped by 'team' column
df.groupby('team').nunique()

team
A 4
B 3
Name: points, dtype: int64

The resulting Series clearly demonstrates the unique scoring distributions. Team A recorded four distinct point totals (5, 7, 9, and 12), while Team B recorded only three distinct point totals (4, 7, and 9). This result indicates that Team A exhibited a slightly wider variety in individual player scoring outcomes during the recorded period compared to Team B. Understanding such statistics is foundational in quantitative **data analysis**.

There are **4** unique 'points' values for team A.
There are **3** unique 'points' values for team B.

Visualizing Unique Elements with the unique() Method

While the `nunique()` method provides the quantitative count--an aggregate number--it often proves useful to visualize the actual unique elements that contribute to that count. The Pandas `unique()` method, when applied after a `groupby()` operation, does precisely this: it returns an array of the distinct, non-duplicate values present in the specified column for each subgroup.

This approach transforms the Series output from numerical counts into a Series of arrays (or lists), where each element in the array represents one of the unique scores. This technique is invaluable for data validation, sanity checks, and preparing categorical features for machine learning models, as it allows analysts to instantly identify the range and nature of unique scores within each category.

Note that we can also use the **unique()** function to display each unique 'points' value by team:

#display unique values in 'points' column grouped by 'team'
df.groupby('team').unique()

team
A
B
Name: points, dtype: object

The output confirms our `nunique()` results. For Team A, we see the list (four unique elements), and for Team B, we see (three unique elements). It is important to remember that while `nunique()` returns an integer Series, `unique()` returns an object Series containing NumPy arrays or lists,

which must be handled appropriately if further numerical computations are required.

Example 2: Advanced Aggregation with Multiple Columns

The true power of the GroupBy method is revealed when we group by multiple columns simultaneously. This allows us to perform multi-dimensional aggregation, analyzing unique counts based on the intersection of several categorical variables, such as calculating unique scores broken down by both the `'team'` and the player's `'position'`.

When grouping by a list of columns (e.g.,), Pandas creates subgroups for every unique combination of these variables (e.g., Team A, Position F; Team A, Position G, and so forth). This hierarchical grouping generates a MultiIndex in the resulting Series, providing a granular view of the unique data distribution across fine-grained subsets of the original **DataFrame**.

The following code shows how to count the number of unique values in the 'points' column, grouped by team *and* position:

#count number of unique values in 'points' column grouped by 'team' and 'position'
df.groupby().nunique()

team position
A F 2
G 2
B F 2
G 1
Name: points, dtype: int64

The output now provides a significantly more detailed breakdown. For instance, players in position 'G' on Team B recorded only 1 unique score (meaning all 'G' players scored the same amount), whereas players in position 'G' on Team A recorded 2 unique scores. This level of detail is crucial for performance metrics and targeted interventions in sports **data analysis**, highlighting specific positional contributions to the team's overall scoring profile.

There are **2** unique 'points' values for players in position 'F' on team A.
There are **2** unique 'points' values for players in position 'G' on team A.
There are **2** unique 'points' values for players in position 'F' on team B.
There is**1** unique 'points' value for players in position 'G' on team B.

Advanced Visualization of Multi-Grouped Unique Elements

Just as in the single-group scenario, we can inspect the actual values that make up the unique

counts when grouping by multiple columns. Applying `unique()` after the multi-column `GroupBy` operation yields a highly descriptive Series that shows the precise scoring metrics for each team-position combination.

This visualization is particularly useful when the unique counts are low, allowing for immediate confirmation of the data points involved. For example, knowing that Team B, Position G, only has 1 unique point value is good, but seeing the value immediately confirms that all players in that category scored 9 points. This provides context that a simple numerical count might obscure.

Once again, we can use the **unique()** function to display each unique 'points' value by team and position:

#display unique values in 'points' column grouped by 'team' and 'position'
df.groupby().unique()

team position
A F
G
B F
G
Name: points, dtype: object

This output is often used as an intermediate step before further processing, such as calculating the mean or median of the unique scores, or for generating summarized pivot tables. By coupling `groupby()` with both `nunique()` and `unique()`, analysts gain both quantitative summary statistics and qualitative insight into the data's underlying structure.

Applications and Use Cases in Data Analysis

Counting unique values across groups is far from a trivial exercise; it is a foundational technique with wide-ranging applications in professional **data analysis**, data quality assessment, and feature engineering. One of the most common applications is evaluating the cardinality of categorical features, which is essential for tasks like database optimization and determining appropriate modeling techniques.

For instance, in a large customer database, grouping by 'Region' and counting the unique 'Customer IDs' can quickly reveal if there are regional overlaps or data entry errors, where the same ID might be mistakenly assigned to multiple regions. Similarly, in time-series data, grouping by 'Date' and counting unique 'Event Types' helps analysts track event diversity over time, spotting trends or sudden changes in system behavior.

Furthermore, in scenarios involving market research or product management, this method is key for measuring product adoption. Grouping by 'Product Category' and counting unique 'User IDs' provides an immediate, clear measure of the penetration or audience diversity for different product lines, allowing business intelligence teams to make data-driven decisions regarding inventory, marketing allocation, and product development strategy. The combination of the GroupBy mechanism and `nunique()` is truly indispensable for generating these critical business metrics.