Learning Pandas: Counting Unique Values in DataFrames with Examples

Authored by

Mohammed looti

November 2, 2025

Introduction to Cardinality and Unique Value Counting in Pandas

Data analysis often requires a foundational understanding of data distribution and quality. One of the most crucial initial steps is assessing the **cardinality** of specific features--that is, determining the number of distinct, non-repeating entries within a dataset column or row. For users working within the Python ecosystem, the Pandas library provides highly optimized tools for this purpose. Specifically, the built-in `nunique()` function is the standard and most efficient method used to count the total number of unique values contained within a DataFrame or Series object.

Understanding cardinality is vital for several reasons, ranging from feature engineering to memory optimization. Low cardinality columns (like 'Gender' or 'Region') are often suitable for categorical encoding, while high cardinality columns (like 'User ID' or 'Timestamp') might require aggregation or hash encoding. The ability to quickly quantify these unique counts allows data scientists to make informed decisions about how to preprocess, visualize, and model their information. This comprehensive guide details the practical application of the `nunique()` method, showcasing its versatility across columns, rows, and complex grouped scenarios.

The function operates straightforwardly, calculating the number of distinct elements, excluding any `NaN` (Not a Number) values by default. This exclusion is generally desirable in data cleansing operations, ensuring that missing data points do not inflate the unique count. Mastery of this function is fundamental for any serious user of the Pandas library, providing immediate insights into the structure and diversity of the underlying data. We will begin by examining the core syntax before diving into practical, runnable examples using a sample dataset designed for demonstration.

The Core Function: `nunique()` Syntax and Parameters

The `nunique()` method is exceptionally flexible, allowing calculations across the entire DataFrame or targeting specific data structures such as a single Series. When applied to a full DataFrame, the function returns a Series where the index represents the column names and the values represent the unique count for that respective column. This default behavior assumes that the user is interested in checking the diversity of features (columns).

The function uses the following basic syntax, which can be modified using the `axis` parameter to control the direction of the count:

#count unique values in each column
df.nunique()

#count unique values in each row
df.nunique(axis=1)

The critical parameter here is `axis`. By default, `axis=0` (or no axis specified) directs the operation downwards along the rows, thus counting unique values per column. When `axis=1` is specified, the operation is directed horizontally across the columns, resulting in the unique count for each row. Understanding the role of the `axis` parameter is essential for accurate data manipulation in Pandas, as it dictates whether the function operates on features or observations.

While the examples below focus primarily on the core functionality, it is worth noting that `nunique()` also accepts parameters like `dropna` (boolean, default is `True`), allowing users to include or exclude `NaN` values from the total count. For most standard data quality checks, the default settings are appropriate, but advanced scenarios might require setting `dropna=False` if missingness itself is considered a unique category.

Setting Up the Demonstration Data Structure

To illustrate the practical application of `nunique()`, we will first create a sample DataFrame. This structure represents idealized sports statistics, containing categorical data (team name) and numerical data (points, assists, rebounds), which provides a robust foundation for observing how unique counts vary across different data types and distributions.

The following code block imports the necessary Pandas library and constructs the sample data. We will then print the resulting DataFrame to visualize the data before performing any calculations. This preparation ensures clarity regarding the input structure and the expected output of the uniqueness checks.

import pandas as pd

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

#view DataFrame
df

team points assists rebounds
0 A 8 5 11
1 A 8 8 8
2 A 13 7 11
3 A 13 9 6
4 B 22 12 6

5 B 22 9 5
6 B 25 9 9
7 B 29 4 12

As shown in the output, our sample data consists of eight records (rows) and four features (columns): `team`, `points`, `assists`, and `rebounds`. By analyzing this data manually, we can anticipate that the `team` column has a low unique count (A and B), while the statistics columns exhibit higher variance, which we will now quantify precisely using the `nunique()` function.

Example 1: Counting Unique Values Across Columns (Features)

The most common use case for `nunique()` is calculating the unique count for every feature (column) within the DataFrame. This calculation provides immediate feedback on the diversity and potential redundancy within the data structure. When `nunique()` is called without any parameters, it defaults to operating along `axis=0`, yielding a count for each column.

The code below demonstrates this default behavior. We are instructing Pandas to scan each column vertically and report how many distinct values exist. This output is returned as a Pandas Series object, which is highly convenient for subsequent analysis or reporting tasks.

#count unique values in each column
df.nunique()

team 2
points 5
assists 5
rebounds 6
dtype: int64

The resulting Series clearly summarizes the cardinality of each feature. We can interpret the results as follows:

The 'team' column has **2** unique values (A and B). This indicates low cardinality, confirming it is a highly categorical variable suitable for simple encoding.

The 'points' column has **5** unique values (8, 13, 22, 25, 29). Despite having 8 rows, only 5 distinct point totals were recorded.

The 'assists' column has **5** unique values (5, 8, 7, 9, 12, 4). The value 9 appears multiple times, reducing the unique count below the total row count.

The 'rebounds' column has **6** unique values (11, 8, 6, 5, 9, 12). This column exhibits the highest diversity among the numerical statistics in this dataset.

Analyzing this output is critical for data cleaning and feature selection. Columns with a unique count equal to the total number of rows (like an ID column) offer no predictive power unless they are used as index identifiers, while columns with very low unique counts might be prime candidates for specific statistical tests or grouping operations.

Example 2: Analyzing Unique Values Row-Wise (Observations)

While counting unique values per column is standard, there are scenarios where calculating the unique count for each row (observation) is necessary. This calculation helps identify rows that exhibit high levels of internal duplication or consistency across their recorded features. For instance, in a survey dataset, a low row-wise unique count might indicate a respondent provided identical answers across several different, independent questions.

To perform row-wise uniqueness checks, we must explicitly set the `axis` parameter to `1`. This instructs the `nunique()` function to iterate horizontally, returning the unique count for each record.

#count unique values in each row
df.nunique(axis=1)

0 4
1 2
2 4
3 4
4 4
5 4
6 3
7 4
dtype: int64

The output is a Series indexed by the original DataFrame row index, providing the unique count for that specific observation. Since our DataFrame has four columns (features), the maximum possible unique count for any row is 4.

Examining the results reveals specific insights into data redundancy at the observation level:

The first row (Index 0) has **4** unique values (A, 8, 5, 11). All four feature values are distinct.

The second row (Index 1) has only **2** unique values. Let's look at the original data: (A, 8, 8, 8). The

values 8 and A are present, meaning the number 8 appears three times, resulting in a count of 2 unique items (A and 8).

The third row (Index 2) and others mostly show **4** unique values, indicating no repetition of values across the different features for those observations.

Row 6 has **3** unique values. Looking at the original data: (B, 25, 9, 9). The value 9 is repeated, resulting in three unique items (B, 25, 9).

Row-wise unique counting is a powerful tool for quality assurance, especially when dealing with data derived from forms or sequential inputs where consistency checks across fields are paramount. A low unique count in a row can often signal data entry errors or specific patterns of homogeneity that warrant further investigation.

Example 3: Advanced Analysis with Grouped Uniqueness

One of the most powerful analytical techniques in Pandas is the split-apply-combine strategy, typically implemented using the `groupby()` function. Combining `groupby()` with `nunique()` allows us to determine the unique count of a variable *within* different subgroups defined by another categorical variable. This is invaluable for answering questions like: "How many unique products did each retailer sell?" or, in our case, "How many unique point totals were achieved by each team?"

To perform this grouped uniqueness calculation, we first select the grouping column (`'team'`), then specify the column we want to count unique values from (`'points'`), and finally apply the `nunique()` aggregation function.

#count unique 'points' values, grouped by team
df.groupby('team').nunique()

team
A 2
B 3
Name: points, dtype: int64

This output provides a highly specific summary: the number of unique point values observed for each distinct team category. This insight is much more granular than the overall unique column count performed in Example 1.

The results clearly show a disparity in the diversity of point totals between the two groups:

Team 'A' has **2** unique 'points' values (8 and 13). Despite having four observations for Team A,

only two distinct point totals were registered.

Team 'B' has **3** unique 'points' values (22, 25, and 29). Team B showed greater variance in their scoring records across their four observations.

This method is indispensable for segmentation analysis. By applying `groupby()` and `nunique()`, data analysts can quickly characterize the behavioral profiles, diversity metrics, or inventory characteristics across different segments of a population or product line, providing actionable insights for business intelligence.

Summary and Additional Resources

The `nunique()` function is a fundamental building block in the Pandas toolkit for data exploration and quality assessment. Whether used in its default column-wise mode, modified for row-wise checks using the `axis=1` parameter, or combined with powerful methods like `groupby()`, it provides immediate and accurate metrics on data cardinality. Mastering the application of this function ensures that analysts can efficiently handle data preprocessing steps, identify categorical variables, and prepare datasets for robust statistical modeling.

The three examples demonstrated the versatility of `nunique()` across different analytical requirements, highlighting its importance in both macro-level (column) and micro-level (row and group) data investigations. Always remember the distinction between counting all values (using `count()`) and counting only the distinct instances (using `nunique()`) to ensure accurate reporting and analysis.

For those looking to deepen their expertise in data manipulation within the Python ecosystem, the following tutorials explain how to perform other common operations in Pandas: