Learning Pandas: Counting Unique Values with the nunique() Function

Authored by

Mohammed looti

November 13, 2025

In the crucial preliminary stages of data processing and exploratory analysis, determining the unique components within a dataset is a fundamental requirement. Data scientists and analysts frequently need to quantify the number of distinct, non-repeating entries across specific features or rows. This count is vital for assessing **data quality**, understanding feature variability, and calculating **data cardinality**. In the universe of Python data manipulation, specifically utilizing the powerful pandas library, this task is efficiently handled by a specialized function designed to operate directly on a pandas DataFrame or Series.

High cardinality in a feature might suggest a need for dimensionality reduction, while low cardinality implies a limited scope of possible values. Mastering the tools that quickly provide this insight is essential for effective data management, particularly when working with large or complex datasets containing intentional or unintentional redundancies.

Introducing the nunique() Function

The standard and most streamlined approach provided by the pandas library for calculating the number of unique elements is the **nunique()** method. This function is optimized to return the count of distinct values along a specified direction (known as the **axis**). It is applicable to both entire DataFrames, where it returns a count per column, and to individual Series objects, where it returns a single integer count.

Understanding the standard invocation and the optional parameters is key to leveraging this function effectively, allowing analysts precise control over how unique values are aggregated and how missing data is handled during the computation.

The foundational syntax for applying this method to a DataFrame object is:

DataFrame.nunique(axis=0, dropna=True)

The behavior of **nunique()** is primarily governed by its two optional parameters:

axis: This parameter dictates the direction of the count operation. By default, **axis=0** specifies that unique values should be counted column-wise (aggregating across rows), which is the standard method for determining feature cardinality. Setting **axis=1**, conversely, specifies row-wise counting (aggregating across columns), providing the unique count for each individual observation. Understanding the axis is crucial for correct data aggregation.
dropna: This boolean parameter controls the inclusion of missing values, which are typically represented in pandas as NaN (Not a Number). When **dropna** is set to its default value of **True**, all **NaN** entries are excluded from the unique value tally. If the analyst requires **NaN** to be treated as a distinct, unique category--for example, to explicitly track missingness--this parameter must be set to **False**.

Example 1: Counting Unique Values per Column (Default Behavior)

To demonstrate the utility of the **nunique()** method, we will first construct a representative sample DataFrame. This dataset simulates basketball performance metrics, containing categorical data (team affiliation) and numerical data (points and assists), along with strategically placed missing values to explore parameter handling.

The creation of this DataFrame uses the standard pandas and NumPy libraries:

import pandas as pd
import numpy as np

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

#view DataFrame
print(df)

team points assists
0 A 12.0 10
1 A 12.0 22
2 B 18.0 24
3 B 13.0 20
4 C NaN 14
5 C NaN 18
6 C 12.0 10
7 D 29.0 12

Our initial objective is to quickly ascertain the number of unique elements within each column, utilizing the function's default parameters. Since the default configuration uses **axis=0** (column-wise) and **dropna=True** (excluding missing values), we simply call `df.nunique()` without any explicit arguments. This returns a pandas Series where the index corresponds to the column names and the values represent the unique count for that feature.

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

team 4
points 4

assists 7
dtype: int64

The resulting output clearly summarizes the variability of our features under the default settings. We find that the **team** column has 4 unique groups (A, B, C, D), while the **assists** column shows the highest diversity with 7 distinct values. Crucially, the **points** column registers 4 unique scores (12, 18, 13, 29). The two **NaN** entries present in the raw data were automatically disregarded because **dropna** defaults to **True**, ensuring that only valid observations contribute to the unique count.

Controlling NaN Inclusion Using dropna Parameter

While excluding missing data is often the desired default, there are compelling reasons in advanced statistical analysis or data profiling to treat the presence of missingness itself as a distinct category. For instance, if the absence of a score holds specific analytical meaning, it should be included in the unique value tally. The **nunique()** function provides explicit control over this behavior through the **dropna** parameter.

To compel pandas to count **NaN** values as a single, separate unique entry, we must set **dropna** to **False**. This adjustment fundamentally alters the counting mechanism for any column containing missing data. We apply this modification to our existing DataFrame to observe the effect, paying close attention to the `points` column, which currently holds two missing records.

#count number of unique values in each column, including NaN in counts
df.nunique(dropna=False)

team 4
points 5
assists 7
dtype: int64

Reviewing the revised output reveals a significant change in the **points** column count, which has increased from 4 to **5**. This increase is directly attributable to treating the two NaN entries as a single, distinct unique category alongside the four numerical scores. In contrast, the counts for the **team** and **assists** columns remain at 4 and 7, respectively, as they did not contain any NaN values to affect the unique count.

Example 2: Applying nunique() to a Single Column (Series)

Often, the analytical task is focused not on the entire DataFrame's feature set but on calculating

the cardinality of just one specific feature. When a single column is selected using standard bracket notation (e.g., `df`), the result is a pandas Series object. The **nunique()** method is directly available on the Series, offering a quick, isolated calculation that is more efficient than running the operation on the entire DataFrame when only one result is needed.

If our goal is specifically to determine the unique count for the **points** column, including missing values as a unique entity, we can chain the column selection and the function call. This method is preferred when the objective is to return a simple scalar integer, streamlining the code and bypassing the generation of a multi-column Series containing counts for irrelevant features.

#count number of unique values in points column only
df.nunique(dropna=False)

5

The execution of this focused syntax returns the integer **5**. This scalar output precisely confirms the unique element count exclusively within the **points** column, adhering to the requirement that missing data be included in the final tally.

Example 3: Analyzing Unique Values Row-wise (axis=1)

While feature cardinality (column-wise analysis, **axis=0**) is the most common use case, there are specialized scenarios--such as assessing the homogeneity of individual observations--where calculating the unique count across rows is necessary. This calculation determines the number of distinct values present within a single record, aggregating across all features defined in that row. To perform this unique count per observation, we must explicitly set the **axis** parameter to **1** within the **nunique()** function call.

When **axis=1** is active, pandas performs data aggregation by iterating row by row, calculating the unique count within that horizontal slice. Applying this configuration to our basketball dataset helps reveal the inherent statistical variation within each player's recorded metrics, with **NaN** values excluded by default (since we do not set `dropna=False`).

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

0 3
1 3
2 3
3 3
4 2

5 2
6 3
7 3
dtype: int64

The resulting pandas Series maps the DataFrame's index to the unique count for that corresponding row. Since our dataset consists of three columns (team, points, assists), a count of 3 signifies that all three values in the observation are distinct. A count lower than 3 indicates that duplicate values were present across the columns for that player, or that missing values were ignored.

For detailed inspection, consider the following results:

Rows 0, 1, 2, 3, 6, and 7 all register **3** unique values, meaning the team, points, and assists were all different for those records.
Rows 4 and 5, however, register only **2** unique values. This is expected because these rows contained a single NaN entry in the 'points' column, and since **dropna=True** by default, the calculation excluded the missing value, leaving only two unique non-missing values (the team name and the assist count).

Summary and Further Reading

The pandas **nunique()** function stands as a critical tool in the data scientist's arsenal, providing a quick, powerful, and flexible mechanism for assessing the variability and cardinality of datasets. Whether the analysis requires counting distinct entries feature-by-feature (**axis=0**) or quantifying variation within individual observations (**axis=1**), **nunique()** handles the calculation with precision.

Mastery of the optional **dropna** parameter is equally important, as it allows analysts to make an informed decision regarding the treatment of missing data, ensuring that the unique counts accurately reflect the specific requirements of the ongoing analytical task.

For a detailed reference on all parameters, advanced usage, and performance considerations for this function, always consult the official pandas documentation.

Additional Resources

Featured Posts

5 Statistical Biases to Avoid

April 25, 2024

5 Free Statistics Courses for Beginners

April 19, 2024

5 MIT Statistics Courses That Are Free

April 18, 2024

5 Free Books to Learn Statistics

April 18, 2024

How to Use the info() Method in Pandas

April 12, 2024

How to Use pct_change() in Pandas

April 12, 2024