How to Extract Unique Values from Pandas DataFrames

Authored by

stats writer

November 20, 2025

Data analysis often requires understanding the distribution and variety within a dataset. In the realm of Python data science, the Pandas library provides highly efficient tools for extracting this information. Specifically, determining the unique values present in a DataFrame column or a Series is a fundamental step in data profiling and preparation.

The primary method utilized for this purpose is the built-in unique() function. This powerful method is designed to quickly scan the selected data structure and return an array containing every distinct entry. Crucially, the output is typically a NumPy array, which maintains high performance, especially when dealing with large datasets. Understanding how to use unique() allows analysts to rapidly assess column cardinality, verify data integrity, and identify unexpected entries. Furthermore, while unique() focuses on value extraction, the related function, drop_duplicates(), offers a complementary approach by removing duplicate rows entirely, enabling the isolation of records based on their unique characteristics within specific columns.

While the standard functionalities of Pandas are extensive, real-world data frequently contains missing values, represented as NaN (Not a Number). The default behavior of certain Pandas operations, including unique(), often includes these NaN values in the resulting list of unique entries. For many analytical tasks, particularly when calculating metrics or generating category lists, it is essential to exclude these missing values. This guide will delve into creating a robust, reusable custom function that seamlessly handles data cleaning during the unique value extraction process, ensuring cleaner and more meaningful results for subsequent analysis.

Understanding the Core Methods for Uniqueness in Pandas

Identifying unique entries is a cornerstone of exploratory data analysis (EDA). Before diving into custom solutions, it is vital to grasp how Pandas handles uniqueness natively. The primary tools at our disposal are the unique() method, designed for Series objects, and the drop_duplicates() method, primarily used on DataFrames. Although they achieve similar goals--the identification of distinct data points--they operate at different structural levels and return different output formats.

The unique() method, when called on a specific column (which is a Series), is the most direct way to generate a list of all distinct values. It is exceptionally fast because it leverages underlying NumPy optimizations. However, its simplicity comes with a caveat: it treats NaN as a value to be included in the uniqueness calculation, which is often counter-intuitive when the goal is to list valid categories or measured quantities. Conversely, the drop_duplicates() method focuses on row-level operations. If applied to an entire DataFrame, it removes any row that is an exact duplicate of a previous row. When used with the optional subset argument, it can isolate rows that are unique based only on the values in specified columns, providing a powerful way to filter data before further processing.

A key difference to remember is the output type. unique() returns a flat NumPy array containing the

unique elements, suitable for immediate listing or iteration. drop_duplicates(), however, returns a filtered DataFrame or Series. Choosing the right method depends entirely on the objective: if the goal is simply to list the values, unique() is preferred; if the goal is to filter the dataset to contain only unique records, drop_duplicates() is the correct choice. When dealing with missing data, the standard unique() approach must often be supplemented, as detailed in the following sections.

The Default Behavior of the unique() Function

The Pandas library adheres to strict data standards, and this includes how it handles missing or invalid data represented by NaN. When the unique() method is executed on a Series, it treats the NaN marker itself as a distinct value. This behavior stems from the nature of NaN in floating-point arithmetic, where it serves as a valid placeholder for data that is unknown, unrepresentable, or missing.

While technically correct from a data typing standpoint--as NaN is distinct from any numerical value--its inclusion in a list of unique values can often muddy the waters during data profiling. For instance, if you are attempting to identify all the unique product IDs or geographical regions present in a dataset, the presence of NaN in the resulting list means that the list cannot be directly used for tasks such as creating categorical variables or populating dropdown menus without an extra cleaning step. The challenge is particularly pronounced when working with large volumes of messy data, where missing values are common and must be explicitly handled rather than simply listed.

To illustrate, imagine a column containing sales figures. If five out of a thousand entries are missing (i.e., contain NaN), the unique() method will return the array of unique sales numbers plus one instance of NaN. Although Pandas offers other methods like nunique() (which excludes NaN by default when counting), the challenge remains for the extraction of the actual unique values themselves. Therefore, for scenarios requiring a clean list of valid, non-missing entries, a pre-processing step or a customized function is necessary to filter out NaN before the uniqueness calculation is performed.

Developing a Custom Function to Exclude NaN Values

To overcome the limitations of the default unique() behavior concerning missing values, we can define a succinct and highly reusable custom function. This function leverages the power of method chaining in Pandas, combining a cleaning step with the uniqueness extraction. The primary method utilized for cleaning is dropna(), which effectively removes all instances of NaN from the selected Series before the unique elements are identified.

The construction of this custom function is straightforward, requiring only two chained methods applied to the input Series (represented here by the variable x). First, x.dropna() is called, which

returns a new Series containing only the non-missing values. Second, the standard .unique() method is called on this cleaned result. This ensures that the uniqueness calculation is performed exclusively on the valid data points, thereby guaranteeing that the final output is free of any NaN markers.

By encapsulating this logic, we create an atomic operation that can be easily applied across various columns and even integrated into more complex aggregation processes, such as those involving groupby(). This approach promotes cleaner code and significantly reduces the risk of errors associated with manually handling missing data every time unique values are needed. Below is the definition for the proposed custom function:

You can define the following custom function to find unique values in Pandas and ignore NaN values:

```
def unique_no_nan(x):
return x.dropna().unique()
```

This function will return a Pandas array that contains each unique value except for NaN values.

Setting Up the Demonstration Dataset

To effectively demonstrate the functionality of both the standard unique() method and our custom unique_no_nan() function, we must first establish a sample dataset. This DataFrame will be structured to mimic common real-world data issues, specifically featuring duplicate entries and missing values (NaN). We will utilize the Pandas library for DataFrame creation and the NumPy library to introduce the missing value placeholder.

Our sample DataFrame will represent basketball game statistics, containing a categorical column ('team') and a numerical column ('points'). By intentionally duplicating scores for the 'Mavs' team and ensuring that one entry for the 'Celtics' team has a missing score, we create a scenario where the differences between standard and custom unique value extraction methods become readily apparent. This setup provides a clean testing ground for verifying the efficacy of the unique_no_nan() function, especially when dealing with numerical data types that frequently exhibit missing values.

The following code block imports the necessary libraries and constructs the sample DataFrame. Reviewing the printed output confirms the presence of duplicates (e.g., Mavs scoring 95.0 points twice) and the critical missing value at index 5, denoted as NaN. This structure is essential for proceeding with the examples:

```
import pandas as pd
```

import numpy as np

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

#view DataFrame
print(df)

team points
0 Mavs 95.0
1 Mavs 95.0
2 Mavs 100.0
3 Celtics 113.0
4 Celtics 100.0
5 Celtics NaN

Example 1: Isolating Unique Values in a Single Column (Handling NaN)

Our first demonstration focuses on extracting the unique scores from the 'points' column, highlighting the practical difference between the standard unique() function and our customized unique_no_nan() function. This scenario is common when attempting to generate a clean list of possible numerical values or categories for validation or visualization purposes.

When applying the standard unique() function directly to the df Series, the output array accurately lists all distinct numerical values observed: 95, 100, and 113. However, consistent with its default behavior, the function also includes nan as the fourth element in the resulting array. While this might be informative if we were counting the total number of unique elements including the missing marker, it complicates tasks where only valid, recorded scores are needed.

The standard application of the unique() function is shown below, confirming the inclusion of the missing value marker:

#display unique values in 'points' column
df.unique()

array()

Notice that the unique() function includes nan in the results by default. To address this, we now apply our custom function, unique_no_nan(). By first invoking dropna() internally, the function strips out the missing value before executing the uniqueness logic. This results in a cleaner, more

analytically useful array containing only the valid numerical scores. This distinction is vital for processes that cannot tolerate non-numerical data points.

The application of our custom function, achieving the desired exclusion of the NaN value, is demonstrated here:

#display unique values in 'points' column and ignore NaN
unique_no_nan(df)

array()

As demonstrated, our function successfully returns each unique value in the 'points' column, effectively excluding the NaN entry.

Example 2: Finding Unique Values Across Groups (Groupby Operations)

A more advanced and frequently encountered task in data analysis is finding unique values within subgroups defined by another variable. In Pandas, this is achieved using the groupby() method. When performing aggregation on grouped data, we often want to know the unique entries associated with each group--in our case, the unique scores recorded by each 'team'.

We first attempt this using the standard aggregation method, combining groupby() and agg(), specifying 'unique' as the aggregation function. This method is concise and powerful. However, when we inspect the results for the 'Celtics' team, we observe that the resulting array of unique scores still contains nan, consistent with the default behavior of the unique() operation even within the aggregation context. While the 'Mavs' results are clean (as they had no missing data), the presence of nan in the 'Celtics' array requires explicit handling if the output is to be used for further processing.

The standard grouping and aggregation method is shown here:

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

unique
team
Celtics
Mavs

Notice that the unique() function includes nan in the results for the 'Celtics' team by default. To effectively apply our custom cleaning logic within the grouping structure, we must switch from agg()

to the more flexible apply() method. The apply() method allows us to pass a custom Python function--in this case, unique_no_nan()--to operate on each subgroup (each team's Series of points) individually. We use a lambda function to seamlessly integrate our pre-defined cleaning logic into the grouping workflow.

By using apply(), each subset of data is first cleaned using dropna(), and then the unique values are extracted. This results in an accurate and clean list of scores for both teams, successfully omitting the missing value associated with the 'Celtics' records:

#display unique values in 'points' column grouped by team and ignore NaN
df.groupby('team').apply(lambda x: unique_no_nan(x))

team
Celtics
Mavs
Name: points, dtype: object

Our function effectively returns each unique value in the 'points' column for each 'team', while successfully omitting the NaN values, providing a clean array for each group.

Alternative Techniques for Uniqueness and Counting

While the combination of dropna() and unique() provides the most direct way to extract clean arrays of values, Pandas offers other complementary techniques essential for comprehensive data exploration related to uniqueness. These methods are crucial for scenarios where the objective is not just to list the values, but to understand their frequency or count.

One of the most powerful related functions is value_counts(). When applied to a Series, this function returns a new Series where the index consists of the unique values and the values themselves are their corresponding counts. By default, value_counts() automatically excludes NaN values, making it an excellent tool for frequency distribution analysis without needing custom cleaning. If NaN counts are required, the optional argument dropna=False can be passed.

Another technique is the nunique() method, which specifically returns the number of unique elements in a Series. Unlike the unique() function, which returns the list of values, nunique() returns a single integer count. Crucially, like value_counts(), nunique() excludes NaN by default. If, for instance, df.nunique() were run on our sample data, the result would be 3 (95, 100, 113), not 4, providing a quick summary statistic ideal for data profiling reports. For cases where NaN must be included in the count, the argument dropna=False must be specified.

In summary, the choice of method depends on the analytical goal:

Use unique() combined with dropna() (i.e., our custom function) when the objective is to retrieve a clean NumPy array containing only the valid unique elements.

Use value_counts() when the objective is to generate a frequency distribution table of the unique values (excluding NaN by default).

Use nunique() when the objective is to quickly ascertain the cardinality (total count of unique items, excluding NaN) of a Series.

Summary and Conclusion

Mastering the extraction of unique values is fundamental to effective data manipulation in Pandas. While the standard unique() function is fast and reliable for listing all distinct entries, its default inclusion of NaN requires analysts to implement custom solutions for cleaner data processing. The proposed unique_no_nan() function offers a robust, Pythonic solution by chaining the necessary data cleaning step (dropna()) before the uniqueness extraction, ensuring the resulting array contains only valid data points.

This custom function proves its versatility by integrating seamlessly into complex workflows, such as those involving group-by operations using the apply() method. By demonstrating its application in both single-column and grouped scenarios, we have shown how analysts can maintain data integrity and clarity even when dealing with imperfect, real-world data containing missing values. Utilizing the right method--whether it is the quick count of nunique(), the detailed distribution of value_counts(), or the precise value extraction of unique_no_nan()--empowers users to perform rigorous and efficient exploratory data analysis.

For those interested in expanding their Pandas expertise, further exploration into related data cleaning and aggregation methods is highly recommended. Understanding these functions ensures that data preparation remains the most efficient part of any data science pipeline.

The following tutorials explain how to perform other common functions in Pandas: