Learning Pandas: A Guide to Identifying Unique Values, Excluding NaN

Authored by

Mohammed looti

November 15, 2025

The Critical Challenge: Identifying Unique Values While Ignoring NaN in Pandas

During the initial phases of data preparation and exploratory data analysis (EDA) using the powerful Pandas library, one of the most frequent and essential operations is the accurate identification of **unique values** within a specific data column, which is typically stored as a Series object. This step is fundamental for gaining insight into data distribution, validating categorical features, and ensuring the overall quality of the dataset before applying advanced analytical models. However, real-world data is inherently messy, and the presence of missing entries introduces a significant hurdle to achieving clean uniqueness summaries.

Missing data, particularly in numerical and statistical contexts, is conventionally represented by the special floating-point marker, NaN (Not a Number). By design, standard Pandas methods, such as the commonly utilized `unique()` function, treat every instance of NaN as a distinct, unique element when compiling the resulting list. While this behavior is mathematically correct based on the underlying numerical standards, it frequently conflicts with the practical requirements of data scientists who need to isolate only the observed, meaningful categories or measured values, thereby explicitly excluding missing indicators.

The default inclusion of NaN complicates subsequent workflows that rely on precise lists of distinct categories for filtering, aggregation, or visualization. This article provides a highly efficient and reusable solution: a dedicated Python function tailored to extract unique values from a Series or DataFrame column while **systematically ignoring** all missing data markers. We will detail the implementation of this custom utility and demonstrate its efficacy through clear, practical examples, ensuring your data exploration efforts remain focused exclusively on valid observations.

Understanding Missing Data Representation: The Peculiar Nature of NaN

Effective management of missing values is non-negotiable in data science, as it profoundly influences the reliability and accuracy of any conclusions drawn. Within the modern Python data stack, which relies heavily on libraries like NumPy and Pandas, missing numerical entries are standardized using the value NaN. This marker originates from the IEEE 754 standard for floating-point arithmetic and is designed to denote undefined or unrepresentable results, ranging from invalid mathematical operations to, more commonly in datasets, simply the absence of a recorded data point.

A key characteristic of NaN that often confuses newcomers is its inherent incomparability. Crucially, a NaN value is not considered equal to any other value, including itself (i.e., the comparison `NaN == NaN` always yields **False**). Despite this comparison anomaly, when the `unique()` function in Pandas computes distinct elements, it groups all instances of NaN into one

category, treating it as a single, distinct entry in the output array of unique values.

From an analytical perspective, where the primary objective is to catalogue distinct categories or accurate measurements, the inclusion of a missing data indicator like NaN can introduce unnecessary clutter and potentially skew interpretations. Therefore, establishing a streamlined and reliable procedure to filter out these non-data indicators prior to calculating uniqueness is a critical technique for generating clean, meaningful categorical summaries that truly reflect observed data.

Defining the Elegant Solution: The `unique_no_nan` Function

To effectively counteract the default behavior of including NaN in uniqueness computations, we can create a custom function that is both concise and highly reusable across different projects. We name this utility `unique_no_nan()`, and it skillfully utilizes the powerful method chaining capabilities inherent to Series objects in Pandas to first purify the data stream and then extract the unique components.

The implementation of our custom function is remarkably straightforward, combining the two necessary sequential operations: the `dropna()` method, which handles null removal, and the `unique()` method, which calculates distinct values.

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

When a Series, denoted here as `x`, is processed by `unique_no_nan()`, the execution follows two clear, sequential steps. Initially, the `dropna()` method is applied, which returns a new Series instance where all **null or missing entries** (including NaN values) have been completely discarded. Subsequently, the standard `unique()` method is invoked on this newly cleaned Series. The final outcome is a NumPy array containing only the distinct, meaningful values present in the original data, having successfully and explicitly excluded all indicators of missingness.

Setting Up the Demonstration: Creating a Sample DataFrame

To effectively illustrate the practical advantage of the `unique_no_nan()` function, we must first establish a representative sample dataset. This example DataFrame is designed to simulate typical real-world data structures that contain both categorical identifiers and numerical observations, including intentionally inserted missing values.

We begin by importing the necessary libraries: Pandas, which serves as the primary tool for data manipulation, and NumPy, which is essential for generating and correctly representing the NaN missing values within our numerical columns.

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

The resulting DataFrame, named `df`, contains the categorical feature `'team'` and the numerical feature `'points'`. Critically, the `'points'` column intentionally includes a NaN value, providing the ideal scenario to highlight the differences between the standard unique value extraction methods and our refined approach that rigorously excludes missing data. This structured dataset allows us to clearly observe how our custom function delivers superior, actionable results compared to Pandas' default handling of missing values.

Example 1: Finding Unique Values in a Single Series

The most straightforward use case for our custom function is isolating the unique elements within a single data column. To fully appreciate why `unique_no_nan()` is necessary, we first demonstrate the outcome of applying the standard `unique()` method directly to the `'points'` Series, which we know contains a missing value.

Applying the default method results in an array that includes the missing data indicator:

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

array()

As the output clearly shows, the standard function correctly reports `nan` as one of the distinct entries. While programmatically accurate, this is often analytically misleading, as it lists a

placeholder for missingness alongside actual observed scores (95, 100, 113). If the analytical goal is simply to obtain a clean list of achieved scores, the presence of `nan` necessitates an additional manual cleanup step after the extraction.

By contrast, utilizing our custom function, `unique_no_nan()`, provides the desired clean output instantly. The function first removes the missing value using `dropna()`, then calculates the unique set:

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

array()

This result confirms that the simple combination of `dropna()` and `unique()` is highly effective. It yields a concise NumPy array containing only the valid numerical scores, thereby greatly simplifying subsequent data processing steps where only observed, valid data points are required.

Example 2: Grouped Unique Aggregation Excluding NaN

A far more complex, yet common, scenario in data analysis involves finding unique values based on characteristics defined by other columns--a task accomplished using the powerful `groupby()` method. This method segments a DataFrame into distinct subsets, defined in our case by the `'team'` key.

When attempting to aggregate the unique scores per team using the standard `agg()` function with the built-in `'unique'` string, the missing values unfortunately persist within the grouped output, failing to provide a clean summary.

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

unique
team
Celtics
Mavs

This result confirms that the 'Celtics' group, which contained the original missing entry, still includes `nan` in its list of unique scores. This demonstrates that the default aggregation mechanism does not automatically filter out missing data during the critical grouping process. To achieve a truly clean, grouped result, integrating our custom function becomes essential.

We can seamlessly incorporate `unique_no_nan()` directly into the `groupby()` workflow by utilizing the `apply()` method. The `apply()` function is perfectly suited for executing custom, row-wise or group-wise operations on subsets of data. We employ a concise lambda function to pass the `'points'` Series for each group directly into our dedicated cleaning function.

#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

The resulting output is perfectly clean and analytically useful: the 'Celtics' team now only reports , having successfully filtered out the missing data indicator before computing uniqueness. This technique powerfully demonstrates how a simple, custom function can be integrated into complex grouped aggregation routines to ensure data fidelity and produce highly reliable analytical summaries across diverse categories.

Conclusion: Streamlining Data Analysis with Precise Custom Functions

Achieving robust and reliable data analysis in Pandas demands meticulous handling of missing data, particularly the ubiquitous challenge presented by NaN values. While Pandas offers an extensive suite of tools, the practice of customizing functions to address specific analytical requirements--such as ignoring missing values during uniqueness checks--significantly improves the efficiency and clarity of data preparation workflows.

The custom `unique_no_nan()` function we developed--by elegantly chaining the `dropna()` and `unique()` methods--provides a highly readable and succinct mechanism for obtaining lists of distinct, observed values. Whether applied to a single Series or integrated into powerful grouped aggregations, this function ensures that your unique value identification process is precise and focused solely on meaningful data points, entirely free from the artifacts of missingness.

Adopting this modular, functional approach allows data professionals to write cleaner code, eliminate repetitive manual filtering steps, and ultimately gain more accurate insights faster. This flexibility underscores the true power and extensibility of Pandas, enabling sophisticated data cleaning and exploratory tasks with remarkable elegance and minimal code complexity.

Additional Resources for Advanced Pandas Techniques

To further your expertise in handling data quality issues and mastering complex aggregations

within Pandas, we highly recommend consulting the following authoritative documentation and guides:

Pandas User Guide: Working with Missing Data
Pandas User Guide: Group By: split-apply-combine
Pandas DataFrame.apply() documentation