How to find unique values in multiple columns in Pandas?

Authored by

stats writer

December 23, 2025

Data cleaning and analysis often necessitate the identification of unique elements within a dataset. When working with the powerful Pandas DataFrame structure in Python, analysts frequently encounter scenarios where they need to extract every distinct value scattered across multiple columns, treating those columns as a single pool of data. While many users are familiar with the `DataFrame.drop_duplicates()` method, which focuses on identifying and removing duplicate rows based on column combinations, a different approach is required when the goal is to list all unique cell values regardless of their row location.

The standard way to handle unique row combinations is by utilizing the built-in `DataFrame.drop_duplicates()` method. This function is invaluable for eliminating redundant entries where all specified columns hold identical values across multiple rows. For instance, if you specify `subset=`, it will return a new DataFrame where no two rows share the same combination of values in those two columns, typically keeping the first occurrence. However, this method does not yield a consolidated list of all unique individual values present within the cells across those columns; instead, it preserves the DataFrame structure and focuses on row-level duplication.

To achieve the objective of listing every distinct item present across a selection of columns, we must employ a strategy that flattens the relevant column data into a single, cohesive Pandas Series or NumPy array before applying the uniqueness check. This is crucial because Pandas functions designed for Series or arrays, like `pandas.unique()`, are optimized for identifying unique elements within a one-dimensional structure. Therefore, understanding the combination of methods required for this transformation--specifically `.values`, `.ravel()`, and `pd.unique()`--is key to efficient and accurate data processing when seeking global column uniqueness.

Understanding the Core Methods: `unique()` and `ravel()`

When the task is to find all distinct values across several columns, the solution lies in combining two specialized Pandas/NumPy functions. This powerful combination first transforms the selected multi-column data into a flat sequence and then efficiently extracts the non-repeating elements. This methodology is preferred because it directly addresses the need for a list of unique items, transcending the row boundaries that typically govern DataFrame operations.

The process leverages the following crucial components, which must be executed in sequence: selection, flattening, and extraction. First, we select the columns of interest, which yields a smaller DataFrame subset. Second, we access the underlying data structure using the `.values` attribute, converting the DataFrame subset into a two-dimensional NumPy array. Third, the `.ravel()` function is applied to this array to collapse it into a single, one-dimensional array. Finally, the resulting flattened array is passed to `pandas.unique()`, which handles the final deduplication step.

`.values` Attribute: This converts the selected Pandas DataFrame subset into its underlying

NumPy array representation, which is necessary for efficient low-level operations like flattening.
`ravel()` Function: This NumPy function returns a flattened one-dimensional view or copy of the array. It transforms the multiple columns into a single, continuous stream of data, allowing us to treat all values from the selected columns uniformly.
`pandas.unique()` Function: Designed to process array-like objects, this function efficiently computes the unique values within the flattened data structure. Crucially, it returns these unique values in the order of their first appearance within the input, which can be beneficial for maintaining data context if order matters.

By using this layered approach, we ensure that every cell value within the designated columns contributes to the final list of unique elements. This is fundamentally different from row-based deduplication, which only considers whether the combination of values in a given row has appeared before. The `unique()` and `ravel()` combination is therefore the canonical solution for achieving a consolidated view of unique field data.

Setting Up the Data: A Practical Example

To demonstrate these methods effectively, we must first establish a sample dataset. This example uses a small Pandas DataFrame containing three columns, two of which are categorical (strings) and one numerical. Note that some values are intentionally repeated across rows and columns to illustrate how the unique value extraction process works.

We will use standard Python and Pandas syntax to import the necessary library and construct the DataFrame. Column 1 (`col1`) and Column 2 (`col2`) contain overlapping string values ('a', 'c', 'e'), ensuring that a simple row-wise operation would miss the combined unique count, while Column 3 (`col3`) contains numerical data, including a repeated value (6).

The initial setup code is as follows, defining the structure that we will analyze throughout the subsequent sections. We are specifically interested in finding the unique values within `col1` and `col2` collectively.

import pandas as pd

#create DataFrame
df = pd.DataFrame({'col1': ,
'col2': ,
'col3': })

#view DataFrame
print(df)

col1 col2 col3

0 a a 11
1 b c 8
2 c e 10
3 d f 6
4 e g 6

Observation of the DataFrame shows that `col1` contains {'a', 'b', 'c', 'd', 'e'} and `col2` contains {'a', 'c', 'e', 'f', 'g'}. The total set of unique values across both columns is {'a', 'b', 'c', 'd', 'e', 'f', 'g'}. Our goal is to programmatically extract this exact set using the combined `.values.ravel()` and `pd.unique()` technique, thus confirming the effectiveness of the chosen methodology.

Method 1: Returning the Unique Values as a NumPy Array

The most direct way to retrieve the set of unique values is by executing the core transformation sequence, resulting in a standard NumPy array. This array format is highly efficient for subsequent computational tasks and provides a clean, ordered list of all distinct elements found within the selected columns.

The code first selects the desired columns using standard DataFrame indexing (`df]`). This intermediate result is a new two-column DataFrame. We then immediately call `.values` to convert this structure into a 2D array, where each inner list corresponds to a row. Applying `.ravel()` flattens this 2D array into a 1D sequence. Finally, wrapping the entire expression in `pd.unique()` performs the deduplication.

Executing this single line of code provides the desired output, listing all unique elements in the order they were first encountered across the flattened sequence. Notice how 'a', 'c', and 'e' appear only once in the final output, even though they were present in both `col1` and `col2` within the original DataFrame.

pd.unique(df].values.ravel())

array(, dtype=object)

From this successful execution, we observe an output array containing seven unique values: 'a', 'b', 'c', 'e', 'd', 'f', and 'g'. This confirms that the combined method accurately identified all distinct entries across the specified columns, successfully treating them as a single pool of data. The resulting NumPy array is the most memory-efficient representation of these unique values.

Method 2: Transforming Unique Values into a Pandas DataFrame

While the NumPy array output is often sufficient for backend processing, analysts frequently prefer to work with a Pandas DataFrame for better integration with existing workflows, easier labeling, or visual presentation. Converting the results from an array into a DataFrame is a trivial step, accomplished by using the `pd.DataFrame()` constructor.

The first step remains identical: calculating the unique array using `pd.unique(df].values.ravel())`. We store this result in a temporary variable, `uniques`. This variable now holds the one-dimensional array of unique elements. The second step involves passing this array directly to the `pd.DataFrame()` function, which automatically converts the array into a single-column DataFrame.

This approach provides a structure that can be easily labeled, indexed, or merged with other data sources. Although it introduces a slight overhead compared to the raw array, the convenience and compatibility with the broader Pandas ecosystem often make it the preferred choice for presentation or further manipulation.

uniques = pd.unique(df].values.ravel())

pd.DataFrame(uniques)

0
0 a
1 b
2 c
3 e
4 d
5 f
6 g

The resulting output is a DataFrame indexed from 0 to 6, with a single column (automatically labeled '0' by default) containing the 7 unique string values. If necessary, the column can be renamed immediately after creation using methods such as `.rename(columns={0: 'Unique_Elements'}, inplace=True)` to improve readability and semantic clarity within the analytical script.

Calculating the Total Count of Unique Elements

In many analytical tasks, the primary interest is not the list of unique values itself, but rather the magnitude--the total number of distinct elements present. Determining this count is straightforward,

as the `pd.unique()` function returns a standard Python-compatible sequence (a NumPy array), allowing us to leverage the built-in `len()` function.

The methodology again involves first generating the array of unique values using the established sequence (selection, flattening, deduplication). Once this array is stored, the `len()` function is applied directly to the array variable. This operation quickly returns an integer representing the count of unique items, which is far more efficient than iterating through or calculating the size of a DataFrame containing the unique results.

This approach is particularly valuable for large datasets where calculating distinct counts is a preliminary step in feature engineering or cardinality assessment. Knowing the count without having to store or manipulate the entire list of unique values can significantly improve performance and resource management, especially when dealing with high-cardinality columns.

uniques = pd.unique(df].values.ravel())

len(uniques)
7

The result, 7, immediately confirms the number of distinct elements present across the two columns. This concise method serves as an excellent diagnostic tool for summarizing the diversity of data within a subset of columns, providing an immediate metric for data quality and distribution analysis.

Advanced Context: Handling Missing Data (NaNs) and Data Types

When dealing with real-world data, the presence of missing values (represented as `NaN` or Not a Number) is common and requires careful consideration. The behavior of `pd.unique()` regarding `NaN` values depends slightly on the data type of the underlying Pandas Series or array being processed.

For most numerical and object (string) arrays, `pd.unique()` will treat `NaN` as a single unique value. If `NaN` appears multiple times across the columns being analyzed, it will only be included once in the final output array, consistent with the function's core purpose of identifying distinct elements. This is usually the desired behavior, as analysts generally consider all missing entries to belong to one category of 'missingness'.

However, if the data types of the combined columns are heterogeneous or if we were using a different method like `df.unique()` on an integer-based Series (which often converts to float when `NaN` is introduced), understanding the precise representation of missing data in the resulting NumPy array is vital. When flattening multiple columns using `.values.ravel()`, Pandas might

coerce the data type to a common denominator (often `object` for mixed strings and numbers, or `float` for mixed numerical types), ensuring consistent handling of `NaN`.

If the user needs to explicitly exclude `NaN` values from the unique count, they should implement a pre-filtering step before applying `.values.ravel()`. For example, ensuring that the selected DataFrame subset is filtered to drop rows or cells containing `NaN` values would be necessary. A simple approach might involve using `.dropna()` on the flattened Series before calling `unique()`, though this requires converting the array back to a Series temporarily, adding a small complexity layer to the process.

Comparing `unique/ravel` vs. `drop_duplicates`

It is important to clearly distinguish the purpose of the `unique()`/`ravel()` combination from the more commonly used `DataFrame.drop_duplicates()` method. Although both deal with uniqueness, their scopes and outputs are fundamentally different, leading to confusion if the user's ultimate goal is not clearly defined.

The `DataFrame.drop_duplicates()` method operates horizontally, focusing on identifying and retaining only one instance of a duplicated *row* based on the values in one or more specified columns. The output is always a Pandas DataFrame containing full rows. It is essential for eliminating redundant observations or records in the dataset.

In contrast, the `pd.unique(df.values.ravel())` approach operates vertically across the cells of the specified columns. It disregards the row structure entirely, treating the cells as a long list of individual data points. The output is a one-dimensional array containing all distinct *cell values*. This method is critical for dictionary creation, categorizing features, or building look-up tables where the full lexicon of terms across multiple columns is needed.

A key difference can be illustrated with our example DataFrame:
If we ran `df.drop_duplicates(subset=)`, since all pairs (a, a), (b, c), (c, e), (d, f), and (e, g) are themselves unique, the resulting DataFrame would still have 5 rows. The method confirms that no *rows* are duplicates based on those two columns. Conversely, our `unique/ravel` method identifies that individual values like 'a', 'c', and 'e' appear more than once overall in the column subset, yielding only 7 unique elements from the 10 total cells examined.

Conclusion and Further Exploration

Finding unique values across multiple columns in Pandas requires a specific sequence of operations: column selection, conversion to a NumPy array, flattening using `.ravel()`, and final deduplication via `pd.unique()`. This robust methodology allows analysts to consolidate distinct elements from heterogeneous column sources into a single, comprehensive list, whether for

enumeration, categorization, or calculation.

This technique provides three primary outputs useful for data exploration: the raw NumPy array of unique values, a structured Pandas DataFrame representation, and a simple integer count of the distinct elements. Mastering this combination is a fundamental skill for efficient data manipulation in Python and Pandas.

For those looking to deepen their Pandas expertise, consider exploring related functionalities that involve multi-column operations, such as:

How to Merge Pandas DataFrames on Multiple Columns: Understanding how to join datasets based on complex multi-key relationships.
How to Filter a Pandas DataFrame on Multiple Conditions: Learning advanced techniques for conditional subset selection across various columns simultaneously.