How to Easily Remove Duplicate Rows in Pandas Based on Multiple Columns

Authored by

stats writer

December 1, 2025

Dealing with redundant entries is a fundamental step in effective data cleaning and analysis. When working with large datasets in the Pandas library, duplicate rows can severely skew statistical results and slow down processing. Fortunately, Pandas provides a highly efficient and versatile tool for this purpose: the `DataFrame.drop_duplicates()` function.

This function is central to maintaining DataFrame integrity. While its default behavior is to examine all columns for exact matches, its true power lies in the optional `subset` argument. By defining a list of column names for the `subset`, you instruct Pandas to only consider values within those specified columns when searching for duplicate entries. This targeted approach is essential when certain columns contain unique identifiers (like timestamps or specific record IDs) but the core data points (like user attributes or transaction details) are what truly define a duplicate record.

The flexibility of `drop_duplicates()` is further enhanced by parameters such as `inplace`, which dictates whether the operation modifies the original DataFrame directly or returns a new, cleaned version. Understanding how to leverage these parameters allows data practitioners to perform precise and non-destructive data manipulation, ensuring the dataset is optimized for subsequent analysis.

You can use the following methods to drop duplicate rows across multiple columns in a pandas DataFrame:

Method 1: Drop Duplicates Across All Columns

This is the default application of the function, where no arguments are passed. Pandas evaluates every single column in the row to determine if a duplicate exists, offering the most conservative approach to data removal.

df.drop_duplicates()

Method 2: Drop Duplicates Across Specific Columns

By utilizing the `subset` argument, you can provide a list of column names (e.g., `'column1'` and `'column3'`) to restrict the duplicate check to only those fields. This is critical for scenarios where differences in non-essential columns, like unique index identifiers or timestamps, should not prevent duplicate identification.

df.drop_duplicates()

The following examples show how to use each method in practice with the following pandas

DataFrame:

import pandas as pd

#create DataFrame representing sample sales data
df = pd.DataFrame({'region': ,
'store': ,
'sales': })

#view the initial DataFrame structure
print(df)

region store sales
0 East 1 5
1 East 1 5
2 East 2 7
3 West 1 9
4 West 2 12
5 West 2 8

Understanding the DataFrame.drop_duplicates() Method

The `DataFrame.drop_duplicates()` method is the cornerstone of row-level duplicate detection in Pandas. When invoked without any arguments, it performs a comprehensive comparison, requiring that values in **every single column** match exactly for two rows to be considered duplicates. This conservative approach guarantees that no unique rows are accidentally removed, making it the safest default operation.

However, real-world data often requires a more nuanced approach. Imagine a scenario where you have multiple entries for the same entity (identified by core identifiers) on the same date, but perhaps a supplementary, non-critical column differs. If you only care about the combination of the identifying columns, checking all columns would incorrectly preserve those rows. This is where the parameters, particularly `subset`, become indispensable tools for fine-tuning the definition of redundancy.

It is vital to remember the default retention behavior. By default, `drop_duplicates()` keeps the first occurrence of a duplicate set (`keep='first'`). This behavior is usually desirable as it often retains the oldest record chronologically. However, if the latest update is preferred, modifying the `keep` parameter is necessary, providing precise control over data preservation.

Key Parameters of drop_duplicates(): Subset, Keep, and Inplace

Mastering the use of `drop_duplicates()` hinges upon a thorough understanding of its primary control arguments. These three parameters--`subset`, `keep`, and `inplace`--define the scope, retention policy, and resulting output of the operation.

The **`subset`** parameter accepts a single column name or, more commonly, a list of column names. When this argument is provided, Pandas restricts its search for identical values exclusively to the columns listed in the array. This is the mechanism used for dropping duplicates across multiple, specific columns rather than the entire DataFrame. For example, setting `subset=` ensures that a duplicate is identified only if both the Customer ID and the Transaction Date match in two different rows, regardless of differences in other fields like `Shipping_Address` or `Quantity`.

The **`keep`** parameter manages which instance of a duplicate set is preserved. It accepts three possible string values: `'first'` (default, keeps the first encountered row), `'last'` (keeps the last encountered row), or `False` (drops all duplicate instances, leaving only truly unique rows). Using `keep=False` is particularly powerful when you need to isolate observations that have no redundancy whatsoever, effectively discarding any row that is part of a duplication group, even the initial occurrence.

Finally, the boolean **`inplace`** argument (often set to `True`) determines if the operation should modify the existing DataFrame object directly. If `inplace=False` (the default), the function returns a new, cleaned DataFrame, leaving the original data untouched. While operating in place can save memory for extremely large datasets, returning a new DataFrame is often the safer choice, particularly during interactive analysis or when the original data integrity must be preserved for subsequent comparisons or auditing.

Example 1: Drop Duplicates Across All Columns

The following code shows how to drop rows that have duplicate values across all columns using the default behavior of the `drop_duplicates()` function. This operation is essential for removing exact, accidental copies of records that pollute the dataset.

#drop rows that have duplicate values across all columns
df.drop_duplicates()

region store sales
0 East 1 5
2 East 2 7
3 West 1 9
4 West 2 12

5 West 2 8

The row in index position 1 had the exact same values across all columns as the row in index position 0 (East, 1, 5), so it was dropped from the resulting DataFrame. The remaining rows are preserved because although some column values match (e.g., store 2 appears twice), the full combination of the three columns is unique.

By default, pandas keeps the first duplicate row (`keep='first'`). However, you can use the **`keep`** argument to specify to keep the last duplicate row instead, which is useful when dealing with updated records or logs where the latter entry holds precedence:

#drop rows that have duplicate values across all columns (keep last duplicate)
df.drop_duplicates(keep='last')

region store sales
1 East 1 5
2 East 2 7
3 West 1 9
4 West 2 12
5 West 2 8

When `keep='last'` is applied, the row at index 0 is dropped, and the row at index 1 is retained. This demonstrates the precise control that the `keep` parameter provides over which record of a duplicate set is ultimately preserved in the data cleansing process.

Example 2: Drop Duplicates Across Specific Columns

This strategy is employed when you need to define redundancy based only on a select group of columns, ignoring variations in supplementary data fields. We will use the `subset` argument to drop rows that have duplicate values across only the **`region`** and **`store`** columns.

In this context, the combination ('West', 2) appears in two rows (index 4 and index 5) but with different sales figures (12 and 8). Since we are restricting the check to just `region` and `store`, Pandas will identify these two rows as duplicates and remove the second one (index 5) by default (`keep='first'`).

#drop rows that have duplicate values across region and store columns
df.drop_duplicates()

region store sales
0 East 1 5
2 East 2 7
3 West 1 9
4 West 2 12

A total of two rows were dropped from the DataFrame: index 1 (due to the (East, 1) duplication found at index 0) and index 5 (due to the (West, 2) duplication found at index 4). This confirms the operation successfully ignored the `sales` column when determining the unique store locations.

Performance and Best Practices in Data Cleansing

While `drop_duplicates()` is highly optimized for Python data manipulation, performance considerations are critical when dealing with truly massive datasets containing tens of millions of records. The function works by comparing hash values of rows or row subsets. The number of columns included in the comparison significantly affects processing time and memory usage.

For very wide datasets (those with hundreds of columns), relying on the default behavior (checking all columns) can be substantially slower than defining a narrow `subset` based only on primary keys or identifiers. By restricting the check to only the 3 or 4 essential identifier columns, you drastically reduce the computational overhead required for hashing and comparison, leading to quicker execution times.

A standard best practice is always to confirm the removal count. After applying `drop_duplicates()`, compare the resulting DataFrame length to the original. If the number of removed rows is unexpectedly high or low, it often indicates either an error in defining the `subset` or a fundamental misunderstanding of the underlying data structure. Furthermore, for critical operations, always use the `inplace=False` default behavior first to inspect the resulting DataFrame before permanently modifying the original data.

Conclusion: Ensuring Data Integrity in Python

Effective data manipulation in Python, particularly within the Pandas ecosystem, relies heavily on robust methods for data cleaning. The `DataFrame.drop_duplicates()` function provides the necessary precision and flexibility to handle complex redundancy scenarios, whether you require a full-row comparison or a targeted check across specific column subsets.

By judiciously applying the `subset`, `keep`, and `inplace` parameters, analysts gain complete control over how duplicate records are defined, identified, and removed. Mastering this function is an essential skill for anyone performing serious data preparation, ensuring the resulting dataset is

clean, accurate, and optimized for subsequent advanced analysis and modeling.