How to Drop Duplicate Rows in a Pandas DataFrame

Authored by

stats writer

December 15, 2025

One of the most frequent and critical tasks in the field of Data cleaning is the identification and removal of redundant or replicate entries. Duplicate records can severely skew statistical analysis, inflate data volumes, and lead to erroneous conclusions. Fortunately, the Pandas library, an essential tool for data manipulation in Python, provides a highly optimized and intuitive method for handling this issue: the **DataFrame.drop_duplicates()** function. Mastering this function is foundational for anyone working with structured data.

The core purpose of the DataFrame.drop_duplicates() method is to streamline your data by examining rows and determining uniqueness based on values across specified columns. This functionality is crucial for maintaining data integrity, ensuring that each observation unit is represented only once according to the defined criteria. When executed, this method returns a new DataFrame where all rows identified as duplicates have been systematically excluded. By leveraging key parameters such as `subset`, `keep`, and `inplace`, practitioners gain fine-grained control over the deduplication process, allowing them to tailor the removal strategy precisely to the needs of their specific dataset.

Understanding the Pandas drop_duplicates() Method

The DataFrame.drop_duplicates() method is exceptionally flexible, allowing users to define exactly what constitutes a duplicate. By default, it considers two rows duplicates if the values in **all** corresponding columns are identical. However, in many real-world scenarios, a duplicate might only be defined by the repetition of values within a specific subset of identifier columns, such as an ID and a timestamp, while ignoring auxiliary data columns. Understanding the syntax and the role of each parameter is key to applying this powerful tool effectively.

The standard syntax for the function is presented below, illustrating the three primary parameters that govern its behavior. These parameters allow the user to control the scope of the search, the retention policy for identified duplicates, and whether the modification should occur directly on the existing object or return a new copy.

The most efficient way to achieve deduplication within a Pandas DataFrame is by utilizing the `drop_duplicates()` method, which follows this functional signature:

df.drop_duplicates(subset=None, keep='first', inplace=False)

The parameters operate as follows:

subset: This parameter accepts a column label or a list of column labels. It dictates which columns should be considered when searching for duplicate rows. If `subset` is set to `None` (the default), all columns in the DataFrame are used to determine uniqueness. Specifying a subset dramatically changes the definition of a duplicate, narrowing the focus to only those key identifying columns.

keep: This critical parameter determines which, if any, of the duplicate rows should be retained in the resulting DataFrame. It allows for precision in managing sequences of duplicate entries.

'first': This is the default setting. It instructs the function to delete all duplicate rows, preserving only the very first occurrence found. This is common when sequential data entry errors are suspected.
'last': This setting instructs the function to delete all duplicate rows, preserving only the last occurrence found. This is often useful when dealing with data that is updated over time, where the most recent entry (the last one encountered) is considered the most accurate.
False: Setting `keep` to `False` results in the removal of all rows that are duplicates of another row--meaning if a row appears more than once, all instances of that row will be dropped. The final DataFrame will only contain rows that are uniquely present in the original dataset.

inplace: A Boolean parameter (`True` or `False`) that dictates whether the operation modifies the original DataFrame directly (`True`) or returns a modified copy (`False`, the default). Using `inplace=True` is memory efficient but irreversible, whereas returning a copy preserves the original data structure.

This tutorial provides several examples of how to use this function in practice on the following DataFrame, which contains intentional duplicate entries:

Setting Up the Environment: Creating the Sample DataFrame

To demonstrate the practical application of the `drop_duplicates()` method, we will utilize a small, illustrative DataFrame. This dataset contains clear examples of duplicates across all columns, as well as duplicates specific to certain column combinations. Observing the output after applying different parameters to this sample data is the clearest way to grasp the method's functionality.

We begin by importing the Pandas library and constructing a DataFrame named `df` that tracks team performance metrics, including team identifier, points scored, and assists made. Notice the intentional redundancy in rows 1/2 and 3/4. This will serve as the testing ground for our deduplication exercises.

import pandas as pd

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

#display DataFrame

print(df)

team points assists
0 a 3 8
1 b 7 6
2 b 7 7
3 c 8 9
4 c 8 9
5 d 9 3

In this starting DataFrame, we can identify two sets of potential duplicates. Rows 3 and 4 are identical across all three columns (team 'c' has 8 points and 9 assists). Rows 1 and 2 share the same team ('b') and points (7), but differ in their 'assists' count (6 vs 7). This distinction is critical as we move to examples using either all columns or specific column subsets for comparison.

Example 1: Removing Duplicates Across All Columns (Default Behavior)

The most common application of `drop_duplicates()` is removing rows that are exact copies of another row across every available column. When the `subset` parameter is omitted, the function performs an exhaustive row-by-row comparison. This ensures that only truly unique records remain in the resulting structure. By default, the function uses `keep='first'`, meaning that if a set of identical rows is found, only the first encountered row is retained, and all subsequent copies are discarded.

Applying the function without any arguments effectively executes the command `df.drop_duplicates(subset=None, keep='first')`. In our sample data, rows 3 and 4 (team 'c', 8 points, 9 assists) are exact duplicates. Since row 3 is the first occurrence, row 4 will be removed, resulting in a cleaner dataset. Rows 1 and 2 are retained because, although they share 'team' and 'points', the 'assists' column differs, meaning they are not considered full-row duplicates.

df.drop_duplicates()

team points assists
0 a 3 8
1 b 7 6
2 b 7 7
3 c 8 9
5 d 9 3

As the output demonstrates, row index 4, which duplicated row index 3, has been successfully

eliminated. The resulting DataFrame is five rows long, containing only those records that are unique when considering the entire set of features. This default behavior is generally the safest starting point for any deduplication effort.

Example 2: Retention Policy Management (keep='last' and keep=False)

While retaining the first occurrence is standard, data analysis frequently requires different retention strategies. The `keep` parameter allows us to specify whether we prefer to keep the last instance of a duplicate or, in some cases, discard all instances of a duplicated row entirely. These options provide necessary flexibility when dealing with noisy or evolving data streams.

Deleting All Duplicates Entirely (keep=False)

A more stringent approach is required when the presence of any duplicate entry signals a corrupted or questionable record, making all instances of that duplicate unusable. By setting `keep=False`, the function ensures that any row that appears more than once is completely removed from the DataFrame. This guarantees that every remaining row is absolutely unique across the entire original dataset.

Applying `keep=False` to our DataFrame will identify rows 3 and 4 as duplicates and remove both. The resulting structure will only contain rows that had no exact match elsewhere. This yields the smallest possible dataset while maximizing confidence in the uniqueness of the remaining observations. This contrasts sharply with `keep='first'` or `keep='last'`, which always retain at least one representative of a duplicated set.

df.drop_duplicates(keep=False)

team points assists
0 a 3 8
1 b 7 6
2 b 7 7
5 d 9 3

As illustrated, both indices 3 and 4 have been dropped because they were copies of one another. The final DataFrame now consists of four perfectly unique rows. This setting is highly effective when aiming to isolate and analyze only those records that represent singular events or observations.

Example 3: Targeting Duplicates in a Specific Column Subset

In many complex datasets, defining uniqueness requires focusing only on primary identifier

columns, ignoring variability in secondary or auxiliary columns. The `subset` parameter allows for this focused definition of duplication. For instance, if we define a duplicate based solely on the combination of 'team' and 'points', then rows 1 and 2, as well as rows 3 and 4, all become duplicates. By using `subset=`, we tell Pandas to ignore the 'assists' column entirely during the comparison process.

The following code executes this subset operation, maintaining the default `keep='first'` policy. This means that among the identified duplicates (rows 1/2 and 3/4), only the first occurrence of each set will be kept. Row 1 is retained over row 2, and row 3 is retained over row 4.

df.drop_duplicates(subset=)

team points assists
0 a 3 8
1 b 7 6
3 c 8 9
5 d 9 3

The resulting DataFrame now confirms the removal of rows 2 and 4. Critically, observe row 1 (team 'b', points 7, assists 6). This row was retained, even though row 2 had a different assist count, because the decision to drop was only based on the 'team' and 'points' columns. This targeted approach is invaluable for managing data where only certain fields define the uniqueness constraint.

Example 4: Combining Subset Selection with Strict Removal (keep=False)

Combining the power of the `subset` parameter with the strict removal policy of `keep=False` allows for the elimination of all records that share a common identifier subset. This is useful if you want to flag or remove any records where the critical key combination appears more than once, regardless of which instance was first or last.

If we apply `subset=` and `keep=False` to our DataFrame, the function identifies teams 'b' (rows 1 and 2) and 'c' (rows 3 and 4) as sets of duplicates based on the key columns. Because `keep=False` is set, all four of these rows are removed, leaving only the truly unique combinations.

df.drop_duplicates(subset=, keep=False)

team points assists
0 a 3 8
5 d 9 3

The resulting DataFrame is drastically reduced, containing only teams 'a' and 'd'. These two teams had unique combinations of 'team' and 'points' within the entire original dataset. Teams 'b' and 'c' were both entirely excluded because their key combination appeared multiple times. This method is highly effective for isolating unique, non-contested records within a larger structure.

Conclusion: Best Practices for Data Cleaning and Deduplication

The DataFrame.drop_duplicates() method is an indispensable component of any robust data preparation pipeline using Pandas. Its flexibility, driven by the `subset`, `keep`, and `inplace` parameters, ensures that developers and data scientists can tailor the deduplication process precisely to the specific requirements of the dataset and the analytical goals.

When applying this function, it is generally best practice to first run the operation using `inplace=False` (the default) to inspect the resulting DataFrame and confirm that the intended records were dropped. Only once the strategy (defined by `subset` and `keep`) is validated should the operation be executed with `inplace=True` for permanent modification or assignment to a new variable. Careful definition of the duplicate criteria using the `subset` parameter prevents accidental data loss and ensures that only truly redundant records are eliminated, thereby preserving the integrity and statistical power of the cleaned dataset.

Mastering this function is a significant step toward achieving high-quality Data cleaning, which underpins the reliability of all subsequent analysis and modeling efforts. For further exploration of data manipulation techniques in Pandas, particularly regarding structural redundancy, consider reviewing related operations.

How to Drop Duplicate Columns in Pandas