Drop Duplicate Rows in a Pandas DataFrame

Authored by

Mohammed looti

November 6, 2025

Introduction: The Necessity of Handling Duplicates in Data Science

Data cleaning is arguably the most critical step in any data analysis workflow. One frequent challenge analysts face is identifying and removing duplicate records from their datasets. Duplicate rows can skew statistical results, lead to inaccurate model training, and generally compromise the integrity of the analysis. Fortunately, the Pandas library provides a highly efficient and intuitive method for addressing this issue: the `drop_duplicates()` function.

Mastering this function is essential for anyone working with tabular data in Python. It offers flexibility, allowing users to define exactly what constitutes a duplicate--whether it's an exact match across all columns or a specific match within a subset of columns. This guide will serve as a comprehensive tutorial, detailing the syntax, parameters, and practical applications of using `drop_duplicates()` on a DataFrame.

Understanding the drop_duplicates() Method

The primary tool for managing redundant entries in a DataFrame is the `drop_duplicates()` method. Its structure is designed to be versatile, enabling fine-grained control over the deduplication process. Understanding the key parameters is vital for effective data manipulation.

The standard syntax for invoking this powerful method is straightforward:

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

This function takes three main optional arguments that dictate how the removal process is executed. Let us examine the purpose of each parameter in detail:

subset: This parameter defines which columns the function should consider when identifying duplicate rows. By default, if `subset=None`, the function considers all columns in the DataFrame for the comparison. Providing a list of column names restricts the check only to those specified columns.
keep: This is a crucial argument that determines which duplicate observation, if any, should be retained. The default behavior is usually appropriate, but various scenarios require different retention strategies.

'first': This is the default setting. It instructs the function to delete all duplicate rows, preserving only the first occurrence encountered in the DataFrame.
'last': This option deletes all duplicate rows, preserving only the last occurrence encountered.
False: Setting this to False results in the deletion of all rows that are duplicates (meaning, if a row is duplicated anywhere else in the dataset, both instances are removed).

inplace: This boolean parameter controls whether the operation modifies the original DataFrame

directly. If set to **True**, the operation is performed in place, and nothing is returned. If set to **False** (the default), a new copy of the DataFrame with duplicates removed is returned, leaving the original data untouched.

Setting Up the Sample DataFrame

To illustrate how the `drop_duplicates()` function works in practice, we will use a small sample DataFrame representing sports team statistics. This dataset contains several intentional duplicate entries that we will target and remove throughout our examples. The setup involves importing the necessary Pandas library and defining the data structure.

Note the structure of the data: rows 1 and 2 share the same `team` and `points` values, but differ in `assists`. Rows 3 and 4 are exact duplicates across all columns.

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

The index labels (0 through 5) are critical when determining which row is considered the 'first' or 'last' duplicate, as Pandas relies on the row order.

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

When the `subset` parameter is omitted, `drop_duplicates()` performs a comprehensive check, requiring that every single column value matches across two or more rows for them to be considered duplicates. In our sample data, only rows 3 and 4 are exact matches.

Using the function without any parameters defaults to `keep='first'` and `inplace=False`. This means it removes row 4 (the second occurrence) while preserving row 3 (the first occurrence), returning a new DataFrame.

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

Notice that row 4, which duplicated row 3, has been successfully removed. Rows 1 and 2 were retained because, although they share the same 'team' and 'points', the 'assists' column differs, meaning they are not considered full-row duplicates.

Example 2: Controlling Retention with the keep Parameter

While preserving the first instance is the most common approach, there are scenarios where analysts might prefer to keep the last recorded entry or remove all instances of a duplicate. The `keep` parameter provides this flexibility.

Retaining the Last Occurrence

If we want to prioritize the most recent data entry, we can set `keep='last'`. In the case of rows 3 and 4, this option will retain row 4 and delete row 3, effectively keeping the highest index value among the duplicate group.

df.drop_duplicates(keep='last')

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

Removing All Duplicate Instances

A more stringent requirement is to remove any row that participates in a duplication event. This is

useful when data redundancy is considered noise, and only unique, single-occurrence observations should remain. Setting `keep=False` achieves this goal.

When using `keep=False`, both rows 3 and 4 are removed because they are duplicates of each other. The remaining rows are entirely unique across all three columns.

df.drop_duplicates(keep=False)

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

Example 3: Targeting Duplicates Based on a Subset of Columns

Often, two rows may contain different data in certain fields (like a unique ID or a timestamp) but represent the same entity based on core identifying features. For instance, we might only care if the combination of `team` and `points` is duplicated, regardless of the `assists` value.

To implement this targeted deduplication, we use the `subset` parameter, providing it with a list of column names. In this scenario, we check for duplicates based only on the `team` and `points` columns. Rows 1 and 2 now qualify as duplicates (Team 'b', 7 points), and rows 3 and 4 also qualify (Team 'c', 8 points).

Using the default `keep='first'` setting, the function retains row 1 and row 3, discarding rows 2 and 4, which were the subsequent occurrences of those specific `(team, points)` pairs.

df.drop_duplicates(subset=)

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

This approach is extremely useful when dealing with transactional data where multiple records might relate to the same physical event or entity, and we must define a unique key based on a subset of available fields.

Controlling Output: The Importance of the inplace Parameter

The Pandas library typically operates immutably, meaning functions return a new object rather than modifying the original data structure. This is the default behavior of `drop_duplicates()` (where `inplace=False`).

For large datasets, creating copies of the DataFrame repeatedly can consume significant memory resources. When memory efficiency is critical and the analyst is certain they no longer need the original dataset containing the duplicates, setting `inplace=True` is the preferred method.

When `inplace=True`, the function executes the removal directly on the calling DataFrame (`df` in our examples) and returns `None`. This is generally discouraged for beginners but is highly optimized for performance in professional data pipelines.

For example, to permanently remove duplicates (based on the default 'first' keeping strategy) from the original `df`, the syntax would be:

```
df.drop_duplicates(inplace=True)
```

Always exercise caution when using `inplace=True`, as the change is irreversible without reloading the data.

Conclusion and Further Resources

The `drop_duplicates()` method is a cornerstone of data cleaning in Pandas. By leveraging the `subset` and `keep` parameters, analysts gain precise control over identifying and resolving data redundancy, ensuring that subsequent analyses are based on clean, reliable data.

Whether you need to remove exact duplicates across all fields or target specific key combinations, this function provides the necessary flexibility. Always remember to consider the implications of `inplace=True` versus creating a new DataFrame copy.

Additional Resources

How to Drop Duplicate Columns in Pandas