How to Easily Remove Duplicate Rows in Pandas and Keep the Latest Entry

Authored by

stats writer

November 21, 2025

Managing data quality often involves handling duplicate records. When working with tabular data structures like the DataFrame in the Pandas library, removing duplicates is a common necessity. However, a frequent requirement is not merely to delete redundant entries, but to ensure that the most relevant or "latest" record is preserved. This article details the precise methodology required to drop duplicate rows in Pandas while explicitly keeping the row associated with the most recent timestamp.

The key insight to solving this problem lies in understanding that the powerful drop_duplicates() method processes rows based on their order within the DataFrame. By default, it keeps the first occurrence it encounters. To override this default behavior and keep the latest record instead, we must first manipulate the order of the data. This preparatory step involves sorting the entire DataFrame based on the time or sequence column, ensuring that the latest entry for any specific item appears last in the sorted sequence.

Once the data is correctly ordered, the subsequent application of the drop_duplicates() method, combined with the `keep='last'` parameter, guarantees the desired outcome. This sequence of operations--sorting by time, followed by targeted removal of duplicates--is the robust and idiomatic approach favored by professional data analysts using Pandas. We will explore the precise syntax and walk through a complete practical example demonstrating this technique.

Understanding the Necessity of Sorting Data

When dealing with records that evolve over time--such as sensor readings, transaction logs, or versioned product data--it is crucial to retain the most current state. Simple duplicate removal, which often keeps the first record encountered, might inadvertently discard valuable, updated information. For instance, if a user updates their profile twice, we need to ensure the most recent profile entry (identified by its creation timestamp) is the one that remains in the clean dataset.

The default behavior of the drop_duplicates() function is to use the `keep='first'` argument. This means that if multiple rows share the same values in the specified subset of columns, the row that appears earliest in the index or sequence is retained, and all subsequent duplicates are discarded. If our data is unsorted, there is no guarantee that the earliest row encountered is actually the oldest or the newest record; it is purely arbitrary based on the insertion order or initial load sequence.

To impose chronological logic on this process, we must leverage the sort_values() function. This function allows us to reorder the entire DataFrame based on a specific column, typically a datetime column. By sorting in ascending order by the time column (the default behavior of sort_values()), we ensure that for any group of duplicates, the chronologically latest record is positioned at the end of that group. This setup is fundamental to utilizing the `keep='last'` parameter effectively.

The Core Syntax for Keeping the Latest Record

The combined method requires chaining two distinct operations: first, sorting the data based on time, and second, dropping duplicates while specifying the preservation of the last encountered entry. This methodology ensures that the latest version of any duplicate set is prioritized for retention. The columns used for identifying duplication (e.g., product ID, user ID) are separate from the column used for chronological sorting (e.g., the `time` column).

The primary goal is to retain rows that are unique based on a specific identifying column (like `item`), but among the duplicates of that item, we want to keep the one with the highest value in the sequencing column (like `time`). The generic Pandas syntax to achieve this is remarkably compact and powerful:

df = df.sort_values('time').drop_duplicates(, keep='last')

In this crucial single line of code, the DataFrame `df` is first sorted using the sort_values() method based on the `time` column. Subsequently, the drop_duplicates() method is applied. It looks for duplication only within the columns specified in the list argument (here,). Because the data is sorted chronologically and `keep='last'` is specified, Pandas retains the final occurrence of each unique `item`, which corresponds precisely to the entry with the latest timestamp.

Deep Dive into the `drop_duplicates()` Parameters

Understanding the parameters of the drop_duplicates() function is essential for precise data manipulation. This function accepts several key arguments that control how duplication is defined and which rows are preserved. The two most relevant parameters in this context are `subset` and `keep`.

The `subset` parameter is mandatory for selective duplicate removal. It takes a list of column names (e.g.,). Pandas defines a duplicate row only if its values match those of a previous row across all columns listed in the `subset`. If this parameter is omitted, Pandas considers a row duplicated only if all values across **all** columns match, which is rarely the requirement when attempting to keep the latest version of a specific entity.

The `keep` parameter dictates which of the duplicate rows should be retained. It accepts three string values: `'first'` (the default), `'last'`, or `'False'`. Setting `keep='last'` instructs Pandas to preserve the last row encountered for any group of duplicates and discard all preceding instances. This is precisely what allows us to retain the latest record, provided the DataFrame has been previously sorted by time. Using `keep='False'`, conversely, removes all duplicates entirely, leaving only truly unique rows (i.e., rows that had no matching counterpart).

Practical Example Setup: Initializing the Data

To illustrate this process, consider a scenario involving a grocery store's sales logs. These logs track the quantity of various fruits sold over several days, often containing multiple entries for the same item if sales occurred at different times. Our objective is to consolidate this data, keeping only the record of the most recent sale for each item.

We begin by creating a sample DataFrame using the Pandas library. Note that the `time` column is crucial here, as it contains the chronological information necessary for correct sorting and identification of the "latest" record. It is also imperative that this column is converted to the appropriate `datetime` dtype for accurate temporal comparison.

The initial structure of the data, including the necessary import and conversion steps, is shown below. Notice the duplicate entries for 'apple' (rows 0 and 2) and 'mango' (rows 3 and 4), where the `sales` value may or may not change, but the `time` certainly does.

import pandas as pd

#create DataFrame
df = pd.DataFrame({'time': ,
'item': ,
'sales': })

#convert time column to datetime dtype
df = pd.to_datetime(df)

#view DataFrame
print(df)

time item sales
0 2022-10-25 04:00:00 apple 18
1 2022-10-25 11:55:12 orange 22
2 2022-10-26 02:00:00 apple 19
3 2022-10-27 10:30:00 mango 14
4 2022-10-27 14:25:00 mango 14
5 2022-10-28 01:15:27 kiwi 11

Executing the Combined Operation

Now that we have established our initial DataFrame, the next step is to apply the dual operation of sorting and dropping. We aim to remove all rows that share an `item` value with another row,

except for the one associated with the most recent timestamp.

The first part of the chain, `df.sort_values('time')`, orders the DataFrame chronologically from oldest to newest. This rearrangement is crucial because it ensures that for the 'apple' records, the row with the timestamp '2022-10-26 02:00:00' comes after '2022-10-25 04:00:00'. Similarly, for 'mango', the row '2022-10-27 14:25:00' is placed after '2022-10-27 10:30:00'.

The second part, `.drop_duplicates(, keep='last')`, then iterates through the sorted DataFrame. When it encounters the duplicate 'apple' entries, it discards the earlier one and retains the later one due to the `keep='last'` setting. The results demonstrate that only the latest entry for 'apple' (row index 2) and 'mango' (row index 4) remains, along with the unique entries for 'orange' and 'kiwi'.

#drop duplicate rows based on value in 'item' column but keep latest timestamp
df = df.sort_values('time').drop_duplicates(, keep='last')

#view updated DataFrame
print(df)

time item sales
1 2022-10-25 11:55:12 orange 22
2 2022-10-26 02:00:00 apple 19
4 2022-10-27 14:25:00 mango 14
5 2022-10-28 01:15:27 kiwi 11

Handling Multiple Columns for Duplication

While the example above used only the `item` column to define duplication, real-world data often requires defining uniqueness based on a combination of several fields. For instance, if we were tracking sales by both `item` and `store_location`, a row would only be considered a duplicate if both the item name and the store location matched across multiple records.

If you need to define duplication across multiple fields, you simply expand the list provided to the `subset` argument of the drop_duplicates() function. The sorting step remains the same--it should always target the chronological column to establish the order of "latestness."

For example, to find the latest record that is duplicated across `item` and `region`, the syntax would be: `df.sort_values('time').drop_duplicates(, keep='last')`. This flexibility allows users to handle complex hierarchical duplicate resolution within the Pandas framework, ensuring that the latest transactional state for every unique combination of identifying variables is preserved.

Best Practices and Performance Considerations

When implementing this duplicate management strategy, especially on very large DataFrames (millions of rows), performance becomes a key concern. Sorting is an O(N log N) operation, meaning it can be computationally intensive. Therefore, it is important to ensure that the timestamp column is correctly cast to the `datetime` dtype before sorting, as string sorting is often slower and less reliable chronologically.

Furthermore, consider if the sorting needs to be applied to the entire DataFrame. If the data is already partially sorted or if the duplicates are concentrated in a specific subset of the data, filtering the DataFrame before the operation might yield efficiency benefits. Always verify the results against a small sample of known duplicate groups to confirm that the `keep='last'` argument is functioning as intended after the preceding sort operation.

This method is highly reliable because it leverages the inherent ordering mechanism of Pandas. By explicitly linking the definition of "latest" (via the sort) to the removal mechanism (via `keep='last'`), data integrity is maintained, ensuring that only the most relevant, up-to-date records remain for subsequent analysis.