How do you drop rows with NaN values in pandas

Authored by

stats writer

December 24, 2025

Handling missing data is a critical step in the data cleaning and preprocessing pipeline. In the pandas library, which is fundamental for data manipulation in Python, missing values are typically represented by **NaN** (Not a Number). When preparing a dataset for analysis or machine learning, it is often necessary to remove records that contain these missing values, as they can interfere with statistical calculations and model training.

Fortunately, pandas provides an extremely versatile and powerful function for this exact purpose: the **DataFrame.dropna()** method. This method allows analysts to efficiently discard rows or columns that contain missing data, providing various parameters to control the granularity of the removal process. Understanding how to deploy these parameters--such as `axis`, `how`, and `thresh`--is essential for cleaning data accurately without inadvertently losing important information.

The default behavior of **dropna()** is designed to be cautious, dropping any row that contains at least one **NaN** value. However, real-world data often requires more nuanced handling. This comprehensive tutorial will delve into the specific methods available within `dropna()`, demonstrating how to selectively remove rows based on the extent and location of missing data, ensuring your **DataFrame** is clean and ready for subsequent analytical tasks.

Preparing Our Sample Data for Demonstration

Before we explore the intricacies of the `dropna()` function, we must establish a sample **DataFrame** containing various instances of missing data. This dataset will serve as the foundation for all subsequent examples, clearly illustrating the effect of each parameter combination. The data below simulates a common scenario where certain observations have incomplete records across different columns, representing player statistics where some metrics might be unavailable.

We utilize the powerful **NumPy** library to explicitly define missing values as `np.nan` when constructing our demonstration **DataFrame**. This practice is standard when handling missing data in Python environments. Note the structure, particularly the first row (index 0) which contains multiple missing values ('rating' and 'points'), and the fourth row (index 3) which has one missing value in the 'assists' column.

import numpy as np
import scipy.stats as stats

#create DataFrame with some NaN values
df = pd.DataFrame({'rating': ,
'points': ,
'assists': ,
'rebounds': })

#view DataFrame
df

rating points assists rebounds
0 NaN NaN 5.0 11
1 85.0 25.0 7.0 8
2 NaN 14.0 7.0 10
3 88.0 16.0 NaN 6
4 94.0 27.0 5.0 6
5 90.0 20.0 7.0 9
6 76.0 12.0 6.0 6
7 75.0 15.0 9.0 10
8 87.0 14.0 9.0 10
9 86.0 19.0 5.0 7

Example 1: Eliminating Rows with Any NaN Value

The most straightforward application of the `dropna()` method involves removing any row (observation) that contains even a single missing value. This behavior is the default setting when no parameters are passed to the function, relying on `how='any'` implicitly. This is the strictest cleaning operation, prioritizing complete records above all else. This approach is highly recommended when subsequent statistical calculations or model training processes cannot tolerate any missing inputs, ensuring data integrity for downstream tasks.

When utilizing the default `dropna()` call, pandas iterates through every row of the data structure. If a row contains `np.nan` in any column, that entire row is excluded from the resulting **DataFrame**. While simple to implement, analysts must proceed with caution, as this method can be overly aggressive, potentially leading to a significant reduction in the dataset size if missing data is sparse but distributed across a large number of rows. It is essential to weigh the benefit of complete records against the potential loss of valuable data points.

For our sample dataset, we expect rows 0, 2, and 3 to be dropped due to the presence of **NaN** values in 'rating' (rows 0, 2) and 'assists' (rows 0, 3). Row 0 has two missing values, while rows 2 and 3 each have one. The remaining rows (1, 4-9) contain complete data and will be retained. The following code executes this default removal process, yielding a clean dataset of observations without any gaps:

df.dropna()

rating points assists rebounds

1 85.0 25.0 7.0 8
4 94.0 27.0 5.0 6
5 90.0 20.0 7.0 9
6 76.0 12.0 6.0 6
7 75.0 15.0 9.0 10
8 87.0 14.0 9.0 10
9 86.0 19.0 5.0 7

Example 2: Targeting Rows Where All Values are Missing (`how='all'`)

In stark contrast to the stringent default behavior of `how='any'`, we sometimes only want to discard rows that are entirely empty or contain non-valid data across all feature columns. This is achieved by setting the `how` parameter to `'all'` within the `dropna()` method. This setting is particularly useful for cleaning up datasets where data acquisition errors or manual entry might have introduced rows that are completely blank, providing absolutely no informational value.

When `how='all'` is specified, the function rigorously checks each row to confirm whether every single entry, across all columns, is an **NaN** value. If even one value in the row is valid (i.e., not null), the row is preserved. This ensures that any observation contributing some meaningful data remains in the **DataFrame**, minimizing unnecessary data loss and making it a far gentler cleaning method than dropping rows with only sporadic missingness.

Applying `df.dropna(how='all')` to our current sample dataset demonstrates its effect. Since our initial **DataFrame** was purposefully constructed such that every row contains at least one piece of valid data--for instance, Row 0, despite missing 'rating' and 'points', still has valid 'assists' and 'rebounds'--the operation results in no rows being dropped. This outcome clearly illustrates that `how='all'` is exclusively reserved for removing truly redundant, empty observations that contribute nothing to the analysis.

df.dropna(how='all')

rating points assists rebounds
0 NaN NaN 5.0 11
1 85.0 25.0 7.0 8
2 NaN 14.0 7.0 10
3 88.0 16.0 NaN 6
4 94.0 27.0 5.0 6
5 90.0 20.0 7.0 9
6 76.0 12.0 6.0 6
7 75.0 15.0 9.0 10

8 87.0 14.0 9.0 10
9 86.0 19.0 5.0 7

Example 3: Applying a Threshold for Missing Data Tolerance (`thresh` parameter)

The `thresh` parameter introduces crucial flexibility by allowing the analyst to define a compromise between data completeness and data volume. This parameter specifies the minimum required number of non-null values (valid data points) that a row must possess to be retained. If a row's count of valid entries falls below this specified threshold, the row is discarded. This method is often the most practical choice in real-world scenarios where some degree of missingness is expected but excessive missingness makes a record unreliable.

For example, if a dataset has five features, and we set `thresh=4`, any row missing two or more values will be dropped, while rows missing only one value will be kept. This is immensely useful in scenarios where data integrity relies on having a certain quantity of core features present, even if some auxiliary features are missing. The threshold provides a defined, quantifiable boundary for what constitutes a "sufficiently complete" observation for the intended statistical analysis.

Let's apply `thresh=3` to our sample data, meaning we require at least three valid metrics (non-**NaN** values) for a row to be kept. We review the original counts: Row 0 has 2 valid entries ('assists', 'rebounds'), Row 2 has 3 valid entries, and Row 3 has 3 valid entries. Since Row 0 only has two valid values, it fails the `thresh=3` requirement and is dropped. Rows 2 and 3, despite having missing data, meet the minimum threshold and are successfully preserved.

df.dropna(thresh=3)

rating points assists rebounds
1 85.0 25.0 7.0 8
2 NaN 14.0 7.0 10
3 88.0 16.0 NaN 6
4 94.0 27.0 5.0 6
5 90.0 20.0 7.0 9
6 76.0 12.0 6.0 6
7 75.0 15.0 9.0 10
8 87.0 14.0 9.0 10
9 86.0 19.0 5.0 7

Example 4: Focusing Removal on Specific Columns (`subset` parameter)

While global missingness controls (like `how='any'` or `thresh`) are useful, data integrity often hinges on the presence of data in specific, key columns. For instance, if 'rating' is the target variable or 'points' is a mandatory identifier, missing values in these columns must result in row removal, even if the rest of the row is complete. The `subset` parameter addresses this need by restricting the **dropna()** operation to a defined list of column names.

When the `subset` parameter is utilized, a row is dropped only if it contains a missing value within one of the specified columns. Missing values in columns outside of the subset are completely ignored during the cleaning process. This targeted approach is highly effective for maintaining the integrity of data points crucial for the study, allowing less important features to retain missing values without penalizing the observation.

In this specific example, we instruct pandas to drop any row where the 'assists' column contains **NaN**. Reviewing our original DataFrame, we find that Row 3 has a missing value in 'assists'. Row 0, despite having missing values elsewhere, contains a valid value for 'assists' (5.0) and is therefore retained. This demonstrates how the `subset` parameter, combined with the default `how='any'` logic applied only to the subset, precisely isolates the detection of missingness to the required features.

df.dropna(subset=)

rating points assists rebounds
0 NaN NaN 5.0 11
1 85.0 25.0 7.0 8
2 NaN 14.0 7.0 10
4 94.0 27.0 5.0 6
5 90.0 20.0 7.0 9
6 76.0 12.0 6.0 6
7 75.0 15.0 9.0 10
8 87.0 14.0 9.0 10
9 86.0 19.0 5.0 7

Example 5: Maintaining Clean Indexing After Row Removal

A common consequence of dropping rows using **dropna()** is the creation of a non-sequential index. When rows are removed, their original index labels remain absent from the resulting DataFrame, leading to gaps (e.g., indices 1, 4, 5, 6...). While these indices do not typically affect internal data manipulation, they can complicate dataset presentation and cause issues if the index

is later relied upon for positional slicing or simple iteration based on expected contiguous ordering. Therefore, analysts frequently need to reset the index to a clean, sequential integer range starting from zero.

To achieve this clean index, we typically chain the `dropna()` method with the `DataFrame.reset_index()` method. The `reset_index()` function generates a new, sequential integer index for the remaining rows. Crucially, when resetting the index after dropping rows, we must include the **`drop=True`** parameter. If `drop=False` (the default behavior), the old, gapped index would be retained and inserted as a new column in the DataFrame, which is generally undesirable when the goal is simply to tidy up the observational order.

In this final example, we first apply the default `dropna()` operation (as demonstrated in Example 1) to clean the data by removing all rows with any missing value. We then immediately apply the `reset_index(drop=True)` function on the resulting cleaned DataFrame. This two-step process yields a clean dataset ready for subsequent analysis, featuring a contiguous, zero-based indexing scheme that simplifies reading and future programmatic access.

#drop all rows that have any NaN values
df = df.dropna()

#reset index of DataFrame
df = df.reset_index(drop=True)

#view DataFrame
df

rating points assists rebounds
0 85.0 25.0 7.0 8
1 94.0 27.0 5.0 6
2 90.0 20.0 7.0 9
3 76.0 12.0 6.0 6
4 75.0 15.0 9.0 10
5 87.0 14.0 9.0 10
6 86.0 19.0 5.0 77

Advanced Considerations: The `axis` and `inplace` Arguments

While all previous examples focused on dropping rows (the default behavior, where `axis=0`), the `dropna()` method is fully capable of dropping columns instead. This behavior is controlled by the `axis` parameter. To apply the cleaning logic to columns, you must set **`axis=1`**. Dropping columns is a valid strategy when a specific feature contains an overwhelming number of missing values--for

example, 70% or more--making imputation unreliable and the column itself ineffective for modeling purposes.

When `axis=1` is set, parameters like `how` and `thresh` apply column-wise rather than row-wise. For instance, running `df.dropna(axis=1, how='any')` would remove any column that contains even a single missing value. Conversely, `df.dropna(axis=1, thresh=N)` dictates that only columns with at least `N` non-null entries will be retained. Understanding the `axis` parameter is critical for flexible data cleaning across both observations (rows) and features (columns) within the data structure.

Another vital argument for efficiency is **`inplace`**. By default, `dropna()` adheres to functional programming principles by returning a new DataFrame with the missing values removed, leaving the original DataFrame unmodified. However, when dealing with extremely large datasets, creating copies can be memory-intensive. Setting `inplace=True` modifies the original DataFrame directly, saving memory and avoiding the need for explicit reassignment (i.e., you don't need `df = df.dropna(...)`). While convenient and efficient, it mandates caution, as the original data state is permanently altered upon execution.

Summary of Key DataFrame.dropna() Parameters

The `dropna()` function offers high granularity in managing missing data. Mastering these parameters ensures that data quality control is precise and tailored to the specific needs of the analysis, providing robust solutions for dealing with incomplete observations. Below is a summary of the primary controls discussed in this guide.

`how='any'` (Default): This is the most stringent method, dropping a row (or column if `axis=1`) if **any** value within that observation is missing.

`how='all'`: This is the least stringent method, dropping a row (or column) only if **all** values within that observation are missing. It primarily targets entirely empty records.

`thresh=N`: This provides a vital middle ground, requiring at least `N` non-null values for a row to be retained, defining an acceptable tolerance for data sparsity.

`subset=`: This parameter restricts the missing value detection to a list of specified columns. The drop operation occurs only if a missing value is detected within the defined subset of features.

`axis`: Controls the dimension of operation. `axis=0` (default) applies the logic row-wise, while `axis=1` applies the logic column-wise.

By strategically combining these parameters, data scientists can execute powerful and efficient data cleaning operations using the robust tools provided by the pandas library, ultimately preparing

their data for reliable analysis.

You can find the complete documentation for the dropna() function here.