# How to Easily Remove Columns with NaN Values in Your Data

Authored by

**stats writer**

November 21, 2025

Data preprocessing and cleaning are vital steps in any robust analytical pipeline. A recurring challenge is handling missing values, which are frequently represented in datasets as NaN (Not a Number). The presence of these missing markers can severely impact statistical calculations, machine learning model performance, and data integrity. While imputation (filling missing values) is often preferred, sometimes the best strategy is to simply remove columns where the missingness is too prevalent or systematic, thereby reducing the size and complexity of the dataset while ensuring that the remaining features contain only valid, usable values.

In the context of Python data manipulation, specifically using the widely adopted pandas library, the removal of rows or columns based on missing data is efficiently handled by the .dropna() method. This method is exceptionally versatile, accepting several parameters that allow the user to control the behavior of the dropping operation. Crucially, when aiming to drop columns, we must specify the `axis` parameter to `1` (or `'columns'`), indicating that we are inspecting and potentially removing entire columns rather than rows.

This comprehensive guide details the three fundamental ways you can leverage the .dropna() function to clean a DataFrame by selectively removing columns based on the quantity and distribution of NaN values. We will start by establishing a common test environment and then proceed through examples demonstrating how to drop columns containing *any* missing data, columns containing *only* missing data, and columns that fail a specified minimum threshold of non-NaN observations.

## Understanding the Core Parameters of dropna()

The .dropna() method utilizes several key parameters that govern how the operation is executed. When focusing on column removal, two parameters are particularly important: `axis` and `how`. The `axis=1` argument directs the function to operate across the column indices, meaning it evaluates the contents of each column independently. The `how` parameter dictates the condition under which a column is considered 'droppable', providing a fine-grained level of control over the cleaning process.

The three methodologies presented below represent distinct strategies for dealing with data sparsity. Choosing the correct method depends entirely on the nature of your data and the tolerance for missingness in your analysis. For instance, if data integrity is paramount, dropping any column with even a single NaN might be appropriate. Conversely, if you are only concerned with entirely useless columns, a less restrictive method is preferable. We will also introduce the `thresh` parameter, which allows for advanced, numerically based dropping conditions.

**Method 1: Strict Removal (Any NaN):** Used when a column must be complete.

**Method 2: Lenient Removal (All NaN):** Used to remove columns that are entirely empty.

**Method 3: Threshold Removal (Thresh):** Used when a minimum number of valid (non-NaN) entries are required.

## Setting Up the Example DataFrame

To effectively demonstrate these column dropping techniques, we will first create a sample DataFrame using the NumPy library for generating placeholder missing values. This DataFrame simulates a simple sports dataset, where some columns have partial missingness (like `position`) and others have complete missingness (like `rebounds`). Observing how each method handles these different levels of sparsity is crucial for understanding their practical application.

We import both pandas and numpy and initialize our example data structure. Note the use of `np.nan` to explicitly represent missing data points in the `position` column and throughout the entire `rebounds` column. This structure provides a clear basis for testing the different removal criteria.

**import pandas as pd**
**import numpy as np**

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

#view DataFrame
print(df)

team position points rebounds
0 A NaN 11 NaN
1 A G 28 NaN
2 A F 10 NaN
3 B F 26 NaN
4 B C 6 NaN
5 B G 25 NaN

## Example 1: Dropping Columns with Any Missing Data (Default Behavior)

The most restrictive approach involves removing any column that contains even a single missing value. This method is often employed when downstream processes cannot tolerate missing data whatsoever, or when the cost of incorrect imputation outweighs the loss of the entire feature. When

you call .dropna() and specify `axis=1` without defining the `how` parameter, it defaults to `how='any'`.

Under this default setting, the function iterates through every column (because of `axis=1`) and checks if the condition specified by `how='any'` is met. If *any* value in the column is NaN, that entire column is dropped. In our example DataFrame, both the `position` column (which has one missing entry) and the `rebounds` column (which is entirely missing) will be removed, leaving only the columns that are perfectly complete: `team` and `points`.

**#drop columns with any NaN values**
**df = df.dropna(axis=1)**

#view updated DataFrame
print(df)

team points
0 A 11
1 A 28
2 A 10
3 B 26
4 B 6
5 B 25

Notice that the **position** and **rebounds** columns were successfully dropped since they both contained at least one NaN value, confirming the rigorous nature of the `how='any'` parameter when applied along the column axis.

## Example 2: Dropping Columns Only If Fully Missing (how='all')

In contrast to the strict `how='any'`, the `how='all'` parameter provides a much more lenient approach. This setting instructs pandas to only drop a column if *every single value* within that column is missing. This is particularly useful for identifying and eliminating truly empty features that contribute no information to the dataset, while preserving columns that, despite having a few missing data points, still contain substantial valuable information.

When we apply `how='all'` in conjunction with `axis=1`, the method only removes columns where the count of non-missing values is zero. Looking at our sample DataFrame, the `points` and `team` columns are obviously retained as they are complete. The `position` column is also retained because it has five valid entries, even though one is missing. The only column that meets the condition for removal is `rebounds`, which contains NaN for all six rows.

**#drop columns with all NaN values**

**df = df.dropna(axis=1, how='all')**

#view updated DataFrame
print(df)

team position points
0 A NaN 11
1 A G 28
2 A F 10
3 B F 26
4 B C 6
5 B G 25

Notice that the **rebounds** column was dropped since it was the only column where all values were NaN. This method preserves the **position** column despite its single missing entry, making it highly effective for targeted removal of completely void features.

## Example 3: Conditional Dropping Using the 'thresh' Parameter

The `thresh` parameter offers a highly customizable middle ground between the strict `'any'` and the lenient `'all'` methods. Instead of dictating the presence or absence of missing values, `thresh` requires a minimum count of *non-missing observations* for a column to be retained. If a column has fewer non-missing observations than the threshold value specified, that column is dropped.

This approach is vital when data quality mandates a certain level of completeness. For example, if a dataset has 100 rows, and you determine that any feature must contain at least 80 valid observations to be statistically meaningful, you would set `thresh=80`. In our small example, where the total number of rows is 6, setting `thresh=2` means that a column must contain at least two non-missing data points to survive the cleaning process. Let's analyze how this impacts our columns:

**team**: 6 valid values (Retained).

**points**: 6 valid values (Retained).

**position**: 5 valid values (Retained, as 5 is >= 2).

**rebounds**: 0 valid values (Dropped, as 0 is < 2).

**#drop columns with at least two NaN values**
**df = df.dropna(axis=1, thresh=2)**

#view updated DataFrame
print(df)

team position points
0 A NaN 11
1 A G 28
2 A F 10
3 B F 26
4 B C 6
5 B G 25

Notice that the **rebounds** column was dropped because it had zero valid entries, falling below the required `thresh=2`. Conversely, the **position** column, despite having one missing value, was retained because its five valid entries exceeded the threshold.

## Summary and Further Documentation

Mastering the .dropna() function is essential for effective data cleaning in pandas. By correctly setting the `axis` parameter to `1`, you shift the focus from row management to column management. Furthermore, utilizing the `how` and `thresh` parameters allows for precise control over the cleaning process, enabling data scientists to enforce specific quality standards based on the analytical requirements of the project.

We have demonstrated three increasingly sophisticated techniques for column removal:

Using the default `how='any'` for maximum strictness.

Using `how='all'` for removing only entirely empty columns.

Using the `thresh` parameter to enforce a minimum number of valid observations per column.

For more advanced usage scenarios, including specifying a subset of columns to evaluate, consult the official dropna() documentation provided by pandas. Consistent application of these methods ensures that your analyses are built upon reliable and robust data structures.