How to Easily Find Rows with NaN Values in Pandas

Authored by

stats writer

November 22, 2025

Working with real-world datasets inevitably involves managing missing data, a common challenge in data analysis. In the Pandas library, missing values are typically represented by the special floating-point value, NaN (Not a Number). Effectively identifying and isolating rows containing these missing indicators is a critical skill for any data scientist preparing data for modeling or analysis. The primary technique for accomplishing this isolation involves generating a boolean mask, which allows for precise filtering of the DataFrame based on the presence or absence of nulls. This method leverages Pandas' highly optimized vector operations, ensuring both efficiency and clarity in your data manipulation workflow.

The fundamental function used for detecting missing values is `isnull()`, which transforms the structure of the DataFrame into a corresponding structure of boolean values--where **True** indicates a missing value and **False** indicates a valid value. Once this mask is generated, it can be applied directly to the original DataFrame using indexing (`df`). For instance, the concise expression `df` returns a new DataFrame containing only the rows where at least one cell was identified as NaN, demonstrating the power of vectorized operations in Pandas.

Conversely, if the goal is to filter out missing data and retain only complete records, the `notnull()` function serves the opposite purpose. When applied, `notnull()` generates a boolean structure where **True** represents valid, non-missing entries. Utilizing the structure `df` effectively isolates all rows that do not contain NaN values, assuming the filtering is applied to the row level. Understanding the distinction and appropriate application of both `isnull()` and `notnull()` is paramount for achieving accurate data cleaning and preparation.

Key Strategies for Targeting Missing Values

When dealing with missing data, the approach you take depends heavily on whether you need to flag any missing value across an entire observation (row) or if you are specifically interested in nulls within a single, critical feature (column). Pandas provides two distinct but related methodologies to handle these scenarios, utilizing the powerful `.loc` indexer combined with aggregation methods like `.any()`.

Method 1: Select Rows with NaN Values in Any Column: This strategy is deployed when the integrity of the entire record is compromised by even a single missing entry. It identifies all rows that contain at least one NaN value anywhere within their columns. The core syntax for this is:

df.loc

Method 2: Select Rows with NaN Values in Specific Column: This approach is more focused, targeting only those rows where the null value appears in a designated column. This is crucial when specific features are non-negotiable for downstream analysis. The core syntax for this is:

df.loc.isnull()]

Both methods rely on first generating the underlying boolean structure using `.isnull()`. For Method 1, subsequent aggregation across the row axis is required to collapse the results into a single boolean value per row. For Method 2, selecting the column prior to calling `.isnull()` isolates the required mask directly. Mastering the combination of these functions provides granular control over missing data imputation or removal processes. The following examples demonstrate how to use each method in practice with a working Pandas DataFrame.

Setting Up the Demonstration DataFrame

To illustrate these concepts clearly, we will establish a sample DataFrame named `df`, simulating typical statistical data where missing observations have been introduced intentionally using the NumPy constant, `np.NaN`. This setup requires importing both the Pandas and NumPy libraries, which is standard practice when handling numerical data and missing values in Python.

Our sample NumPy-based data structure includes several rows with varying combinations of missing values across the **points**, **assists**, and **rebounds** columns. For example, Row 1 is missing points, Row 4 is missing assists, and Row 7 is missing both assists and rebounds. This diversity in missingness allows us to fully explore how the different filtering methods function and interact with the data structure. The initial creation and display of the Pandas object are essential before proceeding with the selection operations.

The following code block demonstrates the necessary initialization steps required to create the `df` object. Pay close attention to how `np.NaN` is used within the list definitions to explicitly denote where the missing values reside in the structure. The use of `np.NaN` is standard practice in Python data analysis pipelines as it integrates seamlessly with NumPy and Pandas numerical types.

import pandas as pd
import numpy as np

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

#view DataFrame
print(df)

Method 1: Isolating Rows with Any Missing Value

The first critical task in cleaning data is often identifying any observation that is incomplete, regardless of which specific feature is missing. To achieve this, we combine the `.isnull()` function with the aggregation method `.any()`, ensuring that the aggregation occurs along the correct dimension using the `axis=1` parameter. The `axis=1` argument directs the operation to summarize across the columns (row-wise), returning a single boolean value for each row. If any element within a given row evaluates to **True** (i.e., is NaN), then `.any(axis=1)` returns **True** for that entire row index.

The resulting boolean Series--which contains **True** for rows with missing data and **False** otherwise--is then used directly within the powerful `.loc` indexer. The `.loc` accessor in Pandas is highly recommended for label-based indexing and filtering, as it improves code readability and prevents ambiguity compared to simple bracket indexing. The overall structure, `df.loc`, is an elegant and efficient way to extract the subset of data requiring attention due to missingness.

The following syntax applies this exact logic to our sample data, creating a new object, `df_nan_rows`. Note how rows 1, 4, and 7 are successfully extracted, confirming that this methodology correctly captures all observations where at least one cell contains a value of NaN. This filtered subset is now ready for subsequent processing, such as imputation or outright removal.

#create new DataFrame that only contains rows with NaNs in any column
df_nan_rows = df.loc

#view results
print(df_nan_rows)

team points assists rebounds
1 B NaN 7.0 8.0
4 E 14.0 NaN 6.0
7 H 28.0 NaN NaN

Method 2: Focusing on Missing Values in a Specific Column

In many analytical contexts, only missing values within a specific, high-priority feature are relevant. For example, if we are building a predictive model and **assists** is a crucial predictor, we might only want to clean or drop rows where **assists** is null, ignoring missing values in less important columns like **points** or **rebounds**. This requires a targeted application of the `.isnull()` function.

The syntax for this targeted selection is significantly simpler than Method 1, as it bypasses the

need for the `.any()` aggregation. By first selecting the column (e.g., `df`) and then calling `.isnull()`, Pandas returns a boolean mask that precisely corresponds to the indices where the specific column contains a NaN value. This mask is then passed to the `.loc` indexer, resulting in a filtered DataFrame. Notice that each row in the resulting DataFrame contains a NaN value in the **assists** column.

When we apply this technique to isolate rows based on missing **assists** values, we observe that only rows 4 and 7 are returned. Crucially, Row 1 (which had a missing **points** value) is excluded because its **assists** entry (7.0) is valid. This confirms the targeted nature of the operation: only rows failing the specific column criterion are included in the result. Understanding this specific column filtering is vital for complex data preparation tasks where feature importance dictates the cleaning strategy.

#create new DataFrame that only contains rows with NaNs in assists column
df_assists_nans = df.loc.isnull()]
#view results
print(df_assists_nans)

team points assists rebounds
4 E 14.0 NaN 6.0
7 H 28.0 NaN NaN

Advanced Filtering: Selecting Rows Based on Multiple Conditions

While the previous methods focused on single criteria, data cleaning often requires combining multiple conditions simultaneously. For instance, you might need to find rows where **points** is missing **AND rebounds** is missing, or where **points** is missing **OR assists** is missing. Pandas facilitates this advanced filtering using standard Python logical operators: `&` for AND, and `|` for OR. It is essential to wrap each individual boolean condition in parentheses to ensure correct operator precedence.

To demonstrate, let's find rows where either **points** is missing **OR rebounds** is missing. We first generate two separate boolean masks using `.isnull()` for each column, and then combine them using the OR operator (`|`). This powerful combination allows for defining highly specific criteria for selecting subsets of data. For example, if we were preparing data for a model that requires both fields, identifying these rows would be the first step toward imputation or dropping them.

Conversely, if we required stricter filtering--such as selecting only rows where the player failed to record both **assists AND rebounds** (simultaneously missing)--we would substitute the OR operator (`|`) with the AND operator (`&`). This flexibility is central to effective data preprocessing.

Remember that the resulting complex mask must still be applied using the `.loc` indexer for precise row selection.

 # Example: Selecting rows where points OR rebounds are missing
df_points_or_rebounds_nans = df.loc.isnull()) | (df.isnull())]
print(df_points_or_rebounds_nans)

team points assists rebounds
1 B NaN 7.0 8.0
4 E 14.0 NaN 6.0
7 H 28.0 NaN NaN

# Example: Selecting rows where assists AND rebounds are missing
df_assists_and_rebounds_nans = df.loc.isnull()) & (df.isnull())]
print(df_assists_and_rebounds_nans)

team points assists rebounds
7 H 28.0 NaN NaN

Leveraging `notnull()` for Complete Records

While often the focus is on isolating missing data, the inverse operation--selecting only complete, valid records--is equally important. The `.notnull()` function simplifies this process significantly. When applied to an entire DataFrame, it returns a boolean structure indicating where values are present (**True**) versus where they are missing (**False**).

To identify rows that are **entirely** complete (i.e., contain no missing values in any column), we utilize `.notnull()` followed by `.all(axis=1)`. Just as `.any(axis=1)` checks if *any* value is True, `.all(axis=1)` checks if *all* values across the row are True. This effectively creates a boolean mask that selects only the rows where every single cell is non-null. This technique is indispensable when the data integrity requirements mandate the use of only full observations.

Using our sample data, we can easily extract the subset of players who have complete records for **points**, **assists**, and **rebounds**. Applying the condition `df.notnull().all(axis=1)` ensures that rows 1, 4, and 7--which we previously identified as containing nulls--are correctly excluded, leaving only the pristine data ready for analysis without any need for imputation or complex handling of nulls.

 # Select rows where ALL columns are NOT null (i.e., complete records)
df_complete_rows = df.loc
print(df_complete_rows)

team points assists rebounds
0 A 18.0 5.0 11.0
2 C 19.0 7.0 10.0
3 D 14.0 9.0 6.0
5 F 11.0 9.0 5.0
6 G 20.0 9.0 9.0

Alternative Indexing Using `query()`

While boolean indexing with `.loc` is the primary and most robust method for row selection, Pandas also offers the `.query()` function as a readable alternative, especially useful for simple filtering tasks involving missing values. The `.query()` method allows users to write SQL-like expressions directly within a string, often enhancing clarity for users familiar with database query languages. Although `.query()` is generally slower than direct boolean indexing for large datasets, its syntax simplicity makes it appealing for exploratory data analysis.

The `.query()` function utilizes the specialized keywords `isna()` and `notna()`, which are direct aliases for `isnull()` and `notnull()`, respectively, specifically designed to work within the query string parsing engine. For example, to select all rows where the **points** column is missing, one simply uses the string `'points.isna()'` passed to the function. This avoids the explicit creation of the boolean mask and the use of the `.loc` indexer, streamlining the code when the criteria are straightforward.

However, it is important to note that `.query()` is optimized for column comparisons and value filtering. For complex, row-level aggregations (like Method 1, finding NaN in *any* column), the direct boolean mask approach using `.loc` and `.any(axis=1)` remains the superior and often necessary choice. The use case for `.query()` is thus generally restricted to column-specific filtering where readability is prioritized over marginal performance gains.

 # Example using .query() to find missing 'points'
df_query_nans = df.query('points.isna()')
print(df_query_nans)

team points assists rebounds
1 B NaN 7.0 8.0

Summary and Best Practices

Mastering the identification and selection of rows containing NaN values is a fundamental requirement for robust data preprocessing using NumPy and Pandas. We have explored the two

core methodologies: filtering based on nulls across the entire row using `.isnull().any(axis=1)`, and targeting nulls within specific columns using column selection combined with `.isnull()`.

For optimal performance and clarity, the following best practices should be adhered to: **Always use `.loc`** when applying complex boolean mask filtering, as it explicitly defines the intent to select by index labels. When combining multiple conditions, ensure that parentheses are used around individual conditions to correctly manage Python's operator precedence, especially when mixing logical operators (`&` and `|`).

Finally, remember that selecting rows with NaNs is only the first step. The next crucial decision involves how to handle these incomplete records, whether through dropping them entirely (using `.dropna()`) or by employing sophisticated imputation techniques to fill the missing spaces. The efficient and precise selection techniques detailed here provide the essential foundation for these advanced data manipulation steps.