Use Pandas fillna() to Replace NaN Values

Authored by

Mohammed looti

November 3, 2025

The Crucial Role of Handling Missing Data

In the realm of data analysis and machine learning, encountering missing values is not just common--it is inevitable. These critical gaps, often represented by the standardized marker **Not a Number (NaN values)**, can severely skew statistical results, introduce systemic bias, and ultimately lead to faulty model predictions if not addressed proactively. Consequently, effective data cleaning constitutes the cornerstone of any reliable data project.

The **Pandas** library, a fundamental and widely-used tool in the Python data science ecosystem, provides robust mechanisms specifically designed for managing incomplete or sparse data. One of the most essential methods for addressing these missing entries is the powerful `fillna()` function. This function allows data professionals to systematically replace NaN values with specified constants, statistically derived measures, or values intelligently derived from adjacent data points.

Understanding how to apply `fillna()` efficiently is vital for ensuring the integrity, usability, and statistical validity of your DataFrame. This comprehensive tutorial focuses on the practical application of this function, demonstrating various use cases ranging from targeted column replacement to sophisticated global imputation strategies.

Understanding the `fillna()` Function Syntax

The `fillna()` function is highly flexible, allowing users to specify not only the replacement value (or method) but also the exact scope of the operation. Whether you need to replace missing values in a single column, a small, select group of columns, or perform a mass replacement across the entire DataFrame, the syntax remains intuitive and straightforward once the data subset is defined.

The basic operational modes of `fillna()` are determined by how you select the data subset immediately before applying the method. The three most common applications are shown below:

#replace NaN values in one column
df = df.fillna(0)

#replace NaN values in multiple columns
df] = df].fillna(0)

#replace NaN values in all columns
df = df.fillna(0)

It is important to recognize that in these initial examples, we are using the simple integer `0` as the replacement value. In professional data imputation, this value is frequently substituted with a

column's calculated mean, median, mode, or a specific string marker, depending entirely on the data type, distribution characteristics, and overall analysis requirements.

Preparing the Example DataFrame

To clearly illustrate the practical usage of **`fillna()`**, we will work with a sample DataFrame representing fictitious player statistics. This dataset contains deliberate missing entries, marked as `np.nan`, across several statistical columns (specifically 'rating', 'points', and 'assists'). We utilize the powerful **NumPy** library to generate these specific **NaN values** for demonstration purposes.

This controlled setup allows us to clearly track how the missing data is modified and updated in each subsequent example. We initialize the DataFrame below and then inspect its initial state to precisely identify where the **NaN values** are located before any imputation is performed.

import numpy as np
import pandas as pd

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

#view DataFrame
df

rating points assists rebounds
0 NaN 25.0 5.0 11
1 85.0 NaN 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

As demonstrated in the output above, the `rating` column has two missing values (at indices 0 and 2), the `points` column contains one missing entry (index 1), and the `assists` column also contains one missing entry (index 3). We will now proceed to demonstrate various methods for filling these

specific gaps.

Example 1: Targeted Imputation in a Single Column

In many real-world analytical scenarios, imputation must be applied selectively to maintain the integrity of other, unrelated columns. If we decide that missing player ratings should be treated as a score of zero (perhaps indicating a player who was unrated or absent), we can target only the `rating` column using standard **Pandas** column indexing, which treats the column as a Series object.

Applying **`fillna()`** directly to a Pandas Series ensures that the operation is completely isolated. This precise method is crucial when different columns require vastly different imputation logic--for instance, replacing missing numerical scores with the mean, but replacing missing categorical data with the mode or a specific string marker like 'Unknown'.

The following code demonstrates replacing the **NaN values** specifically in the `rating` column with the value `0`, while intentionally leaving any missing values in other columns untouched:

#replace NaNs with zeros in 'rating' column
df = df.fillna(0)

#view DataFrame
df

rating points assists rebounds
0 0.0 25.0 5.0 11
1 85.0 NaN 7.0 8
2 0.0 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 2: Applying Imputation Across Multiple Columns

When several columns share similar characteristics or require the exact same default replacement value, it is significantly more efficient to apply **`fillna()`** simultaneously to a predefined subset of the DataFrame. This is achieved by passing a standard Python list of column names (e.g.,) to the

Pandas indexing operator.

This technique is particularly useful for sets of features that belong to the same logical domain (e.g., all monetary values, all temperature readings, or all performance metrics) where a uniform imputation strategy is statistically appropriate. Utilizing this method avoids repetitive code, ensures consistency across related variables, and improves code readability.

Here, we apply the zero replacement to both the `rating` and the `points` columns. Crucially, note that the remaining missing value in the `assists` column (at index 3) remains completely unchanged, demonstrating the targeted nature of this operation:

#replace NaNs with zeros in 'rating' and 'points' columns
df] = df].fillna(0)

#view DataFrame
df

rating points assists rebounds
0 0.0 25.0 5.0 11
1 85.0 0.0 7.0 8
2 0.0 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: Global Imputation Across the Entire DataFrame

For swift initial data preparation, especially in cases where all numerical columns should adopt the same default replacement value, **`fillna()`** can be applied directly to the entire DataFrame object. This approach utilizes the simplest syntax, requiring no explicit column specification, and applies the replacement universally.

A global replacement strategy is typically employed when the underlying mechanism of missingness is assumed to be uniform across the dataset, or when a quick default value (like 0) is acceptable across all features before more sophisticated statistical imputation methods are deployed. It is a powerful, single command for quickly eliminating all **NaN values** from the dataset.

In this final demonstration, we use `fillna()` to replace the remaining missing value in the `assists` column (index 3), effectively ensuring that every cell in the DataFrame contains a valid number:

#replace NaNs with zeros in all columns
df = df.fillna(0)

#view DataFrame
df

rating points assists rebounds
0 0.0 25.0 5.0 11
1 85.0 0.0 7.0 8
2 0.0 14.0 7.0 10
3 88.0 16.0 0.0 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

All **NaN values** have now been successfully replaced by `0.0`, resulting in a fully clean and usable DataFrame ready for subsequent analysis or predictive modeling.

Advanced Imputation Strategies Using `fillna()`

While replacing missing values with a constant like zero is simple and fast, it is often statistically naive and can introduce unintended distortion into the data distribution. A more robust approach utilizes statistical measures or interpolation methods. Fortunately, the `fillna()` function is highly versatile, capable of handling these advanced scenarios by accepting a dictionary of replacement values (one per column) or utilizing its powerful built-in method parameters.

One of the most common and accepted statistical methods is replacing missing values in a column with the column's calculated **mean** or **median**. This is achieved by calculating the statistic dynamically and passing the resulting value as the replacement argument. Using the median is often statistically preferred for skewed data distributions as it is significantly less sensitive to the presence of outliers than the mean.

Alternatively, for analyzing **time-series or sequential data**, interpolation methods like forward fill (`ffill`) or backward fill (`bfill`) are essential. These methods propagate the last valid observed

value forward or the next valid observation backward, respectively, which is critical for preserving sequential context and continuity in time-ordered data.

Imputing with Mean/Median: You can calculate the central tendency of the column and use that precise value for replacement. For example: `df.fillna(df.mean())`.
Forward Fill (`ffill`): Uses the value from the previous valid row to fill the current NaN: `df.fillna(method='ffill')`.
Backward Fill (`bfill`): Uses the value from the subsequent valid row to fill the current NaN: `df.fillna(method='bfill')`.

Conclusion and Further Resources

The **Pandas** **`fillna()`** function is an indispensable utility for rigorous data cleaning, offering scalable and flexible solutions for handling missing data across single columns, multiple columns, or entire datasets. Mastering its basic syntax and understanding its advanced parameters (such as statistical imputation and sequential filling methods) is a cornerstone skill for effective data preparation in Python.

For data scientists and analysts seeking to delve deeper into the nuances of data handling within the Pandas framework, the official documentation provides comprehensive details on all available parameters, advanced techniques, and edge cases associated with the **`fillna()`** function.

You can find the complete online documentation for the **`fillna()`** function here.

Additional Resources for Pandas Operations

Building upon this foundational knowledge, the following tutorials explain how to perform other common and essential operations in **Pandas**, enabling you to become proficient in all aspects of data manipulation:

How to Select Rows and Columns in a DataFrame
Using GroupBy for Data Aggregation
Merging and Joining Multiple DataFrames