How to Easily Replace NaN Values with the Mean in Pandas

Authored by

stats writer

December 1, 2025

Data cleansing is a critical preliminary step in any robust data science or machine learning workflow. Rarely do real-world datasets arrive perfectly structured and complete; instead, they often contain **NaN values** (Not a Number), which represent missing data points. These missing entries, if not handled appropriately, can severely compromise the accuracy and reliability of analytical models. Handling missing data, a process often referred to as imputation, involves replacing these gaps with substituted values based on statistical methods.

The Python **Pandas library** provides indispensable tools for data manipulation, and central to its utility is the ability to manage these nulls efficiently. Among the simplest and most frequently employed strategies for numerical data is mean imputation, where missing values in a feature column are replaced by the average of the observed values in that same column. This method offers a quick and easy solution, preventing the loss of potentially valuable rows that contain missing data, thereby maintaining the dataset's size for subsequent analysis.

This comprehensive guide will demonstrate, using detailed examples, how to leverage the powerful **fillna() function** in Pandas to perform mean imputation across different scopes--single columns, multiple specified columns, or the entirety of a **Pandas DataFrame**. Understanding these distinct approaches is vital for data practitioners who need precise control over their data cleaning processes.

Why Use the Mean for Imputation?

Imputation by the arithmetic **mean** is a foundational technique in descriptive statistics used to estimate missing values. It is based on the assumption that the missing data point is likely close to the average value of the existing data distribution within that specific variable. This method is particularly suitable when the data is roughly normally distributed and the missingness is considered to be Missing At Random (MAR).

The key advantage of using the mean lies in its simplicity and computational efficiency. Unlike more complex model-based imputation techniques, mean imputation requires minimal processing power and can be executed instantaneously, making it highly practical for large datasets. Furthermore, replacing nulls with the mean ensures that the average value of the column remains unchanged after the imputation, preserving the central tendency of the feature.

However, it is crucial to recognize the statistical implications of this method. While simple, mean imputation artificially reduces the variance (spread) of the data, potentially leading to an underestimation of standard errors and biased model results if the proportion of missing data is high. Therefore, while we explore the mechanics of this operation, always consider the characteristics of your dataset before deciding on the imputation strategy.

Prerequisites and Initial Data Setup

Before diving into the imputation methods, we must ensure the necessary libraries are imported and establish a sample **DataFrame** containing **NaN values**. The Pandas library handles the DataFrame structure, while **NumPy** is often used to represent the explicit missing data points (`np.nan`).

The primary mechanism for substituting these null entries is the **fillna()** function provided by the **Pandas library**. This function is highly versatile, accepting constants, dictionary mappings, or, as we will demonstrate, calculated statistical measures like the mean.

Here are three common ways to use this function to target different scopes within your data:

Method 1: Fill NaN Values in One Column with Mean

df = df.fillna(df.mean())

Method 2: Fill NaN Values in Multiple Columns with Mean

df] = df].fillna(df].mean())

Method 3: Fill NaN Values in All Columns with Mean

df = df.fillna(df.mean())

The following examples show how to use each method in practice with the following pandas DataFrame, simulating numerical statistics with various missing entries:

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

Example 1: Fill NaN Values in One Column with Mean

The most straightforward imputation task involves isolating a specific column that requires mean substitution. This approach is ideal when you have heterogeneous data--for instance, if one column contains numerical ratings (suitable for mean imputation) while another contains categorical labels (requiring mode imputation). By focusing on a single column, we ensure that the imputation logic is narrowly applied.

To execute this, we first calculate the mean of the target column using the Series method `.mean()`. Importantly, Pandas automatically ignores **NaN values** when calculating the mean, ensuring the statistic is derived only from observed, valid data points. This calculated mean is then passed directly as the argument to the `.fillna()` method for that specific column. The operation must be assigned back to the column to modify the DataFrame in place.

The following code shows how to fill the **NaN values** in the **rating** column with the calculated **mean** value of the **rating** column:

#fill NaNs with column mean in 'rating' column
df = df.fillna(df.mean())

#view updated DataFrame
df

rating points assists rebounds
0 85.125 25.0 5.0 11
1 85.000 NaN 7.0 8
2 85.125 14.0 7.0 10
3 88.000 16.0 NaN 6
4 94.000 27.0 5.0 6

5 90.000 20.0 7.0 9
6 76.000 12.0 6.0 6
7 75.000 15.0 9.0 10
8 87.000 14.0 9.0 10
9 86.000 19.0 5.0 7

The **mean** value in the **rating** column was calculated to be **85.125** (based on the original eight non-null entries), so each of the **NaN values** in the **rating** column were successfully filled with this calculated statistic.

Example 2: Fill NaN Values in Multiple Columns with Mean

Often, a dataset requires simultaneous imputation across several, but not all, numerical columns. Instead of chaining single-column operations, Pandas offers an efficient way to apply column-specific means to a subset of columns within the **DataFrame**. This method leverages the fact that when `.mean()` is called on a multi-column DataFrame slice, it computes the mean for each column independently.

To perform this simultaneous imputation, we select the desired columns using a list of column names (e.g., `df]`). We then call `.fillna()` on this slice, passing the means of those same columns as the argument. It is crucial to understand that Pandas maintains alignment, meaning the mean of 'col1' is used only to fill the NaNs in 'col1', and the mean of 'col2' is used only for 'col2', even though the operation is performed jointly.

The following code shows how to fill the **NaN values** in both the **rating** and **points** columns with their respective column **means**:

#fill NaNs with column means in 'rating' and 'points' columns
df] = df].fillna(df].mean())

#view updated DataFrame
df

rating points assists rebounds
0 85.125 25.0 5.0 11
1 85.000 18.0 7.0 8
2 85.125 14.0 7.0 10
3 88.000 16.0 NaN 6
4 94.000 27.0 5.0 6
5 90.000 20.0 7.0 9
6 76.000 12.0 6.0 6

7 75.000 15.0 9.0 10
8 87.000 14.0 9.0 10
9 86.000 19.0 5.0 7

Example 3: Fill NaN Values in All Columns with Mean

If a DataFrame consists primarily of numerical features and the data analyst determines that mean imputation is suitable for all columns containing missing data, Pandas provides the most concise syntax for global application. By calling `.mean()` directly on the DataFrame itself, Pandas calculates the mean for every column (Series) individually, ignoring any non-numeric columns by default.

This collective output (a Pandas Series where index labels correspond to column names and values are the means) can then be passed into the **fillna() function** applied to the entire DataFrame. This powerful technique automatically handles column-wise imputation, ensuring that each missing entry is filled using the average of its respective feature column.

The following code shows how to fill the **NaN values** in each column with the column means:

#fill NaNs with column means in each column
df = df.fillna(df.mean())

#view updated DataFrame
df

rating points assists rebounds
0 85.125 25.0 5.000000 11
1 85.000 18.0 7.000000 8
2 85.125 14.0 7.000000 10
3 88.000 16.0 6.666667 6
4 94.000 27.0 5.000000 6
5 90.000 20.0 7.000000 9
6 76.000 12.0 6.000000 6
7 75.000 15.0 9.000000 10
8 87.000 14.0 9.000000 10
9 86.000 19.0 5.000000 7

Notice that the **NaN values** in each column were filled with their column **mean**.

Handling Non-Numeric Columns and Mixed Data

A common complexity when applying Method 3 (global imputation) is the presence of mixed data types. Pandas handles this elegantly: when `df.mean()` is executed, it automatically calculates means only for numerical columns (integer and float types). String, boolean, or categorical columns are automatically excluded from the calculation.

Consequently, when this resulting Series of means is passed to `df.fillna()`, the imputation process only affects the numerical columns. Any non-numeric columns containing NaNs will remain untouched, requiring a different imputation strategy (such as using the mode or a specific constant) tailored for qualitative data. This feature ensures that the global mean imputation method does not inadvertently corrupt categorical features.

For datasets with many features, it is always best practice to first check the data types using `df.dtypes` and verify which columns Pandas considers numerical. If a column is numeric but should not be imputed (perhaps it's an ID), it must be explicitly excluded, either by converting its type or by utilizing Method 2 and listing only the desired columns for imputation.

Potential Drawbacks and Advanced Considerations

While mean imputation is simple and highly effective for quick data cleaning, data scientists must be aware of its limitations. The primary issue is that substituting the mean introduces artificial data points that do not reflect the true variance or relationship structure of the original data. This can lead to misleading correlation metrics and potentially biased machine learning models, especially if the missing rate is high.

Furthermore, mean imputation is highly sensitive to outliers. If a column has extreme values, the calculated mean will be pulled towards these outliers, resulting in an imputed value that is not representative of the typical observation. In such cases, the **median** is often a safer choice for imputation, as it is robust to extreme values and preserves the central tendency without inflating the mean.

For advanced scenarios, practitioners might consider more sophisticated techniques, such as imputing using predictive models (like k-Nearest Neighbors or MICE--Multiple Imputation by Chained Equations) or employing time-series specific methods like interpolation. However, for baseline analyses and when missingness is minimal, the Pandas `.fillna(df.mean())` approach remains an essential tool in the data preparation toolkit.

Conclusion and Resources

Handling missing data is a fundamental aspect of data preparation. The Pandas library provides

highly optimized and flexible methods for applying statistical imputation strategies efficiently. By understanding the scope control offered by the **fillna() function**, users can precisely target single columns, specific subsets, or the entire DataFrame, ensuring data integrity across various analytical tasks.

Whether you choose the simplicity of mean imputation or opt for a more complex method, mastering the syntax demonstrated here is crucial for clean data workflows. This capability ensures that models receive complete datasets, optimizing performance and reliability.

You can find the complete online documentation for the **fillna() function** on the official Pandas website.