How to Fill NA Values for Multiple Columns in Pandas

Authored by

stats writer

December 12, 2025

Handling NA values, or missing data, is one of the most fundamental steps in the data cleaning and preprocessing pipeline. When working with large datasets in Python, the Pandas library provides highly efficient tools for managing these gaps. Specifically, the fillna() function is the primary mechanism for imputing missing entries across a DataFrame.

The versatility of the **fillna()** function allows data scientists to replace missing entries using various strategies, ranging from simple scalar replacements to sophisticated forward or backward propagation, or even statistical imputation based on column aggregates. This tutorial will explore several practical applications of **fillna()**, focusing on how to efficiently apply different filling strategies across multiple selected columns, thereby ensuring data integrity and readiness for analysis.

Understanding how to handle missing data across diverse columns is critical because different data types often require different imputation techniques. For instance, a missing categorical entry might best be filled with a placeholder like 'Unknown', while a missing numerical entry might be best handled by the column's mean or median. Mastering these techniques ensures that the imputation process is both precise and context-aware.

The Pandas **fillna()** method is indispensable for addressing missing values within a DataFrame. This function offers robust control over which values are replaced and what replacement strategy is used.

To demonstrate the various methods for filling missing values, we will use a standardized sample DataFrame representing hypothetical sports team statistics. Note the strategic placement of **NaN** values using the NumPy library to simulate real-world data gaps.

import pandas as pd
import numpy as np

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

#view DataFrame
print(df)

team points assists rebounds
0 A 25.0 5.0 11
1 NaN NaN 7.0 8

2 B 15.0 7.0 10
3 B NaN 9.0 6
4 B 19.0 12.0 6
5 C 23.0 9.0 5
6 C 25.0 NaN 9
7 C 29.0 4.0 12

Example 1: Filling All Missing Values with a Single Scalar

The simplest approach to dealing with NA values is to replace every missing entry in the entire DataFrame with a single, uniform scalar value. While this method is straightforward, it should be used cautiously, as replacing categorical data (like 'team') and numerical data (like 'points') with the same value (e.g., zero) can distort statistical measures and introduce bias. However, if the goal is to simply mark all missing numerical entries as zero for downstream processing, this method is highly effective.

To execute this, we pass the desired replacement value directly to the **value** parameter of the fillna() function. We also use **inplace=True** to modify the original DataFrame directly, avoiding the need to assign the result back to the variable. In this demonstration, we choose to replace all existing **NaN** entries with the numerical value 0.

Observe how the categorical column 'team' (row 1) and the numerical columns 'points' (rows 1 and 3) and 'assists' (row 6) are all uniformly affected by this single command. This showcases the broad impact of applying a scalar replacement across the entire dataset without specifying column subsets.

#replace all missing values with zero
df.fillna(value=0, inplace=True)

#view DataFrame
print(df)

team points assists rebounds
0 A 25.0 5.0 11
1 0 0.0 7.0 8
2 B 15.0 7.0 10
3 B 0.0 9.0 6
4 B 19.0 12.0 6
5 C 23.0 9.0 5
6 C 25.0 0.0 9

7 C 29.0 4.0 12

Example 2: Targeting Specific Columns with a Uniform Value

In many realistic data scenarios, missing values should only be filled within certain columns, leaving others untouched. For example, we might only want to impute missing 'points' and 'assists' values while reserving judgment on how to handle the missing 'team' data. The Pandas library facilitates this targeted imputation through column selection.

To restrict the scope of the **fillna()** operation, we first select the desired columns using double brackets (e.g., `df]`). We then apply **fillna()** specifically to this subset. It is important to remember that when using this technique, we must assign the filled subset back to the original DataFrame columns, as `inplace=True` often behaves unexpectedly when chained after column selection.

In this example, we re-initialize our DataFrame (mentally, or by rerunning the setup code) and focus solely on replacing missing values in the 'points' and 'assists' columns with zero. Notice that the **NaN** value in the 'team' column (row 1) remains unaffected, demonstrating precise control over the imputation scope. This approach is superior to Example 1 when imputation needs to be applied selectively to numerical metrics.

#replace missing values in points and assists columns with zero
df] = df].fillna(value=0)

#view DataFrame
print(df)

team points assists rebounds
0 A 25.0 5.0 11
1 NaN 0.0 7.0 8
2 B 15.0 7.0 10
3 B 0.0 9.0 6
4 B 19.0 12.0 6
5 C 23.0 9.0 5
6 C 25.0 0.0 9
7 C 29.0 4.0 12

Example 3: Filling Multiple Columns with Unique Values Using a Dictionary

When dealing with a heterogeneous dataset, the missing value replacement often needs to be unique for each column based on its data type or statistical properties. The most robust way to

achieve this column-specific imputation is by passing a Python dictionary to the **fillna()** function. The dictionary maps column names (keys) to their corresponding replacement values (values).

This dictionary-based approach allows for immense flexibility. For instance, we can replace missing team names with a string placeholder like 'Unknown', while simultaneously replacing missing 'points' with the integer 0, and perhaps replacing missing 'assists' with a specific value. This method ensures that the imputation strategy is tailored precisely to the semantics of each feature.

In the following code, observe the distinct treatment applied to three different columns: 'team' receives a descriptive string, 'points' receives a numerical zero, and 'assists' receives a unique descriptive string ('zero' in this context, highlighting that fillna() accepts various types). This is the ideal method for simultaneously handling missing values across different data types within a single operation.

#replace missing values in three columns with three different values
df.fillna({'team':'Unknown', 'points': 0, 'assists': 'zero'}, inplace=True)

#view DataFrame
print(df)

team points assists rebounds
0 A 25.0 5 11
1 Unknown 0.0 7 8
2 B 15.0 7 10
3 B 0.0 9 6
4 B 19.0 12 6
5 C 23.0 9 5
6 C 25.0 zero 9
7 C 29.0 4 12

Notice that each of the missing values in the three specified columns were replaced with their unique, designated value, while the 'rebounds' column, which had no missing values, remained unchanged.

Example 4: Imputing Missing Values with Statistical Aggregations (Mean or Median)

Replacing NA values with a fixed scalar like zero often introduces bias, especially if zero is outside the normal distribution range of the data. A far more common and statistically sound imputation technique involves replacing missing data points with a measure of central tendency derived from

the existing data in that column, such as the mean or median. This approach minimizes distortion to the overall distribution of the variable.

To implement this in Pandas, we first calculate the desired aggregation (e.g., `df.mean()`) and then pass this calculated value to the **fillna()** function. This process is typically performed column-by-column, as the mean of one column is statistically irrelevant to the values in another column.

A powerful technique for handling multiple columns simultaneously is to use the dictionary approach from Example 3, but populate the dictionary dynamically with calculated means or medians for each target column. This ensures that each column is imputed using its own, independent statistic. For example, if 'points' is missing, it receives the mean of all non-missing 'points', and if 'assists' is missing, it receives the median of all non-missing 'assists'.

Consider the scenario where we want to use the mean for 'points' and the median for 'assists' to fill their respective missing entries. We calculate these aggregates and construct the dictionary dynamically before calling **fillna()** on the DataFrame. This is often the preferred professional method for numerical imputation.

 # Re-create fresh DataFrame for clear imputation demonstration
df_stat = pd.DataFrame({'points': ,
'assists': })

# Calculate mean for 'points' and median for 'assists'
points_mean = df_stat.mean()
assists_median = df_stat.median()

# Create dictionary mapping columns to their aggregate fill values
fill_values = {'points': points_mean, 'assists': assists_median}

# Apply dictionary to fill NA values
df_stat.fillna(fill_values, inplace=True)

# View result
print(df_stat)

points assists
0 25.000000 5.0
1 22.666667 7.0
2 15.000000 7.0
3 22.666667 9.0
4 19.000000 12.0
5 23.000000 9.0

6 25.000000 7.0
7 29.000000 4.0

Example 5: Positional Filling Methods (Forward and Backward Fill)

Beyond scalar and statistical imputation, Pandas offers positional imputation techniques which are crucial for time series data or datasets where the sequential order of observations is meaningful. These methods utilize existing non-missing values adjacent to the NA values for replacement. The primary methods are 'ffill' (forward fill) and 'bfill' (backward fill).

Forward Fill (ffill) propagates the last valid observation forward until it encounters the next valid observation. This is useful when you assume the missing data point retains the value of the previous recorded measurement. Conversely, **Backward Fill (bfill)** uses the next valid observation to fill the missing gap preceding it. This is helpful when the data point is likely to be the same as the subsequent measurement.

To use these methods, we utilize the **method** parameter within the fillna() function. Unlike the scalar replacement methods, these positional fills do not require a `value` argument, but rather rely on the `method` parameter set to 'ffill' or 'bfill'. We can still restrict this operation to specific columns by selecting the columns before applying the fill method.

The following example demonstrates applying forward fill to the 'points' column and backward fill to the 'assists' column, showcasing how different sequential strategies can be applied simultaneously to different parts of the DataFrame.

 # Re-create fresh DataFrame for positional filling demonstration
df_pos = pd.DataFrame({'points': ,
'assists': })

# Apply forward fill (ffill) to 'points'
df_pos = df_pos.fillna(method='ffill')

# Apply backward fill (bfill) to 'assists'
df_pos = df_pos.fillna(method='bfill')

# View result
print(df_pos)

points assists
0 25.0 5.0
1 25.0 7.0

2 15.0 7.0
3 15.0 9.0
4 19.0 12.0
5 23.0 4.0
6 25.0 4.0
7 29.0 4.0

In the result, notice how the missing 'points' in row 1 was filled by the value from row 0 (25.0), and the missing 'points' in row 3 was filled by the value from row 2 (15.0). For 'assists', the missing values in rows 5 and 6 were filled by the value from row 7 (4.0).

Conclusion: Selecting the Right Imputation Strategy

The fillna() function is a cornerstone of data preparation in Pandas, providing flexible and powerful methods for addressing missing values. Whether you choose to use a simple scalar, a calculated aggregate (mean, median), or a sequential method (ffill, bfill) depends entirely on the nature of your data and the potential impact of the imputation on downstream analysis.

The key takeaway when dealing with multiple columns is to avoid global, uniform imputation unless absolutely necessary. Utilizing the dictionary-based approach, as demonstrated in Example 3 and Example 4, provides the highest degree of control, allowing you to tailor the imputation logic precisely to the requirements of individual columns.

By mastering these techniques--from simple column selection to advanced statistical replacement--you can ensure your DataFrames are clean, complete, and ready for accurate modeling and visualization.