How to Easily Fill NaN Values in Pandas DataFrames Using a Dictionary

Authored by

stats writer

November 21, 2025

RECOMMENDED CITATION

stats writer (2025). *How to Easily Fill NaN Values in Pandas DataFrames Using a Dictionary*. PSYCHOLOGICAL SCALES. Retrieved from https://scales.arabpsychology.com/?p=98816

When dealing with real-world datasets, encountering missing entries is inevitable. These gaps, often represented as NaN (Not a Number) values, pose significant challenges to accurate statistical analysis and machine learning model training. Effective handling of missing data is a critical step in the data preprocessing pipeline. One highly efficient and contextual method for managing these null entries involves conditional replacement based on known relationships defined within an external mapping structure, specifically a dictionary.

The standard approach in the Python data ecosystem utilizes the powerful Pandas DataFrame structure. While simple methods like mean or median replacement exist, they often ignore underlying relationships within the data. By employing a Python dictionary, we gain the ability to perform targeted data imputation, ensuring that the replacement value is logically consistent with other variables in the corresponding row. This method is particularly valuable when missing values are dependent on categorical features.

This sophisticated technique leverages the built-in fillna() function in conjunction with a mapping mechanism. Instead of simply providing a single scalar value for replacement, we generate a Pandas Series derived from a mapping dictionary. This Series acts as the input for the replacement operation, allowing us to fill NaNs in one column based on the unique categorical identifiers found in a separate, related column. Understanding this synergy is key to mastering advanced data cleaning workflows.

Understanding Missing Data and NaN Values in Data Analysis

Missing data is a common phenomenon that can drastically skew analytical results if not addressed properly. The designation NaN is the standard indicator used within numerical computing environments, especially those built on the IEEE 754 floating-point standard, to represent indeterminate or unrepresentable values. In the context of a Pandas DataFrame, these null entries require careful consideration to prevent errors during aggregation or modeling steps.

The decision on how to handle missing data often depends on the nature of the dataset and the reason for the missingness. Simple deletion (dropping rows with NaNs) can lead to a significant loss of information and potentially introduce selection bias if the data is not missing completely at random (MCAR). Therefore, data imputation--the process of replacing missing data with substituted values--is frequently the preferred strategy, preserving the overall size and structure of the dataset while making informed estimations for the gaps.

Traditional imputation methods, such as filling with the column mean, median, or mode, are computationally simple but lack context. For instance, if we are missing the 'Sales' value for a specific 'Store Location', simply using the average sales across all stores might be inaccurate if that specific location historically performs much better or worse than the average. The dictionary-

based method detailed here offers a significant methodological improvement by providing local, context-aware replacement values derived from specified categorical mappings, thereby enhancing the integrity of the imputed dataset.

Leveraging Pandas for Robust Data Imputation

The Pandas DataFrame is the cornerstone of data manipulation in Python, providing powerful methods designed specifically for data cleaning tasks. Central to handling missing data is the fillna() method. This function is exceptionally flexible, allowing users to input a wide range of replacement values, including scalar numbers, forward/backward fill methods, or, crucially for this technique, a Series or dictionary that dictates replacements on a per-index or per-column basis.

When working with conditional imputation--where the value used for filling depends on the value of another column--a direct approach using standard `fillna()` arguments often falls short. This is where the combination of `fillna()` and the Series **.map()** function becomes essential. The **.map()** function is designed to substitute values in a Series using an input dictionary. When applied to the column that contains the categorical keys (e.g., 'Store ID'), it generates a new Series containing the corresponding replacement values defined in the dictionary.

By passing this generated Series of replacement values into the `fillna()` method of the target column (the one containing the NaNs), Pandas efficiently aligns the replacement values based on the DataFrame index. This ensures that the correct replacement value is used for each missing entry, maintaining the row-wise context defined by the categorical column. This systematic and vectorized approach makes the process both highly efficient and programmatically clean.

The Power of fillna() and Dictionary Mapping

You can use the **fillna()** function with a dictionary to replace NaN values in one column of a Pandas DataFrame based on values in another column. This sophisticated technique is critical for ensuring data consistency when missingness is correlated with categorical features.

You can use the following basic syntax to achieve conditional filling:

```
#define dictionary
dict = {'A':5, 'B':10, 'C':15, 'D':20}

#replace values in col2 based on dictionary values in col1
df = df.fillna(df.map(dict))
```

The operation proceeds in two distinct steps within a single assignment line. First, `df.map(dict)` creates a temporary Pandas Series. For every row, this Series looks up the value in `col1` (the key)

and returns the corresponding value from the dictionary. This temporary Series is indexed identically to the original DataFrame.

Second, this generated Series is passed as the argument to the `df.fillna()` method. The `fillna()` function then intelligently checks the target column (`col2`). If a cell in `col2` contains NaN, it uses the value from the corresponding index in the temporary mapped Series for replacement. If the cell is not NaN, the original value in `col2` is retained. This chained method provides a concise and highly readable solution for complex conditional imputation tasks.

Practical Demonstration: Filling NaN Values in Sales Data

To illustrate the efficiency of this method, we will work through a concrete example involving sales data. Imagine we have a dataset tracking sales transactions where some sales figures are missing (NaN). We know, however, that the missing sales values correspond to specific store locations, and we have established baseline or estimated sales figures for each location based on historical data or standard operating procedures. This scenario perfectly justifies the use of dictionary-based imputation.

The following example shows how to use this syntax in practice.

Example: Fill NaN Values in Pandas Using a Dictionary

Suppose we have the following Pandas DataFrame that contains information about the sales made at various retail stores:

import pandas as pd
import numpy as np

#create DataFrame
df = pd.DataFrame({'store': ,
'sales': })

#view DataFrame
print(df)

store sales
0 A 12.0
1 A NaN
2 B 30.0
3 C NaN
4 D 24.0
5 C NaN

6 B NaN
7 D 13.0

We immediately observe that there are several null values present in the **sales** column. These missing values must be systematically addressed before any summary statistics or machine learning models can be reliably applied. Our goal is to leverage the information in the **store** column to provide meaningful substitutes for these missing sales figures, moving beyond a simplistic, overall average data imputation.

Defining the Imputation Mapping Dictionary

The success of this imputation method rests entirely on the quality and accuracy of the mapping dictionary we define. This dictionary serves as the lookup table, linking the categorical variables (the store identifiers) to the quantitative values we wish to use for replacement. Each key in the dictionary must correspond exactly to a unique value found in the categorical column (in our case, the `store` column), and the corresponding value must be the replacement figure for the target column (`sales`).

Suppose we have determined the following standard replacement values based on an independent assessment of each store's typical performance during periods when data might be missing:

Store A has an estimated missing sales value of 5.
Store B has an estimated missing sales value of 10.
Store C has an estimated missing sales value of 15.
Store D has an estimated missing sales value of 20.

This contextual information transforms what would otherwise be arbitrary replacement into informed estimation, significantly improving the quality of the cleaned dataset. The dictionary effectively encapsulates these business rules or statistical estimates for structured data cleaning.

Executing the Conditional Fill Operation

We are now ready to apply the conditional filling logic using the defined dictionary and the combined **.map()** and fillna() functions. We target the `sales` column for modification, using the `store` column as the key source for the mapping operation. The result of this process will be a modified `sales` column where null entries have been replaced according to the store type.

We can use the following syntax to do so:

#define dictionary
dict = {'A':5, 'B':10, 'C':15, 'D':20}

```
#replace values in sales column based on dictionary values in store column
df = df.fillna(df.map(dict))

#view updated DataFrame
print(df)
```

```
store sales
0 A 12.0
1 A 5.0
2 B 30.0
3 C 15.0
4 D 24.0
5 C 15.0
6 B 10.0
7 D 13.0
```

Note how the `df.map(dict)` step first generates the series of replacement values, which is then used by fillna() to conditionally insert values only where the original `sales` column had NaN. This ensures that existing, valid sales figures are preserved, while nulls are replaced based on the contextual key.

Analyzing the Imputation Outcomes

Upon reviewing the updated Pandas DataFrame, we can confirm that all missing values in the `sales` column have been successfully replaced according to the predefined mapping rules. The contextual filling is demonstrated by the specific replacements made:

If store is **A**, replace NaN in sales with the value **5**. (Applied to Row 1)
If store is **B**, replace NaN in sales with the value **10**. (Applied to Row 6)
If store is **C**, replace NaN in sales with the value **15**. (Applied to Rows 3 and 5)
If store is **D**, replace NaN in sales with the value **20**. (Not applicable in this dataset instance, but the rule was established.)

This method provides a robust and scalable solution for data imputation when replacement values are contingent upon existing categorical data. It avoids complex conditional loops and maintains the vectorized performance inherent to the Pandas library, leading to cleaner, more maintainable data preparation code.

You can find the complete online documentation for the **fillna()** function for further reference on its many powerful applications in data cleaning.

Conclusion: Streamlining Data Cleaning Workflows

The combination of Pandas **.map()** and **.fillna()** using a Python dictionary offers a superior technique for conditional missing data replacement. By moving beyond simple scalar imputation and adopting contextual mapping, data scientists can significantly enhance the accuracy and reliability of their foundational datasets, ensuring that subsequent analyses and models are built upon structurally sound and information-rich data. Mastering this technique is essential for anyone engaged in serious data preprocessing and cleaning.