Learning Pandas: A Practical Guide to Filling NaN Values with Dictionaries

Authored by

Mohammed looti

November 16, 2025

In the expansive and complex world of data analysis, data scientists frequently encounter missing data. This absence of information, often represented as NaN (Not a Number) values, poses a significant threat to the accuracy and reliability of any analytical conclusion. Effective handling of these gaps is paramount for maintaining data integrity. Fortunately, the widely adopted Pandas library within the Python ecosystem provides a robust suite of tools designed specifically for data cleaning and manipulation.

One of the most essential methods for addressing missing values is the `fillna()` function. While it is commonly used for simple replacements--such as filling all NaNs with a single constant or an aggregate statistic--its true power is unlocked when implemented in conjunction with a dictionary. This powerful combination enables conditional, context-aware replacement, allowing missing values in a target column to be imputed based on the corresponding category or condition found in a separate reference column.

This article serves as an in-depth guide to leveraging the `fillna()` function alongside a Python dictionary. We will explore the mechanism behind this technique, walk through a highly practical example using sales data, and discuss why this targeted approach to imputation is superior for enhancing the quality and contextual relevance of your data processing workflows.

The Mechanism of Dictionary-Based NaN Imputation

The core philosophy of this technique centers on creating a dynamic mapping system. Instead of applying a uniform fill value across an entire column, we establish a Python dictionary where the keys represent the unique values from a reference column (e.g., 'Store Type'), and the corresponding values are the desired replacements for the missing data in the target column (e.g., 'Sales').

This replacement process is carried out efficiently through the use of the `Series.map()` method. When applied to the reference column, `map()` generates a new Pandas Series where each value is replaced according to the rules defined in the dictionary. If a key is not found in the dictionary, `map()` automatically inserts a NaN, a behavior that is crucial for the subsequent step.

Finally, this newly generated replacement Series is passed directly into the `fillna()` method of the target column. The genius of `fillna()` is its selectivity: it only substitutes values where the original data point was NaN. All existing, non-missing values in the target column are left completely untouched, ensuring data preservation while applying the conditional fills seamlessly.

General Syntax and Code Illustration

Understanding the sequence of operations is key to mastering this technique. The following syntax demonstrates how to define the categorical mapping and then execute the conditional fill operation

within a Pandas DataFrame:

 # Define a dictionary where keys map to values from the reference column ('col1')
 # and values are the specific fill values for the target column ('col2').
dict = {'A':5, 'B':10, 'C':15, 'D':20}

# Step 1: Use .map() on the reference column ('col1') to generate replacement values.
# Step 2: Pass the resulting Series into the .fillna() method of the target column ('col2').
df = df.fillna(df.map(dict))

The elegance of this approach lies in its conciseness. When `df.map(dict)` is executed, it produces a complete Series of potential replacement values, indexed identically to the DataFrame. The final application of `fillna()` then selectively merges this replacement data, ensuring that only the truly missing records receive the contextually appropriate value derived from the mapping dictionary.

Setting Up the Practical Sales Data Example

To demonstrate the utility of dictionary-based filling, let us consider a common business scenario: handling missing sales figures across different retail store categories. Suppose we have a Pandas DataFrame containing `store` types and their corresponding `sales` records, where some sales data is missing.

Before proceeding, we must import the necessary libraries: Pandas for data structuring and NumPy for easily generating the NaN placeholders. We then construct our initial sample dataset:

import pandas as pd
import numpy as np

# Create DataFrame with 'store' (reference column) and 'sales' (target column)
df = pd.DataFrame({'store': ,
'sales': })

# View the initial DataFrame structure
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

As clearly shown in the output, the `sales` column contains multiple NaN values. If we were to use a simple method, like filling all missing sales with the overall average sales (19.75), we would lose the crucial context that sales figures naturally differ based on the `store` type. Our goal is to perform a targeted imputation, replacing the missing sales for Store 'A' with a value specific to 'A', Store 'B' with a value specific to 'B', and so forth, based on predefined business rules.

Executing the Conditional Fill Strategy

To perform the context-dependent replacement, we must first define the mapping rules. In a real-world scenario, these values might come from external benchmarks, statistical averages calculated per store type, or management-defined default targets. For this example, we define a simple dictionary that assigns a unique default sales value to each store category.

We then apply the conditional fill operation using the chain of `map()` and `fillna()`, overwriting the original `sales` column with the imputed values:

 # Define the dictionary mapping store types to their default sales targets
dict = {'A':5, 'B':10, 'C':15, 'D':20}

# Use map() on 'store' to generate replacement Series, then use fillna() on 'sales'
df = df.fillna(df.map(dict))

# View the 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

The result clearly demonstrates the success of the conditional imputation. The original non-NaN values (e.g., 12.0, 30.0) remain unaltered. However, the missing sales for Store 'A' (index 1) are now 5.0, those for Store 'C' (indices 3 and 5) are 15.0, and Store 'B' (index 6) is 10.0. This ensures that the imputed sales figures are aligned with the categorical context, offering a far more meaningful and reliable dataset for subsequent analysis compared to generic replacement methods.

Benefits and Caveats of Contextual Imputation

The dictionary-based approach provides substantial benefits, primarily related to data quality and workflow maintenance. Firstly, it offers unmatched **flexibility and control**. Data analysts can define specific, non-statistical replacement values derived from external factors, business expertise, or domain knowledge for each category. This prevents the bias introduced by applying a single global statistic across heterogeneous groups.

Secondly, this method greatly improves code **readability and maintainability**. The logic for missing value handling is neatly encapsulated within the Python dictionary. Should the underlying business rules or default values change, only the dictionary requires updating, minimizing the complexity of managing the data cleaning pipeline.

However, analysts must be aware of how unmapped values are handled. If the reference column contains a category that is not present as a key in the dictionary, the `Series.map()` function will generate a NaN for that row. If the corresponding target column value was already missing, it will remain NaN after the `fillna()` operation. This intentional behavior ensures that only explicitly defined rules lead to imputation, preventing accidental data modification where no replacement rule was intended.

Conclusion and Final Recommendations

Mastering the technique of using the `fillna()` function in conjunction with `Series.map()` and a mapping dictionary is an essential skill for any data professional working with Pandas. This approach offers a highly precise and elegant solution for performing context-aware imputation, linking missing values directly to categorical attributes within the DataFrame.

By defining explicit replacement rules via the dictionary, data analysts move beyond arbitrary approximations and ensure that imputed values reflect specific business logic or domain expertise. While this technique is ideal for categorical data mapping, remember that Pandas provides alternative methods, such as `groupby().transform()`, which may be more suitable for deriving fill values based on group statistics (like mean or median) in larger datasets. Nonetheless, for straightforward, rule-based replacement, the dictionary-based method remains unmatched in its efficiency and clarity.

Integrating this powerful technique into your data cleaning repertoire will significantly improve the semantic integrity and reliability of your datasets, leading to more trustworthy and interpretable analytical results. Always ensure your chosen imputation strategy aligns with the specific nature of your missing data and the goals of your overall data analysis project.

Additional Resources for Pandas Data Manipulation

To further explore the capabilities discussed in this article, please refer to the official Pandas documentation:

Pandas DataFrame.fillna official documentation
Pandas Series.map official documentation