Understanding and Resolving the “ValueError: cannot convert float NaN to integer” Error in Pandas

Authored by

Mohammed looti

November 2, 2025

The `ValueError: cannot convert float NaN to integer` is one of the most frequently encountered errors when performing critical **data cleaning** and type conversion operations within the pandas library. This exception serves as a strict warning, signaling a fundamental incompatibility between how standard numeric data type representations in Python and NumPy handle missing values. Resolving this error is essential for maintaining data integrity and ensuring efficient computational pipelines.

Specifically, this technical issue arises when a user attempts to force a column in a DataFrame--which is currently stored as a float--to convert to an integer data type, but that column contains one or more instances of **Not a Number** (NaN). Since standard integers are incapable of representing missingness, the conversion fails immediately, leading to the runtime error.

Successfully navigating this problem requires a clear understanding of why NaN exists only within the float framework and how to properly preprocess data before attempting type casting. The comprehensive guide below provides a detailed breakdown of the technical conflict, offers a reproducible example, and outlines the three most robust solutions, including leveraging modern nullable integer types in pandas.

ValueError: cannot convert float NaN to integer

Understanding the Data Type Conflict: Float vs. Integer Representation

To fully appreciate the cause of this conversion failure, we must first recognize the fundamental differences in how standard Python and pandas manage numerical data types. Standard integer types, such as `int64`, are designed exclusively for representing whole numbers. Crucially, these types do not adhere to any standard that allocates a special bit pattern to denote missing or undefined values; they are inherently non-nullable.

In sharp contrast, the float data type (typically `float64`) is governed by the IEEE 754 standard. This standard explicitly defines special values like positive infinity, negative infinity, and the all-important **Not a Number** (NaN). Because `float64` is the only standard numeric type capable of holding this special marker, when pandas reads a dataset containing numeric columns with missing entries, it automatically promotes those columns to the float type to safely accommodate the NaN markers.

The core conflict arises precisely when you attempt to execute the `.astype(int)` method on a column that contains NaN values. The system attempts to assign a whole number representation to a non-numeric, special float value. Since a standard integer cannot logically represent a missing value, Python correctly throws the **ValueError**, preventing data corruption by halting the impossible conversion.

Reproducing the ValueError in a Pandas DataFrame

A practical demonstration helps solidify the conceptual understanding of this error. We can easily replicate this scenario by creating a simple DataFrame where missing values are intentionally introduced. To inject these missing values, we rely on the industry-standard NumPy library, specifically using `np.nan`.

The following code snippet initializes a sample dataset for a sports team, where the 'rebounds' column simulates real-world data gaps by including missing observations. Note how the creation process immediately forces the 'rebounds' column to be represented as a float due to the presence of `np.nan`.

import pandas as pd
import numpy as np

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

#view DataFrame
df

points assists rebounds
0 25 5 11
1 12 7 NaN
2 15 7 10
3 14 9 6
4 19 12 5
5 23 9 NaN
6 25 9 9
7 29 4 12

Despite the 'rebounds' column primarily containing whole numbers, its data type is confirmed as `float64` due to the presence of `np.nan`. If we now attempt the conversion to a standard integer type, the system recognizes the inherent incompatibility and generates the expected error, immediately halting any further processing on that column:

#print data type of 'rebounds' column
df.dtype

dtype('float64')

#attempt to convert 'rebounds' column from float to integer
df = df.astype(int)

ValueError: cannot convert float NaN to integer

This outcome confirms that the presence of **NaN** is the singular blocking factor preventing the successful conversion to a standard, non-nullable integer column. To proceed with the desired type change, we must first implement a strategy to manage or eliminate these missing values.

Solution Strategy 1: Data Cleaning by Dropping Missing Values (`dropna`)

For situations where the missing data is sparse or the rows containing missing values are not critical to the overall analysis, the simplest and most immediate solution is to remove the offending rows. This approach relies on the pandas `.dropna()` function to eliminate all instances of NaN from the target column, thus ensuring the remaining data is clean and ready for type conversion.

Before dropping data, it is often good practice to inspect the affected rows using boolean indexing combined with the `.isnull()` method. This allows the user to quantify the data loss risk before executing the deletion. Once the missing values are identified, we apply `.dropna()`, which, by default, removes any row containing at least one missing value across the specified column or the entire DataFrame.

#print rows in DataFrame that contain NaN in 'rebounds' column
print(df.isnull()])

points assists rebounds
1 12 7 NaN
5 23 9 NaN

#drop all rows with NaN values
df = df.dropna()

#convert 'rebounds' column from float to integer
df = df.astype(int)

#view updated DataFrame
df
points assists rebounds
0 25 5 11
2 15 7 10

3 14 9 6
4 19 12 5
6 25 9 9
7 29 4 12

#view class of 'rebounds' column
df.dtype

dtype('int64')

The conversion is now successful, and the column is correctly stored as `int64`. While this method is computationally fast, analysts must exercise caution, as dropping too many rows can significantly reduce sample size and potentially introduce selection bias into the data.

Solution Strategy 2: Imputation Using Replacement Values (`fillna`)

When preserving all existing observations is a priority, imputation provides the necessary alternative to dropping rows. Imputation involves replacing the missing NaN values with a plausible, fixed numerical placeholder. By ensuring that every cell in the column holds a valid whole number, the column instantly becomes compatible with the standard integer data type.

The specific value chosen for replacement depends heavily on the domain knowledge and context of the data. For count data or metrics where missingness suggests absence, replacing NaN with zero (0) is common. Other robust strategies include replacing missing values with the median or mode of the column, which tends to be less sensitive to outliers than the mean. However, any form of imputation should be approached carefully, as it introduces synthetic data points that may subtly bias statistical results.

For the purpose of resolving the **ValueError** quickly and preserving the sample size, we will apply the simple imputation method: replacing all missing values in the 'rebounds' column with zero (0), thereby enabling the conversion without data loss.

#replace all NaN values with zeros
df = df.fillna(0)

#convert 'rebounds' column from float to integer
df = df.astype(int)

#view updated DataFrame
df

points assists rebounds

0 25 5 11
1 12 7 0
2 15 7 10
3 14 9 6
4 19 12 5
5 23 9 0
6 25 9 9
7 29 4 12

#view class of 'rebounds' column
df.dtype

dtype('int64')

By ensuring that no NaN values remain, we successfully eliminate the cause of the **ValueError**, allowing the `.astype(int)` operation to complete smoothly.

Leveraging Modern Pandas: The Dedicated Nullable Integer Dtype (Int64)

For data professionals using modern versions of pandas (version 0.24.0 and later), there is a vastly superior solution that addresses the core incompatibility without requiring data deletion (`dropna`) or potentially biased modification (`fillna`). Pandas introduced specialized nullable integer data types, known as "Extension Dtypes," which are specifically engineered to accommodate missing values within an integer column context.

These nullable data type representations, such as `Int64`, use a special internal marker (`pd.NA`) instead of the traditional floating-point `np.nan` to signify missingness. This crucial distinction means the column can retain its integer behavior while preserving the information that certain data points are missing. The primary types are distinguished by a capital "I" (e.g., `'Int64'`, `'Int32'`).

By simply passing the string `'Int64'` to the `.astype()` method, we achieve the desired integer data type while simultaneously handling the NaN values gracefully. This is considered the best practice for maintaining data fidelity in analytical workflows where missing data is expected.

 # Re-create the original DataFrame (assuming df contains NaN again)
 # df = pd.DataFrame(...)

#convert 'rebounds' column using the nullable integer type
df = df.astype('Int64')

#view updated DataFrame and its dtype

df

points assists rebounds
0 25 5 11
1 12 7
2 15 7 10
3 14 9 6
4 19 12 5
5 23 9
6 25 9 9
7 29 4 12

df.dtype

Int64Dtype()

This approach is highly recommended for professional data science workflows as it allows for precise type specification without forcing data modification or deletion, making it the most accurate way to represent integer data with gaps.

Conclusion and Best Practices for Handling Missing Data

The **ValueError: cannot convert float NaN to integer** is a predictable consequence of attempting to map the float representation of missing data (NaN) onto a standard, non-nullable integer column. To successfully complete the type conversion, the missing values must be addressed preemptively using one of three proven strategies.

When deciding how to resolve this error in practice, data scientists should evaluate the trade-offs of each method based on the analytical goals and the nature of the missing data:

Dropping Rows (`.dropna()`): This is suitable only for datasets where the quantity of missing data is small and the analyst is willing to accept a small degree of data loss. It provides the quickest fix but carries the greatest risk of altering the sample distribution.

Imputation (`.fillna()`): This technique preserves the full row count of the DataFrame. It is best used when data retention is paramount, but analysts must be cautious about the choice of replacement value (e.g., 0, median), as this introduces potential analytical bias.

Nullable Integers (`.astype('Int64')`): This is the most robust and modern solution for pandas users. It allows the column to be stored as an integer type while retaining the missing data markers, offering the best balance between accurate type representation and data integrity.

By understanding the interaction between NaN and standard numerical types, we can choose the appropriate data management technique, ensuring smooth execution of type conversions and maintaining the structure of the data for downstream analysis.

Additional Resources

For further reading on common Python and data manipulation errors, consider reviewing tutorials covering topics such as:

Handling `TypeError` when mixing data types.

Strategies for advanced time series imputation.

Optimizing memory usage by selecting appropriate data type formats in pandas.