How to Fix “TypeError: no numeric data to plot” in Pandas

Authored by

stats writer

December 3, 2025

The infamous `TypeError: no numeric data to plot` is a frequent stumbling block for data analysts utilizing the Pandas library in Python. This error typically surfaces when a user attempts to visualize data from a Pandas DataFrame or Series, but the underlying columns intended for plotting do not contain appropriate numeric data. Visualization libraries like Matplotlib (which Pandas utilizes under the hood for its `plot()` method) strictly require data to be numerical--specifically integers or floating-point numbers--to render meaningful graphical representations, such as line graphs or scatter plots.

To fix this common issue, the analyst must convert the non-numeric data, which is often stored as **`object`** (string) types, into a recognized numerical format. This transformation is generally accomplished using powerful Pandas methods such as `.astype()` or the more robust `pd.to_numeric()` function. Once this essential data transformation is complete and the data is in the correct numerical format (e.g., **`float64`**), the visualization process can proceed without error, providing the intended insights from the dataset.

Understanding the "no numeric data to plot" Error

One common challenge faced when manipulating datasets with Pandas is ensuring that the data types align with the operations being performed. The specific error message, **`TypeError: no numeric data to plot`**, arises precisely when the plotting function attempts to iterate through columns expecting measurable quantities, only to find data stored as strings or other non-numeric types. This usually happens because data ingestion processes--such as reading CSV files or scraping web data--often default columns containing seemingly numeric values (like '5', '7', '12') to the generic **`object`** type, which Pandas interprets as textual data.

The core of the problem lies in the fact that visualization tools cannot interpret text strings as points in a quantitative space. For a line plot, the tool must know the magnitude of the values to correctly position them on the Y-axis. When a column is tagged as **`object`**, the system treats it similarly to categorical names or arbitrary text, rendering plotting impossible. Therefore, the immediate diagnostic step is always to verify the underlying data types of the columns targeted for visualization, confirming whether they are indeed stored as expected **`int64`** or **`float64`** types.

This tutorial provides a step-by-step walkthrough, starting from reproducing this error using a typical scenario, diagnosing the data type mismatch, and applying the correct conversion methods to successfully generate the required plot. We will focus specifically on how to efficiently transform string representations of numbers into usable numerical formats within a Pandas DataFrame.

Upon execution of a plotting command on non-numeric data, the system returns:

TypeError: no numeric data to plot

This `TypeError` immediately indicates that the selected columns lack the required quantitative structure for graphical representation. This frequently occurs when a user assumes certain columns are numerical based on their content, but the underlying Pandas data types dictate otherwise.

Setting Up the Example: Initial DataFrame Creation

To illustrate this problem clearly, we will construct a small DataFrame designed to mimic real-world data collection issues. In this scenario, variables that should inherently be numerical--such as points scored, rebounds grabbed, and blocks recorded--are inadvertently loaded as strings (or `object` types in Pandas terminology) due to how the data was initially structured or read into the Python environment.

We use the `pd.DataFrame()` constructor to explicitly define our dataset. Crucially, the values for 'points', 'rebounds', and 'blocks' are enclosed in single quotes (e.g., `'5'`), which forces Pandas to treat them as string literals, not numerical values, upon creation. This seemingly minor structural decision is the root cause of the plotting error we are about to encounter.

The resulting DataFrame captures typical sports statistics data:

import pandas as pd

# Create the DataFrame where numeric columns are stored as strings
df = pd.DataFrame({'team': ,
'points': ,
'rebounds': ,
'blocks': })

View the resulting DataFrame
df

team points rebounds blocks
0 A 5 11 4
1 A 7 8 7
2 B 7 10 7
3 B 9 6 6
4 B 12 6 5

Attempting to Plot and Reproducing the Error

Having established our dataset, the next logical step in data analysis is often visualization to

quickly identify trends or outliers. Our objective is to generate a line plot comparing the trends across 'points', 'rebounds', and 'blocks' over the recorded entries. We select the relevant columns using standard DataFrame indexing and chain the `.plot()` method onto the resulting subset.

When this visualization command is executed, the plotting backend checks the data types of the columns passed to it. Because the columns are currently stored as **`object`** (strings), the function cannot determine how to scale the axes or connect the points, leading directly to the termination of the operation and the issuance of the error message. This illustrates the critical distinction between a string containing a number and an actual numerical type in the context of quantitative analysis.

Executing the plotting command yields the expected failure:

 # Attempt to create a line plot for points, rebounds, and blocks
df].plot()

ValueError: no numeric data to plot

We receive this error because the plotting engine cannot process textual data as quantitative coordinates. None of these selected columns are recognized as true numeric data by the visualization tools.

Diagnosing Data Types Using .dtypes

The most immediate and reliable way to confirm the cause of the plotting error is to inspect the data types (dtypes) of the columns within the DataFrame. The Pandas `.dtypes` attribute provides a clear summary of how each column is currently interpreted by the system. This diagnostic step should always precede any data manipulation when unexpected behavior occurs, particularly type-related errors.

By applying the `.dtypes` attribute to our DataFrame `df`, we can definitively see which columns are designated as **`object`** types and which are numerical. For our example, we expect all the statistic columns to be non-numerical, confirming our hypothesis about the source of the plotting error.

Executing the diagnostic check reveals the current data structure:

 # Display data type of each column in DataFrame
df.dtypes

team object
points object
rebounds object
blocks object

dtype: object

As shown, all four columns are categorized as **`object`**. In the context of Pandas, the **`object`** data type is primarily used for strings, confirming precisely why the `plot()` function failed: none of the columns supplied contained true numerical values.

Strategies for Converting Non-Numeric Data

Once the data type issue is confirmed, the immediate solution is data type coercion. Pandas provides several robust methods to convert columns from **`object`** to a suitable numeric data type, such as **`int64`** or **`float64`**. The choice of method often depends on the complexity of the conversion required and how missing or invalid data should be handled.

The two primary tools for this task are `.astype()` and `pd.to_numeric()`. The `.astype()` method is generally preferred for straightforward conversions where we are confident that the entire column contains only valid representations of numbers. It is fast and efficient. However, if the data might contain non-numeric characters (like 'N/A' or hyphens), `pd.to_numeric()` is more flexible, as it offers the **`errors='coerce'`** option, which automatically replaces non-convertible values with **`NaN`**, preventing the script from crashing.

For the purposes of our clean example data, using `.astype()` is the most direct and idiomatic Pandas solution. Since plotting functions are highly compatible with floating-point numbers, we will convert the columns to the **`float`** data type to ensure precision.

Implementing the Fix: Using .astype() for Conversion

To rectify the data types in our example, we must iterate through the 'points', 'rebounds', and 'blocks' columns and explicitly cast their contents to **`float`**. This operation overwrites the existing **`object`** dtypes with the new, numerical **`float64`** type, ensuring that the plotting function will recognize them as valid quantities. It is essential to assign the result of the `.astype()` operation back to the respective column in the DataFrame to permanently save the changes.

We apply the conversion sequentially to the three critical columns:

 # Convert points, rebounds, and blocks columns to the float data type
df=df.astype(float)
df=df.astype(float)
df=df.astype(float)

This conversion is now complete. The DataFrame now holds numerical representations of the

statistics. We can now safely reattempt the visualization process, confident that the data types meet the prerequisites set by the plotting library.

Verifying the Data Type Transformation

Before proceeding with the plot, it is prudent to confirm that the data type conversion was successful. We utilize the `.dtypes` attribute one final time to inspect the structure of the modified DataFrame. This verification step confirms that the columns are now properly designated as **`float64`**, providing proof that the remedial action was effective.

A successful conversion will show 'points', 'rebounds', and 'blocks' as **`float64`**, while 'team' correctly remains an **`object`** type, as it is a categorical variable.

Running the verification check:

 # Display data type of each column in DataFrame after conversion
df.dtypes

team object
points float64
rebounds float64
blocks float64
dtype: object

The output confirms the successful conversion: the 'points', 'rebounds', and 'blocks' columns are now correctly identified as **`float64`**. This numerical designation is exactly what the plotting function requires to calculate and render the graphical output accurately.

Successful Visualization of Numeric Data

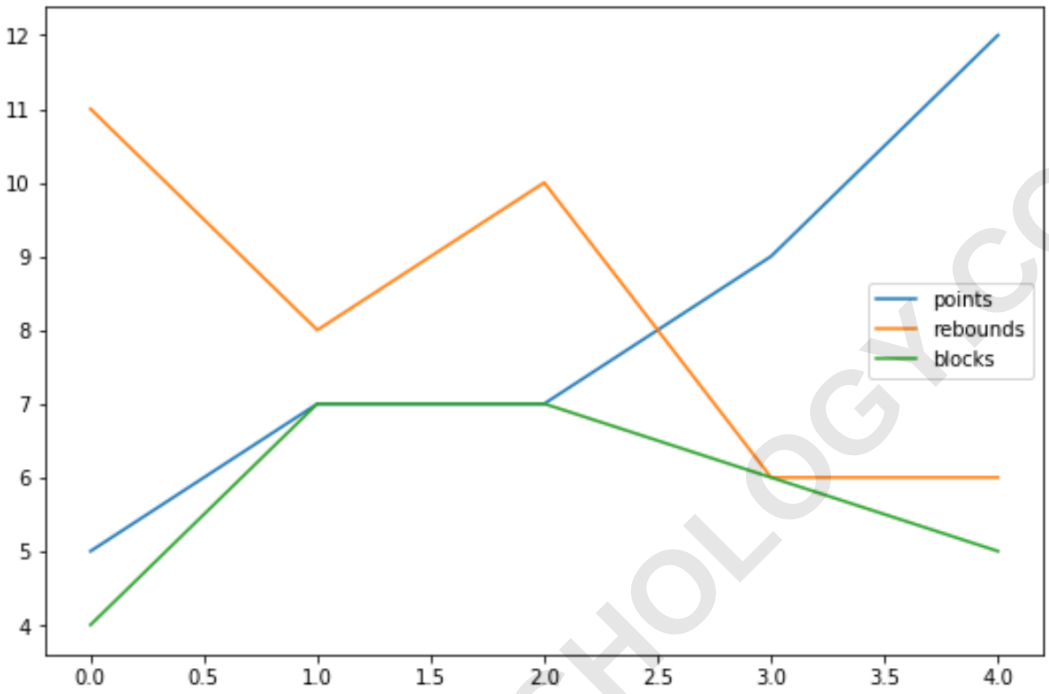
With the data types correctly set, we can now execute the plotting command that previously failed. Because the columns are now recognized as bona fide numeric data, the Pandas `.plot()` method seamlessly utilizes the underlying visualization backend (Matplotlib) to generate the requested line plot.

This successful outcome underscores the importance of data preparation and validation in any data science workflow. Simple type mismatches, though often overlooked, are a common source of runtime errors, particularly in visualization tasks.

We re-run the plotting code:

 # Create line plot for points, rebounds, and blocks using the converted data
df].plot()

The code now executes without error, producing the visual representation of the data:



We have successfully created a line plot showing the trends for points, rebounds, and blocks across the data entries, confirming that the conversion of the data types from **`object`** to **`float64`** was the necessary and sufficient fix for the initial type error.

Alternative Conversion Method: pd.to_numeric()

While `.astype()` is excellent for clean data, it is worthwhile to briefly discuss `pd.to_numeric()`. This function is often more robust when dealing with real-world data containing anomalies. If, for instance, a row in the 'points' column had the entry "missing" instead of a number, `.astype(float)` would raise an immediate **`ValueError`**, halting execution.

`pd.to_numeric()` addresses this with the **`errors='coerce'`** argument. When this argument is used, any value that cannot be converted to a number is automatically replaced with **`NaN`** (Not a Number), allowing the conversion to complete successfully. The resulting **`NaN`** values are typically skipped or handled gracefully by plotting functions, ensuring the script runs smoothly even with imperfect data.

For columns that are expected to be numerical but might contain dirty data points, the use of `df =`

`pd.to_numeric(df, errors='coerce')` is the recommended best practice, providing fault tolerance in complex data pipelines.

Key Takeaways for Data Visualization in Pandas

The primary lesson derived from this error is the fundamental requirement for numerical data types in quantitative plotting. Whenever a visualization function fails with a "no numeric data" error, the analyst must immediately inspect the column dtypes. If columns that appear numerical are listed as **`object`**, conversion is mandatory.

Key actions to ensure successful plotting:

Diagnosis: Always start by checking `df.dtypes`.
Conversion Choice: Use `.astype()` for clean data, or `pd.to_numeric()` with `errors='coerce'` for messy data.
Re-verification: Confirm the change using `.dtypes` before attempting the `plot()` command again.

Adhering to these steps ensures a robust and reliable data analysis workflow, minimizing unexpected runtime errors caused by mismatched data formats.

Conclusion and Further Resources

The **`TypeError: no numeric data to plot`** is a strong reminder that in data analysis, content (what the data looks like) does not always equal type (how the software interprets the data). By employing diagnostic tools like **`.dtypes`** and powerful transformation functions such as **`.astype()`**, analysts can quickly resolve this issue and proceed with meaningful data visualization.

For those looking to troubleshoot other common pitfalls in Python data handling, particularly concerning data type manipulation and error resolution, the following resources may be helpful:

The following tutorials explain how to fix other common errors in Python: