How to Easily Fix “ValueError: Trailing Data” in Python

Authored by

stats writer

December 3, 2025

The ValueError: Trailing data is a common hurdle encountered by developers working with data ingestion and parsing in Python. This specific exception signifies that after a function successfully parsed the expected data structure--such as a delimited record or a valid JSON object--it found additional, unexpected characters or bytes remaining in the input stream. Essentially, the parser reads what it expects to be the end of the input, but the file or string still contains "trailing data."

The presence of trailing data often indicates a mismatch between how the data is structured and how the parsing function is configured to read it. For instance, if you are reading multiple, distinct records written sequentially without proper separators, or if metadata remains in the buffer after the primary data payload has been consumed, this error will be raised. To resolve this issue, the core objective is always twofold: either the extraneous data must be systematically removed before parsing, or, more commonly in structured data scenarios like those involving pandas DataFrames, the parser's configuration must be adjusted to correctly handle the multi-record nature of the input file.

This article specifically addresses this ValueError as it relates to reading JSON files using the powerful read_json function in pandas. We will explore why this occurs when dealing with line-delimited JSON (JSON Lines) format and provide the definitive, easiest method for resolution, ensuring clean and efficient data loading into your analytical environment.

Understanding the Nature of Trailing Data Errors

When any deserializer or parser attempts to convert a string or file stream into an object, it follows strict rules defining where the object begins and where it terminates. In formats like JSON, a single object must be delimited by curly braces `{}` or square brackets for an array. When a parser finishes identifying the closing delimiter for the expected structure, it then checks if the input stream has reached its end (End of File or End of String). If the parser detects any non-whitespace characters after the expected end, it correctly throws the "Trailing data" exception, as it cannot interpret this subsequent data within the context of the initial object it just successfully built.

This error is particularly common when developers attempt to load files that contain multiple valid JSON objects separated by newline characters (often referred to as JSON Lines or JSONL format). Standard JSON parsing libraries, including the default behavior of `json.loads()` or pandas' `read_json()`, are designed to consume only a single, monolithic JSON structure per invocation. Consequently, if a file contains `{"key": 1}n{"key": 2}`, the parser reads the first object `{"key": 1}`, finds the newline and the start of the second object `{"key": 2}`, identifies this as unexpected trailing data, and halts execution immediately. This strict adherence ensures data integrity by preventing accidental concatenation or misinterpretation of sequential records.

It is critical to distinguish this error from other parsing issues, such as syntax errors or malformed

data. A syntax error occurs when the internal structure of the object is broken (e.g., missing commas or mismatched quotes). The "Trailing data" error, conversely, indicates that the internal structure of the first object was **perfectly valid**, but the subsequent text (the trailing data) makes the entire input stream invalid for single-object parsing. Understanding this distinction points us directly toward solutions that involve either iterative parsing (reading one line/object at a time) or, more simply, utilizing specific library parameters designed for multi-record ingestion, as we will demonstrate with pandas.

Why This Error Occurs with JSON and Pandas

When utilizing the powerful pandas library for data analysis in Python, loading external data sources like JSON is a frequent operation. The primary function for this task is `pd.read_json()`. By default, this function expects the input file to contain a single, valid JSON document, which typically represents a list of records or a single dictionary whose structure is easily convertible into columns and rows within a pandas DataFrame. This expectation aligns with standard JSON interchange formats.

However, many large-scale data systems, especially those dealing with streaming data or log files, prefer the JSON Lines (JSONL) format. In this format, each line of the file is a separate, self-contained JSON object, delimited only by a newline character (`n`). This format is highly desirable for its streamability--it allows processing to begin before the entire file is received--and fault tolerance, as the corruption of one record does not affect the others. When `pd.read_json()` attempts to process a JSONL file without explicit configuration, it reads the first line, successfully parses the object, and then attempts to read the rest of the file. Since the rest of the file begins immediately with a newline and then the subsequent valid JSON object, the function interprets this as illicit content following the expected single object, thus raising the `ValueError: Trailing data` exception.

Furthermore, internal structure issues within the data itself can also trigger this error. Specifically, if a string field within a JSON object contains unescaped newline characters (`n`), the standard parser may prematurely interpret that newline as the end of the JSON record, even though the overall object definition is not complete. In the example provided in the original content, the "Review" text contains **n** characters intended to represent line breaks within the data itself. If these are not properly handled or escaped during the initial JSON creation, they can confuse the parser, leading to an early termination of object recognition and the subsequent identification of the rest of the file as trailing data. This dual challenge--handling both external record delimiters and internal string escape sequences--requires careful parameter selection within the pandas function.

One specific error you will encounter when attempting to load JSON Lines data using the default pandas configuration is:

ValueError: Trailing data

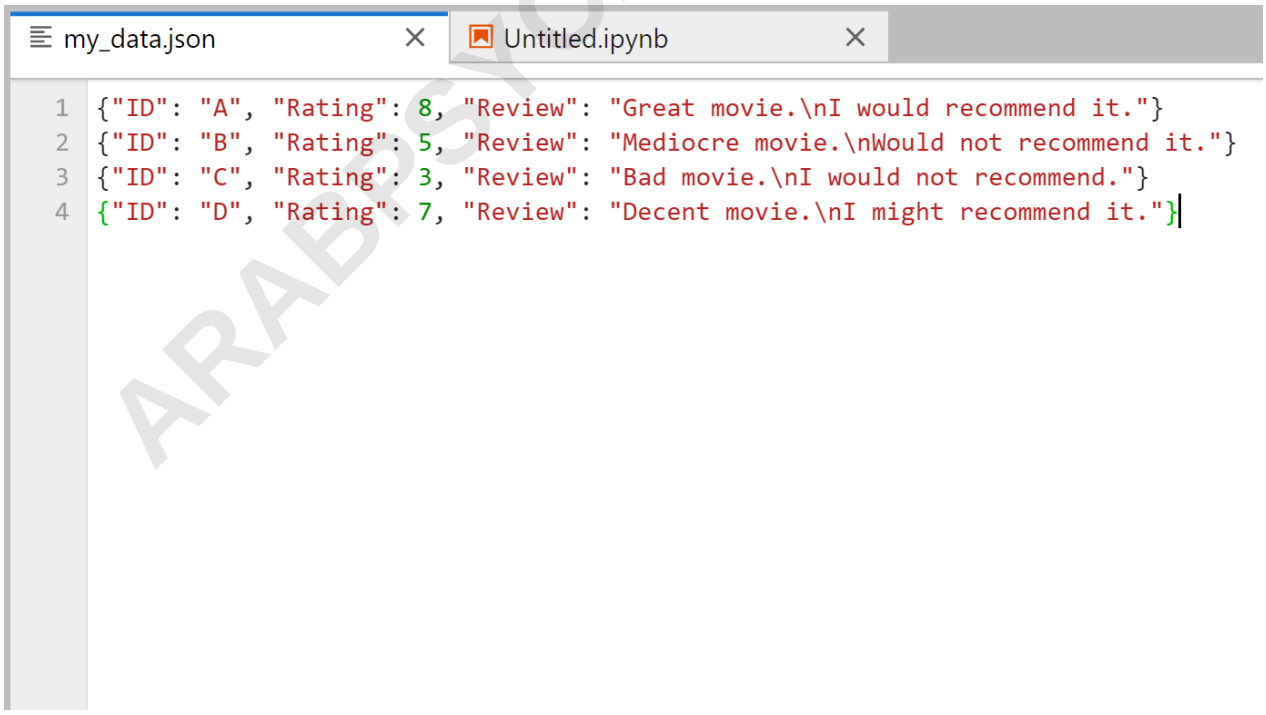
This exception is consistently raised when you try to import a JSON file into a pandas DataFrame where the individual data records are separated by endlines, often represented internally as '**n**'.

Step-by-Step: Reproducing the Error Scenario

To fully appreciate the solution, it is helpful to first recreate the exact conditions that lead to the ValueError: Trailing data. This scenario typically involves a data file structured in the JSON Lines format, where each independent record is placed on a new line. For demonstration, we will consider a fictional dataset contained within a file named `my_data.json`, which holds review data for several items.

The structure of this file is crucial. Notice how the "Review" field spans multiple lines internally, often utilizing the `n` character to denote line breaks within the text string. More importantly, each distinct record (ID, Rating, Review) forms its own complete JSON object, separated from the next object by a physical newline in the file. This structure, while efficient for streaming, violates the single-object expectation of the default `pd.read_json()` method.

Suppose we have the following JSON file structure:



Now, we attempt to ingest this multi-record file directly into a pandas DataFrame without specifying the correct parsing mode. We will use the standard syntax, expecting pandas to figure out the

structure, which results in failure:

 # Attempt to import JSON file structured in JSONL format into pandas DataFrame
df = pd.read_json('Documents/DataFiles/my_data.json')

ValueError: Trailing data

As demonstrated, the execution immediately halts, and the ValueError is raised. This confirms that the parser successfully read the first JSON object but failed upon encountering the newline character and the start of the second object, which it flagged as unprocessable trailing data. This failure clearly signals that we must instruct pandas to treat the input stream not as one large object, but as a sequence of independent lines, each containing a valid record.

The Primary Solution: Utilizing the lines=True Parameter

The definitive and most straightforward method to resolve the ValueError: Trailing data when loading JSON Lines files into pandas is by leveraging the **`lines=True`** parameter within the `pd.read_json()` function. This parameter fundamentally changes the behavior of the pandas parser, shifting its expectation from a single monolithic JSON document to a stream of independent, newline-delimited JSON objects.

When **`lines=True`** is set, pandas effectively iterates over the input file line by line. It treats each physical line as a separate, complete JSON string, parses that string into a record (a row), and then moves immediately to the next line without checking for an End of File marker until it reaches the true end of the input stream. This approach perfectly accommodates the JSON Lines format, where multiple records exist but are logically separated by the newline character (**`n`**). This specific configuration is a powerful demonstration of how well Python data libraries are optimized for handling common data ingestion patterns.

Using this parameter not only fixes the immediate "Trailing data" error but also significantly improves the efficiency and robustness of the data loading process when dealing with large JSONL files. Instead of loading the entire file into memory and attempting to parse it as one unit (which can be memory-intensive and brittle), pandas processes it sequentially. This ensures that even if individual records are very large, the overall ingestion process remains performant and avoids unnecessary exceptions related to unexpected structural delimiters. Therefore, specifying **`lines=True`** is considered the best practice whenever you are certain your source data adheres to the JSON Lines format.

Practical Implementation of the Fix

The implementation of the fix is remarkably simple, requiring only the addition of the **`lines=True`**

argument to the function call that previously failed. This adjustment instructs pandas to correctly interpret the newline separators within the `my_data.json` file, allowing it to load the entirety of the input successfully into a pandas DataFrame.

The easiest way to fix this error is to simply specify **lines=True** when importing the data:

df = pd.read_json('my_data.json', lines=True)

The corrected code snippet for the full example is as follows:

#import JSON file into pandas DataFrame
df = pd.read_json('Documents/DataFiles/my_data.json', lines=True)

#view DataFrame
df

ID Rating Review
0 A 8 Great movie.nI would recommend it.
1 B 5 Mediocre movie.nWould not recommend it.
2 C 3 Bad movie.nI would not recommend.
3 D 7 Decent movie.nI might recommend it.

Upon execution of this revised code, we observe that the process completes without raising the ValueError. The input file has been successfully imported, and the data is organized into a structured pandas DataFrame. Crucially, notice that while the parsing succeeded, the newline characters (**n**) that were embedded within the "Review" text itself are now preserved within the cell values of the DataFrame. They were treated as literal parts of the string data, not as record delimiters, thanks to the **`lines=True`** setting handling the external structure.

Advanced Data Cleaning: Removing Escape Sequences

After successfully loading the data into the DataFrame by setting **`lines=True`**, the next common requirement is to clean the text data within columns that contain embedded escape sequences, such as the newline character (**n**). These characters, while syntactically correct within the JSON string, often interfere with subsequent text processing tasks like tokenization, display rendering, or natural language processing pipelines. Therefore, removing or replacing these control characters is a necessary secondary step in the data preparation workflow.

In pandas, string manipulation on DataFrame columns is efficiently handled through the `.str` accessor, which exposes vectorized string methods. To systematically remove all instances of the **n** characters from the "Review" column, we utilize the `.str.replace()` method. This method

allows us to find all occurrences of a specified substring or regular expression and substitute them with a replacement string, such as a single space or an empty string, depending on whether we want to preserve word separation.

To replace the **n** endlines with a single space, ensuring that words separated by the line break remain readable, the following syntax is employed in Python:

#replace n with empty space in 'Review' column
df = df.str.replace('n', ' ')

#view updated DataFrame
df

ID Rating Review
0 A 8 Great movie. I would recommend it.
1 B 5 Mediocre movie. Would not recommend it.
2 C 3 Bad movie. I would not recommend.
3 D 7 Decent movie. I might recommend it.

As confirmed by the output, the **n** values have been successfully removed from the "Review" column and replaced by single spaces, yielding clean, contiguous text strings suitable for further computational analysis. This two-step process--first structural parsing using **lines=True**, then internal data cleaning using `.str.replace()`--represents a robust solution for handling complex JSON Line data structures in a Python environment.

Alternative Solutions and Robust Data Ingestion

While utilizing `lines=True` in pandas is the most direct fix for the `ValueError: Trailing data` in JSONL contexts, developers should be aware of alternative methods, especially when dealing with extremely large files or non-standard data formats that might still trigger parsing issues. These alternatives offer greater control over the file reading process and are often preferred in high-performance or streaming applications.

One robust alternative involves reading the file line by line using standard Python file handling, parsing each line individually using the built-in `json` library, and then manually constructing the list of records before converting it into a DataFrame. This method, while requiring more boilerplate code, isolates the parsing of each record, making debugging easier if a specific line contains corruption or non-standard encoding. The typical workflow involves opening the file, iterating through `file.readlines()`, wrapping each line in a `json.loads(line)` call, and appending the resulting dictionary to a list. This approach entirely sidesteps the monolithic parsing assumption that causes the trailing data error.

Another powerful consideration is the use of external tools or libraries optimized for complex data formats. For instance, if the data is so large that it exceeds available memory, libraries designed for chunking or streaming data processing, such as Dask or specialized JSON streaming parsers, would be more appropriate. These tools inherently manage the data flow in smaller batches, preventing the parser from ever attempting to read the "trailing data" in the same memory buffer as the previous record. Furthermore, ensuring that the original data source is clean and adheres strictly to the JSON Lines specification--meaning no extra whitespace or non-JSON characters exist between the records--is always the best preventative measure against this and related parsing exceptions.

Summary and Best Practices for Data Loading

The ValueError: Trailing data serves as a critical indicator that your data ingestion function is misinterpreting the structure of your input file. Specifically within the pandas ecosystem, this error frequently arises because the default configuration of `pd.read_json()` expects a single, complete JSON entity, while the data source is formatted as JSON Lines (JSONL), which contains multiple, newline-separated JSON entities.

To ensure smooth and reliable data ingestion, particularly when working with file formats prone to this error, adhere to these best practices. First, always inspect the source data format. If records are separated by newlines, immediately recognize the need to utilize the **`lines=True`** parameter in your pandas function call. Second, if your data contains control characters like **`n`** within string fields, implement a mandatory post-processing step using the pandas `.str.replace()` method to cleanse the text and prepare it for downstream analysis.

By implementing the solution detailed here--setting **`lines=True`**--you effectively instruct pandas to treat the file as a stream of individual records, resolving the structural parsing conflict. This knowledge transforms a frustrating parsing error into a simple parameter adjustment, enabling efficient and accurate loading of high-volume, line-delimited data structures into your analytical environment.