Understanding and Resolving “ValueError: Trailing Data” When Reading JSON with Pandas in Python

Authored by

Mohammed looti

November 1, 2025

When engineering robust data ingestion pipelines within the Python ecosystem, developers frequently rely on powerful libraries like pandas DataFrame to manage and manipulate complex datasets. A crucial aspect of modern data processing involves handling data exchange formats, with JSON being one of the most prevalent standards. However, the process of importing JSON data from external sources often leads to runtime complications, particularly when the file format deviates subtly from expected norms. One of the most frequently encountered and often confusing exceptions during this process is the seemingly cryptic message:

ValueError: Trailing data

This ValueError signals a fundamental misunderstanding between the parsing function--typically `pd.read_json()`--and the structure of the input file. Essentially, the parser successfully interprets the initial valid data structure but then encounters unexpected, additional information before the end of the file is reached. This article serves as an authoritative resource, detailing the precise technical cause of this error and providing the most reliable, efficient, and straightforward solution for successful data ingestion.

Understanding the Root Cause: JSON vs. JSON Lines

To accurately diagnose and permanently resolve the `ValueError: Trailing data`, it is imperative to distinguish between two closely related but structurally distinct formats: standard JSON and JSON Lines (often abbreviated as JSONL or sometimes NDJSON). The core issue lies in the fact that the default parsing behavior of the pandas DataFrame library assumes the file adheres to the traditional, single-structure JSON standard, as defined by RFC 8259.

Under the standard JSON definition, an entire file must constitute a single, complete JSON entity. This entity is usually either a root object (e.g., `{"key": "value"}`) or, more commonly for datasets, a single array containing multiple objects (e.g.,). The parser expects to consume the entire file content as one continuous, valid structure. Any characters, including subsequent valid JSON structures, encountered after the closing brace or bracket of the root element but before the End-of-File marker are interpreted as forbidden "trailing data," causing the immediate failure.

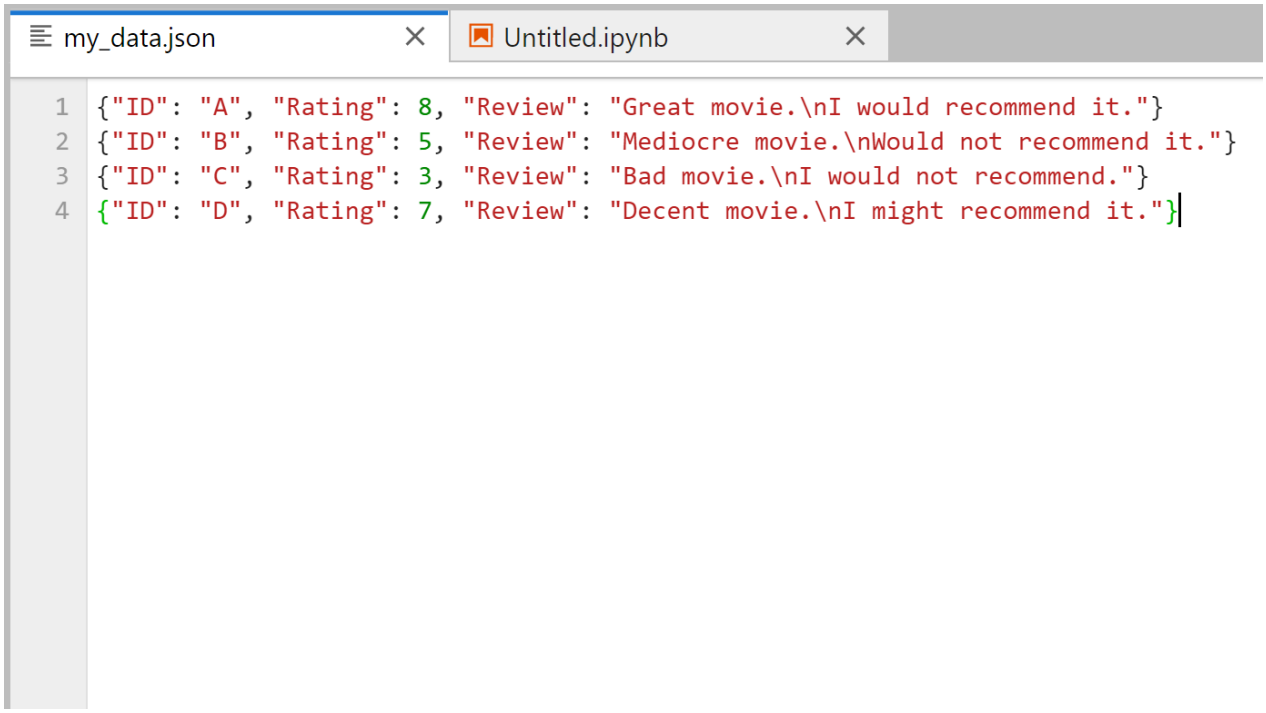
In contrast, the JSON Lines format is designed for streaming data and processing large datasets efficiently. A JSON Lines file consists of multiple independent, valid JSON objects, where each object is placed on its own line and separated from the next by a mandatory newline character (`\n`). This format is highly favored by API logs, streaming services, and data repositories because it allows for record-by-record processing without requiring the entire massive file to be loaded into memory simultaneously. When `pd.read_json` attempts to read such a file without specific configuration, it successfully parses the first line's JSON object, but the parser's standard mode interprets the subsequent newline character and the start of the second object as extraneous data,

thus raising the exception.

Practical Demonstration: Reproducing the Parsing Failure

To clearly illustrate how this error manifests, we can define a common data ingestion scenario involving a file that adheres strictly to the JSON Lines format. Imagine a data scientist attempting to import customer reviews stored in a file named `my_data.json`. Crucially, each review record is structured as an independent JSON object occupying a single line in the file.

Consider the structure of the input file, where the records are clearly delineated by physical line breaks:



It is important to note that, despite appearing as a collection of records, this file is not a single, valid standard JSON document (as it lacks the enclosing brackets of an array). Furthermore, observe the intentional inclusion of the literal `n` string within the "Review" field, intended to represent line breaks in the original review text, which is a separate issue we will address later.

If we proceed to load this file using the default configuration of the read_json function, the anticipated parsing error occurs immediately, halting the script execution:

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

ValueError: Trailing data

The parser successfully processes the first line (Record ID: A), confirming it is a complete JSON object. However, when it encounters the carriage return and the subsequent start of the second record (Record ID: B), it interprets this sequence as data that should not exist after the successful closure of the main structure. This conflict between the file's structure (JSON Lines) and the parser's default expectation (standard JSON) is the sole cause of the ValueError.

The Definitive Fix: Implementing the `lines=True` Parameter

The most elegant, direct, and efficient resolution to the `ValueError: Trailing data` is to simply adjust the parsing mechanism utilized by the `pd.read_json` function. This adjustment is achieved by setting the `lines` parameter to `True`. By implementing this simple flag, we explicitly inform pandas DataFrame that the input file conforms to the JSON Lines convention.

This modification fundamentally alters how the parser interacts with the file. Instead of seeking a single, monolithic JSON structure, the parser switches to a stream-based interpretation. It treats each physical line within the file as an independent, standalone JSON object that must be parsed individually. Each successfully parsed object is then seamlessly translated into a new row, or record, within the resulting pandas DataFrame. This strategy is not only essential for correctness but also dramatically improves performance when dealing with extremely large log files or data streams that naturally use the JSONL format.

Implementing the fix requires minimal alteration to the original import command:

#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.

As clearly demonstrated by the successful output, the data is now correctly loaded into the pandas DataFrame. The records that previously caused the "Trailing data" exception are now recognized and processed as distinct entities. Setting `lines=True` is the definitive, robust solution for importing any file formatted according to the JSON Lines standard using the read_json function.

Beyond Import: Handling Embedded Control Characters

Although setting `lines=True` successfully resolves the primary parsing error, a closer inspection of the resulting pandas DataFrame reveals a secondary data cleanliness issue, specifically within text fields like the "Review" column. The internal formatting characters--the literal newline characters (`n`) that were embedded within the original string data--have been preserved during the import process. While technically imported correctly, these control characters are often undesirable in analytical contexts, as they can interfere with text segmentation, visualization tools, or subsequent storage in relational databases.

To ensure maximum data usability and maintain a clean dataset, it is considered best practice to sanitize these embedded control characters immediately after the initial successful import. Given that we are working with string data within a pandas DataFrame, we can leverage pandas' powerful vectorized string manipulation methods, which are highly optimized for speed and efficiency.

The preferred method for this type of data cleaning is the application of the `.str.replace()` method directly on the target column. This approach avoids the need for inefficient row-by-row iteration using standard Python loops, performing the replacement operation simultaneously across all rows in the column:

#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.

By replacing the literal string `n` with a standard space (`' '`), we effectively remove the disruptive control characters while preserving the natural spacing and readability of the review text. This two-step process--initial import using `lines=True`, followed by vectorized string replacement--results in a structured, clean, and ready-to-use dataset for subsequent analytical tasks.

Advanced Scenarios and Robust Error Handling

While the `lines=True` parameter resolves the vast majority of "Trailing data" exceptions, there are

edge cases involving highly non-standard or corrupted input files where even the JSON Lines parser may fail. These scenarios typically involve files that contain extraneous metadata lines, non-JSON comments, or records that are malformed or truncated, interspersed with valid JSON objects. When automated parsing rules are insufficient, a more manual and granular approach is warranted to ensure no valuable data is lost.

In such complex situations, developers must revert to utilizing Python's built-in `json` module combined with manual file iteration. This strategy sacrifices the speed and optimization of `pd.read_json` but grants the developer explicit control over error handling for every single line of the input file. This allows for customized logging of corrupted lines or the skipping of non-essential metadata.

The robust workflow for this manual parsing strategy is structured as follows:

Initiate the process by opening the source file and iterating through its contents line by line, ensuring proper resource management.

Implement a `try...except` block around the parsing operation, utilizing `json.loads()` to attempt conversion of the current line into a standard Python dictionary object.

If parsing is successful (the `try` block executes), append the resulting Python dictionary to an accumulating list of records.

If parsing fails (the `except` block executes), log the line number and the specific error encountered, allowing the script to skip the corrupted line and continue processing the rest of the file.

Once all lines have been processed, convert the final list of successfully parsed dictionaries into the target pandas DataFrame using the constructor: `pd.DataFrame(data_list)`.

This manual technique, while slower, provides necessary resilience for dealing with inconsistent data streams that resist standardized automatic parsing methods. It ensures that the maximum amount of usable data is salvaged from an otherwise messy source.

Summary and Best Practices

The ValueError: Trailing data serves as a critical diagnostic indicator: the data source is formatted as JSON Lines, characterized by records separated by newline characters, rather than the single-structure standard JSON expected by default. Resolving this issue efficiently is key to maintaining high throughput in data pipelines.

Key takeaways for efficient JSON data handling in Python:

Understand the Error Context: Recognize that "Trailing data" is not typically a data corruption issue but a format mismatch between JSON Lines (JSONL) and the default parser expectation.
Utilize the Optimal Fix: For reliable and scalable ingestion of JSON Lines files, always use the `lines=True` parameter in the read_json function.
Prioritize Data Sanitation: After successful import, always examine text fields for embedded control characters (like `n`) and use vectorized methods (`.str.replace()`) for rapid, post-import cleaning.
Reserve Manual Parsing: Only resort to slower, line-by-line parsing using Python's `json` module when dealing with exceptionally irregular or deliberately corrupted files that standard pandas functions cannot handle.

By adhering to these best practices, developers can confidently manage and process diverse JSON data streams in Python, leading to smoother and more robust analytical workflows.

Additional Resources

To further enhance your skills in data manipulation and processing using the pandas library, consider exploring the following advanced tutorials:

How to efficiently merge large DataFrames in Python.
Advanced techniques for time series data handling in pandas.
Optimizing memory usage when working with massive datasets in Python.