How to convert a Pandas DataFrame to JSON?

Authored by

stats writer

December 24, 2025

The ability to serialize data efficiently is crucial when working across different programming environments or preparing data for web consumption. In the world of data analysis using Pandas, converting a **DataFrame** into a format easily readable by other systems, such as **JavaScript** or **APIs**, is a common requirement. The ideal format for this purpose is often JSON (JavaScript Object Notation).

To perform this transformation, Pandas DataFrames provide a powerful built-in utility: the **to_json() method**. This function seamlessly converts the structured tabular data held within the **DataFrame** into a valid JSON string or writes it directly to a file path. The resulting output is a lightweight, standardized representation of the data, which is highly advantageous for transmission or storage.

When using the **to_json() method**, developers gain flexibility through the `orient` parameter. This parameter dictates the precise structure that the resulting JSON output will adopt, allowing for tailored data serialization based on specific consumer requirements, such as creating an array of objects for easy parsing via `JSON.parse()` in client-side scripts.

Understanding the available output formats is key to successful serialization. The **to_json() method** supports several orientation options, each fundamentally altering how the **DataFrame's** structure (columns, index, and data) is mapped into the hierarchical structure of JSON.

Below are the six primary orientation formats supported by `df.to_json()`:

'split' : Generates a dictionary structure containing explicit keys for `index`, `columns`, and `data` (values).
'records' : Produces a list where each element is a dictionary representing a single row. This is often the most intuitive format for consumers expecting an array of objects.
'index' : Creates a dictionary where the outer keys are the **DataFrame's index** labels, and the values are row dictionaries (column name -> value).
'columns' : Creates a dictionary organized by column names, where the values are dictionaries mapping index labels to data points within that column.
'values' : Outputs a simple list of lists, representing only the data values without any structural keys (columns or index).
'table' : Produces a verbose dictionary structure that includes both the data and an embedded **schema definition**, adhering to the JSON Table Schema specification.

To illustrate the differences between these serialization approaches, we will utilize a simple example DataFrame created in **Python**:

import pandas as pd

Create a sample DataFrame representing player statistics

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

Display the DataFrame structure
df

points assists
0 25 5
1 12 7
2 15 7
3 19 12

Method 1: Using the 'Split' Orientation

The `orient='split'` setting is highly explicit, separating the metadata (columns and index) from the actual data values. This structure is often preferred when the consumer needs easy access to the column headers and index labels independently of the data payload.

The output is a single dictionary containing three distinct keys: `"columns"` (listing column names), `"index"` (listing row labels), and `"data"` (containing the data as a list of lists, where each inner list is a row).

df.to_json(orient='split')

{
"columns": ,
"index": ,
"data": ,
,
,

]
}

This format is robust for reconstruction, as all necessary structural information is present in separate, clearly defined arrays, minimizing ambiguity about how the data array should be interpreted.

Method 2: Using the 'Records' Orientation

The `orient='records'` approach is arguably the most common and user-friendly format for web

services and APIs. It transforms the **DataFrame** into a list of dictionaries, where each dictionary corresponds exactly to one row in the original data structure.

In this format, column names become the keys within the inner dictionaries, and the associated data points are the values. This makes the data immediately accessible and iterable in many modern programming languages, often requiring minimal parsing effort.

df.to_json(orient='records')

Notice that the output is a list (denoted by square brackets) containing individual objects (rows), making it ideal for immediate consumption as an array of objects.

Method 3: Using the 'Index' Orientation

When the row index labels are critically important identifiers for the data, the `orient='index'` option provides a structure that prioritizes them. The resulting output is a dictionary where the keys are the row index values (in our example, 0, 1, 2, 3).

Each index key maps to an inner dictionary that holds the column-value pairs for that specific row. This format is efficient if you frequently need to look up data based on the original **DataFrame's index** label, such as retrieving a single record quickly.

df.to_json(orient='index')

{
"0": {
"points": 25,
"assists": 5
},
"1": {
"points": 12,
"assists": 7
},
"2": {
"points": 15,
"assists": 7
},
"3": {
"points": 19,
"assists": 12

}
}

It is important to remember that if your **DataFrame** uses non-unique index values, this format will still be generated, but direct reverse mapping might lead to issues if uniqueness is assumed by the consumer.

Method 4: Using the 'Columns' Orientation

In contrast to the row-centric formats, the `orient='columns'` setting structures the JSON output based on the columns. The outer keys of the dictionary are the column names (`'points'` and `'assists'` in this case).

Under each column key, you will find a secondary dictionary that maps the row index labels to the corresponding data values within that column. This orientation is particularly useful when analyzing or transmitting time-series data or when processing data column-by-column, rather than row-by-row.

df.to_json(orient='columns')

{
"points": {
"0": 25,
"1": 12,
"2": 15,
"3": 19
},
"assists": {
"0": 5,
"1": 7,
"2": 7,
"3": 12
}
}

The structure inherently groups all values belonging to a specific feature (column) together, which can simplify aggregation or statistical processing on the consuming end if organized this way.

Method 5: Using the 'Values' Orientation

The `orient='values'` option provides the most compact representation, stripping away all metadata, including column names and index labels. The output is purely a list of lists, mirroring the underlying **NumPy array** structure of the DataFrame.

While this is the smallest output size, it requires the consuming application to already know the correct order and meaning of the columns. It is best suited for scenarios where bandwidth optimization is paramount and the schema is fixed and known beforehand.

df.to_json(orient='values')

,
,
,

]

If you require column or index information, you must use one of the other five orientations, as the `'values'` output is strictly focused on the data matrix itself.

Method 6: Using the 'Table' Orientation with Schema

The `orient='table'` setting is unique because it outputs a complete representation that includes not just the data, but also a detailed schema. This format adheres to the JSON Table Schema standard, which is excellent for data validation and ensuring interoperability across diverse systems.

The resulting dictionary contains two main components: `"schema"`, which defines the fields (columns, types, and primary key), and `"data"`, which contains the records themselves, often including the index as a separate field.

df.to_json(orient='table')

{
"schema": {
"fields": ,
"primaryKey": ,
"pandas_version": "0.20.0"
},
"data":

}

This format is verbose but ensures maximum data fidelity and self-description, making it ideal for long-term storage or transfer to systems that strictly enforce schema validation.

Exporting the JSON String to a Physical File

While the **to_json() method** defaults to returning a JSON string, you often need to save this output directly to a file on your local system or a server path. The simplest way to achieve this involves generating the JSON string first and then using standard Python file operations to write the content.

Alternatively, the `to_json()` method itself accepts a path argument (not demonstrated in the original example, but good practice to mention), allowing direct file writing. However, if you need to manipulate the JSON string before writing (e.g., adding headers or metadata), generating the string first is required.

Here is the standard syntax for creating and exporting a JSON file using the highly versatile `'records'` orientation as an example:

 # Convert the DataFrame to a JSON string using the 'records' orientation
json_file = df.to_json(orient='records')

Export the JSON string to a file named 'my_data.json'
with open('my_data.json', 'w') as f:
f.write(json_file)

This process ensures that the data is correctly encoded and saved in a standard `.json` file, ready for external consumption by web applications, databases, or other analytical tools.

Summary and Further Documentation

The flexibility offered by the `orient` parameter in the **Pandas DataFrame** `to_json()` method allows developers to serialize data optimally for diverse use cases, ranging from compact storage (`'values'`) to fully self-describing datasets (`'table'`). Choosing the correct orientation streamlines data consumption and minimizes downstream processing errors.

For more detailed technical specifications, parameters, and advanced functionalities of this robust serialization tool, consult the official documentation.

You can find the complete documentation for the pandas to_json() function here.