Learn How to Convert a Pandas DataFrame to a Python Dictionary

Authored by

Mohammed looti

November 3, 2025

The process of converting a specialized
Pandas DataFrame
into a native
Python dictionary
is a fundamental requirement in modern data workflows. This conversion is crucial when transitioning data from the powerful, analytical environment of Pandas to standard Python applications, particularly for tasks involving
serialization
, passing data through APIs, or integrating with backend services. Pandas addresses this need efficiently and flexibly through the robust
`to_dict()`
function.

The Core Mechanism: Mastering the `df.to_dict()` Function

The primary function used for converting any
Pandas DataFrame
object into a
dictionary
is remarkably straightforward. By default, calling this method without any arguments structures the output in a column-oriented fashion, meaning the column headers of the DataFrame become the primary keys of the resulting dictionary. This nested structure is often the easiest way to preserve the relational integrity of the data.

The fundamental syntax involves simply invoking the method directly on the DataFrame object, as shown below:

df.to_dict()

While this default usage provides a quick conversion, the true utility and power of this function are unlocked by the optional `orient` argument. This parameter is the key control mechanism, dictating the desired structure and nesting level of the output. Selecting the appropriate orientation is critical for ensuring the resulting
dictionary
perfectly aligns with your downstream application's input requirements.

Exploring the `to_dict()` Orientation Parameters

The `to_dict()` method allows developers granular control over how the two-dimensional tabular data is mapped into a nested or flattened Python

data structure
. Understanding these six distinct orientations--or modes--is essential for efficient data preparation and manipulation, as each mode serves a unique purpose in data interchange.

The function accepts the following values for the `orient` parameter, each fundamentally altering how DataFrame indices and column names are translated into dictionary keys and values:

'dict' (Default): This mode produces the standard column-oriented, nested output. The keys of the top-level dictionary are the column names. The values associated with these keys are themselves dictionaries, where the DataFrame index labels serve as inner keys, and the cell data are the final values. This structure is excellent for maintaining the precise relational context of the original data.

'list': This orientation provides a clean, column-wise list output. The dictionary keys remain the column names, but the corresponding values are simple Python lists containing all the data from that column in sequential order. This format is widely preferred for rapid data extraction or when passing data to functions that specifically expect lists of values rather than indexed structures.

'series': Similar to the 'list' orientation, keys are still the column names, but the resulting values are specialized Pandas
Series
objects. This is particularly useful if you need to retain the internal data type information and the explicit index associated with each column after the conversion process.

'split': This orientation is designed for high-fidelity data interchange, serializing the DataFrame into three separate, explicit components: `'columns'` (a list of column names), `'data'` (a list of lists representing the rows), and `'index'` (a list of index labels). This highly structured output is often the preferred format for
JSON serialization
and API communication where structural clarity is paramount.

'records': This is the most common row-oriented approach. The output is a list containing multiple dictionaries, where each individual dictionary represents a single row (or record). The keys of these inner dictionaries are the column names, and the values are the cell data for that specific row. This structure closely mimics how data is handled by database cursors and is ideal for iteration and row-by-row processing.

'index': This orientation is also focused on rows but uses the DataFrame's index labels as the primary keys of the main dictionary. The values are dictionaries representing the row content, where column names map to cell values. This mode is excellent for quick lookups based on the original row index.

Setting Up the Practical Demonstration

To effectively illustrate the distinct structural differences produced by each orientation parameter, we will establish and utilize a simple, small-scale Pandas DataFrame. This controlled baseline dataset is intentionally straightforward, allowing us to clearly track how row indices, column names, and associated values are meticulously mapped during the conversion process using `to_dict()`.

We begin by initializing and viewing our sample DataFrame, which contains data simulating a sports team's performance metrics:

import pandas as pd

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

#view DataFrame
df

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

It is important to note that this DataFrame contains five rows, three columns, and utilizes the default zero-based integer index. We will now proceed to apply the `to_dict()` method sequentially, using each of the major orientation modes to demonstrate their specific outputs.

Example 1: Default Column Orientation (`orient='dict'`)

When the `orient` parameter is omitted, the default `'dict'` setting is used. This generates a powerful, nested dictionary structure. The column names serve as the outer keys, and the corresponding values are dictionaries where the index labels (0 through 4) act as the inner keys, mapping directly to the cell values. This preserves the full context of the original table.

df.to_dict()

{'team': {0: 'A', 1: 'A', 2: 'B', 3: 'B', 4: 'C'},
'points': {0: 5, 1: 7, 2: 9, 3: 12, 4: 9},
'rebounds': {0: 11, 1: 8, 2: 6, 3: 6, 4: 5}}

Example 2: Simple Columnar Lists (`orient='list'`)

The `'list'` orientation is highly practical when the index information is irrelevant and only the raw columnar data is needed. This mode maps column names to keys, and the associated values are straightforward Python lists containing all elements from that column in their original order. This is typically the most concise dictionary representation for simple data transfer.

df.to_dict('list')

{'team': ,
'points': ,
'rebounds': }

Example 3: Preserving Pandas Series (`orient='series'`)

By specifying `'series'`, the conversion ensures that each column is transformed into a Series
object, which is then mapped to the column name key. This method is crucial when the integrity of the column's data type, name, and internal Pandas indexing features must be maintained within the resulting dictionary values.

df.to_dict('series')

{'team': 0 A
1 A
2 B
3 B
4 C
Name: team, dtype: object,
'points': 0 5
1 7
2 9
3 12
4 9
Name: points, dtype: int64,
'rebounds': 0 11

1 8
2 6
3 6
4 5
Name: rebounds, dtype: int64}

Example 4: Structured Data Exchange (`orient='split'`)

The `'split'` orientation generates a highly standardized dictionary that explicitly separates the DataFrame's components into three lists keyed by `'index'`, `'columns'`, and `'data'`. The `'data'` element is a list of lists, where each inner list represents a row. This format is structurally robust, easy to parse, and widely favored for reliable data interchange, especially when dealing with external systems expecting defined metadata.

df.to_dict('split')

{'index': ,
'columns': ,
'data': , , , ,]}

Example 5: List of Row Records (`orient='records'`)

The `'records'` orientation transforms the DataFrame into a list of dictionaries, effectively making the output row-centric. Each dictionary within the list corresponds precisely to one row of the DataFrame. The keys of these inner dictionaries are the column names, mapping directly to the row values. This structure is highly intuitive for processing data sequentially and mirrors common formats retrieved from database query results.

df.to_dict('records')

Example 6: Index-Keyed Row Dictionaries (`orient='index'`)

Finally, the `'index'` orientation generates a dictionary where the keys are derived directly from the DataFrame's index labels (0, 1, 2, etc.). The values mapped to these index keys are dictionaries representing the entire row content, with column names as keys and cell data as values. This is similar to `'records'`, but instead of being placed within a list, the row dictionaries are immediately accessible via their index label.

df.to_dict('index')

{0: {'team': 'A', 'points': 5, 'rebounds': 11},
1: {'team': 'A', 'points': 7, 'rebounds': 8},
2: {'team': 'B', 'points': 9, 'rebounds': 6},
3: {'team': 'B', 'points': 12, 'rebounds': 6},
4: {'team': 'C', 'points': 9, 'rebounds': 5}}

Conclusion and Related Data Conversion Methods

Mastering the efficient conversion between native Python
data structures
and specialized Pandas objects is foundational for robust data science and engineering workflows. The `to_dict()` method, with its versatile `orient` parameter, ensures that developers can rapidly transform tabular data into the exact dictionary structure required for any application, whether it demands columnar lists, index-keyed records, or explicit split metadata.

While `to_dict()` handles dictionary conversion comprehensively, the Pandas library offers a complete suite of functions for various data transformations. For those interested in exploring other common data conversions involving
Pandas DataFrames, consider the following resources:

How to Convert Pandas DataFrame to List