Learn How to Display All Columns in a Pandas DataFrame

Authored by

Mohammed looti

November 3, 2025

The Challenge of Wide Data: Pandas Display Defaults

When engaging in serious **data analysis** or machine learning workflows, the **Pandas DataFrame** stands as the foundational data structure. These workflows are typically executed within interactive environments such as **Jupyter notebooks**, which offer a powerful platform for iterative coding and visualization. However, a common obstacle encountered by developers involves the default display limitations imposed by Pandas itself. This restriction often prevents the comprehensive viewing of wide datasets, where the number of features (columns) exceeds standard screen dimensions.

By default, most Pandas configurations, particularly within notebook environments, enforce a strict limit, typically restricting the output to a maximum of only 20 columns. This limitation is not arbitrary; it is a critical design choice intended to optimize performance and usability. Without this cap, rendering a DataFrame containing hundreds or thousands of features would result in massive, unwieldy output that could severely clutter the interface, drastically slow down the notebook's responsiveness, or even lead to memory issues. While beneficial for performance, this restriction inherently hides the full structure of the data, making initial exploratory tasks--such as verifying feature naming or checking data types across all columns--exceptionally challenging.

To facilitate thorough data inspection and ensure that analysts can visualize the complete breadth of their datasets, the Pandas library provides sophisticated tools for display configuration management. These tools allow users to globally override the default restrictions within the current session. By modifying these settings, we can instruct the environment to render every column present in the **Pandas DataFrame**, offering the complete visualization necessary for successful data validation and in-depth analysis.

Overriding Default Limits: The Power of pd.set_option

The primary mechanism for adjusting session-specific display behavior in Pandas is the highly versatile function, **pd.set_option**. This function serves as the gateway to Pandas' internal configuration system, enabling users to customize global parameters ranging from floating-point precision to warning suppression and, crucially, output display limits. To permanently adjust the column display limit for the duration of the current session or script execution, we must call this function and specify the desired changes.

The specific global parameter responsible for controlling the maximum number of visible columns is designated as **`max_columns`**. To achieve the goal of displaying all available columns, irrespective of their total count, we must assign a special value to this option: `None`. Setting **`max_columns`** to `None` serves as a universal signal to the Pandas rendering engine, indicating that any arbitrary display limit previously enforced should be completely removed, thereby ensuring that the full DataFrame structure is presented upon request.

Executing this configuration change is a straightforward process, requiring only a single line of code. It is essential that this command be run early in your notebook or script, as the change only takes effect for subsequent DataFrame displays within that runtime environment:

pd.set_option('max_columns', None)

Once this modification is successfully applied, any subsequent attempt to print or view a **DataFrame** object will respect the new global setting, ensuring that every column is rendered. This eliminates the frustrating truncation indicated by ellipses and allows analysts to inspect the complete breadth of their dataset without structural information being hidden.

A Practical Walkthrough: Demonstrating Column Truncation

To fully appreciate the necessity and impact of configuring display options, it is instructive to observe the default behavior in action. We can simulate a common scenario by creating a synthetic, wide dataset that intentionally exceeds the default 20-column limit. For this practical demonstration, we leverage the **NumPy** library to efficiently initialize a large DataFrame structure.

Imagine initializing a DataFrame composed of five rows but containing 30 distinct, unlabeled columns. When this DataFrame is created and then immediately called for display within a standard Jupyter cell, the output will clearly demonstrate the default truncation, with columns 20 through 29 being automatically hidden to save screen space and processing time:

```
import pandas as pd
import numpy as np

#create dataFrame with 5 rows and 30 columns
df = pd.DataFrame(index=np.arange(5), columns=np.arange(30))

#view dataFrame
df
```

The resultant output visually confirms the limitations of the default settings, with the hidden columns represented by an ellipsis, clearly obscuring vital structural information:

	0	1	2	3	4	5	6	7	8	9	...	20	21	22	23	24	25	26	27	28	29
0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
1	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
3	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
4	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	...	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

5 rows × 30 columns

To resolve this truncation and ensure all 30 columns are visible for comprehensive visual inspection, we must apply the configuration change by calling **pd.set_option**, followed by a re-display of the DataFrame object:

#specify that all columns should be shown
pd.set_option('max_columns', None)

#view DataFrame
df

Upon re-execution, the environment successfully recognizes the newly set global option, displaying the DataFrame in its full, untruncated form:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
0	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
1	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
3	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
4	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

Extracting Metadata: Viewing Column Names Only

While displaying the entire DataFrame is essential for visual verification, it is often impractical or visually overwhelming when dealing with datasets that possess hundreds of columns. In many cases, the analyst only requires a quick, clean reference list of the column names themselves, perhaps for scripting, debugging, or verifying the consistency of naming conventions. Rendering the entire table structure in such scenarios is an inefficient use of screen real estate and processing time.

A much quicker and cleaner alternative is to bypass the full table rendering entirely and instead

extract and display all column names as a standard **Python list**. This approach is highly effective when the primary concern is the structure's metadata rather than the raw data content. We achieve this by accessing the DataFrame's inherent `.columns` attribute, which returns an Index object containing all column labels. We then immediately convert this object into a standard list using the `.tolist()` method.

This streamlined technique provides a fast, text-based output directly to the console, which is invaluable when dealing with DataFrames too wide for comfortable screen viewing, even after adjusting the **`max_columns`** setting. The following syntax executes this extraction and prints the complete list of column labels:

print(df.columns.tolist())

For the demonstration DataFrame we created earlier, this method provides an exhaustive, unambiguous list confirming the presence and naming of all 30 elements:

print(df.columns.tolist())

Extending Control: Managing Row Display Limits

The display restrictions imposed by Pandas are not limited solely to the horizontal axis; they also apply vertically to the number of rows displayed. By default, Pandas enforces a restriction, typically capping the output at 60 rows (which usually includes 5 header rows and 5 footer rows separated by an ellipsis). This safeguard is crucial when dealing with extremely long time series or observational datasets, preventing memory overflow and performance degradation that would occur if every single record were rendered to the screen simultaneously. If, however, the task requires viewing every single row within a **Pandas DataFrame**, we can apply an identical configuration logic to override this vertical limit.

To instruct Pandas to display every single record, regardless of the dataset's length, we must once again employ **`pd.set_option`**. This time, we target the related parameter named **`max_rows`** and assign it the value of `None`. Just like **`max_columns`**, the `None` assignment signals the system to bypass any fixed numerical display limit:

pd.set_option('max_rows', None)

Alternatively, viewing hundreds of thousands of rows is rarely practical. For routine exploratory analysis, analysts often require a specific, higher limit than the default 60, but still far less than the full dataset size. In this case, instead of using `None`, we can assign a specific integer value to the

`max_rows` option. For instance, to ensure that a maximum of 100 rows are displayed (allowing a comfortable view of the top and bottom of the data), we would execute the following command:

pd.set_option('max_rows', 100)

Setting specific numerical limits for both rows and columns is a highly recommended practice, balancing the need for visual inspection with the necessity of maintaining a responsive and manageable notebook environment.

Maintaining Clean Code: Restoring Default Settings

While overriding the default display limits is frequently necessary for deep inspection and debugging, maintaining these global settings across an entire notebook or script can introduce problems. If a notebook is shared, or if subsequent analysis involves different DataFrames (especially very large ones where truncation is highly desirable), the universally applied configuration can lead to overly verbose outputs or unexpected performance slowdowns. Therefore, adherence to sound **best practice**s dictates that global Pandas options should be reset once the specific inspection task is completed.

To revert the display settings back to their original factory defaults, Pandas provides the dedicated function, `pd.reset_option`. This function is straightforward to use: simply specify the option you wish to reset (e.g., `max_columns` or `max_rows`), and Pandas will automatically restore the initial configuration (typically 20 columns and 60 rows, respectively).

For example, to return the column display limit to the default 20, you would execute the following command:

pd.reset_option('max_columns')

Systematically using `pd.reset_option` ensures that your coding environment remains predictable and clean for subsequent analyses or for colleagues reviewing your work. Failing to reset these global options is a common source of unexpected behavior, resulting in performance issues or unnecessarily verbose output when working with other DataFrames later in the session.

Best Practices: Utilizing Context Managers for Temporary Changes

In highly rigorous data science environments or when preparing production-ready code, relying on manual global state changes (like sequential calls to `pd.set_option` followed by `pd.reset_option`) introduces potential failure points. If an error occurs between the set and the reset commands, the global configuration might be left in an unintended state, leading to difficult-

to-trace side effects. A more robust and sophisticated approach involves using a specialized feature known as a **context manager**.

The `pd.option_context` function provides a mechanism to temporarily set display options for a specific, isolated block of code. Crucially, the context manager guarantees that the original settings are automatically restored immediately after the block of code finishes executing, regardless of whether the execution completed successfully or raised an error.

Although using a context manager is slightly more verbose than a simple global set, the guarantee of automatic reversion makes it the preferred method when writing reusable functions or modules where the isolation of display parameters is paramount. This prevents unintentional global changes from "leaking" out of a defined visualization scope, thereby significantly improving code reliability and maintainability.

The following tutorials explain how to perform other common operations on pandas DataFrames:

Tutorial on modifying float precision display settings.

Guide to adjusting the width of the display output.

Documentation regarding the use of display context managers.