Learning PySpark: How to Display Full Column Content in DataFrames

Authored by

Mohammed looti

November 11, 2025

RECOMMENDED CITATION

Mohammed looti (2025). *Learning PySpark: How to Display Full Column Content in DataFrames*. PSYCHOLOGICAL STATISTICS. Retrieved from https://statistics.arabpsychology.com/?p=16710

The Challenge of Default Data Truncation in PySpark

When undertaking data engineering or analysis tasks using large-scale distributed frameworks, the ability to accurately inspect data is paramount. In the PySpark environment, data validation and debugging frequently rely on the standard show() function, which provides a tabular representation of the dataset. However, by default, this powerful utility applies an automatic truncation mechanism to column content. This feature, designed to keep terminal output neat and manageable, often leads to the frustrating obscuring of long strings, rendering initial data quality checks challenging and sometimes misleading.

This default behavior can mask critical differences within complex string fields, such as long URLs, unique identifiers, or extensive textual data. If a cell contains more than the system-imposed character limit (typically 20 characters), the string is clipped, and an ellipsis (`...`) is appended. While this is helpful for rapid column identification in wide datasets, it actively prevents the user from seeing the full content necessary for deep inspection, anomaly detection, or accurate data fidelity assurance. Data practitioners must understand how to override this limitation to ensure they are always viewing a complete and unmodified snapshot of their records.

Fortunately, PySpark DataFrames offer a simple, built-in solution to bypass this truncation. By supplying a specific argument to the show() action, you can force the display to dynamically expand column widths, accommodating the longest string present in the visible partition. Mastering this adjustment is essential for efficient data exploration and debugging within the Spark ecosystem.

Why PySpark DataFrames Truncate Data by Default

To understand how to disable truncation, we must first grasp the underlying mechanism. The fundamental structure for handling distributed data within Spark is the DataFrame. When a developer executes the `df.show()` command, the system retrieves a sample set of rows (defaulting to 20) and attempts to render them in a clean, terminal-friendly table format. By design, the default display width is capped at **20 characters per column**. Any string exceeding this limit is aggressively clipped to maintain uniformity.

This default setting is primarily a performance-oriented and compatibility feature. In distributed environments, especially those accessed via constrained terminal interfaces or notebooks with limited horizontal screen real estate, allowing columns to expand indefinitely could lead to unreadable output. Imagine a column containing JSON structures that span hundreds of characters; displaying this fully could break the table formatting or necessitate excessive horizontal scrolling, hindering quick visual checks. The 20-character limit provides a necessary balance between showing descriptive data and maintaining a readable table layout.

However, for practical debugging, this feature often becomes counterproductive. If you are comparing two identifiers that share the first 20 characters, the truncated output will incorrectly suggest they are identical. Therefore, when the actual content is more important than the presentation aesthetics, developers must explicitly override the system's default behavior. This is achieved through the `truncate` parameter, which controls the maximum string width displayed. By manipulating this parameter, we instruct the PySpark engine to dynamically adjust the column rendering.

Method 1: The Recommended Approach Using `truncate=False`

The most intuitive and highly recommended method for completely disabling column truncation involves setting the `truncate` parameter to the boolean value **False**. This simple flag explicitly communicates the intent to the PySpark execution engine: all string content should be displayed fully, regardless of its length.

This approach is preferred due to its inherent clarity and readability. Aligning with standard Python programming practices, using a boolean flag like `truncate=False` immediately conveys the functionality being disabled. When other developers review the code, the intention of displaying the full, unclipped data is unmistakable, which contributes significantly to maintainability and reduces ambiguity during collaborative work.

The syntax for implementing this method is straightforward, requiring only the addition of the parameter within the show() function call. It is important to remember that while this disables horizontal truncation, the function still adheres to the default row limit (which is 20 rows, unless the optional `n` parameter is used to specify a different number).

The required syntax looks like this:

df.show(truncate=False)

Upon execution, the column headers and content borders will dynamically resize to accommodate the longest string value present in the data being displayed, ensuring that every character of the string data is visible to the user.

Method 2: Leveraging the Integer Syntax `truncate=0`

An equally effective, though perhaps less semantically clear, alternative to the boolean method is setting the `truncate` parameter to the integer value **0**. Functionally, this method achieves an identical outcome: the truncation mechanism is disabled, and all column contents are displayed fully.

The show() function's `truncate` parameter is designed to accept an integer specifying the maximum display width. While any positive integer (e.g., 50 or 100) would extend the width limit, setting the limit to **0** is conventionally interpreted within the API as a special instruction meaning "no limit" or "disable truncation entirely."

While some developers prefer the explicit nature of `truncate=False`, others may favor the integer approach, particularly if they are accustomed to similar numerical limits in other programming environments. It is crucial to recognize that the output generated by `truncate=0` is exactly the same as that produced by `truncate=False`.

The syntax for using the integer argument is:

df.show(truncate=0)

Developers should select the method that best adheres to their organization's coding conventions, but they can be confident that both **False** and **0** effectively override the default 20-character restriction, providing full visual access to the dataset's string fields.

Practical Walkthrough: Setting Up the Demonstration

To demonstrate the critical difference these parameters make, we will walk through a common scenario: creating a sample DataFrame containing strings that intentionally exceed the default display limit. This setup requires initializing a Spark Session and defining structured data.

Our sample data represents store information, including a `store` identifier, a `sales` figure, and a critical `employees` column which contains a comma-separated list of names. Several rows in the `employees` column contain lists that significantly exceed 20 characters, ensuring that the default display will trigger truncation, thereby hiding essential details.

When the base command `df.show()` is executed without specifying the `truncate` argument, the resulting output clearly illustrates the problem. Notice how the data for Store 'A' and Store 'C' in the `employees` column is visibly clipped and replaced by ellipses, preventing a complete assessment of the staff list.

from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()

#define data
data = ,
,
,

```
,
]

#define column names
columns =

#create dataframe using data and column names
df = spark.createDataFrame(data, columns)

#view dataframe
df.show()

+-----+--------------------+-----+
|store| employees|sales|
+-----+--------------------+-----+
| A|Andy Bob Chad Dou...| 136|
| B| Frank Henry| 223|
| C|Ian John Ken Liam...| 450|
| D| Oscar Prim| 290|
| E| Quentin Ross Sarah| 189|
+-----+--------------------+-----+
```

This visual confirmation of missing data underscores the necessity of overriding the default truncation. The following sections apply the previously defined methods to restore the complete string content, confirming data integrity.

Observing the Results: Full Content Display

When we apply the `truncate=False` argument, the transformation is immediate and effective. The PySpark display logic recalculates the required width for the `employees` column based on the longest string value it contains among the displayed rows. This dynamic adjustment ensures that all characters are visible, eliminating the ellipsis characters that previously obscured the data.

This technique is vital during the initial stages of data pipeline construction, where visually confirming that raw input strings are correctly loaded, parsed, and structured is a primary concern. It provides the highest level of data fidelity assurance during console inspection.

Observe the output below. The `employees` column has expanded significantly, revealing the complete list of employee names for all stores, including the previously truncated rows (A and C):

#view dataframe with full column content using truncate=False

df.show(truncate=False)

```
+-----+-------------------------+-----+
|store|employees |sales|
+-----+-------------------------+-----+
|A |Andy Bob Chad Doug Eric |136 |
|B |Frank Henry |223 |
|C |Ian John Ken Liam Mike Noah|450 |
|D |Oscar Prim |290 |
|E |Quentin Ross Sarah |189 |
+-----+-------------------------+-----+
```

Furthermore, running the demonstration using the integer-based approach, `truncate=0`, confirms the functional equivalence of the two methods. The result is perfectly aligned, demonstrating that both syntaxes are reliable directives for disabling the default character limit and achieving a full-content display.

#view dataframe with full column content using truncate=0
df.show(truncate=0)

```
+-----+-------------------------+-----+
|store|employees |sales|
+-----+-------------------------+-----+
|A |Andy Bob Chad Doug Eric |136 |
|B |Frank Henry |223 |
|C |Ian John Ken Liam Mike Noah|450 |
|D |Oscar Prim |290 |
|E |Quentin Ross Sarah |189 |
+-----+-------------------------+-----+
```

The successful, full display of the **employees** column content using both commands concludes the essential demonstration, resolving the common hurdle of clipped string data in PySpark DataFrames.

Best Practices and Advanced Display Techniques

While disabling truncation using `truncate=False` or `truncate=0` is a powerful technique for data inspection, it is important to treat the `show()` function primarily as a diagnostic tool for sampling small subsets of data. When working with truly massive textual data--strings that span thousands of characters or tables with a very large number of columns--displaying the full, expanded output

can still create usability challenges, potentially overwhelming the terminal interface and making visual parsing difficult.

For scenarios involving extremely wide columns or numerous columns, consider adopting alternative display strategies. One highly effective technique is utilizing the optional `vertical=True` parameter within the show() function. This option transforms the output format from a horizontal table into a vertical, row-by-row structure, listing column names followed by their values. This significantly improves readability when many columns are involved, even if the individual cell content remains long.

Alternatively, for comprehensive textual inspection, developers often bypass the terminal display altogether and opt to write a small, targeted sample slice of the data to a local file format like CSV or JSON, which can then be opened and reviewed using specialized text editors. This allows for easier scrolling, searching, and manipulation of exceptionally long strings without the constraints of the console output.

Mastering the `truncate` parameter remains a foundational skill for efficient data exploration in PySpark. It grants the data professional immediate, precise control over how data is rendered for verification and debugging purposes, thereby enhancing the overall speed and accuracy of the data workflow.

For detailed specifications on all parameters, developers should consult the official Apache Spark documentation for the PySpark `DataFrame.show` function.

Additional PySpark Resources

The following resources detail how to perform other common and essential tasks when working with PySpark DataFrames:

Understanding Schema Definition and Type Casting

Efficiently Filtering and Selecting Columns

Optimizing Joins and Aggregations