Learning PySpark: A Comprehensive Guide to Extracting Day of the Week from DataFrame Dates

Authored by

Mohammed looti

November 11, 2025

When conducting sophisticated time-series analysis or preparing massive datasets within a big data environment, extracting granular temporal features is often paramount. One of the most common requirements is determining the specific day of the week associated with a date column. This capability is fundamental for analysts seeking to uncover inherent weekly or seasonal patterns, optimize business cycles, or schedule automated reports based on specific weekdays. This comprehensive guide details the efficient methods available within the PySpark framework to determine the day of the week for date columns within a DataFrame.
We will systematically explore four distinct approaches, ranging from returning a simple numerical index, which adheres to different regional standards, to providing the full, descriptive name of the day for clearer reporting. Understanding these variations is essential for ensuring data integrity and aligning analysis with specific business or geographical requirements.

The Apache Spark ecosystem is specifically designed for fast and scalable data manipulation, and this efficiency extends seamlessly to date and time processing. By leveraging the powerful, built-in SQL functions provided by PySpark, developers can execute complex date transformations across enormous datasets without compromising performance. These techniques are vital for organizations handling petabytes of streaming or historical data. Crucially, we must address the difference in week definition--specifically, whether the week starts on Sunday (index 1) or Monday (index 1)--as this variation can profoundly impact downstream analytical results and data aggregation strategies, requiring careful implementation based on regional or analytical requirements.

Core PySpark Functions for Date Manipulation

The primary resource for all temporal transformations in PySpark is the `pyspark.sql.functions` module. This library provides specialized functions optimized for handling date and time data types effectively in a distributed computing context. Depending on whether your analysis requires a numerical index for mathematical operations (like clustering or aggregation) or a human-readable, formatted name for reporting, you will select one of two main function families: `F.dayofweek()` or `F.date_format()`.

Choosing the correct function depends entirely on the required output format and the specific analytical definition of the week. For raw numerical extraction based on the calendar, the native `F.dayofweek()` is the default choice. However, for maximum flexibility in formatting and localization--allowing control over abbreviations and full names--`F.date_format()` is invaluable. Mastering both ensures you can adapt your data pipeline to any date extraction requirement thrown at a distributed DataFrame, guaranteeing clean and accurate temporal features.

The four primary methods detailed below cover the spectrum of typical day-of-week extraction use cases encountered in PySpark data engineering workflows. Each method serves a slightly different

purpose in transforming the raw date column into a meaningful temporal feature:

Method 1: Numerical Index (US Standard) - Uses the standard F.dayofweek() function, where the index starts at 1 for Sunday and concludes at 7 for Saturday.
Method 2: Numerical Index (ISO Standard) - Achieves ISO standard compliance (Monday=1) through a calculated transformation utilizing modulo arithmetic to re-index the output.
Method 3: Abbreviated Day Name - Employs `F.date_format()` with the format pattern `'E'` to provide concise, three-letter abbreviations (e.g., 'Mon', 'Wed').
Method 4: Full Descriptive Day Name - Leverages `F.date_format()` with the pattern `'EEEE'` to return the complete, spelled-out name of the day (e.g., 'Monday', 'Friday').

Setting Up the Environment and Sample DataFrame

Before diving into the specific extraction syntax and corresponding examples, it is necessary to establish a functional PySpark environment and generate a sample **DataFrame**. The entire framework relies on the concept of the `SparkSession`, which acts as the foundational entry point for interacting with the distributed cluster via the Dataset and DataFrame API. Ensuring your `SparkSession` is properly initialized is the first crucial step in any Spark application, laying the groundwork for all subsequent distributed computations.

For demonstration purposes, we will construct a simple DataFrame named `df`. This DataFrame is deliberately structured with a standard date column and a corresponding column tracking sales figures. This setup allows us to practically illustrate how the different day-of-week extraction methods operate on realistic data structures using the powerful `withColumn` transformation, which is essential for appending new, derived feature columns to an existing dataset.

The following code snippet initializes the necessary components--the `SparkSession` and the sample data--and displays the base data structure we will be transforming throughout the subsequent examples, ensuring reproducibility and clarity:

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

# Define the raw data, pairing dates and sales volume.
data = ,
,
,
,
,
,
,

]

Define column names for clarity.
columns =

# Create the DataFrame using the defined schema and data.
df = spark.createDataFrame(data, columns)

# Display the initial structure of the DataFrame.
df.show()

+----------+-----+
| date|sales|
|2023-04-11| 22|
|2023-04-15| 14|
|2023-04-17| 12|
|2023-05-21| 15|
|2023-05-23| 30|
|2023-10-26| 45|
|2023-10-28| 32|
|2023-10-29| 47|
+----------+-----+

The resulting DataFrame `df` now serves as the foundation for all subsequent transformations. In each example that follows, we will apply the appropriate PySpark function via the `withColumn` operation, appending a new column to the DataFrame to clearly showcase the distinct outputs of the four primary day-of-week extraction methods.

Implementing Numerical Indexing: Standard vs. ISO Week Start

The numerical index is frequently required for computational tasks, such as grouping data, calculating averages by day type, or feeding features into statistical models. The simplest approach uses the native `F.dayofweek()` function, which returns an integer index ranging from 1 to 7. By default, PySpark adheres to the US calendar standard where the week begins on Sunday (index 1) and concludes on Saturday (index 7). This method is optimal when your internal calendar or regional reporting standards dictate a Sunday start, providing the most direct and computationally efficient solution.

To implement this standard numerical extraction, we utilize the `withColumn` transformation combined with `F.dayofweek()`. In the output below, observe how April 11th, 2023 (a Tuesday), returns 3, and May 21st, 2023 (a Sunday), returns 1. This immediate numerical output is perfectly

suited for quick aggregations where the index mapping is internally managed.

import pyspark.sql.functions as F

# Add new column that displays day of week as number (Sunday=1).
df_new = df.withColumn('day_of_week_us', F.dayofweek('date'))

# View the new DataFrame with the extracted day indices.
df_new.show()

+----------+-----+----------------+
| date|sales|day_of_week_us|
|2023-04-11| 22| 3|
|2023-04-15| 14| 7|
|2023-04-17| 12| 2|
|2023-05-21| 15| 1|
|2023-05-23| 30| 3|
|2023-10-26| 45| 5|
|2023-10-28| 32| 7|
|2023-10-29| 47| 1|
+----------+-----+----------------+

For many international business applications and regulatory standards, including the ISO standard (ISO 8601), **Monday** is explicitly defined as the start of the week (index 1). To align with this critical requirement, we must adjust PySpark's default output using a simple, yet powerful, mathematical transformation based on modulo arithmetic. This transformation ensures that Monday is mapped to 1, and Sunday is repositioned to 7, providing accurate week-based reporting.

The specific arithmetic formula used is `((F.dayofweek(date) + 5) % 7) + 1`. This formula systematically shifts the numerical indices. The addition of 5 and the subsequent modulo 7 operation effectively converts the US standard range (Sun=1 to Sat=7) into the ISO standard range (Mon=1 to Sun=7). This customized indexing is vital when integrating data derived from disparate systems or when adhering to specific international data reporting mandates.

import pyspark.sql.functions as F

# Add new column that displays day of week as number (Monday=1).
df_new = df.withColumn('day_of_week_iso', ((F.dayofweek('date')+5)%7)+1)

# View the new DataFrame.
df_new.show()

```
+----------+-----+---------------+
| date|sales|day_of_week_iso|
|2023-04-11| 22| 2|
|2023-04-15| 14| 6|
|2023-04-17| 12| 1|
|2023-05-21| 15| 7|
|2023-05-23| 30| 2|
|2023-10-26| 45| 4|
|2023-10-28| 32| 6|
|2023-10-29| 47| 7|
+----------+-----+---------------+
```

Extracting Descriptive Day Names: Abbreviation vs. Full Text

When the primary goal is human consumption of the data, numerical indices become less useful than descriptive text. To produce human-readable day names, we utilize the highly flexible `F.date_format()` function from the pyspark.sql.functions module. This function allows us to specify format patterns, similar to those used in Java's `SimpleDateFormat`, to extract the exact textual representation required.

The format pattern `'E'` is the key to extracting the standard three-letter abbreviation of the day name (e.g., 'Mon', 'Wed'). This approach is frequently deployed when designing visualizations or reports where screen real estate is limited. The resulting column provides descriptive clarity while maintaining a compact structure, making it ideal for dense data displays or intermediate quality checks.

import pyspark.sql.functions as F

```
# Add new column that displays day of week as abbreviated name.
df_new = df.withColumn('day_of_week_abbr', F.date_format('date', 'E'))

# View the new DataFrame.
df_new.show()
```

```
+----------+-----+---------------+
| date|sales|day_of_week_abbr|
|2023-04-11| 22| Tue|
|2023-04-15| 14| Sat|
|2023-04-17| 12| Mon|
|2023-05-21| 15| Sun|
```

|2023-05-23| 30| Tue|
|2023-10-26| 45| Thu|
|2023-10-28| 32| Sat|
|2023-10-29| 47| Sun|
+----------+-----+---------------+

For maximum clarity and formal reporting, the full descriptive name of the day is necessary. By using the format pattern `'EEEE'` within the `F.date_format()` function, we instruct **PySpark** to return the entire name (e.g., 'Monday', 'Tuesday'). This format is widely utilized when preparing final, human-readable reports or dashboards, ensuring that there is no ambiguity regarding the temporal context of the data points. This method is crucial for outputs that must be immediately and universally understandable by non-technical audiences.

import pyspark.sql.functions as F

# Add new column that displays day of week as full name.
df_new = df.withColumn('day_of_week_full', F.date_format('date', 'EEEE'))

# View the new DataFrame.
df_new.show()

+----------+-----+---------------+
| date|sales|day_of_week_full|
|2023-04-11| 22| Tuesday|
|2023-04-15| 14| Saturday|
|2023-04-17| 12| Monday|
|2023-05-21| 15| Sunday|
|2023-05-23| 30| Tuesday|
|2023-10-26| 45| Thursday|
|2023-10-28| 32| Saturday|
|2023-10-29| 47| Sunday|
+----------+-----+---------------+

Summary and Best Practices for Temporal PySpark Transformations

Extracting the day of the week from date columns in PySpark is a fundamental skill for any data engineer or analyst working with temporal data. Whether you require a numerical index for computational purposes or a descriptive string for reporting, the `pyspark.sql.functions` library offers robust, scalable tools to achieve your goal. The choice between using `F.dayofweek()` for raw indices and `F.date_format()` for flexible formatting depends entirely on the required output

format and the specific definition of the start of the week needed for your analysis.

For example, if the processed data is intended for machine learning feature engineering, the numerical index (Methods 1 or 2) is almost always the appropriate choice due to its optimization for mathematical operations. Conversely, when populating a business intelligence dashboard or validating data quality during a QA step, the descriptive strings (Methods 3 or 4) offer immediate value and reduced human interpretation error. In all scenarios, the use of Apache Spark's built-in functions ensures that the transformation is executed in a highly scalable and optimized manner across the distributed computing cluster.

A critical best practice involves always verifying the underlying assumption about the start of the week. Failing to account for the difference between the Sunday=1 default and the required Monday=1 ISO standard can lead to subtle yet significant aggregation errors, potentially misrepresenting weekly patterns or business cycle trends in your DataFrame. By utilizing the simple modulo arithmetic correction when necessary, you ensure your temporal analysis remains accurate and aligned with global data standards, thereby maintaining maximum data integrity throughout the entire processing workflow.