



PySpark Syntax Cheat Sheet - Quick Syntax Guide for Data Engineers

Create DataFrame.....	3
With Default Schema.....	3
Explicit Schema.....	3
Using a List of Dictionaries.....	4
Reading Files.....	4
CSV Files.....	4
JSON Files.....	5
CSV Files.....	5
Selecting Columns.....	8
Renaming Columns.....	8
Adding Columns.....	9
Dropping Columns.....	10
Filtering.....	10
Basic Filtering.....	10
Filter with Multiple Conditions.....	10
String Filters.....	11
Null Filters.....	11
Filter from a List.....	11
Data Cleansing.....	12
Grouping.....	12
Basic Aggregations without Grouping.....	12
Aggregations with Grouping.....	13
Common Aggregation Functions.....	13
Joins.....	14
Join Types.....	14
Basic Syntax.....	15
Date and Time Functions.....	15
String to Date Format.....	15
String to Timestamp Format.....	16
Date to String Format.....	17



Timestamp to String Format.....	18
Date Functions.....	19
Time Functions.....	21
Math Functions.....	21
Simple Arithmetic.....	22
Complex Arithmetic.....	22
String Functions.....	23
Basic String Functions.....	24
Trim and Pad Functions.....	25
Advanced String Functions.....	26
Converting to Other Data Types.....	27
Window Functions.....	28
Basic Window Functions.....	28
With Rows Between.....	29
Array Functions.....	30
Creating and Manipulating Arrays.....	30
Array Elements.....	31
Modifying Array Elements.....	31
Arrays to Rows.....	32
Rows to Array.....	32
Running SQL Queries.....	32
With Temp View.....	32
Without Temp View.....	33



Create DataFrame

With Default Schema

```
from pyspark.sql import SparkSession

spark = SparkSession.builder.appName("Example").getOrCreate()

data = [(1, "Alice", 29), (2, "Bob", 35)]
df = spark.createDataFrame(data, ["id", "name", "age"])
df.show()
```

Explicit Schema

```
from pyspark.sql.types import StructType, StructField, IntegerType,
StringType

schema = StructType([
    StructField("id", IntegerType(), True),
    StructField("name", StringType(), True),
    StructField("age", IntegerType(), True)
])

df = spark.createDataFrame(data, schema)
df.printSchema()
df.show()

# Schema as a string
data = [(1, "Alice", 29), (2, "Bob", 35)]
schema = "id INT, name STRING, age INT"
df = spark.createDataFrame(data, schema=schema)

# Schema String with Float and Boolean Types
schema = "id INT, name STRING, salary FLOAT, is_active BOOLEAN"
data = [(1, "Alice", 50000.75, True), (2, "Bob", 60000.50, False)]
df = spark.createDataFrame(data, schema=schema)

# Schema String with Date and Timestamp
from datetime import date, datetime
schema = "id INT, name STRING, join_date DATE, last_login TIMESTAMP"
```



```
data = [(1, "Alice", date(2023, 1, 15), datetime(2024, 3, 10, 14, 30, 0)),  
        (2, "Bob", date(2023, 1, 15), datetime(2024, 3, 10, 14, 30, 0))]  
df = spark.createDataFrame(data, schema=schema)
```

Using a List of Dictionaries

```
from pyspark.sql import SparkSession  
  
spark = SparkSession.builder.appName("Example").getOrCreate()  
  
data = [  
    {"id": 1, "name": "Alice", "age": 29},  
    {"id": 2, "name": "Bob", "age": 35}  
]  
  
df = spark.createDataFrame(data)  
df.show()
```

Reading Files

CSV Files

```
#Basic CSV files  
df = spark.read.format("csv").load("/path/to/sample.csv")  
  
#csv with header  
df = spark.read.option("header",True).csv("/path/to/sample.csv")  
  
# multiple options  
df =  
spark.read.option("inferSchema",True).option("delimiter",",").csv("/path/to/  
/sample.csv")  
  
# with defined schema  
from pyspark.sql.types import StructType, StructField, StringType,  
IntegerType
```



```
schema = StructType([
    StructField("name", StringType(), True),
    StructField("age", IntegerType(), True)
])
df = spark.read.format("csv").schema(schema).load("/path/to/sample.csv")
```

JSON Files

```
# Basic JSON file
df = spark.read.format("json").load("/path/to/sample.json")

# JSON with multi-line records
df = spark.read.option("multiline", True).json("/path/to/sample.json")

# JSON with a defined schema
from pyspark.sql.types import StructType, StructField, StringType,
IntegerType

schema = StructType([
    StructField("name", StringType(), True),
    StructField("age", IntegerType(), True)
])

df = spark.read.format("json").schema(schema).load("/path/to/sample.json")
```

Writing Files

CSV Files

```
# Basic write to CSV
df.write.csv("/path/to/output_csv")

# With header
df.write.option("header", True).csv("/path/to/output_csv")

# With multiple options
df.write.option("header", True) \
.option("delimiter", ",") \
.option("quote", "'") \
.csv("/path/to/output_csv")
```



```
# Overwrite existing files
df.write.mode("overwrite").option("header",
True).csv("/path/to/output_csv")

# Append to existing data
df.write.mode("append").option("header", True).csv("/path/to/output_csv")

# Write as a single file
df.coalesce(1).write.option("header", True).csv("/path/to/output_csv")
```

JSON Files

```
# Basic JSON write
df.write.json("/path/to/output_json")

# Overwrite mode
df.write.mode("overwrite").json("/path/to/output_json")

# Append mode
df.write.mode("append").json("/path/to/output_json")

# Pretty format (for readability)
df.write.option("compression", "none").json("/path/to/output_json")

# Partitioned output
df.write.partitionBy("column_name").json("/path/to/output_json")
```

Parquet Files

```
# Basic Parquet write
df.write.parquet("/path/to/output_parquet")

# Overwrite mode
df.write.mode("overwrite").parquet("/path/to/output_parquet")

# Append mode
df.write.mode("append").parquet("/path/to/output_parquet")

# Partitioned output
df.write.partitionBy("column_name").parquet("/path/to/output_parquet")
```



```
# Compression options (default is snappy)
df.write.option("compression", "gzip").parquet("/path/to/output_parquet")
```

ORC Files

```
# Basic ORC write
df.write.orc("/path/to/output_orc")

# Overwrite mode
df.write.mode("overwrite").orc("/path/to/output_orc")

# Append mode
df.write.mode("append").orc("/path/to/output_orc")

# Partitioned output
df.write.partitionBy("column_name").orc("/path/to/output_orc")

# Compression options
df.write.option("compression", "zlib").orc("/path/to/output_orc")
```

Delta Tables/Files (*Requires Delta Lake*)

```
# Basic Delta write
df.write.format("delta").save("/path/to/output_delta")

# Basic Delta Table
df.write.format("delta").saveAsTable("table_name")

# Overwrite mode
df.write.format("delta").mode("overwrite").save("/path/to/output_delta")

# Append mode
df.write.format("delta").mode("append").save("/path/to/output_delta")

# With Delta-specific options (mergeSchema, overwriteSchema)
df.write.format("delta")\
    .option("overwriteSchema", "true")\
    .mode("overwrite")\
    .save("/path/to/output_delta")
```



```

# Partitioned output
df.write.format("delta").partitionBy("column_name").save("/path/to/output_d
elta")

# Bucketing
df.write.bucketBy(numBuckets=4, col="column_name")\
    .sortBy("column_name")\
    .saveAsTable("table_name")

# Bucketing by "name" into 4 buckets, sorting within each bucket
df.write \
    .bucketBy(4, "name") \
    .sortBy("age") \
    .mode("overwrite") \
    .saveAsTable("bucketed_people")

```

Select, Drop, Rename Columns

Selecting Columns

```

# Select single column
df = df.select("name")

# Select multiple columns
df = df.select("name", "age")

# Select columns dynamically
columns_to_select = ["name", "department"]
df = df.select(*columns_to_select)

```

Renaming Columns

```

# Rename a column
df = df.withColumnRenamed("name", "full_name")

# Rename multiple columns with chained calls
df = df.withColumnRenamed("old_col1", "new_col1")\
    .withColumnRenamed("old_col2", "new_col2")

# Rename columns using select and alias
from pyspark.sql.functions import col

```



```
df = df.select(
    col("old_column_name1").alias("new_column_name1"),
    col("old_column_name2").alias("new_column_name2"),
    # Add more columns as needed
)
```

Adding Columns

```
# Rename a column
df = df.withColumnRenamed("name", "full_name")

# Rename multiple columns with chained calls
df = df.withColumnRenamed("old_col1", "new_col1")\
    .withColumnRenamed("old_col2", "new_col2")

# Rename columns using select and alias
from pyspark.sql.functions import col
df = df.select(
    col("old_column_name1").alias("new_column_name1"),
    col("old_column_name2").alias("new_column_name2"),
    # Add more columns as needed
)
from pyspark.sql.functions import col, lit, expr, when

# Add a new column with a constant value
df = df.withColumn("country", lit("USA"))

# Add a new column with a calculated value
df = df.withColumn("salary_after_bonus", col("salary") * 1.1)

# Add a column using an SQL expression
df = df.withColumn("tax", expr("salary * 0.2"))

# Add a column with conditional logic
df = df.withColumn("high_earner", when(col("salary") > 55000,
    "Yes").otherwise("No"))

# Case When with multiple conditions
df = df.withColumn(
    "salary_category",
    when(col("salary") < 60000, "Low")
```



```
.when((col("salary") >= 60000) & (col("salary") < 90000), "Medium")
.otherwise("High")
)

# Add multiple columns at once
df = df.withColumns({
    "bonus": col("salary") * 0.1,
    "net_salary": col("salary") - (col("salary") * 0.2)
})
```

Dropping Columns

```
# Drop a column
df = df.drop("department")

# Drop multiple columns
df = df.drop('column1', 'column2', 'column3')
```

Filtering

You can refer to columns using any of these notations: `df.age` , `df['age']`, `col('age')`

Basic Filtering

```
# Filter on >, <, >=, <=, == condition
df_filtered = df.filter(df.age > 30)
df_filtered = df.filter(df['age'] > 30)

# Using col() function
from pyspark.sql.functions import col
df_filtered = df.filter(col("age") > 30)
```

Filter with Multiple Conditions

```
# Multiple conditions require parentheses around each condition

# AND condition ( & )
df_filtered = df.filter((df.age > 25) & (df.department == "Engineering"))
```



```
# OR condition ( | )
df_filtered = df.filter((df.age < 30) | (df.department == "Finance"))
```

String Filters

```
# Filter rows where department equals 'Marketing'
df_filtered = df.filter(df.department == "Marketing")

# Case-insensitive filter
df_filtered = df.filter(col("department").like("MARKETING"))

# Contains a substring
df_filtered = df.filter(col("department").contains("Engineer"))

# Filter rows where the name starts with 'A'
df.filter(col("name").startswith("A")).show()

# Filter rows where the name ends with 'e'
df.filter(col("name").endswith("e")).show()

# Filter rows where the name matches a regex
df.filter(col("name").rlike("^A.*")).show()
```

Null Filters

```
# Filter rows where a column is null
df_filtered = df.filter(df.department.isNull())
# Filter rows where a column is not null
df_filtered = df.filter(df.department.isNotNull())
```

Filter from a List

```
# Filter rows where department is in a list
departments = ["Engineering", "Finance"]
df_filtered = df.filter(col("department").isin(departments))
# Negate the filter (not in list)
df_filtered = df.filter(~col("department").isin(departments))
```



Data Cleansing

```
# 1. Drop all fully duplicate rows
# Removes rows where all columns match exactly
df = df.dropDuplicates()

# 2. Drop duplicates based on specific columns
# Keeps the first row for each unique email
df = df.dropDuplicates(["email"])

# 3. Get only distinct rows (same as SELECT DISTINCT)
# Removes duplicates across all columns
df = df.distinct()

# 4. Drop rows with any null values
# Removes rows with even a single null field
df = df.dropna()

# 5. Drop rows with nulls in specific columns
# Only keeps rows where 'email' and 'age' are not null
df = df.dropna(subset=["email", "age"])

# 6. Fill missing values for all columns
# Replaces all nulls with a default value
df = df.fillna("N/A")

# 7. Fill missing values for specific columns
# Sets default age as 0 and country as "Unknown" if missing
df = df.fillna({"age": 0, "country": "Unknown"})
```

Grouping

Import the required functions

```
from pyspark.sql.functions import count, sum, avg, min, max, countDistinct,
collect_list, collect_set
```

Basic Aggregations without Grouping



```

#Count rows
df.count()

#Count Distinct Values in a column
df.select(countDistinct("Department")).show()

#Sum
df.select(sum("Salary")).show()

#Multiple Aggregations
df.select(min("Salary"), max("Salary")).show()

```

Aggregations with Grouping

```

#Group by a single column
df.groupBy("Department").sum("Salary").show()

#GroupBy with Multiple Columns
df.groupBy("Department", "Employee").sum("Salary").show()

#Group by with multiple aggregations
df.groupBy("Department").agg(
    count("Employee").alias("Employee_Count"),
    avg("Salary").alias("Average_Salary"),
    max("Salary").alias("Max_Salary")
)

#Filter after aggregation
df.groupBy("Department").agg(sum("Salary").alias("Total_Salary")).filter("Total_Salary > 8000").show()

```

Common Aggregation Functions

Function	Description	Example
count()	Counts rows in a group.	groupBy("Department").count()
sum()	Sums values in a group.	groupBy("Department").sum("Salary")
avg() / mean()	Calculates average values.	groupBy("Department").avg("Salary")



Function	Description	Example
min()	Finds the minimum value.	groupBy("Department").min("Salary")
max()	Finds the maximum value.	groupBy("Department").max("Salary")
countDistinct()	Counts distinct values in a group.	countDistinct("Employee")
collect_list()	Collects all values into a list.	collect_list("Employee")
collect_set()	Collects unique values into a set.	collect_set("Employee")

Joins

Join Types

Join Type	Syntax	Description
inner	how="inner"	Returns matching rows from both DataFrames based on the join condition.
outer (full)	how="outer"	Returns all rows, with NULL where no match is found in either DataFrame.
left (left_outer)	how="left"	Returns all rows from the left DataFrame, with NULL for unmatched rows in the right.
right (right_outer)	how="right"	Returns all rows from the right DataFrame, with NULL for unmatched rows in the left.
left_semi	how="left_semi"	This is just an inner join of the two DataFrames, but only returns columns of left DataFrame.
left_anti	how="left_anti"	Returns rows from the left DataFrame that do not have a match in the right.
cross	df1.crossJoin(df2)	Returns the Cartesian product of rows from both DataFrames (no join condition).



Basic Syntax

```
# Basic Join
df1.join(df2, on="id", how="inner")

# Join on Multiple Columns
df1.join(df2, on=["col1", "col2"], how="left")

# Conditional Join
df1.join(df2, (df1.id == df2.id) & (df2.city == "New York"), how="inner")
# Multiple join conditions require parentheses around each condition
joined_df = sales_df.join(
    customers_df,
    (sales_df["customer_id"] == customers_df["customer_id"]) &
(sales_df["region"] == customers_df["region"]),
    "inner"
)

# Select ALL columns from df1, and SOME columns from df2 (useful for left
joins)
result = df1.join(df2, on="id", how="left").select(df1["*"], df2["state"] ,
df2["town"])

# Broadcast Join for Small DataFrames
from pyspark.sql.functions import broadcast
df1.join(broadcast(df2), on="id", how="inner")
```

Date and Time Functions

String to Date Format

```
from pyspark.sql.functions import to_date

# 1. Convert string date to date type (using "yyyy-MM-dd")
# Input: "2025-01-25"
# Output: 2025-01-25 (as a Date type)
df = df.withColumn("date_parsed1", to_date("date_str", "yyyy-MM-dd"))

# 2. Convert string date to date type (using "dd-MMM-yyyy")
# Input: "25-Jan-2025"
# Output: 2025-01-25 (as a Date type)
```



```
df = df.withColumn("date_parsed2", to_date("date_str", "dd-MMM-yyyy"))

# 3. Convert string date to date type (using "MM/dd/yyyy")
# Input: "01/25/2025"
# Output: 2025-01-25 (as a Date type)
df = df.withColumn("date_parsed3", to_date("date_str", "MM/dd/yyyy"))

# 4. Convert string date to date type (using "yyyy.MM.dd")
# Input: "2025.01.25"
# Output: 2025-01-25 (as a Date type)
df = df.withColumn("date_parsed4", to_date("date_str", "yyyy.MM.dd"))
```

String to Timestamp Format

```
from pyspark.sql.functions import to_timestamp

# 1. Convert string timestamp to timestamp type (using "yyyy-MM-dd
HH:mm:ss")
# Input: "2025-01-25 10:15:00"
# Output: 2025-01-25 10:15:00 (as a Timestamp type)
df = df.withColumn("timestamp_parsed1", to_timestamp("timestamp_str",
"yyyy-MM-dd HH:mm:ss"))

# 2. Convert string timestamp to timestamp type (using "dd-MMM-yyyy
HH:mm:ss")
# Input: "25-Jan-2025 10:15:00"
# Output: 2025-01-25 10:15:00 (as a Timestamp type)
df = df.withColumn("timestamp_parsed2", to_timestamp("timestamp_str",
"dd-MMM-yyyy HH:mm:ss"))

# 3. Convert string timestamp to timestamp type (using "MM/dd/yyyy
HH:mm:ss")
# Input: "01/25/2025 10:15:00"
# Output: 2025-01-25 10:15:00 (as a Timestamp type)
df = df.withColumn("timestamp_parsed3", to_timestamp("timestamp_str",
"MM/dd/yyyy HH:mm:ss"))

# 4. Convert string timestamp to timestamp type (using "yyyy.MM.dd
HH:mm:ss")
# Input: "2025.01.25 10:15:00"
# Output: 2025-01-25 10:15:00 (as a Timestamp type)
```



```
df = df.withColumn("timestamp_parsed4", to_timestamp("timestamp_str",  
"yyyy.MM.dd HH:mm:ss"))
```

Date to String Format

```
from pyspark.sql.functions import date_format  
# 1. Format date as "yyyy-MM-dd"  
# Input: 2025-01-25 (Date Type)  
# Output: "2025-01-25" (String Type)  
df = df.withColumn("formatted_date1", date_format("date_parsed",  
"yyyy-MM-dd"))  
  
# 2. Format date as "dd-MMM-yyyy"  
# Input: 2025-01-25 (Date Type)  
# Output: "25-Jan-2025" (String Type)  
df = df.withColumn("formatted_date2", date_format("date_parsed",  
"dd-MMM-yyyy"))  
  
# 3. Format date as "MM/dd/yyyy"  
# Input: 2025-01-25 (Date Type)  
# Output: "01/25/2025" (String Type)  
df = df.withColumn("formatted_date3", date_format("date_parsed",  
"MM/dd/yyyy"))  
  
# 4. Format date as "dd/MM/yyyy"  
# Input: 2025-01-25 (Date Type)  
# Output: "25/01/2025" (String Type)  
df = df.withColumn("formatted_date4", date_format("date_parsed",  
"dd/MM/yyyy"))  
  
# 5. Format date as "MMMM dd, yyyy"  
# Input: 2025-01-25 (Date Type)  
# Output: "January 25, 2025" (String Type)  
df = df.withColumn("formatted_date5", date_format("date_parsed", "MMMM dd,  
yyyy"))  
  
# 6. Format date as "EEE, dd MMM yyyy"  
# Input: 2025-01-25 (Date Type)  
# Output: "Sun, 25 Jan 2025" (String Type)  
df = df.withColumn("formatted_date6", date_format("date_parsed", "EEE, dd  
MMM yyyy"))
```



```

# 7. Format date as "yyyy/MM/dd"
# Input: 2025-01-25 (Date Type)
# Output: "2025/01/25" (String Type)
df = df.withColumn("formatted_date7", date_format("date_parsed",
"yyyy/MM/dd"))

# 8. Format date as "yyyy.MM.dd"
# Input: 2025-01-25 (Date Type)
# Output: "2025.01.25" (String Type)
df = df.withColumn("formatted_date8", date_format("date_parsed",
"yyyy.MM.dd"))

```

Timestamp to String Format

```

from pyspark.sql.functions import date_format

# 1. Format timestamp as "yyyy-MM-dd HH:mm:ss"
# Input: "2025-01-25 10:15:00" (Timestamp Type)
# Output: "2025-01-25 10:15:00" (String Type)
df = df.withColumn("formatted_timestamp1", date_format("timestamp",
"yyyy-MM-dd HH:mm:ss"))

# 2. Format timestamp as "dd-MMM-yyyy HH:mm:ss"
# Input: "2025-01-25 10:15:00" (Timestamp Type)
# Output: "25-Jan-2025 10:15:00" (String Type)
df = df.withColumn("formatted_timestamp2", date_format("timestamp",
"dd-MMM-yyyy HH:mm:ss"))

# 3. Format timestamp as "MM/dd/yyyy HH:mm:ss"
# Input: "2025-01-25 10:15:00" (Timestamp Type)
# Output: "01/25/2025 10:15:00" (String Type)
df = df.withColumn("formatted_timestamp3", date_format("timestamp",
"MM/dd/yyyy HH:mm:ss"))

# 4. Format timestamp as "dd/MM/yyyy HH:mm:ss"
# Input: "2025-01-25 10:15:00" (Timestamp Type)
# Output: "25/01/2025 10:15:00" (String Type)
df = df.withColumn("formatted_timestamp4", date_format("timestamp",
"dd/MM/yyyy HH:mm:ss"))

```



```

# 5. Format timestamp as "MMMM dd, yyyy HH:mm:ss"
# Input: "2025-01-25 10:15:00" (Timestamp Type)
# Output: "January 25, 2025 10:15:00" (String Type)
df = df.withColumn("formatted_timestamp5", date_format("timestamp", "MMMM
dd, yyyy HH:mm:ss"))

# 6. Format timestamp as "EEE, dd MMM yyyy HH:mm:ss"
# Input: "2025-01-25 10:15:00" (Timestamp Type)
# Output: "Sun, 25 Jan 2025 10:15:00" (String Type)
df = df.withColumn("formatted_timestamp6", date_format("timestamp", "EEE,
dd MMM yyyy HH:mm:ss"))

# 7. Format timestamp as "yyyy/MM/dd HH:mm:ss"
# Input: "2025-01-25 10:15:00" (Timestamp Type)
# Output: "2025/01/25 10:15:00" (String Type)
df = df.withColumn("formatted_timestamp7", date_format("timestamp",
"yyyy/MM/dd HH:mm:ss"))

# 8. Format timestamp as "yyyy.MM.dd HH:mm:ss"
# Input: "2025-01-25 10:15:00" (Timestamp Type)
# Output: "2025.01.25 10:15:00" (String Type)
df = df.withColumn("formatted_timestamp8", date_format("timestamp",
"yyyy.MM.dd HH:mm:ss"))

# Show the resulting DataFrame
df.show(truncate=False)

```

Date Functions

```

from pyspark.sql import SparkSession
from pyspark.sql.functions import (
    current_date, date_add, date_sub, datediff, add_months,
    trunc, date_format, year, month, dayofmonth, next_day, last_day
)

# 1. Current date
# "2025-01-25" -> Current system date (e.g., "2025-01-25")
df = df.withColumn("current_date", current_date())

# 2. Add 10 days to the date
# "2025-01-25" -> "2025-02-04"

```



```
df = df.withColumn("date_plus_10", date_add("date", 10))

# 3. Subtract 5 days from the date
# "2025-01-25" -> "2025-01-20"
df = df.withColumn("date_minus_5", date_sub("date", 5))

# 4. Difference in days from current date
# "2025-01-25" -> Number of days difference from today (e.g., "-5")
df = df.withColumn("days_diff", datediff(current_date(), "date"))

# 5. Add 2 months to the date
# "2025-01-25" -> "2025-03-25"
df = df.withColumn("add_months", add_months("date", 2))

# 6. Extract year
# "2025-01-25" -> "2025"
df = df.withColumn("year", year("date"))

# 7. Extract month
# "2025-01-25" -> "1"
df = df.withColumn("month", month("date"))

# 8. Extract day of the month
# "2025-01-25" -> "25"
df = df.withColumn("day", dayofmonth("date"))

# 9. Extract day of the week (1 = Sunday, 7 = Saturday)
# Input: "2025-01-25"
# Output: 7 (Saturday)
df = df.withColumn("day_of_week", dayofweek("date"))

# 10. Extract week of the year
# Input: "2025-01-25"
# Output: 4 (Week 4 of the year)
df = df.withColumn("week_of_year", weekofyear("date"))

# 11. Truncate to the first day of the month
# "2025-01-25" -> "2025-01-01"
df = df.withColumn("trunc_month", trunc("date", "MM"))

# 12. Next specified day of the week
# "2025-01-25" -> Next Monday (e.g., "2025-01-27")
```



```
df = df.withColumn("next_monday", next_day("date", "Monday"))

# 13. Last day of the month
# "2025-01-25" -> "2025-01-31"
df = df.withColumn("last_day_month", last_day("date"))
```

Time Functions

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import (
    current_timestamp, hour, minute, second, unix_timestamp, from_unixtime
)

# 1. Current timestamp
# "2025-01-25 10:15:00" -> Current system timestamp (e.g., "2025-01-25
# 10:15:00")
df = df.withColumn("current_timestamp", current_timestamp())

# 2. Extract hour
# "2025-01-25 10:15:00" -> "10"
df = df.withColumn("hour", hour("timestamp"))

# 3. Extract minute
# "2025-01-25 10:15:00" -> "15"
df = df.withColumn("minute", minute("timestamp"))

# 4. Extract second
# "2025-01-25 10:15:00" -> "00"
df = df.withColumn("second", second("timestamp"))

# 5. Convert date to Unix timestamp
# "2025-01-25 10:15:00" -> "1737763200"
df = df.withColumn("unix_timestamp", unix_timestamp("timestamp"))

# 6. Convert Unix timestamp to readable date
# "1737763200" -> "2025-01-25 10:15:00"
df = df.withColumn("from_unix", from_unixtime(unix_timestamp("timestamp")))
```

Math Functions



Simple Arithmetic

```
# 1. Add two columns
# Input: col1 = 10, col2 = 5
# Output: 15 (col1 + col2)
df = df.withColumn("sum", col("col1") + col("col2"))

# 2. Subtract two columns
# Input: col1 = 10, col2 = 5
# Output: 5 (col1 - col2)
df = df.withColumn("difference", col("col1") - col("col2"))

# 3. Multiply two columns
# Input: col1 = 10, col2 = 5
# Output: 50 (col1 * col2)
df = df.withColumn("product", col("col1") * col("col2"))

# 4. Divide two columns
# Input: col1 = 10, col2 = 5
# Output: 2.0 (col1 / col2)
df = df.withColumn("quotient", col("col1") / col("col2"))

# 5. Add a constant to a column
# Input: col1 = 10
# Output: 15 (col1 + 5)
df = df.withColumn("sum_with_constant", col("col1") + 5)

# 6. Subtract a constant from a column
# Input: col1 = 10
# Output: 5 (col1 - 5)
df = df.withColumn("difference_with_constant", col("col1") - 5)
```

Complex Arithmetic

```
from pyspark.sql.functions import (
    abs, round, floor, ceil, exp, log, sqrt, pow
)

# 1. Absolute value
# Input: -2.71
```



```
# Output: 2.71
df = df.withColumn("abs_value", abs("value"))

# 2. Round the number to 2 decimal places
# Input: 3.14159
# Output: 3.14
df = df.withColumn("rounded_value", round("value", 2))

# 3. Floor (round down to the nearest integer)
# Input: 3.14
# Output: 3
df = df.withColumn("floor_value", floor("value"))

# 4. Ceil (round up to the nearest integer)
# Input: 3.14
# Output: 4
df = df.withColumn("ceil_value", ceil("value"))

# 5. Exponent (e raised to the power of the value)
# Input: 2.0
# Output: 7.389056
df = df.withColumn("exp_value", exp("value"))

# 6. Logarithm (log base e of the value)
# Input: 2.718
# Output: 0.999896
df = df.withColumn("log_value", log("value"))

# 7. Square root
# Input: 16
# Output: 4
df = df.withColumn("sqrt_value", sqrt("value"))

# 8. Power (raise the value to the power of 2)
# Input: 3
# Output: 9
df = df.withColumn("pow_value", pow("value", 2))
```

String Functions



Basic String Functions

```
# 1. Concatenate two strings
# Input: "hello world" + " !!!"
# Output: "hello world !!!"
df = df.withColumn("concatenated_2_cols", concat(col("col1"), col("col2")))
df = df.withColumn("concatenated_col_with_lit", concat(col("text"), lit("!!!
!")))

# 2. Concatenate columns with a separator (Space)
# Input: ("John", "Doe", "30")
# Output: "John Doe 30"
df = df.withColumn("full_name", concat_ws(" ", col("first_name"),
col("last_name"), col("age")))

# 3. Concatenate columns with a separator (Comma)
# Input: ("John", "Doe", "30")
# Output: "John, Doe, 30"
df = df.withColumn("full_name_comma", concat_ws(", ", col("first_name"),
col("last_name"), col("age")))

# 4. Concatenate with a custom string
# Input: ("John", "Doe")
# Output: "Name: John Doe"
df = df.withColumn("name", concat_ws("", lit("Name: "), col("first_name"),
lit(" "), col("last_name")))

# 5. Check if string contains a substring
# Input: "hello world" -> "world"
# Output: True
df = df.withColumn("contains_world", col("text").contains("world"))
df = df.withColumn("contains_world2", contains(col("text"), lit("world")))

# 6. Check if string starts with a specific substring
# Input: "hello world" -> "hello"
# Output: True
df = df.withColumn("starts_with_hello", col("text").startswith("hello"))
df = df.withColumn("starts_with_hello2", startswith(col("text"),
lit("hello")))

# 7. Check if string ends with a specific substring
# Input: "hello world" -> "world"
```



```

# Output: True
df = df.withColumn("ends_with_world", col("text").endswith("world"))
df = df.withColumn("ends_with_world2", endswith(col("text"), lit("world")))

# 8. Capitalize the first letter of each word
# Input: "hello world"
# Output: "Hello World"
df = df.withColumn("initcap_text", initcap(col("text")))

# 9. Convert string to uppercase
# Input: "hello world"
# Output: "HELLO WORLD"
df = df.withColumn("upper_text", upper(col("text")))

# 10. Convert string to lowercase
# Input: "HELLO WORLD"
# Output: "hello world"
df = df.withColumn("lower_text", lower(col("text")))

# 11. Get the length of the string
# Input: "hello world"
# Output: 11
df = df.withColumn("length_of_text", length(col("text")))

```

Trim and Pad Functions

```

# 1. Trim: Remove both leading and trailing spaces from first_name
# Input: " John "
# Output: "John"
df = df.withColumn("trimmed_first_name", trim(col("first_name")))

# 2. Ltrim: Remove leading spaces from first_name
# Input: " John"
# Output: "John"
df = df.withColumn("ltrim_first_name", ltrim(col("first_name")))

# 3. Rtrim: Remove trailing spaces from last_name
# Input: "Doe "
# Output: "Doe"
df = df.withColumn("rtrim_last_name", rtrim(col("last_name")))

```



```

# 4. Lpad: Pad first_name with spaces on the left to make the length 10
# Input: "John"
# Output: "      John"
df = df.withColumn("lpad_first_name", lpad(col("first_name"), 10, " "))

# 5. Rpad: Pad last_name with spaces on the right to make the length 10
# Input: "Doe"
# Output: "Doe      "
df = df.withColumn("rpad_last_name", rpad(col("last_name"), 10, " "))

# 6. Lpad with a custom padding character: Pad first_name with "0" on the
# left to make the length 10
# Input: "John"
# Output: "00000John"
df = df.withColumn("lpad_first_name_zeros", lpad(col("first_name"), 10,
"0"))

# 7. Rpad with a custom padding character: Pad last_name with "0" on the
# right to make the length 10
# Input: "Doe"
# Output: "Doe0000000"
df = df.withColumn("rpad_last_name_zeros", rpad(col("last_name"), 10, "0"))

```

Advanced String Functions

```

# 1. Substring: Extract substring from the full_name starting from position
# 1 (inclusive) with length 4
# Input: "John_Doe_30"
# Output: "John"
df = df.withColumn("substring_example", substring(col("full_name"), 1, 4))

# 2. Substring: Extract substring from the full_name starting from position
# 6 (inclusive) with length 3
# Input: "John_Doe_30"
# Output: "Doe"
df = df.withColumn("substring_name", substring(col("full_name"), 6, 3))

# 3. Substring: Extract last 2 characters of the full_name
# Input: "John_Doe_30"
# Output: "30"
df = df.withColumn("substring_age", substring(col("full_name"), -2, 2))

```



```

# 4. Split: Split the full_name into first and last names based on the "_" separator
# Input: "John_Doe_30"
# Output: ["John", "Doe", "30"]
df = df.withColumn("split_name", split(col("full_name"), "_"))

# 5. Split: Split the full_name into first and last names based on the "_" separator and get the first part (first name)
# Input: "John_Doe_30"
# Output: "John"
df = df.withColumn("first_name", split(col("full_name"), "_")[0])

# 6. Split: Split the full_name into first and last names and get the second part (last name)
# Input: "John_Doe_30"
# Output: "Doe"
df = df.withColumn("last_name", split(col("full_name"), "_")[1])

# 7. Split: Split the full_name and get the third part (age)
# Input: "John_Doe_30"
# Output: "30"
df = df.withColumn("age", split(col("full_name"), "_")[2])

```

Converting to Other Data Types

```

# 1. Convert string to integer
# Input: "12345"
# Output: 12345 (as an Integer type)
df = df.withColumn("int_parsed", col("int_str").cast("int"))

# 2. Convert string to float
# Input: "123.45"
# Output: 123.45 (as a Float type)
df = df.withColumn("float_parsed", col("int_str").cast("float"))

# 3. Convert string to double
# Input: "123.4567"
# Output: 123.4567 (as a Double type)
df = df.withColumn("double_parsed", col("int_str").cast("double"))

```



```

# 4. Convert string to long
# Input: "123456789012"
# Output: 123456789012 (as a Long type)
df = df.withColumn("long_parsed", col("int_str").cast("long"))

# 5. Convert integer to string
# Input: 12345
# Output: "12345" (as a String type)
df = df.withColumn("int_to_str", col("int_parsed").cast("string"))

# 6. Convert date to string
# Input: 2025-01-25 (Date type)
# Output: "2025-01-25" (String type)
df = df.withColumn("date_to_str", col("date_parsed").cast("string"))

# 7. Convert timestamp to string
# Input: 2025-01-25 10:15:00 (Timestamp type)
# Output: "2025-01-25 10:15:00" (String type)
df = df.withColumn("timestamp_to_str",
col("timestamp_parsed").cast("string"))

```

Window Functions

Basic Window Functions

Use **orderBy()** when order matters:

- Ranking Functions (row_number, rank, dense_rank)
- Offset Functions (lead, lag)
- Cumulative Aggregations (sum, avg with rowsBetween)

Skip **orderBy()** when order is irrelevant:

- Partition-wise Aggregates (sum, avg, count)
- Row-Agnostic Aggregations (max, min)

```

from pyspark.sql.window import Window
from pyspark.sql.functions import col, row_number, rank, dense_rank, lag,
lead, sum, avg

# Define window specification (partition by department, order by salary
descending)
window_spec =
Window.partitionBy("department").orderBy(col("salary").desc())

```



```

# Apply window functions

# **row_number:** Assigns unique numbers to each row in a partition.
df = df.withColumn("row_number", row_number().over(window_spec))

# **rank:** Similar to row_number but allows rank gaps.
df = df.withColumn("rank", rank().over(window_spec))

# **dense_rank:** Like rank but without gaps.
df = df.withColumn("dense_rank", dense_rank().over(window_spec))

# **lag:** Gets the previous row's value.
df = df.withColumn("previous_salary", lag("salary").over(window_spec))

# **lead:** Gets the next row's value.
df = df.withColumn("next_salary", lead("salary").over(window_spec))

# **sum:** Computes a running total.
df = df.withColumn("running_total", sum("salary").over(window_spec))

# **avg:** Computes a moving average.
df = df.withColumn("moving_avg", avg("salary").over(window_spec))

# Show result
df.show()

```

With Rows Between

```

from pyspark.sql.window import Window
from pyspark.sql.functions import col, sum, avg, min, max, count

#1. Rolling sum over the last 2 rows and current row
window_spec1 =
Window.partitionBy("department").orderBy("salary").rowsBetween(-2, 0)
df = df.withColumn("rolling_sum_last_2", sum("salary").over(window_spec1))

#2. Moving average including previous, current, and next row
window_spec2 =
Window.partitionBy("department").orderBy("salary").rowsBetween(-1, 1)
df = df.withColumn("moving_avg", avg("salary").over(window_spec2))

```



```

#3. Rolling minimum for current and next 2 rows
window_spec3 =
Window.partitionBy("department").orderBy("salary").rowsBetween(0, 2)
df = df.withColumn("rolling_min_next_2", min("salary").over(window_spec3))

#4. Maximum salary over all previous rows (running max)
window_spec4 =
Window.partitionBy("department").orderBy("salary").rowsBetween(Window.unboundedPreceding, 0)
df = df.withColumn("running_max", max("salary").over(window_spec4))

#5. Count total rows within the window (entire partition)
window_spec5 =
Window.partitionBy("department").orderBy("salary").rowsBetween(Window.unboundedPreceding, Window.unboundedFollowing)
df = df.withColumn("total_rows", count("salary").over(window_spec5))

# Show result
df.show()

```

Array Functions

Creating and Manipulating Arrays

```

from pyspark.sql.functions import array, size, sort_array, array_contains,
explode, lit

# 1. Create an array column from multiple columns
# Input: col1 = "A", col2 = "B"
# Output: ["A", "B"]
df = df.withColumn("combined_array", array("col1", "col2"))

# 2. Get the size of an array
# Input: ["A", "B", "C"]
# Output: 3
df = df.withColumn("array_size", size("tags"))

# 3. Sort array elements in ascending order
# Input: [3, 1, 2]
# Output: [1, 2, 3]

```



```
df = df.withColumn("sorted_array", sort_array("numbers"))

# 4. Check if array contains a specific value
# Input: ["red", "blue"], check for "blue"
# Output: true
df = df.withColumn("has_blue", array_contains("colors", "blue"))
```

Array Elements

```
from pyspark.sql.functions import element_at, col

# 1. Get element at a specific position (1-based index)
# Input: ["x", "y", "z"], index = 2
# Output: "y"
df = df.withColumn("second_item", element_at("letters", 2))

# 2. Get element using 0-based index (like Python)
# Input: ["x", "y", "z"], index = 1
# Output: "y"
df = df.withColumn("second_item_alt", col("letters").getItem(1))
```

Modifying Array Elements

```
from pyspark.sql.functions import array_remove, array_distinct, array_union

# 1. Remove a specific element from an array
# Input: [1, 2, 2, 3], remove 2
# Output: [1, 3]
df = df.withColumn("no_twos", array_remove("numbers", 2))

# 2. Remove duplicate elements from array
# Input: [1, 2, 2, 3]
# Output: [1, 2, 3]
df = df.withColumn("unique_numbers", array_distinct("numbers"))

# 3. Merge two arrays and remove duplicates
# Input: [1, 2], [2, 3]
# Output: [1, 2, 3]
df = df.withColumn("merged_array", array_union("arr1", "arr2"))
```



Arrays to Rows

```
from pyspark.sql.functions import explode, posexplode

# 1. Convert array elements into multiple rows
# Input: ["apple", "banana"]
# Output: 2 rows: "apple", "banana"
df = df.withColumn("fruit", explode("fruits"))

# 2. Explode with position (index)
# Input: ["apple", "banana"]
# Output: (0, "apple"), (1, "banana")
df = df.select("id", posexplode("fruits").alias("pos", "fruit"))
```

Rows to Array

```
from pyspark.sql.functions import collect_list, collect_set

# Input: Multiple rows with same category, e.g. {"cat": "A", "val": 1},
# {"cat": "A", "val": 2}
# Output: {"cat": "A", "val_array": [1, 2]}

# 1. Group rows into array (with duplicates)
df_grouped = df.groupBy("cat").agg(collect_list("val").alias("val_array"))

# 2. Group rows into array (unique values only)
df_grouped = df.groupBy("cat").agg(collect_set("val").alias("unique_vals"))
```

Running SQL Queries

With Temp View

```
# Create a temporary SQL table from a DataFrame
df.createOrReplaceTempView("employees")

# Select all columns
df_sql = spark.sql("SELECT * FROM employees")
```



```

# Select specific columns
df_sql = spark.sql("SELECT name, salary FROM employees")

# Filter data
df_sql = spark.sql("SELECT * FROM employees WHERE salary > 50000")

# Aggregations
df_sql = spark.sql("SELECT department, AVG(salary) AS avg_salary FROM
employees GROUP BY department")

# Sorting
df_sql = spark.sql("SELECT * FROM employees ORDER BY salary DESC")

# Using LIMIT
df_sql = spark.sql("SELECT * FROM employees LIMIT 10")

# Using CASE WHEN
df_sql = spark.sql("""
    SELECT name, salary,
    CASE
        WHEN salary > 50000 THEN 'High'
        ELSE 'Low'
    END AS salary_category
    FROM employees
""")

```

Without Temp View

```

# Load any dataframe
df = spark.read.format('csv').option('header',
'true').load('/samples/customers.csv')
# Use Spark SQL with a variable and pass the dataframe
spark.sql("select first_name from {customers_df}",customers_df = df).show()

# Load any dataframe
df2 = spark.read.format('csv').option('header',
'true').load('/samples/orders.csv')
# Use Spark SQL with a variable and pass the dataframe
spark.sql("select order_id from {orders_df}",orders_df = df2).show()

```

