How to Add a Random Number Column to Your PySpark DataFrame

Authored by

stats writer

February 4, 2026

PySpark is an exceptionally powerful framework built upon Apache Spark, designed specifically for large-scale data processing and analysis, particularly leveraging the familiar Python environment. One of the fundamental capabilities essential for advanced data manipulation tasks is the ability to easily augment a DataFrame by generating new columns. A frequently encountered requirement across various analytical workflows involves creating a new column populated with random numbers. This seemingly simple operation is crucial for specialized applications such as controlled data sampling, rigorous A/B testing setup, advanced feature engineering for machine learning models, and the construction of high-fidelity synthetic datasets for testing privacy-sensitive algorithms.

The true efficiency of PySpark shines through its highly optimized built-in functions, which allow users to perform these complex statistical operations across massive, distributed datasets with concise syntax. By harnessing the power of functions like `rand()` and `withColumn()` from the `pyspark.sql.functions` module, data scientists can generate various types of random distributions--from uniform decimal values to discrete integers--and seamlessly integrate them into their data structures. This seamless integration ensures that operations remain efficient even when scaling up to petabytes of data, reinforcing PySpark's status as a leading tool for handling big data challenges. This detailed guide explores the precise methods required to implement random number generation within a PySpark DataFrame, covering both floating-point and integer outputs, and emphasizes the necessary considerations for achieving reproducible results in a distributed environment.

PySpark: Create New Column with Random Numbers

Understanding the Mechanism of Randomness in PySpark

When working with distributed systems like Apache Spark, the concept of generating random numbers requires careful consideration. Unlike sequential environments, where randomness often relies on a single global state, PySpark must ensure that random values can be generated across multiple partitions and worker nodes efficiently and, if necessary, reproducibly. The cornerstone of this functionality is the `rand()` function, which is designed to produce a pseudo-random floating-point value. Furthermore, the `withColumn()` transformation is the primary method used to append this calculated random series as a new column to the existing DataFrame, allowing for non-destructive modification of the data structure.

The `rand()` function, by default, generates a value uniformly distributed between 0 (inclusive) and 1 (exclusive). This base function serves as the foundation, which can then be scaled, transformed, and rounded to meet specific requirements for range and data type. We will focus on two principal methods: generating continuous random decimal numbers, which is ideal for statistical simulation or proportional sampling, and generating discrete random integers, which are often preferred for

assigning random categories or indices. Both methods utilize the power of Spark SQL functions combined with DataFrame transformations to deliver efficient results regardless of dataset size.

Method 1: Creating a New Column with Random Decimal Numbers

To generate a column containing random decimal numbers (floats) within a specified range, the core approach involves invoking the `rand()` function and multiplying its output by the desired maximum value. Since `rand()` produces values between 0 and 1, multiplying it by a scalar, say 100, effectively scales the output range to 0 and 100. This method is mathematically straightforward and highly performance-optimized within the PySpark engine. It is the preferred technique when high precision and continuous distribution are necessary for the analytical task, such as simulating probabilistic outcomes or adding noise to continuous features.

The syntax is clean and relies solely on importing `rand` and chaining the `withColumn` operation directly onto the DataFrame object. The resulting column will maintain the `DoubleType` precision inherent to the float generation process.

Method 1: Create New Column with Random Decimal Numbers

from pyspark.sql.functions import rand

#create new column named 'rand' that contains random floats between 0 and 100
df.withColumn('rand', rand(seed=23)*100).show()

Method 2: Creating a New Column with Random Integers

Generating random integers requires an additional step beyond simple scaling. After utilizing the `rand()` function and scaling the result to the desired range (e.g., multiplying by 100 for a range up to 100), the resulting decimal number must be converted into a discrete integer. The most common and direct way to achieve this in PySpark is by using the `round()` function, also available within `pyspark.sql.functions`. The `round()` function truncates the decimal component, ensuring that the final output is a whole number, fulfilling the requirement for discrete random values.

When applying `round()`, it is crucial to specify the rounding precision. By setting the precision to `0`, we instruct PySpark to round the value to the nearest integer, effectively achieving the desired integer output. This combined approach--scaling the random float and then rounding it--ensures that the distribution remains uniform across the integer range while maintaining the efficiency of the PySpark framework.

Method 2: Create New Column with Random Integers

from pyspark.sql.functions import rand, round

#create new column named 'rand' that contains random integers between 0 and 100
df.withColumn('rand', round(rand(seed=23)*100, 0)).show()

Prerequisites and Initializing the PySpark Environment

Before applying either of the methods described above, a working SparkSession must be initialized, and a target DataFrame must be defined. The SparkSession acts as the entry point for using Spark functionality, allowing Python code to interface with the distributed engine. For demonstration purposes, we will create a small, representative DataFrame manually. This initial setup is fundamental to any PySpark workflow, ensuring that the computational environment is ready and the data structure exists for modification.

The sample data defined below represents a simple dataset containing NBA team names and their respective point totals. This structure will serve as our base for demonstrating how the `rand` column is appended without altering the original data. The use of `createDataFrame` is the standard way to convert local data structures (like Python lists of lists) into a distributed Spark DataFrame for processing.

The following examples show how to use each method in practice with the following PySpark DataFrame:

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()

+-------+------+
| team|points|
+-------+------+
| Mavs| 18|
| Nets| 33|
| Lakers| 12|
| Kings| 15|
| Hawks| 19|
|Wizards| 24|
| Magic| 28|
| Jazz| 40|
|Thunder| 24|
| Spurs| 13|
+-------+------+
```

Example 1: Generating and Analyzing Random Decimal Numbers

The first practical example focuses on implementing Method 1: generating random floating-point values between 0 and 100. This is achieved by multiplying the output of `rand(seed=23)` by 100. The use of the `withColumn()` function is fundamental here; it takes the name of the new column (`rand`) and the SQL expression defining its contents. It is important to remember that `withColumn()` returns a new DataFrame, reflecting the immutable nature of Spark DataFrames, meaning the original `df` remains untouched unless the result is explicitly assigned back to `df`.

Observing the output reveals that the new `rand` column contains high-precision decimal numbers. Since we specified a multiplication factor of 100, all generated values fall within the range [0, 100). This precision is ideal for tasks requiring fine-grained distribution, such as Monte Carlo simulations or assigning weights based on continuous random variables.

Example 1: Create New Column with Random Decimal Numbers

We use the following syntax to add a new column to the DataFrame named **rand** that contains random decimal numbers between 0 and 100:

from pyspark.sql.functions import rand

#create new column named 'rand' that contains random floats between 0 and 100
df.withColumn('rand', rand(seed=23)*100).show()

```
+-------+------+-----------------+
| team|points| rand|
+-------+------+-----------------+
| Mavs| 18| 93.88044512577216|
| Nets| 33|39.432553969527554|
| Lakers| 12|23.260361399084918|
| Kings| 15| 2.339183228862929|
| Hawks| 19| 82.53753350983487|
|Wizards| 24| 88.94415403143505|
| Magic| 28| 80.81524027081029|
| Jazz| 40| 59.56629641640896|
|Thunder| 24| 27.62195585886885|
| Spurs| 13| 70.43214981152886|
+-------+------+-----------------+
```

Notice that the new **rand** column contains random decimal numbers, or floating-point values, distributed uniformly between 0 and 100. The precision displayed reflects the underlying `DoubleType` used by PySpark for numerical calculations. Understanding the range and data type is critical when incorporating this column into downstream operations, such as mathematical models or database storage.

Example 2: Generating and Analyzing Random Integers

The second example demonstrates Method 2, which involves generating discrete integers between 0 and 100. This requires combining the scaling step with the rounding function. By wrapping the scaled `rand()` output within `round(..., 0)`, we guarantee that the final column contains whole numbers. This is particularly useful when random assignment must be categorical or when the target system (e.g., an internal sampling mechanism) strictly requires integer inputs.

Upon execution, we observe that the `rand` column now contains integers (displayed with a `.0` suffix, indicating they are numerical types that have been rounded to zero decimal places). For instance, the original floating-point value 93.88... is correctly transformed into 94.0. This confirms that the combination of scaling and rounding successfully produces the desired discrete random variable within the specified range.

Example 2: Create New Column with Random Integers

We use the following syntax to add a new column to the DataFrame named **rand** that contains random integers between 0 and 100:

from pyspark.sql.functions import rand, round

#create new column named 'rand' that contains random integers between 0 and 100
df.withColumn('rand', round(rand(seed=23)*100, 0)).show()

```
+-------+------+----+
| team|points|rand|
+-------+------+----+
| Mavs| 18|94.0|
| Nets| 33|39.0|
| Lakers| 12|23.0|
| Kings| 15| 2.0|
| Hawks| 19|83.0|
|Wizards| 24|89.0|
| Magic| 28|81.0|
| Jazz| 40|60.0|
|Thunder| 24|28.0|
| Spurs| 13|70.0|
+-------+------+----+
```

Notice that the new **rand** column now contains random integers between 0 and 100. If an absolute integer type (without the `.0` suffix) is strictly required for compatibility with certain systems, an explicit type casting operation (e.g., using `.cast("int")`) should be applied after the rounding step. However, for most analytical purposes within PySpark, the rounded numerical type suffices.

Controlling Randomness: The Importance of the Seed Parameter

A critical aspect of generating random data in any computational environment, especially in highly parallelized systems like Spark, is managing reproducibility. The `rand()` function accepts an optional parameter called **seed**. A seed is a starting value used by the pseudorandom number generator (PRNG) algorithm to produce a sequence of numbers. By default, if no seed is provided, PySpark uses a non-deterministic seed (often related to the current system time), meaning every execution will yield a completely different set of random numbers.

However, by explicitly setting a value for the **seed** parameter within the `rand()` function (as seen

in our examples, `seed=23`), we guarantee that the same sequence of random numbers will be generated every time the code is executed. This is indispensable for quality assurance, debugging, and ensuring that experiments are repeatable. In a distributed computing context, the seed ensures that the generation logic is consistent across all partitions and worker nodes, preventing inconsistencies that could arise from non-deterministic initial states.

Note #1: By specifying a value for **seed** within the **rand()** function, we will be able to generate the same random numbers each time we run the code. Maintaining a consistent seed is a best practice for reliable data science pipelines, ensuring that models trained using random sampling or initialized with random weights can be recreated identically.

Scaling the Output Range

The default output range for the `rand()` function is [0, 1), meaning it produces numbers greater than or equal to zero and strictly less than one. To scale this output to any arbitrary range [0, N), we simply multiply the output of `rand()` by N. In both of our examples, we multiplied the function output by **100**, thus setting the maximum possible output value just below 100.

If the requirement is to generate random numbers within a custom range, say between A and B (where A is the minimum and B is the maximum), the formula must be slightly adjusted. The general formula for generating a uniform random number in the range [A, B) using a function that generates [0, 1) is: `A + (B - A) * rand()`. This allows for precise control over both the minimum and maximum boundaries of the generated random values, offering extensive flexibility for complex statistical modeling requirements where the base range is not zero.

Note #2: The **rand()** function returns a value between 0 and 1 by default. Thus, the number that we multiply the **rand()** function by specifies the max number that can be returned (exclusive). In this example, we set the max to be **100**.

Conclusion and Further Applications

Generating new columns populated with random numbers is a fundamental and frequently necessary step in modern data manipulation using PySpark. Whether the need is for high-precision decimal numbers for statistical simulations or discrete integers for categorical assignments, PySpark provides highly efficient, concise tools through its built-in SQL functions like `rand()` and `withColumn()`. By understanding how to control the output range through scaling and how to ensure reproducibility using the **seed** parameter, users can effectively incorporate robust randomness into their large-scale data processing workflows.

Mastery of these techniques opens the door to more advanced data analysis tasks, including creating training/testing splits (using random sampling), implementing stratified sampling logic, or

dynamically generating identifiers for data masking purposes. The ability to perform these operations quickly and reliably across massive, distributed DataFrames underscores the utility and power of the PySpark framework for big data science.

The following tutorials explain how to perform other common tasks in PySpark: