# Learning PySpark: A Step-by-Step Guide to Adding a Column with Random Numbers

Authored by

**Mohammed looti**

November 11, 2025

When engaging in large-scale data transformation and statistical modeling using PySpark, data engineers and scientists frequently encounter the need to inject controlled randomness into their datasets. This requirement is fundamental for various tasks, including creating training/testing splits, establishing robust A/B testing frameworks, or synthesizing new features for machine learning models. This comprehensive guide provides a detailed methodology for efficiently generating a new column within a DataFrame that is populated with arbitrary random values. We will thoroughly explore techniques for generating both high-precision floating-point numbers and discrete integers, primarily utilizing the powerful functions available in the `pyspark.sql.functions` module, specifically the built-in rand() function and the essential withColumn transformation. Understanding these methods is paramount for anyone managing complex, distributed data workloads.

## The Necessity of Randomness in Distributed Data Science

Introducing random elements into a dataset is not merely a superficial operation; it is a critical step in ensuring the statistical validity and robustness of analytical models. In distributed computing environments, like the one managed by Spark, generating random numbers requires specific, deterministic approaches to guarantee consistent results across numerous executors and partitions. PySpark abstracts away much of this complexity, offering specialized functions that ensure the efficient distribution and calculation of these random values throughout the cluster.

The practical applications for generating random columns are diverse and essential to the data science workflow. For instance, creating a column of uniformly distributed random numbers allows for simple stratified or systematic sampling, enabling efficient data exploration on massive datasets without needing to load the entire volume. Similarly, when preparing data for machine learning, a random column can serve as a hash key for partitioning data into training, validation, and testing sets, ensuring that the splits are unbiased and reproducible.

The core mechanism that facilitates this process is the `withColumn` operation. This standard DataFrame method is the primary tool used for adding new columns or transforming existing ones in a Spark DataFrame. When used with a random generation function, `withColumn` applies the randomization logic row-by-row across the distributed data partitions, ensuring high throughput and parallel execution. This efficiency is what makes PySpark the tool of choice for handling petabyte-scale data randomization.

## Fundamentals of Random Generation using pyspark.sql.functions

To effectively incorporate randomness into a DataFrame, we rely on the suite of functions provided within `pyspark.sql.functions`. The two primary scenarios--generating continuous decimals (floats) or discrete counts (integers)--dictate which specific functions we combine with the

`withColumn` operation. Mastering these core methods allows users to precisely control the type, range, and statistical properties of the synthetic data being generated.

The fundamental function for generating uniformly distributed random numbers is `rand()`. By default, `rand()` produces a floating-point number between 0 (inclusive) and 1 (exclusive). To adjust the range, a simple multiplication operation is applied. For example, multiplying the output of `rand()` by 100 will scale the output to fall between 0 and 100, suitable for generating continuous variables where precision is important.

Conversely, if the requirement calls for discrete, whole numbers, an additional step is necessary. We must integrate a rounding or truncation mechanism to convert the high-precision float into a clean integer. The `round()` function serves this purpose effectively, converting the continuous random variable into a discrete one. This distinction between continuous and discrete random values is essential for data integrity, especially when assigning random group IDs or performing counting simulations.

The two main approaches can be summarized as follows:

**Method 1: Continuous Random Decimal Numbers (Floats)**

This approach leverages the rand() function directly. It is utilized when high-precision, continuously distributed random values are needed, and the resulting column will typically use a double data type to store the fractional component.

**Method 2: Discrete Random Integer Numbers**

This method builds upon the first by chaining the `round()` function. This conversion step transforms the decimal output into discrete integers, making the output suitable for use cases requiring non-fractional categorization or randomization, such as assigning users to experimental groups.

## Practical Implementation: Generating Random Decimal Floats

To introduce a column containing random floating-point numbers, we must explicitly import the `rand` function. The default behavior of `rand()`--producing values between 0 and 1--is easily modified to fit any desired range. If a range between 0 and N is required, the function's output is simply multiplied by N. This scaling transformation is highly efficient as it is performed in parallel across all partitions.

A crucial parameter in any random generation process is the seed. In distributed environments, providing a fixed seed is the only way to ensure that the same sequence of random numbers is generated every time the code is executed. This feature is vital for auditing, debugging, and

guaranteeing the reproducibility of analytical results, preventing subtle variations caused by non-deterministic execution paths.

The following syntax demonstrates how to create a column of random floats scaled between 0 and 100, while ensuring deterministic output:

**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()

In this snippet, the new column, conventionally named 'rand', will be populated with decimal values uniformly scaled between 0 and 100. The specific inclusion of `seed=23` guarantees that if this transformation is run again, perhaps on a different cluster or by another user, the resulting values will be identical, upholding the principles of computational reproducibility in data engineering.

## Deriving Discrete Values: Creating Random Integers

When the requirement shifts from continuous variables to discrete whole numbers, the simple `rand()` function is insufficient on its own. To obtain integral values, we must incorporate the `round` function. This step is essential because it modifies the precision of the output, effectively truncating or rounding the generated decimal value to the nearest integer.

The typical pattern involves generating the scaled random float first and then applying `round(..., 0)`. Specifying zero decimal places (`0`) within the round function ensures the conversion results in a whole number representation. This composite approach ensures that the output is suitable for applications that require non-fractional randomization, such as cohort assignment or simulating discrete probabilities.

This technique is vital when the generated numbers must align with fixed categories or counts. For instance, generating a random integer between 1 and 5 to represent a randomly assigned group identifier. The combination of scaling and rounding guarantees that the resulting column consists solely of whole numbers, maintaining data type integrity for subsequent operations.

**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()

By chaining these functions, we achieve a highly controlled transformation. The output of this operation is a column of discrete numbers, perfectly suited for applications where continuous

variables would introduce unnecessary complexity or statistical error. Both `rand` and `round` must be imported from the `pyspark.sql.functions` module before execution.

## Setting Up the Environment and Sample Data Preparation

Before executing these random generation transformations, it is essential to establish a working PySpark environment and load a sample dataset. This foundational step involves initializing the Spark Session, which serves as the entry point for all distributed operations. For demonstration purposes, we will define a simple, structured dataset that contains fictional basketball team names and their corresponding scores.

We first initialize the `SparkSession` using the `builder.getOrCreate()` method. This ensures that either a new session is started or an existing one is reused. Following initialization, we define the sample data as a list of tuples and specify the column names. This preparation step is crucial, as PySpark operations strictly require a valid DataFrame structure to operate upon.

The creation of the DataFrame is finalized using `spark.createDataFrame(data, columns)`. This action converts the local Python data structure into a distributed Spark object, ready for parallel processing. Viewing the initial DataFrame ensures that the setup is correct before proceeding to the randomization steps.

**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|
+-------+------+
```

The resulting DataFrame `df` provides the necessary foundation for demonstrating how the `withColumn` transformations, coupled with the random generation functions, efficiently append the new randomized features to the existing structured data.

## Comprehensive Examples and Output Analysis

With the environment established, we can now execute the two distinct methods for random column generation. Analyzing the output of each example helps solidify the understanding of when to use floats versus integers in a data processing pipeline.

The first example demonstrates the creation of a continuous random column. We apply the rand() function, multiplied by 100 to set the upper boundary, and use the withColumn operation to insert the result into the DataFrame. The fixed seed value (23) guarantees that the specific floating-point values shown below will be generated every time.

```
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|
+-------+------+------------------+

As demonstrated by the output, the new **rand** column contains high-precision decimal numbers, all guaranteed to be within the desired (0, 100) range. This method is perfectly suited for generating continuous variables following a uniform distribution, such as simulating probabilities or providing randomized offsets for data features.

The second example focuses on discrete integer generation. We wrap the scaled `rand()` output in the `round(..., 0)` function. This ensures that the resulting values are truncated or rounded to the nearest whole number, satisfying requirements for discrete randomization.

**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|
+-------+------+----+

The resulting **rand** column now contains clear, discrete integers between 0 and 100. Note that PySpark often represents these integers as floats (e.g., 94.0) unless the column data type is explicitly cast to an IntegerType, but for all practical purposes, they function as whole numbers.

## Advanced Concepts: Seed Management and Performance Optimization

While the standard rand() function provides a uniform distribution, PySpark also offers more advanced functions for specialized random number generation. For instance, `randn()` is available for generating values that follow a standard normal distribution (Gaussian). The choice between `rand()` and `randn()` depends entirely on the statistical properties required by the synthetic data; a uniform distribution is often preferred for simple sampling, while a normal distribution is crucial for simulating natural phenomena or introducing noise to features.

Performance in large-scale data operations managed by Apache Spark is always a primary concern. Spark executes these random generation functions in parallel across its partitioned structure. The critical importance of maintaining the optional seed parameter lies in ensuring that the deterministic sequence is consistently maintained locally within each partition. This design allows for reproducible and highly stable results, even when handling petabytes of data distributed across a large cluster. If the seed is omitted, the system defaults to a non-deterministic sequence based on current time or executor ID, making reproducibility impossible.

For optimal performance, users should always ensure their data is well-partitioned before applying highly distributed functions like random generation. Poor partitioning can lead to data skew and unnecessary shuffling overhead, diminishing the benefits of Spark's parallel computation model. Proper optimization ensures that the randomization occurs efficiently, maximizing resource utilization across the cluster executors.

The key takeaways regarding the seed value are:

By specifying a value for **seed** within the `rand()` function, we guarantee the ability to generate the exact same sequence of random numbers each time the code is executed, which is vital for testing and consistency.

The `rand()` function inherently returns a value between 0 and 1. The multiplicative factor (e.g., **100** in our examples) explicitly defines the upper limit for the range of random values generated.

## Additional Resources

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