# How to Generate Random Numbers in SAS: A Step-by-Step Guide

Authored by

**stats writer**

December 1, 2025

Generating random numbers is a fundamental requirement across various statistical analyses, simulations, and sampling tasks within the SAS environment. Whether you are conducting Monte Carlo simulations, creating randomized treatment groups, or simply filling sample data, SAS provides robust tools for this purpose. The primary mechanism for generating random values is the powerful **RAND function**.

The **RAND function** in SAS is designed to produce high-quality random variates from a specified distribution. By default, when no arguments are supplied, it returns a continuous uniform value between 0 (inclusive) and 1 (exclusive). However, its true versatility lies in its ability to generate values based on various named probability distributions, making it an essential tool for sophisticated data manipulation within the Data Step.

Crucially, achieving reproducible results when working with randomness requires managing the underlying random number stream. In modern SAS implementations, this is handled through the **CALL STREAMINIT routine**. This routine allows the user to set a specific seed, ensuring that the sequence of generated numbers remains identical across multiple executions of the code. This article delves into the syntax and application of the RAND function, demonstrating practical examples for generating single values, lists, and multiple variables.

## Understanding Pseudorandomness and Seeding

It is important to understand that the random numbers generated by computational software like SAS are not truly random; they are **pseudorandom**. A pseudorandom number generator (PRNG) uses a deterministic algorithm to produce sequences of numbers that approximate the properties of randomness. This sequence is dependent on an initial value, known as the seed.

If the seed remains constant, the sequence of numbers generated by the PRNG will always be the same. In statistical programming, this reproducibility is often highly desirable, especially when debugging code, validating simulation results, or ensuring regulatory compliance. The standard method for setting this initial value in modern SAS programming is using the **CALL STREAMINIT routine**.

The syntax `CALL STREAMINIT(seed_value)` initializes the random number stream. If you omit `CALL STREAMINIT`, SAS defaults to a seed based on the system clock or process ID, leading to different random sequences every time the program runs. Conversely, specifying an integer seed, such as `CALL STREAMINIT(12345)`, locks the sequence, guaranteeing consistency for all subsequent RAND function calls within that Data Step execution.

## Syntax and Common Distributions of the RAND Function

The generic syntax for the **RAND function** is `RAND('distribution', parameters)`. The

distribution argument specifies the type of random variate desired, and the parameters provide the necessary constraints (e.g., mean, standard deviation, lower limit, upper limit).

While the RAND function supports numerous probability distributions--including Normal, Exponential, Beta, and Gamma--one of the most frequent uses is generating uniform or integer values within a bounded range. For generating discrete random integers, which are often required for sampling or basic simulations, the `'INTEGER'` distribution type is utilized. This type requires two parameters: the lower bound (inclusive) and the upper bound (inclusive).

For instance, `RAND('UNIFORM')` generates continuous values between 0 and 1, while `RAND('INTEGER', 1, 100)` generates whole numbers between 1 and 100. Understanding the required parameters for each distribution type is crucial for accurate simulation modeling in SAS. If the specified parameters are invalid or missing for a distribution, SAS will typically return a missing value and issue an error or warning.

## Example 1: Generating a Single Random Integer

This first example demonstrates the most straightforward application: generating a single random integer within a defined range. We aim to create a dataset containing just one observation with a random value between 1 and 10. We use `CALL STREAMINIT(1)` to fix the seed, ensuring that the resulting random number is predictable for demonstration purposes.

The distribution specified is `"integer"`, followed by the limits 1 and 10. Notice that this entire operation takes place within a single iteration of the Data Step. The subsequent `PROC PRINT` step is used simply to display the contents of the newly created dataset, `my_data`.

```
/*create dataset with variable that contain random value*/
data my_data;
call streaminit(1); /*make this example reproducible*/
x = rand("integer", 1, 10);
output;
run;

/*view dataset*/
proc print data=my_data;
```

| Obs | x |
|---|---|
| 1 | 9 |

As illustrated by the output, the single random number generated between 1 and 10 was determined to be **9**. The use of the `CALL STREAMINIT(1)` ensures that if you run this exact code again, the output variable X will consistently hold the value 9, confirming the pseudorandom nature of the process.

If true variability is desired--for instance, in operational code or production simulations--you must either remove the `CALL STREAMINIT` statement entirely or replace the static seed value with a dynamic value, such as a large integer derived from the current system clock.

## Example 2: Generating a Variable with Several Random Numbers

Often, generating a single value is insufficient; researchers typically need to simulate a population or create a large sample of random observations. This requires utilizing iterative programming structures, such as a **DO loop**, within the Data Step.

The following code demonstrates how to generate a dataset containing 10 observations, where each observation includes a variable `x` populated with a random integer between 1 and 20. We initialize the stream with `CALL STREAMINIT(10)`. The `DO i = 1 to 10` loop dictates the number of observations to be created, and the `OUTPUT` statement inside the loop ensures that a new observation is written to the dataset `my_data` during each iteration.

```
/*create dataset with variable that contain random value*/
data my_data;
call streaminit(10);
do i = 1 to 10;
x = rand("integer", 1, 20);
output;
end;
run;

/*view dataset*/
proc print data=my_data;
```

| Obs | i | x |
|---|---|---|
| 1 | 1 | 20 |
| 2 | 2 | 4 |
| 3 | 3 | 7 |
| 4 | 4 | 16 |
| 5 | 5 | 19 |
| 6 | 6 | 20 |
| 7 | 7 | 4 |
| 8 | 8 | 8 |
| 9 | 9 | 19 |
| 10 | 10 | 19 |

The resulting dataset successfully contains 10 observations (indexed by the implicit variable `_N_` or the loop counter `i`). Each value in the variable `x` is a distinct random number falling between the specified bounds of 1 and 20. Importantly, when the **RAND function** is called repeatedly, it draws the next value sequentially from the initialized stream.

## Example 3: Generating Multiple Variables with Several Random Numbers

It is common to require multiple independent variables, each generated using different random number distributions or range specifications. Because the **RAND function** draws the next number from the ongoing stream every time it is called, we can assign different random values to different variables within the same loop iteration.

In this advanced application, we generate 10 observations, just as in Example 2, but we define two variables: `x`, ranging from 1 to 20, and `y`, ranging from 50 to 100. Both variables use the `"integer"` distribution type, but with distinctly separate bounds. The sequence of calls within the loop ensures that `x` and `y` receive sequentially generated (but statistically independent) values from the pseudorandom stream.

```
/*create dataset with variable that contain random value*/
data my_data;
call streaminit(10);
do i = 1 to 10;
x = rand("integer", 1, 20);
y = rand("integer", 50, 100);
output;
end;
```

```
run;

/*view dataset*/
proc print data=my_data;
```

| Obs | i | x | y |
|---|---|---|---|
| 1 | 1 | 20 | 59 |
| 2 | 2 | 7 | 88 |
| 3 | 3 | 19 | 98 |
| 4 | 4 | 4 | 68 |
| 5 | 5 | 19 | 97 |
| 6 | 6 | 15 | 56 |
| 7 | 7 | 19 | 98 |
| 8 | 8 | 18 | 64 |
| 9 | 9 | 14 | 76 |
| 10 | 10 | 16 | 67 |

The final output confirms that the **x** variable contains 10 unique random numbers strictly bounded between 1 and 20, while the **y** variable contains 10 unique random numbers bounded between 50 and 100. This demonstrates the flexibility of using the **RAND function** multiple times within a single iteration to build complex simulation datasets.

## Choosing the Correct Random Number Distribution

While the examples above focused on generating random integers, the true power of the **RAND function** lies in its support for various statistical distributions. The choice of distribution is critical and depends entirely on the nature of the data you are trying to model or simulate.

For simulations involving characteristics that tend to cluster around a mean, such as height, weight, or test scores, the `RAND('NORMAL', mean, std_dev)` distribution is appropriate. If you are simulating event times or waiting periods, the `RAND('EXPONENTIAL', parameter)` function might be required. For generating uniform continuous values (not integers), simply using `RAND('UNIFORM')` is sufficient.

It is paramount that the parameters supplied to the function match the requirements of the specified distribution. Misunderstanding the input parameters, such as supplying a negative standard deviation to the 'NORMAL' distribution, will result in invalid values or errors during the

Data Step execution. Always consult the official SAS documentation for a comprehensive list of distributions and their required parameter inputs.

## Advanced Seed Management and Practical Considerations

For large-scale simulations or distributed computing environments, managing the random number stream efficiently goes beyond simply using `CALL STREAMINIT` once. When running parallel processes or splitting a simulation across multiple SAS sessions, researchers must ensure that these separate processes draw from independent segments of the overall random sequence to maintain statistical validity.

If two separate simulation runs are meant to be entirely independent, they must be initiated using different seed values via `CALL STREAMINIT`. Using the same seed will replicate the results, which is excellent for testing but detrimental for generating unique statistical samples.

Furthermore, while the **RAND function** is generally considered robust, users should be aware of the underlying generator algorithm used by their specific SAS version. Older versions of SAS might rely on different PRNGs, which could affect the quality or cycle length of the generated sequences, particularly in highly demanding computational tasks involving billions of generated random numbers.

## Summary of Key SAS Random Number Techniques

Mastering the generation of random numbers in SAS relies on the effective combination of the **RAND function** and proper initialization of the random stream using `CALL STREAMINIT`. These tools enable statisticians and analysts to perform complex modeling and data preparation tasks with high precision and full control over reproducibility.

The core techniques demonstrated include:

Using `CALL STREAMINIT(seed)` to guarantee consistent results across different runs.

Specifying the distribution type (e.g., `"integer"`, `"normal"`, `"uniform"`) as the first argument to the **RAND function**.

Leveraging **DO loops** and the `OUTPUT` statement within the Data Step to efficiently create large datasets of random values.

Calling the **RAND function** multiple times within a single observation loop to generate statistically independent variables with different distributions or bounds.

By applying these methods, SAS users can confidently integrate stochastic elements into their

programs, whether for basic data masking or for advanced Monte Carlo simulations.