How to Easily Sample Rows with Replacement in Pandas

Authored by

stats writer

November 20, 2025

In the realm of data science and statistical analysis, the ability to accurately sample data is fundamental. When working with the powerful Pandas library in Python, users frequently encounter the need to select subsets of data, either for model training, testing, or statistical procedures like bootstrapping. A crucial distinction in this process is whether the sampling is performed with or without replacement.

The core mechanism for achieving this in Pandas is the `DataFrame.sample()` function. This versatile method provides granular control over the sampling process, allowing developers and analysts to specify the exact number or proportion of rows to retrieve, and crucially, whether the selection of an item prevents its subsequent reselection (sampling without replacement) or permits it (sampling with replacement). Understanding how to correctly leverage the `replace` parameter is essential for ensuring statistical validity in your data operations.

This comprehensive guide will detail the exact syntax and statistical implications of performing sampling with replacement using the `sample()` function in Pandas, providing practical examples and discussing related parameters that enhance reproducibility and accuracy, such as `n`, `frac`, and `random_state`.

Enabling Sampling With Replacement in Pandas

Sampling with replacement means that once an item (in this case, a row) is selected from the dataset, it is effectively returned to the pool of potential selections before the next pick occurs. This allows the same row to appear multiple times in the resulting sample, a necessary feature for many statistical simulations. To enable this behavior within the Pandas environment, one must explicitly set the `replace` argument to `True` when calling the sampling method.

The `sample()` function, when applied to a DataFrame, defaults to sampling without replacement (`replace=False`). Therefore, changing this single parameter is the key to unlocking repeated row selections. This approach ensures that the statistical integrity of methods like bootstrapping, where resamples must be of the same size as the original dataset and allow for repeats, is maintained within your Python workflow.

Below illustrates the foundational syntax for randomly selecting a specific number of rows, denoted by `n`, while ensuring that repeats are permissible in the resulting subset. This is often the first step when constructing simulated datasets or running Monte Carlo methods where replacement is a mandatory condition.

#randomly select *n* rows with repeats allowed
df.sample(n=5, replace=True)

Practical Demonstration: Setting up the Sample Dataset

To provide a clear, executable demonstration of sampling with and without replacement, we will first establish a reference DataFrame. This dataset, representing basketball player statistics, is concise yet sufficient to highlight the difference in output produced by the `replace` parameter. Creating this initial structure is a prerequisite for any reproducible data operation in Pandas.

The DataFrame contains eight rows, each corresponding to a different team (A through H), along with metrics like points, assists, and rebounds. When we apply the sampling function, we will be selecting rows from this indexed list of players. Pay close attention to the index numbers (0 through 7) as they appear in the subsequent sampled outputs, as this explicitly shows which original rows were selected and how often.

Here is the necessary Python code to initialize the dataset, followed by the output showing the structure of our sample data. This setup ensures that all following examples are grounded in a common, easily understandable context, maximizing the clarity of the subsequent sampling results.

import pandas as pd

#create DataFrame
df = pd.DataFrame({'team': ,
'points': ,
'assists': ,
'rebounds': })

#view DataFrame
print(df)

team points assists rebounds
0 A 18 5 11
1 B 22 7 8
2 C 19 7 10
3 D 14 9 6
4 E 14 12 6
5 F 11 9 5
6 G 20 9 9
7 H 28 4 12

Contrasting Default Behavior: Sampling Without Replacement

Before implementing sampling with replacement, it is instructive to observe the default behavior of

the `sample()` function, which operates without replacement (`replace=False`). When sampling without replacement, once a row is selected, it cannot be chosen again in the same operation. This ensures that every row in the resulting sample is unique and represents a distinct entity from the original population, a technique commonly used for splitting data into training and testing sets.

For this demonstration, we instruct Pandas to randomly select six rows (`n=6`) from our eight-row DataFrame. To guarantee that the results are consistent every time this code is executed--a practice essential for debugging and peer review--we include the `random_state` argument. This argument serves as a seed for the random number generator, fixing the sequence of random selections.

Upon reviewing the output, it is clear that six distinct rows have been chosen (indices 6, 2, 1, 7, 3, 0). Importantly, no index appears more than once, confirming the absence of replacement. This example establishes the baseline behavior against which we will compare the core subject of this article: sampling with replacement.

#randomly select 6 rows from DataFrame (without replacement)
df.sample(n=6, random_state=0)

team points assists rebounds
6 G 20 9 9
2 C 19 7 10
1 B 22 7 8
7 H 28 4 12
3 D 14 9 6
0 A 18 5 11

Executing Sampling With Replacement using 'n'

The primary method for conducting sampling with replacement involves utilizing the `replace=True` parameter alongside the `n` parameter, which specifies the absolute number of samples desired. Because replacement is enabled, the value of `n` can actually be larger than the total number of rows in the source DataFrame, although in most statistical scenarios, such as bootstrapping, `n` is typically set equal to the size of the original dataset.

In the following code block, we execute the `sample()` function, requesting six rows while setting `replace=True` and fixing the random sequence using `random_state=0`. The comparison with the previous example is crucial: observe how the inclusion of the replacement mechanism alters the resulting data subset.

The output clearly shows that index 3, corresponding to team "D", appears twice in the sample.

This repetition is the defining characteristic of sampling with replacement. This functionality is indispensable for statistical resampling techniques where the independence of selections must be maintained, allowing for the generation of numerous simulated samples to estimate population parameters and confidence intervals.

#randomly select 6 rows from DataFrame (with replacement)
df.sample(n=6, replace=True, random_state=0)

team points assists rebounds
4 E 14 12 6
7 H 28 4 12
5 F 11 9 5
0 A 18 5 11
3 D 14 9 6
3 D 14 9 6

Controlling Randomness for Reproducibility with random_state

A critical component of any analysis involving random sampling is ensuring that the results are fully reproducible. If the sampling process is truly random, running the same code multiple times will yield different results, which can complicate debugging, testing, and validation. The `random_state` parameter within the Pandas `sample()` function solves this by controlling the initialization of the underlying pseudo-random number generator.

Setting `random_state` to a fixed integer (e.g., 0, 42, or any consistent number) means that the sequence of random numbers generated will always be the same for that operation. While the selection process remains statistically random for a single run, fixing the state allows you and others to recreate the exact sample dataset anytime the code is executed. This is not just a convenience; it is a standard scientific practice that enhances the trustworthiness and verifiability of data-driven conclusions.

When performing sampling with replacement, where the probability of selecting any given row remains constant across selections, maintaining control over the random sequence is even more vital. If `random_state` were omitted, the specific rows that are repeated would change every time the script runs, making it nearly impossible to trace errors or confirm expected statistical distributions. Always use `random_state` when reporting results that depend on randomized sampling.

Using the 'frac' Argument for Fractional Sampling

While the `n` argument allows users to specify an absolute number of rows to sample, the `frac` argument offers an alternative: selecting a random fraction, or proportion, of the total DataFrame size. This is particularly useful when working with large datasets where the total number of rows (N) may vary, but you always need a consistent percentage of data for your sample, such as 10% or 50%.

To use fractional sampling with replacement, one must still ensure that `replace=True` is set. For instance, if a DataFrame has 800 rows and `frac=0.10` is specified, the resulting sample will contain approximately 80 rows, calculated as the floor of N * frac. Note that when the result of N * frac is not an integer, Pandas truncates the result, selecting the nearest whole number of rows.

The following example demonstrates selecting 75% of the total rows (which is 6 rows out of 8) using `frac=0.75`. Even though we are specifying a proportion, the crucial effect of `replace=True` is still evident, as the resulting sample size is fixed, and certain rows are allowed to be drawn multiple times, confirming its status as a sample taken with replacement.

#randomly select 75% of rows (with replacement)
df.sample(frac=0.75, replace=True, random_state=0)

team points assists rebounds
4 E 14 12 6
7 H 28 4 12
5 F 11 9 5
0 A 18 5 11
3 D 14 9 6
3 D 14 9 6

Statistical Applications of Sampling with Replacement

The utility of sampling with replacement extends far beyond simple subset selection. It is the cornerstone of several highly important statistical methodologies used widely in machine learning and econometrics. The most prominent application is bootstrapping, a resampling technique used to estimate the sampling distribution of an estimator (e.g., mean, median, standard deviation) by repeatedly drawing samples with replacement from the observed data.

By generating hundreds or thousands of these resamples, analysts can create an empirical distribution of a statistic, which allows for robust calculation of standard errors, confidence intervals, and bias corrections, especially when theoretical assumptions about the underlying population distribution are difficult to meet or verify. Pandas' `sample(replace=True)` function simplifies the execution of the crucial data-drawing step required for bootstrapping.

Another related application is in ensemble methods in machine learning, such as Random Forests, where Bagging (Bootstrap Aggregating) is used. Each model in the ensemble is trained on a distinct, randomly drawn subset of the training data generated using sampling with replacement. This technique diversifies the training data for each base estimator, significantly reducing variance and improving the overall stability and generalization performance of the final model.

Advanced Sampling: Incorporating Weights

While the examples above assume that every row has an equal probability of being selected, the Pandas `sample()` function also supports non-uniform probability distributions via the `weights` parameter. This advanced feature allows the user to assign a specific likelihood of selection to each row, which is then respected during the random draw, regardless of whether replacement is used.

When conducting sampling with replacement in scenarios involving stratification or imbalance, using weights is critical. For instance, if a dataset contains two classes (A and B), and class A is ten times more prevalent than class B, you might assign a higher weight to the rows belonging to class B to ensure they are adequately represented in the sample. The weights should correspond to the index of the DataFrame and can be passed as a list, array, or Series.

The combination of `replace=True` and a custom `weights` vector provides ultimate control over the sampling distribution, moving beyond simple uniform selection to allow for complex, statistically rigorous resampling methodologies tailored to the specific characteristics and research requirements of the underlying data.