How to Generate Random Samples in R Using the `sample()` Function

Authored by

stats writer

December 31, 2025

The **sample function in R** is a cornerstone tool for any R programmer engaging in statistical analysis or simulation. It is specifically designed to efficiently generate a random sample from an existing population of objects. This population can be represented by a simple vector, an array, or a complex data frame.

When executing the function, the user provides the source dataset and specifies the desired sample size using the `size` argument. The output of the **sample()** function mirrors the structure of the input, returning a vector or data frame that contains the selected subset of elements. Crucially, the generated sample constitutes a non-biased, random selection of objects or rows from the initial set, providing a robust foundation for hypothesis testing and model validation.

Understanding the sample() Function in R

The **sample()** function is fundamental for tasks requiring statistical inference, where examining the entire dataset is often impractical or unnecessary. It provides the mechanism to draw a subset of elements from a larger dataset or a vector, managing the selection process either with or without replacement. This control over replacement is vital, as it determines whether an observation can be selected multiple times, impacting the statistical properties of the resulting sample.

The elegance of the **sample()** function lies in its simplicity and flexibility. By automating the process of random selection, it ensures that every element in the source population has a defined chance of being included in the resulting subset. This capability is paramount in simulations, bootstrap methods, and general exploratory data analysis where partitioning data is a common necessity.

Before diving into practical examples, it is essential to internalize the structure and parameters that govern its operation. Understanding these arguments allows practitioners to tailor the sampling process precisely to their statistical requirements, whether they involve stratified sampling or simple random selection.

Syntax and Key Arguments of sample()

To utilize the sampling capabilities effectively, one must understand the arguments passed to the function. The basic structure of the **sample()** function is straightforward, yet each argument plays a critical role in defining the nature of the output sample. Mastery of these parameters ensures accurate and reproducible results.

The basic syntax for the **sample()** function is as follows, illustrating the four primary inputs:

sample(x, size, replace = FALSE, prob = NULL)

The specific arguments dictate the behavior of the sampling process:

x: This required argument represents the source population--the vector or dataset from which the elements will be chosen. It defines the universe of possible outcomes.
size: This argument specifies the exact number of items desired in the resulting sample. If this argument is omitted, the default behavior attempts to sample the entire population, which is usually not the intended use case.
replace: A logical argument (**TRUE** or **FALSE**) that dictates whether sampling should occur with replacement. By default, it is set to **FALSE**, meaning once an element is selected, it cannot be selected again.
prob: This optional argument takes a vector of probability weights. If provided, the length of this vector must match the length of **x**. These weights skew the randomness, making certain elements more or less likely to be selected.

*For complete and detailed technical specifications regarding the function's implementation, the official documentation for **sample()** can be found here.*

The following examples illustrate practical applications of using **sample()**, moving from simple vector operations to more complex dataset manipulation.

Practical Application 1: Sampling Elements from a Vector

The most straightforward application of **sample()** involves drawing random elements from a simple numerical or character vector. This scenario is frequently encountered when simulating random events, shuffling data, or selecting indices for cross-validation subsets. We will begin by defining a sample vector, *a*, containing ten distinct integers.

Suppose we have vector *a* containing the first 10 positive integers:

#define vector *a* with 10 elements in it
a <- c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

To generate a random sample of five elements from vector *a* without replacement (the default behavior), we only need to specify the source vector (x) and the desired size. Since `replace = FALSE` is the default setting, we can omit it from the function call, resulting in a unique set of five numbers.

#generate random sample of 5 elements from vector *a*
sample(a, 5)

3 1 4 7 5

One key characteristic of random sampling is the inherent variability in results. It is important to note that each time the **sample()** function is executed without fixing the random seed, it is highly likely that a different set of elements will be returned. This is the nature of true randomness in computational environments.

#generate another random sample of 5 elements from vector *a*
sample(a, 5)

1 8 7 4 2

Ensuring Reproducibility with set.seed()

While variability is essential for proper statistical simulation, the ability to reproduce exact results is crucial for research validation and debugging. If we need to ensure that colleagues, reviewers, or even our future selves can regenerate the identical sample, we must utilize the **set.seed()** function.

The **set.seed()** function initializes R's internal random number generator using a specified integer. By setting the seed to a fixed value (e.g., 122), all subsequent random operations, including **sample()** calls, will produce the exact same sequence of pseudo-random numbers, thereby guaranteeing replicable results.

Observe how applying **set.seed()** before the sampling operation results in the same output, even when the function is called multiple times:

#set.seed(some random number) to ensure that we get the same sample each time
set.seed(122)

#define vector *a* with 10 elements in it
a <- c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

#generate random sample of 5 elements from vector *a*
sample(a, 5)

10 9 2 1 4

#generate another random sample of 5 elements from vector *a* (results are identical due to set.seed)
sample(a, 5)

10 9 2 1 4

Using **set.seed()** is a fundamental best practice in computational statistics, ensuring the

transparency and verifiability of research findings, especially when dealing with complex data modeling or cross-validation routines where consistent data splits are mandatory.

Handling Replacement in Sampling

Sampling without replacement, as demonstrated above, is suitable when the sample size is small relative to the population, and we require unique observations. However, in certain advanced statistical techniques, such as bootstrapping or simulating continuous processes, we need to allow the same element to be selected multiple times. This is achieved by setting the `replace` argument to **TRUE**.

When `replace = TRUE` is used, the selection process is altered: after an element is chosen and included in the sample, it is "returned" to the source population, making it eligible for selection again in the subsequent draws. This is particularly necessary when the desired sample size exceeds the total number of elements in the source vector, a scenario that would otherwise cause an error if replacement were disabled.

We can modify our previous example to illustrate sampling with replacement:

#generate random sample of 5 elements from vector *a* using sampling with replacement
sample(a, 5, replace = TRUE)

10 10 2 1 6

Notice in the output above that the value 10 appears twice. This repetition is characteristic of sampling with replacement and is highly useful for generating distribution estimates via methods like the non-parametric bootstrap.

Practical Application 2: Generating Samples from DataFrames (Datasets)

Beyond simple vectors, a more common and powerful use of the **sample()** function is generating a random sample of rows from a larger data frame or dataset. This operation is fundamental for splitting data into training and testing sets in machine learning workflows, or simply examining a random subset of large observational data.

To achieve this, we first generate a random sample of row indices, and then use those indices to subset the original data frame. For this illustration, we will use the famous built-in R dataset, **iris**, which contains 150 total observations. Our goal is to select a random sample of 10 rows.

First, let's inspect the structure of the **iris** dataset:

#view first 6 rows of iris dataset
head(iris)

Sepal.Length Sepal.Width Petal.Length Petal.Width Species
#1 5.1 3.5 1.4 0.2 setosa
#2 4.9 3.0 1.4 0.2 setosa
#3 4.7 3.2 1.3 0.2 setosa
#4 4.6 3.1 1.5 0.2 setosa
#5 5.0 3.6 1.4 0.2 setosa
#6 5.4 3.9 1.7 0.4 setosa

#set seed to ensure that this example is replicable
set.seed(100)

#choose a random vector of 10 elements from all 150 rows in iris dataset (1:nrow(iris))
sample_rows <- sample(1:nrow(iris), 10)
sample_rows

47 39 82 9 69 71 117 53 78 25

#choose the 10 rows of the iris dataset that match the row numbers above
sample_data <- iris
sample_data

Sepal.Length Sepal.Width Petal.Length Petal.Width Species
#47 5.1 3.8 1.6 0.2 setosa
#39 4.4 3.0 1.3 0.2 setosa
#82 5.5 2.4 3.7 1.0 versicolor
#9 4.4 2.9 1.4 0.2 setosa
#69 6.2 2.2 4.5 1.5 versicolor
#71 5.9 3.2 4.8 1.8 versicolor
#117 6.5 3.0 5.5 1.8 virginica
#53 6.9 3.1 4.9 1.5 versicolor
#78 6.7 3.0 5.0 1.7 versicolor
#25 4.8 3.4 1.9 0.2 setosa

In this sequence, `1:nrow(iris)` creates the source population (a vector of row numbers from 1 to 150). The **sample()** function then randomly selects 10 numbers from this range. Finally, we use standard subsetting notation (`iris`) to extract the corresponding rows. Note that if you copy and paste the above code in your own R console, you should get the exact same sample since we used **set.seed(100)** to ensure replicability.

Advanced Sampling: Using Probability Weights (prob Argument)

While simple random sampling (where all elements have an equal chance of selection) is adequate for many tasks, researchers sometimes require unequal probabilities. This might occur in stratified sampling schemes or when attempting to oversample rare classes in classification problems. The `prob` argument allows us to assign custom weights to influence the selection process.

The `prob` argument accepts a vector of non-negative weights that correspond element-wise to the elements in **x**. These weights do not need to sum to one; **sample()** automatically normalizes them. If a weight for a specific element is higher, that element has a greater likelihood of being included in the resulting random sample. Conversely, a weight close to zero makes selection highly improbable.

For example, suppose we want to heavily favor the selection of the number 10 from our vector *a* (which contains 1 through 10). We can assign a weight of 5 to the last element and a weight of 1 to all others:

 # Define weights: 1 for elements 1-9, and 5 for element 10
weights <- c(1, 1, 1, 1, 1, 1, 1, 1, 1, 5)

# Sample 5 elements with replacement, using probability weights
sample(a, 5, replace = TRUE, prob = weights)

10 10 7 10 2

Notice the high frequency of 10, demonstrating the effect of the 'prob' argument.

Using the `prob` argument transforms the operation from simple random sampling to weighted random sampling, providing powerful control necessary for sophisticated statistical analysis and complex survey designs.

Conclusion: Best Practices for Using sample()

The **sample()** function is an indispensable component of the R environment, providing both simple and nuanced control over the creation of random subsets. To ensure the integrity and reliability of your results, always adhere to core best practices.

Firstly, always use **set.seed()** when performing tasks that require reproducibility, such as training machine learning models or writing research papers. Secondly, be acutely aware of the default setting `replace = FALSE`, and explicitly set it to **TRUE** only when your methodology (like bootstrapping) requires observations to be drawn multiple times. Finally, ensure that if you are

sampling rows from a data frame, you are sampling the indices (`1:nrow(data)`) and then using those indices for subsetting, rather than attempting to pass the entire data frame directly, which can lead to unexpected behavior if not handled correctly.

By mastering the arguments of **sample()**--especially `size`, `replace`, and `prob`--data scientists can confidently generate statistically sound random samples, forming the basis for reliable simulations and robust data science projects.