# Learning to Sample Data in R: A Practical Guide to the `sample()` Function

Authored by

**Mohammed looti**

November 9, 2025

## Introduction to Random Sampling in R

The ability to select a representative subset of data is fundamental in statistical analysis, machine learning, and data validation. In the powerful statistical environment of R, this crucial task is efficiently handled by the built-in `sample()` function. This function is designed to facilitate the extraction of a random sample of elements from any given data structure, such as a vector or a list, providing essential flexibility for researchers and analysts. Understanding how to correctly implement `sample()` is a cornerstone of reproducible data science, allowing for the simulation of complex processes or the straightforward creation of training and testing datasets.

The utility of `sample()` extends far beyond simple selection. It provides mechanisms to control whether the sampling process is conducted **with replacement** or **without replacement**, a statistical distinction that fundamentally alters the nature of the resulting subset. When sampling without replacement (the default), an element, once selected, cannot be chosen again. Conversely, sampling with replacement allows the same element to appear multiple times in the final sample, which is vital for techniques like bootstrapping or certain randomization protocols. Mastering these parameters is essential for ensuring that your sampling methodology aligns precisely with your analytical objectives, guaranteeing the validity of subsequent statistical inferences.

This article serves as an expert guide to leveraging the `sample()` function in R. We will meticulously break down its core syntax, explore the crucial role of its arguments, and provide detailed, practical examples ranging from selecting elements from a simple numerical vector to subsetting complex, large-scale datasets. Our goal is to equip you with the knowledge necessary to perform robust and reproducible random selection procedures seamlessly within the R environment.

## Understanding the sample() Function Syntax and Parameters

Before diving into practical code, it is imperative to grasp the formal definition and structure of the `sample()` function. The basic syntax is straightforward, yet the optional parameters offer powerful control over the sampling procedure. By understanding each argument, you can tailor the function's behavior to meet specific statistical requirements, whether you need uniform randomness or weighted probabilities.

The canonical structure of the function call is defined as follows:

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

Each component plays a distinct and critical role in the sampling process:

`x`: This is the primary input--the **vector** or data structure from which the elements will be drawn. If `x`

is an integer of length one (e.g., `sample(10, 5)`), it is interpreted as `1:x`, meaning the sampling occurs from the sequence of integers from 1 up to `x`.
**`size`**: A mandatory integer specifying the exact number of items desired in the resulting sample. If sampling without replacement (the default), `size` cannot exceed the total number of elements in `x`.
**`replace`**: A logical value (**`TRUE`** or **`FALSE`**) determining whether sampling should occur with replacement. The default setting is **`FALSE`**, ensuring that the selection is made without replacement.
**`prob`**: An optional numerical vector of probability weights. If provided, this vector must be the same length as `x`, and the values must be non-negative. It determines the likelihood of each corresponding element in `x` being selected. If `prob` is left as **`NULL`**, all elements have an equal chance of selection.

The flexibility afforded by these parameters ensures that **`sample()`** can be utilized across a wide spectrum of computational statistics tasks, from simple random subsetting to complex Monte Carlo simulations requiring specific selection probabilities. Understanding these foundational elements is the first step toward effective data manipulation in R.

## Practical Application: Sampling from a Vector

A common scenario involves drawing a sample from a simple numerical or character vector. For illustrative purposes, let us define a vector `a` containing the first ten positive integers. This serves as our population from which we will draw our random subset.

We begin by defining our population vector:

**#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 consisting of 5 elements from vector `a`, using the default method (without replacement), we only need to specify the vector `x` and the desired `size`. Notice that because we omit the `replace` argument, R automatically assumes `replace = FALSE`. This selection method guarantees that each number in the sample is unique.

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

# 3 1 4 7 5

It is crucial to recognize the inherent randomness of the **`sample()`** function. Every time this function is executed without additional controls, the underlying random number generator produces a new sequence, resulting in a potentially different set of elements. This variability is essential for

proper statistical randomization but can present challenges when replication of results is necessary. Observing a second execution confirms this behavioral pattern:

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

# 1 8 7 4 2

## Ensuring Reproducibility with set.seed()

While randomness is desirable for generating unbiased samples, the requirement for **reproducibility**--the ability to obtain the exact same results across different runs or by different users--is paramount in scientific computing and data analysis. In R, the core mechanism for controlling randomness and ensuring that identical random results are generated is the `set.seed()` function. This function initializes the random number generator (RNG) with a specific numerical seed, effectively resetting the starting point of the random sequence.

When `set.seed()` is called immediately before a function that relies on randomness (such as `sample()`), it ensures that the subsequent random operations will yield the exact same output every single time the code is executed, provided the same seed number is used. This practice is considered a standard best practice, particularly when sharing code or running simulations where consistent output is vital for verification.

To demonstrate the power of `set.seed()`, we will define a seed value of 122. Notice that by running the sampling procedure twice after setting the seed, the output remains identical, thus achieving perfect reproducibility:

**#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*
sample(a, 5)

# 10 9 2 1 4

If you were to execute this exact block of code on your own R console, you would retrieve the sequence 10, 9, 2, 1, 4 for both calls to **`sample()`**. This capability is non-negotiable for academic publications and collaboration, ensuring that statistical claims can be independently verified by peers.

## Advanced Sampling Techniques: Utilizing Replacement and Probability Weights

The default sampling method, without replacement, is suitable when selecting unique units from a population (e.g., drawing lottery numbers). However, many statistical methods, such as bootstrapping or simulating events where the source population is effectively infinite or replenished, require **sampling with replacement**. This is achieved by simply setting the `replace` argument to **`TRUE`**. When sampling with replacement, an element is returned to the population after being selected, meaning it can be chosen multiple times within the same sample.

Using our vector `a` again, setting `replace = TRUE` allows the random selection to include duplicates, which is evident in the resulting output where the number 10 appears twice:

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

# 10 10 2 1 6

Furthermore, the **`prob`** argument unlocks the potential for weighted sampling, moving beyond simple uniform random selection. Weighted sampling is critical when elements in the population have unequal importance or likelihood of selection. For instance, if you were sampling customer feedback where certain demographics were underrepresented, you might assign higher probability weights to those groups to ensure their presence in the sample. The `prob` vector must align perfectly, element-by-element, with the input vector `x`.

For example, if we wanted to make the numbers 9 and 10 twice as likely to be selected as the others, we would define a probability vector reflecting this differential weight. This allows for nuanced control over the selection process, moving from simple random selection toward stratified or disproportionate sampling techniques based on predetermined criteria.

## Applying sample() to Datasets for Subsetting

Perhaps the most practical application of **`sample()`** in data science is generating a random subset of rows (observations) from a large dataset. This technique is indispensable for tasks like creating training/testing splits for machine learning models or conducting rapid exploration on massive

datasets without needing to load all data into memory. To illustrate this, we will utilize the renowned built-in R dataset, the **iris dataset**, which contains 150 observations across five variables.

Our objective is to extract a random sample of 10 rows from the 150 total rows of the `iris` dataset. Since `sample()` operates primarily on vectors, we must first use it to generate a vector of random row indices (numbers from 1 to 150). We can determine the total number of rows using the `nrow()` function.

First, we examine the structure of the data and ensure reproducibility by setting a seed:

```
#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)
```

Next, we generate a random vector of 10 integers ranging from 1 to the total number of rows in `iris` (150). This list of integers represents the row numbers we want to extract. We then use this vector of indices within R's subsetting brackets () to pull the corresponding rows from the `iris` dataframe, creating our final sample dataset.

```
#choose a random vector of 10 elements from all 150 rows in iris dataset
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 <- iris
sample
```

```
# 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

This two-step process--generating random indices and then subsetting the dataframe--is the standard and most robust method for creating a simple random sample from any dataframe in R. Because we utilized **`set.seed(100)`**, any user executing this code will obtain the exact same subset of 10 rows, guaranteeing verifiable and consistent results across all environments.

## Conclusion and Best Practices for Sampling in R

The **`sample()`** function is an incredibly versatile and powerful tool within the R programming language, essential for tasks ranging from basic simulations to complex machine learning data preparation. Its straightforward syntax belies its statistical power, offering precise control over the randomization process via the `size`, `replace`, and `prob` arguments. Whether you are generating a training dataset or performing a statistical simulation, the ability to draw a valid and unbiased random sample is non-negotiable for sound data analysis.

To ensure optimal use and professional quality in your R scripts, a few best practices should always be observed:

**Always Use `set.seed()`:** Unless true non-reproducible randomness is absolutely required (which is rare), always invoke **`set.seed()`** before any sampling operation. Document the seed value used to guarantee that your results can be verified by others.
**Check the `replace` Argument:** Be explicit about whether you need sampling with or without replacement. The default (`FALSE`) is suitable for selecting unique observations, while setting it to `TRUE` is necessary for methods like bootstrapping or drawing multiple instances of the same element.
**Understand Weighted Sampling:** If your population contains elements that should not be equally likely to be chosen, utilize the **`prob`** argument carefully. Ensure the probability vector sums to 1 (or at least contains non-negative numbers proportional to the desired weights) and matches the length of the input vector `x`.

By integrating these principles and fully leveraging the capabilities of the **`sample()`** function, you

can ensure that your data workflows in R are statistically sound, efficient, and, most importantly, completely reproducible.