How to Include the Endpoint in Your Numpy arange Sequence

Authored by

stats writer

November 20, 2025

The Numpy library is foundational for numerical computing in Python, providing powerful tools for handling arrays and complex mathematical operations efficiently. Among its most frequently used utilities is the `numpy.arange` function, which is designed to generate a sequence of numbers within a defined interval, based on specified start, stop, and step values.

A central design feature of `arange` is its half-open interval representation, denoted mathematically as)

To include 50, we must adjust the `stop` parameter by adding the `step` size (5). We effectively tell `arange` to generate values up to 55 (exclusive), which guarantees that 50 is included in the iteration process:

import numpy as np

#specify start, stop, and step size
start = 0
stop = 50
step = 5

#create array, ensuring endpoint inclusion
np.arange(start, stop + step, step)

array()

This outcome confirms that by adjusting the stop boundary, we successfully forced the inclusion of the endpoint 50, making this a reliable and highly efficient method when working with integer-based sequences in Numpy.

Method 2: Leveraging `numpy.linspace` for Endpoint Inclusion

An alternative approach that avoids modifying the stop parameter is to utilize `numpy.linspace`. This function, which stands for "linear space," is fundamentally different from `arange` because it defines the array based on the `start` value, the `stop` value, and the total `num` (number of samples) required. Crucially, `linspace` includes the endpoint by default, making it perfectly suited for closed interval generation.

The primary consideration when transitioning from `arange` to `linspace` is determining the correct value for `num`. If you know the desired step size, the number of samples can be calculated using the formula: `num = ((stop - start) / step) + 1`. The addition of 1 accounts for the count of the final endpoint itself, which must be included in the total count of samples.

`Numpy.linspace` is generally preferred in scientific computing and signal processing because it is optimized for generating precise floating-point intervals, minimizing the cumulative precision errors that can sometimes affect `arange` when very small or large floating-point steps are used over a wide range. Its structure ensures that the start and end boundaries are hit exactly.

Detailed Implementation of Method 2

Using the same example (0 to 50, step of 5), we first calculate the necessary number of samples for `linspace`. Since `(50 - 0) / 5 = 10` steps, we need 10 steps plus the starting point, plus the final point, totaling 11 points. Therefore, `num` is set to 11.

The following code demonstrates how `numpy.linspace` creates the desired array, including the endpoint, without any parameter modifications beyond specifying the total count:

import numpy as np

#specify start, stop, and number of total values in sequence
start = 0
stop = 50
num = 11

#create array using linspace
np.linspace(start, stop, num)

array()

The resulting array correctly includes 50. Note the presence of the decimal points, indicating that `linspace` typically defaults to generating floating-point arrays, even when inputs are integers. This is an important distinction to consider if strict integer arrays are required for downstream processing.

Comparing `arange` and `linspace` for Array Generation

Choosing between adjusting `arange` or switching to `linspace` depends heavily on whether the fixed step size or the total number of points (`num`) is the dominant factor in your data generation requirements. Both methods effectively solve the endpoint exclusion problem, but they prioritize different aspects of array definition.

The modified **`arange`** method is superior when dealing with integer increments where the step must be guaranteed to be exact (e.g., iterating indices or counting objects). Its complexity arises primarily when using floating-point steps, where the accumulation of small arithmetic errors might

lead to an unexpected result when the boundary condition is tested.

In contrast, **`linspace`** is the definitive function when the range boundaries and the total number of data points are the absolute priority. By calculating the step size internally via division, it provides higher precision for fractional increments and ensures that the array spans the specified interval perfectly, making it the robust choice for mathematical modeling and visualization tasks.

Feature	numpy.arange (Modified)	numpy.linspace
Primary Control Input	Fixed **step size**.	Total **number of samples** (num).
Endpoint Inclusion	Achieved by adding step to stop parameter.	Included by default (`endpoint=True` is default).
Precision Risk	Potential cumulative errors with floating-point steps.	High precision, ideal for calculating intermediate points.

Summary of Best Practices

When generating numerical sequences in Numpy, the necessity of including the endpoint dictates the implementation strategy. For scenarios requiring precise integer increments and defined step size, adapting `numpy.arange` by augmenting the stop value is the preferred and most efficient technique.

For applications demanding rigorous control over the total span and point count, particularly in floating-point domains, adopting `numpy.linspace` offers superior reliability. By mastering these two distinct methods, developers gain comprehensive control over array creation, ensuring that the generated numerical sequence of numbers perfectly aligns with the analytical or computational requirements.

For complete documentation on array creation routines, refer to the official Numpy reference guides.

The following tutorials explain how to perform other common operations in Numpy: