Learning NumPy: Adding Elements to Arrays with Append

Authored by

Mohammed looti

October 29, 2025

Introduction: Essential Methods for Modifying NumPy Arrays

The NumPy library is fundamental to scientific computing in Python, primarily utilizing its powerful N-dimensional array object. While NumPy arrays are generally designed for efficient, fixed-size operations, often we need to dynamically add new elements for tasks like data preprocessing or iterative modeling. Since NumPy arrays are immutable in size once created (unlike Python lists), adding elements technically involves creating a new array.

Fortunately, NumPy provides several high-level functions, such as `np.append()` and `np.insert()`, that handle the creation and population of the new array seamlessly. Understanding when and how to use these functions is key to maintaining efficient Python code, especially when dealing with large datasets.

This comprehensive tutorial explores the four primary ways you can add one or more elements to a NumPy array. We will provide detailed explanations and practical code examples for each technique.

Overview of Array Modification Techniques

Depending on whether you need to add elements to the end of the data structure or at a specific internal location, you can choose from the following four methods:

Method 1: Append One Value to End of Array

```
#append one value to end of array
new_array = np.append(my_array, 15)
```

Method 2: Append Multiple Values to End of Array

```
#append multiple values to end of array
new_array = np.append(my_array, )
```

Method 3: Insert One Value at Specific Position in Array

```
#insert 95 into the index position 2
new_array = np.insert(my_array, 2, 95)
```

Method 4: Insert Multiple Values at Specific Position in Array

```
#insert 95 and 99 starting at index position 2 of the NumPy array
new_array = np.insert(my_array, 2, )
```

For all the following examples, we will be using the same base array, initialized below. This standard setup allows us to clearly demonstrate the effect of each modification function.

import numpy as np

#create NumPy array
my_array = np.array()

#view NumPy array
my_array

array()

Example 1: Appending a Single Value Using np.append()

The most common requirement is simply adding a new element to the end of the existing data structure. The np.append() function is specifically designed for this purpose. When dealing with one-dimensional arrays, `np.append()` takes two primary arguments: the original array and the value(s) to be appended.

It is essential to remember that `np.append()` returns a new array; it does not modify the original array in place. Therefore, you must assign the result to a new variable (here, `new_array`) to capture the modified structure. In this first example, we demonstrate how to append a single integer value, **15**, to the sequence.

The code block below illustrates the syntax and the resulting output, confirming that the new element is successfully placed at the final position.

#append one value to end of array
new_array = np.append(my_array, 15)

#view new array
new_array

array()

As shown, the value **15** has been successfully added to the end of the NumPy array, extending its length by one element.

Example 2: Appending Multiple Values Using np.append()

If you need to add several elements simultaneously, `np.append()` remains the correct tool. Instead of passing a single value as the second argument, you pass an iterable object, such as a Python list or another NumPy array, containing all the values you wish to concatenate.

In this scenario, we aim to add three new integers--**15**, **17**, and **18**--to the end of `my_array`. We encapsulate these values within a Python list . NumPy automatically handles the merging of these structures to produce the final, extended array.

This method is highly efficient for batch operations where data needs to be continuously streamed or added to a growing dataset. The resulting array will contain all the original elements followed by the new sequence, preserving the order of the appended list.

#append multiple values to end of array
new_array = np.append(my_array,)

#view new array
new_array

array()

The values **15**, **17**, and **18** have been successfully concatenated, resulting in a new array that is three elements longer than the original.

Example 3: Inserting a Single Value at a Specific Position using np.insert()

Unlike appending, which only works at the end, inserting allows you to place elements anywhere within the array structure. For this operation, we utilize the np.insert() function. This function requires three arguments: the original array, the target index position, and the value(s) to be inserted.

It is crucial to understand that array indexing in Python and NumPy is zero-based. If we specify an index position of **2**, the new element will be placed just before the element currently residing at index 2, shifting all subsequent elements to the right. In this example, we insert the value **95** at index 2.

This functionality is particularly useful when working with time-series data or structured datasets where precise ordering and placement of new data points are required. Observe how the original element at index 2 (which was 2) shifts to index 3 after the insertion.

#insert 95 into the index position 2
new_array = np.insert(my_array, 2, 95)

#view new array
new_array

array()

The value **95** has been successfully inserted into index position 2 of the NumPy array, demonstrating the precise control offered by np.insert().

Example 4: Inserting Multiple Values at a Specific Position using np.insert()

Just as with `np.append()`, `np.insert()` also supports the insertion of multiple values simultaneously. To achieve this, you provide a list or array of values as the third argument. All elements within this iterable will be inserted starting at the designated index position, maintaining their relative order.

In this final example, we insert both **95** and **99** starting at index 2. This means **95** occupies index 2, and **99** occupies index 3, shifting the rest of the original data further down the line. This is a highly efficient way to batch-insert data into a specific location without executing multiple insert calls.

While insertion methods are powerful, it's important to be mindful of performance. Since inserting into the middle of a large NumPy array requires allocating new memory and copying all subsequent elements, repeated use of `np.insert()` can be computationally expensive compared to appending, which generally involves less data manipulation.

#insert 95 and 99 starting at index position 2 of the NumPy array
new_array = np.insert(my_array, 2,)

#view new array
new_array

array()

The values **95** and **99** have been successfully inserted starting at index position 2, demonstrating the ability of np.insert() to handle multiple elements efficiently.

Conclusion and Further Resources

Adding elements to a NumPy array is a straightforward process using the highly specialized functions `np.append()` for adding data to the end, and `np.insert()` for placing data at a specific point. Although array modification technically involves creating a new array object, these functions abstract away the complexity, providing clean and readable code for data manipulation.

When choosing between these methods, consider the location and the performance implications. Appending is generally faster if order doesn't matter, while inserting offers precision but comes with a higher computational cost for large arrays.

To continue mastering data manipulation in Python, explore these related tutorials that explain how to perform other common tasks in NumPy: