Converting NumPy Matrices to Arrays: A Practical Guide with Examples

Authored by

Mohammed looti

October 28, 2025

Introduction: Bridging NumPy Matrix and Array Structures

The NumPy library is the fundamental package for scientific computing in Python, providing powerful data structures for handling large, multi-dimensional arrays and matrices. While NumPy's primary data structure is the NumPy Array (specifically, the **ndarray** object), it also historically provided the separate NumPy Matrix class. This matrix class, designed for strict two-dimensional linear algebra operations, behaves differently from the standard array, particularly regarding multiplication and shape preservation.

However, the **NumPy Matrix** class is now largely considered legacy and is discouraged in favor of using standard **ndarray** objects, which offer greater flexibility and are the cornerstone of the modern NumPy ecosystem. Consequently, developers frequently encounter the need to convert legacy matrix objects into the more versatile array format. This conversion process is essential for integrating matrix data seamlessly into workflows that rely on standard NumPy functions and tools, ensuring compatibility and leveraging the full performance potential of the library.

We will explore two highly effective methods for performing this conversion, detailing the underlying mechanisms and providing practical code demonstrations for each approach. The two primary methods available to convert a **NumPy Matrix** to an array are:

Method 1: Using the **.A1** attribute.
Method 2: Combining **np.asarray()** with the **.ravel()** function.

Method 1: Utilizing the .A1 Attribute for Direct Conversion

The simplest and most direct way to convert a **NumPy Matrix** into a flattened 1-dimensional **ndarray** is by accessing the special **.A1** attribute. This attribute is specific to the **numpy.matrix** object and is designed precisely for this flattening operation. When you access **.A1** on a matrix object, NumPy returns the underlying data structure as a new 1-D NumPy Array.

It is important to understand that the **.A1** attribute flattens the matrix, meaning that the resulting array loses the original 2-dimensional structure. All elements are arranged sequentially in a single dimension, preserving the order of elements (row-by-row). This method is highly concise and often preferred when the resulting array must be one-dimensional for further processing, such as feeding data into machine learning models or specific statistical functions.

The syntax for using this method is extremely clean and straightforward:

my_array = my_matrix.A1

While this method is concise, developers should be mindful of its behavior: **.A1** specifically targets

the conversion and flattening requirement inherent to the legacy matrix class. If you require the resulting array to maintain its 2D structure, **.A1** is not the appropriate choice; however, for achieving a quick, flattened representation, it is invaluable.

Practical Demonstration: Converting a Matrix using .A1

To illustrate this method in practice, we will first create a sample **NumPy Matrix** and then apply the **.A1** attribute to observe the immediate conversion to a 1D array. This demonstration highlights how seamlessly the underlying data is extracted and restructured into the standard **ndarray** format. We begin by importing the NumPy library and initializing a sample matrix using **np.matrix()** combined with **np.arange()** and **reshape()** functions.

The following code snippet demonstrates the creation of a 5x3 matrix and its subsequent conversion using the **.A1** attribute:

import numpy as np

# Create NumPy matrix with 3 columns and 5 rows
my_matrix = np.matrix(np.arange(15).reshape((5, 3)))

# View the original NumPy matrix
print(my_matrix)

]

# Convert matrix to a 1D array using .A1
my_array = my_matrix.A1

# View the resulting NumPy array
print(my_array)

As the output confirms, the original 5x3 **NumPy Matrix** has been successfully converted into a 1-dimensional array containing all 15 values, flattened in row-major order. To definitively verify the type of the resulting object, we can utilize the built-in **type()** function in Python, which should return **numpy.ndarray**, confirming the successful conversion from the legacy **numpy.matrix** class.

We can confirm the object type using the following command:

 # Check type of my_array
type(my_array)

numpy.ndarray

The output **numpy.ndarray** confirms that the object is indeed a standard NumPy Array, ready for general array operations. This method provides the most concise path for flattening and converting matrix objects.

Method 2: Combining asarray() and ravel()

The second approach is more idiomatic within the broader NumPy ecosystem, relying on two core functions: np.asarray() and the ravel() method. This combination is highly recommended for developers aiming for future-proof code, as it avoids relying on specialized attributes of the deprecated **numpy.matrix** class.

The process involves two distinct steps. First, we use np.asarray(). This function converts the input (in this case, the matrix) into an NumPy Array. Crucially, **np.asarray()** attempts to avoid copying the data if possible, instead returning a view of the original data when the input type already supports the array protocol. When converting a **numpy.matrix**, **np.asarray()** successfully returns a 2-dimensional **ndarray** containing the matrix data.

Second, we chain the ravel() method onto the resulting 2D array. The purpose of ravel() is to flatten the array into a 1-dimensional structure. Unlike some other flattening methods, **ravel()** generally returns a view of the original array whenever possible, which can lead to performance benefits by avoiding unnecessary memory allocation. The entire operation can be executed in a single, fluent line of code:

my_array = np.asarray(my_matrix).ravel()

Although this method requires slightly more typing than using the **.A1** attribute, it utilizes functions that are universal across all ndarray objects, making the code more transferable and readable for those familiar with standard NumPy array manipulation. Furthermore, relying on **np.asarray()** ensures that the core conversion is handled by the preferred array creation routine.

Practical Demonstration: Converting a Matrix using asarray() and ravel()

This example demonstrates the conversion using the two-step approach: converting the matrix to a 2D array first via **np.asarray()**, and then flattening it using ravel(). We will use the identical initialization step as Example 1 to ensure a fair comparison of the output results. Note that the output, in terms of data values and dimensionality, will mirror the result obtained using the **.A1** attribute.

Below is the full implementation, demonstrating the creation of the 5x3 matrix and its conversion to a 1D array using the combined functions:

import numpy as np

# Create NumPy matrix with 3 columns and 5 rows
my_matrix = np.matrix(np.arange(15).reshape((5, 3)))

# View the original NumPy matrix
print(my_matrix)

]

# Convert matrix to array using asarray() then flatten with ravel()
my_array = np.asarray(my_matrix).ravel()

# View the resulting NumPy array
print(my_array)

Just as with the previous method, the output confirms that the data has been successfully converted and flattened into a single-dimensional structure. To ensure that the resulting object is indeed a standard **ndarray**, we again check the type of the variable **my_array**.

The confirmation step using **type()** yields the expected result:

 # Check type of my_array
type(my_array)

numpy.ndarray

The result **numpy.ndarray** confirms the success of the conversion, demonstrating that both methods achieve the desired transformation from a legacy NumPy Matrix object into a standard, flattened NumPy Array.

Understanding the Differences: .A1 vs. asarray().ravel()

Although both **.A1** and **np.asarray().ravel()** produce the same flattened 1D array result, they differ slightly in their performance profile, memory handling, and relevance in modern NumPy practices. Understanding these nuances is crucial for writing efficient and maintainable code.

The primary difference lies in how they handle data copying. The **.A1** attribute is often

implemented to return a contiguous, flattened copy of the data, guaranteeing a new object independent of the original matrix data structure. Conversely, the combined np.asarray().ravel() approach often attempts to return a view (a reference to the original data memory) whenever possible. If the original data layout is already compatible with the desired flattened structure, **ravel()** returns a view, leading to zero-copy operation and improved performance, especially with very large datasets. If the data needs reordering (e.g., C-order vs. F-order), **ravel()** might return a copy.

From a stylistic and compatibility standpoint, the **np.asarray().ravel()** method is strongly preferred. Since the **numpy.matrix** class itself is considered deprecated, relying on its specialized attributes like **.A1** ties the code to legacy structures. Using universal functions like asarray() and ravel() ensures that the code remains robust and compatible with future versions of NumPy, regardless of whether the input data originates from a matrix, a list of lists, or another **ndarray**. Therefore, while **.A1** is shorter, the second method offers greater longevity and adherence to best practices in scientific Python programming.

Conclusion and Resources

Converting a legacy **NumPy Matrix** into a standard **NumPy Array** is a common step required to modernize codebases and ensure compatibility with the vast ecosystem built around the **ndarray** object. Whether you choose the quick access provided by the **.A1** attribute or the more standard, flexible approach of combining **np.asarray()** and ravel(), both methods effectively achieve the desired transformation, yielding a flattened, 1-dimensional array ready for further analysis.

We recommend adopting the **np.asarray().ravel()** method as the standard practice for new development, as it aligns better with the current direction of the NumPy library and provides greater clarity regarding data handling and potential memory optimizations through view operations. By mastering these conversion techniques, developers can ensure that their data structures are optimized for performance and compatibility within the modern Python scientific computing stack.

Additional Resources

For developers interested in further exploring data manipulation and array operations within the NumPy framework, the following tutorials explain how to perform other common tasks and deepen the understanding of array structures:

Official NumPy Documentation on The N-dimensional Array (ndarray)
Understanding Array Views vs. Copies in NumPy.
Detailed guide on using **reshape()** and flattening arrays.