How to Easily Rank Elements in a NumPy Array: A Step-by-Step Guide

Authored by

stats writer

November 22, 2025

Calculating the rank of items within a NumPy **array** is a fundamental task in data processing, essential for understanding the relative position of values. This process involves utilizing functions that return the indices of the sorted elements rather than the sorted elements themselves. The primary core tool for this is the **np.argsort()** function, which generates a new array of indices based on the values provided. By applying a clever double-sort technique using these indices, we can effectively determine the rank of every item relative to the entire dataset.

This powerful technique is highly flexible, allowing developers and data scientists to rank both **one-dimensional** and **two-dimensional** NumPy arrays. Furthermore, it supports both ascending (smallest value ranked first) and descending (largest value ranked first) ranking, although we will primarily focus on ascending examples here. While the official NumPy documentation provides comprehensive details, this guide distills the most effective methods and provides practical, hands-on examples for immediate application.

To calculate the rank of elements within a NumPy array, you generally have two highly reliable and commonly accepted approaches, depending on whether you need built-in flexibility for handling **ties** (duplicate values):

Utilizing Core NumPy or External SciPy for Ranking

Method 1: Utilize argsort() from NumPy Core Library

This method relies solely on the core **NumPy** library. It involves applying the np.argsort() function twice in sequence. This approach is highly efficient and avoids external dependencies, but it uses a predefined method for resolving ties.

```
import numpy as np

ranks = np.array(my_array).argsort().argsort()
```

Method 2: Use rankdata() from SciPy Statistics Module

For scenarios demanding statistical rigor or flexible control over tie-breaking strategies, the **rankdata()** function found within the **scipy.stats** module is the preferred solution. While it requires importing the SciPy library, it offers several different methods for assigning ranks to tied values.

```
from scipy.stats import rankdata

ranks = rankdata(my_array)
```

To illustrate the implementation of both methods and observe how they handle duplicates, we will

use the following six-element **NumPy array** in our practical examples. Notice that this array contains duplicate values (two instances of '9'), which allows us to examine how each function handles **ties**.

```
import numpy as np

#Define array of values for ranking
my_array = np.array()

#View the original array
print(my_array)
```

Example 1: Rank Items in NumPy Array Using argsort()

The **np.argsort()** function is native to the NumPy library and provides an elegant, albeit slightly counter-intuitive, method for determining ranks. The technique requires calling argsort() twice. The first call returns the indices that would sort the original array. The second call, applied to the result of the first, converts those sort indices back into the original array's index positions, effectively yielding the rank of the original element at that position.

While highly efficient, the primary limitation of this pure **NumPy** approach lies in its restricted handling of **tied values**. By default, when two or more values are identical, **np.argsort()** uses an internal stable sort, resulting in an **ordinal ranking**. This means the rank assigned to tied elements depends solely on their original position in the input array; the value that appears earlier in the array receives the lower rank, regardless of magnitude.

The following code shows how to use the **argsort()** function from NumPy to rank the items in the array, using the double-sort technique:

```
#Calculate rank of each item in array using the double argsort trick
ranks = np.array(my_array).argsort().argsort()

#View the resulting ranks
print(ranks)
```

The results show the rank of each item in the original array, using a zero-based index system where **0** represents the smallest value (1 in the original array). Notice how the two '9's, which are tied, received ranks 4 and 5, respectively, because the first '9' appeared earlier in the input sequence.

The core benefit of this strategy is its independence--you do not need to load any extra modules beyond NumPy, ensuring minimal dependencies. However, the limitation, as noted, is that **argsort()** only provides this one fixed method (ordinal ranking) for handling ties, which may not be appropriate for statistical analysis requiring mean or minimum ranks.

Example 2: Rank Items in NumPy Array Using rankdata()

The second, statistically robust method involves leveraging the rankdata() function provided by the **scipy.stats** module. This function is specifically designed for ranking numerical data and offers significant advantages when dealing with datasets containing multiple identical observations, known as **ties**.

Unlike the NumPy method, **rankdata()** adheres to standard statistical conventions by default. It typically assigns the **average rank** to tied observations. For instance, if two values are tied for the 5th and 6th positions, both will receive a rank of 5.5. Furthermore, **rankdata()** returns ranks that are one-based (starting from 1 for the smallest value), contrasting with the zero-based ranks returned by the double-argsort method.

The following implementation imports and utilizes the **rankdata()** function to process our example array. Note the differing output when compared to the **argsort()** method, particularly in how the tied values are represented:

from scipy.stats import rankdata

#Calculate rank of each item in array
ranks = rankdata(my_array)

#View the resulting ranks
print(ranks)

array()

The results show the rank of each item in the original array, with **1** representing the smallest value. Crucially, the tied values of '9' (which would occupy the 5th and 6th positions if distinct) are both assigned the average rank of 5.5.

Normalizing SciPy Ranks to Zero-Based Indexing

If consistency with **NumPy's** zero-based indexing (where the smallest value is rank 0) is required, the ranks generated by **rankdata()** can be easily shifted down by subtracting 1 from the resulting array. This normalization is useful when integrating SciPy ranks into existing NumPy workflows that

rely on zero-based indices for array manipulation.

```
from scipy.stats import rankdata

#Calculate rank of each item in array and shift to zero-based index
ranks = rankdata(my_array) - 1

#View the normalized ranks
print(ranks)
```

Controlling Tie Resolution with rankdata()

As observed, by default, the **rankdata()** function assigns **average ranks** to any values that are tied. This behavior is standard for many statistical procedures, as it maintains the sum of the ranks. However, specific applications may require alternative methods for tie resolution, especially when dealing with discrete data or when replicating results obtained from other systems.

The strength of using **SciPy's rankdata()** lies in its flexibility, provided through the optional `method` argument. This argument allows the user to specify precisely how ranks should be assigned when duplicate values are encountered. This level of control is typically unavailable when relying solely on np.argsort().

For example, we can configure **rankdata()** to replicate the exact behavior of **NumPy's argsort()** by setting the `method` argument to **'ordinal'**. This forces the function to assign ranks based on the order of appearance, ensuring the lowest rank goes to the tied value encountered first in the original array.

```
from scipy.stats import rankdata

#Calculate rank using the ordinal method for ties, and normalize to zero-based index
ranks = rankdata(my_array, method='ordinal') - 1

#View the ranks
print(ranks)
```

This implementation produces results that are mathematically identical to the double **argsort()** method demonstrated in Example 1.

Alternative Tie Handling Methods

Beyond the default 'average' and the replicated 'ordinal' method, **rankdata()** supports several other critical methods for handling ties, each suited for different analytical requirements:

min: Assigns the minimum (lowest) rank in the group to all tied values. If values are tied for 5th and 6th place, both receive rank 5.

max: Assigns the maximum (highest) rank in the group to all tied values. If values are tied for 5th and 6th place, both receive rank 6.

dense: Assigns ranks sequentially without gaps. If values are tied for 5th and 6th place, they both receive rank 5, and the next untied value receives rank 6 (skipping rank 7, unlike other methods which create gaps).

Read about each method in the SciPy documentation.

Summary of Ranking Techniques

Ranking elements within a **NumPy array** can be achieved efficiently using either the pure NumPy double-**argsort()** technique or the versatile **SciPy rankdata()** function. While the double-argsort method is lightweight and fast, `rankdata()` provides essential flexibility, especially when dealing with statistical requirements or when specific tie-breaking methodologies (such as minimum, maximum, or average rank) are mandatory.

Choosing between the two depends primarily on your dependence requirements and the necessary statistical handling of tied observations. For general-purpose, simple ranking where ordinal tie-breaking is acceptable, the NumPy method is usually sufficient. For rigorous data science and statistical applications, the SciPy method is superior due to its deep configurability.

The following tutorials explain how to perform other common tasks in NumPy, enhancing your ability to manipulate and analyze complex datasets: