Expanding and Squeezing a NumPy Array

• Expanding a NumPy array:

Expanding a NumPy array involves adding new axes (or dimensions) to the array, which essentially increases its shape. In NumPy, you can use np.expand_dims() or None (indexing method) to add an extra axis.

Method 1: np.expand_dims()

This method adds a new axis along the specified axis (dimension).

Syntax:

np.expand_dims(a, axis)

a: The input array.

axis: The position in the result shape where the new axis will be added.

Example:

```
import numpy as np

# Original 1D array

arr = np.array([1, 2, 3, 4])

print("Original array:", arr)

# Expanding the array to 2D (axis=0)

expanded_arr = np.expand_dims(arr, axis=0)

print("Expanded array (axis=0):", expanded_arr)

# Expanding the array to 2D (axis=1)

expanded_arr_2 = np.expand_dims(arr, axis=1)

print("Expanded array (axis=1):", expanded_arr_2)
```

Output:

Original array: [1 2 3 4]

Expanded array (axis=0): [[1 2 3 4]]

Expanded array (axis=1): [[1]

[2]

[3]

[4]]

Method 2: Using None or np.newaxis

You can also use None or np.newaxis to expand dimensions by adding new axes at specific locations.

Example:

```
import numpy as np

# Original 1D array

arr = np.array([1, 2, 3, 4])

print("Original array:", arr)

# Expanding the array using None (axis=0)

expanded_arr = arr[None, :]

print("Expanded array (axis=0):", expanded_arr)

# Expanding the array using None (axis=1)

expanded_arr_2 = arr[:, None]

print("Expanded array (axis=1):", expanded_arr_2)
```

Output:

Original array: [1 2 3 4]

Expanded array (axis=0): [[1 2 3 4]]

Expanded array (axis=1): [[1]

[2]

[3]

[4]]

- Squeezing a NumPy array

Squeezing a NumPy array refers to removing any dimensions with a size of 1. It helps in reducing the array's shape by eliminating unnecessary singleton dimensions. You can use the function np.squeeze() to perform this operation.

Syntax:

np.squeeze(a, axis=None)

- **a**: The input array.
- **axis**: Optional. If specified, only the dimensions with size 1 at that axis will be removed. If None (default), all singleton dimensions (size 1) will be removed.

Example: Basic Squeezing

```
import numpy as np

# Creating a 3D array with shape (1, 4, 1)

arr = np.array([[[1], [2], [3], [4]]])

print("Original array shape:", arr.shape)

print("Original array:\n", arr)

# Squeezing the array (removes the singleton dimensions)

squeezed_arr = np.squeeze(arr)

print("Squeezed array shape:", squeezed_arr.shape)

print("Squeezed array:", squeezed_arr)
```

Output

Original array shape: (1, 4, 1)

Original array:

[[[1]

[2]

[3]

[4]]]

Squeezed array shape: (4,)

Squeezed array: [1 2 3 4]

Example: Squeeze with a Specific Axis

You can also specify which dimension you want to squeeze by using the axis parameter. If the dimension at the given axis is not 1, it will raise an error.

```
import numpy as np

# Creating a 4D array with shape (1, 3, 1, 4)

arr = np.array([[[[1], [2], [3], [4]]], [[[5], [6], [7], [8]]], [[[9], [10], [11], [12]]]])

print("Original array shape:", arr.shape)

# Squeeze the dimension at axis 0 and axis 2 (remove dimensions with size 1)

squeezed_arr = np.squeeze(arr, axis=(0, 2))

print("Squeezed array shape:", squeezed_arr.shape)

print("Squeezed array:", squeezed_arr)
```

Output:

Original array shape: (1, 3, 1, 4)

Squeezed array shape: (3, 4)

Squeezed array: [[1 2 3 4]

 [5 6 7 8]

 [9 10 11 12]]

Important Notes:

- **Default behavior (axis=None)**: Removes all dimensions with size 1.
- **Specific axis**: If you want to remove a singleton dimension only along a certain axis, you can specify it with axis. If that dimension doesn't have size 1, it will raise an error.

Example where no squeeze happens

If no dimensions have size 1, np.squeeze() will return the array unchanged.

```
arr = np.array([[1, 2], [3, 4]])

print("Original array shape:", arr.shape)

squeezed_arr = np.squeeze(arr)

print("Squeezed array shape:", squeezed_arr.shape)
```

Output:

Original array shape: (2, 2)

Squeezed array shape: (2, 2)

Here, no singleton dimension was present, so the shape remains the same.

- Sorting in NumPy Arrays

Sorting in NumPy arrays refers to arranging the elements in a specified order (either ascending or descending). NumPy provides a couple of methods to perform sorting efficiently.

Methods for Sorting in NumPy:

np.sort(): Returns a sorted copy of the array.

arr.sort(): Sorts the array in-place, modifying the original array.

np.argsort(): Returns the indices that would sort an array.

np.lexsort(): Performs an indirect sort using a sequence of keys.

1. np.sort()

np.sort() returns a new array with the elements sorted in ascending order.

Syntax:

np.sort(arr, axis=-1, kind='quicksort', order=None)

arr: The input array.

axis: Axis along which to sort. By default, it sorts flattened arrays (all elements in one dimension).

kind: Sorting algorithm ('quicksort', 'mergesort', 'heapsort', or 'stable').

order: A field name or a list of field names for structured arrays.

Example:

```
import numpy as np

arr = np.array([3, 1, 4, 1, 5, 9, 2])

sorted_arr = np.sort(arr)

print("Original array:", arr)

print("Sorted array:", sorted_arr)
```

Output:

Original array: [3 1 4 1 5 9 2]

Sorted array: [1 1 2 3 4 5 9]

2. arr.sort() (In-place sorting)

arr.sort() sorts the array in-place, meaning the original array will be modified and no new array will be returned.

Example:

```
arr = np.array([3, 1, 4, 1, 5, 9, 2])

arr.sort()

print("Array after in-place sorting:", arr)
```

Output:

Array after in-place sorting: [1 1 2 3 4 5 9]

3. np.argsort()

np.argsort() returns the indices that would sort the array. It's useful if you want to know the order of elements without actually changing the array.

Syntax:

```
np.argsort(arr, axis=-1, kind='quicksort', order=None)
```

Example:

```
arr = np.array([3, 1, 4, 1, 5, 9, 2])

sorted_indices = np.argsort(arr)

print("Original array:", arr)

print("Indices that would sort the array:", sorted_indices)
```

```
print("Array sorted using indices:", arr[sorted_indices])
```

Output:

Original array: [3 1 4 1 5 9 2]

Indices that would sort the array: [1 3 6 0 2 4 5]

Array sorted using indices: [1 1 2 3 4 5 9]

4. np.lexsort()

np.lexsort() is used for sorting based on multiple keys. It returns the indices that would sort the array based on a sequence of keys. The first key has the highest priority, and subsequent keys are used in tie-breaking.

Syntax:

```
np.lexsort(keys)
```

keys: A sequence of arrays to be sorted by. These arrays should have the same length.

Example:

```
# Sorting based on two keys: first by the second element, then by the first

arr1 = np.array([1, 4, 3, 2])

arr2 = np.array([3, 1, 4, 2])


# Sorting by arr2 first, then arr1

sorted_indices = np.lexsort((arr1, arr2))

print("Indices that would sort based on arr2 then arr1:", sorted_indices)

print("Array sorted using indices:", arr1[sorted_indices])
```

Output:

Indices that would sort based on arr2 then arr1: [1 3 0 2]

Array sorted using indices: [4 2 1 3]

Sorting Multidimensional Arrays:

You can sort along specific axes in multidimensional arrays using the axis parameter.

Example: Sorting 2D Array along Axis 0 (columns)

```
arr = np.array([[3, 1, 4], [1, 5, 9], [2, 6, 5]])

sorted_arr = np.sort(arr, axis=0)

print("Original array:\n", arr)

print("Sorted array along axis 0 (columns):\n", sorted_arr)
```

Output:

Original array:

[[3 1 4]

[1 5 9]

[2 6 5]]

Sorted array along axis 0 (columns):

[[1 1 4]

[2 5 5]

[3 6 9]]

Example: Sorting 2D Array along Axis 1 (rows)

python

Copy

```
sorted_arr_2 = np.sort(arr, axis=1)
```

```
print("Sorted array along axis 1 (rows):\n", sorted_arr_2)
```

Output:

```
Sorted array along axis 1 (rows):
 [[1 3 4]
 [1 5 9]
 [2 5 6]]
```

Sorting in Descending Order:

To sort in descending order, you can either sort the array and reverse the result, or you can pass the argument -1 to the sorting function for descending order.

Example: Sorting in Descending Order

python

Copy

```
arr = np.array([3, 1, 4, 1, 5, 9, 2])
sorted_arr_desc = np.sort(arr)[::-1]  # Reverse the sorted array
print("Sorted array in descending order:", sorted_arr_desc)
```

Output:

```
Sorted array in descending order: [9 5 4 3 2 1 1]
```

Conclusion:

np.sort(): Returns a sorted copy of the array.

arr.sort(): Sorts the array in-place, modifying the original.

np.argsort(): Returns indices that would sort the array.

np.lexsort(): Sorts by multiple keys.

Sorting along axes: You can sort along specific axes for multidimensional arrays using the axis parameter.