Search the docs ...

Array objects
The N-dimensional array ( ndarray )
Scalars
Data type objects ( dtype )
Indexing routines
numpy.c_
numpy.r_
numpy.s_
numpy.nonzero
numpy.where
numpy.indices
numpy.ix_
numpy.ogrid
numpy.ravel_multi_index
numpy.unravel_index
numpy.diag_indices
numpy.diag_indices_from
numpy.mask_indices
numpy.tril_indices
numpy.tril_indices_from
numpy.triu_indices
numpy.triu_indices_from
numpy.take
numpy.take_along_axis
numpy.choose
numpy.compress
numpy.diag
numpy.diagonal
numpy.select
numpy.lib.stride_tricks.slidi
numpy.lib.stride_tricks.as_s
numpy.place
numpy.put
numpy.put_along_axis
numpy.putmask
numpy.fill_diagonal
numpy.nditer
numpy.ndenumerate
numpy.ndindex
numpy.nested_iters
numpy.flatiter
numpy.lib.Arrayterator
numpy.iterable
Iterating Over Arrays

# numpy.take_along_axis

`numpy.take_along_axis(arr, indices, axis)` [source]

Take values from the input array by matching 1d index and data slices.

This iterates over matching 1d slices oriented along the specified axis in the index and data arrays, and uses the former to look up values in the latter. These slices can be different lengths.

Functions returning an index along an axis, like `argsort` and `argpartition`, produce suitable indices for this function.

*New in version 1.15.0.*

Parameters: **arr** : *ndarray (Ni..., M, Nk...)*

Source array

**indices** : *ndarray (Ni..., J, Nk...)*

Indices to take along each 1d slice of *arr*. This must match the dimension of arr, but dimensions Ni and Nj only need to broadcast against *arr*.

**axis** : *int*

The axis to take 1d slices along. If axis is None, the input array is treated as if it had first been flattened to 1d, for consistency with `sort` and `argsort`.

Returns: **out: ndarray (Ni..., J, Nk...)**

The indexed result.

See also

`take`

Take along an axis, using the same indices for every 1d slice

`put_along_axis`

Put values into the destination array by matching 1d index and data slices

## Notes

This is equivalent to (but faster than) the following use of `ndindex` and `s_`, which sets each of `ii` and `kk` to a tuple of indices:

```
Ni, M, Nk = a.shape[:axis], a.shape[axis], a.shape[axis+1:]
J = indices.shape[axis]  # Need not equal M
out = np.empty(Ni + (J,) + Nk)

for ii in ndindex(Ni):
    for kk in ndindex(Nk):
        a_1d       = a      [ii + s_[:,] + kk]
        indices_1d = indices[ii + s_[:,] + kk]
        out_1d     = out    [ii + s_[:,] + kk]
        for j in range(J):
            out_1d[j] = a_1d[indices_1d[j]]
```

Equivalently, eliminating the inner loop, the last two lines would be:

```
out_1d[:] = a_1d[indices_1d]
```

## Examples

For this sample array

```
>>> a = np.array([[10, 30, 20], [60, 40, 50]])
```

We can sort either by using sort directly, or argsort and this function

```
>>> np.sort(a, axis=1)
array([[10, 20, 30],
       [40, 50, 60]])
>>> ai = np.argsort(a, axis=1); ai
array([[0, 2, 1],
       [1, 2, 0]])
>>> np.take_along_axis(a, ai, axis=1)
array([[10, 20, 30],
       [40, 50, 60]])
```

The same works for max and min, if you expand the dimensions:

```
>>> np.expand_dims(np.max(a, axis=1), axis=1)
array([[30],
       [60]])
>>> ai = np.expand_dims(np.argmax(a, axis=1), axis=1)
>>> ai
array([[1],
       [0]])
>>> np.take_along_axis(a, ai, axis=1)
array([[30],
       [60]])
```

If we want to get the max and min at the same time, we can stack the indices first

```
>>> ai_min = np.expand_dims(np.argmin(a, axis=1), axis=1)
>>> ai_max = np.expand_dims(np.argmax(a, axis=1), axis=1)
>>> ai = np.concatenate([ai_min, ai_max], axis=1)
>>> ai
array([[0, 1],
       [1, 0]])
>>> np.take_along_axis(a, ai, axis=1)
array([[10, 30],
       [40, 60]])
```

Previous
numpy.take

Next
numpy.choose

© Copyright 2008-2022, NumPy Developers.
Created using Sphinx 4.5.0.