Date:-28-08-2020

Data Analysis using Python

Chapter 4: NumPy Basics: Arrays and Vectorized Computation

Fancy Indexing

Fancy indexing is a term adopted by NumPy to describe indexing using integer arrays.

Suppose we had an 8 × 4 array:

```
In []: arr = np.empty((8, 4))

In []: for i in range(8):
.....:     arr[i] = i

In []: arr
Out[]:
array ([[ 0., 0., 0., 0.],
[ 1., 1., 1., 1.],
[ 2., 2., 2., 2.],
[ 3., 3., 3., 3.],
[ 4., 4., 4., 4.],
[ 5., 5., 5., 5.],
[ 6., 6., 6., 6.],
[ 7., 7., 7., 7.]])
```

To select out a subset of the rows in a particular order, you can simply pass a list or ndarray of integers specifying the desired order:

```
In []:    arr[[4, 3, 0, 6]]
Out[]:
array([[ 4., 4., 4., 4.],
[ 3., 3., 3., 3.],
[ 0., 0., 0., 0.],
[ 6., 6., 6., 6.]])
```

Hopefully this code did what you expected! Using negative indices selects rows from the end:

```
In []: arr[[-3, -5, -7]]

Out[]:
array([[ 5., 5., 5., 5.],
[ 3., 3., 3., 3.],
[ 1., 1., 1., 1.]])
```

Passing multiple index arrays does something slightly different; it selects a one dimensional array of elements corresponding to each tuple of indices:

```
In []: arr = np.arange(32).reshape((8, 4))
In []: arr
Out[]:
```

```
array([[ 0, 1, 2, 3],
[ 4, 5, 6, 7],
[ 8, 9, 10, 11],
[12, 13, 14, 15],
[16, 17, 18, 19],
[20, 21, 22, 23],
[24, 25, 26, 27],
[28, 29, 30, 31]])

In []: arr[[1, 5, 7, 2], [0, 3, 1, 2]]
Out[]: array([ 4, 23, 29, 10])
```

Here the elements (1, 0), (5, 3), (7, 1), and (2, 2) were selected. Regardless of how many dimensions the array has (here, only 2), the result of fancy indexing is always one-dimensional.

The behaviour of fancy indexing in this case is a bit different from what some users might have expected (me included).

```
In []: arr[[1, 5, 7, 2]][:, [0, 3, 1, 2]]
Out[]:
array([[ 4, 7, 5, 6],
[20, 23, 21, 22],
[28, 31, 29, 30],
[ 8, 11, 9, 10]])
```

Arrays have the transpose method and also the special T attribute:

```
In []: arr = np.arange(15).reshape((3, 5))
In []: arr
Out[]:
array([[ 0, 1, 2, 3, 4],
[ 5, 6, 7, 8, 9],
[10, 11, 12, 13, 14]])

In []: arr.T
Out[]:
array([[ 0, 5, 10],
[ 1, 6, 11],
[ 2, 7, 12],
[ 3, 8, 13],
[ 4, 9, 14]])
```

When doing matrix computations, you may do this very often—for example, when computing the inner matrix product using np.dot:

```
In []: arr = np.random.randn(6, 3)
In []: arr
Out[]:
array([[-0.8608, 0.5601, -1.2659],
[ 0.1198, -1.0635, 0.3329],
[-2.3594, -0.1995, -1.542 ],
[-0.9707, -1.307 , 0.2863],
[ 0.378 , -0.7539, 0.3313],
[ 1.3497, 0.0699, 0.2467]])
```

```
In []: np.dot(arr.T, arr)
Out[]:
array([[ 9.2291, 0.9394, 4.948 ],
[ 0.9394, 3.7662, -1.3622],
[ 4.948 , -1.3622, 4.3437]])
```

For higher dimensional arrays, transpose will accept a tuple of axis numbers to permute the axes (for extra mind bending):

```
In []: arr = np.arange(16).reshape((2, 2, 4))
In []: arr
Out[]:
array([[[ 0, 1, 2, 3],
[ 4, 5, 6, 7]],
[[ 8, 9, 10, 11],
[12, 13, 14, 15]]])

In []: arr.transpose((1, 0, 2))
Out[]:
array([[[ 0, 1, 2, 3],
[ 8, 9, 10, 11]],
[[ 4, 5, 6, 7],
[12, 13, 14, 15]]])
```

Here, the axes have been reordered with the second axis first, the first axis second, and the last axis unchanged. Simple transposing with .T is a special case of swapping axes. ndarray has the method Swap axes, which takes a pair of axis numbers and switches the indicated axes to rearrange the data:

```
In []: arr
Out[]:
array([[[ 0, 1, 2, 3],
[ 4, 5, 6, 7]],
[[ 8, 9, 10, 11],
[12, 13, 14, 15]]])
In []: arr.swapaxes(1, 2)
Out[]:
array([[[ 0, 4],
[ 1, 5],
[ 2, 6],
[ 3, 7]],
[[ 8, 12],
[ 9, 13],
[10, 14],
[11, 15]]])
```

swapaxes similarly returns a view on the data without making a copy.

From:-
Ritu Meena
Assistant Professor
Shivaji College
Delhi University