Theory: Boolean operations on arrays

16 minutes 0 / 5 problems solved

In previous topics, we have discussed some array operations. NumPy, however, has many other useful functions. We can use comparison operators such as `<`, `>` or `==` on an array in NumPy, and the result of these operations will be another array with Boolean data type elements. In this topic, we will cover the main aspects of these operations.

§1. Integer comparison

You can compare elements in your array with a given integer using the following comparison operators: `<`, `>`, `<=`, `>=`, `==`, `!=`.

```
a1 = np.array([1, 2, 3, 4, 5])
print(a1 < 4)
# [ True  True  True False False]
```

As you can see, the resulting array contains `True` and `False` values. The first three elements are less than 4, so we have three `True` values. The last two elements are bigger, so the two `False` values.

We can use these comparisons with multidimensional arrays as well. Again, each element is compared to an integer, then the `True` or `False` value is returned.

```
a1 = np.array([[11, 22], [33, 44], [55, 66]])
print(a1 >= 44)
# [[False False]
#  [False  True]
#  [ True  True]]
```

Other operators are used in the same way.

§2. Array comparison

Using the operations above, we can compare two arrays as well:

```
a1 = np.array([[12, 23], [16, 40], [15, 16]])
a2 = np.array([[12, 73], [96, 10], [25, 16]])
print(a1 >= a2)
# [[ True False]
#  [False  True]
#  [False  True]]
```

Two arrays must have the same shape or they must be broadcastable for comparisons. In the example below, the second array is broadcastable to the shape of the first one. The comparison is element-wise.

```
a1 = np.array([[12, 23], [16, 40], [15, 76]])
a2 = np.array([[12, 73]])
print(a1 < a2)
# [[False  True]
#  [False  True]
#  [False False]]
```

If arrays don't meet the mentioned requirements, a `ValueError` will be raised.

```
a1 = np.array([[12, 23], [16, 40], [15, 76]])
a2 = np.array([[12, 73, 3]])
print(a1 > a2)
# Traceback (most recent call last):
#   File "main.py", line 5, in
#     print(a1 > a2)

# ValueError: operands could not be broadcast together with shapes (3,2) (1,3)
```

Current topic:

Boolean operations on arrays

Topic depends on:

× Operations with several arrays

Table of contents:

The result of the operations with integers and arrays is always a Boolean array.

§3. Operators and functions of comparison

Apart from comparison operators, NumPy has functions that have the same purpose as the operators. The table below shows them in full.

Operators	Functions
`>`	`np.greater()`
`<`	`np.less()`
`>=`	`np.greater_equal()`
`<=`	`np.less_equal()`
`==`	`np.equal()`
`!=`	`np.not_equal()`

Have a look at the code below; we use the `np.greater()` function instead of the operator. It is a complete equivalent of the expression `a1 > a2`:

```
a1 = np.array([3, 8, 79])
a2 = np.array([1, 0, 75])
print(np.greater(a1, a2))
# [ True  True  True]
```

Similarly, you can use an integer for comparison in the functions:

```
a1 = np.array([3, 8, 79])
print(np.less(a1, 56))
# [ True  True  False]
```

The operators are equal to the functions. You can use any of these two approaches in your programs.

§4. Logic functions

If you want to check whether *any* or *all* the values in an array fulfill a specific condition, `np.any()` and `np.all()` can be used:

```
a1 = np.array([3, 8, 9])
print(np.any(a1 < 4))  # True
```

The code above checks whether the array contains *at least one* element that is less than 4. The condition is fulfilled, so the `True` value is returned. If we change `np.any()` to `np.all()`, the `True` value will be obtained if *all the elements* in the array meet the requirement:

```
a1 = np.array([3, 8, 9])
print(np.all(a1 < 4))  # False
```

In the array, the elements "8" and "9" don't fulfill the condition, so the result of `np.all()` is `False`.

§5. Selecting elements in an array

Sometimes we may want to select elements in the given array based on some condition. There is the `np.where()` function for that. It accepts a Boolean array. The Boolean array defines a criterion for selecting elements. Let's look at the example:

```
a1 = np.array([19, 92, 53, 44, 35])
spec = np.where(a1 > 37)
print(a1[spec])  # [92 53 44]
```

We need to create an array, then a `spec` variable in which we specify that the elements in `a1` should be greater than 37. To understand this idea better, let's compare the two following lines of the code:

```
print(a1 > 37)  # [False  True  True  True False]
print(np.where(a1 > 37))  # (array([1, 2, 3]),)
```

`a > 37` returns a *list* of `True` and `False` values, whereas `np.where()` returns a tuple with the indexes of elements that are `True`. Finally, getting back to our example. We print the `a1[spec]` array; we used the `spec` variable to isolate values that are `True`.

Note that `np.where()` can accept a list of Boolean values right away, instead of an operation that will return a list. It works the same as in the examples above:

```
a1 = np.array([19, 92, 53, 44, 35])
spec = np.where([False, True, True, True, False])
print(a1[spec])  # [92 53 44]
```

We can also apply `np.where()` to two arrays by merging them. Take a look at the following example:

```
num = np.array([1, 2, 3, 4, 5])
a1 = np.array(['red', 'orange', 'green', 'yellow', 'white'])
a2 = np.array(['black', 'brown', 'purple', 'pink', 'blue'])
a3 = np.where(num > 2, a1, a2)
print(a3)  # ['black' 'brown' 'green' 'yellow' 'white']
```

Let's discuss how this array was obtained. The first variable, `num`, contains integers from 1 to 5. Then we create two arrays of the same shape, with which we will work further. After that, `np.where()` accepts a Boolean array with `True` and `False` values, obtained from comparing each `num` element with 2. For every `True` value it chooses a corresponding element from `a1`, whereas for every `False` value a corresponding element from `a2` is chosen. That's why the resulting array is `['black' 'brown' 'green' 'yellow' 'white']`: the first two elements in `num` were `False`, the next three were `True`.

§6. Conclusion

In this topic, we have learned:

- how to use comparison operators for arrays;
- how to check if any or all array values fulfill a condition;
- how to select specified elements using a condition.

Now let's practice new knowledge so that it will be easier for you to use it in the future.

Report a typo

15 users liked this theory. **0** didn't like it. **What about you?**

Start practicing

Comments (0) Hints (0) Useful links (0) Show discussion