# Comparisons, Masks, and Boolean Logic

This section covers the use of Boolean masks to examine and manipulate values within NumPy arrays. Masking comes up when you want to extract, modify, count, or otherwise manipulate values in an array based on some criterion: for example, you might wish to count all values greater than a certain value, or perhaps remove all outliers that are above some threshold. In NumPy, Boolean masking is often the most efficient way to accomplish these types of tasks.

## Comparison Operators as ufuncs

In [Computation on NumPy Arrays: Universal Functions](02.03-Computation-on-arrays-ufuncs.ipynb) we introduced ufuncs, and focused in particular on arithmetic operators. We saw that using +, -, \*, /, and others on arrays leads to element-wise operations. NumPy also implements comparison operators such as < (less than) and > (greater than) as element-wise ufuncs. The result of these comparison operators is always an array with a Boolean data type. All six of the standard comparison operations are available:

import numpy as np  
x = np.array([1, 2, 3, 4, 5])

x < 3 # less than

array([ True, True, False, False, False])

x > 3 # greater than

array([False, False, False, True, True])

x <= 3 # less than or equal

array([ True, True, True, False, False])

x >= 3 # greater than or equal

array([False, False, True, True, True])

x != 3 # not equal

array([ True, True, False, True, True], dtype=bool)

x == 3 # equal

array([False, False, True, False, False], dtype=bool)

It is also possible to do an element-wise comparison of two arrays, and to include compound expressions:

(2 \* x) == (x \*\* 2)

array([False, True, False, False, False])

As in the case of arithmetic operators, the comparison operators are implemented as ufuncs in NumPy; for example, when you write x < 3, internally NumPy uses np.less(x, 3). A summary of the comparison operators and their equivalent ufunc is shown here:

| Operator | Equivalent ufunc || Operator | Equivalent ufunc | |---------------|---------------------||---------------|---------------------| |== |np.equal ||!= |np.not\_equal | |< |np.less ||<= |np.less\_equal | |> |np.greater ||>= |np.greater\_equal |

Just as in the case of arithmetic ufuncs, these will work on arrays of any size and shape. Here is a two-dimensional example:

rng = np.random.RandomState(0)  
x = rng.randint(10, size=(3, 4))  
x

array([[5, 0, 3, 3],  
 [7, 9, 3, 5],  
 [2, 4, 7, 6]])

x < 6

array([[ True, True, True, True],  
 [False, False, True, True],  
 [ True, True, False, False]], dtype=bool)

In each case, the result is a Boolean array, and NumPy provides a number of straightforward patterns for working with these Boolean results.

## Working with Boolean Arrays

Given a Boolean array, there are a host of useful operations you can do. We'll work with x, the two-dimensional array we created earlier.

print(x)

[[5 0 3 3]  
 [7 9 3 5]  
 [2 4 7 6]]

### Counting entries

To count the number of True entries in a Boolean array, np.count\_nonzero is useful:

# how many values less than 6?  
np.count\_nonzero(x < 6)

8

We see that there are eight array entries that are less than 6. Another way to get at this information is to use np.sum; in this case, False is interpreted as 0, and True is interpreted as 1:

np.sum(x < 6)

8

The benefit of sum() is that like with other NumPy aggregation functions, this summation can be done along rows or columns as well:

# how many values less than 6 in each row?  
np.sum(x < 6, axis=1)

array([4, 2, 2])

This counts the number of values less than 6 in each row of the matrix.

If we're interested in quickly checking whether any or all the values are true, we can use (you guessed it) np.any or np.all:

# are there any values greater than 8?  
np.any(x > 8)

True

# are there any values less than zero?  
np.any(x < 0)

False

# are all values less than 10?  
np.all(x < 10)

True

# are all values equal to 6?  
np.all(x == 6)

False

np.all and np.any can be used along particular axes as well. For example:

# are all values in each row less than 8?  
np.all(x < 8, axis=1)

array([ True, False, True], dtype=bool)

Here all the elements in the first and third rows are less than 8, while this is not the case for the second row.

### Boolean operators

NumPy Supports *bitwise logic operators*, &, |, ^, and ~.

Combining comparison operators and Boolean operators on arrays can lead to a wide range of efficient logical operations.

The following table summarizes the bitwise Boolean operators and their equivalent ufuncs:

| Operator | Equivalent ufunc || Operator | Equivalent ufunc | |---------------|---------------------||---------------|---------------------| |& |np.bitwise\_and ||| |np.bitwise\_or | |^ |np.bitwise\_xor ||~ |np.bitwise\_not |

## Boolean Arrays as Masks

In the preceding section we looked at aggregates computed directly on Boolean arrays. A more powerful pattern is to use Boolean arrays as masks, to select particular subsets of the data themselves. Returning to our x array from before, suppose we want an array of all values in the array that are less than, say, 5:

x

array([[5, 0, 3, 3],  
 [7, 9, 3, 5],  
 [2, 4, 7, 6]])

We can obtain a Boolean array for this condition easily, as we've already seen:

x < 5

array([[False, True, True, True],  
 [False, False, True, False],  
 [ True, True, False, False]], dtype=bool)

Now to *select* these values from the array, we can simply index on this Boolean array; this is known as a *masking* operation:

x[x < 5]

array([0, 3, 3, 3, 2, 4])

What is returned is a one-dimensional array filled with all the values that meet this condition; in other words, all the values in positions at which the mask array is True.