## Boolean Indexing, Set Operations, and Sorting

**Erratum** - At time stamp 2:40 in the video above, it is shown to sort the unique elements as: print(np.sort(np.unique(x))). In this statement, the unique() function already returns the **sorted unique elements** of the array. Therefore, the print(np.unique(x)) will give you equivalent result.

Up to now we have seen how to make slices and select elements of an ndarray using indices. This is useful when we know the exact indices of the elements we want to select. However, there are many situations in which we don't know the indices of the elements we want to select. For example, suppose we have a 10,000 x 10,000 ndarray of random integers ranging from 1 to 15,000 and we only want to select those integers that are less than 20. *Boolean* indexing can help us in these cases, by allowing us select elements using logical arguments instead of explicit indices. Let's see some examples:

### Example 1. Boolean indexing

```
# We create a 5 x 5 ndarray that contains integers from 0 to 24

X = np.arange(25).reshape(5, 5)


# We print X

print()

print('Original X = \n', X)

print()


# We use Boolean indexing to select elements in X:

print('The elements in X that are greater than 10:', X[X > 10])

print('The elements in X that less than or equal to 7:', X[X <= 7])

print('The elements in X that are between 10 and 17:', X[(X > 10) & (X < 17)])


# We use Boolean indexing to assign the elements that are between 10 and 17 the value of -1

X[(X > 10) & (X < 17)] = -1


# We print X

print()

print('X = \n', X)

print()
```

```
Original X =
[[ 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]]

The elements in X that are greater than 10: [11 12 13 14 15 16 17 18 19 20 21 22 23 24]
The elements in X that less than or equal to 7: [0 1 2 3 4 5 6 7]
The elements in X that are between 10 and 17: [11 12 13 14 15 16]

X =
[[ 0 1 2 3 4]
 [ 5 6 7 8 9]
 [10 -1 -1 -1 -1]
 [-1 -1 17 18 19]
 [20 21 22 23 24]]
```

In addition to Boolean Indexing NumPy also allows for set operations. This useful when comparing ndarrays, for example, to find common elements between two ndarrays. Let's see some examples:

### Example 2. Set operations

```
# We create a rank 1 ndarray

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


# We create a rank 1 ndarray

y = np.array([6,7,2,8,4])


# We print x

print()

print('x = ', x)


# We print y

print()

print('y = ', y)


# We use set operations to compare x and y:
```

```
print()

print('The elements that are both in x and y:', np.intersect1d(x,y))

print('The elements that are in x that are not in y:', np.setdiff1d(x,y))

print('All the elements of x and y:',np.union1d(x,y))
```

x = [1 2 3 4 5]

y = [6 7 2 8 4]

The elements that are both in x and y: [2 4]
The elements that are in x that are not in y: [1 3 5]
All the elements of x and y: [1 2 3 4 5 6 7 8]

### numpy.ndarray.sort method

Syntax:

```
ndarray.sort(axis=-1, kind=None, order=None)
```

The method above sorts an array in-place. All arguments are optional, see the details here.

Like with other functions we saw before, the sort can be used as a method as well as a function. The difference lies in how the data is stored in memory in this case.

- When numpy.sort() is used as a function, it sorts the ndrrays out of place, meaning, that it doesn't change the original ndarray being sorted.
- On the other hand, when you use numpy.ndarray.sort() as a method, ndarray.sort() sorts the ndarray in place, meaning, that the original array will be changed to the sorted one.

Let's see some examples:

### Example 3. Sort arrays using sort() function

```
# We create an unsorted rank 1 ndarray

x = np.random.randint(1,11,size=(10,))


# We print x

print()

print('Original x = ', x)


# We sort x and print the sorted array using sort as a function.

print()
```

```
print('Sorted x (out of place):', np.sort(x))


# When we sort out of place the original array remains intact. To see this we print x again
print()
print('x after sorting:', x)
```

Original x = [9 6 4 4 9 4 8 4 4 7]

Sorted x (out of place): [4 4 4 4 4 6 7 8 9 9]

x after sorting: [9 6 4 4 9 4 8 4 4 7]

Notice that np.sort() sorts the array but, if the ndarray being sorted has repeated values, np.sort() leaves those values in the sorted array. However, if desired, we can use the unique() function. Let's see how we can sort the unique elements of x above:

```
# Returns the sorted unique elements of an array
print(np.unique(x))
```

[4 6 7 8 9]

Finally, let's see how we can sort ndarrays in place, by using sort as a method:

### Example 4. Sort rank-1 arrays using sort() method

```
# We create an unsorted rank 1 ndarray
x = np.random.randint(1,11,size=(10,))


# We print x
print()
print('Original x = ', x)


# We sort x and print the sorted array using sort as a method.
x.sort()


# When we sort in place the original array is changed to the sorted array. To see this we print x again
print()
print('x after sorting:', x)
```

Original x = [9 9 8 1 1 4 3 7 2 8]

x after sorting: [1 1 2 3 4 7 8 8 9 9]

### numpy.sort function

Syntax:

```
numpy.sort(array, axis=-1, kind=None, order=None)
```

It returns a sorted copy of an array. The axis denotes the axis along which to sort. It can take values in the range -1 to (ndim-1). Axis can take the following possible values for a given 2-D ndarray:

- If nothing is specified, the default value is axis = -1, which sorts along the **last** axis. In the case of a given 2-D ndarray, the last axis value is 1.
- If explicitly axis = None is specified, the array is flattened before sorting. It will return a 1-D array.
- If axis = 0 is specified for a given 2-D array - For one column at a time, the function will sort all rows, without disturbing other elements. In the final output, *you will see that each column has been sorted individually.*
- The output of axis = 1 for a given 2-D array is vice-versa for axis = 0. In the final output, *you will see that each row has been sorted individually.*

**Tip**: As mentioned in this discussion, you can read axis = 0 as "**down**" and axis = 1 as "**across**" the given 2-D array, to have a correct usage of axis in your methods/functions.

Refer here for details about the optional arguments.

When sorting rank 2 ndarrays, we need to specify to the np.sort() function whether we are sorting by rows or columns. This is done by using the axis keyword. Let's see some examples:

### Example 5. Sort rank-2 arrays by specific axis.

```
# We create an unsorted rank 2 ndarray

X = np.random.randint(1,11,size=(5,5))


# We print X

print()

print('Original X = \n', X)

print()


# We sort the columns of X and print the sorted array
```

```
print()

print('X with sorted columns :\n', np.sort(X, axis = 0))


# We sort the rows of X and print the sorted array

print()

print('X with sorted rows :\n', np.sort(X, axis = 1))
```

```
Original X =
[[6 1 7 6 3]
 [3 9 8 3 5]
 [6 5 8 9 3]
 [2 1 5 7 7]
 [9 8 1 9 8]]

X with sorted columns :
[[2 1 1 3 3]
 [3 1 5 6 3]
 [6 5 7 7 5]
 [6 8 8 9 7]
 [9 9 8 9 8]]

X with sorted rows :
[[1 3 6 6 7]
 [3 3 5 8 9]
 [3 5 6 8 9]
 [1 2 5 7 7]
 [1 8 8 9 9]]
```

### Additional Resource

- Try out the advanced examples of indexing available at NumPy.org - Indexing

### Supporting Materials

- Boolean Indexing, Set Operations, and Sorting