COMPARISONS, MASKS, AND BOOLEAN LOGIC

Comparison Operators as ufuncs.

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:

```
x = np.array([1, 2, 3, 4, 5])
x < 3 # less than
array([ True, True, False, False, False], dtype=bool)
x > 3 # greater than
array([False, False, False, True, True], dtype=bool)
x <= 3 # less than or equal
array([ True, True, True, False, False], dtype=bool)
x >= 3 # greater than or equal
array([False, False, True, True, True], dtype=bool)
x != 3 # not equal
array([ True, True, False, True, True], dtype=bool)
x == 3 # equal
array([False, False, True, False, False], dtype=bool)
```

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)
```

The result is a Boolean array, and NumPy provides a number of straightforward patterns for working with these Boolean results.

Working with Boolean Arrays

- np.count_nonzero()
- np.sum()
- np.sum(x , axis)
- np.any()
- np.all()
- np.all(x , axis)

Boolean operators

Operator	Equivalent ufunc
&	np.bitwise_and
|	np.bitwise_or
^	np.bitwise_xor
~	np.bitwise_not

Example

np.sum((inches > 0.5) & (inches < 1))

inches > (0.5 & inches) < 1

np.sum(~((inches <= 0.5) | (inches >= 1)))

Boolean Arrays as Masks

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

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

Example

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

x < 5
array([[False, True, True, True],
       [False, False, True, False],
       [ True, True, False, False]], dtype=bool)
```

Masking operation

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.

FANCY INDEXING

Fancy indexing is like the simple indexing we've already seen, but we pass arrays of indices in place of single scalars. This allows us to very quickly access and modify complicated subsets of an array's values.

Exploring Fancy Indexing

Fancy indexing is conceptually simple: it means passing an array of indices to access multiple array elements at once.

Types of fancy indexing.

- Indexing / accessing more values
- Array of indices
- In multi-dimensional
- Standard indexing

Example

```
import numpy as np
rand = np.random.RandomState(42)
x = rand.randint(100, size=10)
print(x)

[51 92 14 71 60 20 82 86 74 74]
```

Indexing / accessing more values

Suppose we want to access three different elements. We could do it like this:

```
[x[3], x[7], x[2]]

[71, 86, 14]
```

Array of indices

We can pass a single list or array of indices to obtain the same result.

ind = [3, 7, 4]

x[ind]

array([71, 86, 60])

In multi-dimensional

Fancy indexing also works in multiple dimensions. Consider the following array.

```
X = np.arange(12).reshape((3, 4))
X

array([[ 0, 1, 2, 3],
       [ 4, 5, 6, 7],
       [ 8, 9, 10, 11]])
```

Standard indexing

Like with standard indexing, the first index refers to the row, and the second to the column.

row = np.array([0, 1, 2])

col = np.array([2, 1, 3])

X[row, col]

array ([2, 5, 11])

Combined Indexing

For even more powerful operations, fancy indexing can be combined with the other indexing schemes we've seen.

Example array

print(X)

[[0 1 2 3]

[4 5 6 7]

[8 9 10 11]]

- Combine fancy and simple indices

  X[2, [2, 0, 1]]

  array([10, 8, 9])

- Combine fancy indexing with slicing

  X[1:, [2, 0, 1]]

  array([[6, 4, 5], [10, 8, 9]])

- Combine fancy indexing with masking

  mask = np.array([1, 0, 1, 0], dtype=bool)

```
X[row[:, np.newaxis], mask]
array([[ 0, 2],
       [ 4, 6],
       [ 8, 10]])
```

Modifying Values with Fancy Indexing

Just as fancy indexing can be used to access parts of an array, it can also be used to modify parts of an array. Change some value in an array

Modify particular element by index

For example, imagine we have an array of indices and we'd like to set the corresponding items in an array to some value.

```
x = np.arange(10)
i = np.array([2, 1, 8, 4])
x[i] = 99
print(x)
```

```
[ 0 99 99 3 99 5 6 7 99 9]
```

Using assignment operator

We can use any assignment-type operator for this. For example

```
x[i] -= 10
print(x)
```

```
[ 0 89 89 3 89 5 6 7 89 9]
```

Using at()

Use the at() method of ufuncs for other behavior of modifications.

```
x = np.zeros(10)
np.add.at(x, i, 1)
print(x)
```

```
[ 0. 0. 1. 2. 3. 0. 0. 0. 0. 0.]
```