## BASICS OF NUMPY ARRAYS

NumPy (short for Numerical Python) provides an efficient interface to store and operate on dense data buffers. NumPy arrays are like Python's built-in list type, but NumPy arrays provide much more efficient storage and data operations as the arrays grow larger in size.

We'll cover a few categories of basic array manipulations here:

**Attributes of arrays**

Determining the size, shape, memory consumption, and data types of arrays

**Indexing of arrays**

Getting and setting the value of individual array elements

**Slicing of arrays**

Getting and setting smaller subarrays within a larger array

**Reshaping of arrays**

Changing the shape of a given array

**Joining and splitting of arrays**

Combining multiple arrays into one, and splitting one array into many

**NumPy Array Attributes**

- ndim (the number of dimensions),
- shape (the size of each dimension)
- size (the total size of the array)

Example

```
np.random.seed(0) # seed for reproducibility
x1 = np.random.randint(10, size=6) # One-dimensional array
x2 = np.random.randint(10, size=(3, 4)) # Two-dimensional array
x3 = np.random.randint(10, size=(3, 4, 5)) # Three-dimensional array
print("x3 ndim: ", x3.ndim)
print("x3 shape:", x3.shape)
print("x3 size: ", x3.size)

print("dtype:", x3.dtype)

print("itemsize:", x3.itemsize, "bytes")
print("nbytes:", x3.nbytes, "bytes")
```

**Array Indexing:**

- Accessing Single Elements

**Accessing Single Elements**

- Indexing in NumPy will feel quite familiar like list indexing,
- In a one-dimensional array, you can access the ith value (counting from zero) by specifying the desired index in square brackets, just as with Python lists

- To index from the end of the array, you can use negative indices
- In a multidimensional array, you access items using a comma-separated tuple of indices
- Unlike Python lists, NumPy arrays have a fixed type. This means, for example, that if you attempt to insert a floating-point value to an integer array, the value will be silently truncated.

**Array Slicing: Accessing Subarrays**

Just as we can use square brackets to access individual array elements, we can also use them to access subarrays with the slice notation, marked by the colon (:) character.

The NumPy slicing syntax follows that of the standard Python list; to access a slice of an array x, use this:

x[start:stop:step]
start – starting array index
stop – array index to stop ( last value will not be considered)
step – terms has to be printed from start to stop
Default to the values start=0, stop=size of dimension, step=1.

Example

*x = np.arange(10)*
*x*

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

x[:5] # prints first five elements

array([0, 1, 2, 3, 4])

x[5:] *# elements after index 5*

array([5, 6, 7, 8, 9])

x[4:7] *# middle subarray(from 4th index to 6th index)*

array([4, 5, 6])

While using negative indices the defaults for start and stop are swapped. This becomes a convenient way to reverse an array

x[::-1] *# all elements, reversed*

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

x[5::-2] *# reversed every other from index 5*

array([5, 3, 1])

**Multidimensional sub arrays**

Multidimensional slices work in the same way, with multiple slices separated by commas.

For example:

```
x2
array([[12, 5, 2, 4],
       [ 7, 6, 8, 8],
       [ 1, 6, 7, 7]])


x2[:2, :3] # two rows, three columns
array([[12, 5, 2],
       [ 7, 6, 8]])

x2[:3, ::2] # all rows, every other column(every second column)
array([[12, 2],
       [ 7, 8],
       [ 1, 7]])
```

Finally, sub array dimensions can even be reversed together

```
x2[::-1, ::-1]
array([[ 7, 7, 6, 1],
       [ 8, 8, 6, 7],
       [ 4, 2, 5, 12]])
```

**Reshaping of Arrays**

The most flexible way of doing this is with the **reshape()** method. For example, if you want to put the numbers 1 through 9 in a 3×3 grid, you can do the following

```
grid = np.arange(1, 10).reshape((3, 3))
print(grid)
    [[1 2 3]
    [4 5 6]
    [7 8 9]]
```

**Array Concatenation and Splitting**

**Concatenation of arrays**

Concatenation, or joining of two arrays in NumPy, is primarily accomplished through the routines **np.concatenate, np.vstack,** and **np.hstack.** np.concatenate takes a tuple or list of arrays as its first argument.

```
x = np.array([1, 2, 3])
y = np.array([3, 2, 1])
np.concatenate([x, y])
array([1, 2, 3, 3, 2, 1])
```

You can also concatenate more than two arrays at once

```
z = [99, 99, 99]
print(np.concatenate([x, y, z]))
```

```
[ 1 2 3 3 2 1 99 99 99]
```

np.concatenate can also be used for two-dimensional arrays

```
grid = np.array([[1, 2, 3],
                 [4, 5, 6]])
np.concatenate([grid, grid])
```

```
array([[1, 2, 3],
       [4, 5, 6],
       [1, 2, 3],

       [4, 5, 6]])
```

**Concatenate along the second axis (zero-indexed)**

```
np.concatenate([grid, grid], axis=1)
```

```
array([[1, 2, 3, 1, 2, 3],
       [4, 5, 6, 4, 5, 6]])
```

**np.vstack (vertical stack) functions**

```
x = np.array([1, 2, 3])
grid = np.array([[9, 8, 7],
                 [6, 5, 4]])
np.vstack([x, grid])
```

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

np.hstack (horizontal stack) functions

```
y = np.array([[99],
              [99]])
np.hstack([grid, y])
```

```
array([[ 9, 8, 7, 99],
       [ 6, 5, 4, 99]])
```

**Splitting of arrays**

The opposite of concatenation is splitting, which is implemented by the functions **np.split, np.hsplit**, and **np.vsplit**. For each of these, we can pass a list of indices giving the split points.

```
x = [1, 2, 3, 99, 99, 3, 2, 1]
x1, x2, x3 = np.split(x, [3, 5])
print(x1, x2, x3)
[1 2 3] [99 99] [3 2 1]
```

Notice that N split points lead to N + 1 subarrays. The related functions np.hsplit and np.vsplit are similar

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

upper, lower = np.vsplit(grid, [2])
print(upper)
print(lower)

[[0 1 2 3]
[4 5 6 7]]
[[ 8 9 10 11]
[12 13 14 15]]

left, right = np.hsplit(grid, [2])
print(left)
print(right)

[[ 0 1]
[ 4 5]
[ 8 9]
[12 13]]
[[ 2 3]
[ 6 7]
[10 11]
[14 15]]
```

## Computation on NumPy Arrays: Universal Functions

### Introducing UFuncs

NumPy provides a convenient interface into just this kind of statically typed, compiled routine. This is known as a vectorized operation.

Vectorized operations in NumPy are implemented via ufuncs, whose main purpose is to quickly execute repeated operations on values in NumPy arrays. Ufuncs are extremely flexible—before we saw an operation between a scalar and an array, but we can also operate between two arrays

### Exploring NumPy's UFuncs

Ufuncs exist in two flavors: unary ufuncs, which operate on a single input, and binary ufuncs, which operate on two inputs. We'll see examples of both these types of functions here.

### Array arithmetic

NumPy's ufuncs make use of Python's native arithmetic operators. The standard addition, subtraction, multiplication, and division can all be used.

```
x = np.arange(4)
print("x =", x)
print("x + 5 =", x + 5)
print("x - 5 =", x - 5)
print("x * 2 =", x * 2)
```

| Operator | Equivalent ufunc | Description |
|---|---|---|
| + | np.add | Addition (e.g., 1 + 1 = 2) |
| - | np.subtract | Subtraction (e.g., 3 - 2 = 1) |
| - | np.negative | Unary negation (e.g., -2) |
| * | np.multiply | Multiplication (e.g., 2 * 3 = 6) |
| / | np.divide | Division (e.g., 3 / 2 = 1.5) |
| // | np.floor_divide | Floor division (e.g., 3 // 2 = 1) |
| ** | np.power | Exponentiation (e.g., 2 ** 3 = 8) |
| % | np.mod | Modulus/remainder (e.g., 9 % 4 = 1) |

### Absolute value

Just as NumPy understands Python's built-in arithmetic operators, it also understands Python's built-in absolute value function.

```
np.abs()
np.absolute()
x = np.array([-2, -1, 0, 1, 2])
abs(x)
```

array([2, 1, 0, 1, 2])

The corresponding NumPy ufunc is np.absolute, which is also available under the alias np.abs

np.absolute(x)

array([2, 1, 0, 1, 2])

np.abs(x)

array([2, 1, 0, 1, 2])

**Specialized ufuncs**

NumPy has many more ufuncs available like

- Hyperbolic trig functions,
- Bitwise arithmetic,
- Comparison operators,
- Conversions from radians to degrees,
- Rounding and remainders, and much more

**Aggregates**

To reduce an array with a particular operation, we can use the reduce method of any ufunc. A reduce repeatedly applies a given operation to the elements of an array until only a single result remains.

```
x = np.arange(1, 6)
np.add.reduce(x)
```

Similarly, calling reduce on the multiply ufunc results in the product of all array elements

```
np.multiply.reduce(x)
120
```