

# Lecture 8

## Numpy

NumPy stands for Numerical Python, it is a Python library used for working with arrays.

In Python we have lists that serve the purpose of arrays, but they are slow to process.

NumPy aims to provide an array object that is up to 50x faster than traditional Python lists.

The array object in NumPy is called ndarray, it provides a lot of supporting functions that make working with ndarray very easy.

Arrays are very frequently used in data science, where speed and resources are very important.

## Why is NumPy Faster Than Lists?

NumPy arrays are stored at one continuous place in memory unlike lists, so processes can access and manipulate them very efficiently.

This behavior is called locality of reference in computer science.

This is the main reason why NumPy is faster than lists. Also it is optimized to work with latest CPU architectures.

NumPy is a Python library and is written partially in Python, but most of the parts that require fast computation are written in C or C++.

## Import NumPy

Once NumPy is installed, import it in your applications by adding the import keyword:

```
In [1]: import numpy
```

NumPy is usually imported under the np alias.

```
In [2]: import numpy as np
```

## Checking NumPy Version

The version string is stored under **version** attribute.

```
In [3]: import numpy as np  
print(np.__version__)
```

1.24.3

## Create a NumPy ndarray Object

NumPy is used to work with arrays. The array object in NumPy is called ndarray.

We can create a NumPy ndarray object by using the `array()` function.

```
In [4]: import numpy as np

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

print(arr)

print(type(arr))
```

```
[1 2 3 4 5]
<class 'numpy.ndarray'>
```

To create an ndarray, we can pass a list, tuple or any array-like object into the `array()` method, and it will be converted into an ndarray:

```
In [5]: import numpy as np

arr = np.array((1, 2, 3, 4, 5))

print(arr)
```

```
[1 2 3 4 5]
```

## Dimensions in Arrays

A dimension in arrays is one level of array depth (nested arrays).

### 0-D Arrays

0-D arrays, or Scalars, are the elements in an array. Each value in an array is a 0-D array.

```
In [6]: import numpy as np

arr = np.array(42)

print(arr)
```

```
42
```

### 1-D Arrays

An array that has 0-D arrays as its elements is called uni-dimensional or 1-D array.

These are the most common and basic arrays.

```
In [7]: import numpy as np

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

```
print(arr)
```

```
[1 2 3 4 5]
```

## 2-D Arrays

An array that has 1-D arrays as its elements is called a 2-D array.

These are often used to represent matrix or 2nd order tensors.

```
In [8]: import numpy as np

arr = np.array([[1, 2, 3], [4, 5, 6]])

print(arr)

[[1 2 3]
 [4 5 6]]
```

## Check Number of Dimensions

NumPy Arrays provides the `ndim` attribute that returns an integer that tells us how many dimensions the array have.

```
In [9]: import numpy as np

a = np.array(42)
b = np.array([1, 2, 3, 4, 5])
c = np.array([[1, 2, 3], [4, 5, 6]])
d = np.array([[[1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4, 5, 6]]])

print(a.ndim)
print(b.ndim)
print(c.ndim)
print(d.ndim)

0
1
2
3
```

## Access Array Elements

Array indexing is the same as accessing an array element.

You can access an array element by referring to its index number.

The indexes in NumPy arrays start with 0, meaning that the first element has index 0, and the second has index 1 etc.

```
In [10]: import numpy as np

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

print(arr[0])
```

1

## Access 2-D Arrays

To access elements from 2-D arrays we can use comma separated integers representing the dimension and the index of the element.

Think of 2-D arrays like a table with rows and columns, where the dimension represents the row and the index represents the column.

```
In [11]: import numpy as np

arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])

print('2nd element on 1st row: ', arr[0, 1])
```

2nd element on 1st row: 2

## Negative Indexing

Use negative indexing to access an array from the end.

```
In [12]: import numpy as np

arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])

print('Last element from 2nd dim: ', arr[1, -1])
```

Last element from 2nd dim: 10

## Slicing arrays

Slicing in python means taking elements from one given index to another given index.

We pass slice instead of index like this: [start:end].

We can also define the step, like this: [start:end:step].

If we don't pass start its considered 0

If we don't pass end its considered length of array in that dimension

If we don't pass step its considered 1

```
In [13]: import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7])

print(arr[1:5])
```

[2 3 4 5]

```
In [14]: print(arr[4:])
```

[5 6 7]

```
In [15]: print(arr[:4])
```

```
[1 2 3 4]
```

```
In [16]: print(arr[-3:-1])
```

```
[5 6]
```

```
In [17]: print(arr[1:5:2])
```

```
[2 4]
```

```
In [18]: print(arr[::-2])
```

```
[1 3 5 7]
```

## Slicing 2-D Arrays

```
In [19]: import numpy as np
```

```
arr = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]])
```

```
print(arr[1, 1:4])
```

```
[7 8 9]
```

```
In [20]: print(arr[0:2, 2])
```

```
[3 8]
```

```
In [21]: print(arr[0:2, 1:4])
```

```
[[2 3 4]
 [7 8 9]]
```

## Data Types in NumPy

NumPy has some extra data types, and refer to data types with one character, like i for integers, u for unsigned integers etc.

Below is a list of all data types in NumPy and the characters used to represent them.

- i - integer
- b - boolean
- u - unsigned integer
- f - float
- c - complex float
- m - timedelta
- M - datetime
- O - object
- S - string
- U - unicode string
- V - fixed chunk of memory for other type ( void )

## Checking the Data Type of an Array

The NumPy array object has a property called `dtype` that returns the data type of the array:

```
In [22]: import numpy as np

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

print(arr.dtype)

int32
```

```
In [23]: import numpy as np

arr = np.array(['apple', 'banana', 'cherry'])

print(arr.dtype)

<U6
```

## Creating Arrays With a Defined Data Type

We use the `array()` function to create arrays, this function can take an optional argument: `dtype` that allows us to define the expected data type of the array elements:

```
In [24]: import numpy as np

arr = np.array([1, 2, 3, 4], dtype='S')

print(arr)
print(arr.dtype)

[b'1' b'2' b'3' b'4']
|S1
```

## Converting Data Type on Existing Arrays

The best way to change the data type of an existing array, is to make a copy of the array with the `astype()` method.

The `astype()` function creates a copy of the array, and allows you to specify the data type as a parameter.

The data type can be specified using a string, like 'f' for float, 'i' for integer etc. or you can use the data type directly like float for float and int for integer.

```
In [25]: import numpy as np

arr = np.array([1.1, 2.1, 3.1])

newarr = arr.astype('i')

print(newarr)
print(newarr.dtype)
```

```
[1 2 3]
int32
```

```
In [26]: import numpy as np

arr = np.array([1, 0, 3])

newarr = arr.astype(bool)

print(newarr)
print(newarr.dtype)

[ True False  True]
bool
```

## NumPy Array Copy vs View

The main difference between a copy and a view of an array is that the copy is a new array, and the view is just a view of the original array.

The copy owns the data and any changes made to the copy will not affect original array, and any changes made to the original array will not affect the copy.

The view does not own the data and any changes made to the view will affect the original array, and any changes made to the original array will affect the view.

```
In [27]: import numpy as np

arr = np.array([1, 2, 3, 4, 5])
x = arr.copy()
arr[0] = 42

print(arr)
print(x)

[42  2  3  4  5]
[1  2  3  4  5]
```

```
In [28]: import numpy as np

arr = np.array([1, 2, 3, 4, 5])
x = arr.view()
arr[0] = 42

print(arr)
print(x)

[42  2  3  4  5]
[42  2  3  4  5]
```

## Check if Array Owns its Data

As mentioned above, copies owns the data, and views does not own the data, but how can we check this?

Every NumPy array has the attribute `base` that returns `None` if the array owns the data.

Otherwise, the base attribute refers to the original object.

```
In [29]: import numpy as np

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

x = arr.copy()
y = arr.view()

print(x.base)
print(y.base)
```

None

[1 2 3 4 5]

- The copy returns None.
- The view returns the original array.

## Shape of an Array

The shape of an array is the number of elements in each dimension.

### Get the Shape of an Array

NumPy arrays have an attribute called shape that returns a tuple with each index having the number of corresponding elements.

Integers at every index tells about the number of elements the corresponding dimension has.

```
In [30]: import numpy as np

arr = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])

print(arr.shape)

(2, 4)
```

## Reshaping arrays

Reshaping means changing the shape of an array.

The shape of an array is the number of elements in each dimension.

By reshaping we can add or remove dimensions or change number of elements in each dimension.

```
In [31]: import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])

newarr = arr.reshape(4, 3)

print(newarr)
```

```
[[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]
```

```
In [32]: import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])

newarr = arr.reshape(2, 3, 2)

print(newarr)

[[[ 1  2]
  [ 3  4]
  [ 5  6]]

 [[ 7  8]
  [ 9 10]
  [11 12]]]
```

## Returns Copy or View?

```
In [33]: import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7, 8])

print(arr.reshape(2, 4).base)

[1 2 3 4 5 6 7 8]
```

The example above returns the original array, so it is a view.

## Unknown Dimension

You are allowed to have one "unknown" dimension.

Meaning that you do not have to specify an exact number for one of the dimensions in the reshape method.

Pass -1 as the value, and NumPy will calculate this number for you.

```
In [34]: import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7, 8])

newarr = arr.reshape(2, 2, -1)

print(newarr)

[[[1 2]
  [3 4]]

 [[5 6]
  [7 8]]]
```

## Flattening the arrays

Flattening array means converting a multidimensional array into a 1D array.

We can use `reshape(-1)` to do this.

```
In [35]: import numpy as np

arr = np.array([[1, 2, 3], [4, 5, 6]])

newarr = arr.reshape(-1)

print(newarr)

[1 2 3 4 5 6]
```

## Iterating Arrays

Iterating means going through elements one by one.

As we deal with multi-dimensional arrays in numpy, we can do this using basic for loop of python.

If we iterate on a 1-D array it will go through each element one by one.

```
In [36]: import numpy as np

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

for x in arr:
    print(x)

1
2
3
```

## Iterating 2-D Arrays

In a 2-D array it will go through all the rows.

```
In [37]: import numpy as np

arr = np.array([[1, 2, 3], [4, 5, 6]])

for x in arr:
    print(x)

[1 2 3]
[4 5 6]
```

```
In [38]: import numpy as np

arr = np.array([[1, 2, 3], [4, 5, 6]])

for x in arr:
    for y in x:
        print(y)
```

1  
2  
3  
4  
5  
6

## Iterating Arrays Using nditer()

The function `nditer()` is a helping function that can be used from very basic to very advanced iterations. It solves some basic issues which we face in iteration, let's go through it with examples.

Iterating on Each Scalar Element In basic for loops, iterating through each scalar of an array we need to use `n` for loops which can be difficult to write for arrays with very high dimensionality.

```
In [39]: import numpy as np

arr = np.array([[1, 2], [3, 4]], [[5, 6], [7, 8]])

for x in np.nditer(arr):
    print(x)
```

1  
2  
3  
4  
5  
6  
7  
8

## Searching Arrays

You can search an array for a certain value, and return the indexes that get a match.

To search an array, use the `where()` method.

```
In [40]: import numpy as np

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

x = np.where(arr == 4)

print(x)
```

```
(array([3, 5, 6], dtype=int64),)
```

Which means that the value 4 is present at index 3, 5, and 6

```
In [41]: import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7, 8])

x = np.where(arr%2 == 0)
```

```
print(x)
(array([1, 3, 5, 7], dtype=int64),)
```

## Search Sorted

There is a method called `searchsorted()` which performs a binary search in the array, and returns the index where the specified value would be inserted to maintain the search order.

The `searchsorted()` method is assumed to be used on sorted arrays.

```
In [42]: import numpy as np
arr = np.array([6, 7, 8, 9])
x = np.searchsorted(arr, 7)
print(x)
```

1

## Search From the Right Side

By default the left most index is returned, but we can give `side='right'` to return the right most index instead.

```
In [43]: import numpy as np
arr = np.array([6, 7, 8, 9])
x = np.searchsorted(arr, 7, side='right')
print(x)
```

2

## Sorting Arrays

Sorting means putting elements in an ordered sequence.

Ordered sequence is any sequence that has an order corresponding to elements, like numeric or alphabetical, ascending or descending.

The NumPy ndarray object has a function called `sort()`, that will sort a specified array.

```
In [44]: import numpy as np
arr = np.array([3, 2, 0, 1])
print(np.sort(arr))
```

[0 1 2 3]

Note: This method returns a copy of the array, leaving the original array unchanged.

```
In [45]: import numpy as np

arr = np.array(['banana', 'cherry', 'apple'])

print(np.sort(arr))

['apple' 'banana' 'cherry']
```

```
In [46]: import numpy as np

arr = np.array([True, False, True])

print(np.sort(arr))

[False  True  True]
```

## Sorting a 2-D Array

If you use the `sort()` method on a 2-D array, both arrays will be sorted:

```
In [47]: import numpy as np

arr = np.array([[3, 2, 4], [5, 0, 1]])

print(np.sort(arr))

[[2 3 4]
 [0 1 5]]
```

## Filtering Arrays

Getting some elements out of an existing array and creating a new array out of them is called filtering.

In NumPy, you filter an array using a boolean index list.

If the value at an index is `True` that element is contained in the filtered array, if the value at that index is `False` that element is excluded from the filtered array.

```
In [48]: import numpy as np

arr = np.array([41, 42, 43, 44])

x = [True, False, True, False]

newarr = arr[x]

print(newarr)

[41 43]
```

## Creating the Filter Array

In the example above we hard-coded the `True` and `False` values, but the common use is to create a filter array based on conditions.

```
In [49]: import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7])

# Create an empty list
filter_arr = []

# go through each element in arr
for element in arr:
    # if the element is completely divisible by 2, set the value to True, otherwise False
    if element % 2 == 0:
        filter_arr.append(True)
    else:
        filter_arr.append(False)

newarr = arr[filter_arr]

print(filter_arr)
print(newarr)
```

[False, True, False, True, False, True, False]  
[2 4 6]

## Creating Filter Directly From Array

The above example is quite a common task in NumPy and NumPy provides a nice way to tackle it.

We can directly substitute the array instead of the iterable variable in our condition and it will work just as we expect it to.

```
In [50]: import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7])

filter_arr = arr % 2 == 0

newarr = arr[filter_arr]

print(filter_arr)
print(newarr)
```

[False True False True False True False]  
[2 4 6]

In [ ]: