"Introduction to Numpy."

March, 2021

Introduction:

NumPy, *Numerical Python*, is an open source Python library for scientific computing in Python. It is a package consisting of multidimensional array objects and a collection of routines for processing those array objects (mathematical, logical, shape manipulation, sorting, selecting, I/O, discrete Fourier transforms, basic linear algebra, basic statistical operations, random simulation and much more).

NumPy is a developed by a many contributors around the globe. There are several communication channels to learn, share your knowledge and connect with others within the NumPy community. You are strongly encouraged to be part of this community.

Why NumPy?

1. Memory consumption

A NumPy array consumes *less memory* than a Python list:

```
import numpy as np
import sys

rng= range(1000)
print("Size of each element inside the list in bytes: ",sys.getsizeof(rng))
print("Size of the entire list in bytes: ",sys.getsizeof(rng)*len(rng))


arr= np.arange(1000)
print("Size of each element inside the Numpy array in bytes: ",arr.itemsize)
print("Size of the entire Numpy array in bytes: ",arr.size*arr.itemsize)
```

2. Speed of execution

Because NumPy array consumes less memory, it allows the code to be optimized and run faster.

```
import numpy
import time

size = 1000000

# declaring lists
list1 = range(size)
list2 = range(size)

# declaring arrays
array1 = numpy.arange(size)
array2 = numpy.arange(size)


start = time.time()
resultantList = [(a * b) for a, b in zip(list1, list2)]
end =  time.time()
print(f"Time taken by Lists to perform multiplication: {end - start} seconds")

start = time.time()
resultantArray = array1 * array2
end = time.time()
print(f"Time taken by NumPy Arrays to perform multiplication: {end - start} seconds")
```

Installation:

There many ways to install NumPyon your computer. The simplest way is:

- Install the Anaconda distribution. Anaconda will bring you all th necessary packages and dependencies.
- Then, use Anaconda Navigator to start JupyterLab or one of the Python editors.

What's ndarray?

NumPy's array class is called **ndarray**, also known as array. Keep in mind that numpy.array is **not the same** as the Standard Python Library class array.array.

Here are the most important attributes of an ndarray object are:

ndarray.ndim	*the **number of axes** (dimensions).*
ndarray.shape	*a tuple of integers indicating the **size of the array in each dimension**.*
ndarray.size	*the total **number of elements**.*
ndarray.dtype	*an object describing the **type** of the elements.*
ndarray.itemsize	*the **size in bytes** of each element.*
ndarry.data	*the **actual** elements.*

Array creation:

An array is a central data structure of the NumPy library. This is why there are many different mechanisms for creating arrays. Let's explore *some* of them:

1. Conversion from other Python structures (e.g., lists, tuples):

```
>>> import numpy as np
>>> a = np.array([2,3,4])
>>> a
array([2, 3, 4])
>>>
>>> a.dtype
dtype('int64')
>>>
>>> b = np.array([1.2, 3.5, 5.1])
>>>
>>> b.dtype
dtype('float64')
>>>
>>> a
array([2. , 3. , 1.4])
>>>
>>> a.dtype
dtype('float64')
>>>
```

2. Intrinsic numpy array creation objects (e.g., arange, ones, zeros, etc…):

Often, the elements of an array are originally unknown, but its size is known. Also, manually creating an array containing 3 or dimensions, with 3 or more elements by dimension is not easy. Hence, NumPy offers several functions to create arrays with initial placeholder content.

2.0. empty: Create a new array of given shape and type, with uninitialized (*arbitrary*) data:

```
In [1]: np.empty((2, 2, 4), int)
Out[1]:
array([[[94265644104544,              0,    489626271861,    433791696995],
        [  498216206368,    468151435365,    476741369968,    416611827826]],

       [[  450971566194,    519691042924,    502511173664,    416611827822],
        [  416611827830,    463856468073,    420906795105,    433791697004]]])
```

2.1. zeros: Create a new array of given shape and type, filled with zeros:

```
>>> # CREATE ARRAY FILLED WITH ZEROS
>>> np.zeros((3,3,3), dtype=np.int8)
Out >>>
array([[[0, 0, 0],
        [0, 0, 0],
        [0, 0, 0]],

       [[0, 0, 0],
        [0, 0, 0],
        [0, 0, 0]],

       [[0, 0, 0],
        [0, 0, 0],
        [0, 0, 0]]], dtype=int8)
```

2.2. ones: Create a new array of given shape and type, filled with ones.

```
>>> # CREATE ARRAY FILLED WITH ONES
>>> np.ones((2,3))
Out >>>
array([[1., 1., 1.],
       [1., 1., 1.]])

>>>
```

2.3. eye: Create a 2-D array with ones on the diagonal and zeros elsewhere.

```
>>> # CREATE ARRAY WITH THE DIAGONAL FILLED WITH ONES
>>> np.eye(5)
Out >>>
array([[1., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0.],
       [0., 0., 1., 0., 0.],
       [0., 0., 0., 1., 0.],
       [0., 0., 0., 0., 1.]])

>>>
```

2.4. full: Create a new array of given shape and type, filled with specified value.

```
>>> # CREATE ARRAY FILLED WITH 4 EVERYWHERE
>>> np.full((5,6), 4)
Out >>>
array([[4, 4, 4, 4, 4, 4],
       [4, 4, 4, 4, 4, 4],
       [4, 4, 4, 4, 4, 4],
       [4, 4, 4, 4, 4, 4],
       [4, 4, 4, 4, 4, 4]])
```

2.5. arange: Create a new array containing evenly spaced values within a given interval.

np.arange allow you to define the **size** of the **step**.

```
>>> # CREATE AN ARRAY CONTAINING 48 ELEMENTS
>>> # CHANGE THE SHAPE, WE WANT 2 DIMENSIONS
>>> np.arange(48).reshape(8,6)
Out >>>
array([[ 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, 25, 26, 27, 28, 29],
       [30, 31, 32, 33, 34, 35],
       [36, 37, 38, 39, 40, 41],
       [42, 43, 44, 45, 46, 47]])

>>>
```

2.5. linspace: Create a new array containing evenly spaced numbers over a specified interval.

np.linspace allow you to define the **number** of **steps**.

```
>>> # CREATE ARRAY CONTAINING 20 ELEMENTS BETWEEN 10 AND 20, EVENLY SPACED
>>> np.linspace(10, 20, 20)
Out >>>
array([10.        , 10.52631579, 11.05263158, 11.57894737, 12.10526316,
       12.63157895, 13.15789474, 13.68421053, 14.21052632, 14.73684211,
       15.26315789, 15.78947368, 16.31578947, 16.84210526, 17.36842105,
       17.89473684, 18.42105263, 18.94736842, 19.47368421, 20.        ])
```

Nota Bene:

> We cannot see all the available functions. Do not forget, the `help` function is your friend. Use it (`help(np.array)`) to discover all the other interesting functions that can be used with array.

Indexing and slicing:

```
>>> # CREATE AN ARRAY OF 2 DIMENSIONS: (3,5)
>>> arr = np.array([[1,2,3,4,5], [6,7,8,9,10], [11,12,13,14,15]])

>>> arr
Out >>>
array([[ 1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10],
       [11, 12, 13, 14, 15]])

>>> # PRINT THE FIRST COLUMN
>>> arr[:,0]
Out >>> array([ 1,  6, 11])

>>> # PRINT THE FIRST LINE
>>> arr[0,:]
Out >>> array([1, 2, 3, 4, 5])
```

```
>>> # PRINT THE FIRST LINE (SIMPLIFIED)
>>> arr[0]
Out >>> array([1, 2, 3, 4, 5])

>>> # PRINT THE 3RD ELT OF THE 3RD COLUMN
>>> arr[:,2][2]
Out >>> 13

>>> # PRINT THE 3RD COLUMN AND THE FOURTH COLUMN
>>> arr[:,2:4]
Out >>>
array([[ 3,  4],
       [ 8,  9],
       [13, 14]])

>>> # PRINT THE 2ND & 3RD LINES - 3RD & 4TH COLUMNS
>>> arr[1:3,2:4]
Out >>>
array([[ 8,  9],
       [13, 14]])
```

```
>>> # CREATE AN ARRAY OF 35 ELT, CHANGE THE SHAPE TO HAVE 2 DIMENSIONS
>>> arr = np.arange(35).reshape(5,7)

>>> arr
Out >>>
array([[ 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, 25, 26, 27],
       [28, 29, 30, 31, 32, 33, 34]])

>>> # RETRIEVE THE ELEMENTS LOCATED AT: (0,0) (2,1) (4,2)
>>> arr[np.array([0,2,4]), np.array([0,1,2])]
Out >>> array([ 0, 15, 30])

>>> # RETRIEVE THE ELEMENTS LOCATED AT (0,1) (2,1) (4,1)
>>> arr[np.array([0,2,4]), 1]
Out >>> array([ 1, 15, 29])

>>>
```

```
>>> # RETRIEVE THE FIRST, THIRD, FOURTH LINES
>>> arr[np.array([0,2,4])]
Out >>>
array([[ 0,  1,  2,  3,  4,  5,  6],
       [14, 15, 16, 17, 18, 19, 20],
       [28, 29, 30, 31, 32, 33, 34]])

>>>
```

```
>>> arr
Out >>>
array([[ 1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10],
       [11, 12, 13, 14, 15]])

>>> # BOOLEAN ARRAYS USED AS INDICES
>>> arr>10
Out >>>
array([[False, False, False, False, False],
       [False, False, False, False, False],
       [ True,  True,  True,  True,  True]])

>>> # RETURN AN ARRAY CONTAINING THE ELEMENT THAT ARE > 10
>>> arr[arr>10]
Out >>> array([11, 12, 13, 14, 15])
```

Nota Bene:

Indexing and slicing are a very large topic. You are strongly encouraged to learn more by following this link

Adding, removing, and sorting elements:

1. adding element:

For adding an array to another, you can use the np.concatenate function:

```
>>> arr1 = np.arange(10)

>>> arr2 = np.arange(11, 21)

>>> arr1
Out >>> array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

>>> arr2
Out >>> array([11, 12, 13, 14, 15, 16, 17, 18, 19, 20])

>>> np.concatenate((arr1, arr2))
Out >>>
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 11, 12, 13, 14, 15, 16, 17,
       18, 19, 20])
```

```
>>> arr1 = np.array([[1, 2], [3, 4]])

>>> arr2 = np.array([[5, 6], [8,9]])

>>>
>>> np.concatenate((arr1, arr2))
Out >>>
array([[1, 2],
       [3, 4],
       [5, 6],
       [8, 9]])
```

```
>>> # CONCATENATE FOLLOWING THE 2ND DIMENSION
>>> np.concatenate((arr1, arr2), axis=1)
Out >>>
array([[1, 2, 5, 6],
       [3, 4, 8, 9]])
```

2. removing element:

Removing elements from an array is the same as selecting the array minus the elements you want to get rid of. For example:

```
>>> arr2
Out >>>
array([[5, 6],
       [8, 9]])

>>> # KEEP ALL THE ELEMENTS THAT ARE DIFFERENT TO 5
>>> arr2[arr2!=5]
Out >>> array([6, 8, 9])
```

You can also use the function delete:

```
>>> arr2
Out >>>
array([[5, 6],
       [8, 9]])

>>> # DELETE THE FIRST ELEMENT (OF THE FLATTENED ARRAY)
>>> np.delete(arr2, 0)
Out >>> array([6, 8, 9])
```

3. sorting elements:

```
>>> # CREATE AN ARRAY CONTAINING 9 ELEMENTS UNORDERED
>>> # RESHAPE THE ARRAY, WE WANT 2 DIMENSIONS, 3 ELEMENTS PER DIM
>>> arr = np.array([20, 2, 1, 5, 3, 7, 4, 6, 8]).reshape(3, 3)

>>> arr
Out >>>
array([[20,  2,  1],
       [ 5,  3,  7],
       [ 4,  6,  8]])

>>> # SORT ARRAY (BY DEFAULT IT WILL BE SORTED FOLLOWING THE LAST DIM)
>>> np.sort(arr)
Out >>>
array([[ 1,  2, 20],
       [ 3,  5,  7],
       [ 4,  6,  8]])
```

```
>>> # SORT ARRAY FOLLOWING THE 1ST DIM
>>> np.sort(arr, axis=0)
Out >>>
array([[ 4,  2,  1],
       [ 5,  3,  7],
       [20,  6,  8]])
```

Some basic operations on arrays:

1. addition, subtraction, multiplication, division, brodcasting:

Let's consider the 2 following arrays:

```
>>> arr1
Out >>>
array([[1, 2],
       [3, 4]])

>>> arr2
Out >>>
array([[5, 6],
       [8, 9]])
```

Now let's do some basic operations with those 2 arrays:

```
>>> # ADDITION OF ELEMENTS OF arr1 WITH ELEMENTS OF arr2
>>> arr1 + arr2
Out >>>
array([[ 6,  8],
       [11, 13]])
```

```
>>> # SUBSTRATION
>>> arr1 - arr2
Out >>>
array([[-4, -4],
       [-5, -5]])
```

```
>>> # MULTIPLICATION
>>> arr1 * arr2
Out >>>
array([[ 5, 12],
       [24, 36]])
```

```
>>> # DIVISION
>>> arr1 / arr2
Out >>>
array([[0.2       , 0.33333333],
       [0.375     , 0.44444444]])
```

2. maximum, minimum, sum, mean, unique, product, standard deviation:

3. transposing and reshaping:

4. reversing:

Plotting arrays:

Being able to do differents kinds of calculations is good. But you'll often need to plot your results. To do this, there is a library named Matplotlib. Matplotlib is a comprehensive library for creating static, animated, and interactive visualizations in Python.

```
import numpy as np
import matplotlib.pyplot as plt

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

plt.ylabel('y axis')
plt.xlabel("x axis")

plt.plot(x, y)

plt.show()
```

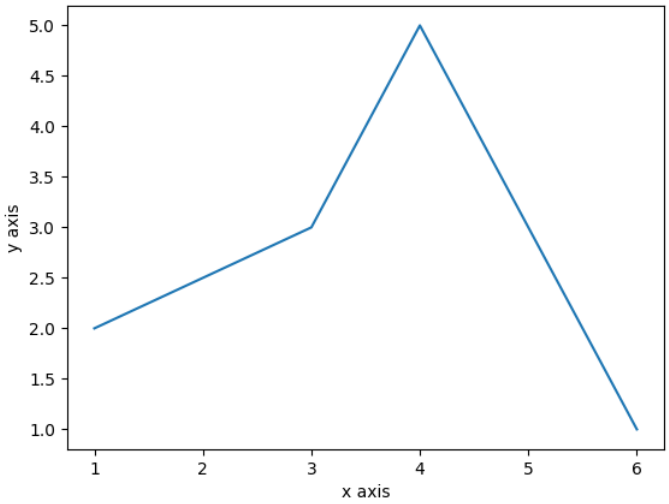

Figure 1: Example of plot

To learn more about NumPy:

To have a better understanding of this topic, we recommend the following links:

- The official NumPy documentation is located at here
- NumPy tutorial, data analysis with Python
- Introduction to Numerical Computing with NumPy.
- Matplotlib tutorial