Course name: **Python programming and analytics by Rahul Sir**

Topic name: **NumPy array (Manav batch)**

Video name: **NumPy arrays 21 sep 2019 Manav batch**

Video length: **1 hour 16 minutes 44 seconds**

Content of the video: What is NumPy Array?
Indexing of Array
Slicing of Array
Multidimensional Sub-array
Creating copies of Array
Splitting of Array

NumPy is a general-purpose array-processing package. It provides a high-performance multidimensional array object, and tools for working with these arrays
It is the fundamental package for scientific computing with Python.

- *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 is called **NUMERICAL PYTHON**

How to import NumPy in Jupyter notebook?

By using the **import function**, you can import the NumPy package in the notebook.

Like, **import** NumPy **as** np

This code is used, if you want an array of random integers in any dimension; here,10 means that you want the array which includes numbers from 1 to 10

```
In [1]: import numpy as np
        np.random.seed(2)  # seed for reproducibility

        x1 = np.random.randint(10, size=6)  # One-dimensional array
        x2 = np.random.randint(10, size=(3, 4))  # Two-dimensional array # (row ,column)
        x3 = np.random.randint(10, size=(3, 4, 5))  # Three-dimensional array #(level, row ,column)

In [2]: x1

Out[2]: array([8, 8, 6, 2, 8, 7])

In [7]: x2

Out[7]: array([[2, 1, 5, 4],
               [4, 5, 7, 3],
               [6, 4, 3, 7]])

In [12]: x3

Out[12]: array([[[8, 1, 5, 9, 8],
                 [9, 4, 3, 0, 3],
                 [5, 0, 2, 3, 8],
                 [1, 3, 3, 3, 7]],

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

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

Seed () function is used to fix the **values**, so that if you run the code again, the values will not change

Array in NumPy is a table of elements (usually numbers), all of the same type, indexed by a tuple of positive integers. In NumPy, number of dimensions of the array is called rank of the array. A tuple of integers giving the size of the array along each dimension is known as shape of the array. Elements in NumPy arrays are accessed by using square brackets and can be initialized by using nested Python Lists.

Creating a NumPy Array

Arrays in NumPy can be created by multiple ways, with various number of Ranks, defining the size of the Array. Arrays can also be created with the use of various data types such as lists, tuples, etc.

```
x4 = np.random.randint(10, size=(4, 4, 5))
print(x4)
print("x4 ndim: ", x4.ndim)
print("x4 shape:", x4.shape)
print("x4 size: ", x4.size)
```

```
[[[3 5 6 0 6]
  [0 3 7 3 8]
  [0 3 8 5 7]
  [5 7 4 1 0]]

 [[6 7 7 8 4]
  [9 2 7 5 9]
  [3 6 4 0 6]
  [1 8 5 1 0]]

 [[5 0 9 8 2]
  [6 9 9 8 2]
  [4 1 1 2 2]
  [6 7 9 8 3]]

 [[9 4 8 4 0]
  [0 4 7 5 9]
  [2 9 6 0 3]
  [6 8 2 1 0]]]
x4 ndim:  3
x4 shape: (4, 4, 5)
x4 size:  80
```

Some more functions 😊

Each array has attributes `ndim` (the number of dimensions), `shape` (the size of each dimension), and `size` (the total size of the array)

In the above picture, you can see we are creating a 3dimensional array X4 = np.random.randint(10, size =(4,4,5)); in the code : size=(4,4,5) first 4 denotes levels of the array; second 4 denotes rows ,5 denotes columns.

Another useful attribute is the `dtype`, the data type of the array

```
print("dtype:", x3.dtype)
```

```
dtype: int32
```

Other attributes include `itemsize`, which lists the size (in bytes) of each array element, and `nbytes`, which lists the total size (in bytes) of the array

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

```
itemsize: 4 bytes
nbytes: 240 bytes
```

In general, we expect that `nbytes` is equal to `total size` and i

Array indexing

You can access an element in an array with the help of index

If you want the element from the end side of the array you can go with negative indexes

```
x1
```

```
array([5, 0, 3, 3, 7, 9])
```

```
x1[0]
```

```
5
```

```
x1[4]
```

```
7
```

To index from the end of the array, you can use negative indices

```
x1[-1]
```

```
9
```

```
x1[-2]
```

```
7
```

In a multi-dimensional array, items can be accessed using a comma-separated list of indices

```
x2
```

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

```
x2[0, 0]
```

```
3
```

```
x2[2, 0]
```

```
1
```

```
x2[2, -1]
```

```
7
```

you can access the items in 2dimensional array by mentioning the index in the form of [rows, columns]. Like, in X2[0,0] means you want to access the item which is placed in 0 indexed row and 0 indexed column that is 3

arrays can also be modified

```
x2[0, 0] = 12
x2
```

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

Same way is used to modify the arrays like you access the item in an array, whatever element in an array you want to modify, the way is in the picture.

Like we want the item in the [0,0] position to be modified to 12

note: in array the elements will be stored as integers even if you add a float data type, in the array it will show as an integer

```
x1[0] = 3.14159  # this will be truncated!
x1
```

```
array([3, 0, 3, 3, 7, 9])
```

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]

In this, **start** is from where you want to start the array slicing; **stop** is till where you want your array to be; **step** is if you want to step/jump any row or column

One-dimensional subarrays

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

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

```
x[:5]  # 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 sub-array
```

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

```
x[::2]  # every other element
```

```
array([0, 2, 4, 6, 8])
```

```
x[:1:2]  # every other element, starting at index 1
```

```
array([0])
```

```
x[:7:3]
```

```
array([0, 3, 6])
```

A potentially confusing case is when the `step` value is negative. In this case, 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])
```

Multi-dimensional subarrays

Multi-dimensional slices work in the same way, with multiple slices separated by commas

```
x2
```

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

```
x2[:2, :3]  # two rows, three columns
```

```
array([[2, 1, 5],
       [4, 5, 7]])
```

```
x2[:3, ::2]  # all rows, every other column
```

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

```
x2[:3, 1::2]
```

```
array([[1, 4],
       [5, 3],
       [4, 7]])
```

Multidimensional subarrays can be created in the same way as 1dimensional subarrays but in the [row, column] format

Finally, subarray dimensions can even be reversed together:

```
x2[::-1, ::-1]
```

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

Accessing array rows and columns:

```
x2
```

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

```
print(x2[:, 0])  # first column of x2
```

```
[2 4 6]
```

```
print(x2[0,])  # first row of x2
```

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

In the case of row access, the empty slice can be omitted for a more compact syntax

```
print(x2[0])  # equivalent to x2[0, :]
```

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

':' here is used if you want all the rows or columns
Like, [:, 0] = [row, column]
Means we want all the rows but only the 0 indexed column, you can also remove the ':', the output will remain the same.

How we can create copies of arrays?
By using the .copy() function.

Reshaping of arrays:

By using the .reshape function, you can change the shape of an array

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

# row vector via reshape
x.reshape((1, 3))
```

```
[1 2 3]

array([[1, 2, 3]])
```

Easy right!

```
# column vector via reshape
x.reshape((3, 1))
```

```
array([[1],
       [2],
       [3]])
```

```
grid = np.arange(1, 10)
grid
```

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

```
grid = np.arange(1, 10).reshape((3, 3))
print(grid)
```

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

Concatenation of arrays:

By using **np.concatenate function, np.vstack, np.hstack**

Np.vstack – to stack/concatenate the arrays vertically

Np.hstack – to stack/concatenate the arrays horizontally

```
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

It can also be used for two-dimensional arrays:

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

```
grid
```

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

```
# concatenate along the first axis
np.concatenate([grid, grid])
```

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

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

# vertically stack the arrays
np.vstack([x, grid])
```

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

np.vstack

```python
x = np.array([1, 2, 3])
grid = np.array([9, 8, 7])

np.hstack([x, grid])
```

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

```python
# horizontally stack the arrays
y = np.array([[99],
              [99]])
np.hstack([grid, y])
```

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

np.hstack

Splitting of arrays:

np.split , np.hsplit, and np.vsplit functions are used to split the arrays

```python
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]
```

X is which array you want to split

we want to split the X into X1, X2, X3

Like, X1 has till 3element and X2 must have from 3 to 5 and the rest in X3

```
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, [3])
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)
```

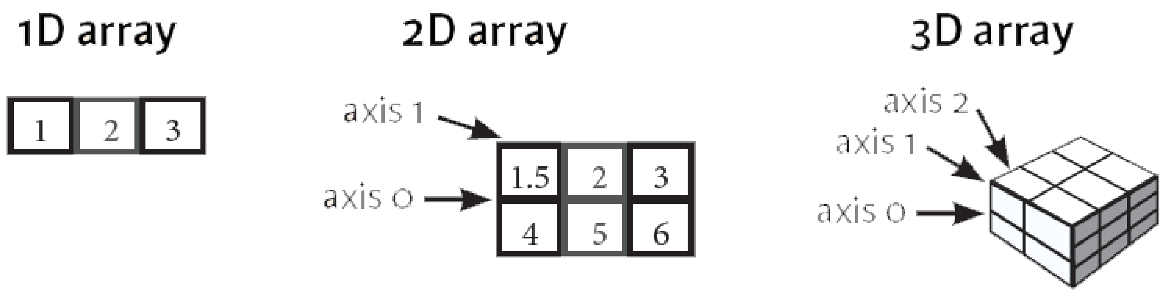
```
[[ 0  1]
 [ 4  5]
 [ 8  9]
 [12 13]]
[[ 2  3]
 [ 6  7]
 [10 11]
 [14 15]]
```

np.vsplit is used to split the array vertically

np.hsplit is used to split the array horizontally

a NumPy array is a central data structure of the numpy library. The library's name is short for "Numeric Python" or "Numerical Python".

In other words, NumPy is a Python library that is the core library for scientific computing in Python. It contains a collection of tools and techniques that can be used to solve on a computer mathematical models of problems in Science and Engineering. One of these tools is a high-performance multidimensional array object that is a powerful data structure for efficient computation of arrays and matrices. To work with these arrays, there's a vast amount of high-level mathematical functions operate on these matrices and arrays.



The array that you see above is, as its name already suggested, a 2-dimensional array: you have rows and columns. The rows are indicated as the "axis 0", while the columns are the "axis 1". The number of the axis goes up accordingly with the number of the dimensions: in 3-D arrays, you'll have an additional "axis 2". Note that these axes are only valid for arrays that have at least 2 dimensions, as there is no point in having this for 1-D arrays;

These axes will come in handy later when you're manipulating the shape of your NumPy arrays.

Don't forget that, in order to work with the np.array() function, you need to make sure that the numpy library is present in your environment. The NumPy library follows an import convention: when you import this library, you have to make sure that you import it as np. By doing this, you'll make sure that other Pythonistas understand your code more easily.

- np.ones(), np.random.random(), np.empty(), np.full() or np.zeros()

the only thing that you need to do in order to make arrays with ones or zeros is pass the shape of the array that you want to make. As an option to np.ones() and np.zeros(), you can also specify the data type. In the case of np.full(), you also have to specify the constant value that you want to insert into the array.

- With np.linspace() and np.arange() you can make arrays of evenly spaced values. The difference between these two functions is that the last value of the three that are passed in the code chunk above designates either the step value for np.linspace() or a number of samples for np.arange(). What happens in the first is that you want, for example, an array of 9 values that lie between 0 and 2. For the latter, you specify that you want an array to start at 10 and per steps of 5, generate values for the array that you're creating.

Remember that NumPy also allows you to create an identity array or matrix with np.eye() and np.identity(). An identity matrix is a square matrix of which all elements in the principal diagonal are ones, and all other elements are zeros. When you multiply a matrix with an identity matrix, the given matrix is left unchanged.

In other words, if you multiply a matrix by an identity matrix, the resulting product will be the same matrix again by the standard conventions of matrix multiplication.

You just make use of the specific help functions that numpy offers to set you on your way:

- Use lookfor() to do a keyword search on docstrings. This is specifically handy if you're just starting out, as the 'theory' behind it all might fade in your memory. The one downside is that you have to go through all of the search results if your query is not that specific, as is the case in the code example below. This might make it even less overviewable for you.
- Use info() for quick explanations and code examples of functions, classes, or modules. If you're a person that learns by doing, this is the way to go! The only downside about using this function is probably that you need to be aware of the module in which certain attributes or functions are in. If you don't know immediately what is meant by that, check out the code example below.

Like this:

```
# Look up info on `mean` with `np.lookfor()`
print(np.lookfor("mean"))

# Get info on data types with `np.info()`
np.info(np.ndarray.dtype)
```

Note that there are two transpose functions. Both do the same; There isn't too much difference. You do have to take into account that T seems more of a convenience function and that you have a lot more flexibility with np.transpose().

```
# Transpose `my_2d_array`
print(np.transpose(my_array))

# Or use `T` to transpose `my_2d_array`
print(my_array.T)
```