Talk On

NumPy

Table of Content

1. About NumPy

- Stands for Numerical Python.
- NumPy is a python library used for working with arrays.
- NumPy array is a collection of homogeneous data types stored in contiguous memory location
- Has function for working in domain of linear algebra, fourier transform, and matrices.
- Written partially in Python, but parts that require fast computation are written in C or C++.
- Source code of NumPy is located at github repository.
    - *https://github.com/numpy/numpy*

NumPy Vs Python List

- NumPy arrays are by default Homogeneous, whereas List can be either Homogeneous or Heterogeneous.
- NumPy arrays have fixed size at creation, unlike python lists which can grow dynamically.
- In NumPy array element wise operation is possible, whereas in list element wise operation is not possible.
- NumPy array can be multidimensional, whereas python list is by default 1 dimensional.
- NumPy are very fast compared Python Lists.

Why NumPy is Fast?

- An array is a collection of homogeneous data types that are stored in contiguous memory locations.
- Vectorized operations are possible in NumPy.
- NumPy package integrates C, C++, and Fortran codes in Python.
  - These programming language have very little execution time compared to Python.

2.NumPy Array manipulation

- Creating arrays
- Attributes of arrays
- Indexing of arrays
- Slicing of arrays
- Reshaping of arrays
- Joining and Splitting of arrays

Creating arrays

Attributes of Numpy arrays

- **ndim:**
  - *Represents the number of dimensions (axes) of the array*
- **shape:**
  - *Given the tuple of integers representing size of the ndarray in each dimension*
- **size:**
  - *Gives the total number of elements in the ndarray. Equals to the product of elements of the result of the attribute **shape***
- **dtype:**
  - *tells the data type of the elements of a Numpy array. In Numpy array, all the elements have the same data type.*
- **itemsize:**
  - *returns the size(in bytes) of each element of a Numpy array.*
- **nbytes:**
  - *Add size(in bytes) of individual elements in the array.*
  - ***Nbytes =** itemsize*size*

Attributes of Numpy arrays: Example

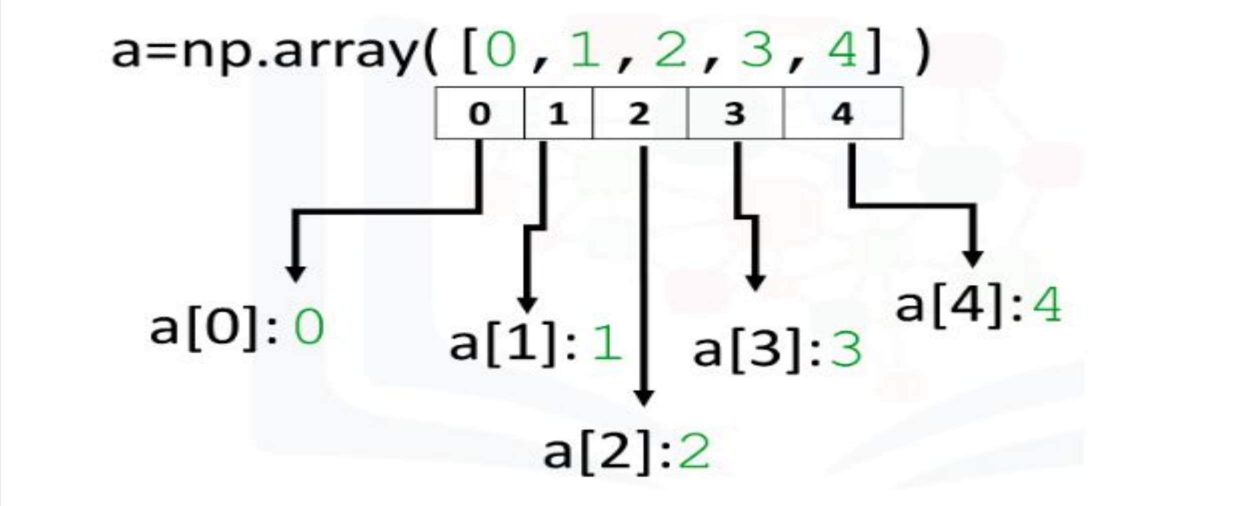
```
In [76]: # create array
         sample_array = np.array([[3, 4, 6], [0, 8, 1]])
         sample_array

Out[76]: array([[3, 4, 6],
                [0, 8, 1]])
```

Now, let's understand mentioned **attributes** of numpy array:

```
In [3]: # ndarray.ndim
        print(f"The dimension of sample array is: {sample_array.ndim}")

        The dimension of sample array is: 2

In [4]: # ndarray.shape
        print(f"The shape of sample array is: {sample_array.shape}")

        The shape of sample array is: (2, 3)

In [5]: # ndarray.size
        print(f"The total number of elements in the sample array is: {sample_array.size}")

        The total number of elements in the sample array is: 6

In [6]: # ndrray.dtype
        print(f"The data type of elements of sample array is: {sample_array.dtype}")

        The data type of elements of sample array is: int64

In [7]: # ndarray.itemsize, returns the size (in bytes) of each element of a Numpy array
        print(f"The size of each element in the numpy array is: {sample_array.itemsize}")

        The size of each element in the numpy array is: 8
```

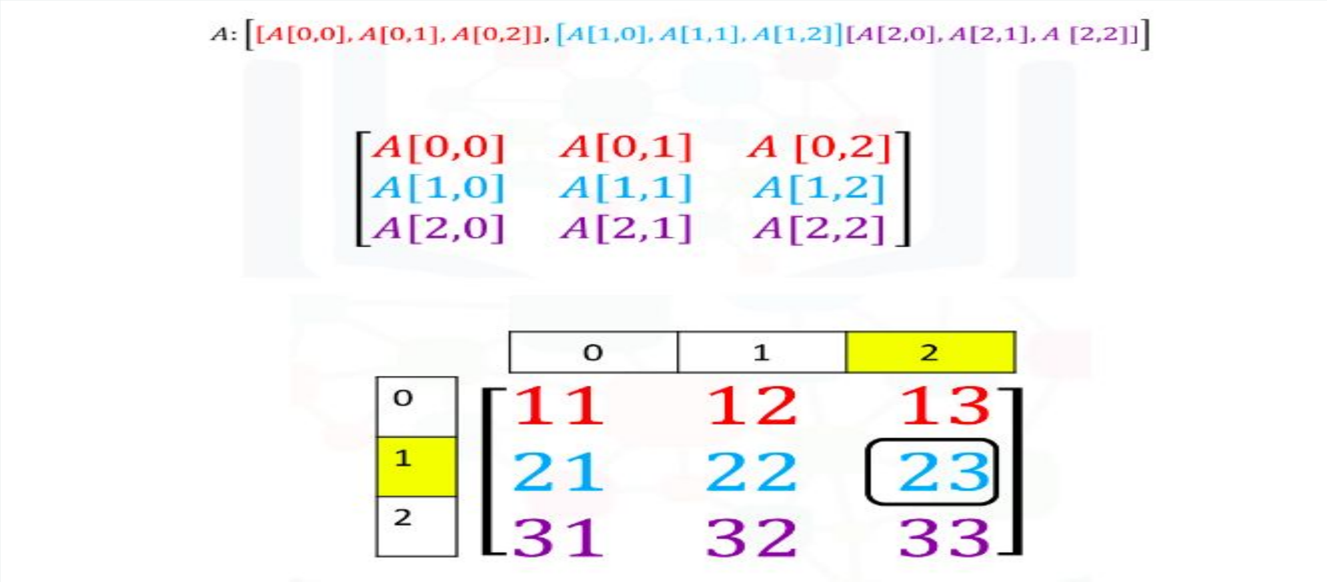
Array Indexing: One-dimensional Array

- Array indexing means accessing element via array.
- ith value can be accessed by specifying desired index in square brackets.



Array Indexing: multi-dimensional array

- Items can be accessed using comma-separated list of indices.

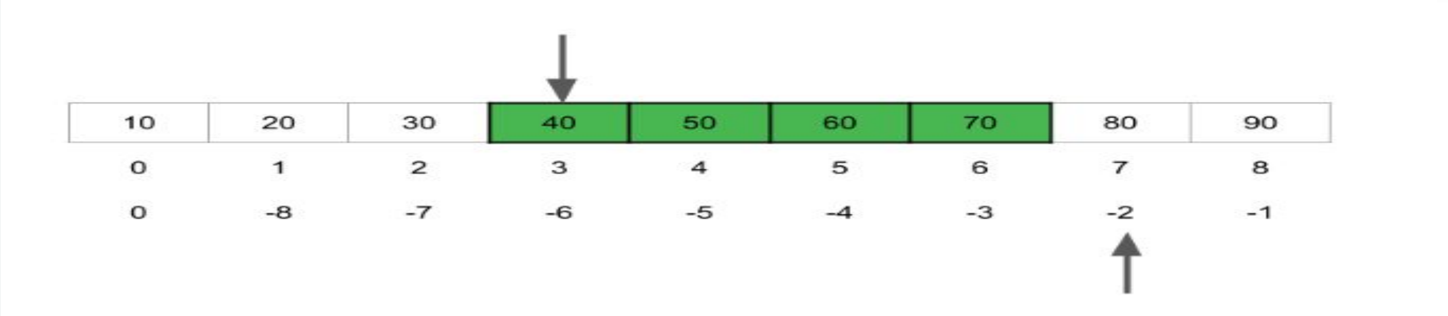


Array Slicing: Accessing Subarrays

- Slicing means taking elements from one given index to another
- Like lists, we can slice numpy array
- We pass slice instead of index like this: **sample_array[start:stop:step]**
    - If we don’t pass start its considered as 0
    - If we don’t pass end its considered length of array in that dimension
    - If we don’t pass step its considered as 1
- Types:
    - Slicing , one-dimensional array (1D array)
    - Slicing, multi-dimensional array (e.g. 2D array)

Slicing, one-dimensional array

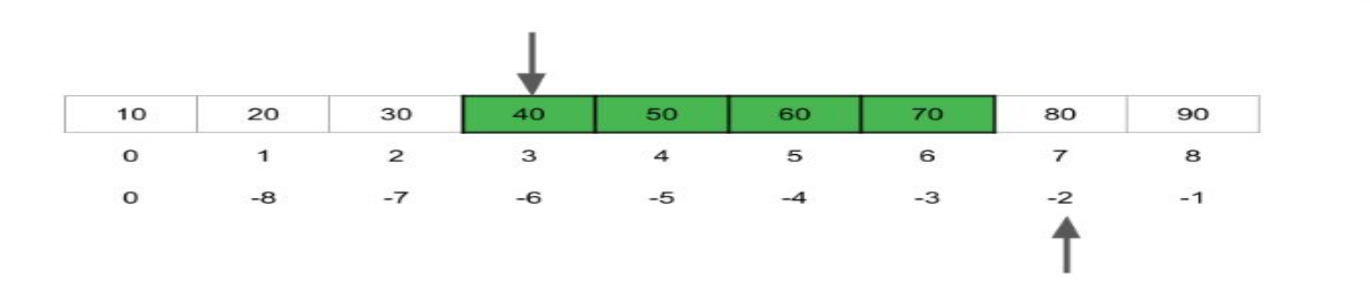
- **Case 1**: Step is positive
    - **Example:** Access subarray *[40, 50, 60, 70]* from array shown:



- using positive index: **array[3:7]**
- using negative index: **array[-6:-2]**
- using both positive and negative index: **array[3:-2]**
- **Note:** end index is exclusive

Slicing, one-dimensional array (Cont...)

- **Case 2**: Step is negative
    - Negative step refer to an index from the end
    - **Example:** Access subarray *[70, 50, 60, 40]* from array shown:



  - using positive index: **array[6:2:-1]**
  - using negative index: **array[-3:-7:-1]**
  - using both positive and negative index: **array[6:-7:-1]**
  - **Note:** end index is exclusive

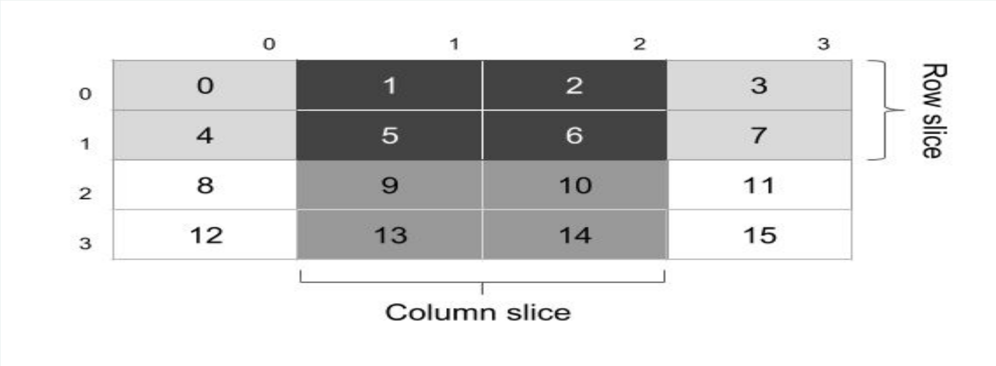
Slicing, one-dimensional array (Code)

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

         # Define array
         a = np.array([10, 20, 30, 40, 50, 60, 70, 80, 90])

         # Case1: Step is positive
         print("***Case 1: Step is positive***")
         print(f"Slicing using positive index: {a[3:7]}")
         print(f"Slicing using negative index: {a[-6:-2]}")
         print(f"Slicing using both positive and negative index: {a[3:-2]}")

         # Case2 : Step is negative
         print("***\nCase 2: Step is negative***")
         print(f"Slicing using positive index: {a[6:2:-1]}")
         print(f"Slicing using negative index: {a[-3:-7:-1]}")
         print(f"Slicing using both positive and negative index: {a[6:-7:-1]}")
```

```
***Case 1: Step is positive***
Slicing using positive index: [40 50 60 70]
Slicing using negative index: [40 50 60 70]
Slicing using both positive and negative index: [40 50 60 70]
***
Case 2: Step is negative***
Slicing using positive index: [70 60 50 40]
Slicing using negative index: [70 60 50 40]
Slicing using both positive and negative index: [70 60 50 40]
```

Slicing, multi-dimensional array

- Consider the case of 2D array.
- Slicing a 2D array is similar to a 1D array.
- Use a comma to separate the row slice and the column slice.
- **Example:** Access subarray *[[1, 2], [5, 6]]* from array shown below:



➢ **Syntax:** array[0:2, 1:3]

Slicing, multi-dimensional array(code)

```
In [13]: # Slicing, 2D array

         # define array
         a = np.array([[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11], [12, 13, 14, 15]])

         # slice array,
         print(f"Slicing subarrays containing row=[row1, row2] and column=[colum1, column2]:\n {a[0:2, 1:3]}")

Slicing subarrays containing row=[row1, row2] and column=[colum1, column2]:
 [[1 2]
 [5 6]]
```

View and Copy of array

```
In [46]: # Copy

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

         # make copy
         copy_arr = arr.copy()

         # display original and copy of an array
         print(f"Original array= {arr}   Copy array= {copy_arr}")

         # verify memory is not shared
         print(f"\nDo original array and view of an array shares memory?\nAns: {np.shares_memory(arr, copy_arr)}")

         # edit index 0 in copy array i.e. from 1 to -1
         copy_arr[0] = -1
         print(f"\nCopy array changed from [1 2 3 4 5] to {copy_arr}")
         print(f"But original array doesnot changed i.e. {arr}")
```

```
Original array= [1 2 3 4 5]   Copy array= [1 2 3 4 5]

Do original array and view of an array shares memory?
Ans: False

Copy array changed from [1 2 3 4 5] to [-1  2  3  4  5]
But original array doesnot changed i.e. [1 2 3 4 5]
```

Subarrays as no-copy view

```python
# Define array
arr = np.array([[0, 1, 2, 3], [4, 5, 6, 7], [8, 9, 10, 11]])
print(f"Original array:\n{arr}")

# Slice
sliced_arr = arr[0:2, 1:3]
print(f"\nmemory address of original array: {id(arr)}")
print(f"memory address of sliced subarray: {id(sliced_arr)}")
print(f"\nDo original array and sliced subarray shares memory?\nAns: {np.shares_memory(arr, sliced_arr)}")

# edit sliced array, i.e. index=[0, 1] of original array
sliced_arr[0][0] = 100
print(f"\nChanges reflected to original array i.e.\n {arr}")
```

```
Original array:
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]

memory address of original array: 139968340233152
memory address of sliced subarray: 139968340353392

Do original array and sliced subarray shares memory?
Ans: True

Changes reflected to original array i.e.
 [[  0 100   2   3]
 [  4   5   6   7]
 [  8   9  10  11]]
```

Reshaping of arrays

- The shape of an array is the number of elements in each dimension
- Reshaping means changing the shape of an array.
- By reshaping we can add or remove dimensions or change the number of elements in each dimension.
- **Example:**
    - *Reshape from 1D to 2D*
    - *Reshape from 1D to 3D*

Example: Reshape from 1D to 2D

Q. Convert the following 1D array with 12 elements into a 2D array

In [19]:
```
# create a array
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])

new_arr = arr.reshape(4, 3)

print(f"array before reshaping:\n {arr}")

print(f"\narray after reshaping to shape (4, 3):\n {new_arr}")
```

```
array before reshaping:
 [ 1  2  3  4  5  6  7  8  9 10 11 12]

array after reshaping to shape (4, 3):
 [[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]
```

Example: Reshape from 1D to 3D

Q. Covert the following 1D array with 12 elements into a 3D array

```
In [20]: # sample array
         arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])

         # reshape to 3D array
         new_arr = arr.reshape(2, 3, 2)

         print(f"array before reshaping to 3D is:\n {arr}")

         print(f"\narray after reshaping to 3D is:\n {new_arr}")
```

```
array before reshaping to 3D is:
 [ 1  2  3  4  5  6  7  8  9 10 11 12]

array after reshaping to 3D is:
 [[[ 1  2]
  [ 3  4]
  [ 5  6]]

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

Can we Reshape into any Shape

- **Yes,** as long as elements required for reshaping are equal in both shapes.
- **Question:**
  - Convert 1D array with 8 elements to a 2D array with shape (3, 3)
  - Is it possible?
- **Answer:**
  - This process will raise an exception as there are 8 elements in 1D and 9 elements in 2D after reshaping

- **Coding example:**

Exception when Reshaping

```
In [21]: arr = np.array([1, 2, 3, 4, 5, 6, 7, 8])

         # reshape to 3x3
         newarr = arr.reshape(3, 3)

         print(newarr)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-21-416b01fa8732> in <module>
      2 
      3 # reshape to 3x3
----> 4 newarr = arr.reshape(3, 3)
      5 
      6 print(newarr)

ValueError: cannot reshape array of size 8 into shape (3,3)
```

As we can see this lead to an exception as expected

Array Joining

- Array Joining means putting content of two or more array in a single array
- In SQL we join tables based on keys, whereas in NumPy we join arrays by axes.
- Array Joining can be done in two ways:
  - **Concatenation**: using ***concatenate()*** function
  - **Stacking**: using ***stack()*** function

- Now, lets understand each of them.

Concatenation

- **Syntax:** **numpy.concatenate(***(a1, a2, ...), axis=0, out=None***)**¶
  - a1, a2, ... : *Sequence of array_like. The array must have the same shape*
  - axis: *Int, optional. axis along which array will be concatenated*
  - out: *ndarray, optional. The destination to place the result*

- We pass sequence of arrays that we want to the **concatenate()** function.

- After concatenating dimensions of resulting array will be same as that of individual array.

- Order of concatenation will be based on the order in which array are passed to the **concatenate()** function.

Concatenation(Cont..)

- **Concatenate 1D array**
- **Concatenate 2D array**
  - Concatenate along row axis (axis=0)
  - Concatenate along column axis (axis=1)
- **Concatenate 3D array**
  - Concatenate along axis=0
  - Concatenate along axis=1
  - Concatenate along axis=2

Concatenate 1D array

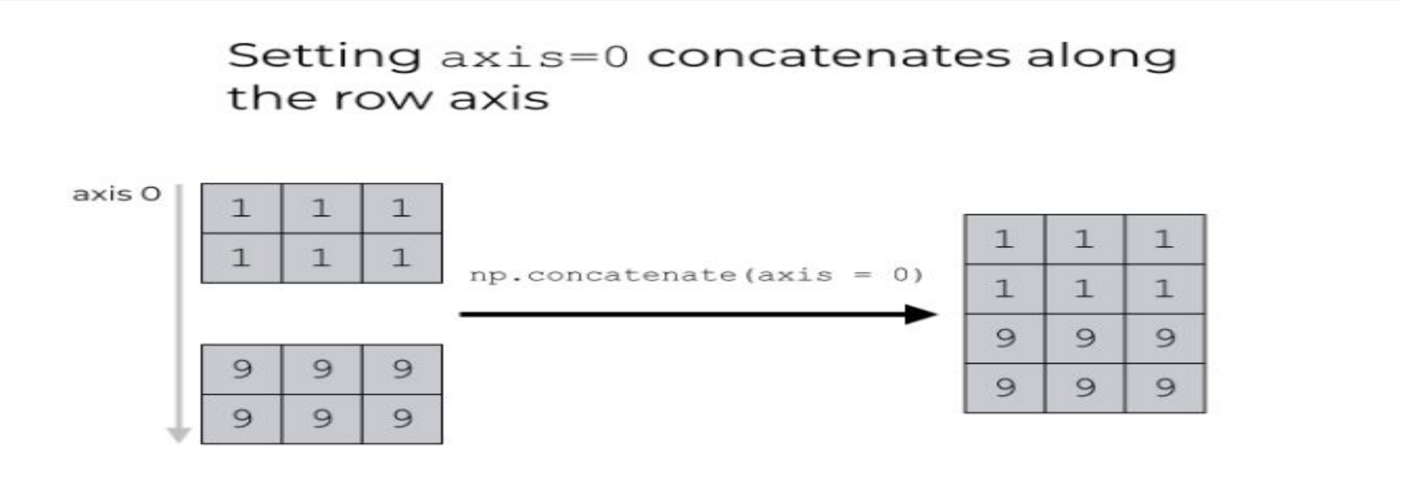
- 1D array only have one axis i.e **axis=0**
- So, concatenation is only along **axis=0**
- **Example:**

```
In [2]: arr1 = np.array([1, 2, 3])

        arr2 = np.array([4, 5, 6])

        # display array
        print(f"Array 1: {arr1}")
        print(f"Array 2: {arr2}")

        arr = np.concatenate((arr1, arr2))

        print(f"Concatenating 1D array: {arr}")
```

```
Array 1: [1 2 3]
Array 2: [4 5 6]
Concatenating 1D array: [1 2 3 4 5 6]
```

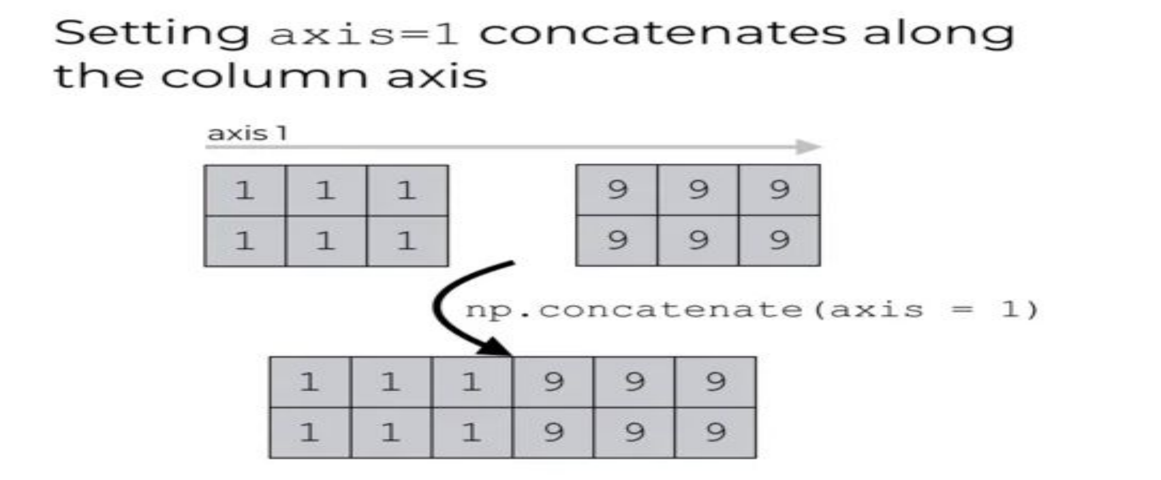
Concatenate 2D array (Along axis=0)

- Setting axis=0 concatenates along the row axis.
- Similar to NumPy **vstack()** i.e. vertical stacking function.
    - *numpy.vstack((arr1, arr2))*



Concatenate 2D array (Along axis=1)

- Setting axis=1 concatenates along the column axis.
- Similar to NumPy **hstack()** i.e. horizontal stacking function.
    - *numpy.hstack((arr1, arr2))*



Stacking

```
In [69]: # Define two array
         arr1 = np.array([[1, 2], [3, 4]])
         arr2 = np.array([[5, 6], [7, 8]])

         # display array
         print(f"array1:\n{arr1}")
         print(f"\narray2:\n{arr2}")

         # stack array
         np.stack((arr1, arr2), axis=0)

         array1:
         [[1 2]
          [3 4]]

         array2:
         [[5 6]
          [7 8]]

Out[69]: array([[[1, 2],
                 [3, 4]],

                [[5, 6],
                 [7, 8]]])
```

Splitting

- Splitting is reverse operations of joining
    - Joining merges multiple arrays into one.
    - Splitting breaks one array into multiple one.
- Can be implemented using numpy functions:
    - ***numpy.split()***
    - ***numpy.array_split()***
    - ***numpy.hsplit()***
    - ***numpy.vsplit()***
- We pass array we want to split and the number of split.
- Sub-arrays obtained after splitting is only view of the original arrays.
    - Changes made to the sub-arrays also reflect changes in the original array

array_split() VS split()

- Both are use to split numpy arrays.
- **numpy.split():**
  - Does not adjust elements when elements are less in source array.
  - Meaning: if the array has less element than required for splitting, it will raise an exception.
- **numpy.array_split():**
  - Adjust elements when elements are less in source array.
  - Meaning: if the array has less element than required for splitting, the last end array is null.
- **Coding Example:**

array_split() vs split() (Example)

Q. Split [1, 2, 3, 4, 5, 6] into four parts

```python
# define array
arr = np.array([1, 2, 3, 4, 5, 6])
print(f"origina array: {arr}")

# split
splitted_arr = np.array_split(arr, 4)
print(f"splitted array using array_split(): {splitted_arr}")
```

```
origina array: [1 2 3 4 5 6]
splitted array using array_split(): [array([1, 2]), array([3, 4]), array([5]), array([6])]
```

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

print("***As Expected split() throws an error:***\n")

split_arr = np.split(arr, 4)
```

```
***As Expected split() throws an error:***

ValueError: array split does not result in an equal division
```

Splitting (via specifying axis name)

- Can specify axis to do the split around.
- For 2D arrays, splitting can be done in two ways:
  - ***Splitting across axis=0***
    - Similar to ***numpy.hsplit()*** for equal division case
  - ***Splitting across axis=1***
    - Similar to ***numpy.vsplit()*** for equal division case

- Now let's understand each of them separately

Splitting 2D array (axis=0)

- **Syntax:** *numpy.array_split(arr, split_num, axis=0)*
- Comparison with ***numpy.vsplit()***
    - Similar to ***numpy.vsplit()*** when splitting results in the equal division.
    - ***numpy.array_split():*** can work with both equal and unequal division case
    - ***numpy.vsplit():***
        - work only in the equal division case
        - raise an exception in the case of unequal division..

- **Example:**

Splitting 2D array (axis=0) Example

```
# define array
arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12], [13, 14, 15], [16, 17, 18]])
print(f"sample array:\n {arr}")

# split via array_split()
print(f"array_split(axis=0) result:\n {np.array_split(arr, 3, axis=0)}")

# split via vstack()
print(f"vsplit() result:\n {np.vsplit(arr, 3)}")
```

```
sample array:
 [[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]
 [13 14 15]
 [16 17 18]]
```

```
array_split(axis=0) result:
 [array([[1, 2, 3],
       [4, 5, 6]]), array([[ 7,  8,  9],
       [10, 11, 12]]), array([[13, 14, 15],
       [16, 17, 18]])]
```

```
vsplit() result:
 [array([[1, 2, 3],
       [4, 5, 6]]), array([[ 7,  8,  9],
       [10, 11, 12]]), array([[13, 14, 15],
       [16, 17, 18]])]
```

Splitting 2D array(axis=1)

- **Syntax:** *numpy.array_split(arr, split_num, axis=1)*
- Comparison with ***numpy.hsplit()***
  - Similar to ***numpy.hsplit()*** when splitting results in the equal division.
  - ***numpy.array_split():*** can work with both equal and unequal division case.
  - ***numpy.hsplit():***
    - work only in the equal division case.
    - raise an exception in the case of unequal division.

- **Example:**

Splitting 2D array (axis=1) Example

```
# define array
arr = np.array([[1, 2, 3], [3, 4, 3], [5, 6, 3], [7, 8, 3]])
print(f"Sample array:\n {arr}")

# Split via array_split with axis=1
print(f"\narray_split(axis=1) result:\n {np.array_split(sample_arr, 3, axis=1)}")

# split via hsplit()
print(f"\nhsplit() result:\n {np.hsplit(sample_arr, 3)}")
```

```
Sample array:
 [[1 2 3]
 [3 4 3]
 [5 6 3]
 [7 8 3]]
```

```
array_split(axis=1) result:
 [array([[1],
       [3],
       [5],
       [7]]), array([[2],
       [4],
       [6],
       [8]]), array([[3],
       [3],
       [3],
       [3]])]
```

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

3.NumPy Universal functions (Ufuncs)

- Looping over the array to perform repeated task like addition, subtraction, etc on each array element are common.
- Computation time to perform such repeated task increases with relatively larger data.
- NumPy makes this faster by using vectorized operations, implemented through **ufuncs**.
- **Types:**
  - ***Unary ufuncs***
  - ***Binary ufuncs***
- **Example:**
  - ***Compare computation time for element-wise multiplication without and with using ufuncs***

Types of Ufuncs

- **Unary ufuncs**
  - Operates on single inputs.
  - **Example:** negation of input array x
- **Binary ufuncs**
  - Operates on two inputs.
  - **Example:** addition of two input array ***x*** and ***y***

- **Operators and Equivalent ufunc:**

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`)

Computation time comparison

Element-Wise multiplication using Ufuncs

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

# Define two numpy array
x1 = np.arange(1, 1000)
x2 = np.arange(1, 1000)
tic = time.process_time()

# Element-wise multiplication using Ufuncs
mul = np.multiply(x1,x2)

# Displyay Element-wise multiplication time using Ufuncs
toc = time.process_time()
print ("elementwise multiplication using Ufuncs" "\n ----- Computation time = " + str(1000*(toc - tic)) + "ms")
```

```
elementwise multiplication using Ufuncs
 ----- Computation time = 0.09755300000025002ms
```

- Using **ufuncs** computation time is less compared to classic python for loops.
- The reason behind less computation time using ufuncs is due to **Vectorization**

Vectorization

- It’s a technique in order to getting rid of explicit ***for loop*** in python.
- Vectorization can be performed using parallelization instruction called **SIMD instruction.**
- SIMD stands for **single instruction multiple data.**
- In python **SIMD** is enabled through Ufuncs.
    - Helps python NumPy to take much better advantage of parallelism to do computation faster.

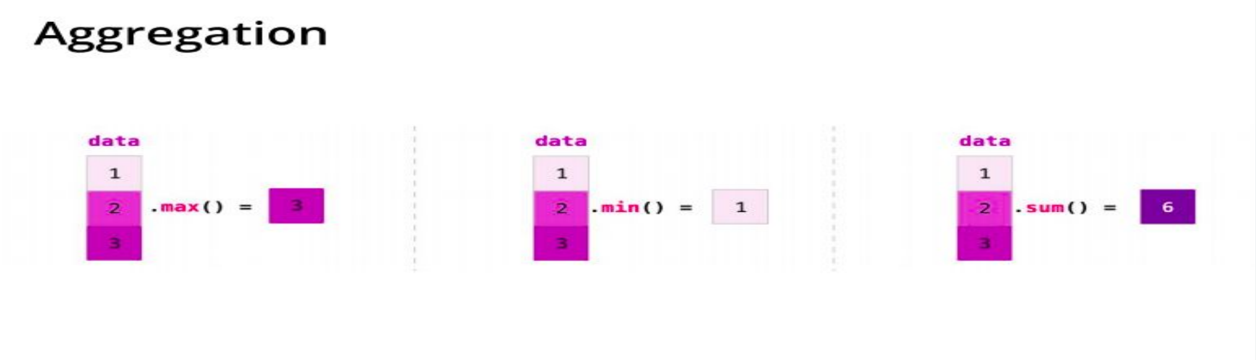
Vectorization in Deep Learning

- Famous in different task such as Classification, Segmentation, Detection, etc.
- Deep Learning demands large number of data that helps model to generalize.
- Large number of training set leads to large number of time to run the code i.e. need to wait long time to get the result.
- So, in deep learning ability to perform **vectorization** has become key skill.

- **Note:**
  - *whenever possible avoid using explicit for loops*

4.Aggregation: Min, Max

- As data analyst, the first step is to explore and understand the data.
- One way to understand the data is to compute summary statistics.
- Mean and standard deviation are most common statistical method to summarize the data.
- These are called **Aggregates.**
- Different useful aggregates are: sum, product, minimum, maximum, etc
- **Visual interpretation of python built-in aggregation:**



NumPy aggregation functions

- Like python NumPy also has built-in aggregation functions.
- Specially designed for working with Numpy arrays.
- **Different NumPy aggregation functions:**

Function Name	NaN-safe Version	Description
`np.sum`	`np.nansum`	Compute sum of elements
`np.prod`	`np.nanprod`	Compute product of elements
`np.mean`	`np.nanmean`	Compute mean of elements
`np.std`	`np.nanstd`	Compute standard deviation
`np.var`	`np.nanvar`	Compute variance
`np.min`	`np.nanmin`	Find minimum value
`np.max`	`np.nanmax`	Find maximum value
`np.argmin`	`np.nanargmin`	Find index of minimum value
`np.argmax`	`np.nanargmax`	Find index of maximum value
`np.median`	`np.nanmedian`	Compute median of elements
`np.percentile`	`np.nanpercentile`	Compute rank-based statistics of elements
`np.any`	N/A	Evaluate whether any elements are true
`np.all`	N/A	Evaluate whether all elements are true

Python VS NumPy aggregation

- **Q. Why to use NumPy aggregate functions when they are already available in Python?**

- **Speed:**
  - NumPy aggregate functions are must faster than Python aggregate functions.
- **Aware of dimensions:**
  - NumPy aggregate functions are aware of dimensions.
  - However python aggregate functions behave differently on multi-dimensional arrays.
- **Note:**
  - ***Whenever possible make use of NumPy version of aggregates when operating on NumPy arrays***

Python VS NumPy aggregation (Cont...)

- **Aware of dimensions:**
    - Consider code snippet as shown:

```
In [28]:  1 array =  np.arange(10).reshape(2,5)
          2 print(array)
          3 print('Summation:',sum(array))

[[0 1 2 3 4]
 [5 6 7 8 9]]
Summation: [ 5  7  9 11 13]
```

    - **Expected output: 45** i.e. *sum of all elements in the array*
    - But output are not according to expectation using python inbuilt **sum()**.
    - **Using numpy sum:**

```
print(f"Result using NumPy sum(): {np.sum(array)}")

Result using NumPy sum(): 45
```

5.Application of NumPy

- **MATLAB Replacement**
  - Can do all sorts of mathematics with NumPy.
  - There are functions for Linear algebra, fourier transform, and so on.
- **Machine Learning**
  - Knowing NumPy you can do some of the stuff with machine learning.
- **NumPy with pandas**
  - Pandas interoperates with NumPy for faster computations.
  - Using both libraries together is a very helpful resource for scientific computation.

Thank you

for your

Valuable time….