SAGAR CHOUKSEY

Numpy

Stands for Numerical Python.

Created by Travis Oliphant in 2005 to make the otherwise slow Python loops fast and memory efficient.

Features :- Speed in (50-100x faster than normal Python lists)
:- Less Memory Occupying
:- Easy Math Operations
:- Used almost everywhere

List vs Numpy Arrays

List	Np Arrays
zed - Slow	Fast (50-100x)
mory - Memory usage More	Less Memory used
culations - uses Smaller level Lists	Large Dataset

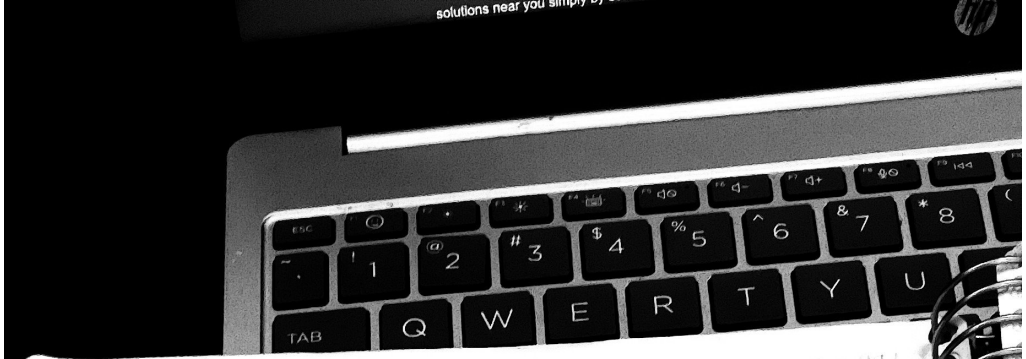
Uses :- Data science
:- ML (AI)
:- Finance
:- Medical Computing
:- Image Processing

Install in your local Machine:

pip3 install numpy
(install in .py)

!pip install numpy
(install in .ipynb)

Pre installed in Google Colab.



The Academic Industry Gap

What Industry Demands

- Messy data and ambiguous problems
- Hands on tools
- Relevant 'things' and projects —— Revision

~~Framework :— Making Academic~~

What Academia Teaches

- Theoritical foundations and algorithms
- clean datasets and controlled environments
- Individual assignments with clear objectives
- Emphasis on accuracy metrices

```
python_list = [1,2,3,4,5]
Print (python_list)
```

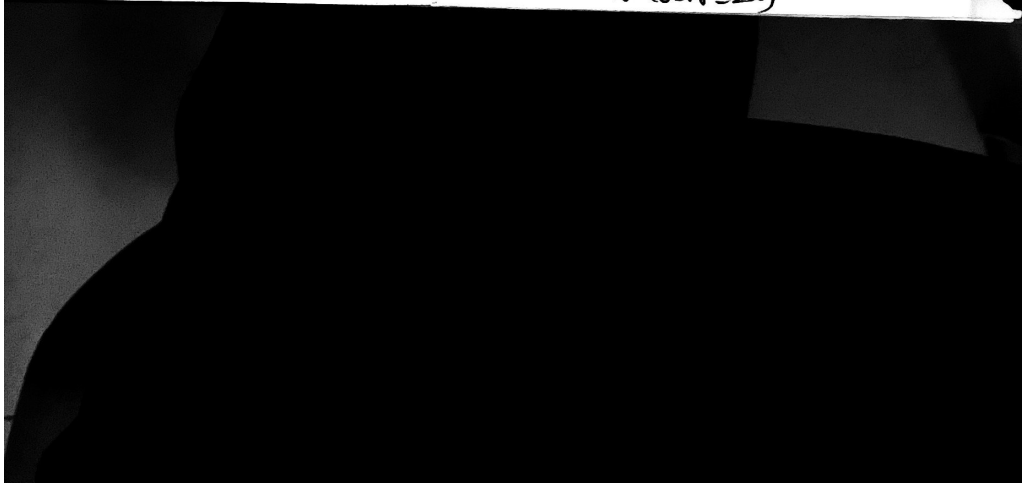
```
import numpy as np
numpy_array = np.array([1,2,3,4,5])
```

← Array in Numpy

1D Array in Numpy =>

2D Array in Numpy →

```
import numpy as np
arr_2d = np.array
    ([[1,2,3],
      [4,5,6],
      [7,8,9]])
print (arr_2d)
```



Pythonic lists to Numpy Arrays

⇒ import numpy as np

⇒ arr = np.array([1,2,3,4])

⇒ print(arr)

creating Arrays with default values

import numpy as np

z_arr = np.zeroes(3)

print(z_arr)

↓

[0. 0. 0.]

Creating Arrays with ones' as default values

import numpy as np

0_arr = np.ones(3)

print(0_arr) ⟿ [1. 1. 1.]

NOTE = WE CAN ALSO DO THE SAME with a 2D System

import numpy as np

ones_array = np.ones((2,3))

print(ones_array)

↳ [[1. 1. 1.]
[1. 1. 1.]]

Filling specific VALUE

(Full Function)

Full(shape, value)

import numpy as np

np.full((2,2), 7)

Creating SEQUENCE OF Numbers in NUMPY

use ⇒ arange(start, stop, step

import numpy as np

arr = np.arange(1,10,2)

print(arr)

[1 3 5 7 9]

Creating Identity Matrixes

Eye Method i.e eye(num)

import numpy as np

i1 = np.eye(3)

print(i1)

→ [[1. 0. 0.]
[0. 1. 0.]
[0. 0. 1.]]

Identity Matrices are special matrices with one's on the diagonal and zeroes elsewhere.

PROPERTIES OF NUMPY ARRAYS and OPERATIONS

→ ① shape, size, types

* ~~Name~~ Number of dimensions!
* warehouse manager. → product ⟹ Quantity
  name ⟹ price.

① shape = Returns the shape (number of rows and colums) in the data as an integer based tuple)

```
* import numpy as np
  arr_2d = np.array([[1, 2, 3],
                     [4, 5, 6]])
  print(arr_2d, shape)
```

→ (2,3) Ans

② SIZE = Total number of elements in an array.

```
import numpy as np
arr = np.array([[10, 20, 30],
                [40, 50, 60]])
print(arr.size) → 6
```

NDIM = Returns the dimensions of the matrix.

```
import numpy as np
arr_1d = np.array([1,2,3])
arr_2d = np.array([[1,2,3],[4,5,6]])
arr_3d = np.array([[[1,2],[3,4],[5,6],[7,8]]])
print(arr_1d.ndim)
print(arr_2d.ndim)
print(arr_3d.ndim
```

1 → 1D Array
2 → 2D Array
3 → 3D Array

④ DTYPE — Datatype of the elements will be returned in a 64 bit format.

```
import numpy as np
arr = np.array([10, 20, 30.5, 40])
print(arr.dtype)
```

→ float64 = This will give me a hint that there's a float value within the list.

) AsType — This essentially helps us do an explicit type casting in Numpy

SYNTAX :- ___.astype(datatype)

eg = import numpy as np

arr = np.array([1.2, 2.5, 3.8])

print(arr.dtype) ——→ <float64>

int_arr = arr.astype('int') ——→ <int64>

means as a 64 bit integer.

MATHEMATICAL OPERATIONS THAT CAN BE PERFORMED ON THE SAID ARRAY

~~Import~~ All Arithmetic options that we can perform on Arrays can be achieved in Numpy.

maths.py ⟹ import numpy as np

```
arr = np.array([10, 20, 30])
print(arr + 5)  ⟶ [15, 25, 35]
print(arr * 2)  ⟶ [20, 40, 60]
print(arr **2)  ⟶ [100, 400, 900]
```

As Numpy allows broadcasting, we don't need loop calling

AGGREGATION FUNCTIONS IN THE NUMPY LIBRARY

FUNCTION	WHAT IT DOES
np.sum(array)	Add all Elements
np.mean(array)	Returns the Average of All Elements
np.std(array)	Returns the Standard Deviation of all Arrays
np.min(array)	Returns the Minimum of the Array Elements
np.max(array)	Returns the Maximum Value of the Array Elements
np.var(array)	Returns the Variance of the Array Elements

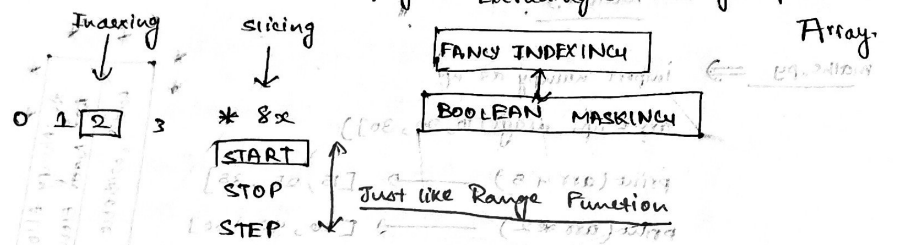
eg—

```
import numpy as np
arr = np.array([10, 20, 30, 40, 50])
print(np.sum(arr))
print(np.mean(arr))
print(np.min(arr))
print(np.max(arr))
print(np.std(arr))
print(np.var(arr))
```

Looking for computer software dealers in Bhubaneshwar? Justdial is the place for you...
solutions near you simply by searching for computer software ...

⊛ INDEXING + SLICING → we can access any particular element at any particular position of the Array — that is known as Slicing.

Slicing → Extracting subsets of a parent Array.



* Fancy Indexing = It is a method of using an array (or list) of indices to select multiple, non-continuous elements

How to use these? — This process involves two main steps

(1) Select the desired subset of data using an index array or a boolean mask.

Apply an Aggregation Function that can then be used

```
- import numpy as np
  data = np.array([10,20,30,40,50])
  indices = np.array([1,4,7,8])
  selected_elements = data[indices]
  print(f"Selected Elements : {selected_elements}")
  avg_value = np.mean(selected_elements)
  print(f"Average of selected elements is : {avg_value}")
```

BOOLEAN MASKING ⟶ It is a powerful Numpy Technique for filtering, selecting, modifying, or counting elements in an array based on specific conditions.

eg—

SYNTAX ⟶ array[index] # 1D Array

array[row][column] # 2D Array

eg- import numpy as np

arr = np.array([10,20,30,40,50])

print(arr[0]) ⟶ 10

print(arr[2]) ⟶ 30

print(arr[-1]) ⟶ 50 (last element)

Index Number as well call it

* Uses '0' based indexing

* Uses -ve indexing as well

arr[start, stop] goes from start to end-1

eg- import numpy as np

arr = np.array([10,20,30,40,50,60])

print(arr[1:5]) ⟶ 20, 30, 40, 50

print(arr[:4]) ⟶ 10, 20, 30, 40

print(arr[::2]) ⟶ 10, 30, 50

print(arr[::-1]) ⟶ 60, 50, 40, 30, 20, 10

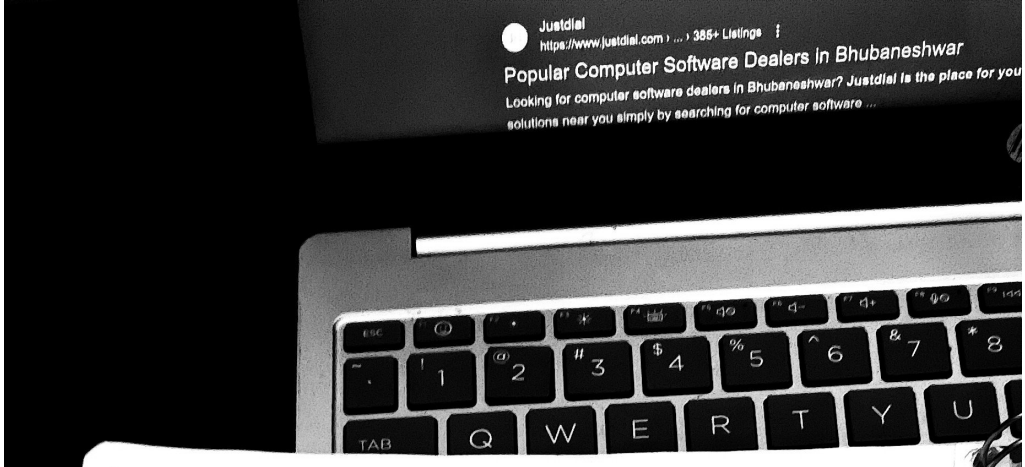
~~FILTERING DATA~~

BOOLEAN MASKING ⟶ eg :=

```
import numpy as np

arr = np.array([10, 20, 30, 40, 50])

print(arr[arr > 25])
```



(i) **RESHAPING AND MANIPULATING** ⟶ Reshaping is changing the dimensions of data without modifying its metadata elements

eg- 1D Array ⟶ 2D "
2D " ⟶ 3D"

* The total number of elements remain after reshaping / conversion remains the same.

SYNTAX = arr.reshape ()

Two arguments are taken as row and a column.

** # doesn't create a copy, rather it creates a view, (original Array)

** Note = This can only be done if and only if the DIMENTIONS Match otherwise you can't Reshape it.

eg - import numpy as np

arr = np.array ([10, 20, 30, 40, 50, 60])

~~arr. reshape~~

reshaped_arr = arr. reshape (2, 3)

print (reshaped_arr) ⟶

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

FLATTENING ARRAYS ⟶ ravel() / flatten()

Used for :-
Multi dimentional Array ⟶ 1D Array

- ravel()
- flatten()

⟶ Both work in similar ways

* Difference => .ravel() just changes the view only ~~but flatten changes the~~

but flatter creates a copy of the table.

```
import numpy as np

arr_2d = np.array([[1,2,3],
                   [4,5,6]])

print(arr_2d.ravel())  ⟶  [1. 2. 3. 4. 5. 6.]

print(arr_2d.flatten())  ⟶  [1. 2. 3. 4. 5. 6.]
```

RESHAPING AND MANIPULATING ARRAYS ⟹ Dimension Change without Modifying

→ Element

Add, Remove, Stack, Split

Syntax => np.insert(____, ____, ____, ____)

Array, Index, Value, axis = None by Default

axis = 0 (for Row-Wise) ⟵ NUM (for Flattened Array)

↳ 1 (for column wise)

[If it is none then inserted into Flattened array]

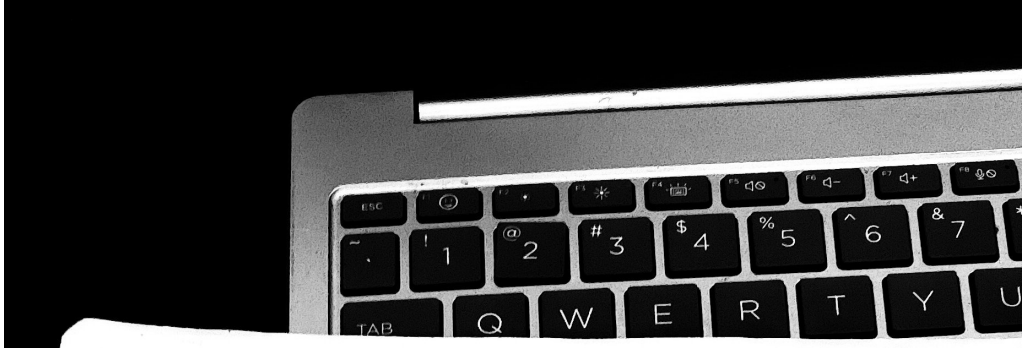
eg — import numpy as np

```
arr = np.array([10,20,30,40,50,60])
print(f"Original Array: {arr}")
new_arr = np.insert(arr, 2, 100)
print(f"New Array: {new_arr}")
```

[10, 20, 30, 40, 50, 60]

⟶ [10, 20, 100, 30, 40, 50, 60]

NOW HOW TO DO THIS IN 2D



```
import numpy as np
arr_2d = np.array([[1,2],[3,4]])
print(arr_2d)  ——→  [[1 2]
                     [3 4]]
# Inserted at Position (1)
new_arr_2d = np.insert(arr_2d, 1, [5,6], axis = None)
print(new_arr_2d)
        ↳ [1,5,6,2,3,4]
```

It will 1st Flatten the 2D Array into a 1D then insert.

APPEND IN NUMPY ARRAYS

(Inserts at the End of the particular Numpy Array)

```
import numpy as np
arr = np.array([10,20,30])
new_array = np.append(arr, [40,50,60])
print(new_arr)  ↝  [10,20,30,40,50,60]
```

CONCATENATE 2 ARRAYS

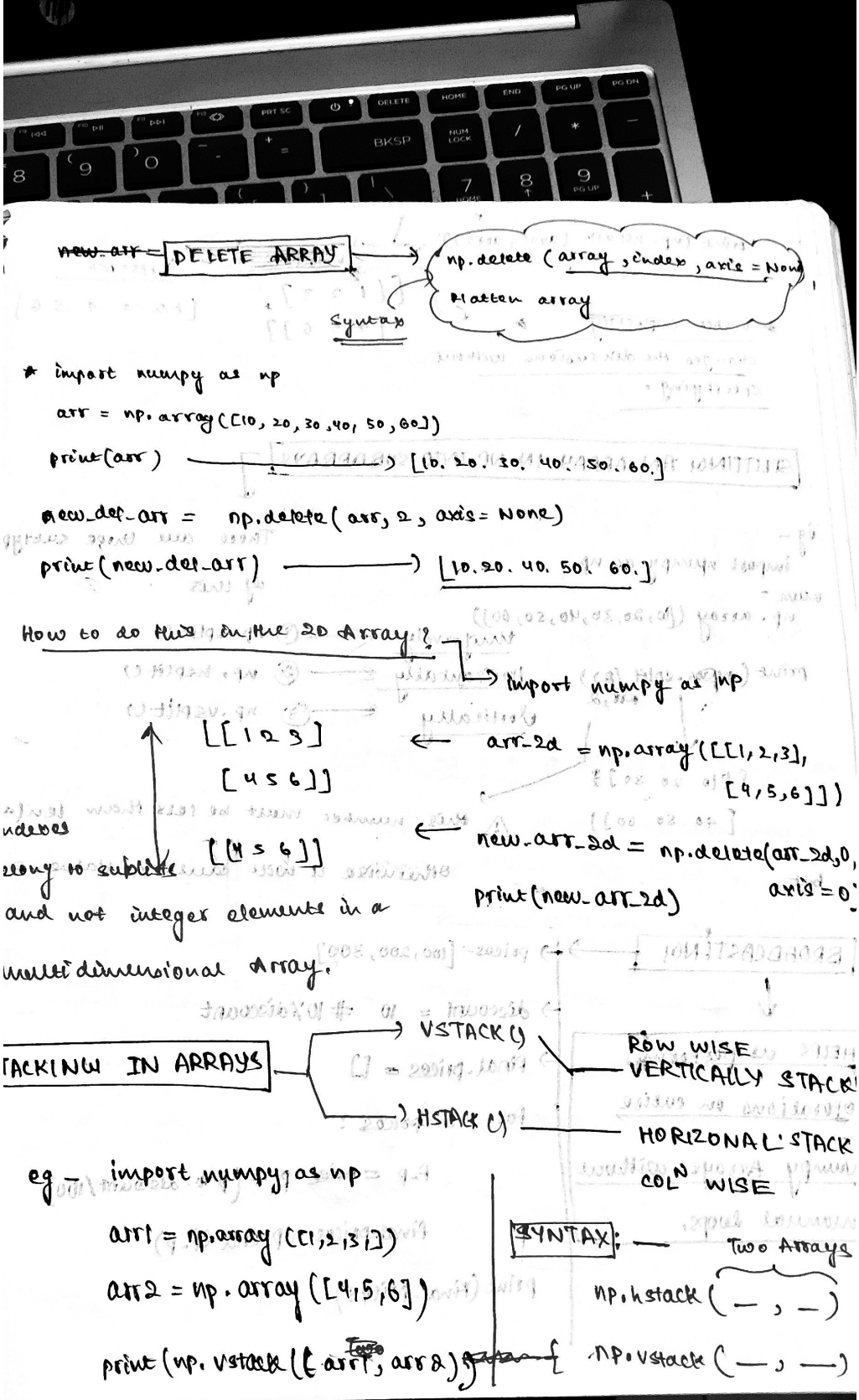
Syntax: `np.concatenate((array1, array2), axis = )`

`NP.CONCATENATE((Array1, Array2), axis = None)`

Axis = 0 → Vertical Stacking

Axis = 1 → Horizontal Stacking

```
import numpy as np
arr1 = np.array([1,2,3])
arr2 = np.array([4,5,6])
new_array = np.concatenate((arr1, arr2))  → [1 2 3 4 5 6]
print(new_array)
```



new_arr = DELETE ARRAY → np.delete(array, index, axis = None)
Syntax
Flatten array

* import numpy as np

arr = np.array([10, 20, 30, 40, 50, 60])

print(arr) ⟶ [10. 20. 30. 40. 50. 60.]

new_del_arr = np.delete(arr, 2, axis = None)

print(new_del_arr) ⟶ [10. 20. 40. 50. 60.]

How to do this in the 2D Array?

→ import numpy as np

arr_2d = np.array([[1,2,3],
[4,5,6]])

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

new_arr_2d = np.delete(arr_2d, 0, axis = 0)

[[4 5 6]]

print(new_arr_2d)

indexes
array to subtract
and not integer elements in a
multidimensional Array.

STACKING IN ARRAYS
→ VSTACK() → ROW WISE VERTICALLY STACK
→ HSTACK() → HORIZONAL STACK COL^N WISE

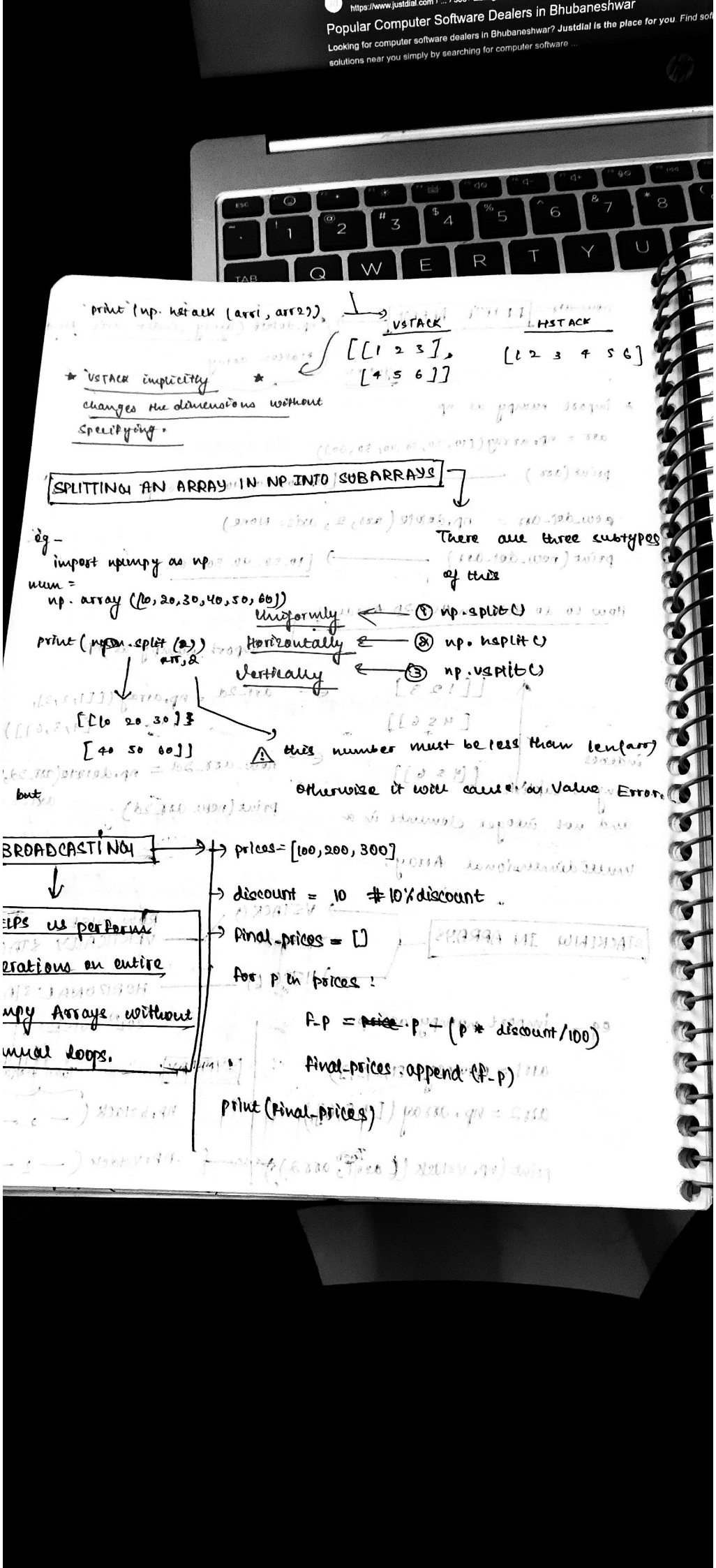
eg — import numpy as np

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

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

print(np.vstack((arr1, arr2)))

SYNTAX: Two Arrays
np.hstack(—, —)
np.vstack(—, —)


Popular Computer Software Dealers in Bhubaneshwar
Looking for computer software dealers in Bhubaneshwar? Justdial is the place for you. Find soft
solutions near you simply by searching for computer software ...
print (np. hstack (arr1, arr2))
VSTACK
[[1 2 3],
[4 5 6]]
HSTACK
[1 2 3 4 5 6]
* VSTACK implicitly changes the dimensions without specifying.
SPLITTING AN ARRAY IN NP INTO SUBARRAYS
There are three subtypes of this
eg -
import numpy as np
num =
np. array ([10, 20, 30, 40, 50, 60])
print (np.split (a))
a=2
[[10 20 30]
[40 50 60]]
Uniformly ← ① np.split()
Horizontally ← ② np. hsplit()
Vertically ← ③ np.vsplit()
this number must be less than len(arr)
Otherwise it will cause a Value Error.
but
BROADCASTING
prices = [100, 200, 300]
discount = 10 # 10% discount
final_prices = []
for p in prices :
f_p = p - (p * discount/100)
final_prices.append(f_p)
print (final_prices)
elps us perform
erations on entire
mpy Arrays without
unual loops.

```
import numpy as np
prices = np.array([100,200,300])
discount = 10
final_prices = prices - (prices * discount/100)
print(final_prices) ⟹ [This will be a Numpy Array]
```

HOW NUMPY HANDLE ARRAYS OF DIFFERENT SHAPES

1) Matching dimensions := arr1 [1,2,3] + arr2 [4,5,6] → [5,7,9]

2) Expanding single dimensions :=

eg → [1,2,3] + 10

~~import numpy~~ = [11,12,13]

↳ If we add a single element to an array the single element would generally be expanded to operate with each element of the list)

3) INCOMPATIBLE SHAPES := Any other combination (other than ① and ②) returns in a Value Error due to incompatible operand types.

[1,2,3] + [1,2]

(3) (2)

↳ Error.

eg —

```
import numpy as np
~~arr = numpy~~
arr = np.array([100,200,300])
```

```
result = arr * 2
print (result)  ——→ [200, 400, 600]
```

eg(2) ⟹

```
import numpy as np
matrix = np.array ([1,2,3],[4,5,6])
vector = np.array ([10,20,30]
result = matrix + vector
print (result) ⟹ [[ 11 22 33 ]
                  [ 14 25 36 ]]
```

This is what we call as a Matrix Dimension Rules system

eg(3) ⟹

```
import numpy as np
arr1 = np.array ([[1,2,3],[4,5,6]])   # shape (2,3)
arr2 = np.array ([1,2])               # shape (2,)
result = arr1 + arr2
print (result) ⟹ Value Error
```

VECTORIZATION ——→ Iterating over the loop at the last without LOOP is known as Vectorization.

~~list~~

g —— import numpy as np

```
arr_1 = np.array ( [1,2,3] )
arr_2 = np.array ( [4,5,6] )
result = arr1 + arr2
print (result) ⟹ ([5,7,9])
```

```
import numpy as np
arr = np.array([10, 20, 30])
multi = arr * 2
print(multiplied) ⟶ [30. 40. 90.]
```

VECTORIZATION is about 100x faster than traditional python loops as the backend part is written in C/C++ for speed.

HANDLING MISSING / INCORRECT VALUES → These are useful in handling broken Null, Empty or NaN values.

Built in Functions

* np.isNaN = Detect Missing Values
* np.nan_to_num() ⇒ Converts NaN
* np.isinf() ⇒ Detects infinite value.

What is NaN ?

NaN (Not A Number) — NaN is a Javascript Term which means the adept absence of a Number.

eg –

```
import numpy as np
arr = np.array([1, 2, np.nan, 4, np.nan, 6])
print(np.isnan(arr)) ⟶ [False False True False True False]
```

↳ RETURNS a Boolean Array True for NaN False otherwise.

```
cleaned_arr = np.nan_to_num(arr)
print(cleaned_arr)
```

⇒ [1, 2, 0, 4, 0, 6] ↳ If we dont specify any value then NaN will become Replaced with 0

```
clel = np.nan_to_num(arr, nan=100)
print(clel) ⟶ [1, 2, 100, 4, 100, 6]
```


```
import numpy as np
arr = np.array([1, 2, np.inf, 4, -np.inf, 6])
print(np.isinf(arr))
```

works with both inf and -∞ values

→ BOOLEAN ARRAY

(False False True False True False)

how to we convert it to some finite number

```
cleaned_arr = np.nan_to_num(arr, posinf=1000, neginf=-1000)
print(cleaned_arr)
```

Replaces +∞ with some value (0 by def)

Replaces -∞ with some value (0 by default)