coding_notes_by_vipul

NumPy

complete handwritten notes

by vipul yadav

Together Learn

Numpy

Numpy is a library for the Python programming language, adding support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate ~~to~~ on these arrays.

Numpy is a famous Python library used for working with arrays. One of the important functions of this library is stack().

Why is Numpy faster?

-fixed type

- Faster to read less bytes of memory

x

- No type checking when iterating through objects

- It utilize contiguous memory.

Benefits.

- SIMD. (Single Instruction multiple Data) vector processing.

- Effective Cache Utilization.

Applications:

Mathematics (MATLAB Replacement)

plotting. (Matplotlib)

BackEnd (Pandas, Connect 4, Digital Photography).

Machine learning.

Basics.

```
import numpy as np.

a = np.array.([1,2,3])
    print(a).
b = np.array ([[9.0,8.0,7.0],[6.0,5.0,4.0]])
    print(b)

# Get dimensions
  a.ndim . or b.ndim.

# Get shape
  a.shape or b.shape

# Get type
  ~~a.type~~ a.dtype or b.dtype
```

Get size

a.itemsize or b.itemsize

Get total size

a.size * a.itemsize or b.size * b.itemsize

or a.nbytes or b.nbytes

Accessing/changing specific elements, rows, columns etc

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

Get a specific elem [row, column]

```
a[1,5]
```

→ = 13

Get a specific row

```
a[0,:]
```

→ array([1,2,3,4,5,6,7])

Get a specific column

```
a[:,2]
```

→ array([3,10])

Getting a little more fancy [startingIndex : endIndex : stepSize]

```
a[0,1:-1:2]
```

→ array([2,6,8])

= [3,5,7,9]

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

=

changing an element.

```
a[1,5] = 20
print(a)
→ [[1 2 3 4 5 6 7]
   [8 9 10 11 12 20 14]]

a[:,2] = [1,2]
print(a)
→ [[1 2 1 4 5 6 7]
   [8 9 2 11 12 20 14]]
```

3d example

```
b = np.array([[[1,2],[3,4]],[[5,6],[7,8]]])
print(b)
→ [[[1 2]
    [3 4]]
   [[5 6]
    [7 8]]]
```

Get specific element.

```
b[0,1,1]
→ 4
b[0,1,:]
→ array([3, 4])

b[:,1,:] → array([[3,4]
                  [7,8]])
```

replace

b[:1,:] = [[9,9],[8,8]]

Initializing different types of Arrays.

All 0s matrix

np.zeros((2,3))

→ array([[0., 0., 0.],
[0., 0., 0.]])

All 1s matrix

np.ones((4,2,2))

Any other number

np.full((2,2), 99)

→ array([[99, 99],
[99, 99]])

Any other (full_like)

~~np.full(a.shape, 4)~~ np.full_like(a, 4)

Random decimal numbers

np.random.rand(4,2)

Random Integer numbers

np.random.randint(7, size=(3,3))

The Identity matrix

```
np.identity(5)
```

Repeat an array.

```
arr = np.array([[1,2,3]])
r1 = np.repeat(arr, 3, axis=0)
print(r1)
```

→
```
[[1 2 3]
 [1 2 3]
 [1 2 3]]
```

#. Task 1

Print array by methods

```
1 1 1 1 1
1 0 0 0 1
1 0 9 0 1
1 0 0 0 1
1 1 1 1 1
```

```
output = np.ones((5,5))
z = np.zeros((3,3))
z[1,1] = 9
output[1:4,1:4] = z
```

MATHEMATICS

```
a = np.array([1,2,3,4])
print(a)
→ [1 2 3 4]
a+2
→ array([3,4,5,6])
a-2
→ array([-1,0,1,2])
a*2
→ array([2,4,6,8])
a/2
→ array([0.5,1,1.5,2])
b = np.array([1,0,1,0])
a+b
→ array([2,2,4,4])
a**2
→ array([1,4,9,16])
# Take the trigonometry.
np.sin(a)
→ array([0.8414, 0.9092, 0.1411, -0.7568])
```

Linear Algebra

```
a = np.ones((2,3))
print(a)
→ [[1. 1. 1.]
   [1. 1. 1.]]
```

```
b = np.full((3,2), 2)        →  [[2 2]
print(b)                        [2 2]
                                [2 2]]
```

→

```
np.matmul(a, b).
```

→

```
array([[6., 6.],
       [6., 6.]])
```

#Find the determinant

```
c = np.identity(3).
np.linalg.det(c)
```

→ 1.0.

Statistics

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

→

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

```
np.min(stats)
```

→ 1

```
np.max(stats)
```

→ 6

```
np.sum(stats, axis=0).
```

→ array([5, 7, 9])

Recognizing Arrays.

```
before = np.array([[1,2,3,4],[5,6,7,8]])
print(before)
→ [[1 2 3 4]
~~after~~ [5 6 7 8]]
after = before.reshape((8,1))
print(after)
[[1]
 [2]
 [3]
 [4]
 [5]
 [6]
 [7]
 [8]]
```

```
# Vertically stacking vectors.
v1 = np.array([1,2,3,4])
v2 = np.array([5,6,7,8])

~~np.stack~~ np.vstack([v1,v2])
→
array([[1, 2, 3, 4],
       [5, 6, 7, 8]])
```

Horizontal stack

```
h1 = np.ones((2,4))
h2 = np.zeros((2,2))

np.hstack((h1, h2))
```

MISCELLENEOUS

Load Data from file.

```
x = np.genfromtxt('filename', delimiter=',')
x = x.astype('int32')
x
```

Revision of Numpy.

BASIC

linspace() :- To create an array of evenly spaced numbers over ~~an~~ a specified interval.

```
c = np.linspace(1, 10, 10)
                 |   |   |
              starting ending  no. of evenly spaced no.
```

⇒ [1. 2. 3. 4. 5. 6. 7. 8. 9. 10.]

diff : 1

```
c = np.linspace(1, 10, 5)
```

⇒ [1. 3.25 5.5 7.75 10.]

difference = ~~0.25~~ 2.25

```
c = np.linspace(1, 6, 5)
```

⇒ [1. 2.25 3.5 4.75 6.]

difference: 1.25

Array attributes :-

```
let a = np.~~are~~ array([[1, 2, 3],
                         [4, 5, 6]])
```

C.ndim : dimensions = 2.

C.shape : shape = (2,3).

C.size : size = 6.

C.dtype : data type = int64.

Slicing.

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

⇒ [6 7 8 9 10]

d = C[1:10:2] ⇒ [2 4 6 8 10]
d = C[2:10:2] ⇒ [3 5 7 9]

Converting list to Numpy Array.

```
my_list = [1,2,3,4,5]
print(my_list)
C = np.array(my_list)
print(C)
```

INTERMMEDIATE

reshape : Conversion of shape

```
arr = np.arange(1,13)
print(arr)
reshaped_arr = arr.reshape(2,6)
print(reshaped_arr)
```

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

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

ravel : ~~resha~~ again reshaped in the original sha

```
arr = np.arange(1,13)
reshaped_arr = arr.reshape(2,6)
flattened_arr = reshaped_arr.ravel()
print(flattened_arr)
```

flatten : Same ~~thing~~ as ravel().

Stacking and splitting:

Stacking refers to combining multiple arrays along a new axis.

1. np.stack : Combines arrays along a new axis.

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

result = np.stack((a,b), axis = 0)
print(result).
```

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

```
result = np.stack((a,b), axis = 1)
print(result)
```

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

2. np.hstack : stacks arrays horizontally (along axis 1).

```
result = np.hstack((a,b))
print(result)
```

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

3. np.vstack : stacks arrays vertically (along axis 0).

```
result = np.vstack((a,b))
print(result)
```

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

4. np.dstack : stacks arrays along the third dimension (depth).

```
result = np.dstack((a,b))
print(result)
```

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

5. np.column_stack : stacks 1D arrays as columns into a 2D array.

```
result = np.column_stack((a,b))
print(result)
```

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

6. np.row_stack ~~((a, b))~~ : stacks arrays row-wise (similar to vstack).

```
result = np.row_stack((a,b))
print(result)
```

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

Splitting :- refers to dividing an array into multiple sub-arrays.

1. numpy.split()

- Splits an array into equal-sized sub-arrays.
- If the array cannot be split evenly, it raises a ~~valueerror~~ ValueError.

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

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

2. np.array_split()

- similar to split(), but allows uneven splitting.
- useful when the array size isn't divisible by the no. of splits.

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

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

3. numpy.hsplit()

- Splits an array horizontally (column-wise)
- Works on arrays with at least 2 dimensions.

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

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

4. numpy.vsplit()

- splits an array vertically (row-wise)
- works on arrays with at least 2 dimensions.

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

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

5. numpy.dsplit()

- splits an array along the third dimension (depth-wise)
- Works on 3D arrays.

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

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

Copy vs View

~~Copy:~~

VIEW :- The view is just a view of the original ndarray. and the view does not own the data.
When we make changes to the view it affects the original array, and when changes are made to the original array it affects the view. This is also known as shallow cop

Ex: import numpy as np.

```
arr = np.array([2, 4, 6, 8, 10])
v = arr.view()
print("id of arr", id(arr))
print("id of v", id(v))

arr[0] = 12

Print('original array -', arr)
print("view -", v)
```

COPY: The copy is completely a new array and the copy owns the data.
When we make changes to the copy it does not affect the original array and vice-versa.
This is also known as Deep Copy.

Ex: Import numpy as np

```
arr = np.array([2, 4, 6, 8, 10])
c = arr.copy()
print("id of arr", id(arr))
print("id of c", id(c))

arr[0] = 12

print("original array:", arr)
print("copy:", c)
```

~~Broa~~

Broadcasting.

It refers to the ability to perform arithmetic operations on arrays of different shapes. This feature allows Numpy to treat arrays with different dimensions during arithmetic operations, ensuring they have compatible shapes without making unnecessary copies of data.

Ex: Import numpy as np.

```
a = np.array([1.0, 2.0, 3.0])
b = 2.0
result = a * b
print(result).
```

Boolean Masking and filtering.

Boolean Masking in Numpy is a powerful feature that allows you to filter or manipulate elements of an array based on conditions. It involves creating a boolean array (true/false values) that serves as a mask to select specific elements from another array.

Ex 1: Basic Boolean Masking.

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

filtered = arr[mask]
print(filtered)
```

Ex 2: Modify elements using a mask

```
arr[arr > 3] = 99
print(arr)                    [1, 2, 3, 99, 99]
```

Ex 3: Boolean Masking with 2-D Arrays

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

mask = matrix > 5

filtered = ~~arr(mask)~~ matrix[mask].
print(filtered). [6 7 8 9]

Filtering in Numpy refers to selecting elements from an array based on a condition or set of conditions. This is typically done using boolean indexing or logical operations. ~~Below are~~

Ex1: Filtering with a condition

import numpy as np

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

filtered = arr[arr > 30]
print(filtered) [40, 50]

Ex2: Filtering with multiple conditions.

filtered = arr[(arr >= 20) & (arr <= 40)]
print(filtered) [20 30 40]

Ex3: Filtering using np.where

```
indices = np.where(arr > 25)
print(indices).          Array([2, 3, 4])

filtered = arr[indices]
print(filtered).         [30 40 50]
```

Data Type Conversion (astype())

- Check the data type of a Numpy array:-

```
import numpy as np

arr = np.array([1, 2, 3])
print(arr.dtype).        ⇒ int64
```

Ex 1:- Convert Integer to float.

```
arr = np.array([1, 2, 3], dtype = np.int32)
float_arr = arr.astype(np.float64)
print(float_arr)  ⇒ [1. 2. 3.]
print(float_arr.dtype)  ⇒ float64
```

Ex 2:- Convert float to Integer

```
arr = np.array([1.5, 2.7, 3.9])
int_arr = arr.astype(np.int32)
print(int_arr)
```

Ex 3:- Convert string to Numeric.

```
str_arr = np.array(['1', '2', '3'])
num_arr = str_arr.astype(np.int32)
print(num_arr)          → [1 2 3]
```

Ex 4:- Convert Numeric to Boolean.

```
arr = np.array([0, 1, 2, 0])
bool_arr = arr.astype(bool)
print(bool_arr)   [False, True ~~False~~ True False]
```

Matrix Operations:

- Dot product (.dot), matrix multiplication.

```
import numpy as np

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

result = np.dot(A,B)
print(result)
```

Output :- [[19 22]
[43 50]]

linalg module.

The numpy.linalg module in Numpy provides a collection of linear algebra functions. These functions are optimized for performance and are built on top of BLAS and LAPACK libraries.

Inverse of a matrix

```
inv = np.linalg.inv(A)
print("Inverse :\n", inv)
```

Determinant of a matrix

```
det = np.linalg.det(A)
print("Determinant :", det).
```

Eigenvalues and Eigenvectors

```
eigenvalues, eigenvectors = np.linalg.eig(A)
print("Eigen values :", eigenvalues).
print("Eigen vectors :", eigenvectors).
```

Random module.

The random module in Numpy is a powerful tool for generating random numbers, arrays, and performing random sampling.

Generate random no.

uniform Distribution : floats between 0 to 1.

```
random_no = np.random.rand(5)
```

normal distribution : no. from a normal distribution

```
normal_no = np.random.randn(5)
```

Integers :- random integers with a specified range.

```
random_integers = np.random.randint(1, 10, size=5)
```

Seed for Reproducibility. : If fixed the no. every time you run.

```
np.random.seed(42)
random_no = np.random.rand(3)
```

Performance comparision.

vectorization vs. Python loops.

1. Vectorization

Definition :- Vectorization refers to performing operations on entire arrays or matrices at once,

Performance :- significantly faster because it uses highly optimized C/Fortran libraries and avoids Python's per-iteration overhead

Code Simplicity :- cleaner and more concise code, as operations are applied directly to arrays without explicit loops.

Ex :-

```
import numpy as np

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

result = a+b
#vectorized operation.
```

2 Python loops.

Definition : Python loops (e.g. for or while) iterate over elements one at a time, performing operations explicitly.

Performance :- slower due to python's interpreted nature and the overhead of loop execution.

Code simplicity :- More verbose and prone to errors especially for large datasets.

Ex :-

```
a = [1, 2, 3]
b = [4, 5, 6]

result = [a[i] + b[i] for i in range (len (a))]
```

Key difference

Aspect	vectorization	Python loops
Speed.	Faster	slower
Ease of use	Concise and readable	verbose and manual
Scalability	Handles large datasets efficiently	Struggles with large datasets
Flexibility.	limited to support operations	Fully customizable logic

- np.where

The numpy.where() func^n is a versatile tool in Numpy that allows you to return elements based on a condition.

Basic usage :- condition only., returns the indices.

```
import arr = np.array([10, 15, 20, 25, 30])
result = np.where(arr > 20).
print(result)
⇒ array([3, 4])
```

Condition selection (cond^, x, y) :- returns x where cond^ is true and y where cond^ is false.

```
arr = np.array([10, 15, 20, 25, 30])
result = np.where(arr > 20, arr, -1)
print(result)
⇒ output : [-1, -1, -1, 25, 30]
```

2D Arrays :-

```
arr = np.array([[10, 15], [20, 25]])
result = np.where(arr > 15)
print(result)
⇒ output : array([0, 1, 1])
           array([1, 0, 1])
```

np.select.

In numpy np.select is a function used to construct an array based on multiple conditions and corresponding choices. It evaluates a list of conditions and selects values from the corresponding list of choices based on which condn is true. If no condn is true, an optional default value can be specified.

Syntax :- numpy.select(condlist, choicelist, default=0)

Basic usage :-

```
x = np.array([0, 1, 2, 3, 4])
conditions = [x < 2, x >= 2]

choices = [x * 10, x * 100]

result = np.select(conditions, choices)
print(result)
```

output :- [0 10 200 300 400]

Using a default value.

```
x = np.array([0, 1, 2, 3, 4])
conditions = [x < 2, x == 3]
choices = [x * 10, x * 100]

result = np.select(conditions, choices, default = -1)
print(result)
```

output :- [0 10 -1 300 -1]

Complex conditions

```
x = np.array([0, 1, 2, 3, 4])
conditions = [(x % 2 == 0), (x % 2 != 0)]
choices = ['even', 'odd']

result = np.select(conditions, choices, default = 'unknown')
print(result)
```

output :- ['even' 'odd' 'even' 'odd' 'even']

Practice Problem.

```
import numpy as np.

dice_roll = np.random.randint (1, 7, size = 100)

print (dice_roll).

frequency . = np.bincount (dice_roll)[1:]

for faces, count in enumerate (frequency, start = 1):
        print (f"faces {face} appeard {count} times")
```

NUMPY

Sorting an multidimension array.

```
Import numpy as np.

a = np.array([[6, 9, 1],
              [5, 3, 7],
              [2, 9, 1]])
```

Wrong way.

```
b = np.sort(np.sort(a, axis=0), axis=1)
print(b)
```

⇒
```
[[1 2 3]
 [1 5 9]
 [6 7 9]]
```

~~Right~~ → Correct way

```
b=np.sort(a.flatten()).reshape(a.shape)
print(b)
```

⇒
```
[[1 1 2]
 [3 5 6]
 [7 9 9]]
```

// Breakdown.

b = a.flatten() // [6 9 1 5 3 7 2 9 1]

c = np.sort(b) // [1 1 2 3 5 6 7 9 9]

d = c.reshape(a.shape) // [[1 1 2]
[3 5 6]
[7 9 9]]

Iterating

```
import numpy as np

a = np.arange(12).reshape(3,4)
print(a)

//Not a professional way

for row in a:
    for element in row:
        print(element, end=' ')

//Linebreak Print()

// Professional way

for element in np.nditer(a):
    print(element, end=' ')
```

Masking

```
import numpy as np
import numpy.ma as ma

arr = np.array([1, 2, 3, np.NaN, 4, np.inf])
masked_arr = ma.masked_array(arr, mask=[0,0,0,1,0,1])
print(masked_arr)
```

```
arr. #multidimensional array.
arr = np.array([[1,2,3],[4,5,6]])
masked_arr = ma.masked_array(arr, mask=[[0,0,1],
  [1,0,0]]).
print(masked_arr)
print(ma.getmask(masked_arr))
print(ma.masked_greater(arr, value=4))
print(ma.masked_inside(arr, v1=2, v2=4)).
print(ma.masked_outside(arr, v1=2, v2=4))
print(ma.masked_where(arr%2==0, arr))
print(ma.masked_invalid(arr)).
```

Vectorization.

```
import numpy as np

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

def square_if_even(x):
    if x % 2 == 0:
        return x**2
    else:
        return x

vectorized_square_if_even = np.vectorize(square_if_even)
print(vectorized_square_if_even(arr))
```