Numpy Advance Operations

```
import numpy as np
```

Numpy Broadcasting

In NumPy, we can perform mathematical operations on arrays of different shapes. An array with a smaller shape is expanded to match the shape of a larger one. This is called broadcasting.

```
# Rule 1 (Arrays with same shape are compatible)

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

```
[5 7 9]
```

```
# Rule 2 (Arrays with dimensions of size 1 are stretched or repeated
along that dimension.)
'''
Example:
Shape(1,3) and Shape(4,3)
Shape(2,1) and Shape(2,3)
Shape(1,1) and Shape(3,1)
'''
a = np.array([1,2,3]) # 1 x 3
b = np.array([
    [4,5,6],          # 2 x 3
    [7,8,9]
])
result = a + b
print(result)
```

```
[[ 5  7  9]
 [ 8 10 12]]
```

```
# Rule 3 (Array with size 1 in a particular dimension act as if they
had the size of the maximum shape along that dimension.)

a = np.array([10,20,30])
scalar = 5
result = a / scalar
print(result)
```

```
[2. 4. 6.]
```

From buffer

Creates a new one-dimensional array from a buffer object.

A buffer object is a memory object that store in a sequence of bytes such as a string, bytes, bytearray etc.

Syntax:

numpy.frombuffer(buffer_object,dtype,count,offset)

buffer_object : Buffer object you want to convert into array.

dtype : Resulted data type of an array.

count : How much item you want to fetch from buffer object? If set to -1, it will read all items.

offset : Buffer object's starting position, default is 0 index

```
myBytes = b'\x01\x02\x03\x04\x05'
myArray = np.frombuffer(myBytes,dtype=np.uint8)
print(myArray)
```

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

Fromiter

It creates a new one-dimensional array from an iterable object like list,tuple,string. Syntax : numpy.fromiter(iterable, dtype, count = -1)

Parameters :

```
iterable : The iterable object providing data for the array.

dtype : [data-type] Data-type of the returned array.

count : [int, optional] Number of items to read.

Returns : [ndarray] The output array.
```

```
my_iterable=[1,2,3,4,5,6]
print(type(my_iterable))
my_array=np.fromiter(my_iterable,dtype=int)
print(my_array)
print(type(my_array))
```

```
<class 'list'>
[1 2 3 4 5 6]
<class 'numpy.ndarray'>
```

```
iterable = (x * x for x in range(5))
my_array = np.fromiter(iterable,float)
print(my_array)
```

```
[ 0.  1.  4.  9. 16.]
```

```
a = "python"
# creating 1-d array
b = np.fromiter(a, dtype='U2')
print(b)
```

```
['p' 'y' 't' 'h' 'o' 'n']
```

Matrix operations

A matrix is a two-dimensional data structure where numbers are arranged into rows and columns.

array() creates a matrix

dot() performs matrix multiplication

transpose() transposes a matrix

linalg.inv() calculates the inverse of a matrix

linalg.det() calculates the determinant of a matrix

flatten() transforms a matrix into 1D array

```
## Matrox Multiplication (dot.)
matrix1 = np.array([[1,3,],
                    [5,7]])

matrix2 = np.array([[2,4],
                   [6,8]])

result = np.dot(matrix1,matrix2)
print(result)
```

```
[[20 28]
 [52 76]]
```

```
## Transpose
matrix1 = np.array([[1, 3],
                    [5, 7]])
result = np.transpose(matrix1)

print(result)
```

```
[[1 5]
 [3 7]]
```

```
## Inverse of matrix
matrix1 = np.array([[1,3,5],
                    [7,9,2],
                   [4,6,8]])

result = np.linalg.inv(matrix1)
print(result)

'''
Note: If we try to find the inverse of a non-square matrix, we will
get an error message:
'''
```

```
[[-1.11111111 -0.11111111  0.72222222]
 [ 0.88888889  0.22222222 -0.61111111]
 [-0.11111111 -0.11111111  0.22222222]]
```

```
'\nNote: If we try to find the inverse of a non-square matrix, we will
get an error message:\n'
```

```
## Determinant of matrix
'''
In 2 x 2 matrix,
 Matrix A = [a b       determinant = ad-bc
             c d]

In 3 x 3 matrix,
Matrix A = [a b c
            d e f
            g h i]

determinant = a(ef - fh) −b(di − fg) + c(dh − eg)

Note: Alternate signs are applied: +,−,+
'''

matrix1 = np.array([[1, 2, 3],
                    [4, 5,1],
                   [2, 3, 4]])

result = np.linalg.det(matrix1)
print(f"{result:.2f}")
```

```
-5.00
```

```
## Flaten matrix
'''
Flattening a matrix simply means converting a matrix into a 1D array.
```

```
'''
matrix1 = np.array([[1, 2, 3],
                    [4, 5, 7]])

result = matrix1.flatten()
print("Flattened 2x3 matrix:", result)
```

```
Flattened 2x3 matrix: [1 2 3 4 5 7]
```

Numpy Set Operations

A set is a collection of unique data. That is, elements of a set cannot be repeated.

NumPy set operations perform mathematical set operations on arrays like union, intersection, difference, and symmetric difference.

```
## unionn
A = np.array([1,3,5])
B = np.array([0,2,3])

result = np.union1d(A,B)
print(result)
```

```
[0 1 2 3 5]
```

```
## intersection
A = np.array([1,3,5])
B = np.array([0,2,3])

result = np.intersect1d(A,B)
print(result)
```

```
[3]
```

```
## Difference
# A- B
A = np.array([1,3,5])
B = np.array([0,2,3])
result = np.setdiff1d(A,B)
print(result)

# B - A
A = np.array([1,3,5])
B = np.array([0,2,3])

result = np.setdiff1d(B,A)
print(result)
```

```
[1 5]
[0 2]
```

```
## Symmetric Difference
#The symmetric difference between two sets A and B includes all
elements of A and B without the common elements.

A = np.array([1, 3, 5])
B = np.array([0, 2, 3])

result = np.setxor1d(A, B)

print(result)
```

```
[0 1 2 5]
```

```
## Unique Values
array1 = np.array([1,1, 2, 2, 4, 7, 7, 3, 5, 2, 5])
result = np.unique(array1)
print(result)
```

```
[1 2 3 4 5 7]
```

Vectorization

We've used the concept of vectorization many times in NumPy. It refers to performing element-wise operations on arrays.

```
# Lets start with simple example
array1 = np.array([1,2,3,4,5])
number = 10
result = array1 + number
print(result)
```

```
[11 12 13 14 15]
```

```
## Vectorization to add two array together

array1 = np.array([[1,2,3],
                   [4,5,6]])
array2 = np.array([[9,8,7],
                   [6,5,4]])
array_sum = array1 + array2

print(array_sum)
```

```
[[10 10 10]
 [10 10 10]]
```

Array Manipulation

NumPy provides several functions to manipulate arrays, such as reshaping, flattening, stacking, and splitting

Reshaping Arrays

Reshaping allows you to change the shape of an array without changing its data. For example, you can convert a 1D array into a 2D array.

Flattening Arrays

Flattening converts a multi-dimensional array into a 1D array.

Stacking Arrays

Stacking combines multiple arrays along a new axis.

Splitting Arrays

Splitting divides an array into multiple sub-arrays.

```
# Reshaping Arrays
array = np.arange(1, 10)
reshaped_array = array.reshape(3, 3)
print("Reshaped Array:\n", reshaped_array)

# Flattening Arrays
flattened_array = reshaped_array.flatten()
print("Flattened Array:", flattened_array)

# Stacking Arrays
array1 = np.array([1,2,3])
array2 = np.array([4, 5, 6])
stacked_array = np.stack((array1, array2))
print("Stacked Array:\n", stacked_array)

# Splitting Arrays
split_array = np.split(array, 3)
print("Split Array:", split_array)
```

```
Reshaped Array:
 [[1 2 3]
 [4 5 6]
 [7 8 9]]
Flattened Array: [1 2 3 4 5 6 7 8 9]
Stacked Array:
 [[1 2 3]
 [4 5 6]]
Split Array: [array([1, 2, 3]), array([4, 5, 6]), array([7, 8, 9])]
```

Statistical Operations

NumPy provides functions to compute statistical measures such as mean, median, standard deviation, sum, min, and max.

Mean, Median, Standard Deviation

These functions help you understand the central tendency and spread of your data.

Sum, Min, Max

These functions provide basic summary statistics for your array.

```
# Mean, Median, Standard Deviation
array = np.array([1, 2, 3, 4, 5])
mean = np.mean(array)
median = np.median(array)
std_dev = np.std(array)
print("Mean:", mean)
print("Median:", median)
print("Standard Deviation:", std_dev)

# Sum, Min, Max
total_sum = np.sum(array)
min_value = np.min(array)
max_value = np.max(array)
print("Sum:", total_sum)
print("Min:", min_value)
print("Max:", max_value)
```

```
Mean: 3.0
Median: 3.0
Standard Deviation: 1.4142135623730951
Sum: 15
Min: 1
Max: 5
```

Linear Algebra

NumPy's **linalg** module provides functions for linear algebra operations such as eigenvalues, eigenvectors, and singular value decomposition (SVD).

Eigenvalues and Eigenvectors

Eigenvalues and eigenvectors are fundamental in many areas of linear algebra, including stability analysis and quantum mechanics.

Singular Value Decomposition (SVD)

SVD is a factorization method that decomposes a matrix into three other matrices, useful in signal processing and data compression. python

```
# Eigenvalues and Eigenvectors
matrix = np.array([[4, 1], [2, 3]])
```

```
eigenvalues, eigenvectors = np.linalg.eig(matrix)
print("Eigenvalues:", eigenvalues)
print("Eigenvectors:\n", eigenvectors)

# Singular Value Decomposition (SVD)
U, S, V = np.linalg.svd(matrix)
print("U:\n", U)
print("S:\n", S)
print("V:\n", V)
```

```
Eigenvalues: [5. 2.]
Eigenvectors:
 [[ 0.70710678 -0.4472136 ]
 [ 0.70710678  0.89442719]]
U:
 [[-0.76775173 -0.64074744]
 [-0.64074744  0.76775173]]
S:
 [5.11667274 1.95439508]
V:
 [[-0.85065081 -0.52573111]
 [-0.52573111  0.85065081]]
```

Interpolation

Interpolation is used to estimate values between known data points. NumPy provides tools for linear and spline interpolation.

```
x = np.array([0, 1, 2, 3, 4])
y = np.array([1, 3, 5, 7, 9])

# Interpolate at x = 2.5
interpolated_value = np.interp(2.5, x, y)
print("Interpolated Value at x = 2.5:", interpolated_value)
```

```
Interpolated Value at x = 2.5: 6.0
```