Numpy Assignment Questions

1. What is a Python library? Why do we use Python libraries?

A Python library is a collection of pre-written code, functions, and modules that can be easily imported and used in your Python scripts or programs. These libraries provide a set of tools and functionalities that help developers perform common tasks without having to write the code from scratch. Python libraries are designed to be reusable and can save a significant amount of time and effort by providing pre-built solutions for various tasks.

Eg: Numpy, Pandas, Flask

Here are some reasons why we use Python libraries:

1. **Code Reusability:** Libraries contain pre-written code that can be reused in different projects, saving time and effort.
2. **Efficiency:** Python libraries are often optimized for performance and efficiency. Using well-established libraries allows developers to benefit from optimized algorithms and well-tested code, leading to more efficient programs.
3. **Domain-Specific Functionality:** Many Python libraries are specialized for specific domains, such as data science, machine learning, web development, and more. By using these libraries, developers can easily access tools and functions tailored to their specific needs.
4. **Community Contributions:** Python has a large and active community of developers who contribute to the creation and maintenance of libraries. This results in a rich ecosystem of libraries that cover a wide range of applications and domains.
5. **Standardization:** Some Python libraries have become de facto standards for certain tasks. For example, libraries like NumPy and Pandas are widely used for numerical and data manipulation tasks in the field of data science.
6. **Documentation and Support:** Popular Python libraries usually come with extensive documentation and have a large user base. This means that developers can easily find help, tutorials, and examples, making it easier to learn and use the library effectively.

2. What is the difference between Numpy array and List?

The main difference between a NumPy array and a list in Python lies in their performance, homogeneity, and memory management. Here are the key differences:

1. **Performance**: NumPy arrays are faster than lists, especially when performing operations like calculating mean or sum, which are much faster on NumPy arrays.
2. **Homogeneity**: NumPy arrays have strict requirements on the homogeneity of the objects they contain, meaning they can only store elements of the same type. Lists, on the other hand, can hold elements of varying data types, such as integers, floating-point numbers, strings, boolean values, or even other data structures like dictionaries.
3. **Memory Management**: NumPy arrays store elements in adjacent memory locations, reducing fragmentation and allowing for efficient access. Lists, however, store additional information about each element, such as its type and reference count, which can lead to significant overhead when dealing with a large number of elements.
4. **Creation**: Lists are built-in data structures in Python, while NumPy arrays require importing the NumPy library.

3. Find the shape, size and dimension of the following array?

[[1, 2, 3, 4],

[5, 6, 7, 8],

[9, 10, 11, 12]]

```
import numpy as np

arr = np.array([[1, 2, 3, 4],
[5, 6, 7, 8],
[9, 10, 11, 12]])

print("Shape of the array:", arr.shape)
print("Size of the array:",arr.size)
print("Dimension of the array:", arr.ndim)
```

```
Shape of the array: (3, 4)
Size of the array: 12
Dimension of the array: 2
```

4. Write python code to access the first row of the following array?

[[1, 2, 3, 4]

[5, 6, 7, 8],

[9, 10, 11, 12]]

```
import numpy as np

arr = np.array([[1, 2, 3, 4],
[5, 6, 7, 8],
[9, 10, 11, 12]])

first_row = arr[0]

print("Original array:\n",arr)
print("First row:",first_row)
```

```
Original array:
 [[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
First row: [1 2 3 4]
```

5. How do you access the element at the third row and fourth column from the given numpy array?

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

```
import numpy as np

arr = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])

required_ele = arr[2][3]
```

```
print("Original array:\n",arr)
print("\nElement at third row and fourth column:", required_ele)
```

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

Element at third row and fourth column: 12
```

6. Write code to extract all odd-indexed elements from the given numpy array?

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

```
import numpy as np

arr = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])

odd_indexed_ele = arr[:,1::2].flatten() #To convert into single
dimentional array


print("Original array:\n",arr)
print("\nOdd indexed elements:", odd_indexed_ele)
```

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

Odd indexed elements: [ 2  4  6  8 10 12]
```

7. How can you generate a random 3x3 matrix with values between 0 and 1?

```
import numpy as np

#creating a 3x3 matrix with random elements between 0 and 1

random_matrix = np.random.rand(3,3)

print(random_matrix)
```

```
[[0.26092322 0.24138778 0.05351538]
 [0.97703495 0.38656924 0.8584963 ]
 [0.72570802 0.17403751 0.00168888]]
```

8. Describe the difference between np.random.rand and np.random.randn?

`np.random.rand:`

- This function generates random numbers from a **uniform distribution** over the interval [0, 1).
- The syntax is numpy.random.rand(d0, d1, ..., dn), where d0, d1, ..., dn are the dimensions of the output array.
- The generated numbers are floats uniformly distributed between 0 (inclusive) and 1 (exclusive).

```
# For example

import numpy as np
arr1 = np.random.rand(3,3)
print(arr1)
```

```
[[0.594662   0.91168252 0.24572413]
 [0.85884978 0.21464153 0.43000393]
 [0.11755084 0.33436876 0.91550355]]
```

`np.random.randn:`

- This function generates random numbers from a **standard normal distribution (also known as a Gaussian distribution)** with mean 0 and standard deviation 1.
- The syntax is numpy.random.randn(d0, d1, ..., dn), where d0, d1, ..., dn are the dimensions of the output array.
- The generated numbers follow a bell-shaped curve centered around 0, and their values can range from negative to positive infinity.

```
# For example

import numpy as np
arr = np.random.randn(2,2)
print(arr)
```

```
[[-1.28637271  1.13006625]
 [-0.19564593  0.59682645]]
```

9. Write code to increase the dimension of the following array?

[[1, 2, 3, 4],

[5, 6, 7, 8],

[9, 10, 11, 12]]

```
import numpy as np

arr = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])

print("Original array:\n",arr)
print("original dimention:",arr.ndim)

new_arr = np.expand_dims(arr,axis = 1)

print("\nArray with increased dimension:\n",new_arr)
print("Now dimention becomes:",new_arr.ndim)
```

```
Original array:
 [[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
original dimention: 2

Array with increased dimension:
 [[[ 1  2  3  4]]

 [[ 5  6  7  8]]

 [[ 9 10 11 12]]]
Now dimention becomes: 3
```

10. How to transpose the following array in NumPy?

[[1, 2, 3, 4]

[5, 6, 7, 8],

[9, 10, 11, 12]]

```
import numpy as np
```

```
arr = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
print("Original array:\n",arr)
tranposed_arr = arr.T
print("\nTransposed array:\n",arr.T)
```

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

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

11. Consider the following matrix:

Matrix A: [[1, 2, 3, 4], [5, 6, 7, 8],[9, 10, 11, 12]]

Matrix B: [[1, 2, 3, 4], [5, 6, 7, 8],[9, 10, 11, 12]]

Perform the following operation using Python:

1. Index wise multiplication
2. Matrix multiplication
3. Add both the matrics
4. Subtract matrix B from A
5. Divide Matrix B by A

```
import numpy as np
matrix_a = np.array([[1, 2, 3, 4], [5, 6, 7, 8],[9, 10, 11, 12]])
matrix_b = np.array([[1, 2, 3, 4], [5, 6, 7, 8],[9, 10, 11, 12]])
print("Matrix A:\n",matrix_a)
print("\nMatrix B:\n",matrix_b)

##1. Index wise multiplication
print("\nIndexwise multiplication:\n",matrix_a * matrix_b)

##2. Matrix multiplication
print("\nMatrix multiplication:\n",matrix_a @ matrix_b.T) #we have
taken the transpose of Matrix B for matrix multiplication

##3. Addition of both the matrix
print("\nAddition of both the matrix:\n",matrix_a + matrix_b)
```

```python
##4. Subtraction of matrix B from matrix A
print("\nSubtraction of matrix b from matrix A:\n",matrix_a -
matrix_b)

##5. Division of matrix B by A
print("\nDivision of matrix B by matrix A:\n",matrix_b/matrix_a)
```

```
Matrix A:
 [[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]

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

Indexwise multiplication:
 [[  1   4   9  16]
 [ 25  36  49  64]
 [ 81 100 121 144]]

Matrix multiplication:
 [[ 30  70 110]
 [ 70 174 278]
 [110 278 446]]

Addition of both the matrix:
 [[ 2  4  6  8]
 [10 12 14 16]
 [18 20 22 24]]

Subtraction of matrix b from matrix A:
 [[0 0 0 0]
 [0 0 0 0]
 [0 0 0 0]]

Division of matrix B by matrix A:
 [[1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]]
```

12. Which function in Numpy can be used to swap the byte order of an array?

The `numpy.ndarray.byteswap` method can be used to swap the byte order of an array in NumPy. This method swaps the byte order of the elements of the array in place.

```
import numpy as np

# Create an array with a specific byte order
original_array = np.array([1, 2, 3, 4], dtype='>i4')  # '>i4'
represents big-endian 32-bit integer

# Print the original array
print("Original array:\n", original_array)

# Swap the byte order
swapped_array = original_array.byteswap()

# Print the array after byte swapping
print("\nArray after byte swapping:\n", swapped_array)
```

```
Original array:
 [1 2 3 4]

Array after byte swapping:
 [16777216 33554432 50331648 67108864]
```

13. What is the significance of the np.linalg.inv function?

The `np.linalg.inv` function in NumPy is used to compute the (multiplicative) inverse of a square matrix(with non-zero determinant). The result of this function is another matrix, known as the inverse matrix, which, when multiplied by the original matrix, results in the identity matrix.

For a given square matrix A , if A^-1 is the inverse of A , then the following relationship holds:

A . A^-1 = A^-1 . A = I

where I is the identity matrix.

The significance of `np.linalg.inv` lies in its applications, such as:

1. **Solving Linear Systems of Equations:**
   - Given a linear system of equations Ax = B , where A is a coefficient matrix, x is the column vector of variables, and B is the column vector on the right-hand side, the solution can be found as x = A^-1B.

2. **Eigenvalue Problems:**
   - In eigenvalue problems, the inverse of a matrix is often used. For a square matrix A and its eigenvector matrix V (columns are eigenvectors), the inverse A^-1 is used in the relationship V^-1AV = Λ, where (Λ) is a diagonal matrix of eigenvalues.

Here's a simple example of using `np.linalg.inv`:

```
import numpy as np

# Create a square matrix
matrix_a = np.array([[1, 2], [3, 4]])

# Compute the inverse matrix
inverse_matrix_a = np.linalg.inv(matrix_a)

# Verify the result: A * A_inv should be an identity matrix
identity_matrix = np.dot(matrix_a, inverse_matrix_a)

print("Original matrix A:\n", matrix_a)
print("\nInverse matrix A_inv:\n", inverse_matrix_a)
print("\nIdentity matrix (A * A_inv):\n", identity_matrix)
```

```
Original matrix A:
 [[1 2]
 [3 4]]

Inverse matrix A_inv:
 [[-2.   1. ]
 [ 1.5 -0.5]]

Identity matrix (A * A_inv):
 [[1.00000000e+00 1.11022302e-16]
 [0.00000000e+00 1.00000000e+00]]
```

14. What does the np.reshape function do, and how is it used?

The `np.reshape` function in NumPy is used to change the shape of an array without changing its data. It returns a new array with a modified shape, and the data remains the same. The reshaping can be performed as long as the total number of elements in the original array is equal to the total number of elements in the reshaped array.

```
import numpy as np

# Example 1: Reshape a 1D array to a 2D array
arr1 = np.array([1, 2, 3, 4, 5, 6])
```

```
reshaped_arr1 = np.reshape(arr1, (2, 3))
print("Original array:\n", arr1)
print("Reshaped array:\n", reshaped_arr1)

# Example 2: Reshape a 2D array to a 1D array
arr2 = np.array([[1, 2, 3], [4, 5, 6]])
reshaped_arr2 = np.reshape(arr2, 6)
print("\nOriginal array:\n", arr2)
print("Reshaped array:\n", reshaped_arr2)
```

```
Original array:
 [1 2 3 4 5 6]
Reshaped array:
 [[1 2 3]
 [4 5 6]]

Original array:
 [[1 2 3]
 [4 5 6]]
Reshaped array:
 [1 2 3 4 5 6]
```

15. What is broadcasting in Numpy?

`Broadcasting` in NumPy is a powerful mechanism that allows universal functions to work with arrays of different shapes. It enables element-wise operations between arrays of different shapes by automatically aligning the arrays and duplicating elements as needed. This allows for more concise and efficient code, as it eliminates the need for explicit looping over the arrays. Broadcasting follows a set of rules to determine how the arrays should be aligned, and it can also be used to perform operations between arrays and scalars.

```
#for example

##example 1

import numpy as np

a = np.array([1.0, 2.0, 3.0])
b = 2.0

# Performing element-wise multiplication without explicitly creating a
copy of b
result = a * b

print("Array a:\n", a)
print("Scalar b:", b)
print("\nResult after broadcasting:\n", result)
```

```
Array a:
 [1. 2. 3.]
Scalar b: 2.0

Result after broadcasting:
 [2. 4. 6.]
```

```
##Example 2
import numpy as np

# Create a 1D array
a = np.array([1, 2, 3])
print(a,"\n")
# Create a 2D array
b = np.array([[1], [2], [3]])
print(b,"\n")
# Add the two arrays
c = a + b

# Print the result
print(c)
```

```
[1 2 3]

[[1]
 [2]
 [3]]

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