Numpy_Assignment

September 10, 2024

[1]:
```
#Theoretical Questions:
```

[ ]:
```
#Q1. Explain the purpose and advantages of NumPy in scientific computing and␣
↪data analysis. How does it enhance Python's capabilities for numerical␣
↪operations?
'''
1)Purpose of NumPy in Scientific Computing and Data Analysis
Efficient Data Handling: NumPy provides the ndarray, an n-dimensional array␣
↪object that is more efficient than Python’s built-in data structures like␣
↪lists. It allows for the storage and manipulation of large datasets with␣
↪minimal memory overhead.

Mathematical Operations: NumPy offers a vast array of mathematical functions to␣
↪perform operations on arrays, such as linear algebra, statistical␣
↪operations, Fourier transformations, and more. These functions are optimized␣
↪for performance, making them faster than standard Python operations.

Interfacing with C/C++ and Fortran: NumPy can interface with C, C++, and␣
↪Fortran, enabling the reuse of existing scientific libraries and making it␣
↪easier to write high-performance code.

Foundational Library: NumPy serves as the foundation for many other scientific␣
↪libraries in Python, such as SciPy, pandas, and scikit-learn. It provides␣
↪the base data structures and functions on which these libraries build.





2)Advantages of NumPy

Speed and Performance:

Vectorization: NumPy performs operations on entire arrays rather than␣
↪individual elements, which is known as vectorization. This reduces the need␣
↪for loops and results in more concise and readable code, as well as␣
↪significant performance improvements.
```

```
Optimized C Implementation: Many of NumPy's operations are implemented in C,␣
 ↪which is compiled and highly optimized. This makes NumPy operations much␣
 ↪faster than equivalent operations in pure Python.
Memory Efficiency:

Compact Data Types: NumPy arrays are more compact than Python lists because␣
 ↪they use less memory to store the same amount of data. This efficiency is␣
 ↪particularly important when working with large datasets.
Contiguous Memory Layout: NumPy arrays are stored in contiguous blocks of␣
 ↪memory, which enhances performance by allowing for more efficient data␣
 ↪access and manipulation.
Broadcasting:

Flexible Operations: NumPy's broadcasting allows operations on arrays of␣
 ↪different shapes and sizes without the need to explicitly reshape them. This␣
 ↪simplifies the code and avoids the overhead of creating additional data␣
 ↪structures.
Comprehensive Functionality:

Mathematical Functions: NumPy provides a wide range of mathematical functions,␣
 ↪including trigonometric, exponential, and logarithmic functions, as well as␣
 ↪linear algebra and random number generation.
Integration with Other Libraries: As a core library, NumPy seamlessly␣
 ↪integrates with other Python libraries used in data science, such as pandas␣
 ↪for data manipulation, Matplotlib for plotting, and TensorFlow for machine␣
 ↪learning.
Ease of Use and Accessibility:

High-level Syntax: NumPy's syntax is relatively simple and intuitive, making it␣
 ↪accessible for users who may not have extensive programming experience.
Community and Ecosystem: NumPy has a large and active community, with extensive␣
 ↪documentation and a rich ecosystem of related tools and libraries.



3)Enhancing Python's Capabilities
Python, by itself, is an interpreted language and is not designed for␣
 ↪high-performance numerical computing. NumPy bridges this gap by providing␣
 ↪efficient array operations, enabling Python to handle large-scale numerical␣
 ↪data and perform complex computations at high speed. This makes Python a␣
 ↪viable tool for scientific computing, data analysis, and machine learning.
'''
```

[ ]:
```
#Q2. Compare and contrast np.mean() and np.average() functions in NumPy. When␣
 ↪would you use one over the other?
'''
```

```
1)np.mean():

Purpose: Calculates the arithmetic mean (average) of the elements along a␣
↪specified axis of an array. If no axis is specified, it computes the mean of␣
↪the entire array.
Syntax: np.mean(array, axis=None, dtype=None, out=None, keepdims=False)
Usage: np.mean() is straightforward and is typically used when you want to␣
↪calculate the average of an array without considering weights.
np.average():

Purpose: Calculates the weighted average of the elements along a specified axis␣
↪of an array. If weights are not provided, it defaults to calculating the␣
↪arithmetic mean, similar to np.mean().
Syntax: np.average(array, axis=None, weights=None, returned=False)
Usage: np.average() is used when you need to compute a weighted average, where␣
↪each element in the array may have a different level of importance or␣
↪contribution.



2)When to Use:

Use np.mean() when you need a simple average of all elements or along a␣
↪particular axis.
Use np.average() when you need to account for weights and calculate a weighted␣
↪average.
'''
```

[2]:

```
#Q#3 Describe the methods for reversing a NumPy array along different axes.␣
↪Provide examples for 1D and 2D arrays.
'''
Reversing a NumPy Array Along Different Axes
1D Array: To reverse a 1D array, you can use slicing with [::-1].
2D Array: For reversing along different axes, you can use slicing or specific␣
↪functions like np.flip.
'''
#Examples:

import numpy as np

# 1D array
arr_1d = np.array([1, 2, 3, 4, 5])
reversed_1d = arr_1d[::-1]
print(reversed_1d)

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

```python
print(arr_2d)

# Reverse along the first axis (rows)
reversed_2d_rows = arr_2d[::-1, :]
print(reversed_2d_rows)

# Reverse along the second axis (columns)
reversed_2d_columns = arr_2d[:, ::-1]
print(reversed_2d_columns)

# Reverse both axes
reversed_2d_both = arr_2d[::-1, ::-1]
print(reversed_2d_both)
```

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

[ ]:
```python
#Q4 How can you determine the data type of elements in a NumPy array? Discuss␣
↪the importance of data types in memory management and performance.
'''
Determining the Data Type of Elements in a NumPy Array
Method: Use the .dtype attribute to determine the data type of the elements in␣
↪a NumPy array.
'''
import numpy as np

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

'''
Importance of Data Types:

Memory Management: Different data types occupy different amounts of memory.␣
↪Choosing the appropriate data type can save memory, especially when working␣
↪with large datasets.
Performance: NumPy operations are optimized based on the data type. Using␣
↪smaller or appropriate data types can enhance computation speed and␣
↪efficiency.
'''
```

[ ]:
```
#Q5 Define ndarrays in NumPy and explain their key features. How do they differ␣
↪from standard Python lists?
'''
ndarray:

-->An ndarray (n-dimensional array) is a versatile array object in NumPy that␣
↪can hold items of the same type and supports efficient operations on large␣
↪datasets.
->Key Features:
Homogeneous: All elements are of the same data type.
Efficient Memory Layout: Stored in contiguous blocks of memory.
Vectorized Operations: Supports element-wise operations without the need for␣
↪explicit loops.
Support for Multidimensional Arrays: Can represent matrices, tensors, and other␣
↪n-dimensional data structures.
Difference from Python Lists:

->Efficiency: ndarrays are more memory-efficient and faster for numerical␣
↪computations compared to Python lists.
->Functionality: ndarrays support a wide range of mathematical operations and␣
↪broadcasting, which are not available for lists.
->Fixed Size: ndarrays have a fixed size, whereas lists can dynamically grow or␣
↪shrink.
'''
```

[ ]:
```
#Q6 Analyze the performance benefits of NumPy arrays over Python lists for␣
↪large-scale numerical operations
'''
1)Performance Benefits of NumPy Arrays Over Python Lists
2)Vectorization: NumPy arrays perform operations on entire arrays rather than␣
↪element-by-element, eliminating the need for loops and speeding up execution.
3)Memory Efficiency: NumPy arrays use less memory by storing data more␣
↪compactly.
4)Optimized C Backend: Many NumPy functions are implemented in C, providing␣
↪significant speed advantages over Python's built-in functions.
5)Contiguous Memory Allocation: Improves cache efficiency, leading to faster␣
↪access and processing.
'''
```

[3]:
```
#Q7  Compare vstack() and hstack() functions in NumPy. Provide examples␣
↪demonstrating their usage and output.
'''
Comparing vstack() and hstack() Functions in NumPy

1)vstack():
```

```
Purpose: Vertically stacks arrays (along the first axis).
Usage: Useful when you want to stack multiple arrays one on top of the other.
'''
#Example:


arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])
vstacked = np.vstack((arr1, arr2))  # Output: [[1, 2, 3], [4, 5, 6]]
print(vstacked)

'''
2)hstack():

Purpose: Horizontally stacks arrays (along the second axis).
Usage: Useful when you want to concatenate arrays side by side.
'''
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])
hstacked = np.hstack((arr1, arr2))  # Output: [1, 2, 3, 4, 5, 6]
print(hstacked)
```

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

[4]:

```
#Q8  Explain the differences between fliplr() and flipud() methods in NumPy,␣
 ↪including their effects on various array dimensions.
'''
fliplr():

Purpose: Flips a 2D array from left to right (i.e., it reverses the order of␣
 ↪columns).
'''


# Example:
arr = np.array([[1, 2, 3], [4, 5, 6]])
fliplr_arr = np.fliplr(arr)
print(fliplr_arr)

'''
Purpose: Flips a 2D array upside down (i.e., it reverses the order of rows).
'''
#Example:

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

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

[5]:
```
#Q9 Discuss the functionality of the array_split() method in NumPy. How does it␣
↪handle uneven splits?
'''
Purpose: Splits an array into multiple sub-arrays. Unlike split(),␣
↪array_split() can handle uneven splits, meaning the array can be split into␣
↪sub-arrays of different sizes.
'''
#Example:
arr = np.array([1, 2, 3, 4, 5])
split_arr = np.array_split(arr, 3)
print(split_arr)
```

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

[ ]:
```
#Q10. Explain the concepts of vectorization and broadcasting in NumPy. How do␣
↪they contribute to efficient array operations?
'''
Concepts of Vectorization and Broadcasting in NumPy
Vectorization:

Definition: The process of performing operations on entire arrays rather than␣
↪element-by-element, leading to more concise code and faster execution.
'''
#Example:
import numpy as np

arr = np.array([1, 2, 3])
result = arr * 2
print(result)

'''
Broadcasting:

Definition: Allows NumPy to perform operations on arrays of different shapes by␣
↪automatically expanding the smaller array to match the shape of the larger␣
↪array.
'''
#Example:
arr1 = np.array([1, 2, 3])
arr2 = np.array([[1], [2], [3]])
result = arr1 + arr2
print(result)
'''
Contribution to Efficiency:
```

```
Vectorization eliminates explicit loops, resulting in cleaner and faster code.
Broadcasting simplifies code by allowing operations between arrays of different␣
↪shapes without the need to manually reshape or expand arrays, thus improving␣
↪both development speed and runtime efficiency.
'''
```

[ ]:

```

```

[ ]:

```
#Practical Questions:
```

[6]:

```
#Q1 Create a 3x3 NumPy array with random integers between 1 and 100. Then,␣
↪interchange its rows and columns.
import numpy as np

# Create a 3x3 array with random integers between 1 and 100
arr = np.random.randint(1, 101, size=(3, 3))
print("Original Array:")
print(arr)

# Interchange rows and columns (transpose the array)
transposed_arr = np.transpose(arr)
print("\nTransposed Array:")
print(transposed_arr)
```

```
Original Array:
[[16 45 47]
 [72 81 11]
 [48 33  6]]

Transposed Array:
[[16 72 48]
 [45 81 33]
 [47 11  6]]
```

[7]:

```
#Q2  Generate a 1D NumPy array with 10 elements. Reshape it into a 2x5 array,␣
↪then into a 5x2 array.

# Generate a 1D array with 10 elements
arr_1d = np.arange(10)
print("1D Array:")
print(arr_1d)

# Reshape the 1D array into a 2x5 array
arr_2x5 = arr_1d.reshape(2, 5)
print("\n2x5 Array:")
print(arr_2x5)
```

```
# Reshape the 2x5 array into a 5x2 array
arr_5x2 = arr_2x5.reshape(5, 2)
print("\n5x2 Array:")
print(arr_5x2)
```

```
1D Array:
[0 1 2 3 4 5 6 7 8 9]

2x5 Array:
[[0 1 2 3 4]
 [5 6 7 8 9]]

5x2 Array:
[[0 1]
 [2 3]
 [4 5]
 [6 7]
 [8 9]]
```

[8]:

```
#Q3 Create a 4x4 NumPy array with random float values. Add a border of zeros␣
↪around it, resulting in a 6x6 array.

import numpy as np

# Create a 4x4 array with random float values
arr_4x4 = np.random.rand(4, 4)
print("Original 4x4 Array:")
print(arr_4x4)

# Add a border of zeros around the 4x4 array
arr_6x6 = np.pad(arr_4x4, pad_width=1, mode='constant', constant_values=0)
print("\n6x6 Array with Zero Border:")
print(arr_6x6)
```

```
Original 4x4 Array:
[[0.60576917 0.04260094 0.61754829 0.08652999]
 [0.33246653 0.92720753 0.54303768 0.30087915]
 [0.05869365 0.18957677 0.86591651 0.36605555]
 [0.35574955 0.75810242 0.20462922 0.08394335]]

6x6 Array with Zero Border:
[[0.         0.         0.         0.         0.         0.        ]
 [0.         0.60576917 0.04260094 0.61754829 0.08652999 0.        ]
 [0.         0.33246653 0.92720753 0.54303768 0.30087915 0.        ]
 [0.         0.05869365 0.18957677 0.86591651 0.36605555 0.        ]
 [0.         0.35574955 0.75810242 0.20462922 0.08394335 0.        ]
```

```
 [0.          0.          0.          0.          0.          0.          ]]
```

[9]:
```python
#Q4 Using NumPy, create an array of integers from 10 to 60 with a step of 5.

import numpy as np
arr = np.arange(10, 65, 5)
print("Array from 10 to 60 with step 5:")
print(arr)
```

```
Array from 10 to 60 with step 5:
[10 15 20 25 30 35 40 45 50 55 60]
```

[10]:
```python
#Q5. Create a NumPy array of strings ['python', 'numpy', 'pandas']. Apply␣
 ↪different case transformations
#(uppercase, lowercase, title case, etc.) to each element.

import numpy as np

# Create a NumPy array of strings
arr = np.array(['python', 'numpy', 'pandas'])

# Apply uppercase transformation
upper_arr = np.char.upper(arr)
print("Uppercase:")
print(upper_arr)

# Apply lowercase transformation
lower_arr = np.char.lower(arr)
print("\nLowercase:")
print(lower_arr)

# Apply title case transformation
title_arr = np.char.title(arr)
print("\nTitle Case:")
print(title_arr)
```

```
Uppercase:
['PYTHON' 'NUMPY' 'PANDAS']

Lowercase:
['python' 'numpy' 'pandas']

Title Case:
['Python' 'Numpy' 'Pandas']
```

[11]:
```python
#Q6 Generate a NumPy array of words. Insert a space between each character of␣
 ↪every word in the array.
```

```
import numpy as np

# Create a NumPy array of words
arr = np.array(['python', 'numpy', 'pandas'])

# Insert a space between each character of every word
spaced_arr = np.char.join(' ', arr)
print("Array with spaces between characters:")
print(spaced_arr)
```

```
Array with spaces between characters:
['p y t h o n' 'n u m p y' 'p a n d a s']
```

[12]:
```
#Q7 Create two 2D NumPy arrays and perform element-wise addition, subtraction,
↪multiplication, and division.

import numpy as np

# Create two 2D NumPy arrays
array1 = np.array([[1, 2, 3], [4, 5, 6]])
array2 = np.array([[7, 8, 9], [10, 11, 12]])

# Perform element-wise addition
addition = np.add(array1, array2)
print("Element-wise Addition:")
print(addition)

# Perform element-wise subtraction
subtraction = np.subtract(array1, array2)
print("\nElement-wise Subtraction:")
print(subtraction)

# Perform element-wise multiplication
multiplication = np.multiply(array1, array2)
print("\nElement-wise Multiplication:")
print(multiplication)

# Perform element-wise division
division = np.divide(array1, array2)
print("\nElement-wise Division:")
print(division)
```

```
Element-wise Addition:
[[ 8 10 12]
 [14 16 18]]
```

```
Element-wise Subtraction:
[[-6 -6 -6]
 [-6 -6 -6]]

Element-wise Multiplication:
[[ 7 16 27]
 [40 55 72]]

Element-wise Division:
[[0.14285714 0.25       0.33333333]
 [0.4        0.45454545 0.5       ]]
```

[13]:

```
#Q8 Use NumPy to create a 5x5 identity matrix, then extract its diagonal␣
 ↪elements.
import numpy as np

# Create a 5x5 identity matrix
identity_matrix = np.eye(5)
print("5x5 Identity Matrix:")
print(identity_matrix)

# Extract the diagonal elements
diagonal_elements = np.diag(identity_matrix)
print("\nDiagonal Elements:")
print(diagonal_elements)
```

```
5x5 Identity Matrix:
[[1. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0.]
 [0. 0. 1. 0. 0.]
 [0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 1.]]

Diagonal Elements:
[1. 1. 1. 1. 1.]
```

[14]:

```
#Q9 Generate a NumPy array of 100 random integers between 0 and 1000. Find and␣
 ↪display all prime numbers in this array.

import numpy as np

# Function to check if a number is prime
def is_prime(n):
    if n <= 1:
        return False
    if n <= 3:
        return True
```

```
    if n % 2 == 0 or n % 3 == 0:
        return False
    i = 5
    while i * i <= n:
        if n % i == 0 or n % (i + 2) == 0:
            return False
        i += 6
    return True

# Generate a NumPy array of 100 random integers between 0 and 1000
random_integers = np.random.randint(0, 1001, size=100)
print("Array of 100 Random Integers:")
print(random_integers)

# Find all prime numbers in the array
primes = np.array([num for num in random_integers if is_prime(num)])
print("\nPrime Numbers in the Array:")
print(primes)
```

```
Array of 100 Random Integers:
[228 241 925 846 233 625 192 706 539 308 734 261 278 201  67 201 808 171
 390 163 162  86 829 822 951 974 980 483 710 204 585  62 687 546 355 977
 856 465 386 501 912 284  33 808 585 604 760 102 324 400 327 997 898 586
  12 375 918 687 488 956 607 788 306 898 878 610 483  64 445 233 913 100
 530 639 656 968  34 463  53  69 214 926 169   0 674 797 312 492 388 127
 512 220 477 980 878 449 442 972 362 156]

Prime Numbers in the Array:
[241 233  67 163 829 977 997 607 233 463  53 797 127 449]
```

[15]:
```
#Q10 Create a NumPy array representing daily temperatures for a month.␣
 ↪Calculate and display the weekly averages.

import numpy as np

daily_temperatures = np.random.randint(20, 40, size=30)
print("Daily Temperatures for the Month:")
print(daily_temperatures)


weeks = daily_temperatures[:28].reshape(4, 7)
last_days = daily_temperatures[28:]

weekly_averages = np.mean(weeks, axis=1)
print("\nWeekly Averages:")
print(weekly_averages)
```

```python
if len(last_days) > 0:
    last_days_avg = np.mean(last_days)
    print("\nAverage Temperature for the Last Days:")
    print(last_days_avg)
```

```
Daily Temperatures for the Month:
[31 32 33 26 36 38 22 27 29 34 20 30 33 38 24 22 39 38 25 28 29 25 31 30
 24 25 33 37 25 24]

Weekly Averages:
[31.14285714 30.14285714 29.28571429 29.28571429]

Average Temperature for the Last Days:
24.5
```

[ ]:

[ ]:

[ ]:

[ ]: