**Numpy tasks**

1. What are the internal data structures used by NumPy arrays (ndarray)?
2. How does NumPy achieve performance gains over Python lists?
3. When would you use np.array() vs np.asarray()?
4. Explain how memory is managed in NumPy. What is a buffer?
5. How do you check if a NumPy array is a view or a copy?
6. What is broadcasting, and how can it be used to write memory-efficient code?
7. How does NumPy’s vectorization help in optimizing code? Give an example.
8. How would you avoid loops in NumPy and perform operations efficiently?
9. Explain memory layout order (C vs F) and how it affects performance.
10. How would you profile the performance of NumPy operations in a large project?
11. How do you handle missing values (NaNs or Infs) in a large NumPy array?
12. What’s the most efficient way to normalize a dataset using NumPy?
13. How would you merge multiple arrays with different shapes into one?
14. How do you sort a structured NumPy array by multiple fields?
15. How would you implement outlier removal using NumPy?
16. Compare np.dot(), np.matmul(), and @. When would you use each?
17. How do you solve a system of linear equations using NumPy?
18. What’s the use of np.linalg.svd() and how is it useful in data compression?
19. Explain the difference between np.linalg.inv() and np.linalg.pinv().
20. Given a large dataset, how would you apply a rolling window operation efficiently?
21. How can NumPy be integrated into a real-time data pipeline?
22. How would you process images (as arrays) using NumPy for computer vision tasks?
23. How do you ensure data type consistency across multiple NumPy arrays?
24. What are common pitfalls when using NumPy in multi-threaded applications?

ANSWERS

1.shape,size,ndim

2. NumPy achieves **significant performance gains** over Python lists through several key mechanisms such as Contiguous Memory and Homogeneous Data, Vectorization, Broadcasting, Low-level Optimizations, Strides and Views, Precompiled Functions.

3. You would use np.array() when you want to create a new NumPy array and ensure that a copy of the data is made, even if the input is already a NumPy array. This is useful when you want to modify the new array without affecting the original data. In contrast, np.asarray() is preferred when you want to convert input (like a list or tuple) to a NumPy array **without copying the data** if it's already an array. This makes np.asarray() more efficient in cases where you only need a view or wrapper around the data, especially in read-only contexts. Therefore, use np.array() when data isolation is important, and np.asarray() when performance and memory usage are a concern.

4. In NumPy, **memory is managed efficiently** using a combination of contiguous memory blocks, data buffers, and metadata. When you create a NumPy array (ndarray), the actual data is stored in a **single, contiguous block of memory**, which allows for fast access and operations.The core of this system is the **buffer** — a block of raw memory that holds the array’s data in binary format (e.g., integers, floats). This buffer is referenced internally by the array, and NumPy uses metadata like shape, dtype, and strides to interpret the data correctly without copying it unnecessarily. A **buffer** in NumPy is a low-level memory container that stores the actual array elements. It is a flat (1D) sequence of bytes, regardless of the array’s dimensions. This buffer can be **shared**, allowing multiple views (e.g., slices or reshaped versions) to reference the same memory, improving performance and reducing memory usage.

5. To check if a NumPy array is a **view** or a **copy**, you can use the **.base attribute** of the array. If array.base is None, the array **owns its data**,it's a **copy**.If array.base is not None, the array is a **view**, and .base points to the original array.

6. **Broadcasting** in NumPy is a powerful feature that allows **arrays of different shapes** to be used together in arithmetic operations **without copying or reshaping data**. It lets NumPy perform element-wise operations efficiently by **virtually expanding** the smaller array across the larger one.

When performing operations like addition or multiplication on arrays with different shapes, NumPy compares their dimensions **from right to left** and applies these rules:

1. If the dimensions are equal, they are compatible.
2. If one of the dimensions is 1, it gets **broadcast (repeated)** to match the other.
3. If the dimensions are unequal and neither is 1, a ValueError is raised.

7. NumPy’s **vectorization** optimizes code by replacing slow Python loops with **fast, low-level C operations** that work on entire arrays at once. This results in **significantly faster execution**, **cleaner code**, and **better memory usage**.

**How Vectorization Helps:**

* Eliminates the overhead of interpreted Python loops.
* Uses highly optimized C/Fortran libraries internally.
* Reduces the number of lines and complexity in your code.
* Exploits CPU-level optimizations like SIMD (Single Instruction Multiple Data).

8. To avoid loops and perform operations efficiently in **NumPy**, you can use **vectorized operations**, **broadcasting**, and built-in **ufuncs (universal functions)**. These techniques allow you to operate on entire arrays at once using fast, low-level code under the hood.

9. In NumPy, the memory layout of arrays can be controlled using two primary orders: **C-order** (C-style) and **F-order** (Fortran-style). These refer to the way data is stored in memory, and the choice of layout can significantly impact the performance of certain operations.

**C-order (Row-major order)**

* **C-order** stores elements row by row. In other words, the rightmost dimension (columns) is stored **contiguously** in memory.
* **Most commonly used** (default in NumPy).
* Suitable for operations that access array data **row-wise**.

Example of C-order:

[[1, 2, 3],

[4, 5, 6],

[7, 8, 9]]

In memory, this would be stored as: [1, 2, 3, 4, 5, 6, 7, 8, 9].

**F-order (Column-major order)**

* **F-order** stores elements column by column, i.e., the leftmost dimension (rows) is stored **contiguously** in memory.
* Common in **Fortran**-based applications or when dealing with scientific computations where column-wise access is frequent.

Example of F-order:

[[1, 4, 7],

[2, 5, 8],

[3, 6, 9]]

In memory, this would be stored as: [1, 2, 3, 4, 5, 6, 7, 8, 9].

10. To profile the performance of NumPy operations in a large project, you can use several tools that provide varying levels of detail. For basic timing, the time module can be used to measure execution time, while timeit offers more accuracy, especially for short code snippets, by reducing noise from CPU cache and runtime fluctuations. For more detailed analysis, the cProfile module provides a function-level breakdown of where time is spent, which is ideal for large projects. If you need even more granular details, the line\_profiler package can be used to profile specific lines of code, revealing which lines are the most time-consuming. For memory profiling, the memory\_profiler package helps track memory consumption during execution, useful for detecting large memory spikes when working with NumPy arrays. Additionally, NumPy’s built-in ufunc profiler can be helpful for quickly profiling element-wise operations. By combining these tools, you can effectively identify performance bottlenecks and optimize critical sections of your NumPy-based code for better efficiency and faster execution.

11. Handling missing values in large NumPy arrays can be done through various techniques depending on your needs:

* **Identifying** missing values using np.isnan() and np.isinf().
* **Replacing** missing values using np.nan\_to\_num() or np.where().
* **Removing** NaNs or Infs with boolean indexing.
* **Imputing** values with the mean, median, or other statistics.
* Using **pandas** if the handling of missing values is more complex.

12. **Min-Max Normalization** is ideal when you want to scale the data to a specific range (usually [0, 1]), and it is especially useful for algorithms that rely on distance calculations, like k-NN or SVM. **Z-Score Normalization** (Standardization) is useful when you need the data to have a mean of 0 and a standard deviation of 1, which is commonly required by algorithms like linear regression, logistic regression, or neural networks.

13.  **np.concatenate()**: Best for combining arrays along an axis if the arrays have compatible shapes along that axis.

 **np.vstack(), np.hstack(), np.dstack()**: Used for stacking arrays along vertical, horizontal, or depth axes.

 **Reshaping with np.reshape()**: Useful for flattening arrays or reshaping them to have compatible dimensions before merging.

 **Padding with np.pad()**: Adds padding to arrays to match dimensions before concatenating or stacking them.

 **np.append()**: Simple but can be less efficient for large arrays compared to other methods.

14. To sort a structured NumPy array by multiple fields (i.e., columns), you can use the np.sort() function combined with the order parameter, which allows you to specify the fields you want to sort by. The fields are specified in a list or tuple, and the array is sorted lexicographically, meaning it will first sort by the first field, then by the second, and so on.

15.  Use **Z-score** for normally distributed data. Use **IQR** when data is skewed or contains non-normal distributions.

16. **np.dot()**

* Performs **dot product** for 1-D arrays.
* For 2-D arrays, it performs **matrix multiplication**.
* For arrays with more than 2 dimensions, it does not follow standard broadcasting and behaves inconsistently.

**np.matmul()**

* Introduced in NumPy 1.10.
* Always performs **matrix multiplication** (not element-wise dot product).
* Works with higher-dimensional arrays and supports **broadcasting**.

 Use **np.dot()** for simple dot products or 2D matrix multiplication where you don't need broadcasting.

 Use **np.matmul()** for clear and consistent matrix multiplication, especially with more than 2 dimensions.

 Use **@** for readable syntax in code that uses a lot of matrix math (recommended for Python 3.5+).

17. **Step 1: Represent as matrices**

import numpy as np

A = np.array([[2, 3], [3, 4]]) # Coefficient matrix

b = np.array([8, 11]) # Constants

**🔸 Step 2: Solve using np.linalg.solve()**

x = np.linalg.solve(A, b)

print(x) # Output: [1. 2.]

This gives x = 1 and y = 2.

18. SVD is widely used for **dimensionality reduction and data compression**, especially in image compression. The idea is to approximate the original matrix using only the top *k* singular values (and corresponding vectors), which retains most of the important information while significantly reducing storage.

19. **np.linalg.inv(): Matrix Inverse**

* Computes the **exact inverse** of a **square** (n×n), **non-singular** matrix.
* Fails (raises LinAlgError) if the matrix is singular (non-invertible) or not square.

**np.linalg.pinv(): Moore-Penrose Pseudo-Inverse**

* Computes the **pseudo-inverse**, which is a generalization of the inverse for **non-square** or **singular** matrices.
* Always returns a result using **SVD (Singular Value Decomposition)** under the hood

20. To efficiently apply a **rolling window operation** on a large dataset using NumPy, you can use the numpy.lib.stride\_tricks.sliding\_window\_view() function. This allows you to create overlapping views of the data **without copying memory**, making it both **fast** and **memory-efficient**.

21. NumPy can be effectively integrated into a **real-time data pipeline** by using it for **fast numerical computations**, **vectorized operations**, and **data transformations** as data streams through the system. Here’s how:

In a real-time pipeline (e.g., sensor feeds, financial tick data, or server logs), NumPy can process batches of incoming data efficiently. You can collect data in chunks (e.g., using a queue or a buffer), convert them to NumPy arrays, and apply transformations such as normalization, filtering, feature extraction, or statistical analysis on-the-fly.

For example, in a streaming setup using tools like **Kafka**, **ZeroMQ**, or **sockets**, the incoming data can be parsed and quickly loaded into NumPy arrays. Then, NumPy operations can be used to compute rolling averages, standard deviations, or detect anomalies in near real-time. You can also integrate NumPy with frameworks like **Pandas**, **Dask**, or **Numba** for better scalability and parallelism.

In summary, NumPy acts as the fast in-memory computation engine within the pipeline, making it ideal for performing low-latency analytics and transformations on streaming data.

22. In computer vision, images are represented as **NumPy arrays**, where each pixel's intensity or color is stored as numerical values. NumPy provides powerful tools to efficiently process these arrays for tasks like filtering, transformation, and feature extraction.

23. To ensure **data type consistency** across multiple NumPy arrays, you can explicitly set and check the data types (dtype) during array creation and before performing operations. This helps avoid unexpected behavior, especially in mathematical computations or when combining arrays.

24. In multi-threaded applications, common NumPy pitfalls include:

1. **GIL (Global Interpreter Lock)**: Python threads are limited by the GIL, so they can't execute NumPy code in parallel effectively. Use multiprocessing or tools like **Joblib** instead.
2. **Thread Safety**: NumPy operations are thread-safe when threads access different data but can cause issues if they modify the same array concurrently.
3. **Memory Contention**: Multiple threads accessing large arrays can degrade performance due to cache contention.
4. **Internal Multi-threading**: NumPy can use multi-threading internally (via libraries like OpenBLAS), and using it inside threads may cause over-subscription, reducing performance.

To avoid these issues, control thread usage via environment variables (e.g., OMP\_NUM\_THREADS=1) and use multiprocessing for parallel tasks.