50 Hard NumPy Questions & Answers (Q&A)

Basics & Array Creation

1. **Q:** How to create a NumPy array?
   A: import numpy as np; arr = np.array([1,2,3]).
2. **Q:** Difference between Python list and NumPy array?
   A: Here's a **clear, interview-friendly comparison** between **Python list** and **NumPy array**:

Feature	**Python List**	**NumPy Array**
Library	Built-in Python	Requires numpy library
Data Type	Can store **heterogeneous elements** (different types)	Stores **homogeneous elements** (same type)
Performance	Slower for numerical computations	Faster due to **optimized C implementation**
Memory Efficiency	Less memory-efficient	More memory-efficient; stores data in **contiguous blocks**
Mathematical Operations	Needs **loops** for element-wise operations	Supports **vectorized operations** without loops
Methods/Functions	Basic methods like append, remove, pop	Rich set of mathematical, statistical, and linear algebra functions (np.sum, np.mean, np.dot)
Slicing/Indexing	Standard Python slicing	Supports **advanced slicing, boolean indexing, and broadcasting**
Example	python\nlst = [1, 2, 3]\nlst * 2 # [1,2,3,1,2,3]	python\nimport numpy as np\narr = np.array([1,2,3])\narr * 2 # [2,4,6]

3. **Q:** How to create an array of zeros or ones?
   A: np.zeros((3,3)), np.ones((3,3)).
4. **Q:** Create an identity matrix?
   A: np.eye(3) or np.identity(3).
5. **Q:** How to create arrays with a range of numbers?
   A: np.arange(0,10,2) → 0,2,4,6,8.
6. **Q:** How to create linearly spaced numbers?
   A: np.linspace(0,1,5) → 5 numbers from 0 to 1.
7. **Q:** How to create random arrays?
   A: np.random.rand(3,3), np.random.randint(0,10,(3,3)).
8. **Q:** Difference between np.array and np.asarray?
   A: asarray avoids copying if input is already an array.

   Here's a clear and concise explanation of **np.array vs np.asarray** in NumPy:

Feature	np.array	np.asarray
Purpose	Creates a **new NumPy array** from data, **always copying** by default (can avoid with copy=False)	Converts input to a NumPy array **without copying if input is already an ndarray**
Copy Behavior	Makes a **copy** of the data by default	Makes a **view** of the input if it's already an ndarray (no new memory allocated)
Use Case	When you want a **completely independent array**	When you want **minimal memory overhead** and can work with the original array
Example	python\nimport numpy as np\nlst = [1,2,3]\narr1 = np.array(lst)\narr1[0] = 100\nprint(lst) # [1,2,3]	python\nimport numpy as np\narr2 = np.asarray(arr1)\narr2[0] = 200\nprint(arr1) # [200,2,3]

9. **Q:** How to check array shape and size?
   A: arr.shape, arr.size.
10. **Q:** Change array shape without changing data?
    A: arr.reshape((new_rows,new_cols)).

Indexing & Slicing

11. **Q:** How to access elements in NumPy array?
    A: arr[0], arr[1,2] (for 2D).
12. **Q:** Slice rows and columns?
    A: arr[0:2,1:3].
13. **Q:** Boolean indexing?
    A: arr[arr>5] → elements >5.
14. **Q:** Fancy indexing?
    A: arr[[0,2],[1,3]] → selects specific elements.
15. **Q:** How to select entire row or column?
    A: Row → arr[0,:]; Column → arr[:,1].
16. **Q:** Negative indexing?
    A: arr[-1] → last element; arr[-2,:] → second last row.
17. **Q:** Difference between view and copy?
    A: View → shares data; copy → independent array.

    Here's a clear explanation of **view vs copy** in NumPy:

Feature	View	Copy
Definition	A **new array object** that **shares the same data** as the original array	A **new array object** with **its own independent data**

Feature	View	Copy
Memory	No new memory is allocated for the data	Allocates new memory and duplicates the data
Effect on Original	Modifying the view **affects the original array**	Modifying the copy **does not affect the original array**
Creation	Using .view() or slicing	Using .copy()
Use Case	Efficient operations without extra memory	When you need a **completely independent array**
Example	python\nimport numpy as np\narr = np.array([1,2,3])\nv = arr.view()\nv[0] = 100\nprint(arr) # [100,2,3]	python\nc = arr.copy()\nc[0] = 200\nprint(arr) # [100,2,3]

Key Points

1. **View** → shares data, low memory, changes reflect in original.
2. **Copy** → independent data, safe for modifications, uses more memory.

---

18. **Q:** How to flatten an array?
    A: arr.flatten() or arr.ravel().
19. **Q:** How to transpose a matrix?
    A: arr.T.
20. **Q:** How to add/remove dimensions?
    A: arr[:,np.newaxis] → add; np.squeeze(arr) → remove.

    Here's a clear explanation of **adding and removing dimensions in NumPy**:

---

1. Adding Dimensions

a) Using np.newaxis / None

- Adds a new axis (dimension) to an array.

```
import numpy as np


arr = np.array([1, 2, 3])  # Shape: (3,)

arr2 = arr[:, np.newaxis]  # Shape: (3, 1)

arr3 = arr[np.newaxis, :]  # Shape: (1, 3)


print(arr2.shape)  # (3,1)
```

```
print(arr3.shape)  # (1,3)
```

b) Using reshape

```
arr2 = arr.reshape(3, 1)

arr3 = arr.reshape(1, 3)
```

c) Using expand_dims

```
arr2 = np.expand_dims(arr, axis=1)  # Adds a new axis at position 1
```

---

2. Removing Dimensions

a) Using squeeze

- Removes axes with size 1.

```
arr = np.array([[1], [2], [3]])  # Shape: (3,1)

arr_squeezed = np.squeeze(arr)   # Shape: (3,)


print(arr_squeezed.shape)  # (3,)
```

b) Using reshape

- Reshape to the desired shape without the unwanted dimension.

```
arr_squeezed = arr.reshape(3,)
```

Key Points

1. **Add dimension** → np.newaxis, reshape, expand_dims
2. **Remove dimension** → squeeze, reshape
3. Adding/removing dimensions is often needed for **broadcasting** or **matrix operations**.

Mathematical Operations

21. **Q:** Add, subtract, multiply, divide arrays?
    A: a+b, a-b, a*b, a/b.
22. **Q:** Element-wise vs matrix multiplication?
    A: * → element-wise; @ or np.dot() → matrix multiplication.
23. **Q:** Sum, mean, std, min, max?
    A: np.sum(arr), np.mean(arr), np.std(arr), np.min(arr), np.max(arr).
24. **Q:** Axis parameter in functions?
    A: Axis=0 → column-wise; Axis=1 → row-wise.

25. **Q:** How to find unique elements?
    A: np.unique(arr).
26. **Q:** How to sort an array?
    A: np.sort(arr) (returns sorted copy).
27. **Q:** How to find indices of max/min?
    A: np.argmax(arr), np.argmin(arr).
28. **Q:** Cumulative sum or product?
    A: np.cumsum(arr), np.cumprod(arr).
29. **Q:** Difference between np.sum(arr) and arr.sum()?
    A: Functionally same; method → object-oriented style.

    Here's a clear and concise explanation of the difference between **np.sum(arr)** and **arr.sum()** in NumPy:

Feature	**np.sum(arr)**	**arr.sum()**
Type	**NumPy function** (standalone)	**Method** of ndarray object
Syntax	np.sum(arr, axis=..., dtype=..., keepdims=...)	arr.sum(axis=..., dtype=..., keepdims=...)
Flexibility	Can work on **any array-like object**, e.g., lists or tuples	Works **only on ndarray objects**
Functionality	Same as method for ndarray	Same as function for ndarray
Example	python\nimport numpy as np\narr = np.array([1,2,3])\nprint(np.sum(arr)) # 6	python\nprint(arr.sum()) # 6

Key Points

1. Both **give the same result for ndarrays**.
2. np.sum is more general; arr.sum() is object-oriented and only works on arrays.
3. Both support **axis, dtype, keepdims** parameters.

Interview-Friendly 1-Line

np.sum(arr) is a general NumPy function that works on arrays or array-like objects, while arr.sum() is a method of ndarray objects; both produce the same result for ndarrays. ✅

---

30. **Q:** Element-wise comparison → greater, less?
    A: arr>5, arr<=10 → returns boolean array.

Linear Algebra

31. **Q:** How to calculate dot product?
    A: np.dot(a,b) or a@b.
32. **Q:** Matrix determinant?
    A: np.linalg.det(matrix).
33. **Q:** Matrix inverse?
    A: np.linalg.inv(matrix).
34. **Q:** Eigenvalues and eigenvectors?
    A: np.linalg.eig(matrix).
35. **Q:** Solve linear equations Ax=b?
    A: np.linalg.solve(A,b).
36. **Q:** Singular Value Decomposition?
    A: U,S,V=np.linalg.svd(matrix).
37. **Q:** Trace of a matrix?
    A: np.trace(matrix).
38. **Q:** Rank of a matrix?
    A: np.linalg.matrix_rank(matrix).
39. **Q:** Norm of a vector?
    A: np.linalg.norm(v).
40. **Q:** Cross product of two vectors?
    A: np.cross(a,b).

Advanced / Tricky

41. **Q:** Difference between ravel() and flatten()?
    A: ravel() → view; flatten() → copy.

Feature	`ravel()`	`flatten()`
Return Type	Returns a **view** of the original array whenever possible	Returns a **copy** of the array
Memory Efficiency	More memory-efficient (no copy if possible)	Less memory-efficient (always a new array)
Modifying Returned Array	Modifying may affect the original array (if it's a view)	Modifying the flattened array **does not affect the original array**
Syntax	`arr.ravel(order='C')`	`arr.flatten(order='C')`
Use Case	When you want a **1D view** without extra memory	When you need a **1D independent copy**

Example

```
import numpy as np

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

# ravel
```

```
r = arr.ravel()
r[0] = 100
print(arr)  # [[100   2]
            #  [  3   4]]  (original modified)

# flatten
f = arr.flatten()
f[0] = 200
print(arr)  # [[100   2]
            #  [  3   4]]  (original unchanged)
```

Interview-Friendly 1-Line

*`ravel()` returns a 1D **view** of the array (may affect original), while `flatten()` returns a 1D **copy** (original remains unchanged).* ✅

42. **Q:** Memory layout C vs F order?
    A: C → row-major; F → column-major.
43. **Q:** How to concatenate arrays?
    A: np.concatenate([a,b], axis=0).
44. **Q:** How to split arrays?
    A: np.split(arr,3).
45. **Q:** Broadcasting in NumPy?
    A: Arrays of different shapes operated element-wise if compatible.

    Here's a clear explanation of **broadcasting in NumPy**:

    ---

    1. Definition

    Broadcasting is a technique that allows **NumPy to perform arithmetic operations on arrays of different shapes** by automatically **stretching the smaller array** to match the shape of the larger one, **without copying data**.

    ---

    2. Rules of Broadcasting

1. If arrays have **different dimensions**, prepend **1s to the shape of the smaller array**.
2. Two dimensions are compatible if:
    - They are equal, **or**
    - One of them is **1**.
3. NumPy "stretches" the array with dimension 1 along that axis.

    ---

    3. Example

    import numpy as np

```
A = np.array([[1, 2, 3],
              [4, 5, 6]])  # Shape: (2,3)
B = np.array([10, 20, 30])  # Shape: (3,)

C = A + B  # B is broadcasted to shape (2,3)
print(C)
```

Output:

```
[[11 22 33]
 [14 25 36]]
```

- Here, B is automatically stretched along rows to match A.

---

4. Another Example with 2D & 1D

```
X = np.array([[1], [2], [3]])  # Shape: (3,1)
Y = np.array([10, 20, 30])     # Shape: (3,)
Z = X + Y  # X is broadcast to (3,3), Y is broadcast to (3,3)
print(Z)
```

Output:

```
[[11 21 31]
 [12 22 32]
 [13 23 33]]
```

---

5. Key Points

1. Broadcasting avoids **explicit loops** → faster code.
2. Useful for **vectorized operations** on arrays of different shapes.
3. Always follow the **broadcasting rules** to avoid errors.

Interview-Friendly 1-Line

Broadcasting in NumPy lets arrays of different shapes participate in arithmetic operations by automatically stretching the smaller array to match the larger one. ✅

46. **Q:** How to mask array elements?
A: arr[arr>5]=0 → mask with condition.

Here's a clear explanation of **masking array elements in NumPy**:

---

1. What is Masking?

Masking means **selecting elements of an array based on a condition** and optionally **modifying or using only those elements**.

- It is also called **boolean indexing**.

---

2. How to Mask Elements

a) Using a Boolean Condition

```
import numpy as np


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


# Mask elements greater than 3
mask = arr > 3
print(mask)        # [False False False  True  True]
print(arr[mask])    # [4 5]
```

- You can combine conditions using & (and) and | (or):

```
print(arr[(arr > 2) & (arr < 5)])  # [3 4]
```

---

b) Using np.where()

- np.where(condition, x, y) returns **elements from x or y based on the condition**.

```
arr_new = np.where(arr > 3, arr*10, arr)
print(arr_new)  # [1 2 3 40 50]
```

---

c) Masking Multi-Dimensional Arrays

```
arr2 = np.array([[1,2,3],
                 [4,5,6]])


mask2 = arr2 % 2 == 0  # Mask even numbers
print(arr2[mask2])     # [2 4 6]
```

---

3. Key Points

1. Masking does **not modify the original array** unless you assign back.
2. Supports **complex conditions** with logical operators.

3. Very useful for **filtering, replacing, or analyzing subsets of data**.

Interview-Friendly 1-Line

Masking in NumPy selects elements based on a condition using boolean indexing or np.where(), allowing you to filter or modify specific array elements. ✅

47. **Q:** Difference between np.copy() and assignment =?
A: = → reference; np.copy() → independent array.

Here's a clear explanation of the **difference between np.copy() and simple assignment (=) in NumPy**:

Feature	**Assignment (=)**	**np.copy()**
Operation	Creates a **new reference** to the same array	Creates a **new independent array** (deep copy)
Memory	Both variables point to **same memory**	Allocates **separate memory** for the new array
Effect of Modification	Changing one array **affects the other**	Changing one array **does not affect the other**
Use Case	Quick reference without duplicating data	Needed when you want a **true independent copy**
Example	python\nimport numpy as np\narr = np.array([1,2,3])\narr2 = arr\narr2[0] = 100\nprint(arr) # [100,2,3]	python\narr_copy = np.copy(arr)\narr_copy[0] = 200\nprint(arr) # [100,2,3]

Key Points

1. = → **shallow assignment**, both names share the same data.
2. np.copy() → **deep copy**, independent array.
3. Always use np.copy() when you need to **preserve the original array**.

Interview-Friendly 1-Line

= creates a new reference to the same array (changes reflect in both), while np.copy() creates an independent array that can be modified safely. ✅

48. **Q:** How to save/load arrays?
A: np.save('file.npy',arr), arr=np.load('file.npy').
49. **Q:** Difference between np.where and boolean indexing?
A: np.where(condition) → indices; boolean indexing → values.

50. **Q:** How to generate random normal distribution?
    A: np.random.randn(3,3) → standard normal; np.random.normal(mean,std,(3,3)) → custom.
