🟦 1. Introduction to NumPy

🔷 What is NumPy?

- **NumPy (Numerical Python)** is a powerful Python library used for numerical and scientific computing.
- It provides a **multidimensional array object** called `ndarray`, which is much faster and more efficient than Python's built-in lists.
- NumPy is the foundation for many other libraries in **data science, machine learning, and scientific computing** (such as Pandas, SciPy, Scikit-learn, TensorFlow, PyTorch).

👉 In short: NumPy = **fast math with arrays + a huge toolbox for linear algebra, statistics, and random numbers**.

🔷 Why NumPy vs Python Lists?

1. **Speed**

- NumPy is written in **C** under the hood, so operations are executed much faster than pure Python lists.

Example:

```
import numpy as np
import time

# Using Python list
python_list = list(range(1_000_000))
start = time.time()
[ x*2 for x in python_list ]
print("Python list time:", time.time() - start)

# Using NumPy array
numpy_array = np.arange(1_000_000)
start = time.time()
numpy_array * 2
print("NumPy array time:", time.time() - start)
```

➡️ NumPy will be **10–100x faster** depending on the operation.

---

2. **Memory Efficiency**

- A Python list stores each element as a **full Python object** with extra overhead.
- A NumPy array stores data in a **continuous block of memory** with fixed data types, making it much more memory efficient.

Example:

```
import numpy as np
import sys

# Python list of 1 million integers
python_list = list(range(1_000_000))
print("Size of Python list:", sys.getsizeof(python_list))

# NumPy array of 1 million integers
numpy_array = np.arange(1_000_000)
print("Size of NumPy array:", numpy_array.nbytes)
```

➡️ NumPy arrays take **much less memory**.

---

3. **Functionality**

- Python lists only support basic operations like append, insert, etc.
- NumPy arrays support:
  - **Vectorized operations** (e.g., `array * 2` works on all elements at once)

- **Mathematical functions** (`sin`, `log`, `exp`, etc.)
- **Linear algebra** (`dot`, `inv`, eigenvalues)
- **Random numbers**
- **Statistics** (mean, median, variance, etc.)

Example:

```
import numpy as np

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

print("Multiply by 2:", arr * 2)
print("Square:", arr ** 2)
print("Mean:", np.mean(arr))
print("Sine:", np.sin(arr))
```

🔷 Installing NumPy

You can install NumPy in your Python environment using **pip**:

```
pip install numpy
```

If you are using **Anaconda**, NumPy usually comes pre-installed. If not, install via:

```
conda install numpy
```

Then verify installation in Python:

```
import numpy as np
print(np.__version__)
```

```
Start coding or generate with AI.
```

🟦 2. NumPy Basics (Full Guide)

🔷 2.1 Importing NumPy

```
import numpy as np
```

⚡ **Tip:** Always use `np` as alias (standard in the Python community).

🔷 2.2 Creating Arrays

1. From Lists and Tuples

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

- NumPy will **infer dtype automatically**.
- Override with `dtype`:

```
arr = np.array([1, 2, 3], dtype=np.float32)
```

⚡ **Tip:** Use `dtype` wisely for memory efficiency in big data.

---

2. Predefined Arrays

```
zeros = np.zeros((2, 3))     # All zeros
ones = np.ones((3, 2))       # All ones
full = np.full((2, 2), 9)    # All filled with 9
```

⚡ **Tip:** Faster for initializing matrices before computations.

---

3. Identity and Diagonal

```
I = np.eye(3)                # Identity matrix
diag = np.diag([5, 10, 15])  # Custom diagonal
```

⚡ **Tip:** Very useful for **linear algebra**.

---

4. Range-Based Arrays

```
a = np.arange(0, 10, 2)     # [0,2,4,6,8]
b = np.linspace(0, 1, 5)    # 5 values evenly spaced
c = np.logspace(1, 3, 3)    # [10, 100, 1000]
```

⚡ **Tip:** Use `linspace` in plotting & simulation (smooth values).

---

5. Random Arrays

```
np.random.seed(42)  # reproducibility

uniform = np.random.rand(2, 3)     # Uniform [0,1)
normal = np.random.randn(2, 3)     # Standard normal
integers = np.random.randint(1, 10, size=(3, 3))  # 1-9
choice = np.random.choice([10, 20, 30], size=5)   # Random pick
```

⚡ **Tip:** Always fix `seed` in experiments for **reproducible results**.

---

6. Special Arrays

```
empty = np.empty((2, 2))  # Random memory garbage
identity = np.identity(4)
```

⚠️ **Warning:** `np.empty()` is fast but contains **uninitialized values**.

🔷 2.3 Array Attributes

```
arr = np.arange(12).reshape(3, 4)

print("Array:\n", arr)
print("Shape:", arr.shape)     # (3,4)
print("Dimensions:", arr.ndim) # 2
print("Size:", arr.size)       # 12
print("Data type:", arr.dtype)
print("Item size:", arr.itemsize, "bytes")
print("Total memory:", arr.nbytes, "bytes")
```

⚡ **Tip:** Use `arr.shape` unpacking:

```
rows, cols = arr.shape
```

🔷 2.4 Reshaping Arrays

```
arr = np.arange(12)         # 0..11
reshaped = arr.reshape(3, 4)

print("Reshaped:\n", reshaped)
```

Flattening:

```
print("Flatten:", reshaped.flatten())  # copy
print("Ravel:", reshaped.ravel())      # view
```

⚡ **Tip:** Use `reshape(-1)` to flatten quickly.

🔷 2.5 Copy vs View

```
a = np.array([1, 2, 3])
b = a.view()   # view (linked to original)
c = a.copy()   # copy (independent)

a[0] = 99
print("Original:", a)  # [99, 2, 3]
print("View:", b)      # [99, 2, 3]
print("Copy:", c)      # [1, 2, 3]
```

⚡ **Tip:** Always use `.copy()` if you don't want accidental modifications.

🔷 2.6 Data Type Casting

```
arr = np.array([1.5, 2.8, 3.9])
int_arr = arr.astype(int)
print(int_arr)  # [1, 2, 3]
```

⚡ **Tip:** `.astype()` is critical in ML preprocessing (float32 for GPUs).

🟩 Examples (Zero → Hero)

✅ Beginner: Create & Inspect

```
arr = np.arange(1, 10).reshape(3, 3)
print("Array:\n", arr)
print("Mean:", np.mean(arr))
print("Sum of all elements:", arr.sum())
```

✅ Intermediate: Dice Simulator

```
np.random.seed(0)
dice = np.random.randint(1, 7, size=20)
print("Dice rolls:", dice)
print("Frequency of 6:", np.count_nonzero(dice == 6))
```

✅ Advanced: Monte Carlo π

```
np.random.seed(42)
N = 1_000_000
x, y = np.random.rand(N), np.random.rand(N)

inside = (x**2 + y**2) <= 1
pi_est = 4 * np.sum(inside) / N
```

```
print("Estimated π:", pi_est)
```

✅ Hero: Gradient Image

```
import matplotlib.pyplot as plt

gradient = np.linspace(0, 1, 256)
image = np.outer(gradient, gradient)

plt.imshow(image, cmap="viridis")
plt.colorbar()
plt.title("Generated Gradient")
plt.show()
```

📝 Exercises

🔰 Beginner

1. Create a 1D array from 10 to 50 with step 5.
2. Create a 3x3 array filled with True (boolean dtype).
3. Create an array of 100 random numbers between 0 and 1 and find its mean.

⚡ Intermediate

4. Create a 4x4 identity matrix and replace the diagonal with numbers 1–4.
5. Create a 5x5 array with values from 0 to 24, then reshape it into 25x1.
6. Generate 10 random integers between 1 and 100. Find the max, min, and mean.

🚀 Advanced

7. Create a 6x6 checkerboard pattern (0s and 1s alternating).
8. Generate 1 million random numbers (normal distribution) and calculate mean & std.
9. Simulate tossing 2 dice 1000 times and estimate the probability of getting a sum of 7.

```
Start coding or generate with AI.
```

📝 Exercises with Solutions

🔰 Beginner

1. Create a 1D array from 10 to 50 with step 5.

```
import numpy as np

arr = np.arange(10, 55, 5)
print(arr)   # [10 15 20 25 30 35 40 45 50]
```

✅ **Tip:** `np.arange(start, stop, step)` is like Python `range()` but returns an array.

---

2. Create a 3x3 array filled with True (boolean dtype).

```
arr = np.ones((3, 3), dtype=bool)
print(arr)
```

✅ **Tip:** `np.ones` + `dtype=bool` is a quick trick for boolean masks.

---

3. Create an array of 100 random numbers between 0 and 1 and find its mean.

```
arr = np.random.rand(100)
print("Mean:", arr.mean())
```

✅ **Tip:** NumPy arrays have built-in aggregation methods (`.mean()`, `.sum()`).

⚡ Intermediate

4. Create a 4x4 identity matrix and replace the diagonal with numbers 1–4.

```
arr = np.eye(4)
arr[np.arange(4), np.arange(4)] = [1, 2, 3, 4]
print(arr)
```

✅ **Tip:** Use `arr[np.arange(n), np.arange(n)]` to select diagonals efficiently.

---

5. Create a 5x5 array with values from 0 to 24, then reshape it into 25x1.

```
arr = np.arange(25).reshape(5, 5)
print("5x5:\n", arr)

reshaped = arr.reshape(25, 1)
print("25x1:\n", reshaped)
```

✅ **Tip:** `reshape(-1, 1)` is a shortcut to flatten into a column vector.

---

6. Generate 10 random integers between 1 and 100. Find the max, min, and mean.

```
arr = np.random.randint(1, 101, 10)
print("Numbers:", arr)
print("Max:", arr.max())
print("Min:", arr.min())
print("Mean:", arr.mean())
```

✅ **Tip:** NumPy has `arr.max()`, `arr.min()`, `arr.mean()` optimized in C.

🚀 Advanced

7. Create a 6x6 checkerboard pattern (0s and 1s alternating).

```
arr = np.zeros((6, 6), dtype=int)
arr[1::2, ::2] = 1   # odd rows, even cols
arr[::2, 1::2] = 1   # even rows, odd cols
print(arr)
```

✅ **Tip:** Use **slicing tricks** (`::2`) for pattern generation.

---

8. Generate 1 million random numbers (normal distribution) and calculate mean & std.

```
arr = np.random.randn(1_000_000)
print("Mean:", arr.mean())
print("Standard Deviation:", arr.std())
```

✅ **Tip:** For large data, NumPy is **much faster** than Python's `statistics` module.

---

9. Simulate tossing 2 dice 1000 times and estimate the probability of getting a sum of 7.

```
dice1 = np.random.randint(1, 7, 1000)
dice2 = np.random.randint(1, 7, 1000)
sums = dice1 + dice2
```

```
prob_7 = np.count_nonzero(sums == 7) / 1000
print("Estimated Probability of 7:", prob_7)
```

✅ **Tip:** Vectorized operations (`dice1 + dice2`) make this super efficient.

🎯 Summary (NumPy Basics Module)

By now you've learned: ✓ How to create arrays (zeros, ones, random, ranges, identity, etc.) ✓ How to check attributes (shape, size, dtype, memory) ✓ How to reshape, flatten, copy, and cast arrays ✓ Real applications (dice rolls, Monte Carlo, image generation) ✓ Solved exercises from beginner to advanced

```
Start coding or generate with AI.
```

📘 Section 3: Array Attributes in NumPy

When working with NumPy, understanding the **properties of arrays** is essential. These attributes help you **inspect, reshape, and manipulate arrays efficiently**.

🔷 3.1 Basic Attributes

Let's create a sample array:

```
import numpy as np

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

Output:

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

1. Shape (`.shape`)

Returns the **dimensions** of the array as a tuple.

```
print("Shape:", arr.shape)  # (2, 3)
```

✅ This means 2 rows × 3 columns.

---

2. Size (`.size`)

Returns the **total number of elements**.

```
print("Size:", arr.size)  # 6
```

---

3. Dimensions (`.ndim`)

Returns the number of **dimensions (axes)**.

```
print("Dimensions:", arr.ndim)  # 2
```

---

4. Data Type (`.dtype`)

Returns the **type of elements** in the array.

```
print("Data Type:", arr.dtype)  # int64 (depends on system)
```

5. Item Size & Total Memory

```
print("Item size (bytes):", arr.itemsize)
print("Total size (bytes):", arr.nbytes)
```

✅ Handy for memory optimization.

⚡ **Tip:** You can convert data type with `arr.astype(new_dtype)`

```
float_arr = arr.astype(float)
print(float_arr.dtype)  # float64
```

🔷 3.2 Reshaping Arrays

Sometimes, you want to change the **shape** without changing data.

1. reshape()

```
arr = np.arange(12)
reshaped = arr.reshape(3, 4)
print(reshaped)
```

Output:

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
```

⚡ **Trick:** Use `-1` to let NumPy infer the dimension.

```
print(arr.reshape(2, -1))   # 2 rows, NumPy auto-calculates columns
```

---

2. ravel()

Flattens array into **1D view** (does not copy data).

```
flat_view = reshaped.ravel()
flat_view[0] = 99
print("Flat view:", flat_view)
print("Original:", reshaped)   # Also changes!
```

---

3. flatten()

Flattens array into **1D copy** (independent).

```
flat_copy = reshaped.flatten()
flat_copy[0] = -1
print("Flat copy:", flat_copy)
print("Original:", reshaped)   # Not affected
```

⚡ **Rule of Thumb:**

- `ravel()` → faster, returns a **view** (changes affect original).
- `flatten()` → safer, returns a **copy** (original stays intact).

🔷 3.3 Practical Examples

Example 1: Image Reshaping

If you have a grayscale image with 28×28 pixels:

```
img = np.arange(784).reshape(28, 28)
print(img.shape)  # (28, 28)

flat_img = img.ravel()
print(flat_img.shape)  # (784,)
```

✅ Used in **deep learning (MNIST dataset)** where each image must be flattened.

Example 2: Converting Sensor Data

Suppose you receive **a stream of 1D sensor values** (100 readings):

```
data = np.arange(100)
reshaped = data.reshape(10, 10)  # group into 10 samples × 10 features
```

✅ This way, machine learning models can process it in batches.

🔷 3.4 Exercises

Beginner

1. Create a 1D array of 15 elements and check: shape, size, ndim, and dtype.
2. Convert an array of integers into float type.

Intermediate

3. Create a 3×4 array and reshape it into:
   - (2, 6)
   - (6, 2, 1)
4. Flatten an array using both `ravel()` and `flatten()`. Show the difference by modifying one element.

Advanced

5. Create a 3D array of shape (2, 3, 4).
   - Find its size, ndim, and reshape it into (4, 6).
6. Generate a (100,) array of random values. Reshape it into (10, 10) and calculate the mean of each row.

🔷 3.5 Solutions

Beginner

```
arr = np.arange(15)
print(arr.shape)  # (15,)
print(arr.size)   # 15
print(arr.ndim)   # 1
print(arr.dtype)  # int64
```

```
float_arr = arr.astype(float)
print(float_arr.dtype)  # float64
```

Intermediate

```
arr = np.arange(12).reshape(3, 4)
print(arr.reshape(2, 6))
print(arr.reshape(6, 2, 1))
```

```
flat_view = arr.ravel()
flat_copy = arr.flatten()

flat_view[0] = 999
print("Original after ravel change:\n", arr)

flat_copy[0] = -1
print("Original after flatten change:\n", arr)
```

Advanced

```
arr = np.arange(24).reshape(2, 3, 4)
print("Size:", arr.size)   # 24
print("Dimensions:", arr.ndim)  # 3

reshaped = arr.reshape(4, 6)
print(reshaped.shape)
```

```
data = np.random.rand(100).reshape(10, 10)
row_means = data.mean(axis=1)
print("Row means:", row_means)
```

✅ With this, you now master:

- **Array inspection (shape, size, ndim, dtype)**
- **Reshaping arrays safely**
- **Flattening (ravel vs flatten)**
- **Practical use-cases (ML, images, sensors)**

```
Start coding or generate with AI.
```

🟦 3. Array Attributes in NumPy

NumPy arrays (`ndarray`) come with built-in **attributes** that describe their structure. These attributes tell us about **dimensions, shape, size, and data type**.

🔷 3.1 Shape, Size, ndim, and dtype

✅ `shape`

- Returns a **tuple** showing the number of elements in each dimension.

```
import numpy as np

arr = np.array([[1, 2, 3], [4, 5, 6]])
print("Array:\n", arr)
print("Shape:", arr.shape)   # (2 rows, 3 columns)
```

✅ `size`

- Returns the **total number of elements** in the array.

```
print("Size:", arr.size)  # 6 elements in total
```

✅ `ndim`

- Returns the **number of dimensions** (axes) of the array.

```
print("Dimensions:", arr.ndim)  # 2 (rows and columns)
```

✅ `dtype`

- Shows the **data type** of array elements.

```
print("Data type:", arr.dtype)   # int64 (on most systems)
```

👉 You can also **force a dtype** when creating arrays:

```
arr_float = np.array([1, 2, 3], dtype=np.float32)
print("Forced dtype:", arr_float.dtype)
```

🔷 3.2 Reshaping Arrays

Reshaping allows you to **change the structure** of an array without changing its data.

✅ `reshape()`

- Creates a new array with a new shape (must have same total elements).

```
arr = np.arange(12)    # [0,1,2,...,11]
reshaped = arr.reshape(3, 4)
print("Reshaped (3x4):\n", reshaped)
```

👉 Tip: Use `-1` as a placeholder to let NumPy calculate automatically.

```
arr = np.arange(12)
reshaped = arr.reshape(2, -1)   # 2 rows, auto columns
print("Reshaped with -1:\n", reshaped)
```

✅ `ravel()`

- Returns a **flattened 1D array**, but gives a **view (reference)**, not a copy.
- Changing the flattened array may affect the original.

```
arr = np.array([[1, 2], [3, 4]])
ravelled = arr.ravel()
print("Ravelled:", ravelled)
ravelled[0] = 99
print("Modified original:\n", arr)   # arr changes too
```

✅ `flatten()`

- Returns a **copy** of the array in 1D form.
- Changing it does **not** affect the original.

```
arr = np.array([[1, 2], [3, 4]])
flattened = arr.flatten()
flattened[0] = 99
print("Flattened:", flattened)
print("Original stays same:\n", arr)
```

🔷 3.3 Tips & Tricks

✅ Use `.shape` before reshaping to avoid mismatches. ✅ Prefer `ravel()` over `flatten()` for efficiency (when you don't need a copy). ✅ Use `reshape(-1)` to quickly flatten arrays. ✅ Use `.ndim` when writing general functions to handle arrays of any dimension.

🔷 3.4 Examples from Zero to Hero

Example 1: Reshape for ML Dataset

Suppose you load 28×28 images (like MNIST dataset) and need to flatten them into a 1D vector for training:

```
images = np.arange(28*28*10).reshape(10, 28, 28)   # 10 images
print("Original shape:", images.shape)   # (10, 28, 28)

flattened = images.reshape(10, -1)   # (10, 784)
print("Flattened shape:", flattened.shape)
```

Example 2: Converting 1D → 2D Matrix for Math

```
arr = np.arange(9)
matrix = arr.reshape(3, 3)
print("Matrix:\n", matrix)
```

Example 3: Understanding Copy vs View

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

# Ravel (view)
r = arr.ravel()
r[0] = 100
print("After ravel modification:", arr)

# Flatten (copy)
f = arr.flatten()
f[1] = 200
print("After flatten modification:", arr)  # Original unchanged
```

🔷 3.5 Exercises

💡 Try these exercises to master attributes:

1. Create a `5x5` NumPy array of numbers `0-24`. Print its `shape`, `size`, `ndim`, and `dtype`.
2. Reshape it into a `25x1` column vector.
3. Use `ravel()` to flatten it, then modify one element and check if the original changed.
4. Do the same with `flatten()` and compare results.
5. Create a random array of shape `(2, 3, 4)`. Reshape it into `(4, 6)` using `-1`.

✅ Now you know **array attributes and reshaping tricks** — essential for any data science or ML workflow.

```
Start coding or generate with AI.
```

🟦 4. Indexing and Slicing in NumPy

NumPy arrays are **indexed and sliced** similar to Python lists, but with much more power.

🔷 1D Array Indexing

```
import numpy as np

arr = np.array([10, 20, 30, 40, 50])
print(arr[0])   # First element -> 10
print(arr[-1])  # Last element  -> 50
print(arr[2])   # Third element -> 30
```

✅ Works the same as Python lists.

🔷 2D Array Indexing

- For **2D arrays**, use `[row, column]` format.

```
arr2d = np.array([[10, 20, 30],
                  [40, 50, 60],
                  [70, 80, 90]])

print(arr2d[0, 0])  # 10 (1st row, 1st column)
print(arr2d[1, 2])  # 60 (2nd row, 3rd column)
print(arr2d[-1, -1]) # 90 (last row, last column)
```

🔷 3D Array Indexing

- For **3D arrays**, indexing goes `[depth, row, column]`.

```
arr3d = np.array([[[1, 2], [3, 4]],
                  [[5, 6], [7, 8]]])

print(arr3d[0, 0, 1])  # 2  (1st block, 1st row, 2nd col)
print(arr3d[1, 1, 0])  # 7  (2nd block, 2nd row, 1st col)
```

🔷 Slicing Arrays (start:stop:step)

1D Example:

```
arr = np.array([0, 10, 20, 30, 40, 50, 60])

print(arr[1:5])      # [10 20 30 40]   (from index 1 to 4)
print(arr[:4])       # [0 10 20 30]    (from start to 3)
print(arr[::2])      # [0 20 40 60]    (every 2nd element)
print(arr[::-1])     # [60 50 40 30 20 10  0] (reversed)
```

2D Example:

```
arr2d = np.array([[10, 20, 30, 40],
                  [50, 60, 70, 80],
                  [90, 100, 110, 120]])

print(arr2d[0:2, 1:3])  # [[20 30], [60 70]]
print(arr2d[:, 2])      # [30 70 110] (entire 3rd column)
print(arr2d[1, :])      # [50 60 70 80] (entire 2nd row)
```

🔷 Negative Indexing

- NumPy supports negative indexing to count from the end.

```
arr = np.array([10, 20, 30, 40, 50])
print(arr[-1])     # 50
print(arr[-3:])    # [30 40 50]
print(arr[:-2])    # [10 20 30]
```

🔷 Fancy Indexing (using lists/arrays)

- Fancy indexing lets you extract **multiple elements at once** using a list/array of indices.

```
arr = np.array([10, 20, 30, 40, 50])

print(arr[[0, 2, 4]])  # [10 30 50]
print(arr[[1, -1]])    # [20 50]
```

2D Example:

```
arr2d = np.array([[10, 20, 30],
                  [40, 50, 60],
                  [70, 80, 90]])

print(arr2d[[0, 2], [1, 2]])
# Picks (0,1) → 20 and (2,2) → 90 → [20 90]
```

🔷 Boolean Indexing

- Boolean indexing is super powerful for filtering arrays.

```
arr = np.array([10, 20, 30, 40, 50])

print(arr[arr > 25])   # [30 40 50]
print(arr[arr % 20 == 0]) # [20 40]
```

With 2D Arrays:

```
arr2d = np.array([[5, 10, 15],
                  [20, 25, 30],
                  [35, 40, 45]])

print(arr2d[arr2d > 20])  # [25 30 35 40 45]
```

⚡ Tips & Tricks

1. Use **slicing with steps** (`arr[::2]`) instead of loops.
2. Boolean indexing is great for **filtering** datasets (`arr[arr > threshold]`).
3. Fancy indexing can select **non-contiguous elements** efficiently.
4. `np.where(condition)` can combine Boolean and fancy indexing.

Example:

```
arr = np.array([5, 10, 15, 20, 25])
indices = np.where(arr > 12)
print(indices)     # (array([2, 3, 4]),)
print(arr[indices]) # [15 20 25]
```

📝 Exercises

Beginner

1. Create an array `[1, 2, 3, 4, 5, 6, 7, 8, 9]` and extract:
   - The last 3 elements
   - Every second element
   - The array reversed
2. Create a 3×3 array of numbers `1-9`. Extract:
   - The 2nd row
   - The 1st column
   - The bottom-right element

Intermediate

3. Create a 4×4 array of numbers `0–15`. Extract:
    - The submatrix `[[5, 6], [9, 10]]`
    - All even numbers
4. Using Boolean indexing, replace all numbers greater than 10 with `-1`.

Advanced (Hero 🦸)

5. Create a 5×5 array with numbers from `0–24`.
    - Extract the diagonal using fancy indexing.
    - Extract all border elements (top row, bottom row, first col, last col).
6. Given an array `[3, 6, 9, 12, 15, 18, 21]`:
    - Use `np.where` to find indices of numbers divisible by 9.
    - Replace those numbers with `99`.

```
Start coding or generate with AI.
```

🟦 4. Indexing and Slicing Exercises with Solutions

🔷 1. 1D Array Indexing

Q1. Create a 1D NumPy array of numbers 10 to 19. Print:

- the 1st element
- the last element
- elements from index 2 to 5

```
import numpy as np

arr = np.arange(10, 20)   # [10 11 12 13 14 15 16 17 18 19]

print("Array:", arr)

# First element
print("First element:", arr[0])

# Last element
print("Last element:", arr[-1])

# Elements from index 2 to 5
print("Elements 2 to 5:", arr[2:6])
```

✅ **Output:**

```
Array: [10 11 12 13 14 15 16 17 18 19]
First element: 10
Last element: 19
Elements 2 to 5: [12 13 14 15]
```

🔷 2. 2D Array Indexing

Q2. Create a 3×4 array with values from 1 to 12. Print:

- element at row 1, col 2
- entire 2nd row
- entire 3rd column

```
arr2d = np.arange(1, 13).reshape(3, 4)

print("2D Array:\n", arr2d)

# Element at row 1, col 2
print("Element (row=1, col=2):", arr2d[1, 2])

# Entire 2nd row
print("Second row:", arr2d[1])

# Entire 3rd column
print("Third column:", arr2d[:, 2])
```

✅ **Output:**

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

Element (row=1, col=2): 7
Second row: [5 6 7 8]
Third column: [ 3  7 11]
```

🔷 3. 3D Array Indexing

Q3. Create a 2×2×3 array with numbers 1–12. Print:

- first 2D block
- element at block 1, row 0, col 2

```
arr3d = np.arange(1, 13).reshape(2, 2, 3)

print("3D Array:\n", arr3d)

# First 2D block
print("First block:\n", arr3d[0])

# Element at block=1, row=0, col=2
print("Element [1,0,2]:", arr3d[1, 0, 2])
```

✅ **Output:**

```
3D Array:
 [[[ 1  2  3]
   [ 4  5  6]]

  [[ 7  8  9]
   [10 11 12]]]

First block:
 [[1 2 3]
  [4 5 6]]
Element [1,0,2]: 9
```

🔷 4. Slicing

Q4. For array `[0,1,2,3,4,5,6,7,8,9]`:

- elements from index 2 to 7 (exclusive)
- every 2nd element
- reverse the array

```
arr = np.arange(10)

print("Array:", arr)
print("Index 2 to 7:", arr[2:8])
print("Every 2nd element:", arr[::2])
print("Reversed:", arr[::-1])
```

✅ **Output:**

```
Array: [0 1 2 3 4 5 6 7 8 9]
Index 2 to 7: [2 3 4 5 6 7]
Every 2nd element: [0 2 4 6 8]
Reversed: [9 8 7 6 5 4 3 2 1 0]
```

🔷 5. Negative Indexing

Q5. From `[10, 20, 30, 40, 50]` print:

- last 3 elements using negative slicing
- last element

```
arr = np.array([10, 20, 30, 40, 50])

print("Last 3 elements:", arr[-3:])
print("Last element:", arr[-1])
```

✅ **Output:**

```
Last 3 elements: [30 40 50]
Last element: 50
```

🔷 6. Fancy Indexing

Q6. From `[10,20,30,40,50]`, select elements at positions 0, 2, and 4.

```
arr = np.array([10, 20, 30, 40, 50])
indices = [0, 2, 4]
print("Selected elements:", arr[indices])
```

✅ **Output:**

```
Selected elements: [10 30 50]
```

🔷 7. Boolean Indexing

Q7. From `[1,2,3,4,5,6,7,8,9,10]`, select:

- even numbers
- numbers greater than 5

```
arr = np.arange(1, 11)

print("Even numbers:", arr[arr % 2 == 0])
print("Greater than 5:", arr[arr > 5])
```

✅ **Output:**

```
Even numbers: [ 2  4  6  8 10]
Greater than 5: [ 6  7  8  9 10]
```

📝 Practice Exercises (Try Yourself)

1. Create a 2D array `5x5` with numbers 1–25. Extract:
   - the center element
   - the first row
   - the last column
2. Create a 3D array `3x3x3` with numbers 1–27. Extract:
   - the last block
   - the diagonal elements of the first block
3. From `[5,10,15,20,25,30,35,40]`, get:
   - numbers divisible by 10
   - reverse using slicing

```
Start coding or generate with AI.
```

5. Array Operations in NumPy

---

1. **Element-wise Operations**

NumPy supports **element-wise arithmetic** on arrays of the same shape.

```
import numpy as np

a = np.array([1, 2, 3, 4])
b = np.array([10, 20, 30, 40])

print(a + b)   # [11 22 33 44]
print(a - b)   # [-9 -18 -27 -36]
print(a * b)   # [10 40 90 160]
print(b / a)   # [10. 10. 10. 10.]
print(a ** 2)  # [ 1  4  9 16]
```

✅ **Tip:** NumPy automatically **vectorizes** operations → much faster than Python loops.

---

2. **Universal Functions (ufuncs)**

NumPy provides **fast mathematical functions** (ufuncs) that work element-wise.

```
arr = np.array([1, 4, 9, 16, 25])

print(np.sqrt(arr))   # Square root [1. 2. 3. 4. 5.]
print(np.exp(arr))    # Exponential
print(np.log(arr))    # Natural log
print(np.log10(arr))  # Log base 10
print(np.sin(arr))    # Trigonometric functions
print(np.cos(arr))
```

✅ **Tip:** Ufuncs are highly optimized in C → thousands of times faster than manual calculations.

---

3. **Aggregations**

NumPy allows **aggregation functions** to summarize arrays.

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

print(data.sum())   # 15
print(data.mean())  # 3.0
print(data.max())   # 5
```

```
print(data.min())   # 1
print(data.std())   # Standard deviation
print(data.var())   # Variance
```

---

4. **Axis Operations**

For multi-dimensional arrays, operations can be applied along **rows or columns**.

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

print(matrix.sum(axis=0))  # Sum by columns [12 15 18]
print(matrix.sum(axis=1))  # Sum by rows    [6 15 24]

print(matrix.mean(axis=0)) # Column mean [4. 5. 6.]
print(matrix.mean(axis=1)) # Row mean [2. 5. 8.]
```

✅ **Tip:** `axis=0` → column-wise, `axis=1` → row-wise.

📝 Exercises

Q1. Element-wise Operations

Given:

```
x = np.array([2, 4, 6, 8])
y = np.array([1, 3, 5, 7])
```

1. Add `x` and `y`.
2. Subtract `y` from `x`.
3. Multiply `x` and `y`.
4. Compute `x / y`.
5. Square all elements of `x`.

Q2. Universal Functions

Given:

```
arr = np.array([0, np.pi/2, np.pi])
```

1. Compute `sin(arr)`.
2. Compute `cos(arr)`.
3. Compute `tan(arr)`.

Q3. Aggregations

Given:

```
data = np.array([10, 20, 30, 40, 50])
```

1. Find the sum, mean, max, min, std, var.

Q4. Axis Operations

Given:

```
mat = np.array([[2, 4, 6],
                [1, 3, 5],
                [7, 9, 11]])
```

1. Find column sums.
2. Find row means.
3. Find max in each row.

✅ Solutions

Solution Q1

```
print(x + y)   # [ 3  7 11 15]
print(x - y)   # [1 1 1 1]
print(x * y)   # [ 2 12 30 56]
print(x / y)   # [2. 1.33333333 1.2 1.14285714]
print(x ** 2)  # [ 4 16 36 64]
```

Solution Q2

```
print(np.sin(arr))  # [0. 1. 0.]
print(np.cos(arr))  # [ 1. 0. -1.]
print(np.tan(arr))  # [ 0. 1.63312394e+16  0.] (≈ infinity at pi/2)
```

Solution Q3

```
print(data.sum())  # 150
print(data.mean()) # 30.0
print(data.max())  # 50
print(data.min())  # 10
print(data.std())  # 14.1421
print(data.var())  # 200.0
```

Solution Q4

```
print(mat.sum(axis=0))   # [10 16 22]
print(mat.mean(axis=1))  # [4. 3. 9.]
print(mat.max(axis=1))   # [ 6  5 11]
```

```
Start coding or generate with AI.
```

6. Broadcasting in NumPy

What is Broadcasting?

Broadcasting is a method that NumPy uses to **perform arithmetic operations on arrays of different shapes**. Instead of manually repeating values to match dimensions, NumPy automatically "broadcasts" smaller arrays across larger arrays so that they are compatible for element-wise operations.

Think of it as NumPy **stretching the smaller array** without making actual copies in memory → which is **fast and memory-efficient**.

Rules of Broadcasting

For two arrays to be broadcastable:

1. Compare their shapes starting from the **trailing dimensions**.
2. Two dimensions are compatible if:
   - They are equal, OR
   - One of them is 1.

If all dimensions match under these rules, broadcasting is possible.

✅ Examples of Broadcasting Rules:

- `(4, 3)` and `(3,)` → broadcast → `(4, 3)`
- `(5, 1)` and `(1, 6)` → broadcast → `(5, 6)`
- `(2, 3, 4)` and `(3, 1)` → broadcast → `(2, 3, 4)`
- ❌ `(2, 3)` and `(4,)` → Not compatible

Examples

1. Broadcasting with Scalars

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5])
print(arr + 10)   # Add scalar to array
print(arr * 2)    # Multiply array by scalar
```

Output:

```
[11 12 13 14 15]
[ 2  4  6  8 10]
```

---

2. Broadcasting with Vectors

```
a = np.array([1, 2, 3])       # Shape (3,)
b = np.array([[10], [20], [30]])  # Shape (3,1)

print(a + b)
```

Output:

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

👉 NumPy stretched both arrays to shape `(3,3)`.

---

3. Broadcasting with Matrices

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

vector = np.array([10, 20, 30]) # Shape (3,)

print(matrix + vector)
```

Output:

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

👉 `vector` is broadcast across each row of the `matrix`.

---

4. Complex Example

```
A = np.ones((3, 4))   # Shape (3, 4)
B = np.arange(4)      # Shape (4,)

print(A + B)
```

Output:

```
[[1. 2. 3. 4.]
 [1. 2. 3. 4.]
 [1. 2. 3. 4.]]
```

Tips & Tricks

- Broadcasting avoids explicit loops → **much faster** than Python loops.
- Use `np.newaxis` (or `None`) to reshape arrays for broadcasting:

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

print(a[:, np.newaxis] + b)  # Shape (3,1) + (3,) -> (3,3)
```

Exercises

1. Create a 1D array `[1,2,3,4]` and broadcast it to add `100` to each element.
2. Add a column vector `[1,2,3]` to a row vector `[10,20,30]` using broadcasting.
3. Given `A = np.array([[1],[2],[3]])` and `B = np.array([10,20,30])`, apply broadcasting to compute `A * B`.
4. Use broadcasting to normalize a matrix `M` (subtract mean of each column).

✅ Solutions

```
# 1
arr = np.array([1,2,3,4])
print(arr + 100)
# [101 102 103 104]

# 2
col = np.array([[1],[2],[3]])
row = np.array([10,20,30])
print(col + row)
# [[11 21 31]
#  [12 22 32]
#  [13 23 33]]

# 3
A = np.array([[1],[2],[3]])
B = np.array([10,20,30])
print(A * B)
# [[10 20 30]
#  [20 40 60]
#  [30 60 90]]

# 4 Normalization by column
M = np.array([[1,2,3],
              [4,5,6],
              [7,8,9]])
col_mean = M.mean(axis=0)   # shape (3,)
M_normalized = M - col_mean
print(M_normalized)
# [[-3. -3. -3.]
#  [ 0.  0.  0.]
#  [ 3.  3.  3.]]
```

```
Start coding or generate with AI.
```

7. Linear Algebra with NumPy

NumPy provides a powerful `numpy.linalg` module for linear algebra operations. These are widely used in data science, machine learning, physics, engineering, and mathematics.

7.1 Dot Product

The **dot product** of two vectors is the sum of the products of their corresponding elements.

🔷 Formula:

$$a \cdot b = \sum_{i=1}^{n} a_i b_i$$

Example:

```
import numpy as np

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

dot1 = np.dot(a, b)   # Method 1
dot2 = a @ b          # Method 2 (Python 3.5+)

print(dot1)  # 32
print(dot2)  # 32
```

✅ Tip: Use `@` operator for cleaner code.

7.2 Matrix Multiplication

Matrix multiplication is different from element-wise multiplication.

🔷 Formula: If A is of shape (m, n) and B is (n, p), then result is (m, p).

Example:

```
A = np.array([[1, 2], [3, 4]])
B = np.array([[5, 6], [7, 8]])

C1 = np.matmul(A, B)   # Method 1
C2 = A @ B             # Method 2

print(C1)
# [[19 22]
#  [43 50]]
```

7.3 Transpose of a Matrix

Transpose flips a matrix over its diagonal (rows become columns).

```
A = np.array([[1, 2, 3], [4, 5, 6]])
print("Original:\n", A)
print("Transpose:\n", A.T)
```

7.4 Determinant

The determinant is a scalar value that provides information about a matrix (invertibility, volume scaling).

```
A = np.array([[1, 2], [3, 4]])
det = np.linalg.det(A)
```

```
print("Determinant:", det)  # -2.0
```

✅ Tip: If determinant = 0 → Matrix is singular (non-invertible).

7.5 Inverse of a Matrix

The inverse of a matrix A^{-1} satisfies:

$$A \cdot A^{-1} = I$$

```
A = np.array([[1, 2], [3, 4]])
inv = np.linalg.inv(A)
print("Inverse:\n", inv)

# Verification
print("A @ inv:\n", A @ inv)  # Should be Identity Matrix
```

7.6 Eigenvalues & Eigenvectors

For a square matrix A:

$$Av = \lambda v$$

Where λ = eigenvalue, v = eigenvector.

```
A = np.array([[4, -2],
              [1,  1]])

eigenvalues, eigenvectors = np.linalg.eig(A)

print("Eigenvalues:", eigenvalues)
print("Eigenvectors:\n", eigenvectors)
```

📝 Exercises

Q1. Compute the dot product of `[2, 3, 4]` and `[1, 0, -1]`.

Q2. Multiply

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}, B = \begin{bmatrix} 2 & 0 \\ 1 & 2 \end{bmatrix}$$

Q3. Find the determinant of

$$\begin{bmatrix} 6 & 1 \\ 4 & -2 \end{bmatrix}$$

Q4. Find the inverse of

$$\begin{bmatrix} 2 & 1 \\ 7 & 4 \end{bmatrix}$$

Q5. Compute the eigenvalues and eigenvectors of

$$\begin{bmatrix} 3 & 1 \\ 0 & 2 \end{bmatrix}$$

✅ Solutions

```
# Q1
a = np.array([2, 3, 4])
b = np.array([1, 0, -1])
print(np.dot(a, b))  # -2
```

```
# Q2
A = np.array([[1, 2], [3, 4]])
B = np.array([[2, 0], [1, 2]])
print(A @ B)
# [[ 4  4]
#  [10  8]]

# Q3
A = np.array([[6, 1], [4, -2]])
print(np.linalg.det(A))  # -16.0

# Q4
A = np.array([[2, 1], [7, 4]])
print(np.linalg.inv(A))
# [[ 4. -1.]
#  [-7.  2.]]

# Q5
A = np.array([[3, 1], [0, 2]])
vals, vecs = np.linalg.eig(A)
print("Eigenvalues:", vals)      # [3. 2.]
print("Eigenvectors:\n", vecs)   # [[1. 0.]
                                 #  [0. 1.]]
```

⚡ **Tips & Tricks**:

- Always check if a matrix is **square & non-singular** before computing its inverse.
- Use `np.allclose(A @ inv, np.eye(n))` to verify inversion accuracy.
- For performance, prefer `np.dot` / `@` over looping through elements.

Start coding or generate with AI.

8. Random Module in NumPy

NumPy provides powerful tools for generating random numbers, which are essential in simulations, testing, and machine learning. These random functions are available in **`numpy.random`**.

8.1 Generating Random Numbers

1. np.random.rand()

- Generates random numbers from a **uniform distribution** between **0 and 1**.
- You can specify shape as arguments.

```
import numpy as np

# Single random number
print(np.random.rand())

# 1D array of 5 random numbers
print(np.random.rand(5))

# 2D array (3x3) of random numbers
print(np.random.rand(3, 3))
```

---

2. np.random.randn()

- Generates random numbers from a **standard normal distribution** (mean=0, variance=1).

```
# Single random number
print(np.random.randn())
```

```
# 1D array of 5 random numbers
print(np.random.randn(5))

# 2D array (2x3) from standard normal distribution
print(np.random.randn(2, 3))
```

---

3. np.random.randint()

- Generates random **integers** in a given range `[low, high)`.

```
# Single integer between 1 and 10
print(np.random.randint(1, 10))

# Array of 5 integers between 0 and 50
print(np.random.randint(0, 50, 5))

# 2D array of random integers between 10 and 100
print(np.random.randint(10, 100, size=(3, 4)))
```

8.2 Random Choice

Select random elements from a sequence.

```
arr = np.array([10, 20, 30, 40, 50])

# Pick one random element
print(np.random.choice(arr))

# Pick 3 random elements (with replacement)
print(np.random.choice(arr, 3))

# Pick 3 random elements (without replacement)
print(np.random.choice(arr, 3, replace=False))

# Probability distribution
print(np.random.choice(arr, 5, p=[0.1, 0.2, 0.3, 0.1, 0.3]))
```

8.3 Seeding Random Numbers

Random numbers are generated based on a seed value. If you set a seed, the random numbers will always be the **same**, which is useful for reproducibility in experiments.

```
np.random.seed(42)

print(np.random.rand(3))
print(np.random.randint(1, 10, 3))

# Resetting seed gives the same result
np.random.seed(42)
print(np.random.rand(3))
```

✅ Exercises

1. Generate a **3x3 matrix** of random integers between **1 and 100**.
2. Create an array of **10 random numbers** from a **normal distribution**.
3. Use `np.random.choice` to pick **5 random students** from the list:

   ```
   students = ["Ali", "Sara", "John", "Mary", "Ahmed", "Zara", "Omar", "Fatima"]
   ```

4. Set a seed and generate **two identical arrays** of random numbers.

🎯 Solutions

```
# 1. 3x3 random integers
mat = np.random.randint(1, 100, size=(3, 3))
print("3x3 Random Integers:\n", mat)

# 2. 10 numbers from normal distribution
normal_nums = np.random.randn(10)
print("Normal Distribution:\n", normal_nums)

# 3. Random choice from list
students = ["Ali", "Sara", "John", "Mary", "Ahmed", "Zara", "Omar", "Fatima"]
picked = np.random.choice(students, 5, replace=False)
print("Picked students:", picked)

# 4. Seeding
np.random.seed(7)
arr1 = np.random.rand(5)

np.random.seed(7)
arr2 = np.random.rand(5)

print("Array 1:", arr1)
print("Array 2:", arr2)
```

🔥 Tips & Tricks

- Use `np.random.seed()` to ensure **reproducibility**.
- Use `replace=False` in `np.random.choice()` for **sampling without repetition**.
- For **machine learning experiments**, always fix the seed before data splitting or weight initialization.

```
Start coding or generate with AI.
```

9. Working with Files in NumPy

Often, after processing or generating data, we need to save it for later use. NumPy provides efficient ways to store and retrieve arrays.

9.1 Saving Arrays

1. np.save()

- Saves an array in **binary format** (`.npy`).
- Best for saving and loading arrays quickly without losing precision.

```
import numpy as np

arr = np.array([10, 20, 30, 40, 50])

# Save array
np.save("my_array.npy", arr)

print("Array saved successfully!")
```

---

2. np.savetxt()

- Saves an array in **text format** (`.txt` or `.csv`).
- Can specify delimiters (`,` or `\t`).
- Useful when you want to **share data in a readable format**.

```
arr2d = np.array([[1, 2, 3], [4, 5, 6]])

# Save as text file
np.savetxt("my_array.txt", arr2d)

# Save as CSV with delimiter
np.savetxt("my_array.csv", arr2d, delimiter=",")
```

9.2 Loading Arrays

1. np.load()

- Loads arrays stored in `.npy` or `.npz` files.

```
loaded = np.load("my_array.npy")
print("Loaded Array:", loaded)
```

---

2. np.loadtxt()

- Loads arrays stored in **text files** (`.txt`, `.csv`).

```
loaded_txt = np.loadtxt("my_array.txt")
print("Loaded from TXT:", loaded_txt)

loaded_csv = np.loadtxt("my_array.csv", delimiter=",")
print("Loaded from CSV:\n", loaded_csv)
```

---

3. np.genfromtxt()

- Similar to `np.loadtxt()` but **handles missing values**.
- Useful when files have incomplete data.

```
# Example CSV with missing values
# 1,2,3
# 4,,6
# 7,8,9

loaded_gen = np.genfromtxt("my_incomplete.csv", delimiter=",", filling_values=-1)
print("Loaded with genfromtxt:\n", loaded_gen)
```

✅ Exercises

1. Save a **2D array** into a `.npy` file and load it back.
2. Save an array into a **CSV file** with commas as delimiters, then load it.
3. Load a text file with missing values using **np.genfromtxt** and replace missing values with **0**.

🎯 Solutions

```
# 1. Save and load .npy
arr = np.array([[10, 20, 30], [40, 50, 60]])
np.save("matrix.npy", arr)
loaded = np.load("matrix.npy")
print("Loaded from .npy:\n", loaded)

# 2. Save and load CSV
np.savetxt("matrix.csv", arr, delimiter=",")
loaded_csv = np.loadtxt("matrix.csv", delimiter=",")
print("Loaded from CSV:\n", loaded_csv)

# 3. Handle missing values
```

```
# Assume file 'data_with_missing.csv' contains some empty entries
loaded_gen = np.genfromtxt("data_with_missing.csv", delimiter=",", filling_values=0)
print("Loaded with missing values replaced:\n", loaded_gen)
```

🔥 **Tips & Tricks**

- Use **`.npy` files** for fast saving/loading of arrays in projects.
- Use **`.csv` or `.txt`** when data needs to be shared with others.
- Use `np.genfromtxt()` when dealing with **messy real-world datasets**.

```
Start coding or generate with AI.
```

10. Advanced Topics in NumPy

10.1 Vectorization vs. Loops

- NumPy is optimized for **vectorized operations** (using array math instead of Python loops).
- Vectorization uses **C-optimized code**, making it much faster.

🔷 Example: Compute the square of numbers from 1 to 1,000,000

```
import numpy as np
import time

# Using loop (slow)
arr = np.arange(1, 1000001)
start = time.time()
result_loop = [x**2 for x in arr]
end = time.time()
print("Loop time:", end - start)

# Using vectorization (fast)
start = time.time()
result_vec = arr ** 2
end = time.time()
print("Vectorized time:", end - start)
```

✅ Vectorization is **10–100x faster** than Python loops.

10.2 Memory Views & Copying

- NumPy arrays can **share memory** (views) or create a **new copy**.
- Views are more memory-efficient.

```
arr = np.array([10, 20, 30, 40])

# Create a view (changes affect original)
view = arr.view()
view[0] = 99
print("Original after view change:", arr)

# Create a copy (independent array)
copy = arr.copy()
copy[1] = 77
print("Original after copy change:", arr)
```

✅ Use `.copy()` when you need an independent object.

10.3 Structured Arrays & dtypes

NumPy supports **structured arrays**, where each element can have multiple fields (like database rows).

```
# Define structured dtype
student_dtype = np.dtype([
    ('name', 'U10'),    # string up to 10 chars
    ('age', 'i4'),      # 4-byte integer
    ('marks', 'f4')     # 4-byte float
])

# Create structured array
students = np.array([
    ("Alice", 20, 85.5),
    ("Bob", 22, 90.0),
    ("Charlie", 21, 78.0)
], dtype=student_dtype)

print(students)
print("Names:", students['name'])
print("Marks:", students['marks'])
```

✅ Structured arrays are useful for datasets with **mixed data types**.

10.4 Masking & Conditional Selection

- Boolean masking allows **filtering arrays**.
- Similar to SQL queries but inside NumPy.

```
arr = np.array([5, 12, 18, 7, 30, 22])

# Get elements greater than 15
mask = arr > 15
print("Mask:", mask)
print("Filtered:", arr[mask])

# Direct condition
print("Even numbers:", arr[arr % 2 == 0])
```

✅ Very useful for **data cleaning and filtering**.

10.5 Stacking & Splitting Arrays

Stacking

Combine multiple arrays.

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

# Horizontal stacking
print("hstack:", np.hstack((a, b)))

# Vertical stacking
print("vstack:\n", np.vstack((a, b)))
```

Splitting

Divide arrays into multiple parts.

```
arr = np.array([10, 20, 30, 40, 50, 60])

# Split into 3 equal parts
parts = np.split(arr, 3)
print("Split into 3 parts:", parts)
```

```
# 2D split
matrix = np.array([[1, 2], [3, 4], [5, 6]])
upper, lower = np.vsplit(matrix, 2)
print("Upper:\n", upper)
print("Lower:\n", lower)
```

✅ Quick Recap

1. **Vectorization** → Faster than loops.
2. **Copy vs View** → `.copy()` makes new memory, `.view()` shares memory.
3. **Structured Arrays** → Store mixed data types.
4. **Masking** → Filter arrays with conditions.
5. **Stacking & Splitting** → Combine or divide arrays.

Start coding or generate with AI.

NumPy Complete Tutorial (Beginner to Advanced)

This guide covers **all essential and advanced topics in NumPy** with examples and explanations. Use it as a reference or a learning roadmap.

Table of Contents

1. Introduction to NumPy
2. NumPy Arrays (Creation & Basics)
3. Array Indexing & Slicing
4. Array Operations
5. Universal Functions (ufuncs)
6. Broadcasting
7. Mathematical & Statistical Functions
8. Linear Algebra
9. Random Module in NumPy
10. Advanced Topics

---

1. Introduction to NumPy

- NumPy = Numerical Python → powerful library for numerical computing.
- Provides **N-dimensional arrays** (ndarrays).
- Faster than Python lists because operations are vectorized and written in C.

```
import numpy as np
print(np.__version__)
```

---

2. NumPy Arrays (Creation & Basics)

```
# Create array from list
arr = np.array([1, 2, 3])
print(arr)

# Special arrays
zeros = np.zeros((2, 3))      # 2x3 matrix of zeros
ones = np.ones((3, 3))        # 3x3 matrix of ones
arange = np.arange(0, 10, 2)  # [0,2,4,6,8]
linspace = np.linspace(0, 1, 5) # 5 numbers between 0 and 1
```

- Attributes:

```
print(arr.ndim)   # dimensions
print(arr.shape)  # shape
print(arr.size)   # total elements
print(arr.dtype)  # data type
```

---

3. Array Indexing & Slicing

```
arr = np.array([10, 20, 30, 40, 50])
print(arr[0])   # first element
print(arr[-1])  # last element

# Slicing
print(arr[1:4])  # [20,30,40]

# 2D indexing
mat = np.array([[1,2,3],[4,5,6],[7,8,9]])
print(mat[0,1])   # element at row 0 col 1 → 2
print(mat[:,0])   # first column
```

---

4. Array Operations

```
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])

print(arr1 + arr2)   # [5,7,9]
print(arr1 * arr2)   # [4,10,18]
print(arr1 ** 2)     # [1,4,9]
```

---

5. Universal Functions (ufuncs)

- Fast element-wise functions.

```
arr = np.array([1, 4, 9, 16])
print(np.sqrt(arr))
print(np.exp(arr))
print(np.log(arr))
```

---

6. Broadcasting

- NumPy automatically expands smaller arrays.

```
mat = np.array([[1,2,3],[4,5,6]])
vec = np.array([10,20,30])
print(mat + vec)
```

---

7. Mathematical & Statistical Functions

```
arr = np.array([1,2,3,4,5])
print(np.sum(arr))
print(np.mean(arr))
print(np.std(arr))
print(np.min(arr))
print(np.max(arr))
```

---

8. Linear Algebra

```
A = np.array([[1,2],[3,4]])
B = np.array([[5,6],[7,8]])

# Matrix multiplication
print(np.dot(A, B))

# Transpose
print(A.T)

# Inverse
print(np.linalg.inv(A))

# Eigenvalues & vectors
vals, vecs = np.linalg.eig(A)
print(vals)
print(vecs)
```

---

9. Random Module in NumPy

```
np.random.seed(42)  # reproducibility

print(np.random.rand(3))     # uniform [0,1)
print(np.random.randn(3))    # normal distribution
print(np.random.randint(1,10,5))  # random integers
```

---

10. Advanced Topics

10.1 Vectorization vs Loops

```
import time
arr = np.arange(1, 1000001)

start = time.time()
res = [x**2 for x in arr]
print("Loop time:", time.time()-start)

start = time.time()
res = arr**2
print("Vectorized time:", time.time()-start)
```

10.2 Memory Views vs Copies

```
arr = np.array([10,20,30])
view = arr.view()
view[0] = 99
print(arr)

copy = arr.copy()
copy[1] = 77
print(arr)
```

10.3 Structured Arrays

```
dt = np.dtype([('name','U10'),('age','i4'),('marks','f4')])
students = np.array([
    ("Alice",20,85.5),
    ("Bob",22,90.0)
```

```
], dtype=dt)

print(students['name'])
print(students['marks'])
```

10.4 Masking & Conditional Selection

```
arr = np.array([5,12,18,7,30])
print(arr[arr > 15])
print(arr[arr % 2 == 0])
```

10.5 Stacking & Splitting

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

print(np.hstack((a,b)))
print(np.vstack((a,b)))

arr = np.array([10,20,30,40,50,60])
print(np.split(arr,3))
```

✅ Recap

- NumPy is essential for scientific computing.
- Learn **creation, indexing, operations, broadcasting, linear algebra, random module, and advanced memory tricks**.
- Prefer **vectorization over loops** for speed.
- Use **masking & stacking** for clean data manipulation.

📌 You now have a **complete NumPy roadmap** from basics → advanced. Would you like me to also add a **set of practice exercises with solutions** at the end of this guide (like a mini workbook)?

```
Start coding or generate with AI.
```