Mastering NumPy: From Basics to Expert Level

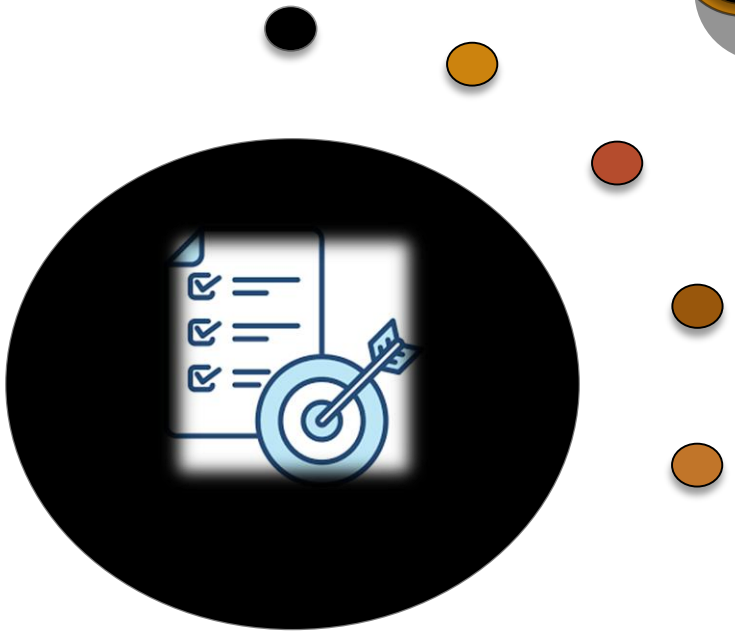
Numerical Computing with Python

NumPy

Dated: 14 June 2025

Instructor : Fawad Bahadur

Training Objectives & Outcomes



1. Understand the fundamentals of NumPy arrays
2. Perform efficient mathematical and statistical operations
3. Master advanced techniques like broadcasting, indexing, and memory handling
4. Apply NumPy to real-world ML, data science, and scientific problems

Installation: NumPy

☑ Using pip (Recommended)

```
pip install numpy
```

☑ Using conda (For Anaconda users)

```
conda install numpy
```

🔍 Verify Installation

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

Introduction to NumPy

Definition

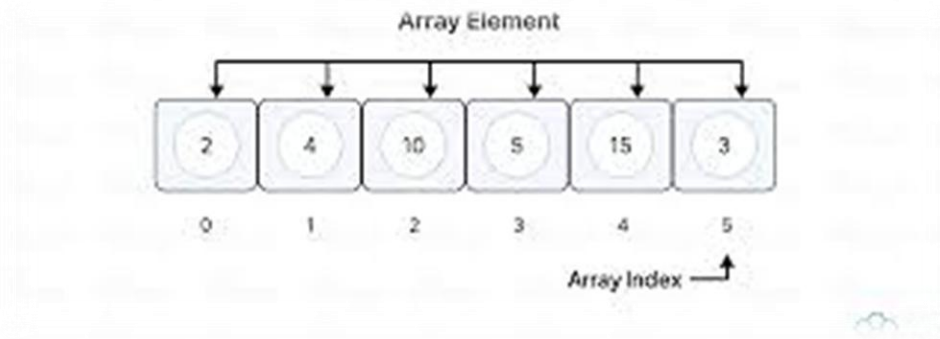
- NumPy (**Numerical Python**) is the foundational library for numerical computing in Python.
- Provides the powerful **ndarray** object for multi-dimensional arrays.

Example

```
import numpy as np
arr = np.array([1, 2, 3])
print(arr * 2)  # Output: [2 4 6]
```

Why Use NumPy?

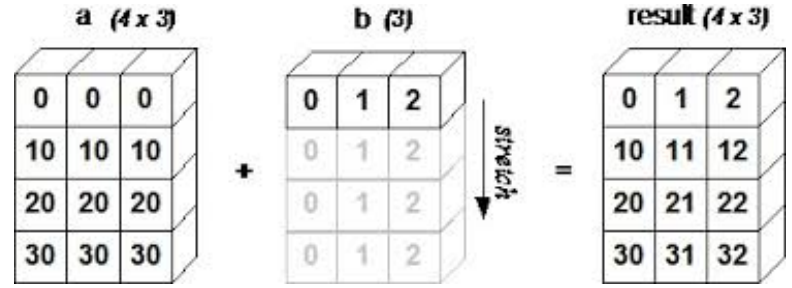
Fast: Written in C, supports vectorized operations
Efficient: Less memory than Python lists
Versatile: Used in ML, data science, image processing, simulations

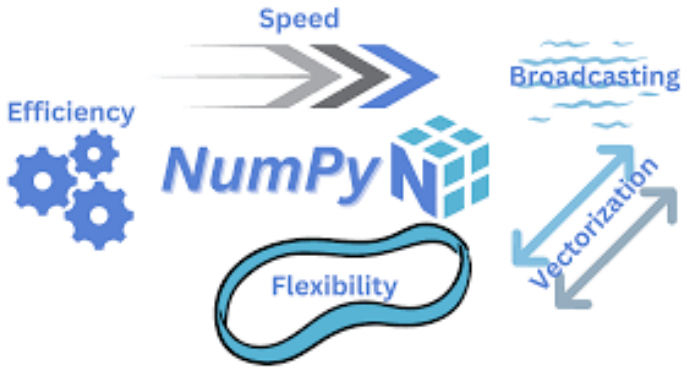


Introduction to NumPy

Key Features

- N-dimensional arrays
- Broadcasting
- Faster
- Efficient
- Mathematical and logical operations
- Linear algebra & random number generation
- Integration with other libraries (Pandas, TensorFlow, Scikit-learn)





Creating Arrays in NumPy

1. Using np.array()

```
import numpy as np
a = np.array([1, 2, 3])          # 1D
b = np.array([[1, 2], [3, 4]])   # 2D
```

2. Predefined Functions

```
np.zeros((2, 3))       # Array of zeros
np.ones((2, 3))        # Array of ones
np.full((2, 2), 7)     # Array filled with
np.eye(3)              # Identity matrix
```

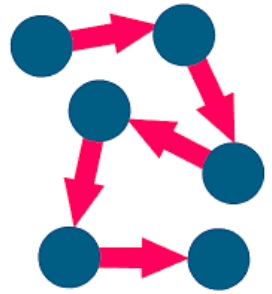
3. Generating Sequences

```
np.arange(0, 10, 2)     # [0, 2, 4, 6, 8]
```

(start, stop, step)

```
np.linspace(0, 1, 5)    # [0. , 0.25, ..., 1.]
```

(start, stop, num)



4. Random Arrays

```
np.random.rand(2, 3)
```

Uniform [0,1)

```
np.random.randint(0, 10, 5)
```

(low, high, size)

Random integers

Note

Use .dtype to control data type if needed.

Understanding NumPy Data Types

What is dtype?

- Specifies the type of elements in an array
- Controls memory size and operations behavior

Specifying dtype

```
a = np.array([1, 2, 3], dtype=np.float32)
print(a.dtype)      # float32
```

Converting dtypes

```
b = a.astype(np.int64)
print(b.dtype)      # int64
```

dtype	Description
Int32	32-bit integer
Int64	64-bit integer
Float32	32-bit floating point
Float64	64-bit floating point
Bool	Boolean True/False
Complex	Complex numbers

Note

Choosing correct dtype improves performance and saves memory.

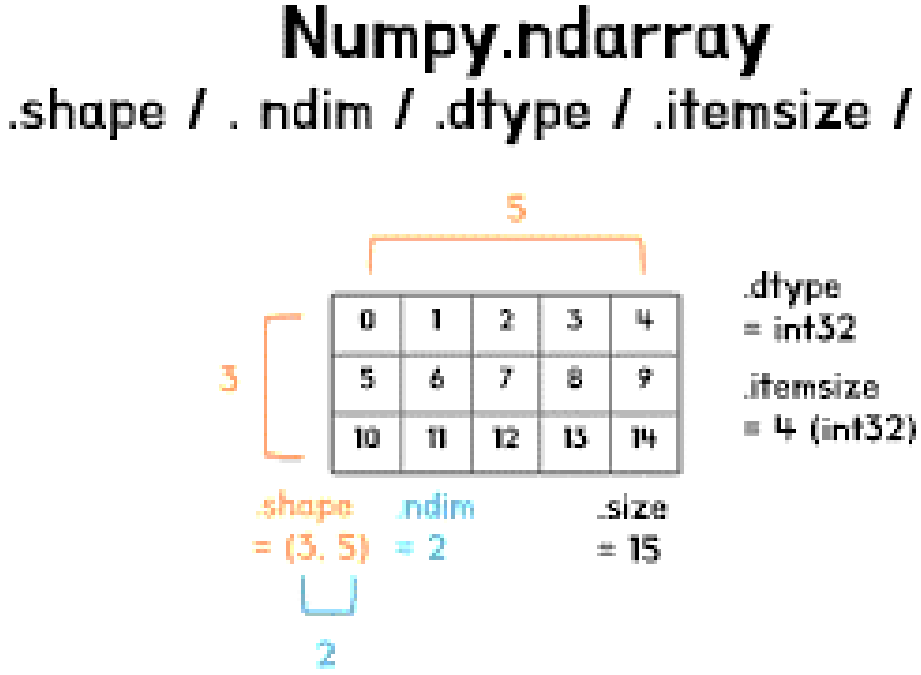
Understanding NumPy Array Attributes

1. Key Attributes of ndarray

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

2. Common Attributes

- arr.shape → (2, 3): Dimensions (rows, columns)
- arr.ndim → 2: Number of dimensions
- arr.size → 6: Total number of elements
- arr.dtype → int64: Data type of elements
- arr.itemsize → 8: Bytes per element
- arr.nbytes → 48: Total memory (size × itemsize)



3. View in Practice

```
print(arr.shape, arr.ndim, arr.dtype)
```

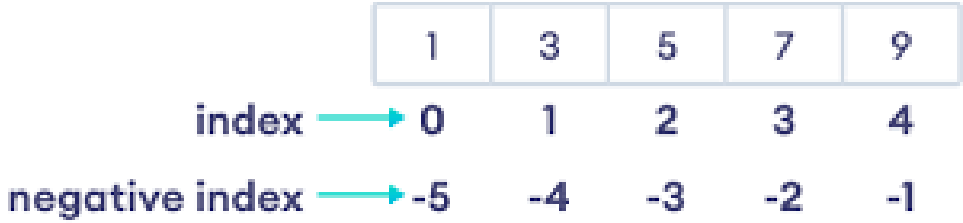
Tip

Understanding these helps in debugging, optimization & reshaping operations.

Indexing & Slicing in NumPy Arrays

1. Basic Indexing

```
arr = np.array([[10, 20, 30], [40, 50, 60]])
arr[0, 1]     # Output: 20
arr[1][2]     # Output: 60
```



2. Slicing Syntax

```
arr[0, :2]    # First row, first two columns → [10 2
arr[:, 1:]    # All rows, columns from index 1 → [[20 30], [50 60]]
```

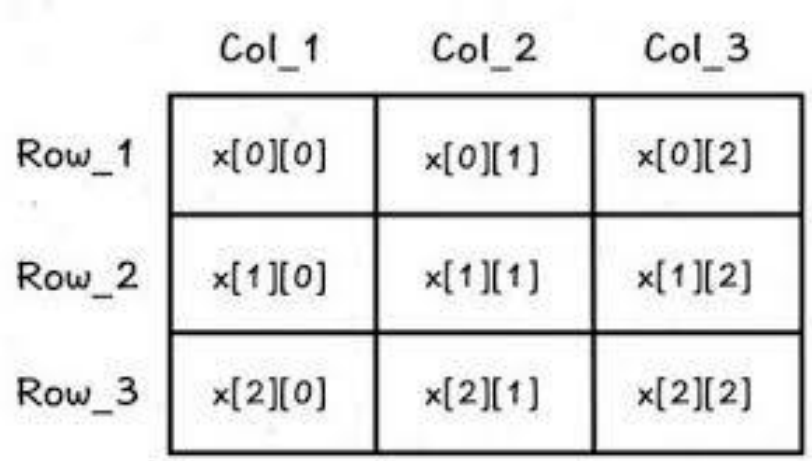
3. Negative Indexing

```
arr[-1, -2]   # Second last element in last row → 50
```

4. 1D vs 2D Behavior

```
a = np.arange(10)
a[2:7:2]      # Output: [2 4 6]
```

array[start:stop:step]



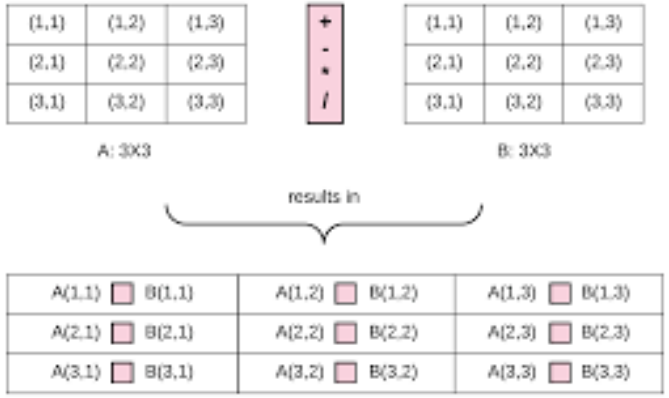
Tip

Use slicing to avoid loops and write efficient code.

Arithmetic & Logical Operations in NumPy

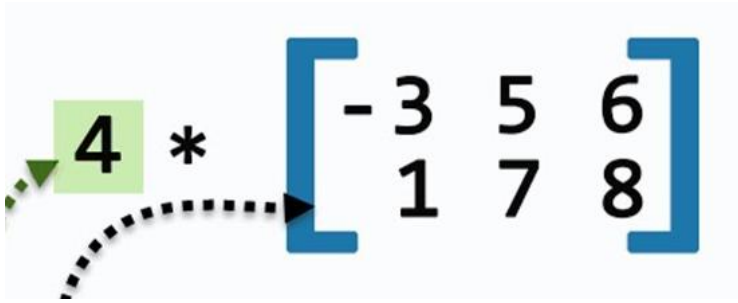
1. Element-wise Arithmetic

```
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])
a + b    # [5 7 9]
a * b    # [4 10 18]
a / b    # [0.25 0.4 0.5]
```



2. Scalar Operations

```
a * 10   # [10 20 30]
a 1      # [0 1 2]
```



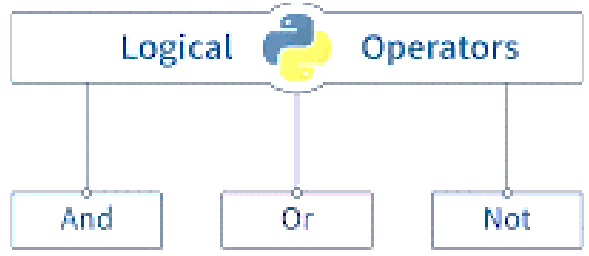
3. Comparison Operators

```
a > 2    # [False False  True]
b == 5   # [False  True False]
```

Operator	Meaning
==	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

4. Logical Operations

```
np.logical_and(a > 1, b < 6)  # [False  True False]
```



Note

All operations are vectorized — no explicit loops needed.

Universal Functions in NumPy

1. What Are ufuncs?

- Functions that operate element-wise on arrays.
- Highly optimized, fast, and support broadcasting.

2. Math Functions

```
a = np.array([1, 2, 3])
np.sqrt(a)      # [1.        1.414213  1.732050]
np.exp(a)       # [ 2.718  7.389052  20.085536]
np.log(a)       # [0.        0.693141  1.09862]
```

3. Trigonometric Functions

```
np.sin(np.pi / 2)   # 1.0
np.cos(0)           # 1.0
```

Tip

ufuncs are faster and more memory-efficient than loops.

4. Aggregation Functions

```
np.sum(a)       # 6
np.mean(a)      # 2.0
np.max(a)       # 3
```

5. Apply ufuncs to Multi-Dim Arrays

```
b = np.array([[1, 2], [3, 4]])
np.sum(b, axis=0)   # Column-wise sum: [4 6]
```

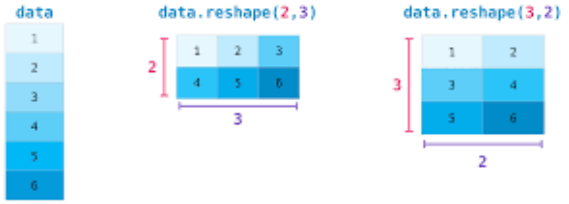
Function	Description
np.add	Element-wise addition
np.subtract	Element-wise subtraction
np.multiply	Element-wise multiplication
np.divide	Element-wise division
np.exp	exponential functions

NumPy

Reshaping & Manipulating Array Shapes

Reshape Arrays

```
arr = np.arange(6)
reshaped_arr = arr.reshape(3, 2)
```



Flatten Arrays

```
b.flatten()          # [0 1 2 3 4 5]
```

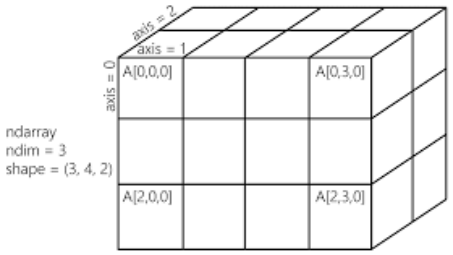
[[1,2,3],
[4,5,6]]

Flatten

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

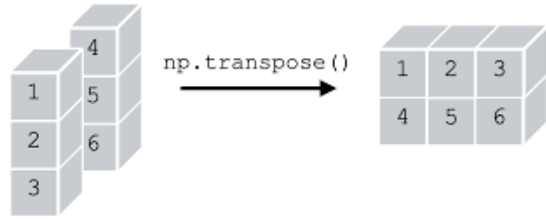
Add/Remove Dimensions

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



Transpose Arrays

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



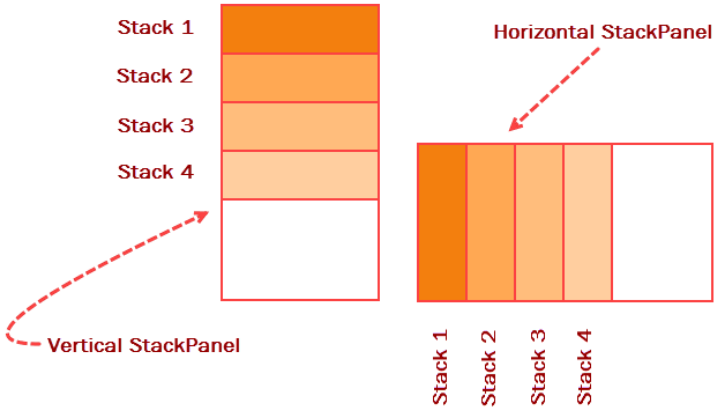
Note

Always ensure compatibility before reshaping to avoid errors.

Stacking & Splitting Arrays in NumPy

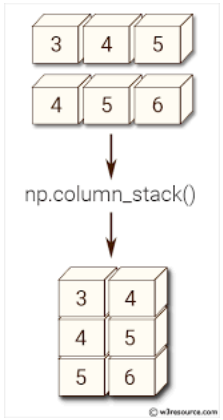
1. Vertical & Horizontal Stacking

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



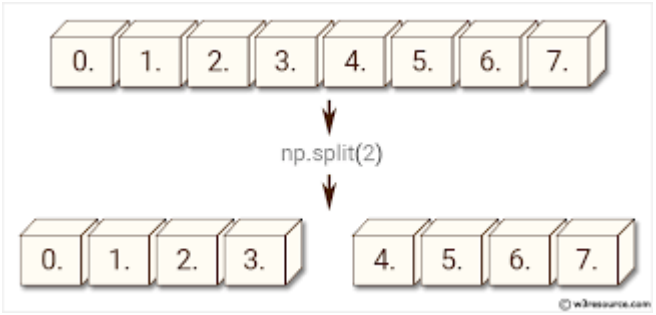
2. Column & Row Stack (2D only)

```
a = np.array([[1], [2]])
b = np.array([[3], [4]])
np.column_stack((a, b))  # [[1 3]     #  [2 4]]
np.row_stack((a.T, b.T)) # [[1 2]     #  [3 4]]
```



3. Splitting Arrays

```
x = np.array([[1, 2, 3], [4, 5, 6]])
np.hsplit(x, 3)    # Split into 3 columns
np.vsplit(x, 2)    # Split into 2 rows
```



Note

Ensure dimensions match for stacking/splitting.

Copy vs View in NumPy Arrays

What is a View?

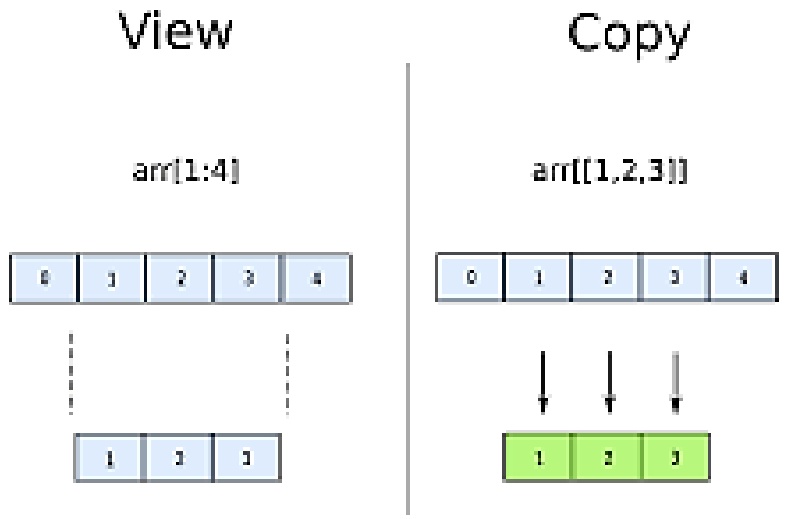
- A view is a new array object that looks at the same data.
- Changes in view affect the original array.

```
a = np.array([1, 2, 3])
v = a.view()
v[0] = 10
print(a)      # [10  2  3]
```

What is a Copy?

- A copy creates a new independent array.
- Changes in copy don't affect the original.

```
c = a.copy()
c[0] = 20
print(a)      # [10  2  3]
print(c)      # [20  2  3]
```



When to Use?

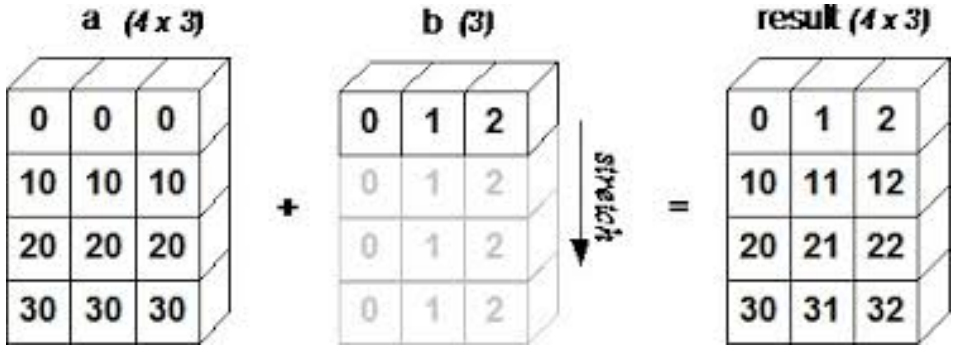
- Use view to save memory (if you want linked changes)
- Use copy to avoid unintentional changes

Broadcasting in NumPy

What is Broadcasting?

Broadcasting enables operations between:

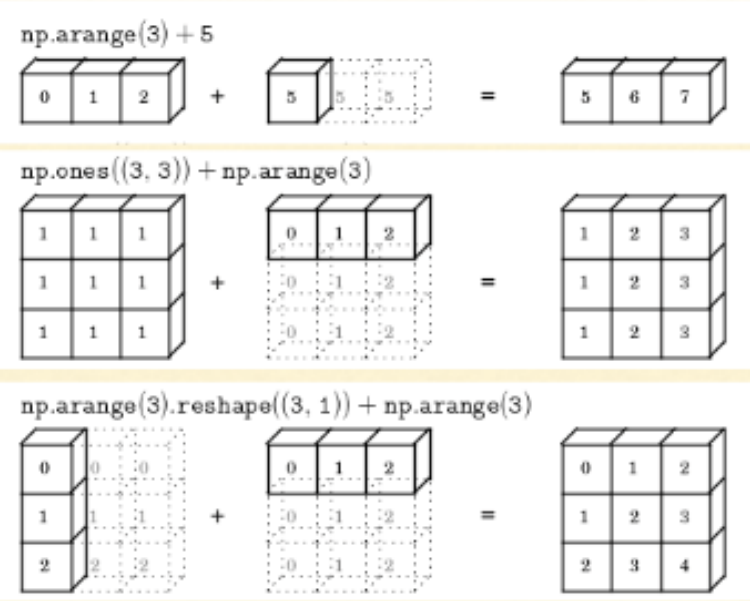
- Arrays of different sizes
- Arrays with different dimensions
- A scalar and an array



Rules of Broadcasting

1. If arrays differ in dimensions, prepend 1s to smaller array's shape.
2. Arrays are compatible if dimensions are equal or one of them is 1.
3. Broadcasting happens to match shapes.

```
a = np.array([[1, 2, 3], [4, 5, 6]])        # Shape (2,3)
b = np.array([10, 20, 30])                  # Shape (3,)
print(a + b)                                # Output:[[11 22 33][14 25 36]]
```



Broadcasting with Scalars

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

Tip

Broadcasting helps avoid explicit loops, improving speed and code clarity.

Conti...

Example 1

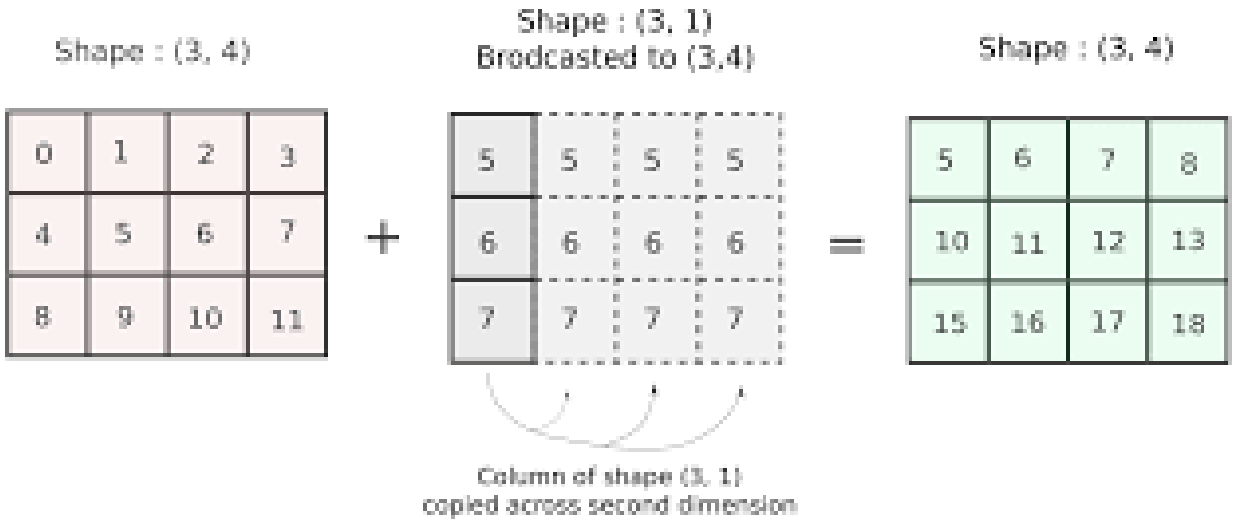
```
a = np.ones((3, 4))
b = np.arange(4)
print((a + b).shape)  # (3, 4)
```

Example 2

```
a = np.ones((5, 1, 4))
b = np.ones((1, 3, 1))
print((a + b).shape)  # (5, 3, 4)
```

Incompatible Shapes

```
a = np.ones((2, 3))
b = np.ones((3, 2))
a + b  # Raises ValueError
```



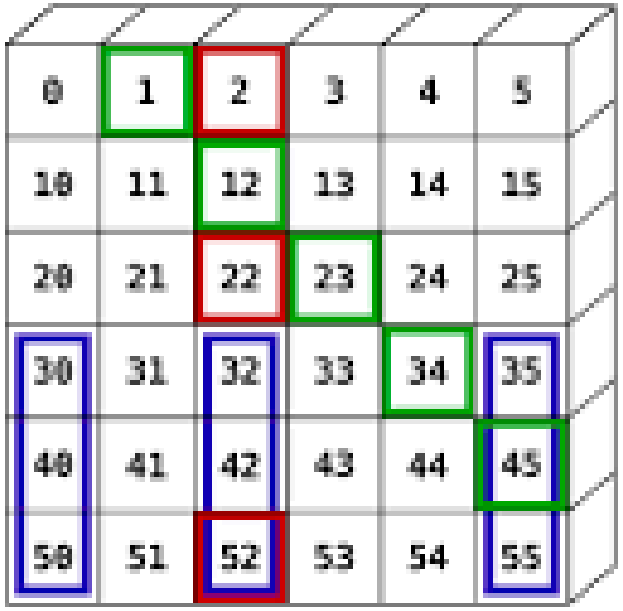
Tip

Use broadcasting to write efficient vectorized code without loops.

Fancy Indexing in NumPy

What is Fancy Indexing?

- Indexing arrays using integer arrays or lists.
- Allows selecting multiple arbitrary elements.



Example with 1D Array

```
a = np.array([10, 20, 30, 40, 50])
indices = [1, 3, 4]
print(a[indices])  # Output: [20 40 50]
```

Example with 2D Array

```
b = np.array([[1, 2], [3, 4], [5, 6]])
rows = [0, 1]cols = [1, 0]
print(b[rows, cols])  # Output: [2 3]
```

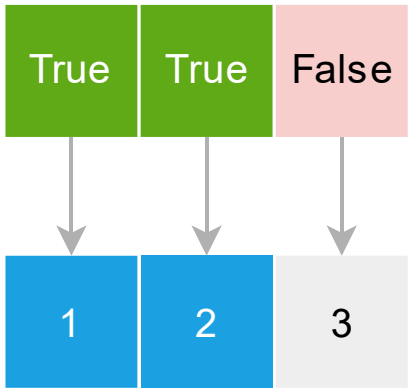
Tip

Fancy indexing returns a copy, not a view.

Boolean Indexing in NumPy

What is Boolean Indexing?

- What is Boolean Indexing?
- Select elements based on conditions.
- Returns elements where the condition is True.




```
a = np.array([10, 20, 30, 40, 50])
mask = a > 25
print(a[mask])   # Output: [30 40 50]
```

Combining Conditions

```
print(a[(a > 15) & (a < 45)])   # Output: [20 30 40]
print(a[(a < 15) | (a > 45)])   # Output: [10 50]
```

Modify Elements Using Boolean Indexing

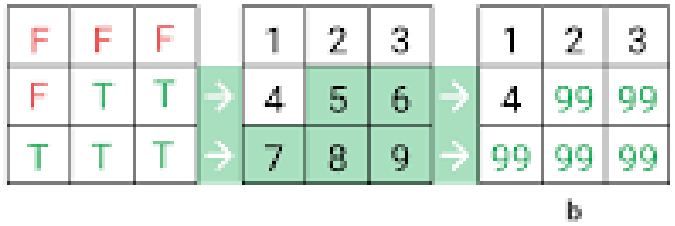
```
a[a < 30] = 0print(a)   # Output: [ 0  0 30 40 50]
```

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

1	2	3
4	5	6
7	8	9

b

```
b[b > 4] = 99
```



Tip

Boolean indexing is a powerful tool for filtering and updating arrays efficiently.

Random Number Generation with NumPy

Why Use NumPy Random?

- Generate random numbers for simulations, ML, testing.
- Supports various distributions.

Common Functions

Function	Description	Example
np.random.rand(d0, d1)	Uniform distribution [0,1)	np.random.rand(2,3)
np.random.randn(d0, d1)	Standard normal distribution (mean=0, std=1)	np.random.randn(3)
np.random.randint(low, high, size)	Random integers between low (inclusive) and high (exclusive)	np.random.randint(0, 10, 5)
np.random.choice(array, size)	Random sample from array	np.random.choice([1,2,3], 2)

Tip

Use np.random.seed() to get consistent results.

Aggregation Functions in NumPy

What are Aggregation Functions?

Perform computations over array elements to produce single or reduced values.

Example

```
a = np.array([[1, 2], [3, 4]])
print(np.sum(a))          # 10
print(np.mean(a))         # 2.5
print(np.min(a))          # 1
print(np.max(a, axis=0))  # [3 4] (max of columns)
```

Common Aggregations

Function	Description
np.sum()	Sum of all elements
np.mean()	Arithmetic mean
np.median()	Median value
np.min()	Minimum value
np.max()	Maximum value
np.std()	Standard deviation
np.var()	Variance

Tip

Use axis parameter to aggregate along rows or columns.

Linear Algebra Operations in NumPy

What are Aggregation Functions?

Perform computations over array elements to produce single or reduced values.

Example

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

print(np.dot(a, b))          #[[19 22][43 50]]
print(np.linalg.inv(a))
```

Common Aggregations

Function	Description
np.dot(a, b)	Dot product of two arrays
np.matmul(a, b)	Matrix multiplication
np.transpose(a)	Transpose of a matrix
np.linalg.inv(a)	Inverse of a square matrix
np.linalg.det(a)	Determinant of a square matrix
np.linalg.eig(a)	Eigenvalues and eigenvectors
np.linalg.norm(a)	Vector or matrix norm

Tip

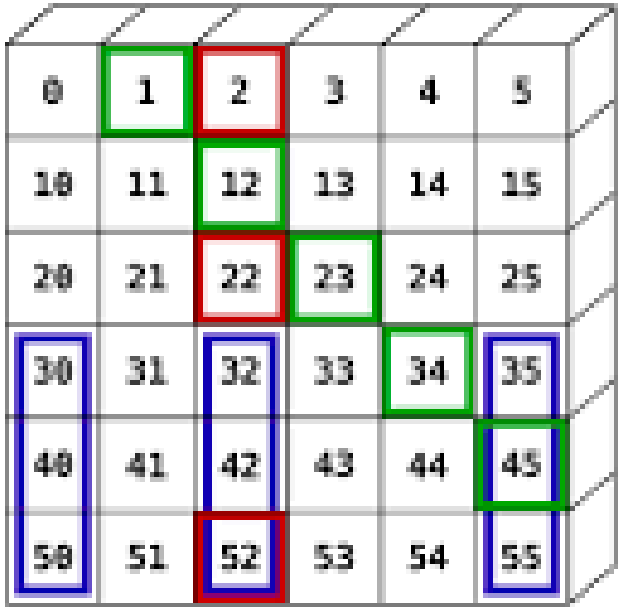
For solving linear systems, use np.linalg.solve(A, b)..

Advanced Indexing in NumPy

Combining Indexing Types

Mix of integer, slice, and boolean indexing

```
a = np.arange(20).reshape(4,5)
print(a[1:3, [0, 2, 4]])  #Select rows 1-2, columns 0,2,4
```



Using np.ix_ for Cross-Indexing

```
rows = [0, 2]
cols = [1, 3]
print(a[np.ix_(rows, cols)])  # Select rows 0 & 2 and cols 1 & 3
```

Ellipsis (...)

Represents “all preceding/succeeding dimensions”

```
print(a[..., 2])   #All rows, column 2
```

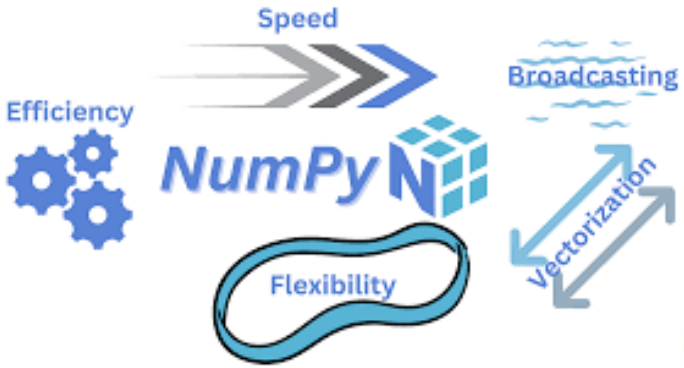
Tip

Advanced indexing returns copies, use carefully for large arrays.

Vectorization in NumPy

What is Vectorization?

Vectorization means using NumPy operations to replace **explicit loops** with array-level operations. It leads to **cleaner code** and **massive speed** improvements.



Example

Loop

```
a = [1, 2, 3, 4]
b = [x**2 for x in a]
```

Vectorized

```
import numpy as np
a = np.array([1, 2, 3, 4])
b = a ** 2  # Vectorized
```

Performance Comparison

```
%timeit [x**2 for x in range(100000)]   # Slower
%timeit np.arange(100000) ** 2          # Faster
```

Pro Tip

Use np.where(), np.sum(), np.dot(), and logical ops to replace loops wherever possible.

NumPy in Real-World Applications

1. Machine Learning & AI

- Data preprocessing (normalization, scaling)
- Vectorized loss functions and activations
- Feeding NumPy arrays into ML models (Scikit-learn, TensorFlow)

2. Image Processing

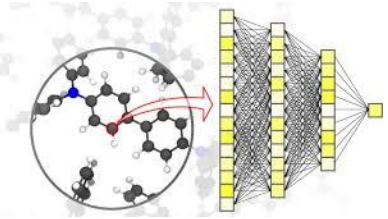
- Images as NumPy arrays (Height × Width × Channels)
- Operations: cropping, resizing, filtering

```
from PIL import Imageimg =
np.array(Image.open("image.jpg"))
```

3. Time Series Analysis

- Slicing and rolling window operations
- Computing moving averages, FFT, etc.

4. Scientific Simulations



Physics/chemistry simulations
Matrix-heavy modeling with real-time data

5. Finance & Data Analytics

Portfolio simulations
Risk modeling using random sampling
Fast computations on large datasets

6. Backend for Pandas & Other Libraries

- Pandas DataFrames use NumPy internally
- TensorFlow/PyTorch accept NumPy arrays as input

NumPy vs Alternative Libraries

Feature	NumPy	Pandas	TensorFlow NumPy / PyTorch
Primary Use	Numerical arrays	Data analysis (tables)	ML & deep learning tensors
Data Structures	Ndarray	DataFrame, Series	Tensors
Ease of Use	Low-level, flexible	High-level, tabular	ML-focused, GPU-enabled
Performance	CPU optimized	Built on NumPy	GPU & TPU acceleration
Broadcasting Support	Yes	Limited	Yes
GPU Support	No	No	Yes
Autograd (Gradient)	No	No	Yes
Ecosystem Integration	Core scientific Python	Data science	Deep learning

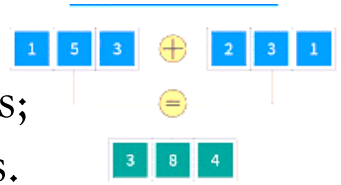
When to Use

- Use NumPy for raw numerical computations and scientific work.
- Use Pandas for data manipulation and analysis.
- Use TensorFlow/PyTorch when building/train deep learning models.

NumPy

Best Practices for Efficient NumPy Coding

1. Use Vectorized Operations

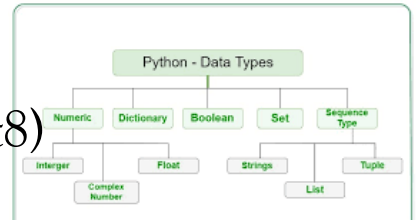


Always Avoid explicit Python loops; leverage NumPy's fast array computations.

2. Prefer Views Over Copies

Use slicing and .view() to save memory and improve speed.

3. Choose Appropriate dtypes

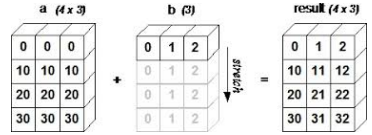


Use smaller types (float32, int8) if precision allows, to save memory.

4. Avoid Growing Arrays in Loops

Preallocate arrays with np.empty() or np.zeros() when possible.

5. Use Broadcasting Wisely



Understand broadcasting rules to avoid unexpected bugs or performance issues.

6. Document Complex Operations

Write comments or use helper functions to keep code readable.

7. Test with Realistic Data Sizes

Check performance and memory on data similar to your production workloads.

8. Keep NumPy Updated

Use the latest stable version to benefit from performance improvements.


THANK
YOU