Lecture 9: Introduction to NumPy (Numerical Computing)

Overview

- Introduction to NumPy and arrays
- Creating arrays and array indexing
- Array operations: element-wise arithmetic, broadcasting
- Basic linear algebra operations using NumPy

NumPy

🧠 What is NumPy?

- NumPy stands for **Numerical Python**. It's a core Python library used for **fast and efficient numerical computing**, especially when working with large amounts of data.

🔧 Why is NumPy Important?

- It's designed for **high performance** and **low memory usage**.
- It's much faster than regular Python lists and loops—often **10 to 100 times faster**.
- It's essential for **data analysis**, **machine learning**, and **scientific computing**.

NumPy

📦 Key Features of NumPy

- **ndarray**: A powerful object for creating arrays with any number of dimensions (1D, 2D, 3D, etc.).
- Stores data in a **continuous block of memory**, making it faster and more memory-efficient than Python's built-in data types.
- Supports **vectorized operations**—you can perform math on entire arrays without writing loops.
- Includes **standard math functions** that work directly on arrays.

🧮 Terminology

- **Array**, **NumPy array**, and **ndarray** all refer to the same thing: a structured, efficient container for elements.
- **Vectorization** means applying operations to whole arrays at once, instead of looping through elements.

Basics

Install NumPy:

```
pip install numpy
```

NumPy documentation:

https://numpy.org/devdocs/user/

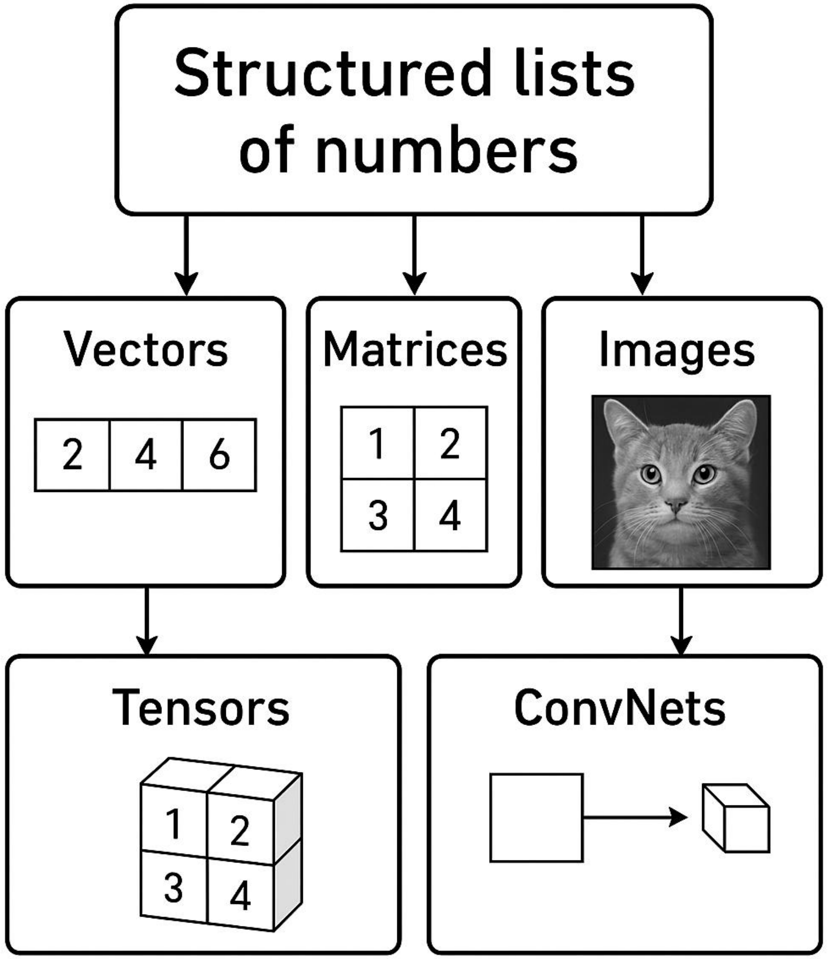
Arrays

- An **array** is a way to organize numbers in a structured format.
- It can take different shapes depending on how the data is arranged:

◆ Vector:

$$\begin{bmatrix} p_x \\ p_y \\ p_z \end{bmatrix}$$

- A **vector** is a **one-dimensional array**.
- It's like a simple list of numbers.
- **Example**: [2, 5, 7]
- **Uses**: Representing features, directions, or simple data points in math and machine learning.

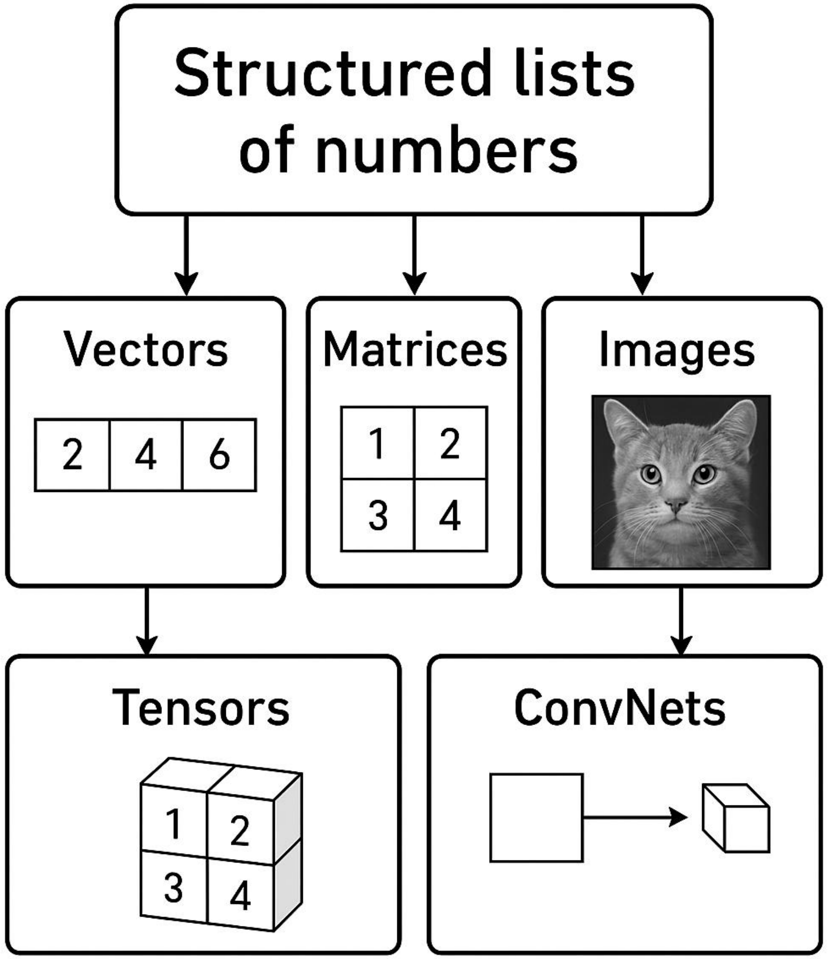


Arrays

- Matrices:

- A matrix is a two-dimensional array .
- It looks like a table with rows and columns .
- Example [[4 ,3] ,[2 ,1]]
- Uses: Linear algebra, transformations, and image representation.

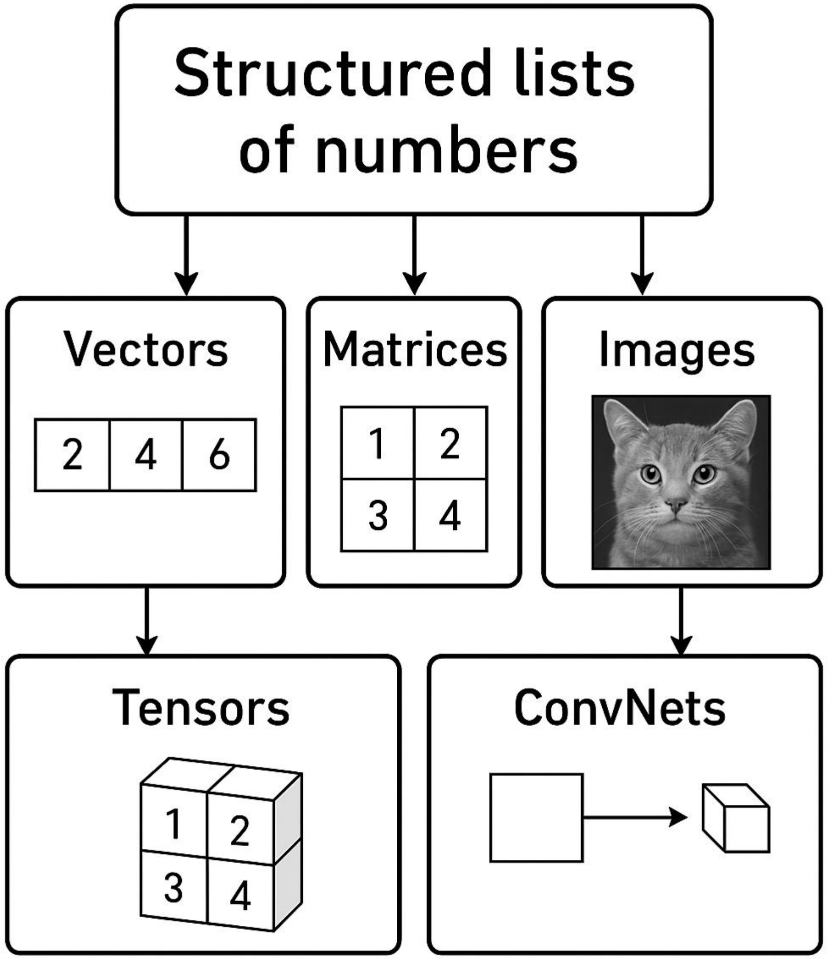
$$\begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{bmatrix}$$



Arrays

- Image

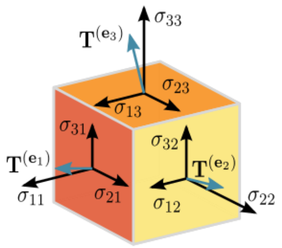
- An **image** is usually stored as a matrix or a 3D array.
- **Grayscale image** → 2D matrix of pixel values.
- **Color image** → 3D array: (height, width, channels)
- **Uses**: Computer vision, graphics, and photography.

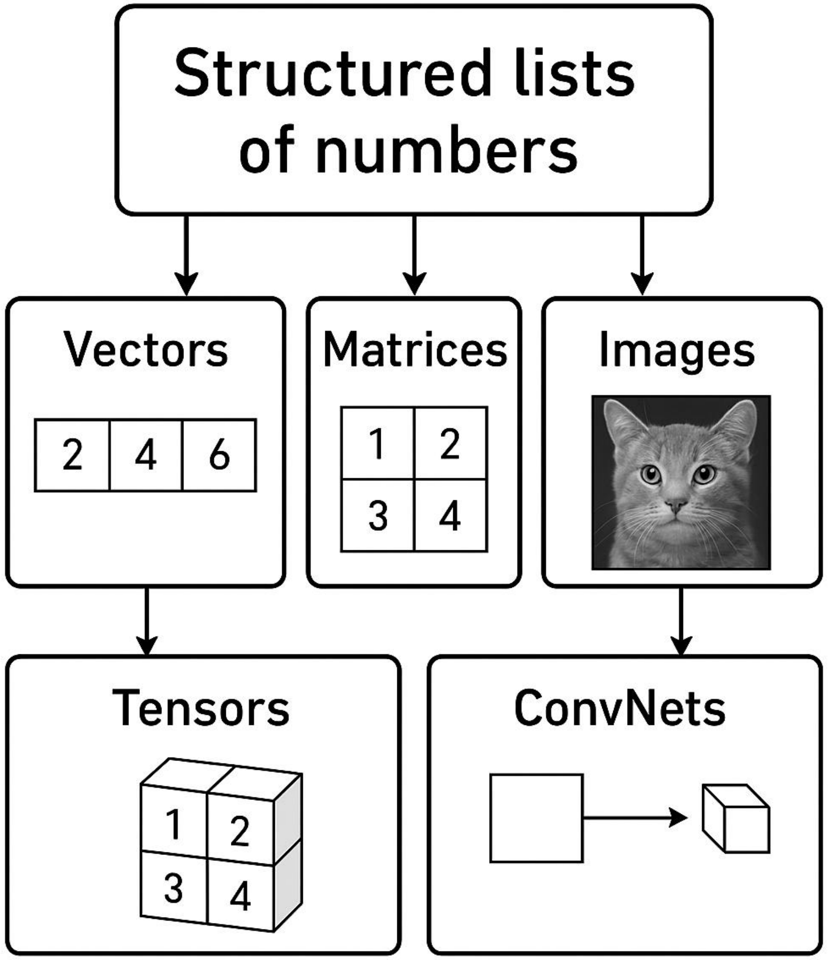


Arrays

◆ Tensor

- A **tensor** is a general term for arrays with **any number of dimensions**.
- Vectors → 1D, Matrices → 2D, Images → 3D, Videos → 4D
- **Uses**: Deep learning, physics, and scientific computing.

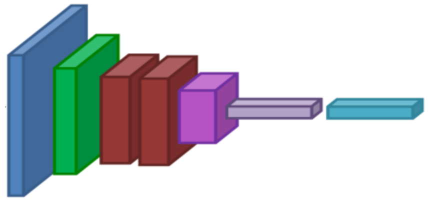


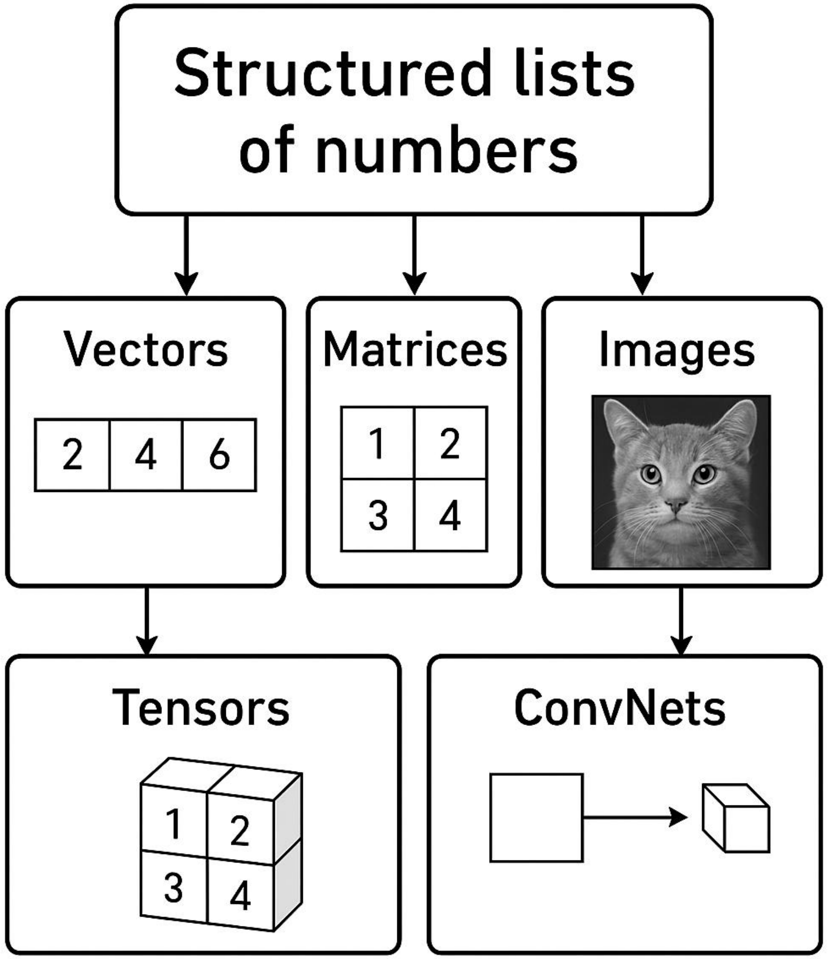


Arrays

ConvNet (Convolutional Neural Network)

- A **ConvNet** is a type of neural network designed to process tensors, especially images.
- It uses filters to detect patterns like edges, shapes, and textures.
- **Uses**: Image classification, object detection, medical imaging, and more.





Arrays, Basic Properties

1. Arrays can have any number of dimensions, including zero (a scalar).
2. Arrays are typed: uint8, int32 ,int64, float32, float64, str, u
3. Arrays are dense. Each element of the array exists and has the same type.
4. Arrays have three main attributes:
5. Suppose **a** is an array:

`a.ndim`: Returns the **number of dimensions** of the array **a**.

`a.shape`: Returns the **shape** of the array as a tuple.

`a.dtype`: Returns the **data type** of the elements in the array.

0D NumPy array

Example code	Output
import numpy as np a = np.array(42) print (a.ndim) print (a.shape) print (a.dtype)	 0 ( ) int32

- This code creates a **0-dimensional array** (also called a scalar array) containing the single value 42 and prints three properties:
- a.ndim: Number of dimensions → 0 (This is a scalar, so it has no dimensions like rows or columns).
- a.shape: Shape of the array → () (An empty tuple, indicating no dimensions).
- a.dtype: Data type of the elements → int32 (The value 42 is stored as a 32-bit integer).

1D NumPy array

Example code	Output
import numpy as np a= np.array([1, 2, 3]) print (a.ndim) print (a.shape) print (a.dtype)	 1 (3,) int32

- This code creates a **1-dimensional array** (like a vector) and prints three properties:
- a.ndim: Number of dimensions → 1 (the array is one-dimensional).
- a.shape: Shape of the array → (3,) (meaning it has 3 elements in one dimension).
- a.dtype: Data type of the elements → int32 (typically int64 or int32, depending on your system).

2D NumPy array

Example code	Output
import numpy as np a = np.array([[1,2,3],[4,5,6]],dtype=np.float32) print (a.ndim) print (a.shape) print (a.dtype)	 2 (2, 3) float32

- This code creates a **2D NumPy array** with a specified data type (float32) and prints three properties (list of lists):
- a.ndim: Number of dimensions → 2
- a.shape: Shape of the array → (2, 3) (2 rows, 3 columns)
- a.dtype: Data type of the elements → float32

3D NumPy array

Example code	Output
import numpy as np a = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]], dtype=np.float32) print (a.ndim) print (a.shape) print (a.dtype)	 3 (2, 2, 3) float32

- This code creates a 3D array with shape (2, 2, 3) in a specified data type (float32) and prints three properties:
- a.ndim: Number of dimensions → 3
- a.shape: Shape of the array → (2, 2, 3) (The size of the array in each dimension, like a stack of 2 matrices, each of size 2×3).
  - First dimension (2): There are **2 blocks** (or layers).
  - Second dimension (2): Each block has **2 rows**.
  - Third dimension (3): Each row has **3 elements** (columns).
- a.dtype: Data type of the elements → float32

NumPy array data types

- All elements in a NumPy array must have the **same data type**.
- Choosing the right data type helps optimize **memory usage** and **performance**.

Data Type	Description	Example code	Typical Use Case
np.uint8	Unsigned 8-bit integer (0 to 255)	np.array([255, 128], dtype=np.uint8)	Image processing (pixel values)
np.int32	Signed 32-bit integer	np.array([1000], dtype=np.int32)	General-purpose integers, memory-efficient
np.int64	Signed 64-bit integer	np.array([10000000000], dtype=np.int64)	Large integer computations
np.float32	32-bit floating point	np.array([3.14], dtype=np.float32)	Scientific computing, lower memory usage
np.float64	64-bit floating point (default for floats)	np.array([3.1415926535], dtype=np.float64)	High-precision numerical analysis
np.bool_	Boolean type (True or False)	np.array([True, False], dtype=np.bool_)	Logical operations, masking
np.complex64	Complex number (64-bit: 2×float32)	np.array([1+2j], dtype=np.complex64)	Signal processing, scientific computing
np.complex128	Complex number (128-bit: 2×float64)	np.array([1+2j], dtype=np.complex128)	High-precision complex calculations

Arrays, creation

- np.ones, np.zeros
- np.arange
- np.concatenate
- np.astype
- np.zeros_like, np.ones_like
- np.random.random

Arrays, creation

Function	Description	Example code	Output
`np.ones(shape)`	Creates an array filled with ones	`np.ones((2, 3))`	`[[1. 1. 1.]` ` [1. 1. 1.]]`
`np.zeros(shape)`	Creates an array filled with zeros	`np.zeros((2, 3))`	`[[0. 0. 0.]` ` [0. 0. 0.]]`
`np.arange(start, stop, step)`	Creates evenly spaced values within a range	`np.arange(0, 10, 2)`	`[0 2 4 6 8]`
`np.concatenate((a1, a2))`	Joins arrays along an axis	`np.concatenate(([1, 2], [3, 4]))`	`[1 2 3 4]`
`np.astype(dtype)`	Converts array to a specified data type	`np.array([1.5, 2.5]).astype(int)`	`[1 2]`
`np.zeros_like(array)`	Creates a zero array with same shape/type as another	`np.zeros_like(np.array([[1, 2], [3, 4]]))`	`[[0 0]` ` [0 0]]`
`np.ones_like(array)`	Creates a ones array with same shape/type as another	`np.ones_like(np.array([[1, 2], [3, 4]]))`	`[[1 1]` ` [1 1]]`
`np.random.random(shape)`	Generates random floats in range [0.0, 1.0)	`np.random.random((2, 2))`	`[[0.45635497 0.22328338]` ` [0.15761189 0.94341577]]`

More Examples

- np.ones, np.zeros
- np.arange
- np.concatenate
- np.astype
- np.zeros_like, np.ones_like
- np.random.random

```
>>> np.ones((3,5),dtype=np.float32)
array([[ 1.,  1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.,  1.]], dtype=float32)
```

```
>>> np.zeros((6,2),dtype=np.int8)
array([[0, 0],
       [0, 0],
       [0, 0],
       [0, 0],
       [0, 0],
       [0, 0]], dtype=int8)
```

Arrays, creation

- np.ones, np.zeros
- **np.arange**
- np.concatenate
- np.astype
- np.zeros_like, np.ones_like
- np.random.random

```
>>> np.arange(1334,1338)
array([1334, 1335, 1336, 1337])
```

Examples

- np.ones, np.zeros
- np.arange
- **np.concatenate**
- np.astype
- np.zeros_like, np.ones_like
- np.random.random

```
>>> A = np.ones((2,3))
>>> B = np.zeros((4,3))
>>> np.concatenate([A,B])
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.],
       [ 0.,  0.,  0.],
       [ 0.,  0.,  0.],
       [ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
>>>
```

Arrays, creation

- np.ones, np.zeros
- np.arange
- **np.concatenate**
- np.astype
- np.zeros_like, np.ones_like
- np.random.random

```
>>> A = np.ones((4,1))
>>> B = np.zeros((4,2))
>>> np.concatenate([A,B], axis=1)
array([[ 1.,  0.,  0.],
       [ 1.,  0.,  0.],
       [ 1.,  0.,  0.],
       [ 1.,  0.,  0.]])
```

Arrays, creation

- np.ones, np.zeros
- np.arange
- np.concatenate
- **np.astype**
- np.zeros_like, np.ones_like
- np.random.random

```
>>> A
array([[ 4670.5,  4670.5,  4670.5],
       [ 4670.5,  4670.5,  4670.5],
       [ 4670.5,  4670.5,  4670.5],
       [ 4670.5,  4670.5,  4670.5],
       [ 4670.5,  4670.5,  4670.5]], dtype=float32)
>>> print(A.astype(np.uint16))
[[4670 4670 4670]
 [4670 4670 4670]
 [4670 4670 4670]
 [4670 4670 4670]
 [4670 4670 4670]]
```

Arrays, creation

- np.ones, np.zeros
- np.arange
- np.concatenate
- np.astype
- np.zeros_like, np.ones_like
- np.random.random

```
>>> a = np.ones((2,2,3))
>>> b = np.zeros_like(a)
>>> print(b.shape)
```

Arrays, creation

- np.ones, np.zeros
- np.arange
- np.concatenate
- np.astype
- np.zeros_like, np.ones_like
- **np.random.random**

```
>>> np.random.random((10,3))
array([[ 0.61481644,  0.55453657,  0.04320502],
       [ 0.08973085,  0.25959573,  0.27566721],
       [ 0.84375899,  0.2949532 ,  0.29712833],
       [ 0.44564992,  0.37728361,  0.29471536],
       [ 0.71256698,  0.53193976,  0.63061914],
       [ 0.03738061,  0.96497761,  0.01481647],
       [ 0.09924332,  0.73128868,  0.22521644],
       [ 0.94249399,  0.72355378,  0.94034095],
       [ 0.35742243,  0.91085299,  0.15669063],
       [ 0.54259617,  0.85891392,  0.77224443]])
```

Arrays, danger zone

- Must be dense, no holes.
- Must be one type
- Cannot combine arrays of different shape

```
>>> np.ones([7,8]) + np.ones([9,3])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: operands could not be broadcast together
 with shapes (7,8) (9,3)
```

Shaping

- Shaping refers to changing or inspecting the structure of a NumPy array .
- It defines how many dimensions an array has and how many elements are in each dimension .
- You can reshape an array using .reshape() as long as the total number of elements stays the same .
- Shaping is useful for preparing data for operations like broadcasting, matrix multiplication, or feeding into machine learning models .

Shaping

Example code	Description	Output	Result shape
`a = np.array([1,2,3,4,5, 6])`	Creates a 1D NumPy array with 6 elements	`[1 2 3 4 5 6]`	Shape: `(6,)`

`a.reshape(3,2)`	Reshapes array into 3 rows and 2 columns	`[[1 2]`
		` [3 4]`
		` [5 6]]`
			Shape: `(3, 2)`

`a.reshape(2,-1)`	Reshapes array into 2 rows, inferring columns automatically	`[[1 2 3]` ` [4 5 6]]`	Shape: `(2, 3)`
`a.ravel()`	Flattens the array into 1D (row-major order by default)	`[1 2 3 4 5 6]`	Shape: `(6,)`
Total elements fixed	You cannot reshape to a shape with a different total number of elements	`reshape(2, 4)` on 6 elements → ❌ Error	Must match original total: 6
Use -1 to infer shape	-1 lets NumPy calculate the appropriate dimension automatically	`reshape(2, -1)` → `[[1,2,3],[4,5,6]]`	NumPy infers 3 columns

Indexing

- Indexing is the process of accessing specific elements or sections of a NumPy array .
- NumPy uses zero-based indexing, meaning the first element is at index 0
- You can index arrays using single values, slices, or tuples for multi-dimensional arrays .
- Multi-dimensional indexing uses comma-separated values: array[row, column] .
- You can use negative indices to access elements from the end of an array .
- Indexing can also be done with lists or boolean arrays for advanced selection .
- Slicing returns a view (shared memory), while fancy indexing returns a copy .

Indexing

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

Expression	Description	Output
`x[0, 0]`	Top-left element (row 0, column 0)	1
`x[0, -1]`	First row, last column	3
`x[0, :]`	All columns of the first row	**[1,2,3]**
`x[:, 0]`	All rows of the first column	[1 ,4]

Slicing arrays

- Slicing is used to extract a portion of an array using a range of indices .
- Slicing arrays is almost the same as slicing lists, except you can specify multiple dimensions.
- The basic syntax is **start:stop:step** .stsil nohtyP ot ralimis ,
- You can slice across multiple dimensions using comma-separated slices:

  array[row_start:row_stop, col_start:col_stop] .
- Slicing returns a view of the original array, meaning changes to the slice affect the original data .

Example: Slicing arrays

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

Expression	Description	Output
a[1]	Selects the second row (index 1)	[4 ,5 ,6]
a[1, :]	Selects all columns of the second row	[4 ,5 ,6]
a[1, 1:]	Selects columns from index 1 onward in row 1	[5 ,6]
a[:1, 1:]	Selects row 0 and columns from index 1 onward	[3,2]

Return values

- Numpy functions return either views or copies.
- Views share memory with the original array. Changes affect both
- Copies are independent. Changes do not affect the original .
- Slicing usually returns a view .
- Fancy indexing (lists, boolean arrays) returns a copy .
- The numpy documentation says which functions return views or copies
- np.copy, np.view make explicit copies and views.

Views vs Copies

```
a = np.array([0,1,2,3,4])
```

Operation	Type	Description	Code Example	Output / Behavior
`a[1:4]`	View	Slicing returns a view (shared data)	`b = a[1:4]` `b[0]=99`	**a** → `[0,99,2,3,4]` (original modified)
`np.copy(a)`	Copy	Creates a deep copy (independent data)	`b = np.copy(a)` `b[0]=100`	**a** remains unchanged
`a[[1,3]]`	Copy	Fancy indexing (list or boolean) returns a copy	`b = a[[1,3]]` `b[0]=88`	**a** remains unchanged
`a.view()`	View	Explicitly creates a view of the array	`b = a.view();` `b[0]=77`	**a** → `[77,...]` (original modified)
`a[:,0]`	View	Column slice in 2D array returns a view	`a = np.array([[1,2],[3,4]]); b = a[:,0]; b[0]=9`	**a** → `[[9,2],[3,4]]`
`a[0,:]`	View	Row slice in 2D array returns a view	`b = a[0,:]; b[1]=99`	**a** → `[[9,99],[3,4]]`

Mathematical operators

- Arithmetic operations are element-wise
- Logical operator return a bool array
- In place operations modify the array

Arithmetic Operations (Element-wise)

- Any arithmetic operations between equal-size arrays applies the operation element-wise:

Operation	Code Example	Output
Addition	`np.array([1, 2]) + np.array([3, 4])`	[4,6]
Subtraction	`np.array([5, 6]) - np.array([2, 1])`	[3,5]
Multiplication	`np.array([2, 3]) * np.array([4, 5])`	[8,15]
Division	`np.array([10, 20]) / np.array([2, 4])`	[5.0 ,5.0]

Logical Operations (Boolean Arrays)

- Logical operator return a bool array

Operation	Code Example	Output
Greater Than	`np.array([1, 2, 3]) > 2`	`[False, False, True]`
Less Than or Equal	`np.array([1, 2, 3]) <= 2`	`[True, True, False]`
Equality Check	`np.array([1, 2, 3]) == np.array([1, 0, 3])`	`[True, False, True]`
Logical AND	`np.logical_and([True, False], [True, True])`	`[True, False]`

In-place Operations (Modify Original Array)

- In place operations modify the array

Operation	Code Example	Output
In-place Addition	`a = np.array([1, 2]); a += [3, 4]`	`a = [4, 6]`
In-place Subtraction	`a = np.array([5, 6]); a -= [1, 2]`	`a = [4, 4]`
In-place Multiplication	`a = np.array([2, 3]); a *= 2`	`a = [4, 6]`
In-place Division	`a = np.array([8.0, 4.0]); a /= 2`	`a = [4.0, 2.0]`

More Examples

- Arithmetic operations are element-wise
- Logical operator return a bool array
- In place operations modify the array

```
>>> a
array([1, 2, 3])
>>> b
array([ 4,  4, 10])
>>> a * b
array([ 4,  8, 30])
```

More Examples

- Arithmetic operations are element-wise
- **Logical operator return a bool array**
- In place operations modify the array

```
>>> a
array([[ 0.93445601,  0.42984044,  0.12228461],
       [ 0.06239738,  0.76019703,  0.11123116],
       [ 0.14617578,  0.90159137,  0.89746818]])
>>> a > 0.5
array([[ True, False, False],
       [False,  True, False],
       [False,  True,  True]], dtype=bool)
```

More Examples

- Arithmetic operations are element-wise
- Logical operator return a bool array
- In place operations modify the array

```
>>> a
array([[ 4, 15],
       [20, 75]])
>>> b
array([[ 2,  5],
       [ 5, 15]])
>>> a /= b
>>> a
array([[2, 3],
       [4, 5]])
```

Math, upcasting

Just as in Python and Java, the result of a math operator is cast to the more general or precise datatype.

uint64 + uint16 => uint64

float32 / int32 => float32

Warning: upcasting does not prevent overflow/underflow. You must manually cast first.

Use case: images often stored as uint8. You should convert to float32 or float64 before doing math.→ explanation in next slide

Overflow problem

- Even though NumPy can upcast automatically, it **does so *after* the operation** — not before. So if the operation itself overflows in the original data type, the result is already corrupted before casting.
- Example (overflow problem):

```
a = np.array([250], dtype=np.uint8)# image
b = np.array([10], dtype=np.uint8)
print(a + b)
```

```
[4]
```

Because `uint8` can only store numbers 0–255.
`250 + 10 = 260`, but 260 doesn't fit, so it wraps around to 4.
Even if NumPy later upcasts to a bigger type, the **damage is already done**.

Overflow problem(cont.)

- To avoid overflow, **convert (cast) your array to a higher type before performing math.**

```
a = np.array([250], dtype=np.uint8)
b = np.array([10], dtype=np.uint8)

result = a.astype(np.float32) + b.astype(np.float32)
print(result)
```

```
[260.]
```

No overflow — because the computation happened in floating point.

Math, universal functions

Function	Description	Code Example	Output
`np.exp(x)`	Exponential function: e^x **e** refers to **Euler's number**, which is a **fundamental mathematical constant** $e = 2.71$	`np.exp([0, 1, 2])`	`[1.         2.71828183 7.3890561 ]`
`np.sqrt(x)`	Square root of each element	`np.sqrt([0, 1, 4])`	`[0. 1. 2.]`
`np.sin(x)`	Sine of each element (in radians)	`np.sin([0, np.pi/2, np.pi])`	`[0.0000000e+00 1.0000000e+00 1.2246468e-16]`
`np.cos(x)`	Cosine of each element (in radians)	`np.cos([0, np.pi/2, np.pi])`	`[ 1.000000e+00  6.123234e-17 -1.000000e+00]`
`np.isnan(x)`	Checks for NaN (Not a Number) values	`np.isnan([1.0, np.nan, 2.0])`	`[False  True False]`

More Examples

Also called ufuncs

Element-wise

Examples:

- np.exp
- np.sqrt
- np.sin
- np.cos
- np.isnan

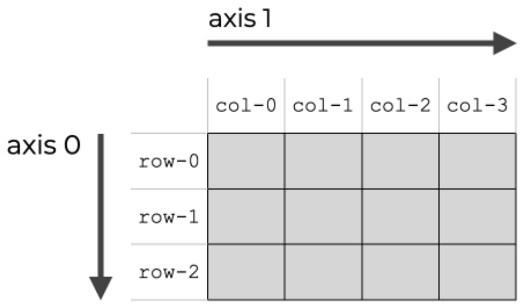
```
>>> a
array([[ 1,  4],
       [ 9, 16],
       [25, 36]])
>>> np.sqrt(a)
array([[ 1.,  2.],
       [ 3.,  4.],
       [ 5.,  6.]])
```

Axes

```
a.sum() # sum all entries

a.sum(axis=0) # sum over rows

a.sum(axis=1) # sum over columns

a.sum(axis=1, keepdims=True)
```

1. Use the axis parameter to control which axis NumPy operates on
2. Typically, the axis specified will disappear, keepdims keeps all dimensions

Axes



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

Code Example	Output	Description
`a.sum()`	21	Sums all elements in the array.
`a.sum(axis=0)`	[5 ,7 ,9]	Sums over rows (i.e., column-wise sum).
`a.sum(axis=1)`	[6 ,15]	Sums over columns (i.e., row-wise sum).
`a.sum(axis=1, keepdims=True)`	[[6] ,[15]]	Same as above, but keeps the original number of dimensions.

https://sharpsight.ai/blog/numpy-axes-explained/

Transposition

- **Transpose (.T)**: Swaps the first two axes of a 2D array.
- **Shape must be compatible**: Transposing doesn't change the number of elements, only their arrangement.

Code example	Description	Output	Shape

`a = np.arange(10).reshape(5, 2)`	Creates a 2D array with shape (5, 2) from values 0 to 9	`[[0 1]`
		` [2 3]`
		` [4 5]`
		` [6 7]`
		` [8 9]]`
			Shape: `(5, 2)`

`a.T`	Transposes the array (swaps rows and columns)	`[[0 2 4 6 8]` ` [1 3 5 7 9]]`	Shape: `(2, 5)`

Broadcasting

- When working with arrays of different shapes, Python uses **broadcasting** to make them compatible for operations like addition or multiplication.

Broadcasting Rules in Simple Terms:

- **Start comparing shapes from the end** (right to left).
- If a dimension is 1, it can be **stretched** to match the other.
- If a dimension is **not 1**, it must be **exactly the same** as the other.
- If needed, Python will **add extra dimensions of size 1 to the beginning of the shape** to help match.
- If the shapes can't be matched using these rules, Python will raise an error.

Broadcasting

Example	Array A Shape	Array B	Broadcasted Shapes	Result Shape	Explanation
`A + 1`	(4 ,3)	() *(scalar)*	(4 ,3) + (4 ,3)	(4 ,3)	Scalar 1 is broadcast to every element
`A + B`	(4 ,3)	(4 ,1)	(4 ,3) + (4 ,3)	(4 ,3)	Row vector is repeated across 3 rows
`A + B`	(4 ,3)	(1 ,3)	(4 ,3) + (4 ,3)	(4 ,3)	Column vector is repeated across 4 columns
`A + B`	(1 ,3)	(4 ,1)	(4 ,3) + (4 ,3)	(4 ,3)	Both arrays broadcast to full shape
`A + B`	(5 ,4 ,3)	(5 ,4)	(5 ,4 ,1) + (5 ,4 ,3)	(5 ,4 ,3)	Extra dimension added to left of B
`A + B`	(5 ,1 ,3)	(1 ,4 ,1)	(5 ,4 ,3) + (5 ,4 ,3)	(5 ,4 ,3)	All dimensions broadcast
`A + B`	(4 ,3)	(4 ,2)	❌ Incompatible	❌ Error	First dimensions (3 vs 2) don't match

Example 1:

```python
a = np.array([1, 2, 3])
b = 5
print(a + b)
```

```
[6 7 8]
```

- Shape of a: (3,)
- Shape of b: () — scalar
- NumPy “stretches” the scalar 5 into [5, 5, 5] → adds elementwise.
- ✔️ Shapes matched via broadcasting → (3,)

Example 2:

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

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

- Shapes: `(2, 3)` and `(3,)`
- NumPy adds a dimension to `B` → `(1, 3)`
- Then broadcasts it over the 2 rows.

✔️ Works because the last dimensions match (3).

Example 3:

```python
import numpy as np

A = np.array([[1, 2]])             # shape (1, 2)
B = np.array([[10],[20],[30]])     # shape (3, 1)
print(a+b)
```

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

- •Shapes: (1, 3) and (2,1)
- •For the **last dimension**: 2 vs 1 → stretch B's last dimension from 1 to 2.
- •For the **second-to-last dimension**: 1 vs 3 → stretch A's first dimension from 1 to 3.
- •So both arrays become shape (3, 2) during the operation.

Example 4:

```
import numpy as np
a=np.array([[1,2,3],[4,5,6],[7,8,9]])
b=np.array([[1,2],[3,4],[5,6]])
print(a.shape)
print(b.shape)
print(a+b)
```

```
(3, 3)
(3, 2)
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
/tmp/ipython-input-2706095471.py in <cell line: 0>()
      4 print(a.shape)
      5 print(b.shape)
----> 6 print(a+b)

ValueError: operands could not be broadcast together with shapes (3,3) (3,2)
```

Because the second dimensions (3 vs 2) can't be stretched to match.

Broadcasting example

Suppose we want to add a color value to an image

a.shape is 100, 200, 3

b.shape is 3

a + b will pad b with two extra dimensions so it has an effective shape of 1 x 1 x 3.

So, the addition will broadcast over the first and second dimensions.

Broadcasting failures

If a.shape is 100, 200, 3 but b.shape is 4 then a + b will fail. The trailing dimensions must have the same shape (or be 1)

References

- Jayarathna, S. (2021). *CS 620 / DASC 600: Introduction to Data Science & Analytics* [Course syllabus]. Old Dominion University.