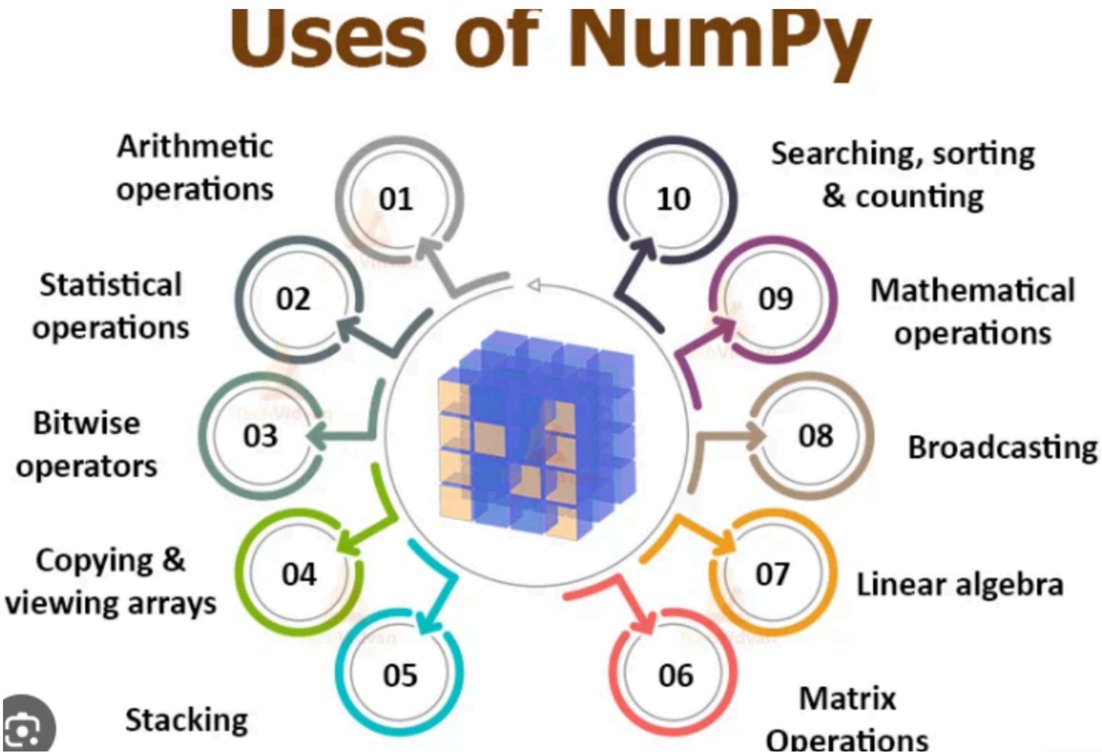
Complete NUMPY DOCUMENTATION



1. Array Creation Functions

In [3]:
```python
import numpy as np
```

In [4]:
```python
# Create an array from a list
a = np.array([1, 2, 3])
print("Array a:", a)
```

```
Array a: [1 2 3]
```

In [5]:
```python
# Create an array with evenly spaced values
b = np.arange(0, 10, 2)  # Values from 0 to 10 with step 2
print("Array b:", b)
```

```
Array b: [0 2 4 6 8]
```

In [6]:
```python
# Create an array with linearly spaced values
c = np.linspace(0, 1, 5)  # 5 values evenly spaced between 0 and 1
print("Array c:", c)
```

```
Array c: [0.   0.25 0.5  0.75 1.  ]
```

In [7]:
```python
# Create an array filled with zeros
d = np.zeros((2, 3))  # 2x3 array of zeros
print("Array d:\n", d)
```

```
Array d:
 [[0. 0. 0.]
 [0. 0. 0.]]
```

In [8]:
```
# Create an array filled with ones
e = np.ones((3, 2))  # 3x2 array of ones
print("Array e:\n", e)
```

```
Array e:
 [[1. 1.]
 [1. 1.]
 [1. 1.]]
```

In [9]:
```
# Create an array filled with ones
e = np.ones((3, 2), dtype=int)  # 3x2 array of ones
print("Array e:\n", e)
```

```
Array e:
 [[1 1]
 [1 1]
 [1 1]]
```

In [10]:
```
# Create an identity matrix
f = np.eye(4)  # 4x4 identity matrix
print("Identity matrix f:\n", f)
```

```
Identity matrix f:
 [[1. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 1. 0.]
 [0. 0. 0. 1.]]
```

In [11]:
```
# Create an identity matrix
f = np.eye((4), dtype=int)  # 4x4 identity matrix
print("Identity matrix f:\n", f)
```

```
Identity matrix f:
 [[1 0 0 0]
 [0 1 0 0]
 [0 0 1 0]
 [0 0 0 1]]
```

2. Array Manipulation Functions

In [13]:
```
# Reshape an array
a1 = np.array([1, 2, 3])
reshaped = np.reshape(a1, (1, 3))  # Reshape to 1x3
print("Reshaped array:", reshaped)
```

```
Reshaped array: [[1 2 3]]
```

In [14]:
```
# Reshape an array
a1 = np.array([1, 2, 3])
reshaped = np.reshape(a1, (3, 1))  # Reshape to 3x1
print("Reshaped array:\n", reshaped)
```

```
Reshaped array:
 [[1]
 [2]
 [3]]
```

In [15]:
```
# Flatten an array
f1 = np.array([[1, 2], [3, 4]])
```

```python
flattened = np.ravel(f1)  # Flatten to 1D array
print("Flattened array:", flattened)
```

```
Flattened array: [1 2 3 4]
```

In [16]:
```python
print(f1[0, 1])  # Access element in 0th row, 1st column
```

```
2
```

In [17]:
```python
# Transpose an array
e1 = np.array([[1, 2], [3, 4]])
transposed = np.transpose(e1)  # Transpose the array
print("Transposed array:\n", transposed)
```

```
Transposed array:
 [[1 3]
 [2 4]]
```

- Step-by-Step Explanation:

Original Array e1:

[[1 2] [3 4]]

This is a 2x2 matrix:

Row 0: [1, 2] Row 1: [3, 4]

- Transposing with np.transpose(e1):
- Transposing swaps the rows with columns.

So:

- The first column [1, 3] becomes the first row.
- The second column [2, 4] becomes the second row.

Resulting Transposed Array:

[[1 3] [2 4]]

In [19]:
```python
# Stack arrays vertically
a2 = np.array([1, 2])
b2 = np.array([3, 4])
stacked = np.vstack([a2, b2])  # Stack a and b vertically
print("Stacked arrays:\n", stacked)
```

```
Stacked arrays:
 [[1 2]
 [3 4]]
```

3. Mathematical Functions

In [21]:
```python
# Add two arrays
g = np.array([1, 2, 3, 4])
```

```
added = np.add(g, 2)  # Add 2 to each element
print("Added 2 to g:", added)
```

```
Added 2 to g: [3 4 5 6]
```

In [22]:
```
# Square each element
squared = np.power(g, 2)  # Square each element
print("Squared g:", squared)
```

```
Squared g: [ 1  4  9 16]
```

In [23]:
```
# Square root of each element
sqrt_val = np.sqrt(g)  # Square root of each element
print("Square root of g:", sqrt_val)
```

```
Square root of g: [1.         1.41421356 1.73205081 2.        ]
```

In [24]:
```
print(a1)
print(g)
```

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

In [25]:
```
# Dot product of two arrays
a2 = np.array([1, 2, 3])
dot_product = np.dot(a2, g)  # Dot product of a and g
print("Dot product of a and g:", dot_product)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[25], line 3
      1 # Dot product of two arrays
      2 a2 = np.array([1, 2, 3])
----> 3 dot_product = np.dot(a2, g)  # Dot product of a and g
      4 print("Dot product of a and g:", dot_product)

ValueError: shapes (3,) and (4,) not aligned: 3 (dim 0) != 4 (dim 0)
```

In [34]:
```
print(a)
print(a1)
```

```
[1 2 3]
[1 2 3]
```

In [45]:
```
a3 = np.array([1, 2, 3])
dot_product = np.dot(a1, a)  # Dot product of a and g
print("Dot product of a1 and a:", dot_product) # 1.1 + 2.2 + 3.3 = 1 + 4 + 9 = 1
```

```
Dot product of a1 and a: 14
```

- What is the dot product?

The dot product of two 1D vectors a and b of the same length is calculated as:

dot(a,b) = $a1 \cdot b1 + a2 \cdot b2 + a3 \cdot b3$

- Example with values:

a1 = np.array([1, 2, 3])

a2 = np.array([4, 5, 6])

Now compute:

1·4 + 2·5 + 3·6 = 4 + 10 + 18 = 32

Dot product of a1 and a2: 32

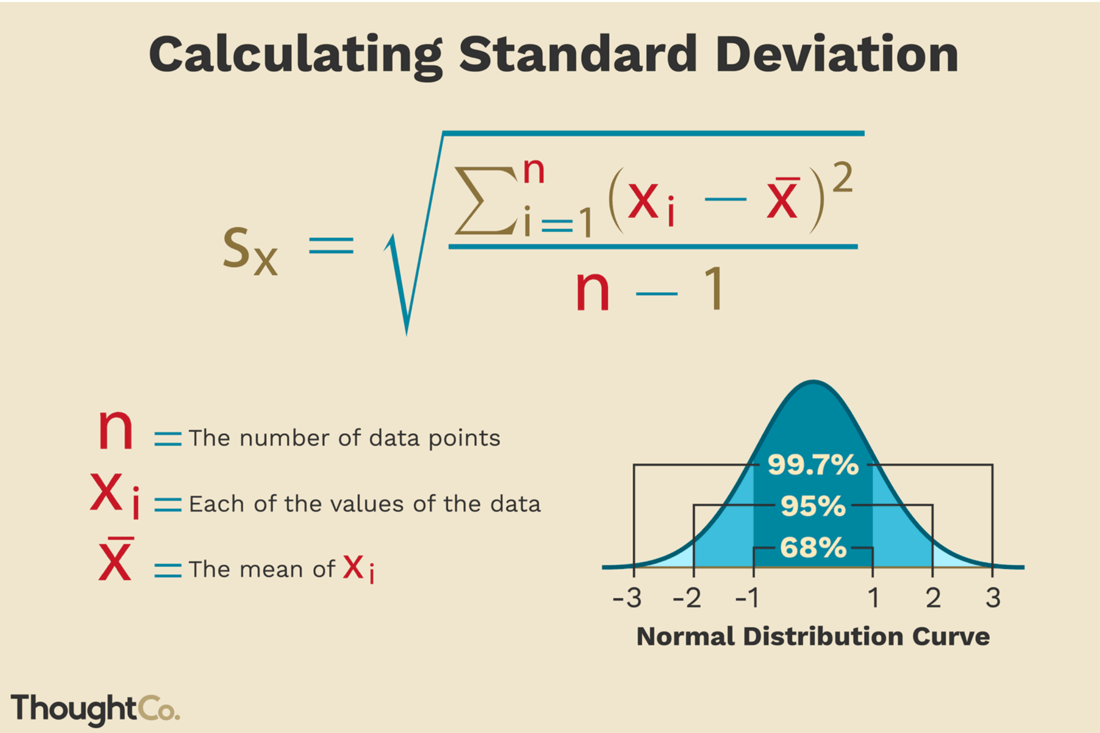
4. Statistical Functions

In [49]:
```
s = np.array([1, 2, 3, 4])
mean = np.mean(s)
print("Mean of s:", mean)
```

```
Mean of s: 2.5
```

In [51]:
```
# Standard deviation of an array
std_dev = np.std(s)
print("Standard deviation of s:", std_dev)
```

```
Standard deviation of s: 1.118033988749895
```



In [53]:
```
# Minimum element of an array
minimum = np.min(s)
print("Min of s:", minimum)
```

```
Min of s: 1
```

In [55]:
```
# Maximum element of an array
maximum = np.max(s)
print("Max of s:", maximum)
```

```
Max of s: 4
```

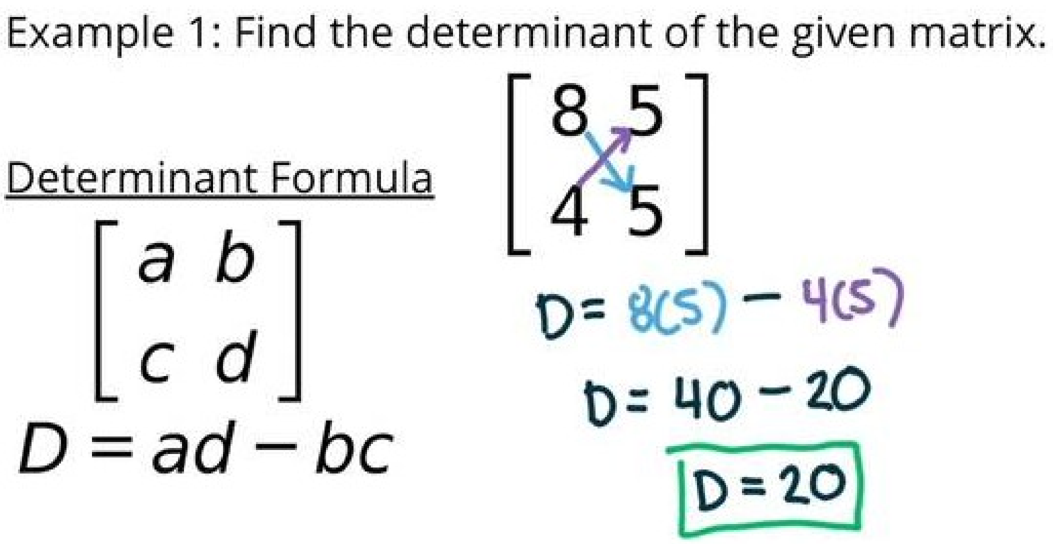
5. Linear Algebra Functions

In [59]:
```
# Create a matrix
matrix = np.array([[1, 2], [3, 4]])
print(matrix)
```

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

In [61]:
```
# Determinant of a matrix
determinant = np.linalg.det(matrix)
print("Determinant of matrix:", determinant)
```

```
Determinant of matrix: -2.0000000000000004
```



- What is a determinant?

The determinant of a 2×2 matrix:

[ab

cd]

is calculated using the formula:

det = ad−bc

🧮 Applying it to your matrix: Your matrix is:

[12

34]

So det=(1×4)−(2×3)=4−6=−2 ✅ Output:

Determinant of matrix: -2.0000000000000004

- ⚠️ Note: You may see -2.0000000000000004 instead of exactly -2 due to floating-point precision errors common in numerical computing.

✅ Summary: The determinant of the matrix [[1, 2], [3, 4]] is -2.

It's calculated using 1*4* - 2*3*.

The tiny difference in decimal (-2.0000000000000004) is normal when working with floating-point numbers in NumPy.

In [64]:
```
# Inverse of a matrix
inverse = np.linalg.inv(matrix)
print("Inverse of matrix:\n", inverse)
```

```
Inverse of matrix:
 [[-2.   1. ]
 [ 1.5 -0.5]]
```

✅ What is the inverse of a matrix?

For a 2×2 matrix:

$$\text{Matrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

The inverse is:

$$\text{Matrix}^{-1} = \frac{1}{ad - bc} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix}$$

But this is only valid **if** the determinant $ad - bc \neq 0$.

🧮 Apply it to your matrix:

Given:

$$\text{matrix} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

- $a = 1, b = 2, c = 3, d = 4$
- Determinant = $1 \times 4 - 2 \times 3 = 4 - 6 = -2$

Now compute the inverse:

$$\text{inverse} = \frac{1}{-2} \begin{bmatrix} 4 & -2 \\ -3 & 1 \end{bmatrix} = \begin{bmatrix} -2.0 & 1.0 \\ 1.5 & -0.5 \end{bmatrix}$$

6. Random Sampling Functions

In [79]:
```
# Generate random values between 0 and 1
random_vals = np.random.rand(3)  # Array of 3 random values between 0 and 1
```

```
print("Random values:", random_vals)
```

```
Random values: [0.37853944 0.52519042 0.93221937]
```

In [83]:
```
# Set seed for reproducibility
np.random.seed(0)

# Generate random values between 0 and 1
random_vals = np.random.rand(3)  # Array of 3 random values between 0 and 1
print("Random values:", random_vals)
```

```
Random values: [0.5488135  0.71518937 0.60276338]
```

In [99]:
```
# Generate random integers
rand_ints = np.random.randint(0, 10, size=5)  # Random integers between 0 and 10
print("Random integers:", rand_ints)
```

```
Random integers: [6 7 7 8 1]
```

In [93]:
```
# Set seed for reproducibility
np.random.seed(0)

# Generate random integers
rand_ints = np.random.randint(0, 10, size=5)  # Random integers between 0 and 10
print("Random integers:", rand_ints)
```

```
Random integers: [5 0 3 3 7]
```

- np.random.seed(0) sets the seed for NumPy's random number generator.
- Think of it like "planting" a predictable starting point for generating random numbers.
- Why is this useful?

Random number generators in Python (and other languages) are deterministic — if you set the same seed, you get the same sequence of random numbers every time.

This is helpful when you want reproducibility, for example:

1. In experiments
2. Debugging
3. Teaching/demo code

If you run this every time with np.random.seed(0) before it, you will always get the same numbers.

Without setting the seed, the output would vary on every run.

✅ Summary:

np.random.seed(0) makes random output predictable and repeatable.

Use it when you want consistent results — like in testing or tutorials.

Change the number (0) to get a different sequence (e.g., np.random.seed(42)).

7. Boolean & Logical Functions

In [104…

```
# Check if all elements are True
# all
logical_test = np.array([True, False, True])
all_true = np.all(logical_test)  # Check if all are True
print("All elements True:", all_true)
```

```
All elements True: False
```

In [108…

```
# Check if all elements are True
logical_test = np.array([True, True])
all_true = np.all(logical_test)  # Check if all are True
print("All elements True:", all_true)
```

```
All elements True: True
```

In [112…

```
# Check if all elements are True
logical_test = np.array([False, False, False])
all_true = np.all(logical_test)  # Check if all are True
print("All elements True:", all_true)
```

```
All elements True: False
```

In [116…

```
# Check if all elements are True
logical_test = np.array([False, True, False])
all_true = np.all(logical_test)  # Check if all are True
print("All elements True:", all_true)
```

```
All elements True: False
```

In [118…

```
# Check if any elements are True
# any
any_true = np.any(logical_test)  # Check if any are True
print("Any elements True:", any_true)
```

```
Any elements True: True
```

8. Set Operations

In [120…

```
# Intersection of two arrays
set_a = np.array([1, 2, 3, 4])
set_b = np.array([3, 4, 5, 6])
intersection = np.intersect1d(set_a, set_b)
print("Intersection of a and b:", intersection)
```

```
Intersection of a and b: [3 4]
```

In [122…

```
# Union of two arrays
union = np.union1d(set_a, set_b)
print("Union of a and b:", union)
```

```
Union of a and b: [1 2 3 4 5 6]
```

9. Array Attribute Functions

In [124…

```
# Array attributes
a = np.array([1, 2, 3])
shape = a.shape  # Shape of the array
size = a.size    # Number of elements
dimensions = a.ndim  # Number of dimensions
dtype = a.dtype   # Data type of the array

print("Shape of a:", shape)
print("Size of a:", size)
print("Number of dimensions of a:", dimensions)
print("Data type of a:", dtype)
```

```
Shape of a: (3,)
Size of a: 3
Number of dimensions of a: 1
Data type of a: int32
```

10. Other Functions

In [126…

```
# Create a copy of an array
a = np.array([1, 2, 3])
copied_array = np.copy(a)  # Create a copy of array a
print("Copied array:", copied_array)
```

```
Copied array: [1 2 3]
```

In [135…

```
# Size in bytes of an array
array_size_in_bytes = a.nbytes  # Size in bytes
print("Size of a in bytes:", array_size_in_bytes)
```

```
Size of a in bytes: 12
```

Explanation:

- a = np.array([1, 2, 3]) creates a 1D NumPy array of integers:

a=[1,2,3]

- a.nbytes returns the total number of bytes consumed by the array's data only (not metadata like shape, type, etc.).

✅ How is it calculated?

- The formula is: nbytes = number of elements × size of each element in bytes

You can verify this with:

- print(a.size) # Number of elements: 3
- print(a.itemsize) # Size of one element in bytes

Assuming a.dtype is int64 (which is common on 64-bit systems):

- Each integer takes 8 bytes
- Total size = 3 elements × 8 bytes = 24 bytes

✅ Sample Output:

Size of a in bytes: 24

⚠️ If you're using a 32-bit system or specified dtype=np.int32, the result would be 12 bytes (3 × 4).

In [141…

```
# Check if two arrays share memory
# a = np.array([1, 2, 3])
# copied_array = np.copy(a)
shared = np.shares_memory(a, copied_array)  # Check if arrays share memory
print("Do a and copied_array share memory?", shared)
```

```
Do a and copied_array share memory? False
```

- Why is the answer False?

np.copy() creates a deep copy of the array.

That means all the data is duplicated into new memory.

So, the original (a) and the copy (copied_array) do not share memory.

✅ If you had used a view, like this:

b = a.view()

print(np.shares_memory(a, b)) # True

That would return True, because view() creates a shallow copy that shares memory with the original.

✅ Summary:

np.copy() → deep copy → no shared memory → False

np.view() or slicing → shallow copy → shared memory → True

========== THANK YOU ============

In [ ]: