NumPy Universal Functions

- ❖ Universal Functions Introduction
- ❖ Universal Functions Create Function
- ❖ Universal Functions Simple Arithmetic
- ❖ Universal Functions Rounding Decimals
- ❖ Universal Functions Logs
- ❖ Universal Functions Summations
- ❖ Universal Functions Products
- ❖ Universal Functions Differences
- ❖ Universal Functions Finding LCM
- ❖ Universal Functions Finding GCD
- ❖ Universal Functions Trigonometric
- ❖ Universal Functions Hyperbolic
- ❖ Universal Functions Set Operations

NumPy Universal Functions Introduction

What are Universal Functions in NumPy?

Universal Functions, commonly called **ufuncs**, are **special NumPy functions** that work **element-by-element** on NumPy arrays.
They allow you to perform **fast mathematical and logical operations** without using loops.

Why are ufuncs important?

Universal functions are important because they:

- Work **much faster** than normal Python loops
- Support **array operations directly**
- Automatically handle **broadcasting**
- Use **less memory**
- Are perfect for **data analysis and scientific computing**

Key Features of NumPy ufuncs

1. **Element-wise operation**
   Each element is processed independently.
2. **High performance**
   Implemented in C, so execution is very fast.
3. **Broadcasting support**
   Can operate on arrays of different shapes.
4. **Vectorized code**
   No need for for loops.

NumPy Universal Functions – Creating a Custom ufunc

In NumPy, apart from using built-in universal functions, we can also **create our own Universal Function (ufunc)**. A **custom ufunc** allows a user-defined Python function to behave like a NumPy ufunc, meaning it will work **element-by-element on arrays** and support vectorized operations.

To create a universal function in NumPy, we use the **np.frompyfunc()** method. This method converts a normal Python function into a NumPy ufunc.

Why Create a Custom ufunc?

Creating a custom ufunc is useful when:

- A required operation is **not available** in built-in ufuncs
- The same logic must be applied to **every array element**
- You want **clean and reusable** array-based code

Syntax to Create a ufunc

np.frompyfunc(function, input_count, output_count)

Parameters:

- **function** → Normal Python function
- **input_count** → Number of input arguments
- **output_count** → Number of values returned

Example: Creating a Custom Universal Function

Step 1: Define a normal Python function

```
def square(x):
    return x * x
```

Step 2: Convert it into a NumPy ufunc

```
import numpy as np

square_ufunc = np.frompyfunc(square, 1, 1)
```

Step 3: Apply it to a NumPy array

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

result = square_ufunc(arr)

print(result)
```

Output:

```
[1 4 9 16 25]
```

The function is automatically applied to **each element** of the array.

Example 2: Custom ufunc with Two Inputs

```
def add_numbers(a, b):
   return a + b

add_ufunc = np.frompyfunc(add_numbers, 2, 1)

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

y = np.array([10, 20, 30])

print(add_ufunc(x, y))
```

Output:

```
[11 22 33]
```

NumPy Universal Functions – Simple Arithmetic

In NumPy, **Universal Functions (ufuncs)** are used to perform **simple arithmetic operations** directly on arrays. These functions apply the operation **element by element**, which means the same calculation is automatically performed on every value in the array.

Simple arithmetic ufuncs remove the need for loops and make calculations **faster, cleaner, and more efficient**. They are commonly used in data analysis and numerical computing.

Common Simple Arithmetic Universal Functions

1. Addition (np.add)

Adds elements of two arrays.

```
import numpy as np

a = np.array([2, 4, 6])

b = np.array([1, 3, 5])

result = np.add(a, b)

print(result)
```

Output:

```
[3 7 11]
```

---

2. Subtraction (np.subtract)

Subtracts elements of one array from another.

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

```
y = np.array([2, 5, 10])

result = np.subtract(x, y)

print(result)
```

Output:

```
[ 8 15 20]
```

---

3. Multiplication (np.multiply)

Multiplies corresponding elements of arrays.

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

n = np.array([5, 6, 7])

result = np.multiply(m, n)

print(result)
```

Output:

```
[10 18 28]
```

---

4. Division (np.divide)

Divides elements of one array by another.

```
p = np.array([20, 40, 60])

q = np.array([2, 4, 5])

result = np.divide(p, q)

print(result)
```

Output:

[10. 10. 12.]

---

5. Power (np.power)

Raises elements of one array to the power of another.

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

exp = np.array([3, 2, 2])

result = np.power(base, exp)

print(result)
```

Output:

[8 9 16]

NumPy Universal Functions (Ufuncs) – Rounding Decimals

What are Universal Functions (Ufuncs)?

NumPy **Universal Functions (ufuncs)** are built-in functions that work **element-wise** on NumPy arrays.
They are:

- Very fast
- Easy to use
- Applied to each value in an array automatically

Rounding functions are a special type of ufunc used to **round decimal numbers**.

Why Rounding Decimals is Important?

In data analysis:

- We round prices, percentages, and averages
- We clean noisy decimal values
- We make data more readable in reports

Example:

Original value: 23.678945

Rounded value : 23.68

NumPy Rounding Decimal Functions

np.around() / np.round()

Rounds numbers to a specified number of decimal places.

Syntax:

np.around(array, decimals)

Example:

```
import numpy as np

data = np.array([12.345, 67.891, 45.567, 89.999])

rounded = np.around(data, 2)

print(rounded)
```

Output:

```
[12.35 67.89 45.57 90.  ]
```

✓ Rounds each value to **2 decimal places**

---

np.round() (same as np.around)

np.round(data, 1)

✓ Works exactly like around()

Rounding Without Decimals (Whole Numbers)

np.floor()

Rounds **down** to the nearest integer.

Example:

values = np.array([4.7, 2.3, 9.9, 5.1])

print(np.floor(values))

Output:

[4. 2. 9. 5.]

✓ Always rounds **down**

---

np.ceil()

Rounds **up** to the nearest integer.

print(np.ceil(values))

Output:

[5. 3. 10. 6.]

✓ Always rounds **up**

---

np.trunc()

Removes decimal part (no rounding).

print(np.trunc(values))

Output:

[4. 2. 9. 5.]

✓ Just **cuts** the decimal part

Difference Between Rounding Functions

Function	Behavior	Example (4.7)
round	Normal rounding	5
floor	Always down	4
ceil	Always up	5
trunc	Remove decimal	4

Real-Life Example (Marks Data)

marks = np.array([78.456, 89.123, 67.999, 92.555])

final_marks = np.round(marks, 1)

print(final_marks)

Output:

[78.5 89.1 68. 92.6]

NumPy Universal Functions (Ufuncs) – Logarithmic Functions

What are Logarithmic Functions?

A **logarithm** answers the question:

“Power kitni hai?”
If
[
a^x = b
]
then
[
\log_a(b) = x
]

In data science, logarithms are used to:

- Reduce very large values
- Handle exponential growth
- Normalize skewed data
- Improve model performance

NumPy provides **logarithmic ufuncs** that work **element-wise** on arrays.

Why NumPy Log Ufuncs?

✓ Very fast
✓ No loops required
✓ Apply automatically to each element
✓ Widely used in data analysis & ML

Types of Logarithmic Functions in NumPy

---

np.log() – Natural Log (Base e)

Definition:

Natural logarithm where base **e ≈ 2.718**

Syntax:

np.log(array)

Example:

import numpy as np

data = np.array([1, 2, 4, 10])

result = np.log(data)

print(result)

Output:

[0. 0.69314718 1.38629436 2.30258509]

✓ Used in **machine learning**, **statistics**, **growth rate analysis**

---

np.log10() – Log Base 10

Syntax:

np.log10(array)

Example:

values = np.array([1, 10, 100, 1000])

print(np.log10(values))

Output:

[0. 1. 2. 3.]

✓ Common in:

- Scientific calculations
- Earthquake scale
- Sound intensity (decibel)

---

np.log2() – Log Base 2

Syntax:

np.log2(array)

Example:

nums = np.array([1, 2, 4, 8, 16])

print(np.log2(nums))

Output:

[0. 1. 2. 3. 4.]

✓ Used in:

- Computer science
- Binary systems
- Data compression

Comparison of Log Functions

Function	Base	Example Input	Output
np.log()	e	10	2.302
np.log10()	10	100	2
np.log2()	2	8	3

Log of Zero & Negative Numbers

```
arr = np.array([1, 0, -5])

print(np.log(arr))
```

Output:

```
[0.    -inf   nan]
```

Explanation:

- log(0) → -inf
- log(negative) → nan (Not a Number)

✓ Always clean data before applying logs

Real-Life Example – Sales Data Normalization

```
sales = np.array([100, 500, 2000, 10000])


log_sales = np.log10(sales)

print(log_sales)
```

Output:

```
[2. 2.69897 3.30103 4. ]
```

✓ Helps reduce **data skewness**

Log with Custom Base (Formula Method)

NumPy does not directly provide custom base log, but we use:

[
\log_b(x) = \frac{\log(x)}{\log(b)}
]

Example:

```
x = np.array([8, 16, 32])

base = 2



custom_log = np.log(x) / np.log(base)

print(custom_log)
```

Output:

```
[3. 4. 5.]
```

NumPy Universal Functions (Ufuncs) – Products

What is Product in NumPy?

Product means multiplying numbers together.

In NumPy, product-related operations are handled using **universal functions (ufuncs)** and fast aggregation functions that:

- Work **element-wise**
- Are **high-performance**
- Easily handle **1D, 2D, and multi-dimensional arrays**

Why Use NumPy for Product Operations?

✓ Faster than Python loops
✓ Cleaner and shorter code
✓ Works directly on arrays
✓ Very useful in statistics & data analysis

Important NumPy Product Functions

---

np.prod() – Product of All Elements

Syntax:

np.prod(array)

Example (1D Array):

```
import numpy as np


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

result = np.prod(data)


print(result)
```

Output:

24

✓ Calculation: 2 × 3 × 4 = 24

Product in 2D Array

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


print(np.prod(matrix))
```

Output:

24

✓ Multiplies **all values**

---

axis Parameter in Product ⭐

a) Column-wise Product (axis=0)

```
print(np.prod(matrix, axis=0))
```

Output:

[3 8]

✓ Calculation:
Column 1 → 1 × 3 = 3
Column 2 → 2 × 4 = 8

---

b) Row-wise Product (axis=1)

print(np.prod(matrix, axis=1))

Output:

[2 12]

✓ Row 1 → 1 × 2 = 2
✓ Row 2 → 3 × 4 = 12

Visual Meaning of Axis

Axis	Operation
axis = 0	Column-wise
axis = 1	Row-wise

---

np.multiply() – Element-wise Multiplication (Ufunc)

Syntax:

np.multiply(array1, array2)

Example:

a = np.array([2, 4, 6])

b = np.array([3, 5, 7])

print(np.multiply(a, b))

Output:

[6 20 42]

✓ Multiplies corresponding elements

Broadcasting Example

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

print(np.multiply(arr, 2))

Output:

[2 4 6]

✓ Multiplies each element by 2

Cumulative Product – np.cumprod()

Definition:

Each element is the product of previous elements.

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

print(np.cumprod(values))

Output:

[2 6 24]

✓ Useful for:

- Growth calculation
- Compound interest
- Probability chains

---

np.nanprod() – Ignore NaN Values

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

print(np.nanprod(arr))
```

Output:

```
24.0
```

✓ Ignores missing values (NaN)

Real-Life Example – Stock Growth Factor

```
growth = np.array([1.05, 1.10, 0.95])

total_growth = np.prod(growth)


print(total_growth)
```

Output:

```
1.09725
```

✓ Shows total growth after multiple periods

Difference Between prod() and multiply()

Function	Purpose
np.prod()	Multiplies all elements
np.multiply()	Element-wise multiplication

NumPy Universal Functions (Ufuncs) – Differences

What is "Difference" in NumPy?

Difference means finding how much a value has **changed** compared to another value.

In NumPy, "differences" are commonly used to:

- Measure **change** between consecutive values
- Analyze **trends** in time-series data
- Calculate **increments / decrements**
- Detect sudden jumps or drops in data

NumPy provides fast functions (built on ufuncs) to compute differences **without loops**.

Why Use NumPy Difference Functions?

✓ Very fast
✓ Simple syntax
✓ Works on arrays directly
✓ Perfect for data analysis & statistics

Main NumPy Function for Differences

np.diff() – Difference Between Consecutive Elements

Definition:

np.diff() calculates the difference between **adjacent elements**.

Formula:

[
\text{diff}[i] = a[i+1] - a[i]
]

Syntax:

np.diff(array)

Example (1D Array):

import numpy as np

data = np.array([10, 15, 25, 40])

result = np.diff(data)

print(result)

Output:

[5 10 15]

✓ Calculations:

- 15 − 10 = 5
- 25 − 15 = 10
- 40 − 25 = 15

Length Change Rule ⚠️

If array length is **n**, the result length is **n − 1**

Difference in 2D Arrays

```
matrix = np.array([[1, 4, 9],
                   [2, 6, 12]])


print(np.diff(matrix))
```

Output:

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

✓ Difference is calculated **row-wise** by default

Using axis in np.diff()

a) Row-wise Difference (axis=1) – Default

```
np.diff(matrix, axis=1)
```

✓ Difference across columns

---

b) Column-wise Difference (axis=0)

```
np.diff(matrix, axis=0)
```

Output:

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

✓ Calculates difference between rows

Visual Understanding of Axis

Axis	Meaning
axis = 0	Vertical (between rows)
axis = 1	Horizontal (between columns)

Higher Order Differences (Second Difference)

Definition:

Difference of differences

values = np.array([2, 5, 10, 17])

print(np.diff(values, n=2))

Output:

[2 2]

✓ Used in:

- Polynomial trend analysis
- Acceleration calculation

Difference vs Subtraction (np.subtract())

Element-wise Subtraction:

a = np.array([20, 30, 40])

b = np.array([5, 10, 15])

print(np.subtract(a, b))

Output:

[15 20 25]

✓ np.subtract() subtracts **corresponding elements**

Key Difference Table

Function	Purpose
np.diff()	Difference between consecutive values
np.subtract()	Element-wise subtraction

Real-Life Example – Daily Sales Change

sales = np.array([100, 120, 115, 140, 160])

daily_change = np.diff(sales)

print(daily_change)

Output:

[20 -5 25 20]

✓ Shows daily increase or decrease in sales

Handling Negative Differences

Negative values mean:

- Decrease

- Drop in data

✓ Very useful for trend analysis

NumPy Universal Functions – Finding LCM (Least Common Multiple)

What is LCM?

LCM (Least Common Multiple) of two or more numbers is the **smallest positive number** that is **divisible by all the given numbers**.

Example (Math Concept)

- LCM of **4 and 6**
  - Multiples of 4 → 4, 8, 12, 16, ...
  - Multiples of 6 → 6, 12, 18, ...
  - Common smallest multiple → **12**
- So, **LCM = 12**

What is NumPy LCM ufunc?

NumPy provides a **universal function** called:

np.lcm()

It calculates the **LCM element-wise** for arrays or between numbers.

Being a **ufunc**, it is:

- Fast

- Vectorized
- Works on arrays without loops

Syntax

numpy.lcm(x, y)

Parameters:

- x → first number or array
- y → second number or array

Returns:

- LCM of x and y

Example 1: LCM of Two Numbers

```
import numpy as np


result = np.lcm(6, 8)

print(result)
```

Output:

24

Explanation:

- Multiples of 6 → 6, 12, 18, 24
- Multiples of 8 → 8, 16, 24
- Smallest common multiple → **24**

Example 2: LCM of Two Arrays

```
import numpy as np


a = np.array([2, 4, 6])

b = np.array([3, 5, 9])


result = np.lcm(a, b)

print(result)
```

Output:

```
[ 6 20 18 ]
```

Explanation:

a	b	LCM
2	3	6
4	5	20
6	9	18

Each element is calculated **pair-wise**.

Example 3: LCM of Array and Single Number

```
import numpy as np


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

result = np.lcm(arr, 10)
```

```
print(result)
```

Output:

```
[20 10 30]
```

Explanation:

- LCM(4,10) = 20
- LCM(5,10) = 10
- LCM(6,10) = 30

Example 4: LCM Using Multiple Values (Step by Step)

To find LCM of more than two numbers:

```
import numpy as np

from functools import reduce


numbers = np.array([4, 6, 8])

lcm_result = reduce(np.lcm, numbers)

print(lcm_result)
```

Output:

```
24
```

✓ Explanation:

- LCM(4,6) = 12
- LCM(12,8) = 24

Example 5: LCM in 2D Array

```
import numpy as np


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

b = np.array([[6, 9], [10, 15]])


result = np.lcm(a, b)

print(result)
```

Output:

```
[[ 6  9]
 [20 15]]
```

NumPy Universal Functions – Finding GCD

Full form :- (Greatest Common Divisor)

What is GCD?

GCD (Greatest Common Divisor) is the **largest positive number** that divides **two or more numbers exactly**, without leaving any remainder.

Simple Math Example

- Numbers: **18 and 24**

- Factors of 18 → 1, 2, 3, 6, 9, 18
- Factors of 24 → 1, 2, 3, 4, 6, 8, 12, 24
- Largest common factor → **6**

GCD = 6

What is NumPy GCD ufunc?

NumPy provides a **Universal Function (ufunc)** called:

np.gcd()

It calculates the **GCD element-by-element** for numbers or arrays.

Why is it a ufunc?

- Works on arrays automatically
- No need for loops
- Fast and optimized
- Handles large datasets easily

Syntax

numpy.gcd(x, y)

Parameters:

- x → first number or array
- y → second number or array

Returns:

- GCD of x and y

Example 1: GCD of Two Numbers

```
import numpy as np


result = np.gcd(20, 30)

print(result)
```

Output:

```
10
```

Explanation:

- Common divisors → 1, 2, 5, 10
- Greatest one → **10**

Example 2: GCD of Two Arrays

```
import numpy as np


a = np.array([12, 18, 24])

b = np.array([6, 12, 36])


result = np.gcd(a, b)

print(result)
```

Output:

```
[ 6  6 12 ]
```

Explanation:

a	b	GCD
12	6	6
18	12	6
24	36	12

Each element is processed **pair-wise**.

Example 3: GCD of Array and Single Number

```
import numpy as np


arr = np.array([15, 25, 35])

result = np.gcd(arr, 5)

print(result)
```

Output:

```
[5 5 5]
```

Explanation:

- 5 divides all values completely

Example 4: GCD of Multiple Numbers

To find GCD of more than two numbers:

```
import numpy as np

from functools import reduce
```

```
numbers = np.array([16, 24, 32])

gcd_result = reduce(np.gcd, numbers)

print(gcd_result)
```

Output:

```
8
```

Explanation:

- GCD(16,24) = 8
- GCD(8,32) = 8

Example 5: GCD in a 2D Array

```
import numpy as np


a = np.array([[8, 16], [18, 24]])

b = np.array([[4, 8], [6, 12]])


result = np.gcd(a, b)

print(result)
```

Output:

```
[[ 4  8]

 [ 6 12]]
```

GCD vs LCM (Quick Comparison)

Feature	GCD	LCM
Meaning	Largest common divisor	Smallest common multiple
NumPy function	np.gcd()	np.lcm()
Use	Simplification	Scheduling

NumPy Universal Functions – Trigonometric Functions

What are Trigonometric Functions?

Trigonometric functions are **mathematical functions** used to find relationships between **angles and sides of a triangle**.

The most common trigonometric functions are:

- **sin (sine)**
- **cos (cosine)**
- **tan (tangent)**

NumPy provides these as **Universal Functions (ufuncs)**, which means they work **fast and element-wise on arrays**.

Important Point (Very Important for Students)

NumPy trigonometric functions work in RADIANS, not degrees

Degrees	Radians
0°	0
30°	π/6
45°	π/4
60°	π/3
90°	π/2

To convert degrees to radians, NumPy provides:

np.deg2rad()

Common Trigonometric ufuncs in NumPy

Function	Description
np.sin()	Sine of angle
np.cos()	Cosine of angle
np.tan()	Tangent of angle
np.arcsin()	Inverse sine
np.arccos()	Inverse cosine
np.arctan()	Inverse tangent
np.deg2rad()	Degree → Radian
np.rad2deg()	Radian → Degree

Example 1: Sine Function

```
import numpy as np



angle = np.pi / 2   # 90 degrees

result = np.sin(angle)

print(result)
```

Output:

1.0

Explanation:

- sin(90°) = 1
- π/2 radians = 90 degrees

Example 2: Cosine Function

```
import numpy as np


angle = 0

result = np.cos(angle)

print(result)
```

Output:

1.0

Explanation:

- cos(0°) = 1

Example 3: Tangent Function

```
import numpy as np


angle = np.pi / 4   # 45 degrees
```

```
result = np.tan(angle)

print(result)
```

Output:

```
0.9999999999999999
```

✓ This is approximately **1** (small floating-point error).

Example 4: Trigonometric Functions on Array

```
import numpy as np


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

result = np.sin(angles)

print(result)
```

Output:

```
[0. 1. 0.]
```

Explanation:

- sin(0°) = 0
- sin(90°) = 1
- sin(180°) = 0

Example 5: Degrees to Radians

```
import numpy as np
```

```
degrees = np.array([0, 30, 45, 60, 90])

radians = np.deg2rad(degrees)

print(radians)
```

Output:

```
[0.         0.52359878 0.78539816 1.04719755 1.57079633]
```

Example 6: Sin of Angles in Degrees

```
import numpy as np

degrees = np.array([0, 30, 60, 90])

radians = np.deg2rad(degrees)

result = np.sin(radians)

print(result)
```

Output:

```
[0.       0.5      0.8660254 1.       ]
```

Example 7: Inverse Trigonometric Functions

```
import numpy as np

value = 1

angle_rad = np.arcsin(value)

angle_deg = np.rad2deg(angle_rad)

print(angle_deg)
```

Output:

90.0

Explanation:

- arcsin(1) = 90°

Example 8: Trigonometric Functions in 2D Array

```
import numpy as np

angles = np.array([[0, np.pi/2], [np.pi/4, np.pi]])

result = np.cos(angles)

print(result)
```

Output:

```
[[ 1.  0.]

 [ 0.70710678 -1.]]
```

NumPy Universal Functions – Hyperbolic Functions

What are Hyperbolic Functions?

Hyperbolic functions are **mathematical functions** similar to trigonometric functions, but they are based on **hyperbolas** instead of circles.

They are widely used in:

- Engineering

- Physics
- Machine learning
- Signal processing
- Mathematical modeling

In NumPy, hyperbolic functions are provided as **Universal Functions (ufuncs)**, which means they:

- Work element-wise on arrays
- Are fast and optimized
- Do not require loops

Common Hyperbolic Functions in NumPy

Function	**Meaning**
np.sinh()	Hyperbolic sine
np.cosh()	Hyperbolic cosine
np.tanh()	Hyperbolic tangent
np.arcsinh()	Inverse hyperbolic sine
np.arccosh()	Inverse hyperbolic cosine
np.arctanh()	Inverse hyperbolic tangent

Important:
Hyperbolic functions **do NOT use degrees or radians**.
They work directly with **real numbers**.

Mathematical Idea (Simple Understanding)

Hyperbolic functions are defined using **exponential functions**:

- $\sinh(x) = (e^x - e^{-x}) / 2$
- $\cosh(x) = (e^x + e^{-x}) / 2$
- $\tanh(x) = \sinh(x) / \cosh(x)$

(You do **not** need to memorize this for coding, but it helps conceptually.)

Example 1: Hyperbolic Sine – np.sinh()

```
import numpy as np

x = 1

result = np.sinh(x)

print(result)
```

Output:

```
1.1752011936438014
```

Explanation:

- Calculates hyperbolic sine of 1
- Result grows faster than normal sine

Example 2: Hyperbolic Cosine – np.cosh()

```
import numpy as np


x = 1

result = np.cosh(x)

print(result)
```

Output:

```
1.5430806348152437
```

Explanation:

- cosh(x) is always **positive**
- cosh(0) = 1

Example 3: Hyperbolic Tangent – np.tanh()

```
import numpy as np

x = 1

result = np.tanh(x)

print(result)
```

Output:

```
0.7615941559557649
```

Explanation:

- tanh(x) value is always between **-1 and 1**
- Very popular in **neural networks**

Example 4: Hyperbolic Functions on an Array

```
import numpy as np



values = np.array([0, 1, 2])

print(np.sinh(values))

print(np.cosh(values))

print(np.tanh(values))
```

Output:

[0. 1.17520119 3.62686041]

[1. 1.54308063 3.76219569]

[0. 0.76159416 0.96402758]

Example 5: Inverse Hyperbolic Sine – np.arcsinh()

```
import numpy as np


x = 1.5

result = np.arcsinh(x)

print(result)
```

Explanation:

- Finds the value whose sinh is x

Example 6: Inverse Hyperbolic Cosine – np.arccosh()

```
import numpy as np


x = 2

result = np.arccosh(x)

print(result)
```

Note:

- Input must be **≥ 1**

Example 7: Inverse Hyperbolic Tangent – np.arctanh()

```
import numpy as np


x = 0.5

result = np.arctanh(x)

print(result)
```

Note:

- Input must be between **-1 and 1**

Example 8: Hyperbolic Functions on 2D Array

```
import numpy as np


matrix = np.array([[0, 1], [2, 3]])

result = np.tanh(matrix)

print(result)
```

Output:

```
[[0.        0.76159416]

 [0.96402758 0.99505475]]
```

NumPy Universal Functions – Set Operations

What are Set Operations?

Set operations are used to **compare two or more arrays** and find:

- Common elements
- Unique elements
- Differences between arrays

In NumPy, **set operations treat arrays like mathematical sets**.

Important:

- Output contains **unique values**
- Result is usually **sorted**
- Mostly used with **1-D arrays**

Common NumPy Set Operation Functions

Function	Description
np.unique()	Finds unique elements
np.union1d()	Union of two arrays
np.intersect1d()	Common elements
np.setdiff1d()	Difference of arrays
np.setxor1d()	Symmetric difference
np.in1d()	Membership test

Example 1: Unique Elements – np.unique()

```
import numpy as np

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

```
result = np.unique(arr)

print(result)
```

Output:

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

Explanation:

- Removes duplicate values
- Returns sorted unique elements

Example 2: Union of Two Arrays – np.union1d()

```
import numpy as np

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

b = np.array([3, 4, 5])

result = np.union1d(a, b)

print(result)
```

Output:

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

Explanation:

- Combines both arrays
- Removes duplicates

Example 3: Intersection – np.intersect1d()

```
import numpy as np
```

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

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

result = np.intersect1d(a, b)

print(result)
```

Output:

```
[3 4]
```

Explanation:

- Finds common elements between arrays

Example 4: Difference – np.setdiff1d()

```
import numpy as np

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

b = np.array([2, 4])

result = np.setdiff1d(a, b)

print(result)
```

Output:

```
[1 3]
```

Explanation:

- Elements present in a but not in b

Example 5: Symmetric Difference – np.setxor1d()

```
import numpy as np

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

b = np.array([3, 4, 5])

result = np.setxor1d(a, b)

print(result)
```

Output:

```
[1 2 4 5]
```

Explanation:

- Elements that are in **either array**
- But **not in both**

Example 6: Membership Test – np.in1d()

```
import numpy as np

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

b = np.array([2, 4])

result = np.in1d(a, b)

print(result)
```

Output:

```
[False  True False  True]
```

Explanation:

- Checks which elements of a exist in b
- Returns Boolean values

Example 7: Set Operations with Characters

```
import numpy as np


a = np.array(['a', 'b', 'c'])

b = np.array(['b', 'c', 'd'])


print(np.union1d(a, b))

print(np.intersect1d(a, b))
```

Output:

```
['a' 'b' 'c' 'd']

['b' 'c']
```

Example 8: Set Operations in Real-Life Scenario

Student Roll Numbers

```
import numpy as np

class_A = np.array([101, 102, 103, 104])

class_B = np.array([103, 104, 105, 106])

print("All students:", np.union1d(class_A, class_B))

print("Common students:", np.intersect1d(class_A, class_B))

print("Only in Class A:", np.setdiff1d(class_A, class_B))
```

Why Use NumPy Set Operations?

Removes duplicates automatically
Faster than Python loops
Clean and readable code
Useful in data analysis