7. Universal Functions (ufuncs)

Sub-topics covered:

- `np.sqrt`
- `np.log`
- `np.exp`
- `np.abs`
- `np.round`

1. Topic Overview

What this topic is
Universal functions (ufuncs) are NumPy functions that work **element-wise** on arrays.

Why it exists
Python loops are slow.
ufuncs apply math directly on array memory using C-level speed.

One real-world analogy
A ufunc is like a machine that applies the **same operation to every item on a conveyor belt** at once.

2. Core Theory (Deep but Clear)

What is a ufunc internally

- A ufunc:
  - Takes NumPy arrays as input
  - Applies an operation element by element
  - Returns a new NumPy array
- Internally:
  - Uses contiguous memory blocks
  - Works with array **shape**

    - Respects **dtype**
    - Avoids Python loops

Example internal thinking:

```
Array shape: (5,)
Memory: continuous
dtype: float64
Operation: sqrt
Apply sqrt to each memory slot
```

Important properties of ufuncs

- Element-wise operation
- Supports broadcasting
- Very fast
- Vectorized
- Predictable output dtype

3. Syntax & Examples

3.1 `np.sqrt`

What it does

Computes square root of each element.

Basic syntax

```
np.sqrt(array)
```

Example 1

```python
import numpy as np

x = np.array([1, 4, 9, 16])
y = np.sqrt(x)
print(y)
```

Output

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

Explanation

- Each element is processed separately
- Output is float
- dtype changes from int to float

Example 2

```python
x = np.array([0, 2, 10])
print(np.sqrt(x))
```

Output

```
[0.         1.41421356 3.16227766]
```

3.2 `np.log`

What it does

Computes natural logarithm (base e).

Basic syntax

```python
np.log(array)
```

Example 1

```
x = np.array([1, np.e, np.e**2])
print(np.log(x))
```

Output

```
[0. 1. 2.]
```

Explanation

- log(e) = 1
- log(e²) = 2

Example 2

```
x = np.array([1, 10, 100])
print(np.log(x))
```

Output

```
[0.         2.30258509 4.60517019]
```

Important

- Input must be **positive**
- log(0) or log(negative) gives `-inf` or `nan`

3.3 `np.exp`

What it does

Computes exponential: e^x

Basic syntax

```
np.exp(array)
```

Example 1

```
x = np.array([0, 1, 2])
print(np.exp(x))
```

Output

```
[1.         2.71828183 7.3890561 ]
```

Explanation

- exp(0) = 1
- exp(1) = e

Example 2

```
x = np.array([-1, 0, 1])
print(np.exp(x))
```

3.4 `np.abs`

What it does

Returns absolute value.

Basic syntax

```
np.abs(array)
```

Example 1

```
x = np.array([-5, -2, 0, 3])
print(np.abs(x))
```

Output

```
[5 2 0 3]
```

Example 2 (floats)

```
x = np.array([-1.5, 2.7])
print(np.abs(x))
```

3.5 `np.round`

What it does

Rounds numbers to given decimals.

Basic syntax

```
np.round(array, decimals)
```

Example 1

```
x = np.array([1.234, 5.678])
print(np.round(x, 2))
```

Output

```
[1.23 5.68]
```

Example 2

```
x = np.array([1.5, 2.5, 3.5])
print(np.round(x))
```

Output

```
[2. 2. 4.]
```

Important

- Uses banker's rounding
- Not always round-half-up

4. Why This Matters in Data Science

Data cleaning

- `np.abs` for error magnitude
- `np.log` for skewed values
- `np.round` for precision control

Feature engineering

- `np.log` for log-transform
- `np.sqrt` for variance stabilization
- `np.exp` for reversing log features

Model input preparation

- Neural networks expect stable ranges
- Log and sqrt reduce scale problems

ML / DL pipelines

- Used in loss functions
- Used in normalization
- Used in activation math

What breaks if you don’t know this

- Slow loops
- Numerical instability
- Wrong feature scales
- Model divergence

5. Common Mistakes (VERY IMPORTANT)

1. Using Python `math` instead of NumPy
    - `math.sqrt` fails on arrays

- Always use NumPy ufuncs

2. Applying `np.log` on zero values
   - Produces `-inf`
   - Fix using clipping or masking
3. Ignoring dtype changes
   - Int input becomes float output
   - Can break downstream code
4. Looping instead of vectorizing
   - Extremely slow
   - ufuncs exist to avoid this
5. Relying on `np.round` for exact decimals
   - Floating-point is approximate
   - Never compare rounded floats directly

6. Performance & Best Practices

When ufuncs are fast

- Large arrays
- Continuous memory
- No Python loops

When they are slow

- Tiny arrays in tight loops
- Repeated unnecessary calls

Memory warnings

- ufuncs create new arrays
- Watch memory usage on large data
- Use `out=` parameter when needed

7. Practice Problems (NO SOLUTIONS)

Easy (5)

1. Compute square root of a 1D array
   Input: `[4, 9, 16]`
   Output: float array
2. Apply `np.abs` to sensor error values
3. Round feature values to 3 decimals
4. Compute log of strictly positive array
5. Apply `np.exp` to zero-centered data

Medium (7)

6. Log-transform income column with zeros
7. Reverse a log-transformed feature
8. Compute absolute residuals between y and y_pred
9. Apply sqrt to variance features only
10. Round probabilities before saving CSV
11. Identify invalid values after log
12. Combine `abs` and `round` in pipeline

Hard (5)

13. Prevent `-inf` in log feature engineering
14. Use ufuncs without extra memory allocation
15. Apply ufuncs on 2D feature matrix
16. Compare loop vs ufunc performance
17. Handle dtype issues in ML input

Industry-Level (3)

18. Preprocess skewed financial data for XGBoost

19. Stabilize loss computation using log and exp
20. Build feature pipeline using only ufuncs

8. Mini Checklist

- ufuncs are element-wise
- No Python loops
- Output dtype may change
- log needs positive input
- exp grows fast
- abs is safe
- round is not exact
- Used everywhere in ML math

7. Universal Functions (ufuncs) – MUST-KNOW EXTENSIONS

Additional ufuncs covered:

- `np.sum`
- `np.mean`
- `np.std`
- `np.min` , `np.max`
- `np.clip`
- `np.where`
- `np.isnan`
- `np.isinf`
- `np.maximum` , `np.minimum`

1. Topic Overview

What this topic is
These are core NumPy ufuncs used for **aggregation, condition checks, and numerical safety**.

Why it exists
ML models need:

- Clean numbers
- Stable ranges
- Fast vectorized logic
  These ufuncs do that without Python loops.

One real-world analogy
Like quality control machines that **scan, fix, and summarize data automatically**.

2. Core Theory (Deep but Clear)

How NumPy treats these ufuncs

- Operate on raw array memory
- Work element-wise or along axes
- Respect:
  - shape
  - dtype
  - contiguous memory
- Many support `axis` for 2D+ data

Key internal idea:

```
Array → memory block
Apply C-level loop
No Python overhead
```

3. Syntax & Examples

3.1 `np.sum`

What it does

Adds elements.

Syntax

```
np.sum(array, axis=None)
```

Example 1

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

Output

```
6
```

Example 2 (2D)

```
x = np.array([[1, 2], [3, 4]])
print(np.sum(x, axis=0))
```

Output

```
[4 6]
```

Explanation

- axis=0 → column-wise
- axis=1 → row-wise

3.2 `np.mean`

What it does

Computes average.

```
np.mean(array, axis=None)
```

```
x = np.array([2, 4, 6])
print(np.mean(x))
```

Output

```
4.0
```

Used in normalization and baselines.

3.3 `np.std`

What it does

Computes standard deviation.

```
np.std(array)
```

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

Used in:

- feature scaling
- z-score normalization

3.4 `np.min` and `np.max`

```
np.min(array)
np.max(array)
```

```
x = np.array([5, 1, 9])
print(np.min(x), np.max(x))
```

Output

```
1 9
```

Used for:

- clipping
- range checks

3.5 `np.clip`

VERY IMPORTANT

What it does

Limits values to a fixed range.

```
np.clip(array, min, max)
```

```
x = np.array([-10, 0, 5, 100])
print(np.clip(x, 0, 10))
```

Output

```
[ 0  0  5 10]
```

Used to:

- avoid overflow

- stabilize logs and loss functions

3.6 `np.where`

What it does

Vectorized if-else.

```
np.where(condition, value_if_true, value_if_false)
```

```
x = np.array([1, -2, 3])
print(np.where(x > 0, x, 0))
```

Output

```
[1 0 3]
```

Used in:

- feature rules
- masking
- label logic

3.7 `np.isnan`

What it does

Checks missing values.

```
np.isnan(array)
```

```
x = np.array([1.0, np.nan, 2.0])
print(np.isnan(x))
```

Output

```
[False  True False]
```

MANDATORY for data cleaning.

3.8 `np.isinf`

```
np.isinf(array)
```

```python
x = np.array([1, np.inf, -np.inf])
print(np.isinf(x))
```

Used after `log` , division, exp.

3.9 `np.maximum` and `np.minimum`

```
np.maximum(a, b)
np.minimum(a, b)
```

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

Output

```
[2 5 4]
```

Used in:

- ReLU logic
- thresholding
- constraints

4. Why This Matters in Data Science

Data cleaning

- `isnan`, `isinf`, `where`
- Remove invalid rows

Feature engineering

- `mean`, `std` for scaling
- `clip` for stability
- `maximum` for ReLU-style features

Model input preparation

- No NaNs allowed
- No inf allowed
- Consistent ranges

ML / DL pipelines

- Loss computation
- Gradient stability
- Batch preprocessing

If you don't know these

- Training crashes
- Silent bugs
- Bad model performance

5. Common Mistakes

1. Using Python `sum()` instead of `np.sum`
   - Slow and unsafe
2. Ignoring axis in 2D data
   - Wrong statistics

3. Forgetting to handle NaNs
    - Models fail silently
4. Using loops instead of `where`
    - Performance disaster
5. Not clipping before log/exp
    - Overflow and inf values

6. Performance & Best Practices

- Prefer ufuncs over loops
- Use `axis` correctly
- Use `clip` before `log`
- Check `isnan` and `isinf`
- Avoid repeated recomputation

7. Practice Problems (NO SOLUTIONS)

Easy (5)

1. Compute column-wise mean of a matrix
2. Detect NaNs in a feature array
3. Clip values between 0 and 1
4. Replace negatives with zero
5. Find min and max of features

Medium (7)

6. Normalize features using mean and std
7. Remove rows with NaN values
8. Apply ReLU using `maximum`
9. Stabilize log input using clip
10. Mask outliers using where
11. Detect infinite values after division

12. Compute batch-wise statistics

Hard (5)

13. Build z-score normalization without loops
14. Prevent overflow in exp
15. Combine where and isnan for cleaning
16. Handle mixed dtype arrays
17. Debug axis-related bugs

Industry-Level (3)

18. Build preprocessing step for neural network input
19. Clean financial data with NaNs and infs
20. Design feature safety checks before training

8. Mini Checklist

- sum, mean, std are mandatory
- isnan and isinf are non-negotiable
- clip prevents crashes
- where replaces loops
- axis matters
- NaNs break models
- ufuncs = speed + safety