6. Vectorized Operations

1. Topic Overview

What this topic is

Vectorized operations mean performing calculations on **entire NumPy arrays at once** without using Python loops.

Why it exists

Python loops are slow.
NumPy uses optimized C code to process arrays faster.

One real-world analogy

Instead of calculating salary for each employee one by one, you press one button and update salaries for everyone at once.

2. Core Theory (Deep but Clear)

How NumPy thinks internally

- NumPy arrays are stored in **contiguous memory**
- Operations run in **compiled C loops**, not Python loops
- Same operation applies to **every element**
- Shape and dtype must be compatible

Sub-topic 1: Element-wise Arithmetic (`+ - * /`)

What it means

Each element is operated with the corresponding element.

Internal rule

- Shapes must match OR be broadcastable
- dtype decides result precision

Example logic:

```
[1, 2, 3] + [10, 20, 30]
= [1+10, 2+20, 3+30]
```

Sub-topic 2: Power Operation (`**`)

What it means

Raise each element to a power.

Internal rule

- Implemented using fast math libraries
- Much faster than looping `pow()`

Example logic:

```
[2, 3, 4] ** 2
= [4, 9, 16]
```

Sub-topic 3: Comparison Operations

Operations:

- `>`
- `<`

- `>=`
- `<=`
- `==`
- `!=`

What it returns

- Boolean array (`True` / `False`)
- Used for masking and filtering

Internal rule

- Result dtype is `bool`
- Same shape as input

Sub-topic 4: Replacing Python Loops

Python loop

- Interpreted
- Slow
- High overhead

NumPy vectorization

- Compiled
- Fast
- Memory efficient

3. Syntax & Examples

Element-wise Addition, Subtraction, Multiplication, Division

Basic Syntax

```
result = arr1 + arr2
result = arr1 - arr2
result = arr1 * arr2
result = arr1 / arr2
```

Example 1

```
import numpy as np

a = np.array([1, 2, 3])
b = np.array([10, 20, 30])

print(a + b)
```

Output

```
[11 22 33]
```

Explanation

- 1 + 10 = 11
- 2 + 20 = 22
- 3 + 30 = 33

Example 2

```
x = np.array([10, 20, 30])
print(x * 2)
```

Output

```
[20 40 60]
```

Explanation

- Scalar `2` is applied to every element

Power Operation (**)

Basic Syntax

```
result = arr ** power
```

Example

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

Output

```
[ 8 27 64]
```

Explanation

- $2^3 = 8$
- $3^3 = 27$
- $4^3 = 64$

Comparison Operations

Basic Syntax

```
mask = arr > value
```

Example 1

```
scores = np.array([45, 60, 72, 30])
print(scores > 50)
```

Output

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

Explanation

- Each element is checked independently

Example 2

```
print(scores[scores > 50])
```

Output

```
[60 72]
```

Explanation

- Boolean mask selects valid elements

Replacing Python Loops

Python Loop (BAD)

```
result = []
for i in range(len(a)):
    result.append(a[i] * 2)
```

Vectorized (GOOD)

```
result = a * 2
```

4. Why This Matters in Data Science

Data Cleaning

- Replace invalid values
- Normalize columns
- Apply thresholds

Feature Engineering

- Scaling features
- Log or power transforms
- Creating ratios

Model Input Preparation

- Fast matrix operations
- Batch preprocessing
- GPU-friendly pipelines

ML / DL Pipelines

- Loss calculations
- Gradient updates
- Activation functions

What breaks if you don’t know this

- Your code becomes slow
- You misuse loops
- You fail interviews
- Large datasets become impossible to process

5. Common Mistakes (VERY IMPORTANT)

1. **Using Python loops**
    - Reason: Habit from basic Python

    - Fix: Always think in arrays
2. **Shape mismatch errors**
    - Reason: Ignoring array dimensions
    - Fix: Check `.shape`
3. **Integer division mistakes**
    - Reason: Integer dtype
    - Fix: Use float arrays
4. **Using `math` instead of NumPy**
    - Reason: Confusion
    - Fix: Use `np.*` functions
5. **Forgetting boolean masks**
    - Reason: Not understanding comparisons
    - Fix: Practice filtering logic

6. Performance & Best Practices

When it is fast

- Large arrays
- Simple operations
- Contiguous memory

When it is slow

- Object dtype arrays
- Very small arrays
- Repeated reshaping

Warnings

- Avoid chained operations on huge arrays
- Watch memory usage during broadcasting

7. Practice Problems (NO SOLUTIONS)

Easy (5)

1. Add two arrays of length 10.
2. Multiply all elements of an array by 5.
3. Square every element in an array.
4. Find elements greater than 100.
5. Divide an array by a scalar.

Medium (7)

6. Normalize a feature using vector operations.
7. Create a boolean mask for missing values.
8. Apply a threshold to sensor data.
9. Convert Celsius to Fahrenheit for 1M records.
10. Compute profit = revenue − cost arrays.
11. Filter salaries above company average.
12. Apply power transform to skewed data.

Hard (5)

13. Compute feature ratios safely.
14. Clip outliers using comparisons.
15. Build a custom scoring formula.
16. Vectorize a loss function.
17. Replace nested loops using NumPy ops.

Industry-Level Tasks (3)

18. Preprocess 10M rows without loops.
19. Apply batch feature scaling for ML input.
20. Create boolean masks for anomaly detection.

8. Mini Checklist

- Vectorized ops replace loops
- Operations are element-wise
- Comparisons return boolean arrays
- Faster due to C-level execution
- Shape and dtype always matter

Yes.
What you listed is **not complete**.

If you stop at `+ - * / ** comparisons` , you will be **under-prepared** for real Data Science work and interviews.

Below are the **other vectorized operations that are mandatory**.
Same structure. Same strict style. Clean notes.

You can add this as a **continuation of Topic 6** in your repo.

6. Vectorized Operations (MANDATORY EXTENSIONS)

1. Topic Overview

What this topic is

These are **additional vectorized operations** that work on full NumPy arrays without loops.

Why it exists

Real Data Science needs:

- Conditions
- Cleaning
- Safe replacements
- Mathematical transforms

Loops fail at scale.

One real-world analogy

You apply rules to an entire Excel column using formulas, not cell by cell typing.

2. Core Theory (Deep but Clear)

NumPy supports **universal functions (ufuncs)**.

They:

- Work element-wise
- Run in C
- Preserve shape
- Respect dtype

These operations **combine math + logic + safety**.

Sub-topic 1: Logical Operations

Operators:

- `&` (AND)
- `|` (OR)
- `~` (NOT)

Internal rule

- Works on boolean arrays
- Uses bitwise logic
- Shapes must match

Never use `and` / `or` with NumPy arrays.

Sub-topic 2: Mathematical Unary Functions (ufuncs)

Common ones:

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

Internal rule

- Applied element-wise
- Vectorized C loops
- Faster than Python `math`

Used heavily in ML math.

Sub-topic 3: `np.where()` (Conditional Vectorization)

What it does

Choose values based on condition.

Internal logic

```
if condition:
    use A
else:
    use B
```

But applied to **entire arrays at once**.

Sub-topic 4: Clipping Values (`np.clip`)

What it does

Force values into a safe range.

Used for:

- Outliers
- Numerical stability

Sub-topic 5: Handling Invalid Values

Functions:

- `np.isnan`
- `np.isinf`
- `np.isfinite`

These are **vectorized checks**.

3. Syntax & Examples

Logical Operations

Syntax

```
mask = (arr > 0) & (arr < 100)
```

Example

```
import numpy as np

a = np.array([10, -5, 200, 50])
mask = (a > 0) & (a < 100)
print(mask)
```

Output

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

Explanation

- Only values between 0 and 100 pass

Unary Math Functions

Syntax

```
np.log(arr)
np.sqrt(arr)
```

Example

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

Output

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

Explanation

- Natural log applied to every element

`np.where()`

Syntax

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

Example

```
scores = np.array([35, 60, 80])
result = np.where(scores >= 50, "Pass", "Fail")
print(result)
```

Output

```
['Fail' 'Pass' 'Pass']
```

Explanation

- Condition checked element-wise
- Result array keeps same shape

`np.clip()`

Syntax

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

Example

```
values = np.array([1, 5, 100, 200])
print(np.clip(values, 0, 100))
```

Output

```
[  1   5 100 100]
```

Explanation

- Values above 100 are forced to 100

Invalid Value Checks

Example

```
x = np.array([1, np.nan, np.inf, 5])
print(np.isnan(x))
```

Output

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

4. Why This Matters in Data Science

Data Cleaning

- Remove NaNs
- Clip outliers
- Replace invalid values

Feature Engineering

- Log transforms
- Absolute scaling
- Conditional features

Model Input Preparation

- Numerical stability
- Valid ranges
- Masked arrays

ML / DL Pipelines

- Loss functions
- Activation functions
- Gradient safety

What breaks if you skip this

- NaNs destroy models
- Training becomes unstable
- Results become meaningless

5. Common Mistakes (VERY IMPORTANT)

1. **Using `and/or` instead of `&` `|`**
   - Causes ValueError
   - Always use bitwise ops
2. **Forgetting parentheses in conditions**
   - Operator precedence breaks logic
   - Always wrap comparisons
3. **Applying log to zero or negative**
   - Produces NaN
   - Clip or shift first
4. **Ignoring NaNs**
   - Models silently fail
   - Always check with `isnan`
5. **Using loops for conditionals**
   - Very slow
   - Replace with `np.where`

6. Performance & Best Practices

Fast

- Large numeric arrays

- Float32 / Float64
- Contiguous memory

Slow

- Object dtype
- Python strings in arrays

Warnings

- `np.where` creates new arrays
- Clipping large arrays increases memory usage

7. Practice Problems (NO SOLUTIONS)

Easy (5)

1. Find values between 10 and 50.
2. Apply square root to positive values.
3. Replace negatives with zero.
4. Detect NaN values.
5. Clip values between 0 and 1.

Medium (7)

6. Apply log transform safely.
7. Create pass/fail labels.
8. Remove infinite values.
9. Normalize after clipping.
10. Build boolean mask for outliers.
11. Replace missing values conditionally.
12. Vectorize threshold-based scoring.

Hard (5)

13. Stabilize loss function inputs.
14. Create custom feature using conditions.
15. Clean sensor data using masks.
16. Prevent overflow using clip.
17. Chain vectorized conditions correctly.

Industry-Level Tasks (3)

18. Prepare ML-ready numeric features.
19. Detect anomalies using logical masks.
20. Build preprocessing pipeline without loops.

8. Mini Checklist

- Logical ops use `& | ~`
- Math ufuncs are vectorized
- `np.where` replaces if-else
- `np.clip` controls outliers
- Invalid values must be handled

Brutal truth (important)

If you **do not master all of this**, you are **not using NumPy properly**.
Loops + ignorance of masks = weak Data Scientist.