17. Performance Concepts (NumPy)

1. Topic Overview

What this topic is

Performance concepts explain **why NumPy is fast** and **how to keep it fast**.

Why it exists

NumPy handles large data.
Wrong usage makes it slow or memory-heavy.
Correct usage makes it near C-level speed.

One real-world analogy

Think of a factory line.
Vectorized machines are fast.
Manual hand work (loops) is slow.

2. Core Theory (Deep but Clear)

NumPy performance depends on **how arrays are stored and processed**.

We will cover these sub-topics:

1. Vectorization vs Python loops
2. Memory layout (C vs F order)
3. Views vs copies
4. Broadcasting cost
5. Dtype impact
6. Contiguous arrays
7. Temporary arrays

2.1 Vectorization vs Python Loops

Vectorization

- Operations run in compiled C code.
- Loop happens internally.
- Python loop overhead is avoided.

Python loop

- Loop runs in Python.
- Each iteration is slow.

NumPy thinks in:

- Whole arrays
- Not element by element

2.2 Memory Layout (C vs F Order)

Arrays are stored in memory linearly.

- **C-order**: Row-major (default)
- **F-order**: Column-major

Accessing data in memory order is faster.

NumPy cares about:

- Shape
- Strides
- Contiguity

2.3 Views vs Copies

View

- Shares same memory.

- Fast.
- Changes affect original array.

Copy

- New memory allocation.
- Slower.
- Safe but costly.

Many NumPy operations return views silently.

2.4 Broadcasting Cost

Broadcasting avoids data duplication.
But:

- Large broadcasts create temporary arrays.
- Memory usage can explode.

Shape compatibility matters.

2.5 Dtype Impact

Smaller dtype:

- Less memory
- Better cache usage
- Faster operations

Larger dtype:

- More precision
- Slower

NumPy never auto-optimizes dtype for speed.

2.6 Contiguous Arrays

Contiguous = data stored sequentially.

Non-contiguous arrays:

- Created by slicing or transpose
- Slower for math operations

Some NumPy functions silently make copies.

2.7 Temporary Arrays

Expressions like:

```
a = b * c + d
```

Create:

- One temporary for `b * c`
- Then add `d`

More temporaries = more memory + slower.

3. Syntax & Examples

3.1 Vectorization

Syntax

```
array + 10
```

Example 1

```
import numpy as np

a = np.array([1, 2, 3])
b = a + 10
print(b)
```

Output

```
[11 12 13]
```

Explanation

- NumPy loops internally
- No Python loop

Example 2 (Slow way)

```
result = []
for x in a:
    result.append(x + 10)

print(result)
```

Explanation

- Python loop
- Much slower for large arrays

3.2 Memory Order

Syntax

```
np.array(data, order='C')
np.array(data, order='F')
```

Example

```
a = np.array([[1,2],[3,4]], order='C')
print(a.flags['C_CONTIGUOUS'])
print(a.flags['F_CONTIGUOUS'])
```

Output

```
True
False
```

Explanation

- Stored row-wise
- Fast row access

3.3 Views vs Copies

Example

```
a = np.array([1,2,3,4])
b = a[1:3]
b[0] = 100
print(a)
```

Output

```
[  1 100   3   4]
```

Explanation

- Slice returns view
- Same memory

Copy

```
c = a[1:3].copy()
```

3.4 Broadcasting Cost

```
a = np.ones((1000, 1000))
b = np.ones((1000,))
c = a + b
```

Explanation

- `b` is broadcast
- Temporary array created internally

3.5 Dtype Impact

```
a = np.ones(1_000_000, dtype=np.int64)
b = np.ones(1_000_000, dtype=np.float64)
```

Explanation

- `float64` uses more memory
- Slower arithmetic

3.6 Contiguity Check

```
a = np.arange(10)
b = a[::2]
print(b.flags['C_CONTIGUOUS'])
```

Output

```
False
```

Explanation

- Strided slice
- Non-contiguous

3.7 Temporary Arrays

```
a = np.random.rand(1000)
b = np.random.rand(1000)
c = np.random.rand(1000)

result = a * b + c
```

Explanation

- `a * b` creates temp
- Then `+ c`

4. Why This Matters in Data Science

Data Cleaning

- Vectorized masks are fast
- Python loops freeze pipelines

Feature Engineering

- Broadcasting used everywhere
- Wrong shape causes memory blowup

Model Input Preparation

- Contiguous arrays needed for ML libs

- Wrong dtype wastes RAM

ML / DL Pipelines

- NumPy feeds TensorFlow, PyTorch
- Non-contiguous arrays trigger hidden copies

What breaks if you ignore this

- Training becomes slow
- Memory errors
- Unstable pipelines

5. Common Mistakes (VERY IMPORTANT)

1. Using Python loops over arrays
   Reason: Python overhead
   Fix: Always vectorize
2. Ignoring views vs copies
   Reason: Hidden memory sharing
   Fix: Use `.copy()` when needed
3. Broadcasting huge arrays blindly
   Reason: Temporary memory explosion
   Fix: Check shapes first
4. Using default `float64` everywhere
   Reason: Wasted memory
   Fix: Use `float32` when possible
5. Operating on non-contiguous arrays
   Reason: Cache misses
   Fix: Use `np.ascontiguousarray()`

6. Performance & Best Practices

Fast When

- Vectorized ops
- Contiguous memory
- Small dtype
- Fewer temporaries

Slow When

- Python loops
- Repeated slicing
- Large broadcasting
- Implicit copies

Warnings

- Transpose often creates non-contiguous arrays
- Chained expressions create temporaries
- Always inspect `.flags` and `.dtype`

7. 20 Practice Problems (MANDATORY)

Easy (5)

1. Check if an array is contiguous.
2. Convert a non-contiguous array to contiguous.
3. Compare memory size of `float32` vs `float64`.
4. Replace Python loop with vectorized code.
5. Detect if slicing returns a view.

Medium (7)

6. Optimize a slow feature scaling step.

7. Remove unnecessary temporary arrays.
8. Fix a broadcasting memory issue.
9. Convert model input to optimal dtype.
10. Identify hidden copies in slicing.
11. Speed up column-wise operations.
12. Prepare NumPy output for PyTorch.

Hard (5)

13. Optimize chained NumPy expressions.
14. Debug memory leak due to views.
15. Speed up large matrix preprocessing.
16. Convert non-contiguous batch data.
17. Reduce RAM usage in pipeline.

Industry-Level Tasks (3)

18. Optimize NumPy preprocessing for 10M rows.
19. Fix slow inference caused by dtype mismatch.
20. Audit NumPy code for hidden performance bugs.

8. Mini Checklist

- Avoid Python loops
- Know when you get a view
- Control dtype explicitly
- Watch broadcasting shapes
- Minimize temporary arrays
- Prefer contiguous memory
- Inspect `.flags`, `.dtype`, `.shape`

17. Performance Concepts (Advanced & Mandatory Add-ons)

1. Topic Overview

What this topic is

These are **advanced performance controls** in NumPy.
They decide **how far you can scale** your data pipelines.

Why it exists

At scale:

- Code that is “fast enough” becomes slow
- Memory becomes the real bottleneck
- Small inefficiencies crash jobs

One real-world analogy

Driving fast is not enough.
You must also know:

- Fuel usage
- Engine heat
- Gear efficiency

2. Core Theory (Deep but Clear)

This section covers **new performance concepts** only:

1. In-place operations
2. Memory alignment
3. Cache friendliness

4. Chunking large arrays
5. NumPy vs BLAS backend
6. Lazy computation limits
7. GIL and NumPy

2.1 In-place Operations

In-place operation

- Modifies existing array
- No new memory allocation

Example:

```
a += 1
```

Not in-place:

```
a = a + 1
```

NumPy internally:

- Reuses same memory buffer
- Avoids allocation and copy

2.2 Memory Alignment

Aligned memory:

- Matches CPU word boundaries
- Faster SIMD operations

Misaligned arrays:

- Slower math
- Hidden copies

NumPy tracks:

- Alignment at allocation time
- Not visible unless inspected

2.3 Cache Friendliness

CPU cache is small and fast.

If data access:

- Is sequential → fast
- Is random → slow

NumPy is fast **only when memory access is predictable**.

Row-wise operations are faster than column-wise (C-order).

2.4 Chunking Large Arrays

Large arrays do not fit in cache or RAM.

Chunking:

- Breaks data into blocks
- Processes block by block

This is manual in NumPy.

NumPy will NOT auto-chunk for you.

2.5 NumPy vs BLAS Backend

Linear algebra uses BLAS libraries.

Examples:

- OpenBLAS

- MKL

Performance depends on:

- Which BLAS NumPy is linked to
- Number of threads
- CPU architecture

Same code can be **2x–10x faster** depending on BLAS.

2.6 Lazy Computation Limits

NumPy is **eager**, not lazy.

This means:

- Every operation runs immediately
- Memory allocated immediately

Unlike:

- Dask
- JAX

You must manually control execution order.

2.7 GIL and NumPy

Python has GIL.

NumPy:

- Releases GIL in many operations
- Uses multi-threaded C code internally

But:

- Only for large operations
- Small arrays stay single-threaded

3. Syntax & Examples

3.1 In-place Operations

Syntax

```
a += b
```

Example

```
import numpy as np

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

Output

```
[6 7 8]
```

Explanation

- Same memory
- No temporary array

3.2 Memory Alignment

```
a = np.ones(10)
print(a.flags['ALIGNED'])
```

Output

```
True
```

Explanation

- Proper CPU alignment
- Faster math ops

3.3 Cache-Friendly Access

```
a = np.random.rand(10000, 10000)
row_sum = a.sum(axis=1)
```

Explanation

- Sequential memory access
- Faster than column-wise sum

3.4 Chunking

```
for i in range(0, len(a), 1000):
    chunk = a[i:i+1000]
```

Explanation

- Manual chunking
- Prevents memory spikes

3.5 BLAS Backend Effect

```
import numpy as np
np.show_config()
```

Explanation

- Shows BLAS library
- Critical for matrix ops

3.6 Eager Execution

```
a = np.ones(1_000_000)
b = a * 2
```

Explanation

- Executes immediately
- Allocates memory now

3.7 GIL Release

```
a = np.random.rand(1_000_000)
b = np.random.rand(1_000_000)
c = a @ b
```

Explanation

- Uses native threads
- GIL released internally

4. Why This Matters in Data Science

Data Cleaning

- In-place ops reduce RAM
- Chunking avoids crashes

Feature Engineering

- Cache-friendly layout speeds transforms

- BLAS accelerates matrix ops

Model Input Preparation

- Alignment improves tensor conversion speed
- Avoids hidden copies

ML / DL Pipelines

- BLAS controls training speed
- GIL behavior affects parallelism

What breaks if you ignore this

- Out-of-memory errors
- Slow matrix math
- Unscalable pipelines

5. Common Mistakes (VERY IMPORTANT)

1. Using `a = a + b` instead of `a += b`
   Reason: Creates copy
   Fix: Prefer in-place
2. Ignoring cache access patterns
   Reason: CPU cache misses
   Fix: Operate row-wise
3. Processing huge arrays at once
   Reason: RAM exhaustion
   Fix: Chunk manually
4. Assuming NumPy is lazy
   Reason: Immediate execution
   Fix: Control expression order
5. Ignoring BLAS backend
   Reason: Huge speed differences
   Fix: Always check config

6. Performance & Best Practices

Fast When

- In-place ops
- Cache-friendly access
- Optimized BLAS
- Chunked processing

Slow When

- Repeated full-array passes
- Misaligned memory
- Huge eager allocations

Warnings

- In-place ops can overwrite data
- Chunk size must match cache/RAM
- BLAS threads can oversubscribe CPU

7. 20 Practice Problems (MANDATORY)

Easy (5)

1. Convert out-of-place ops to in-place.
2. Check memory alignment of arrays.
3. Identify eager execution points.
4. Inspect BLAS backend.
5. Measure row vs column access speed.

Medium (7)

6. Optimize feature scaling with in-place ops.
7. Reduce RAM in preprocessing step.

8. Chunk a large CSV-loaded array.
9. Fix slow matrix multiplication.
10. Improve cache locality in loop.
11. Remove redundant full-array passes.
12. Control BLAS threading.

Hard (5)

13. Optimize NumPy code for limited RAM.
14. Debug performance drop due to misalignment.
15. Rewrite pipeline to avoid eager memory spikes.
16. Improve inference speed using BLAS.
17. Handle multi-threaded NumPy safely.

Industry-Level Tasks (3)

18. Optimize NumPy ETL for 50M rows.
19. Fix production job crashing due to memory.
20. Audit NumPy pipeline for CPU cache efficiency.

8. Mini Checklist

- Prefer in-place ops
- Watch memory alignment
- Respect CPU cache
- Chunk large data
- Know your BLAS
- NumPy is eager
- GIL is partly released

This **completes NumPy performance mastery**.