Session 4: NumPy for Data Work

Arrays, Vectorisation, Masks, and Aggregations

LEARNING OBJECTIVES

What You'll Master Today

By the end of this session, you will possess the foundational skills to work confidently with NumPy in real-world data science projects. These objectives represent the core competencies that separate efficient, professional data work from inefficient approaches.

Understanding ndarrays

Explain what an ndarray is and how to interpret its fundamental properties: shape, dtype, and dimensionality. You'll understand why these attributes matter for data integrity and performance.

Vectorised Operations

Create arrays and perform vectorised computations without writing Python loops for mathematical operations. This is the key to NumPy's performance advantage.

Boolean Masking

Use boolean masks and advanced indexing techniques to filter and select data based on conditions—a fundamental skill for data cleaning and analysis.

Statistical Aggregation

Compute statistics using the axis parameter and handle missing values (NaN) appropriately, ensuring robust calculations even with incomplete data.

Data Normalisation

Standardise and normalise numeric data using NumPy operations, preparing datasets for machine learning algorithms that require scaled features.

Why NumPy in Data Projects

Understanding why NumPy exists and when to use it is crucial for making informed architectural decisions in data science projects. The performance difference isn't marginal—it's often orders of magnitude.

Python Lists: Flexible but Slow

Python lists are incredibly versatile data structures that can hold mixed types and grow dynamically. However, this flexibility comes at a steep cost for numeric workloads. Each element is a full Python object with overhead, and operations require Python's interpreter loop, making numerical computations prohibitively slow for large datasets.

NumPy: Contiguous Memory Storage

NumPy stores data in contiguous blocks of memory with a fixed data type, enabling operations to be executed in optimised C code. This architecture allows the processor to work efficiently with cache lines and enables SIMD (Single Instruction, Multiple Data) operations, resulting in dramatic performance improvements for array operations.

Performance Reality: For a typical numerical operation on 1 million elements, NumPy can be 100-200x faster than pure Python loops. This difference compounds in real workflows.

Pandas: Built on NumPy

Pandas DataFrames are essentially collections of NumPy arrays with added structure and functionality. Understanding NumPy fundamentals transfers directly to Pandas work—every column in a DataFrame is backed by a NumPy array. Mastering NumPy makes you more effective with Pandas and helps you understand its performance characteristics.

The Core Object: ndarray

The ndarray (n-dimensional array) is NumPy's fundamental data structure. Unlike Python lists, ndarrays are homogeneous containers—every element must be the same data type. This constraint is precisely what enables NumPy's performance advantages.

Key Attributes

- **shape**: A tuple describing the dimensions of the array (e.g., (3, 4) for 3 rows and 4 columns)
- **dtype**: The data type of elements (e.g., float64, int32, bool). This determines memory usage and precision
- **ndim**: The number of dimensions (1 for vectors, 2 for matrices, etc.)
- **size**: Total number of elements in the array
- **itemsize**: Size in bytes of each element

Creating and Inspecting

```
import numpy as np

x = np.array([1, 2, 3], dtype=float)
print(x.shape)   # (3,)
print(x.dtype)   # float64
print(x.ndim)    # 1
print(x.size)    # 3
print(x.itemsize) # 8 bytes
```

Understanding these attributes is essential for debugging shape mismatches, managing memory efficiently, and ensuring numerical precision in your calculations.

Creating Arrays: Common Patterns

NumPy provides multiple factory functions for array creation, each suited to different use cases. Choosing the right constructor improves code clarity and performance.

np.array()

Creates an array from an existing sequence (list, tuple). Use when you have data already in Python structures. The dtype can be inferred or explicitly specified.

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

np.zeros() / np.ones()

Allocates arrays filled with zeros or ones. Ideal for initialising arrays before filling them with computed values, or creating masks and weight matrices.

```
zeros = np.zeros((3, 4))  # 3x4 matrix of 0.0
ones = np.ones((2, 2), dtype=int)
```

np.arange()

Creates evenly spaced values within a specified interval, similar to Python's range() but returns an array. Perfect for generating index sequences or discrete ranges.

```
b = np.arange(0, 10, 2)  # [0, 2, 4, 6, 8]
c = np.arange(5)         # [0, 1, 2, 3, 4]
```

np.linspace()

Returns evenly spaced numbers over a specified interval, where you specify the number of points rather than the step size. Essential for creating smooth ranges for plotting or numerical methods.

```
c = np.linspace(0, 1, 5)  # [0.0, 0.25, 0.5, 0.75, 1.0]
d = np.linspace(0, 10, 100)  # 100 points from 0 to 10
```

np.eye() / np.identity()

Creates identity matrices (ones on the diagonal, zeros elsewhere). Critical for linear algebra operations and matrix initialisation.

```
I = np.eye(3)  # 3x3 identity matrix
# [[1, 0, 0],
#  [0, 1, 0],
#  [0, 0, 1]]
```

np.random functions

Generate arrays with random values from various distributions. Essential for simulations, initialising neural network weights, and statistical sampling.

```
rand = np.random.random((2, 3)) # Uniform [0, 1)
norm = np.random.randn(1000) # Standard normal
ints = np.random.randint(0, 10, size=5)
```

Vectorisation: The Big Idea

Vectorisation is the cornerstone of efficient NumPy code. It means expressing operations on entire arrays at once, rather than looping through individual elements. This paradigm shift is fundamental to writing performant data science code.

How Vectorisation Works

When you write vectorised code, NumPy operations are executed in compiled C code with optimised loops, rather than in Python's interpreter. The operation is applied element-wise across the entire array in a single pass, often using processor-level optimisations like SIMD instructions that can process multiple elements simultaneously.

The Performance Difference

Consider multiplying every element in an array by 2 and adding 1. A Python loop touches each element individually through the interpreter. A vectorised operation hands the entire array to C code, which processes it orders of magnitude faster.

Python Loop (Slow)

```
x = [1, 2, 3, 4, 5]
y = []
for element in x:
    y.append(element * 2 + 1)
# Result: [3, 5, 7, 9, 11]
```

Vectorised (Fast)

```
x = np.array([1, 2, 3, 4, 5])
y = x * 2 + 1
# Result: [3 5 7 9 11]
```

Golden Rule: If you're writing a Python for loop to process numeric data element-by-element, there's almost always a faster vectorised alternative in NumPy.

Broadcasting: Shapes That "Stretch"

Broadcasting is NumPy's powerful mechanism for performing operations on arrays of different shapes. Understanding broadcasting rules is essential for writing concise, efficient code and avoiding cryptic shape errors.

Broadcasting Rules

NumPy automatically "broadcasts" smaller arrays across larger ones when their shapes are compatible. Two dimensions are compatible when:

1. They are equal, or
2. One of them is 1

Broadcasting starts from the trailing (rightmost) dimension and works backward. Missing dimensions are assumed to be 1.

Common Use Case: Centring Data

A typical data science operation is subtracting the per-column mean from a 2D matrix. The mean has shape (n_features,) but X has shape (n_samples, n_features). Broadcasting automatically aligns them.

Practical Example

```
X = np.array([[1, 2],
              [3, 4],
              [5, 6]], dtype=float)
print(X.shape)  # (3, 2)

mu = X.mean(axis=0)  # shape (2,)
print(mu)  # [3. 4.]

X_centered = X - mu  # broadcast mu across rows
print(X_centered)
# [[-2. -2.]
#  [ 0.  0.]
#  [ 2.  2.]]
```

Here, the (2,) array mu is conceptually stretched to (3, 2) by replicating its values across rows, though no actual copying occurs in memory.

Indexing and Slicing: 1D and 2D

NumPy's indexing system is both powerful and intuitive once you understand its patterns. Mastering indexing is crucial for data manipulation and feature engineering.

1D Indexing

Similar to Python lists: use integers for single elements, slices for ranges. Remember that indexing is zero-based and slices are half-open [start:stop).

2D Indexing

Use [row, col] notation. A single colon : selects all elements along that dimension. This syntax is more powerful and clear than nested bracket notation.

Returning Views vs Copies

Basic slicing returns views (no data copying), while fancy indexing returns copies. Understanding this distinction is crucial for memory efficiency and avoiding subtle bugs.

1D Array Examples

```
x = np.array([10, 20, 30, 40, 50])

print(x[0])     # 10 (first element)
print(x[-1])    # 50 (last element)
print(x[1:4])   # [20 30 40] (elements 1-3)
print(x[::2])   # [10 30 50] (every 2nd element)
print(x[:3])    # [10 20 30] (first 3)
print(x[2:])    # [30 40 50] (from index 2 onwards)
```

2D Array Examples

```
X = np.array([[10, 20, 30],
 [40, 50, 60]])

print(X[0, 0]) # 10 (element at row 0, col 0)
print(X[1, 2]) # 60 (element at row 1, col 2)
print(X[:, 0]) # [10 40] (entire first column)
print(X[0, :]) # [10 20 30] (entire first row)
print(X[1, :]) # [40 50 60] (entire second row)
print(X[:, 1:]) # columns 1 onwards
# [[20 30]
# [50 60]]
```

Boolean Masks: Filtering Data

Boolean masking is one of NumPy's most elegant and powerful features. It allows you to filter arrays based on conditions without writing explicit loops, making data cleaning and selection both intuitive and efficient.

The Two-Step Pattern

1. **Create a boolean mask**: Apply a condition to your array, which returns a boolean array of the same shape with True where the condition is met and False elsewhere.
2. **Apply the mask**: Use the boolean array as an index to select only the elements where the mask is True.

Building Complex Conditions

You can combine multiple conditions using bitwise operators:

- **&** for AND (both conditions must be True)
- **|** for OR (either condition can be True)
- **~** for NOT (inverts True/False)
- Use parentheses around each condition to avoid operator precedence issues

Basic Masking Example

```
x = np.array([10, 5, 8, 20, 15, 3])

# Step 1: Create mask
mask = x > 9
print(mask)
# [True False False True True False]

# Step 2: Apply mask
print(x[mask])  # [10 20 15]
```

Complex Conditions

```
x = np.array([10, 5, 8, 20, 15, 3])

# Values between 5 and 15 (inclusive)
mask = (x >= 5) & (x <= 15)
print(x[mask])  # [10 5 8 15]

# Values outside range 8-12
mask = (x < 8) | (x > 12)
print(x[mask])  # [5 3 20 15]
```

2D Masking

```
X = np.array([[1, 2, 3],
 [4, 5, 6]])
mask = X > 3
print(X[mask]) # [4 5 6] (flattened)
```

Reductions and the axis Parameter

Reduction operations aggregate data along specified dimensions. Understanding the axis parameter is essential for computing statistics correctly in multi-dimensional data, especially when working with datasets where rows represent samples and columns represent features.

What Are Reductions?

Reduction operations take an array and produce a result with fewer dimensions by aggregating values. Common reductions include sum, mean, min, max, std (standard deviation), and var (variance). These operations can be applied to the entire array or along specific axes.

Understanding axis=0

axis=0 aggregates down rows, computing one result per column. Think of it as "collapsing the rows" or "computing per feature". For a (100, 5) dataset, axis=0 operations return shape (5,)—one value per column. This is commonly used to compute feature-wise statistics.

Understanding axis=1

axis=1 aggregates across columns, computing one result per row. Think of it as "collapsing the columns" or "computing per sample". For a (100, 5) dataset, axis=1 operations return shape (100,)—one value per row. Use this for sample-wise operations.

Practical Example

```
X = np.array([[1, 2, 3],
              [10, 20, 30]])
print(X.shape)  # (2, 3)

# axis=0: per-column stats (down rows)
print(X.sum(axis=0))   # [11 22 33]
print(X.mean(axis=0))  # [5.5 11. 16.5]

# axis=1: per-row stats (across columns)
print(X.sum(axis=1))   # [6 60]
print(X.mean(axis=1))  # [2. 20.]

# No axis: aggregate everything
print(X.sum())   # 66
print(X.mean())  # 11.0
```

Common Reduction Functions

- **np.sum() / .sum()**: Total of all values
- **np.mean() / .mean()**: Arithmetic average
- **np.std() / .std()**: Standard deviation
- **np.var() / .var()**: Variance
- **np.min() / .min()**: Minimum value
- **np.max() / .max()**: Maximum value
- **np.median()**: Median value (no method form)
- **np.argmin() / .argmin()**: Index of minimum
- **np.argmax() / .argmax()**: Index of maximum

Memory Tip: axis=0 is "down" (imagine gravity pulling values down to collapse rows). axis=1 is "across" (left to right).

Handling Missing Values with NaN

Missing data is ubiquitous in real-world datasets. NumPy represents missing numeric values using NaN (Not a Number), a special floating-point value defined by IEEE 754 standard. Understanding how NaN propagates and how to work around it is critical for robust data analysis.

NaN Propagation Problem

The challenge with NaN is that it "infects" calculations: any operation involving NaN typically returns NaN. This means a single missing value in a dataset can render entire statistical summaries useless if not handled properly.

```
x = np.array([1.0, np.nan, 3.0, 4.0])

print(np.mean(x))   # nan
print(np.sum(x))    # nan
print(np.std(x))    # nan
print(np.max(x))    # nan
```

Even one NaN value causes all standard aggregation functions to return NaN, which is mathematically correct but often not what we want for data analysis.

Robust Statistics: np.nan* Functions

NumPy provides NaN-aware versions of all major statistical functions that simply ignore missing values in their calculations:

```
x = np.array([1.0, np.nan, 3.0, 4.0])

print(np.nanmean(x))   # 2.666... (mean of 1,3,4)
print(np.nansum(x))    # 8.0
print(np.nanstd(x))    # 1.247...
print(np.nanmax(x))    # 4.0
print(np.nanmin(x))    # 1.0
print(np.nanmedian(x)) # 3.0
```

Available NaN-Safe Functions

- **np.nanmean()**: Mean ignoring NaN
- **np.nanstd()**: Standard deviation ignoring NaN
- **np.nanvar()**: Variance ignoring NaN
- **np.nansum()**: Sum ignoring NaN
- **np.nanmin() / np.nanmax()**: Min/max ignoring NaN
- **np.nanmedian()**: Median ignoring NaN

These functions are essential tools in your data cleaning arsenal and should be your default choice when working with real-world data.

Cleaning and Replacing Values

Data cleaning often requires detecting and replacing problematic values. NumPy provides elegant tools for conditional replacement without explicit loops, keeping your code vectorised and efficient.

Detect Missing Values

Use np.isnan() to create a boolean mask identifying NaN values. This returns a boolean array of the same shape as the input.

```
x = np.array([1.0, np.nan, 3.0, np.nan])
mask = np.isnan(x)
print(mask)
# [False True False True]
```

Conditional Replacement

Use np.where(condition, value_if_true, value_if_false) for elegant conditional replacement. This is vectorised and far more efficient than loops.

```
x = np.array([1.0, np.nan, 3.0, np.nan])
x2 = np.where(np.isnan(x), 0.0, x)
print(x2)
# [1. 0. 3. 0.]
```

In-Place Modification

For memory efficiency with large arrays, modify values in-place using boolean indexing rather than creating new arrays.

```
x = np.array([1.0, np.nan, 3.0, np.nan])
x[np.isnan(x)] = 0.0
print(x)
# [1. 0. 3. 0.]
```

Replacing Outliers

```
# Cap values above threshold
data = np.array([1, 2, 100, 4, 5, 200])
threshold = 50
data_capped = np.where(data > threshold,
                       threshold,
                       data)
print(data_capped)  # [1 2 50 4 5 50]
```

Multiple Conditions

```
# Complex replacement logic
data = np.array([-5, -1, 0, 1, 5, 10])

# Replace negative with 0, >5 with 5
result = np.where(data < 0, 0,
         np.where(data > 5, 5, data))
print(result)  # [0 0 0 1 5 5]
```

Replacing with Mean

```
# Replace NaN with column mean
X = np.array([[1.0, 2.0],
              [np.nan, 4.0],
              [5.0, np.nan]])
col_means = np.nanmean(X, axis=0)
for col in range(X.shape[1]):
    mask = np.isnan(X[:, col])
    X[mask, col] = col_means[col]
```

Detecting and Counting

```
x = np.array([1.0, np.nan, 3.0, np.nan, 5.0])

# Count missing values
n_missing = np.isnan(x).sum()
print(f"Missing: {n_missing}") # Missing: 2

# Percentage missing
pct_missing = np.isnan(x).mean() * 100
print(f"{pct_missing}% missing") # 40.0%
```

Standardisation: Z-Score Transformation

Standardisation (also called z-score normalisation) transforms features to have mean 0 and standard deviation 1. This is crucial preprocessing for many machine learning algorithms that are sensitive to feature scales, such as gradient descent-based methods, distance-based algorithms like k-NN, and regularised models.

Why Standardise?

- **Scale invariance**: Features with larger scales won't dominate the model
- **Faster convergence**: Gradient descent converges more quickly
- **Interpretability**: Coefficients become comparable across features
- **Numerical stability**: Prevents overflow/underflow in calculations

The Formula

For each feature (column), compute:

$$z = \frac{x - \mu}{\sigma}$$

Where μ is the mean and σ is the standard deviation. The result has mean 0 and standard deviation 1.

Handling Missing Values

When your dataset contains NaN values, use nanmean() and nanstd() instead of their standard counterparts. This ensures your statistics are computed on valid data only.

Implementation with NaN Handling

```
X = np.array([[1.0, 2.0],
              [3.0, np.nan],
              [5.0, 6.0],
              [np.nan, 8.0]])

# Compute per-column mean and std
mu = np.nanmean(X, axis=0)
sigma = np.nanstd(X, axis=0)

print(f"Means: {mu}")     # [3. 5.33...]
print(f"Stds: {sigma}")   # [1.63... 2.49...]

# Prevent division by zero for constant columns
sigma_safe = np.where(sigma == 0, 1.0, sigma)

# Apply standardisation
Z = (X - mu) / sigma_safe
print(Z)
# [[-1.22  -1.33]
#  [ 0.    nan  ]
#  [ 1.22   0.27 ]
#  [ nan    1.07 ]]
```

Critical Detail: Always handle zero standard deviation (constant features) by replacing with 1.0 to avoid division by zero errors. These features provide no information variance anyway.

Outlier Detection with Z-Scores

Once data is standardised, z-scores provide an intuitive measure of how extreme a value is relative to the distribution. This forms the basis of a simple but effective outlier detection method.

The Three-Sigma Rule

A common statistical rule of thumb states that in a normal distribution:

- ≈68% of values fall within 1 standard deviation ($|z| \leq 1$)
- ≈95% of values fall within 2 standard deviations ($|z| \leq 2$)
- ≈99.7% of values fall within 3 standard deviations ($|z| \leq 3$)

Therefore, values with $|z| > 3$ are extremely rare and often considered outliers. This threshold is a reasonable default but should be adjusted based on your domain knowledge and the specific distribution of your data.

Creating Outlier Masks

```
# Assuming Z is standardised data
outliers = np.abs(Z) > 3

# Count outliers per feature
n_outliers = outliers.sum(axis=0)
print(f"Outliers per column: {n_outliers}")

# Find rows with any outlier
rows_with_outliers = outliers.any(axis=1)
print(f"Rows containing outliers: {rows_with_outliers.sum()}")
```

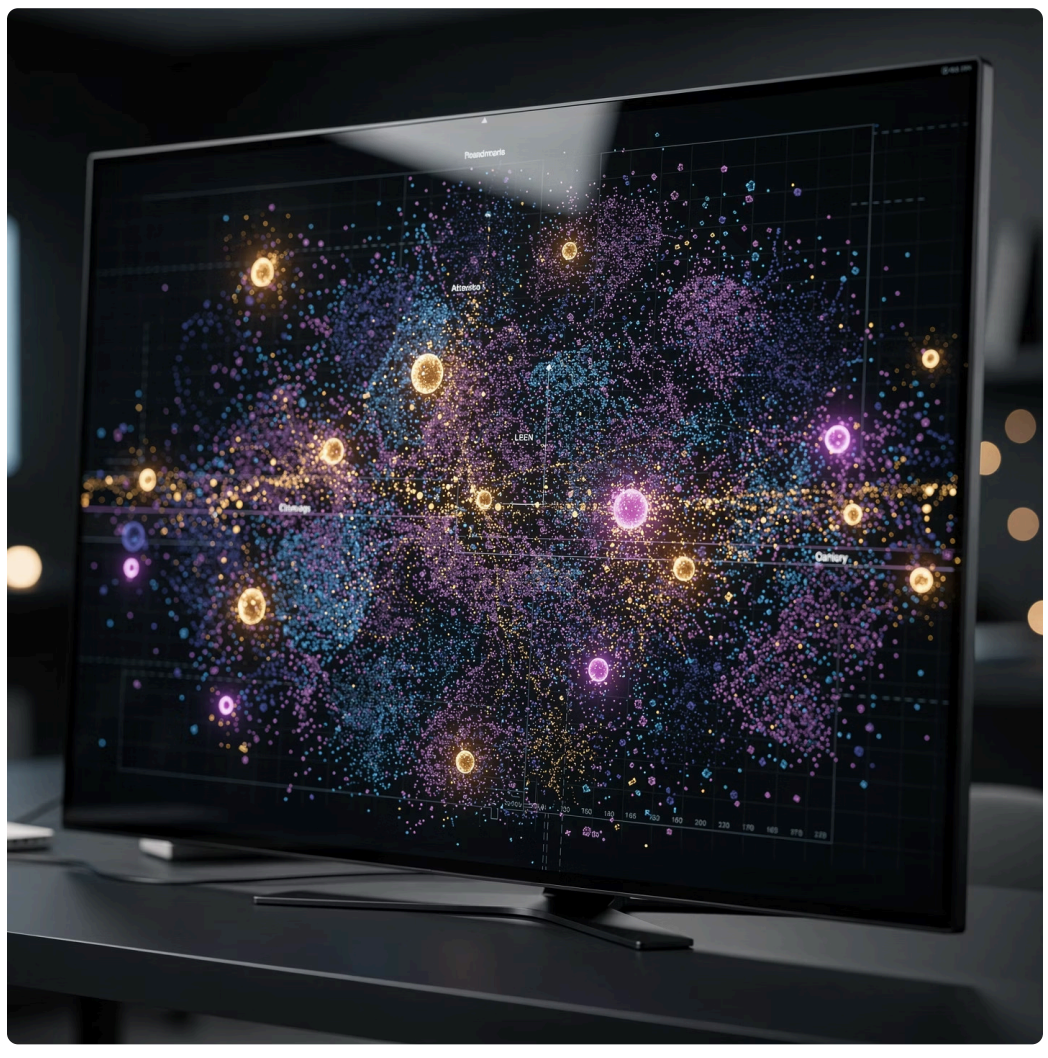
Practical Example

```
np.random.seed(42)
# Generate normal data with some outliers
data = np.random.randn(1000, 3) * 10 + 50
# Add some extreme values
data[0, 0] = 200  # outlier
data[5, 1] = -50  # outlier

# Standardise
mu = data.mean(axis=0)
sigma = data.std(axis=0)
Z = (data - mu) / sigma

# Detect outliers
outliers = np.abs(Z) > 3
print(f"Total outlier values: {outliers.sum()}")

# Identify outlier rows
outlier_rows = np.where(outliers.any(axis=1))[0]
print(f"Rows with outliers: {outlier_rows}")
```



Next Steps After Detection

- Investigate flagged points (data errors vs genuine extremes)
- Remove outliers if justified
- Cap/winsorise values to threshold
- Use robust statistics (median, MAD) instead of mean/std
- Try more sophisticated outlier detection methods

BEST PRACTICES

Performance Mindset: Practical Rules

Writing efficient NumPy code requires understanding a few key principles. These aren't premature optimisations—they're fundamental patterns that separate professional, production-ready code from slow prototypes. Following these rules will make your code faster, more maintainable, and more Pythonic.

1

Vectorise Operations

Always prefer vectorised NumPy operations over Python loops for numerical computations. If you find yourself writing `for` loops to process array elements, stop and look for the vectorised alternative. The performance difference is dramatic—often 10-100x faster. Examples: use `arr * 2` instead of `[x * 2 for x in arr]`, use `np.sum(arr > threshold)` instead of counting in a loop.

2

Minimise Type Conversions

Avoid repeated conversions between Python lists and NumPy arrays inside loops or frequently called functions. Convert once at the boundary of your code, do all processing in NumPy, then convert back if necessary. Each conversion has overhead that compounds quickly in inner loops. Keep your data pipeline in NumPy throughout the processing chain.

3

Use Appropriate dtypes

Keep arrays numeric and consistent in dtype whenever possible. Mixed-type arrays force NumPy to use the less efficient object dtype. Choose the smallest dtype that represents your data adequately: use int32 instead of int64 for smaller integers, float32 instead of float64 if precision allows. This reduces memory usage and can improve cache performance.

4

Leverage Broadcasting

Use broadcasting instead of manually tiling or repeating arrays. Broadcasting is memory-efficient and fast because it avoids creating intermediate copies. Understanding broadcasting rules lets you write elegant, efficient code for operations like centring data or computing pairwise differences.

5

Preallocate When Possible

If you must build arrays incrementally, preallocate with np.zeros() or np.empty() and fill values in-place rather than repeatedly appending or concatenating. Growing arrays dynamically is expensive because it requires reallocation. If you don't know the final size, consider using Python lists during construction, then converting to NumPy once.

6

Profile Before Optimising

Use `%%timeit` in Jupyter or Python's profiling tools to identify actual bottlenecks before optimising. Intuition about performance is often wrong. Focus optimisation effort where it matters. Sometimes a single vectorised operation that replaces a nested loop can eliminate 99% of your runtime.

Remember: Readable, correct code beats "clever" code. Write clear vectorised operations first, then optimise only if profiling shows it's necessary. NumPy's default operations are already highly optimised—trust them.