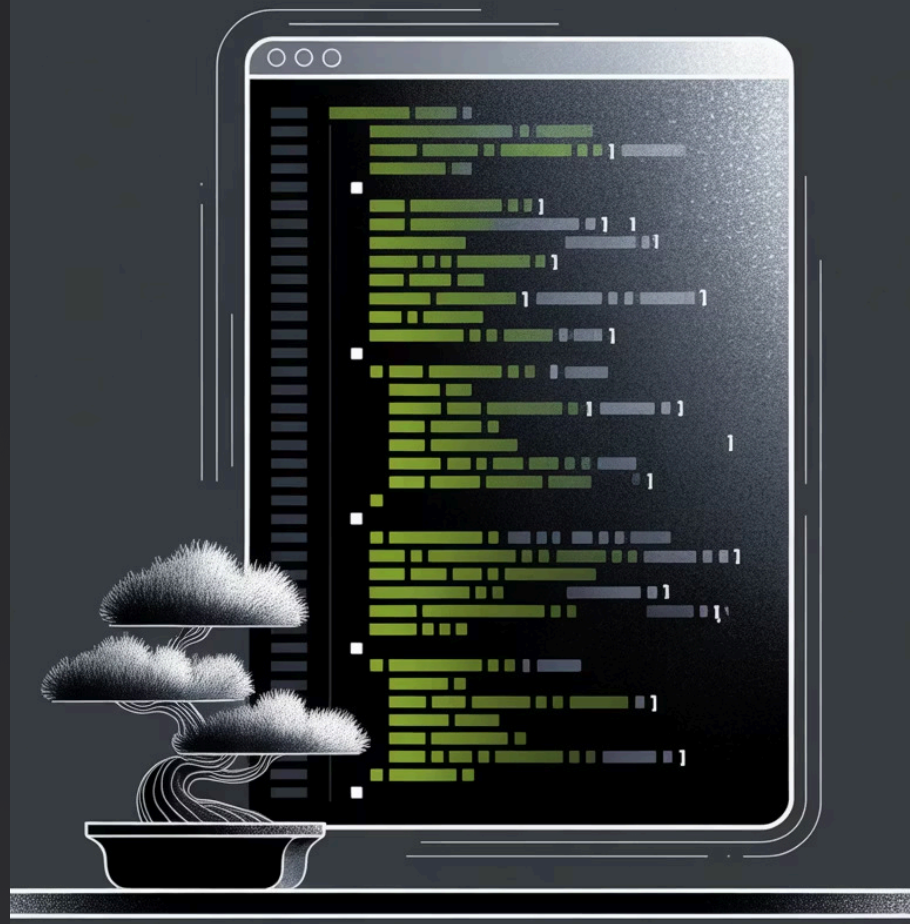
NumPy in Pythons

Array Operations



Course Overview

Array Slicing Fundamentals

Master views, copies, and memory management

Advanced Indexing

Boolean and fancy indexing techniques

Reshaping Operations

Transform array dimensions efficiently

Broadcasting Rules

Perform operations on different-sized arrays

Array Slicing

Array slicing is the cornerstone of NumPy operations. Understanding how to efficiently extract portions of arrays using slice notation enables powerful data manipulation and analysis workflows.

Basic Slicing Syntax

1D Array Slicing

```
import numpy as np
arr = np.array([0, 1, 2, 3, 4, 5])

# Basic slicing
arr[1:4]    # [1 2 3]
arr[:3]     # [0 1 2]
arr[2:]     # [2 3 4 5]
arr[::2]    # [0 2 4]
```

2D Array Slicing

```
arr_2d = np.array([[1, 2, 3, 4],
                   [5, 6, 7, 8],
                   [9, 10, 11, 12]])

# Row and column slicing
arr_2d[1:, 2:]    # [[7 8] [11 12]]
arr_2d[:2, 1:3]   # [[2 3] [6 7]]
```

Slicing follows the pattern `[start:stop:step]` and works across multiple dimensions simultaneously.

Views vs Copies: Critical Difference

Views (Shallow Copy)

Share the same memory as original array. Changes affect both arrays. Created by slicing operations.

```
arr = np.array([1, 2, 3, 4])
view = arr[1:3]
view[0] = 99
print(arr)  # [1 99 3 4]
```

Copies (Deep Copy)

Independent memory allocation. Changes don't affect original. Created explicitly with .copy().

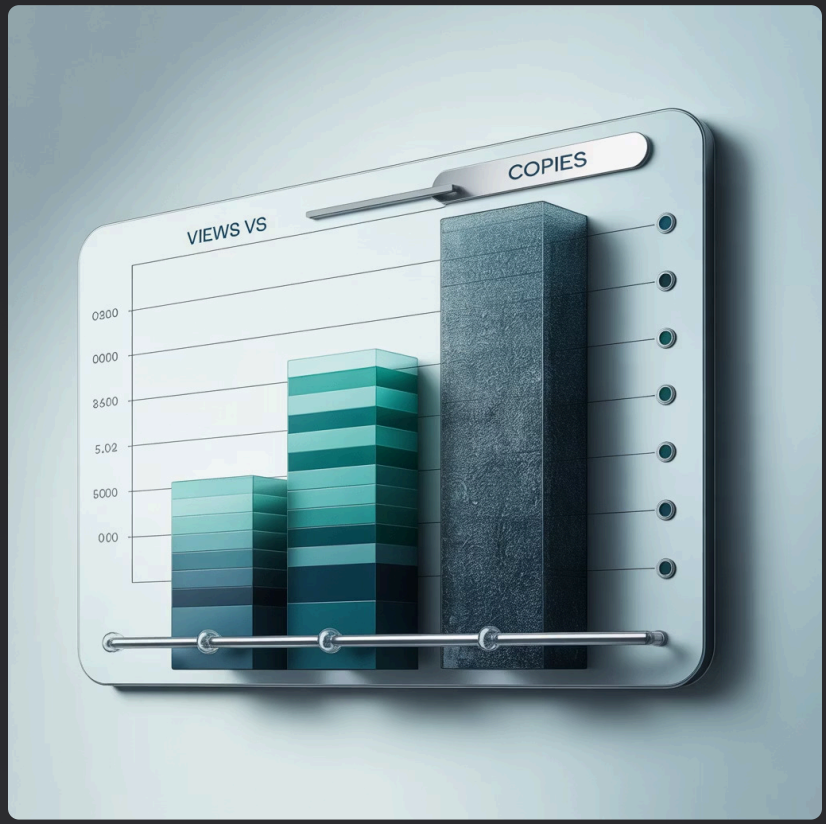
```
arr = np.array([1, 2, 3, 4])
copy = arr[1:3].copy()
copy[0] = 99
print(arr) # [1 2 3 4]
```

Memory Efficiency with Views

Why Views Matter

Views provide memory-efficient access to array data without duplication. This is crucial for large datasets where copying would consume excessive memory and processing time.

- Zero memory overhead for slicing operations
- Instant access to array segments
- Modifications propagate to original data
- Essential for real-time data processing



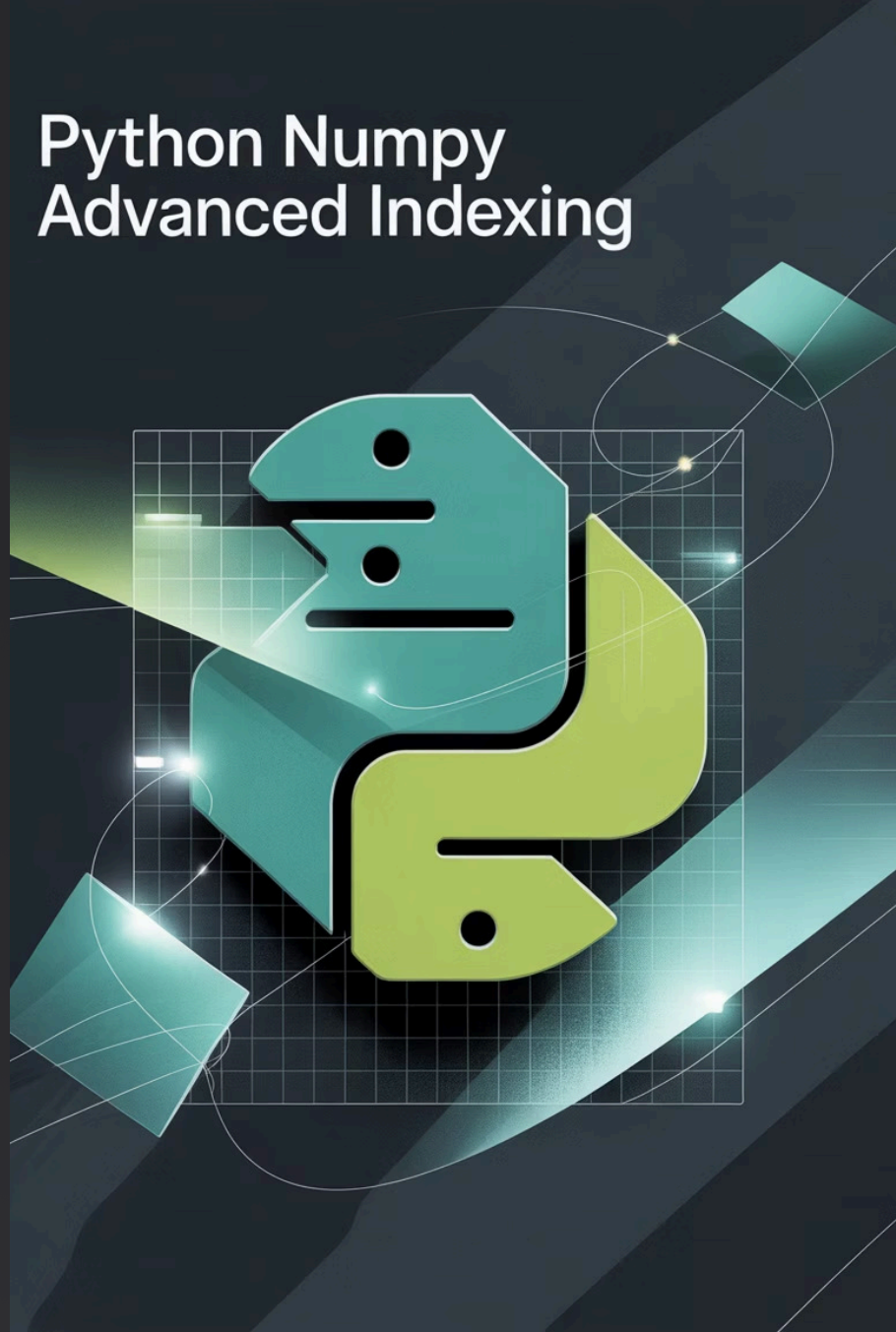
Checking Views and Copies

```
import numpy as np

arr = np.array([[1, 2, 3], [4, 5, 6]])
slice_view = arr[0:1, 1:]
explicit_copy = arr.copy()

# Check if arrays share memory
print(np.shares_memory(arr, slice_view))    # True
print(np.shares_memory(arr, explicit_copy)) # False

# Check base array
print(slice_view.base is arr)             # True
print(explicit_copy.base is None)         # True
```

Use `np.shares_memory()` and the `.base` attribute to verify whether you're working with views or copies.

Advanced Indexing

Advanced indexing goes beyond simple slicing, enabling sophisticated data selection using boolean conditions and integer arrays.



Boolean Indexing

Creating Boolean Masks

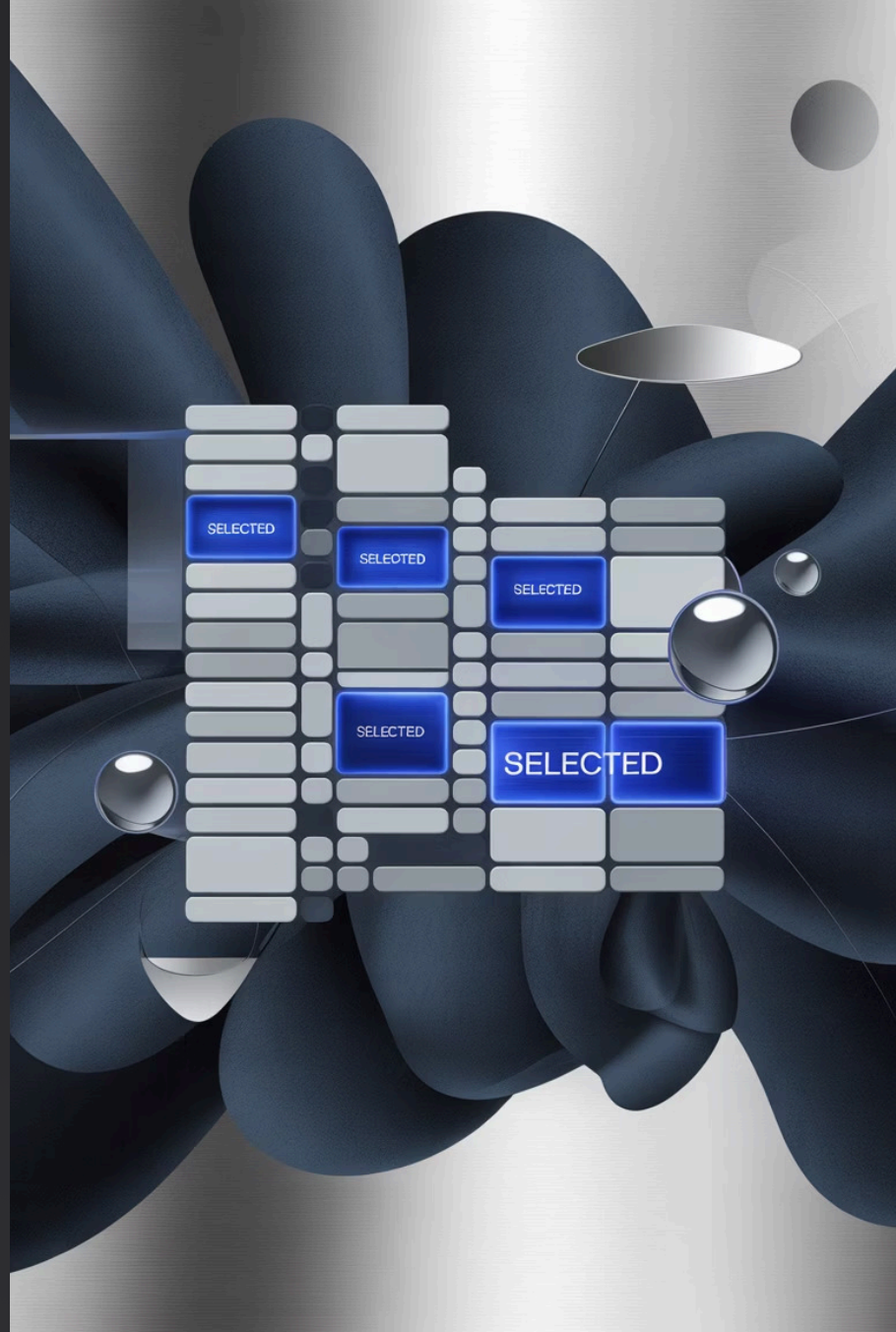
```
arr = np.array([1, 5, 8, 3, 9, 2])
mask = arr > 4
print(mask)
# [False True True False True False]

# Apply boolean mask
result = arr[mask]
print(result)  # [5 8 9]
```

Complex Conditions

```
# Multiple conditions with & and |
arr = np.array([1, 5, 8, 3, 9, 2])

# Values between 3 and 8
mask = (arr >= 3) & (arr <= 8)
print(arr[mask]) # [5 8 3]

# Values less than 3 OR greater than 7
mask = (arr < 3) | (arr > 7)
print(arr[mask]) # [1 8 9]
```

Boolean Indexing with 2D Arrays

```
data = np.array([[1, 2, 3, 4],
 [5, 6, 7, 8],
 [9, 10, 11, 12]])

# Find all values greater than 6
mask = data > 6
print(data[mask]) # [7 8 9 10 11 12]

# Replace values conditionally
data[data > 8] = 0
print(data)
# [[ 1 2 3 4]
# [ 5 6 7 8]
# [ 0 0 0 0]]
```



Fancy Indexing

Integer Array Indexing

```
arr = np.array([10, 20, 30, 40, 50])
indices = np.array([1, 3, 4])
result = arr[indices]
print(result)  # [20 40 50]
```

Select specific elements using integer arrays

2D Fancy Indexing

```
arr_2d = np.array([[1, 2], [3, 4], [5, 6]])
row_indices = [0, 2]
col_indices = [1, 0]
result = arr_2d[row_indices, col_indices]
print(result)  # [2 5]
```

Simultaneously index rows and columns

Advanced Selection Techniques

```
# Combine boolean and fancy indexing
data = np.random.randint(0, 100, (5, 4))
print("Original data:")
print(data)

# Select rows where first column > 50
mask = data[:, 0] > 50
selected_rows = data[mask]
print("\nRows where first column > 50:")
print(selected_rows)

# Select specific columns from these rows
columns_to_select = [1, 3]
final_result = selected_rows[:, columns_to_select]
print("\nColumns 1 and 3 from selected rows:")
print(final_result)
```

Array Reshaping

Reshaping allows you to change array dimensions while preserving data integrity, enabling flexible data structure transformations for various computational needs.

Basic Reshaping Operations

The reshape() Method

```
arr = np.array([1, 2, 3, 4, 5, 6])
print("Original shape:", arr.shape)  # (6,)

# Reshape to 2D
arr_2d = arr.reshape(2, 3)
print("2x3 shape:")
print(arr_2d)
# [[1 2 3]
#  [4 5 6]]

# Reshape to 3D
arr_3d = arr.reshape(2, 1, 3)
print("3D shape:", arr_3d.shape)  # (2, 1, 3)
```

Automatic Dimension Inference

```
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8])

# Use -1 to infer dimension
reshaped = arr.reshape(2, -1)
print(reshaped.shape) # (2, 4)
print(reshaped)
# [[1 2 3 4]
# [5 6 7 8]]

# Multiple -1 not allowed
# arr.reshape(-1, -1) # ValueError
```

Flattening and Raveling

1

flatten()

Creates a copy of the array collapsed into 1D

```
arr = np.array([[1, 2], [3, 4]])
flat = arr.flatten()
flat[0] = 99
print(arr[0, 0])  # Still 1 (copy)
```

2

ravel()

Returns a view when possible, copy when necessary

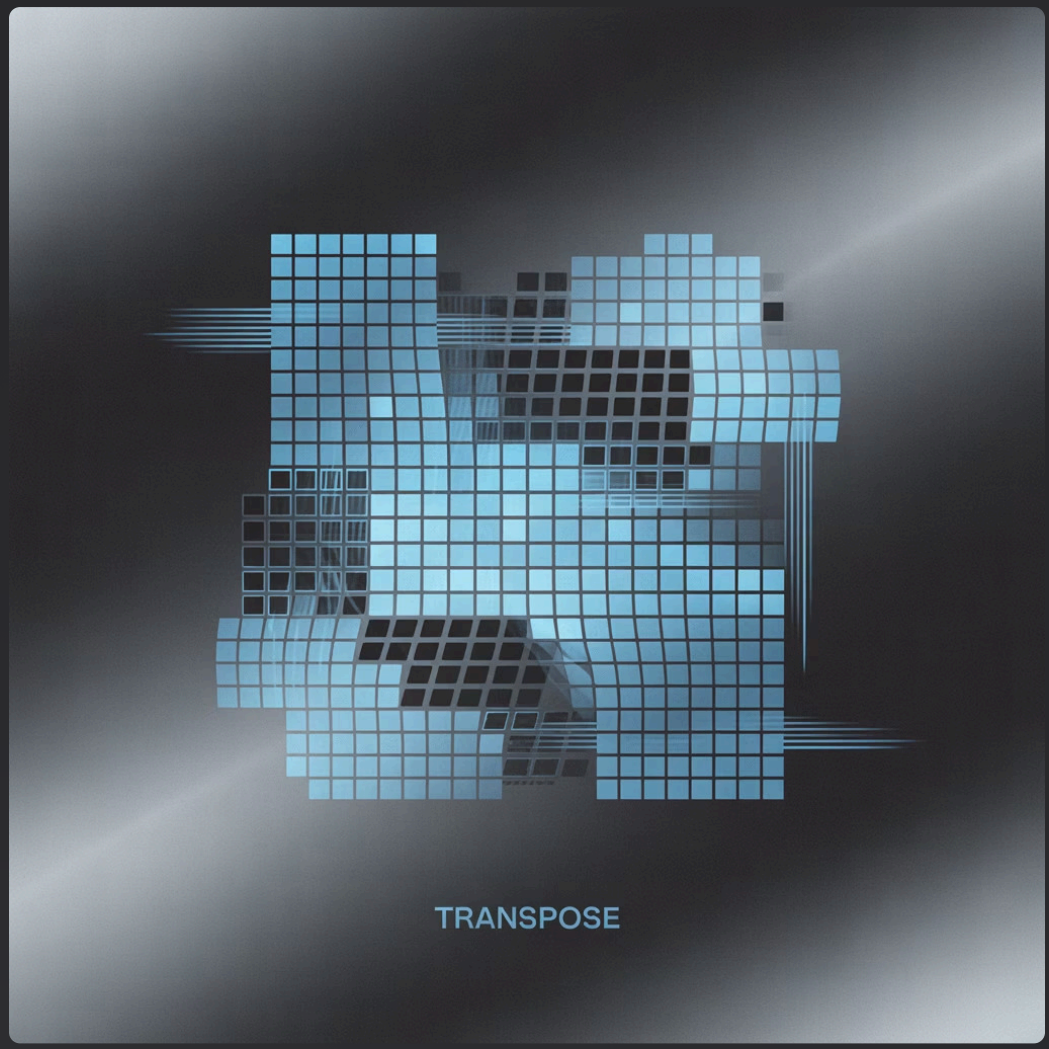
```
arr = np.array([[1, 2], [3, 4]])
rav = arr.ravel()
rav[0] = 99
print(arr[0, 0])  # Now 99 (view)
```

Choose `ravel()` for memory efficiency, `flatten()` when you need guaranteed independence.

Transposing Arrays

2D Transpose

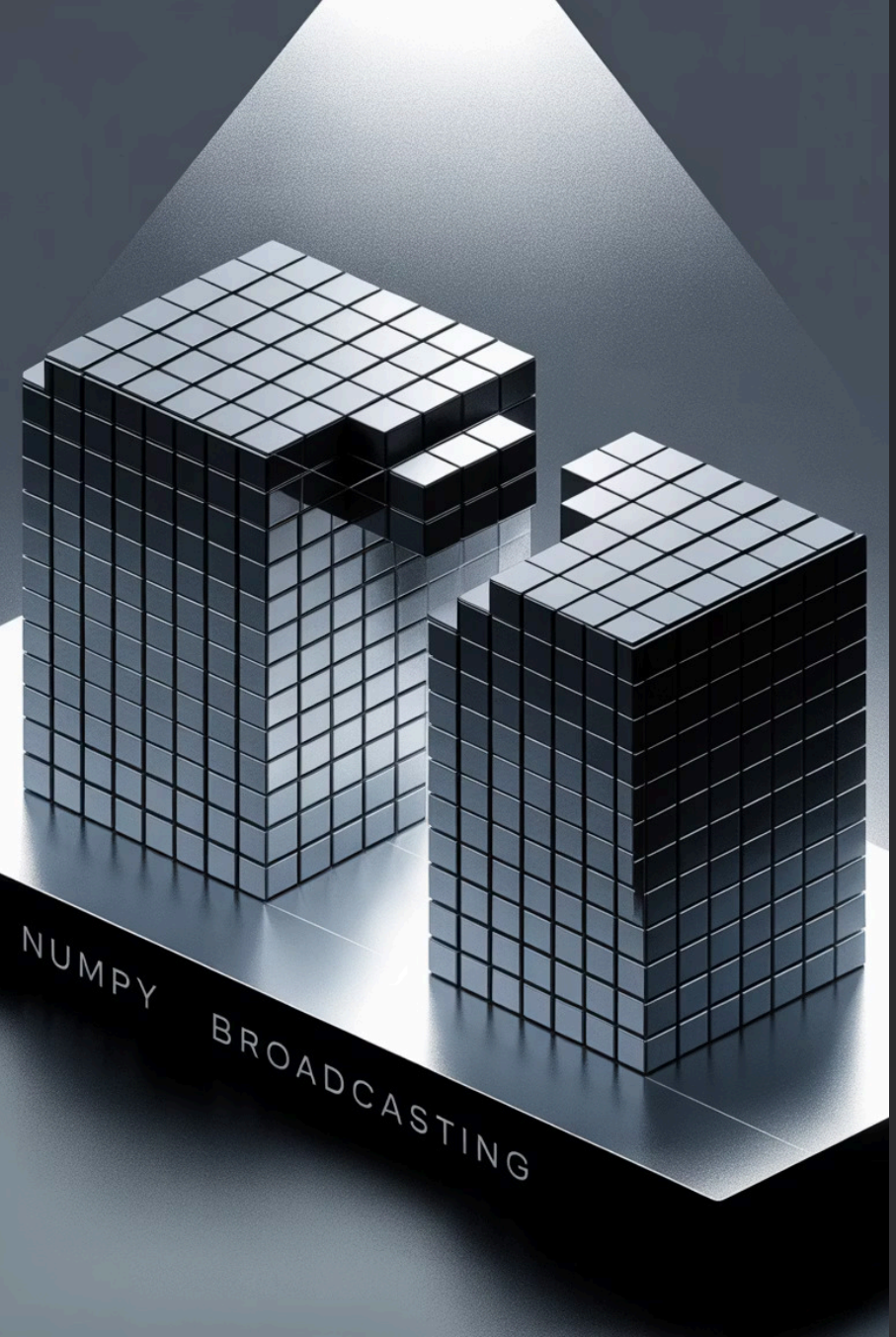
```
arr = np.array([[1, 2, 3],
                [4, 5, 6]])
print("Original shape:", arr.shape)  # (2, 3)

# Transpose methods
transposed = arr.T
# or arr.transpose()
print("Transposed shape:", transposed.shape)  # (3, 2)
print(transposed)
# [[1 4]
#  [2 5]
#  [3 6]]
```



Multi-dimensional Transpose

```
arr_3d = np.random.random((2, 3, 4))
print("Original shape:", arr_3d.shape)  # (2, 3, 4)

# Specify axis order for transpose
transposed = arr_3d.transpose(2, 0, 1)
print("Transposed shape:", transposed.shape)  # (4, 2, 3)

# Using axes parameter
transposed_alt = np.transpose(arr_3d, axes=(1, 2, 0))
print("Alternative transpose:", transposed_alt.shape)  # (3, 4, 2)
```

For arrays with more than 2 dimensions, specify the desired axis order explicitly using the `axes` parameter.



Broadcasting

Broadcasting enables arithmetic operations between arrays of different shapes, following specific rules that make NumPy operations both powerful and intuitive.

Broadcasting Rules

Align Dimensions

Compare shapes from right to left

Compatible Sizes

Dimensions must be equal, 1, or missing

Stretch Dimensions

Size-1 dimensions are stretched to match

Perform Operation

Element-wise operation on aligned arrays

Broadcasting Examples

Scalar Broadcasting

```
arr = np.array([[1, 2, 3],
                [4, 5, 6]])
result = arr + 10
print(result)
# [[11 12 13]
#  [14 15 16]]

# Shape compatibility:
# (2, 3) + () → (2, 3)
```

1D Array Broadcasting

```
arr = np.array([[1, 2, 3],
 [4, 5, 6]])
col_vector = np.array([[10], [20]])
result = arr + col_vector
print(result)
# [[11 12 13]
# [24 25 26]]

# Shape compatibility:
# (2, 3) + (2, 1) → (2, 3)
```

Complex Broadcasting Scenarios

```
# Broadcasting with different dimensional arrays
arr_2d = np.array([[1, 2, 3],
 [4, 5, 6]]) # Shape: (2, 3)
arr_1d = np.array([10, 20, 30]) # Shape: (3,)

result = arr_2d + arr_1d
print("2D + 1D result:")
print(result)
# [[11 22 33]
# [14 25 36]]

# 3D broadcasting example
arr_3d = np.random.random((2, 3, 4)) # Shape: (2, 3, 4)
arr_2d = np.random.random((3, 1)) # Shape: (3, 1)
result_3d = arr_3d + arr_2d # Shape: (2, 3, 4)
```

Broadcasting Errors and Solutions

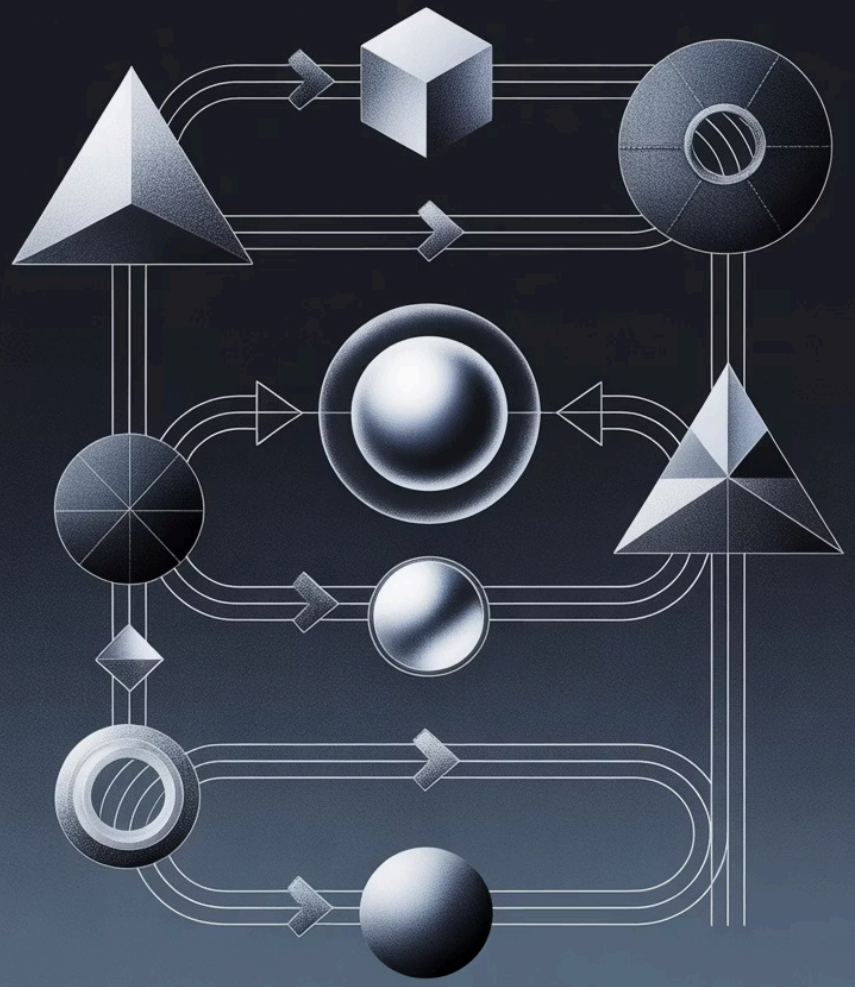
Common Broadcasting Error

```
arr1 = np.array([[1, 2]])     # (1, 2)
arr2 = np.array([[3], [4], [5]]) # (3, 1)
# This works: (1, 2) + (3, 1) → (3, 2)

arr3 = np.array([1, 2, 3])     # (3,)
arr4 = np.array([[4], [5]])    # (2, 1)
# ValueError: shapes (3,) and (2,1) not compatible
```

Solutions

```
# Reshape to compatible dimensions
arr3_reshaped = arr3.reshape(1, 3) # (1, 3)
result = arr3_reshaped + arr4 # (2, 3)

# Or use newaxis
arr3_newaxis = arr3[np.newaxis, :] # (1, 3)
result = arr3_newaxis + arr4 # (2, 3)
```

Insights Engine



Practical Broadcasting Applications

Data Normalization

```
data = np.random.random((1000, 5))
mean = data.mean(axis=0)  # Shape: (5,)
std = data.std(axis=0)    # Shape: (5,)
normalized = (data - mean) / std
```

Image Processing

```
image = np.random.randint(0, 256, (100, 100, 3))
brightness = 0.8
adjusted = image * brightness  # Broadcast scalar
```

Matrix Operations

```
matrix = np.random.random((10, 10))
row_sums = matrix.sum(axis=1, keepdims=True)
probabilities = matrix / row_sums # Normalize rows
```

Performance Considerations

Memory and Speed Benefits

- **Views vs Copies:** Slicing creates views by default, saving memory and improving performance
- **Broadcasting:** Avoids explicit loops and temporary array creation
- **Vectorization:** NumPy's C-optimized operations are faster than Python loops
- **Memory Layout:** Understanding row-major (C) vs column-major (Fortran) order impacts performance

> Always profile your code when working with large arrays. Use `%timeit` in Jupyter notebooks or the `timeit` module to compare different approaches.

Key Takeaways

Master Views and Copies

Understanding when NumPy creates views vs copies is crucial for memory management and avoiding unexpected behavior in your data analysis pipelines.

Leverage Advanced Indexing

Boolean and fancy indexing enable sophisticated data selection and filtering operations that form the foundation of data analysis workflows.

Embrace Broadcasting

Broadcasting rules allow elegant operations between different-shaped arrays, eliminating the need for explicit loops and improving both code clarity and performance.

These NumPy fundamentals will accelerate your data science journey and enable more efficient, readable code for complex array operations.