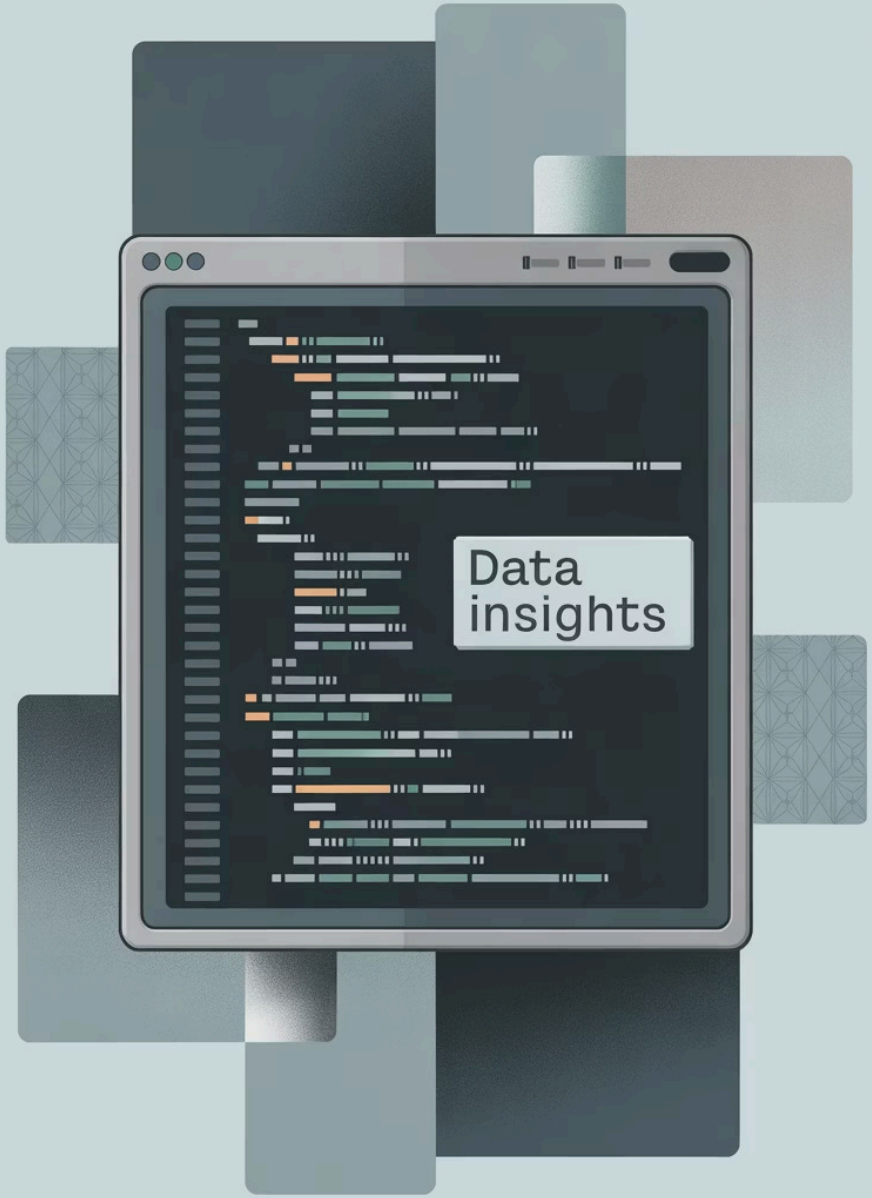
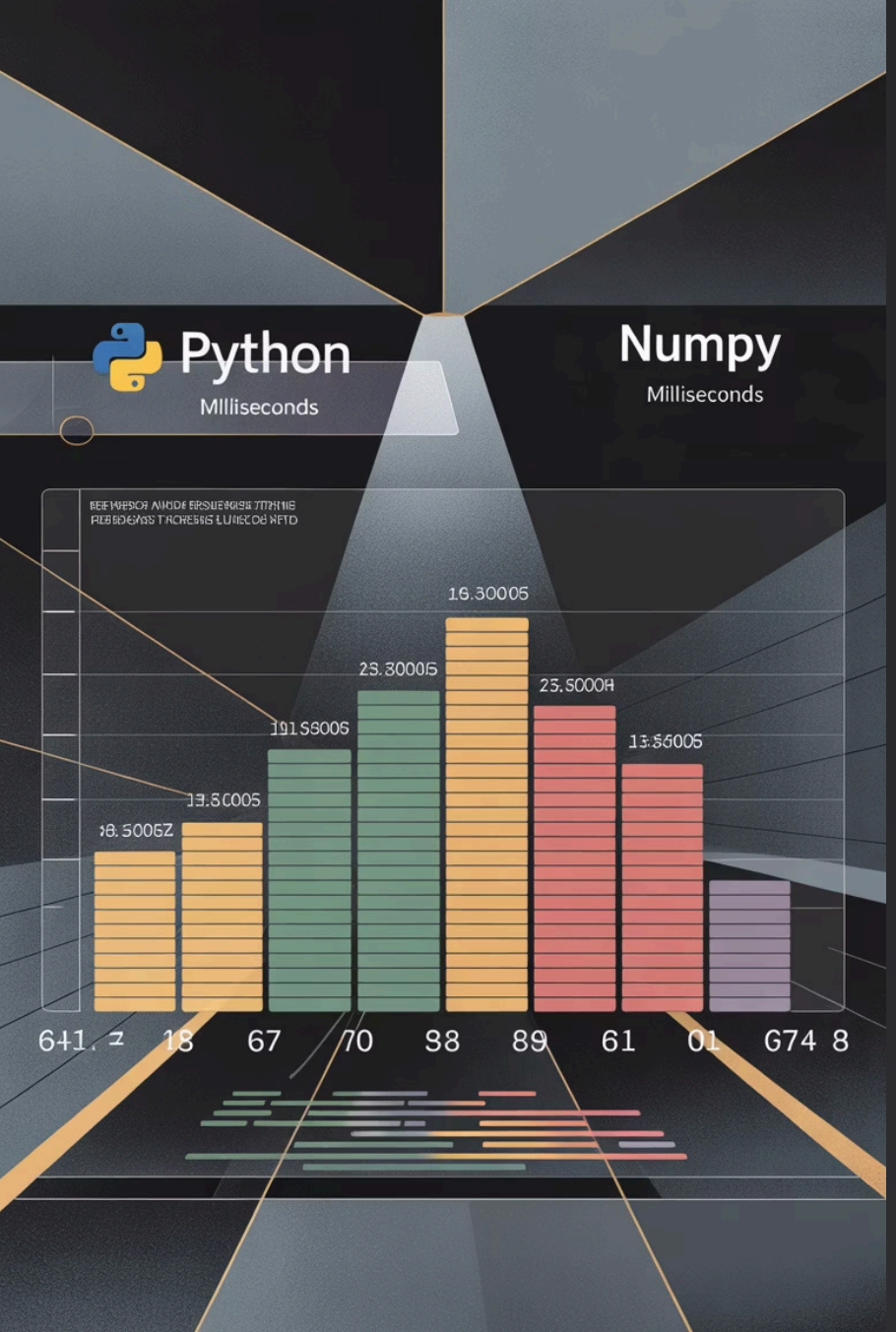


NumPy in Python

ndarrays



Why NumPy Matters for Python Developers

Performance

NumPy arrays are 10-50x faster than Python lists for numerical operations. Written in C and optimized for vectorized computations, NumPy transforms slow Python loops into lightning-fast array operations.

Foundation

Every major data science library builds on NumPy. Pandas DataFrames, scikit-learn models, and matplotlib visualizations all use NumPy arrays under the hood as their core data structure.

Introducing N-Dimensional Arrays

At the heart of NumPy lies the **ndarray** (n-dimensional array) - a powerful container for homogeneous data. Unlike Python lists, ndarrays store elements of the same data type in contiguous memory blocks, enabling efficient computation.

Homogeneous Data

All elements share the same data type, enabling optimized operations and predictable memory usage patterns.

Contiguous Memory

Elements stored sequentially in memory allow for cache-friendly access and vectorized operations.

N-Dimensional

Support for arrays of any dimensionality, from simple 1D vectors to complex multi-dimensional tensors.

Creating Your First NumPy Array

Let's start with the basics. NumPy provides multiple ways to create arrays, each suited for different scenarios.

```
import numpy as np

# From Python list
arr1 = np.array([1, 2, 3, 4, 5])
print(arr1) # [1 2 3 4 5]

# From nested lists (2D)
arr2 = np.array([[1, 2, 3], [4, 5, 6]])
print(arr2)
# [[1 2 3]
# [4 5 6]]

# Using built-in functions
zeros = np.zeros((3, 4)) # 3x4 array of zeros
ones = np.ones((2, 2)) # 2x2 array of ones
range_arr = np.arange(0, 10, 2) # [0 2 4 6 8]
```

Understanding Array Dimensions

1D Arrays (Vectors)

Single dimension arrays like `[1, 2, 3, 4]`. Think of them as mathematical vectors or simple lists of values.

2D Arrays (Matrices)

Two-dimensional arrays with rows and columns. Perfect for representing tables, images, or mathematical matrices.

3D+ Arrays (Tensors)

Multi-dimensional arrays used in deep learning, image processing, and scientific simulations. Think RGB images or time series data.

Array Shape: The Blueprint of Your Data

Every NumPy array has a **shape** - a tuple describing the size of each dimension. Understanding shape is crucial for array manipulation and avoiding common errors.

```
import numpy as np

# 1D array
arr1d = np.array([1, 2, 3, 4, 5])
print(f"Shape: {arr1d.shape}") # Shape: (5,)
print(f"Dimensions: {arr1d.ndim}") # Dimensions: 1

# 2D array
arr2d = np.array([[1, 2, 3], [4, 5, 6]])
print(f"Shape: {arr2d.shape}") # Shape: (2, 3)
print(f"Dimensions: {arr2d.ndim}") # Dimensions: 2

# 3D array
arr3d = np.zeros((2, 3, 4))
print(f"Shape: {arr3d.shape}") # Shape: (2, 3, 4)
print(f"Total elements: {arr3d.size}") # Total elements: 24
```

NumPy Data Types: Precision and Memory

NumPy offers a rich collection of data types, each optimized for specific use cases. Choosing the right data type affects both memory usage and computational performance.

Integers

- `int8`, `int16`, `int32`, `int64`
- `uint8`, `uint16`, `uint32`, `uint64`
- Signed and unsigned variants
- Memory: 1-8 bytes per element

Floating Point

- `float16`, `float32`, `float64`
- `float64` is the default
- Higher precision = more memory
- Memory: 2-8 bytes per element

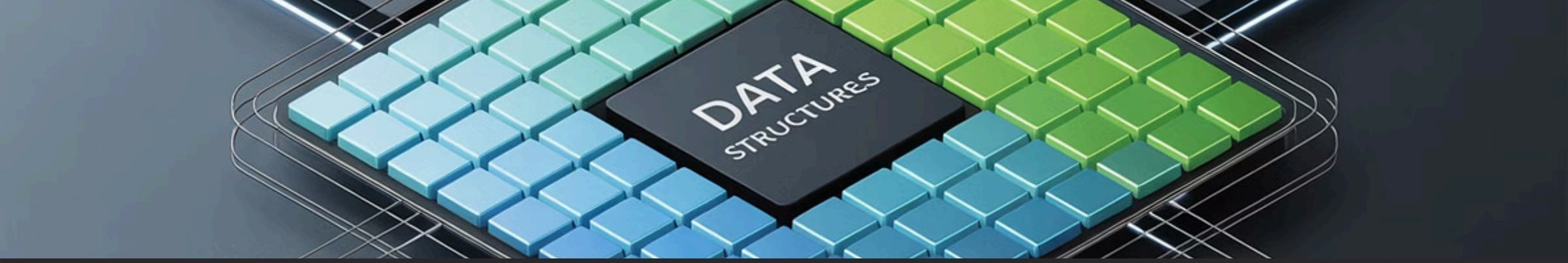
Other Types

- `bool` - True/False values
- `complex64`, `complex128`
- `datetime64`, `timedelta64`
- Custom structured types

Working with Data Types in Practice

Explicit data type control helps optimize memory usage and ensures numerical precision for your specific application needs.

```
import numpy as np

# Specify data type during creation
int_arr = np.array([1, 2, 3], dtype=np.int32)
float_arr = np.array([1.0, 2.0, 3.0], dtype=np.float32)
bool_arr = np.array([True, False, True], dtype=bool)

print(f"Integer array: {int_arr.dtype}") # int32
print(f"Float array: {float_arr.dtype}") # float32
print(f"Boolean array: {bool_arr.dtype}") # bool

# Convert between types
original = np.array([1.7, 2.3, 3.9])
as_int = original.astype(np.int32)
print(f"Original: {original}") # [1.7 2.3 3.9]
print(f"As integer: {as_int}") # [1 2 3]

# Memory usage comparison
print(f"Float64 memory: {np.array([1.0]).nbytes} bytes") # 8 bytes
print(f"Float32 memory: {np.array([1.0], dtype=np.float32).nbytes} bytes") # 4
bytes
```



Memory Layout: Row-Major vs Column-Major

NumPy arrays can be stored in memory using different layouts, affecting performance for different access patterns. Understanding memory layout is crucial for optimizing numerical computations.

C-style (Row-Major)

Default NumPy layout. Stores elements row by row in memory. Optimal for operations that access data row-wise, such as iterating through matrix rows.

```
# Row-major (C-style)
arr = np.array([[1, 2, 3],
                [4, 5, 6]], order='C')
print(arr.flags.c_contiguous)  # True
```

Fortran-style (Column-Major)

Stores elements column by column in memory. Better performance for column-wise operations, common in linear algebra libraries and scientific computing.

```
# Column-major (Fortran-style)
arr = np.array([[1, 2, 3],
 [4, 5, 6]], order='F')
print(arr.flags.f_contiguous) # True
```

Array Strides: Understanding Memory Access

Strides define how many bytes to skip in memory when moving along each axis. This low-level concept explains how NumPy achieves efficient array operations without copying data.

```
import numpy as np

# 2D array with shape (3, 4)
arr = np.arange(12).reshape(3, 4)
print("Array:")
print(arr)
# [[ 0  1  2  3]
#  [ 4  5  6  7]
#  [ 8  9 10 11]]

print(f"Shape: {arr.shape}")       # (3, 4)
print(f"Strides: {arr.strides}")   # (32, 8) on 64-bit system
print(f"Item size: {arr.itemsize}") # 8 bytes (int64)

# Understanding strides:
# - Moving to next row: skip 32 bytes (4 elements × 8 bytes)
# - Moving to next column: skip 8 bytes (1 element × 8 bytes)
```

Pro Tip: Understanding strides helps explain why some operations are faster than others and how views work in NumPy.

Reshaping Arrays: Changing Dimensions

NumPy allows you to change an array's shape without copying data, as long as the total number of elements remains the same. This is a powerful feature for data manipulation and preparation.

```
import numpy as np

# Start with 1D array
original = np.arange(12)
print(f"Original: {original}") # [0 1 2 3 4 5 6 7 8 9 10 11]
print(f"Shape: {original.shape}") # (12,)

# Reshape to 2D
matrix = original.reshape(3, 4)
print(f"3x4 matrix:\n{matrix}")
# [[ 0 1 2 3]
# [ 4 5 6 7]
# [ 8 9 10 11]]

# Reshape to 3D
tensor = original.reshape(2, 2, 3)
print(f"2x2x3 tensor shape: {tensor.shape}")

# Use -1 for automatic dimension calculation
auto_shape = original.reshape(-1, 2) # NumPy calculates: (6, 2)
print(f"Auto-calculated shape: {auto_shape.shape}")
```

Array Views vs Copies: Memory Efficiency

NumPy distinguishes between views (sharing memory) and copies (independent memory). Understanding this distinction prevents unexpected behavior and optimizes memory usage in data processing pipelines.

Views Share Memory

Changes to a view affect the original array. Operations like slicing, reshaping, and transposing typically create views, not copies.

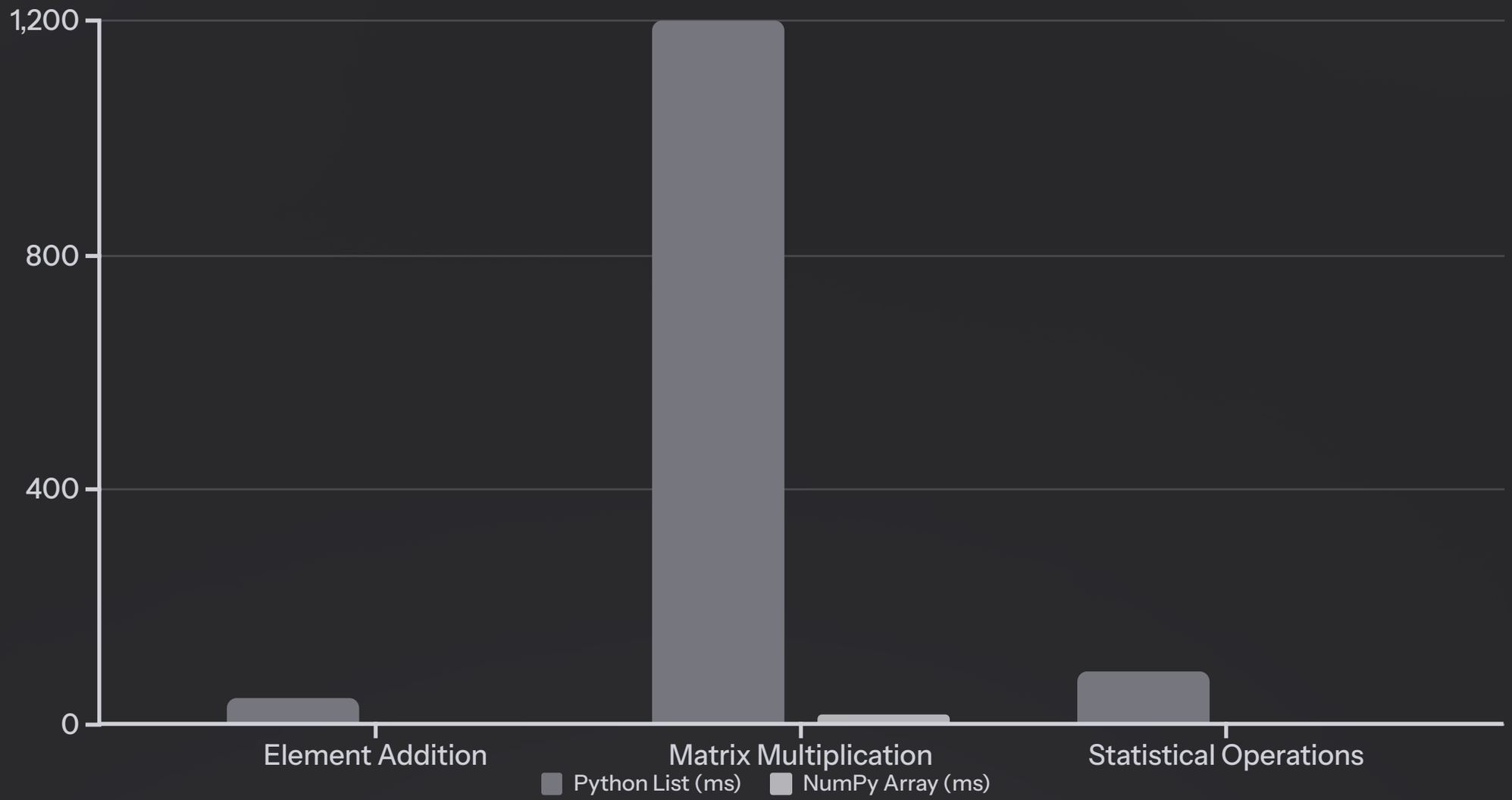
Copies Are Independent

Copies have their own memory space. Modifying a copy doesn't affect the original array. Use when you need independent data manipulation.

```
import numpy as np

original = np.array([[1, 2, 3], [4, 5, 6]])

# Creating a view (shares memory)
view = original[0:2, 0:2] # Slice creates a view
view[0, 0] = 999
print(f"Original after view change:\n{original}")
# [[999 2 3]
# [ 4 5 6]]

# Creating a copy (independent memory)
copy = original.copy()
copy[0, 0] = 777
print(f"Original after copy change:\n{original}") # Unchanged
```

Performance Implications of Array Design

The design choices in NumPy arrays directly impact computational performance. Understanding these concepts helps you write efficient numerical code that scales with your data.

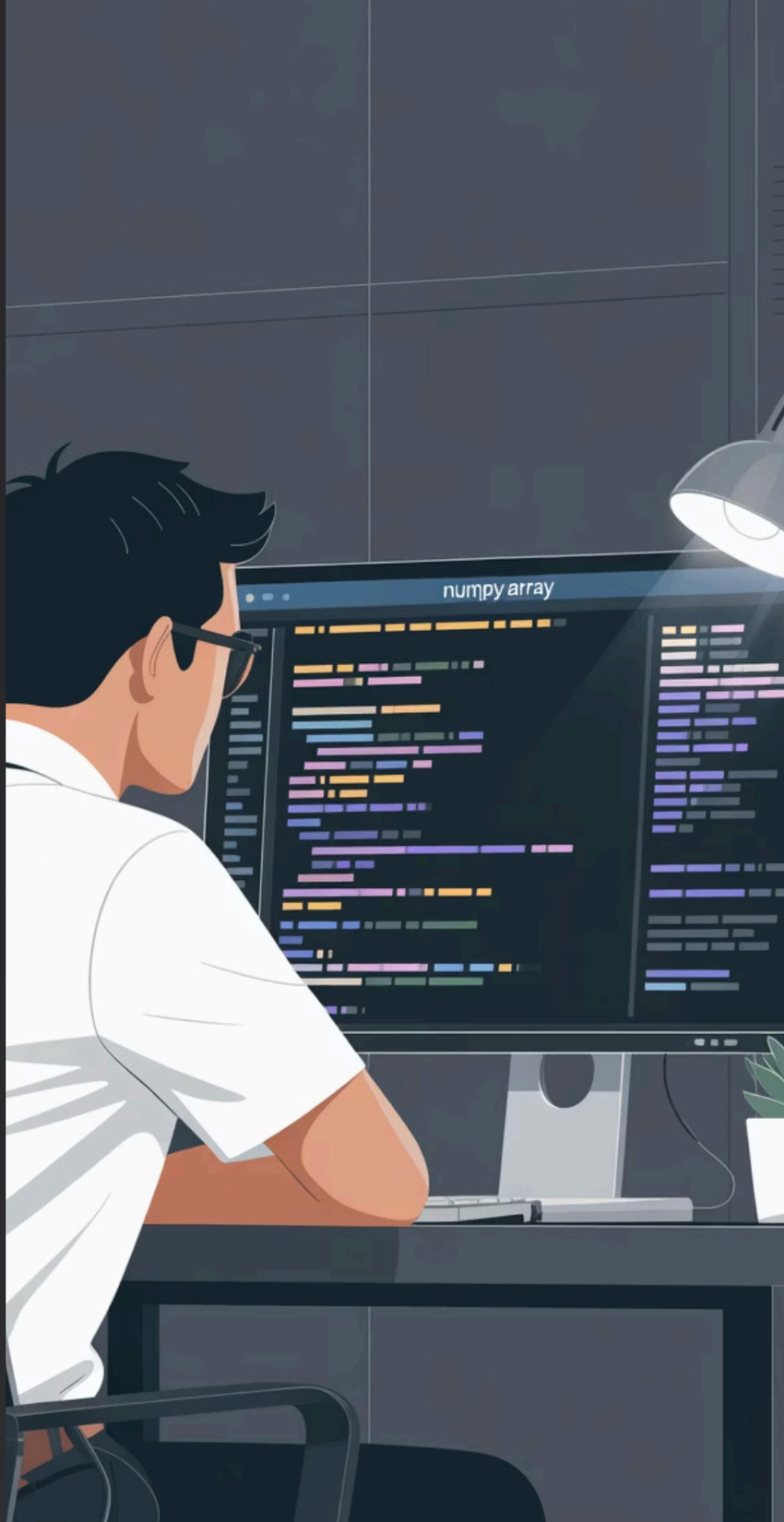


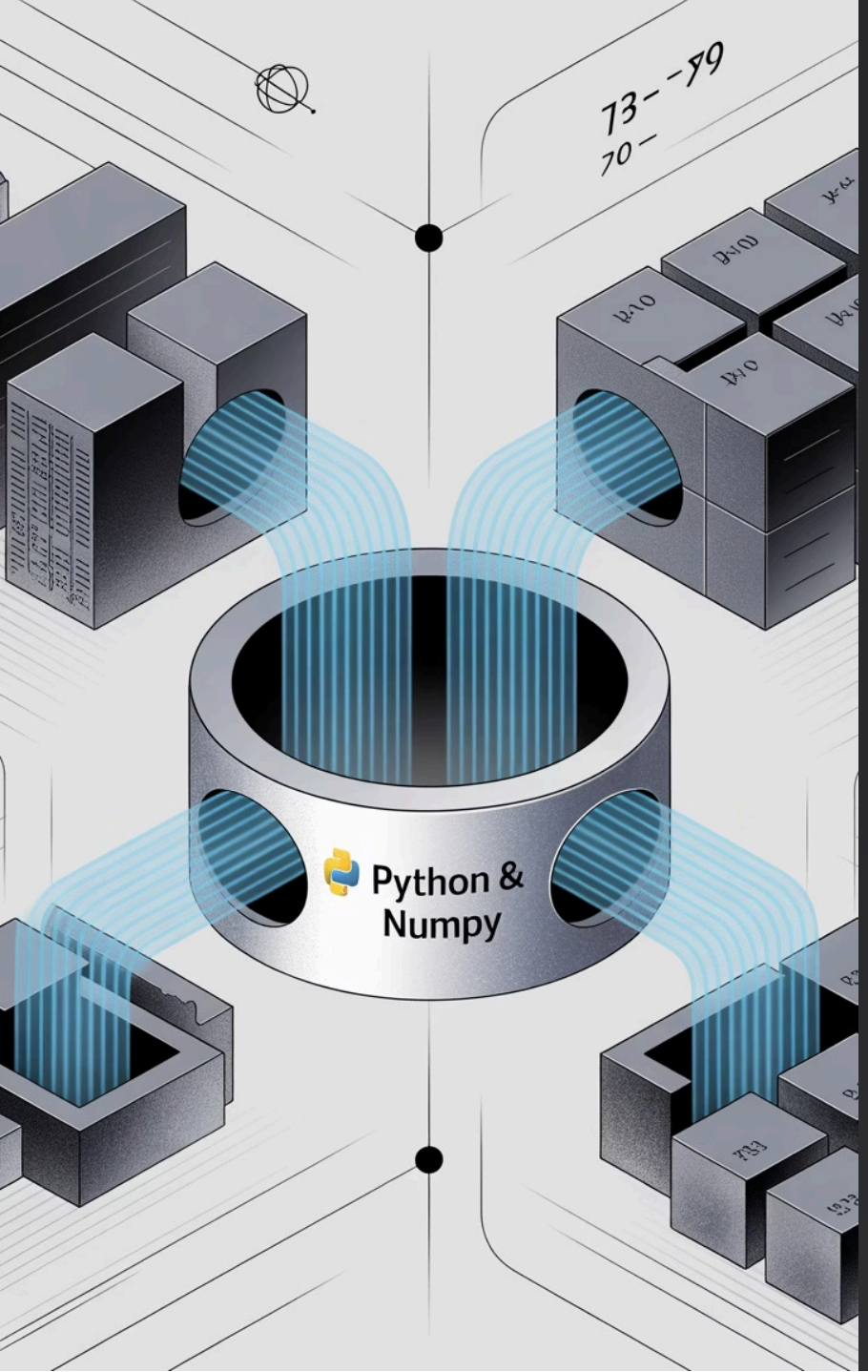
NumPy's performance advantages come from vectorized operations, optimized C implementations, and efficient memory layouts that minimize cache misses during computation.

Common Array Inspection Techniques

Debugging and understanding your arrays is essential for effective NumPy usage. These inspection techniques help you verify array properties and diagnose issues in your code.

```
import numpy as np

arr = np.random.randint(0, 100, size=(4, 5)).astype(np.float32)

# Essential array properties
print(f"Shape: {arr.shape}") # Dimensions
print(f"Data type: {arr.dtype}") # Element type
print(f"Size: {arr.size}") # Total elements
print(f"Item size: {arr.itemsize}") # Bytes per element
print(f"Total bytes: {arr.nbytes}") # Total memory usage
print(f"Dimensions: {arr.ndim}") # Number of axes

# Memory layout information
print(f"C-contiguous: {arr.flags.c_contiguous}")
print(f"F-contiguous: {arr.flags.f_contiguous}")
print(f"Strides: {arr.strides}")

# Quick data preview
print(f"Min value: {arr.min()}")
print(f"Max value: {arr.max()}")
print(f"Mean: {arr.mean():.2f}")
```





Next Steps in Your NumPy Journey

You've mastered the fundamentals of NumPy arrays - data types, shapes, and memory layout. These concepts form the foundation for all advanced NumPy operations and scientific computing in Python.

Array Operations

Learn broadcasting, mathematical operations, and universal functions to perform efficient computations on your arrays.

Advanced Indexing

Master Boolean indexing, fancy indexing, and multi-dimensional slicing for sophisticated data selection and manipulation.

Linear Algebra

Explore NumPy's linear algebra capabilities for matrix operations, eigenvalues, and solving systems of equations.

With these fundamentals in place, you're ready to tackle real-world data analysis challenges and build sophisticated numerical applications with confidence.