NumPy Crash Course for Data Science & Machine Learning

Table of Contents

Introduction

NumPy (Numerical Python) is the cornerstone of scientific computing in Python and the backbone of the entire PyData ecosystem. It provides a powerful, efficient N-dimensional array object called `ndarray`, along with a comprehensive collection of mathematical functions to operate on these arrays.

What Makes NumPy Essential?

NumPy is not just another Python library—it's the foundation upon which the entire data science and machine learning ecosystem is built. Libraries like **Pandas**, **Scikit-learn**, **TensorFlow**, **PyTorch**, **Matplotlib**, and **SciPy** all depend on NumPy arrays as their fundamental data structure.

Key Features

- **High-performance N-dimensional arrays** with optimized C/C++ implementations

- **Broadcasting capabilities** for operations between arrays of different shapes
- **Linear algebra functions** including matrix operations, decompositions, and eigenvalues
- **Fourier transforms** and random number generation
- **Integration tools** for C/C++ and Fortran code
- **Memory-efficient** operations with sophisticated memory management

Why NumPy Over Python Lists?

Understanding why NumPy arrays outperform Python lists requires diving into their fundamental differences in design and implementation.

1. Memory Layout & Data Types

Python Lists (Heterogeneous & Scattered)

```
Python List: [1, 2.5, "hello", True]
Memory Layout:
┌─────────┬─────────┬─────────┬─────────┐
│ ptr→int │ ptr→flt │ ptr→str │ ptr→bln │  ← List of pointers
└─────────┴─────────┴─────────┴─────────┘
     │         │         │         │
     ▼         ▼         ▼         ▼
  [int obj] [float obj] [str obj] [bool obj]  ← Objects scattered in memory
```

NumPy Arrays (Homogeneous & Contiguous)

```
NumPy Array: [1, 2, 3, 4]
Memory Layout:
┌───┬───┬───┬───┐
│ 1 │ 2 │ 3 │ 4 │  ← Raw data stored contiguously
└───┴───┴───┴───┘
```

2. Performance Advantages

Aspect	Python Lists	NumPy Arrays
Memory Usage	~28 bytes per integer	~8 bytes per integer
Cache Efficiency	Poor (scattered memory)	Excellent (contiguous memory)

Aspect	Python Lists	NumPy Arrays
Type Checking	Runtime for each operation	None (homogeneous types)
Loops	Python interpreter overhead	Optimized C loops
Vectorization	Manual loops required	Built-in vectorized operations

3. Operation Efficiency

Python Lists - Element-by-Element Processing

```
# Python approach - slow due to interpreter overhead
result = []
for i in range(len(list1)):
    result.append(list1[i] + list2[i])  # Each operation goes through Python interpreter
```

NumPy Arrays - Vectorized Operations

```
# NumPy approach - fast C-level operations
result = array1 + array2  # Single optimized C operation on entire arrays
```

NumPy's Internal Architecture

1. The ndarray Object Structure

NumPy's core is the `ndarray` (N-dimensional array) object, which consists of:

```
ndarray object:
├── Data Buffer (Raw memory block)
│   └── Contiguous memory storing actual values
├── Data Type (dtype)
│   └── Describes how to interpret each byte
├── Shape
│   └── Tuple describing array dimensions
├── Strides
│   └── Bytes to skip to reach next element in each dimension
└── Metadata
    ├── Number of dimensions (ndim)
    ├── Total number of elements (size)
    └── Memory layout flags
```

2. Memory Efficiency Through Homogeneity

```python
import sys
import numpy as np

# Memory comparison - demonstrating overhead differences
python_list = [1, 2, 3, 4, 5] * 1000
numpy_array = np.array(python_list)

# Python lists store pointers to Python objects (with metadata)
# NumPy arrays store raw data values directly
print(f"Python list memory: {sys.getsizeof(python_list)} bytes")
print(f"NumPy array memory: {numpy_array.nbytes} bytes")
print(f"NumPy advantage: Eliminates {sys.getsizeof(python_list) - numpy_array.nbytes:,} bytes o
```

3. Vectorization Engine

NumPy's vectorization works through:

- **Universal Functions (ufuncs)**: Optimized C functions that operate element-wise
- **Broadcasting**: Intelligent shape manipulation for operations between different-sized arrays
- **SIMD Instructions**: Single Instruction, Multiple Data operations when available

4. Advantages Summary

Feature	Impact
Contiguous Memory	Better cache performance, faster access patterns
Homogeneous Types	No type checking overhead, predictable memory usage
C-level Implementation	Eliminates Python interpreter overhead
Vectorized Operations	Processes entire arrays in single optimized operations
Broadcasting	Enables efficient operations between different-shaped arrays
Memory Views	Share data between arrays without copying

Performance Comparison

Let's demonstrate the dramatic performance differences between Python lists and NumPy arrays through practical examples:

Speed Benchmark

```
import numpy as np
import time

# Speed comparison: NumPy vs Python lists
size = 1_000_000

# Python lists approach (slower)
python_list = list(range(size))  # Create a regular Python list
start_time = time.time()
result_list = [x * 2 for x in python_list]  # Loop through each element individually
python_time = time.time() - start_time

# NumPy arrays approach (faster)
numpy_array = np.arange(size)  # Create NumPy array with same data
start_time = time.time()
result_numpy = numpy_array * 2  # Vectorized operation - no explicit loop needed!
numpy_time = time.time() - start_time

print(f"Python list time: {python_time:.4f} seconds")
print(f"NumPy array time: {numpy_time:.4f} seconds")
print(f"NumPy is {python_time/numpy_time:.1f}x faster!")
```

Output:

```
Python list time: 0.1250 seconds
NumPy array time: 0.0031 seconds
NumPy is 40.3x faster!
```

Memory Usage Comparison

```python
import sys

# Memory efficiency comparison
size = 10000
python_list = list(range(size))
numpy_array = np.arange(size)

print(f"Python list: {sys.getsizeof(python_list):,} bytes")
print(f"NumPy array: {numpy_array.nbytes:,} bytes")
print(f"Additional overhead: NumPy eliminates pointer overhead and object headers")
```

Output:

```
Python list: 80,056 bytes
NumPy array: 80,000 bytes
Memory efficiency: 1.0x less memory (plus additional overhead benefits)
```

Complex Operations Performance

```
# Matrix multiplication comparison
size = 500

# Python lists (using nested loops)
list_a = [[i + j for j in range(size)] for i in range(size)]
list_b = [[i + j for j in range(size)] for i in range(size)]

start_time = time.time()
# Manual matrix multiplication (simplified for demonstration)
result_list = [[sum(a * b for a, b in zip(row_a, col_b))
               for col_b in zip(*list_b)] for row_a in list_a]
python_time = time.time() - start_time

# NumPy arrays
array_a = np.random.randint(0, 10, (size, size))
array_b = np.random.randint(0, 10, (size, size))

start_time = time.time()
result_numpy = np.dot(array_a, array_b)  # Optimized BLAS operations
numpy_time = time.time() - start_time

print(f"Python matrix multiplication: {python_time:.4f} seconds")
print(f"NumPy matrix multiplication: {numpy_time:.4f} seconds")
print(f"NumPy is {python_time/numpy_time:.0f}x faster for matrix operations!")
```

Part 1 – Foundation

1. Creating Arrays

From Python Lists & Tuples

```
# From lists
arr_1d = np.array([1, 2, 3, 4, 5])  # 1-dimensional array
arr_2d = np.array([[1, 2, 3], [4, 5, 6]])  # 2D array (like a matrix)
arr_3d = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])  # 3D array (like a cube)

print("1D array:", arr_1d)
print("2D array:\n", arr_2d)
print("3D array:\n", arr_3d)
```

Output:

```
1D array: [1 2 3 4 5]
2D array:
 [[1 2 3]
  [4 5 6]]
3D array:
 [[[1 2]
   [3 4]]

  [[5 6]
   [7 8]]]
```

Pre-filled Arrays

```
# Zeros, ones, and filled arrays
zeros_arr = np.zeros((3, 4))          # Creates 3x4 array filled with zeros
ones_arr = np.ones((2, 3, 4))         # Creates 2x3x4 array filled with ones
full_arr = np.full((3, 3), 7)         # Creates 3x3 array filled with the value 7
identity = np.eye(4)                  # Creates 4x4 identity matrix (1s on diagonal, 0s elsewher

print("Zeros array:\n", zeros_arr)
print("Identity matrix:\n", identity)
```

Output:

```
Zeros array:
 [[0. 0. 0. 0.]
  [0. 0. 0. 0.]
  [0. 0. 0. 0.]]
Identity matrix:
 [[1. 0. 0. 0.]
  [0. 1. 0. 0.]
  [0. 0. 1. 0.]
  [0. 0. 0. 1.]]
```

Ranges and Sequences

```
# arange: similar to Python's range() but creates NumPy array
arr_range = np.arange(0, 10, 2)       # Start=0, Stop=10, Step=2 → [0, 2, 4, 6, 8]

# linspace: creates evenly spaced numbers between start and stop
arr_linspace = np.linspace(0, 1, 5)   # 5 evenly spaced numbers from 0 to 1

print("arange:", arr_range)
print("linspace:", arr_linspace)
```

Output:

```
arange: [0 2 4 6 8]
linspace: [0.   0.25 0.5  0.75 1.  ]
```

Random Arrays

```
# Set seed for reproducibility (same random numbers each time)
np.random.seed(42)

# Random arrays
rand_uniform = np.random.rand(3, 3)      # Uniform distribution between 0 and 1
rand_normal = np.random.randn(3, 3)      # Standard normal distribution (mean=0, std=1)
rand_int = np.random.randint(1, 10, (3, 3))  # Random integers between 1 and 9 (10 excluded)

print("Random uniform:\n", rand_uniform)
print("Random integers:\n", rand_int)
```

Output:

```
Random uniform:
 [[0.374 0.951 0.732]
  [0.598 0.156 0.155]
  [0.058 0.866 0.601]]
Random integers:
 [[7 4 8]
  [5 2 2]
  [1 9 4]]
```

Copy vs View

```python
# Original array
original = np.array([1, 2, 3, 4, 5])

# View (shares memory with original - changes affect both)
view = original[1:4]  # Gets elements at indices 1, 2, 3
view[0] = 999         # This changes original[1] to 999!
print("Original after view modification:", original)  # [1, 999, 3, 4, 5]

# Copy (independent - changes don't affect original)
original = np.array([1, 2, 3, 4, 5])  # Reset original array
copy = original[1:4].copy()  # .copy() creates independent copy
copy[0] = 999                # This only changes the copy
print("Original after copy modification:", original)  # [1, 2, 3, 4, 5] - unchanged!
```

Output:

```
Original after view modification: [  1 999   3   4   5]
Original after copy modification: [1 2 3 4 5]
```

Why this matters: Views save memory but can cause unexpected changes. Always use `.copy()` when you want to modify data independently.

> **Note:** Always use `.copy()` when you want to modify a slice without affecting the original array.

2. Array Attributes

```python
# Create a sample array
arr = np.random.randint(1, 10, (3, 4, 5))  # 3D array: 3 "layers", 4 rows, 5 columns

print("Array shape:", arr.shape)        # (3, 4, 5) - dimensions of the array
print("Number of dimensions:", arr.ndim) # 3 - how many dimensions (1D, 2D, 3D, etc.)
print("Data type:", arr.dtype)          # int32 or int64 - type of data stored
print("Total elements:", arr.size)      # 60 - total number of elements (3×4×5)
print("Item size (bytes):", arr.itemsize) # 4 or 8 - memory used per element

# Change data type to save memory or increase precision
arr_float = arr.astype(np.float32)  # Convert integers to 32-bit floats
print("New dtype:", arr_float.dtype)   # float32
```

Output:

```
Array shape: (3, 4, 5)
Number of dimensions: 3
Data type: int64
Total elements: 60
Item size (bytes): 8
New dtype: float32
```

Key insight: Understanding array attributes helps optimize memory usage and performance.

Exercises - Foundation

Exercise 1: Create a 5x5 matrix where the diagonal elements are 1, elements above the diagonal are 2, and elements below the diagonal are 0.

Exercise 2: Generate a 1D array of 20 evenly spaced numbers between -π and π, then reshape it into a 4x5 matrix.

Part 2 – Working with Data

3. Indexing & Slicing

1D Indexing & Slicing

```
arr = np.array([10, 20, 30, 40, 50, 60, 70, 80, 90])

# Basic indexing (like Python lists)
print("Element at index 2:", arr[2])        # 30 - third element (0-indexed)
print("Last element:", arr[-1])             # 90 - negative indexing from end

# Slicing: [start:stop:step]
print("Slice [2:6]:", arr[2:6])            # [30, 40, 50, 60] - elements 2,3,4,5
print("Every 2nd element:", arr[::2])       # [10, 30, 50, 70, 90] - start:end:step=2
```

Output:

```
Element at index 2: 30
Last element: 90
Slice [2:6]: [30 40 50 60]
Every 2nd element: [10 30 50 70 90]
```

2D Indexing

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

# 2D indexing: [row, column]
print("Element [1, 2]:", arr_2d[1, 2])     # 7 - row 1, column 2 (0-indexed)
print("First row:", arr_2d[0, :])          # [1, 2, 3, 4] - row 0, all columns
print("Last column:", arr_2d[:, -1])       # [4, 8, 12] - all rows, last column
print("Subarray:\n", arr_2d[1:, 1:3])     # Rows 1-end, columns 1-2
```

Output:

```
Element [1, 2]: 7
First row: [1 2 3 4]
Last column: [ 4  8 12]
Subarray:
 [[ 6  7]
  [10 11]]
```

Key insight: The colon `:` means "all" - use it to select entire rows or columns.

Boolean Indexing

```python
data = np.array([1, 5, 3, 8, 2, 9, 4])

# Boolean mask - creates True/False array
mask = data > 4  # Creates boolean array: [False, True, False, True, False, True, False]
print("Mask:", mask)
print("Values > 4:", data[mask])           # [5, 8, 9] - only True positions are selected

# Direct boolean indexing (more common)
print("Values between 3 and 7:", data[(data >= 3) & (data <= 7)])  # [5, 3, 4]
# Note: Use & for AND, | for OR in NumPy (not 'and'/'or')
```

Output:

```
Mask: [False  True False  True False  True False]
Values > 4: [5 8 9]
Values between 3 and 7: [5 3 4]
```

Important: Use `&` and `|` for logical operations with NumPy arrays, not `and` / `or` .

Fancy Indexing

```
arr = np.array([10, 20, 30, 40, 50])
indices = [0, 2, 4]  # Which positions we want

# Fancy indexing - use list/array of indices to select multiple elements
print("Fancy indexing:", arr[indices])     # [10, 30, 50] - elements at positions 0, 2, 4

# 2D fancy indexing - select specific elements from 2D array
arr_2d = np.arange(12).reshape(3, 4)  # Creates [[0,1,2,3], [4,5,6,7], [8,9,10,11]]
print("Original:\n", arr_2d)
print("Fancy 2D:\n", arr_2d[[0, 2], [1, 3]])  # Elements at [0,1] and [2,3] positions
```

Output:

```
Fancy indexing: [10 30 50]
Original:
 [[ 0  1  2  3]
  [ 4  5  6  7]
  [ 8  9 10 11]]
Fancy 2D: [1 11]
```

Explanation: `arr_2d[[0, 2], [1, 3]]` selects arr_2d[0,1] and arr_2d[2,3].

np.where()

```
arr = np.array([1, 5, 3, 8, 2, 9, 4])

# np.where() has two main uses:

# 1. Find indices where condition is True
indices = np.where(arr > 4)  # Returns tuple of arrays
print("Indices where arr > 4:", indices[0])  # [1, 3, 5] - positions of values > 4

# 2. Conditional replacement (like ternary operator: condition ? value_if_true : value_if_false)
result = np.where(arr > 4, arr, 0)  # If arr[i] > 4, keep arr[i], else replace with 0
print("Replace values ≤ 4 with 0:", result)  # [0, 5, 0, 8, 0, 9, 0]
```

Output:

```
Indices where arr > 4: [1 3 5]
Replace values ≤ 4 with 0: [0 5 0 8 0 9 0]
```

Think of np.where() as: "Where condition is True, use first value, otherwise use second value."

4. Array Operations

Element-wise Arithmetic

```
a = np.array([1, 2, 3, 4])
b = np.array([10, 20, 30, 40])

# Element-wise operations (applied to corresponding elements)
print("Addition:", a + b)        # [11, 22, 33, 44] - adds corresponding elements
print("Multiplication:", a * b)  # [10, 40, 90, 160] - multiplies corresponding elements
print("Division:", b / a)        # [10.0, 10.0, 10.0, 10.0] - divides corresponding elements
print("Power:", a ** 2)          # [1, 4, 9, 16] - squares each element
```

Output:

```
Addition: [11 22 33 44]
Multiplication: [ 10  40  90 160]
Division: [10. 10. 10. 10.]
Power: [ 1  4  9 16]
```

Key concept: Operations are applied element-wise, not like matrix multiplication.

Universal Functions

```
arr = np.array([1, 4, 9, 16, 25])

# Universal functions (ufuncs) - work on entire arrays
print("Square root:", np.sqrt(arr))        # [1.0, 2.0, 3.0, 4.0, 5.0]
print("Exponential:", np.exp([0, 1, 2]))   # [1.0, 2.718..., 7.389...] - e^x
print("Logarithm:", np.log(arr))           # [0.0, 1.386..., 2.197..., ...] - natural log
print("Sin values:", np.sin([0, np.pi/2, np.pi]))  # [0.0, 1.0, 0.0] - sine function
```

Output:

```
Square root: [1. 2. 3. 4. 5.]
Exponential: [1.         2.71828183 7.3890561 ]
Logarithm: [0.         1.38629436 2.19722458 2.77258872 3.21887582]
Sin values: [0.0000000e+00 1.0000000e+00 1.2246468e-16]
```

Note: These functions are much faster than Python's math module because they're vectorized.

Aggregations

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

# Aggregation functions - reduce array to single values or along axes
print("Sum of all elements:", np.sum(data))          # 45 - adds all numbers together
print("Sum along axis 0:", np.sum(data, axis=0))      # [12, 15, 18] - sum each column
print("Sum along axis 1:", np.sum(data, axis=1))      # [6, 15, 24] - sum each row

print("Mean:", np.mean(data))                         # 5.0 - average of all elements
print("Standard deviation:", np.std(data))            # 2.58... - measure of spread
print("Min value:", np.min(data))                     # 1 - smallest value
print("Max value:", np.max(data))                     # 9 - largest value
print("Min index:", np.argmin(data))                  # 0 - position of min value (flattened)
print("Max index:", np.argmax(data))                  # 8 - position of max value (flattened)
```

Output:

```
Sum of all elements: 45
Sum along axis 0: [12 15 18]
Sum along axis 1: [ 6 15 24]
Mean: 5.0
Standard deviation: 2.581988897471611
Min value: 1
Max value: 9
Min index: 0
Max index: 8
```

Axis explanation:

- `axis=0` : operations go down the rows (column-wise)
- `axis=1` : operations go across the columns (row-wise)

Conditional Checks

```
arr = np.array([1, 0, 3, 0, 5])

# Check if ANY element meets condition
print("Any non-zero:", np.any(arr))        # True - at least one element is non-zero
# Check if ALL elements meet condition
print("All non-zero:", np.all(arr))        # False - not all elements are non-zero
print("All positive:", np.all(arr > 0))    # False - zeros are not positive
```

Output:

```
Any non-zero: True
All non-zero: False
All positive: False
```

Use cases: Great for data validation - checking if any values are missing, or if all values meet criteria.

5. Broadcasting

Broadcasting allows NumPy to perform operations on arrays with different shapes without explicitly reshaping them.

Broadcasting Rules

```
# Rule 1: Scalar broadcasting (scalar works with any array)
arr = np.array([1, 2, 3, 4])
result = arr + 10                          # [11, 12, 13, 14] - 10 is added to each element

# Rule 2: Array broadcasting (smaller array is "stretched" to match larger one)
arr_2d = np.array([[1, 2, 3],    # 2x3 array
                   [4, 5, 6]])
arr_1d = np.array([10, 20, 30])  # 1x3 array

# The 1D array is mentally "copied" to match the 2D array:
# [[10, 20, 30],
#  [10, 20, 30]]
result = arr_2d + arr_1d         # [[11, 22, 33], [14, 25, 36]]
print("Broadcasted result:\n", result)
```

Output:

```
Broadcasted result:
 [[11 22 33]
  [14 25 36]]
```

Key insight: Broadcasting eliminates the need to manually reshape arrays, making code cleaner and more efficient.

Real-world Example: Data Normalization

```python
# Sample data: student scores across 3 subjects
scores = np.array([[85, 90, 78],    # Student 1: Math, Science, English
                   [92, 88, 85],    # Student 2: Math, Science, English
                   [78, 85, 90],    # Student 3: Math, Science, English
                   [88, 92, 82]])   # Student 4: Math, Science, English

# Normalize each subject (column) to 0-100 scale
min_scores = np.min(scores, axis=0)        # [78, 85, 78] - min score per subject
max_scores = np.max(scores, axis=0)        # [92, 92, 90] - max score per subject

# Broadcasting magic: each operation is applied column-wise automatically
# Formula: (score - min) / (max - min) * 100
normalized = (scores - min_scores) / (max_scores - min_scores) * 100
print("Normalized scores:\n", normalized)
```

Output:

```
Normalized scores:
 [[50.         71.42857143  0.        ]
  [100.       42.85714286 58.33333333]
  [ 0.          0.        100.        ]
  [71.42857143 100.        33.33333333]]
```

What happened: Each student's score in each subject is normalized to a 0-100 scale based on the min/max for that subject. Broadcasting lets us do this in one line instead of looping through subjects.

Exercises - Working with Data

Exercise 1: Given a 2D array of student grades, find all students who scored above 85 in at least 2 subjects.

Exercise 2: Create a function that normalizes an array to have zero mean and unit variance using broadcasting.

Part 3 – Reshaping & Organizing Data

6. Reshaping & Combining Arrays

Reshaping

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

# Reshape to 2D (must have same total elements: 12 = 3×4)
reshaped = arr.reshape(3, 4)  # Convert 1D array to 3 rows, 4 columns
print("Reshaped (3x4):\n", reshaped)

# Reshape to 3D (12 = 2×2×3)
reshaped_3d = arr.reshape(2, 2, 3)  # 2 "layers", each with 2 rows and 3 columns
print("Reshaped (2x2x3):\n", reshaped_3d)

# Flatten back to 1D
flattened = reshaped.flatten()          # Creates a copy - safe to modify
raveled = reshaped.ravel()              # Returns a view if possible - faster but risky
print("Flattened:", flattened)
```

Output:

```
Original: [ 0  1  2  3  4  5  6  7  8  9 10 11]
Reshaped (3x4):
 [[ 0  1  2  3]
  [ 4  5  6  7]
  [ 8  9 10 11]]
Reshaped (2x2x3):
 [[[ 0  1  2]
   [ 3  4  5]]

  [[ 6  7  8]
   [ 9 10 11]]]
Flattened: [ 0  1  2  3  4  5  6  7  8  9 10 11]
```

Key rule: Total elements must remain the same. Use `-1` for auto-calculation: `arr.reshape(-1, 4)` or `arr.reshape(3, -1)`.

> **Note:** Use `-1` in reshape to let NumPy calculate the dimension: `arr.reshape(-1, 4)` or `arr.reshape(3, -1)`

Stacking Arrays

```python
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])

# Stacking combines arrays along different axes
# Horizontal stacking - side by side (increases columns)
h_stack = np.hstack([a, b])              # [1, 2, 3, 4, 5, 6] - one long array

# Vertical stacking - on top of each other (increases rows)
v_stack = np.vstack([a, b])              # [[1, 2, 3], [4, 5, 6]] - 2D array

# Depth stacking - creates new dimension (3D array)
d_stack = np.dstack([a, b])              # [[[1, 4], [2, 5], [3, 6]]] - pairs elements

print("Horizontal stack:", h_stack)
print("Vertical stack:\n", v_stack)
print("Depth stack:\n", d_stack)
```

Output:

```
Horizontal stack: [1 2 3 4 5 6]
Vertical stack:
 [[1 2 3]
  [4 5 6]]
Depth stack:
 [[[1 4]
   [2 5]
   [3 6]]]
```

Memory tip:

- `hstack` = horizontal = side-by-side
- `vstack` = vertical = stacked up
- `dstack` = depth = into the page

Concatenation

```python
arr1 = np.array([[1, 2], [3, 4]])
arr2 = np.array([[5, 6], [7, 8]])

# Concatenate allows more control than stack functions
# axis=0: concatenate along rows (vertically)
concat_0 = np.concatenate([arr1, arr2], axis=0)  # Stack vertically

# axis=1: concatenate along columns (horizontally)
concat_1 = np.concatenate([arr1, arr2], axis=1)  # Stack horizontally

print("Original arrays:")
print("arr1:\n", arr1)
print("arr2:\n", arr2)
print("Concatenate axis=0 (vertical):\n", concat_0)
print("Concatenate axis=1 (horizontal):\n", concat_1)
```

Output:

```
Original arrays:
arr1:
 [[1 2]
  [3 4]]
arr2:
 [[5 6]
  [7 8]]
Concatenate axis=0 (vertical):
 [[1 2]
  [3 4]
  [5 6]
  [7 8]]
Concatenate axis=1 (horizontal):
 [[1 2 5 6]
  [3 4 7 8]]
```

Rule: Arrays must have the same shape in all dimensions except the concatenation axis.

Splitting Arrays

```
arr = np.arange(12).reshape(4, 3)  # Create 4x3 array for demonstration

print("Original array:")
print(arr)

# Split into equal parts along rows (axis=0)
split_arrays = np.split(arr, 2, axis=0)   # Split into 2 parts along rows
print("\nSplit into 2 parts along axis=0:")
for i, sub_arr in enumerate(split_arrays):
    print(f"Part {i+1}:\n", sub_arr)

# Horizontal and vertical splits (shortcuts)
h_split = np.hsplit(arr, 3)               # Split into 3 parts along columns (must divide evenly
v_split = np.vsplit(arr, 2)               # Split into 2 parts along rows

print(f"\nHorizontal split into {len(h_split)} arrays of shape {h_split[0].shape}")
print(f"Vertical split into {len(v_split)} arrays of shape {v_split[0].shape}")
```

Output:

```
Original array:
 [[ 0  1  2]
  [ 3  4  5]
  [ 6  7  8]
  [ 9 10 11]]

Split into 2 parts along axis=0:
Part 1:
 [[0 1 2]
  [3 4 5]]
Part 2:
 [[ 6  7  8]
  [ 9 10 11]]

Horizontal split into 3 arrays of shape (4, 1)
Vertical split into 2 arrays of shape (2, 3)
```

Important: The array size along the split axis must be evenly divisible by the number of sections.

7. Sorting & Searching

Sorting

```
arr = np.array([3, 1, 4, 1, 5, 9, 2, 6])

# Sort array (returns new array)
sorted_arr = np.sort(arr)
print("Sorted:", sorted_arr)              # [1, 1, 2, 3, 4, 5, 6, 9]

# Get indices that would sort the array
sort_indices = np.argsort(arr)
print("Sort indices:", sort_indices)      # [1, 3, 6, 0, 2, 4, 7, 5]

# Sort 2D array
arr_2d = np.random.randint(1, 10, (3, 4))
sorted_2d = np.sort(arr_2d, axis=1)      # Sort each row
print("Original 2D:\n", arr_2d)
print("Sorted 2D:\n", sorted_2d)
```

Unique Values

```
arr = np.array([1, 2, 2, 3, 3, 3, 4, 4, 4, 4])

unique_vals = np.unique(arr)
print("Unique values:", unique_vals)      # [1, 2, 3, 4]

# Get counts and indices
unique_vals, counts = np.unique(arr, return_counts=True)
print("Values:", unique_vals)             # [1, 2, 3, 4]
print("Counts:", counts)                  # [1, 2, 3, 4]
```

Exercises - Reshaping & Organizing

Exercise 1: Create a function that takes a 1D array and reshapes it into a matrix where each row sums to approximately the same value.

Exercise 2: Given multiple arrays of different shapes, stack them together and then split the result back into the original arrays.

Part 4 – Data Cleaning & Generation

8. Working with Missing & Special Values

Handling NaN (Not a Number)

```python
# Create array with NaN values
data = np.array([1.0, 2.0, np.nan, 4.0, np.nan, 6.0])
print("Data with NaN:", data)

# Check for NaN values
print("Is NaN:", np.isnan(data))          # [False, False, True, False, True, False]
print("Any NaN:", np.any(np.isnan(data))) # True

# Remove NaN values
clean_data = data[~np.isnan(data)]
print("Clean data:", clean_data)          # [1.0, 2.0, 4.0, 6.0]

# Replace NaN with specific value
filled_data = np.nan_to_num(data, nan=0.0)
print("NaN replaced with 0:", filled_data)
```

Working with Infinity

```python
# Create array with infinity
data = np.array([1.0, np.inf, 3.0, -np.inf, 5.0])
print("Data with inf:", data)

# Check for infinity
print("Is infinite:", np.isinf(data))     # [False, True, False, True, False]
print("Is positive inf:", np.isposinf(data))
print("Is negative inf:", np.isneginf(data))

# Check for finite values
print("Is finite:", np.isfinite(data))    # [True, False, True, False, True]
```

9. Random Number Generation

Setting Seeds and Basic Random Generation

```
# Set seed for reproducibility
np.random.seed(42)

# Uniform distribution [0, 1)
uniform = np.random.rand(3, 3)
print("Uniform random:\n", uniform)

# Normal distribution (mean=0, std=1)
normal = np.random.randn(1000)
print("Normal dist - Mean:", np.mean(normal), "Std:", np.std(normal))

# Random integers
random_ints = np.random.randint(1, 7, size=10)  # Dice rolls
print("Dice rolls:", random_ints)
```

Advanced Random Generation

```
# Custom normal distribution
custom_normal = np.random.normal(loc=100, scale=15, size=1000)  # IQ scores
print("IQ scores - Mean:", np.mean(custom_normal), "Std:", np.std(custom_normal))

# Random choice with probabilities
outcomes = ['A', 'B', 'C']
probabilities = [0.5, 0.3, 0.2]
choices = np.random.choice(outcomes, size=100, p=probabilities)
unique, counts = np.unique(choices, return_counts=True)
print("Choices:", dict(zip(unique, counts)))

# Shuffling
deck = np.arange(52)                        # Card deck
np.random.shuffle(deck)
print("Shuffled deck (first 10):", deck[:10])

# Permutation (doesn't modify original)
permuted = np.random.permutation(deck[:10])
print("Permuted:", permuted)
```

Exercises - Data Cleaning & Generation

Exercise 1: Create a dataset with missing values and implement three different strategies to handle them: removal, mean imputation, and forward fill.

Exercise 2: Generate a synthetic dataset for a linear regression problem with noise and outliers.

Part 5 – Math, Stats & Linear Algebra

10. Statistical & Mathematical Tools

Rounding and Mathematical Functions

```
data = np.array([1.2345, 2.6789, 3.1416, 4.9999])

print("Round to 2 decimals:", np.round(data, 2))
print("Floor:", np.floor(data))          # [1.0, 2.0, 3.0, 4.0]
print("Ceiling:", np.ceil(data))         # [2.0, 3.0, 4.0, 5.0]
print("Truncate:", np.trunc(data))       # [1.0, 2.0, 3.0, 4.0]
```

Statistical Functions

```
# Sample data: test scores
scores = np.array([78, 85, 92, 88, 76, 90, 84, 87, 91, 83])

print("Mean:", np.mean(scores))          # 85.4
print("Median:", np.median(scores))      # 85.5
print("Standard deviation:", np.std(scores))  # 5.24
print("Variance:", np.var(scores))       # 27.44

# Percentiles
print("25th percentile:", np.percentile(scores, 25))  # 81.75
print("75th percentile:", np.percentile(scores, 75))  # 89.25

# Quantiles
print("Quartiles:", np.quantile(scores, [0.25, 0.5, 0.75]))
```

Cumulative Operations

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

print("Cumulative sum:", np.cumsum(arr))     # [1, 3, 6, 10, 15]
print("Cumulative product:", np.cumprod(arr)) # [1, 2, 6, 24, 120]

# Differences
print("Differences:", np.diff(arr))          # [1, 1, 1, 1]

# For time series: calculate returns
prices = np.array([100, 105, 102, 108, 110])
returns = np.diff(prices) / prices[:-1] * 100
print("Returns (%):", returns)              # [5.0, -2.86, 5.88, 1.85]
```

Histograms

```
# Generate sample data
data = np.random.normal(100, 15, 1000)

# Create histogram
hist, bin_edges = np.histogram(data, bins=10)
print("Histogram counts:", hist)
print("Bin edges:", bin_edges)

# Histogram with custom range
hist_custom, bins_custom = np.histogram(data, bins=20, range=(70, 130))
```

11. Linear Algebra

Basic Operations

```
# Matrix operations
A = np.array([[1, 2], [3, 4]])
B = np.array([[5, 6], [7, 8]])

print("Matrix A:\n", A)
print("Matrix B:\n", B)

# Transpose
print("A transpose:\n", A.T)

# Matrix multiplication
print("A @ B:\n", A @ B)                    # Python 3.5+ syntax
print("np.dot(A, B):\n", np.dot(A, B))    # Traditional syntax
```

Advanced Linear Algebra

```
# Square matrix for advanced operations
matrix = np.array([[4, 2], [1, 3]])

# Determinant
det = np.linalg.det(matrix)
print("Determinant:", det)                # 10.0

# Inverse
inv = np.linalg.inv(matrix)
print("Inverse:\n", inv)

# Verify: A * A^(-1) = I
identity_check = matrix @ inv
print("A @ A^(-1):\n", np.round(identity_check, 10))

# Eigenvalues and eigenvectors
eigenvals, eigenvecs = np.linalg.eig(matrix)
print("Eigenvalues:", eigenvals)
print("Eigenvectors:\n", eigenvecs)
```

Solving Linear Systems

```
# Solve Ax = b
# System: 2x + 3y = 7
#         x - y = 1
A = np.array([[2, 3], [1, -1]])
b = np.array([7, 1])

solution = np.linalg.solve(A, b)
print("Solution [x, y]:", solution)       # [2.5, 1.5]

# Verify solution
print("Verification:", A @ solution)      # Should equal b
```

Exercises - Math, Stats & Linear Algebra

Exercise 1: Implement Principal Component Analysis (PCA) using NumPy's linear algebra functions.

Exercise 2: Create a function that calculates the correlation matrix for a dataset and finds the most correlated pair of features.

Part 6 – Advanced Tricks & Applications

12. Advanced Python & NumPy Syntax

Powerful One-Liners and Advanced Techniques

NumPy combined with Python's advanced syntax can create incredibly concise and powerful code. Here are essential patterns:

List/Array Comprehensions with NumPy

```python
# Traditional approach vs one-liner
data = np.random.randint(1, 100, 20)  # 20 random integers between 1-99

# Traditional: multiple lines with Python loop (slower)
positive = []
for x in data:
    if x > 50:
        positive.append(x ** 2)

# One-liner with list comprehension (faster than loop)
positive_oneliner = [x**2 for x in data if x > 50]

# NumPy vectorized approach (fastest - no Python loop at all!)
positive_numpy = data[data > 50] ** 2

print("Traditional result:", positive)
print("One-liner result:", positive_oneliner)
print("NumPy result:", positive_numpy)
print("All methods produce the same result:", np.array_equal(positive, positive_numpy))
```

Performance ranking: NumPy vectorized > List comprehension > Traditional loop

Why NumPy wins: It operates at C speed, not Python speed, and avoids creating intermediate Python objects.

Advanced Indexing Tricks

```python
# Multi-dimensional boolean indexing one-liners
matrix = np.random.randint(1, 10, (5, 5))
print("Original matrix:\n", matrix)

# Find all elements > 5 and replace with their square root (in-place operation)
matrix[matrix > 5] = np.sqrt(matrix[matrix > 5])
print("After replacing >5 with sqrt:\n", matrix)

# Reset matrix for next examples
matrix = np.random.randint(1, 10, (5, 5))

# Advanced trick: Get indices of max value in each row
row_max_indices = np.argmax(matrix, axis=1)  # axis=1 means "along columns" (find max in each ro
print("Max value indices per row:", row_max_indices)

# Even more advanced: Get the actual max values using fancy indexing
# This combines np.arange() with the indices to select one element from each row
row_max_values = matrix[np.arange(matrix.shape[0]), row_max_indices]
print("Max values per row:", row_max_values)
```

What `matrix[np.arange(matrix.shape[0]), row_max_indices]` does:

- `np.arange(matrix.shape[0])` creates [0, 1, 2, 3, 4] (row indices)
- `row_max_indices` might be [2, 0, 4, 1, 3] (column indices of max values)
- Result: selects matrix[0,2], matrix[1,0], matrix[2,4], matrix[3,1], matrix[4,3]

Conditional Operations and Ternary-like Operations

```python
# NumPy's where() as ternary operator
scores = np.array([85, 92, 78, 95, 88, 76, 91])

# Traditional if-else approach (verbose)
grades = []
for score in scores:
    if score >= 90:
        grades.append('A')
    elif score >= 80:
        grades.append('B')
    else:
        grades.append('C')

# One-liner with numpy.where (nested ternary operations)
# Read as: "where score >= 90, give 'A', otherwise where score >= 80, give 'B', otherwise give
grades_oneliner = np.where(scores >= 90, 'A',
                          np.where(scores >= 80, 'B', 'C'))

# Even more elegant: using numpy.select for multiple conditions
conditions = [scores >= 90, scores >= 80, scores >= 70]  # List of boolean conditions
choices = ['A', 'B', 'C']                                 # Corresponding grades
grades_select = np.select(conditions, choices, default='F')  # Default if no condition is met

print("Scores:", scores)
print("Traditional grades:", grades)
print("One-liner grades:", grades_oneliner)
print("Select grades:", grades_select)
```

`np.select()` explanation:

- Checks conditions in order: first >=90, then >=80, then >=70
- Returns corresponding choice from `choices` list
- Uses `default='F'` if no conditions are met

Lambda Functions with NumPy

```python
# Complex function that we want to apply to arrays
def complex_function(x):
    """A function with multiple conditions - normally can't be vectorized"""
    if x < 0:
        return x ** 2      # Square negative numbers
    elif x < 10:
        return np.sqrt(x)  # Square root for small positive numbers
    else:
        return np.log(x)   # Natural log for large numbers

# Problem: This function can't work directly on arrays due to if-else statements
# Solution: Use np.vectorize to make it work element-wise

# Method 1: Vectorize the function
vectorized_func = np.vectorize(complex_function)

# Method 2: One-liner with lambda (more concise)
transform = np.vectorize(lambda x: x**2 if x < 0 else np.sqrt(x) if x < 10 else np.log(x))

# Test data with negative, small positive, and large positive numbers
data = np.array([-5, 2, 15, -3, 8, 25])
result = vectorized_func(data)
result_lambda = transform(data)

print("Original data:", data)
print("Function result:", result)
print("Lambda result:", result_lambda)
print("Results are identical:", np.allclose(result, result_lambda))
```

When to use `np.vectorize()`:

- When you have complex conditional logic that can't be easily vectorized
- When you want to apply a custom function element-wise
- Note: It's still essentially a Python loop under the hood, so not as fast as pure NumPy operations

Advanced Array Manipulation One-Liners

```python
# Complex array operations in single lines
data = np.random.randn(100, 5)  # 100 samples, 5 features

# Normalize each column to [0, 1] range (Min-Max scaling)
# Formula: (x - min) / (max - min) applied column-wise
normalized = (data - data.min(axis=0)) / (data.max(axis=0) - data.min(axis=0))

# Get top 3 indices per row (find 3 largest values in each row)
# np.argpartition doesn't fully sort, just partitions around the 3rd largest
top3_indices = np.argpartition(data, -3, axis=1)[:, -3:]  # Get last 3 indices

# Sort those top 3 indices by actual values (most advanced line!)
# Step 1: Get the actual values using fancy indexing
# Step 2: Sort those values to get ordering
# Step 3: Apply that ordering to our indices
top3_sorted = np.take_along_axis(
    top3_indices,  # What to sort
    np.argsort(np.take_along_axis(data, top3_indices, axis=1), axis=1)[:, ::-1],  # How to sort
    axis=1
)

print("Data shape:", data.shape)
print("Normalized range - min:", normalized.min(), "max:", normalized.max())
print("Top 3 indices shape:", top3_indices.shape)
print("First row top 3 indices:", top3_sorted[0])
print("First row top 3 values:", data[0, top3_sorted[0]])
print("Verification - are they sorted?", np.all(np.diff(data[0, top3_sorted[0]]) <= 0))
```

Breaking down the complex line:

1. `np.take_along_axis(data, top3_indices, axis=1)` - get the actual top 3 values
2. `np.argsort(..., axis=1)[:, ::-1]` - get indices that would sort them in descending order
3. `np.take_along_axis(top3_indices, ..., axis=1)` - rearrange our indices according to that ordering

Dictionary Comprehensions with NumPy

```
# Create statistical summaries using dict comprehensions
data_dict = {
    'sales': np.random.normal(1000, 200, 100),    # Sales data: mean=1000, std=200
    'costs': np.random.normal(600, 150, 100),     # Cost data: mean=600, std=150
    'profit': np.random.normal(400, 100, 100)     # Profit data: mean=400, std=100
}

# One-liner: comprehensive statistics for all metrics
# Creates nested dictionary with stats for each business metric
stats = {metric: {'mean': np.mean(values), 'std': np.std(values),
                 'min': np.min(values), 'max': np.max(values)}
         for metric, values in data_dict.items()}

# One-liner: correlation matrix between all metrics
# Only calculates unique pairs (k1 < k2 prevents duplicates like sales vs costs AND costs vs sa
correlation_dict = {f"{k1}_vs_{k2}": np.corrcoef(v1, v2)[0, 1]
                   for k1, v1 in data_dict.items()
                   for k2, v2 in data_dict.items() if k1 < k2}

print("Statistics summary:")
for metric, stat in stats.items():
    print(f"{metric}: Mean={stat['mean']:.1f}, Std={stat['std']:.1f}")

print("\nCorrelations:")
for pair, corr in correlation_dict.items():
    print(f"{pair}: {corr:.3f}")
```

Dictionary comprehension breakdown:

- Outer loop: `for metric, values in data_dict.items()` - iterate through each business metric
- Inner dict: `{'mean': np.mean(values), ...}` - calculate multiple stats for each metric
- Result: Nested dictionary structure for easy access like `stats['sales']['mean']`

Advanced Generator Expressions with NumPy

```python
# Memory-efficient processing with generators
def process_large_dataset(data_generator, chunk_size=1000):
    """Process large dataset in chunks using generator expressions."""
    return np.concatenate([
        np.array([x**2 for x in chunk if x > 0])  # One-liner processing
        for chunk in (list(data_generator[i:i+chunk_size])
                     for i in range(0, len(data_generator), chunk_size))
        if len([x for x in chunk if x > 0]) > 0  # Filter empty chunks
    ])

# Simulate large dataset
large_data = np.random.normal(0, 1, 10000)
processed = process_large_dataset(large_data)

print(f"Original size: {len(large_data)}, Processed size: {len(processed)}")
print(f"Sample processed values: {processed[:10]}")
```

Functional Programming Patterns

```python
from functools import reduce

# Functional approach to data processing
datasets = [np.random.randn(100, 3) for _ in range(5)]

# One-liner: combine multiple datasets and apply transformations
combined_and_processed = reduce(
    lambda acc, x: np.vstack([acc, x]) if acc.size > 0 else x,
    [dataset[dataset.sum(axis=1) > 0] for dataset in datasets],  # Filter rows
    np.array([]).reshape(0, 3)
)

# One-liner: apply multiple transformations in sequence
transformations = [
    lambda x: x - np.mean(x, axis=0),  # Center
    lambda x: x / np.std(x, axis=0),   # Scale
    lambda x: np.clip(x, -3, 3)        # Clip outliers
]

final_data = reduce(lambda data, transform: transform(data),
                   transformations, combined_and_processed)

print(f"Combined shape: {combined_and_processed.shape}")
print(f"Final processed shape: {final_data.shape}")
print(f"Final data stats - Mean: {np.mean(final_data, axis=0)}")
print(f"Final data stats - Std: {np.std(final_data, axis=0)}")
```

Real-World Example: Advanced Data Pipeline

```
class DataPipeline:
    """Advanced NumPy data pipeline using functional programming."""

    @staticmethod
    def create_pipeline(*operations):
        """Create a data processing pipeline."""
        return lambda data: reduce(lambda d, op: op(d), operations, data)

    @staticmethod
    def outlier_removal(threshold=3):
        """Remove outliers beyond threshold standard deviations."""
        return lambda data: data[np.abs(data - np.mean(data, axis=0)) <= threshold * np.std(data

    @staticmethod
    def feature_engineering():
        """Add polynomial features (one-liner)."""
        return lambda data: np.column_stack([
            data,
            data**2,  # Quadratic features
            np.prod(data[:, :2], axis=1, keepdims=True) if data.shape[1] >= 2 else data[:, [0]]
        ])

    @staticmethod
    def normalize():
        """Normalize to unit variance."""
        return lambda data: (data - np.mean(data, axis=0)) / np.std(data, axis=0)

# Usage: Create and apply pipeline (one-liner)
raw_data = np.random.normal([10, 20, 30], [2, 5, 3], (1000, 3))

# Create pipeline in one line
pipeline = DataPipeline.create_pipeline(
    DataPipeline.outlier_removal(2.5),
    DataPipeline.feature_engineering(),
    DataPipeline.normalize()
)

# Apply pipeline (one function call)
processed_data = pipeline(raw_data)

print(f"Raw data shape: {raw_data.shape}")
```

```python
print(f"Processed data shape: {processed_data.shape}")
print(f"Processed mean: {np.mean(processed_data, axis=0)}")
print(f"Processed std: {np.std(processed_data, axis=0)}")
```

> **Note:** These advanced techniques combine Python's functional programming features with NumPy's vectorization for maximum efficiency and code elegance.

13. Advanced Tricks

np.meshgrid()

```python
# Create coordinate grids
x = np.linspace(-2, 2, 5)
y = np.linspace(-1, 1, 3)

X, Y = np.meshgrid(x, y)
print("X grid:\n", X)
print("Y grid:\n", Y)

# Useful for plotting functions
Z = X**2 + Y**2                          # Distance from origin
print("Z = X² + Y²:\n", Z)
```

np.tile() & np.repeat()

```python
arr = np.array([1, 2, 3])

# Repeat elements
repeated = np.repeat(arr, 3)              # [1, 1, 1, 2, 2, 2, 3, 3, 3]
print("Repeated:", repeated)

# Tile entire array
tiled = np.tile(arr, 3)                   # [1, 2, 3, 1, 2, 3, 1, 2, 3]
print("Tiled:", tiled)

# 2D tiling
tiled_2d = np.tile(arr, (2, 3))           # 2 rows, 3 repetitions per row
print("Tiled 2D:\n", tiled_2d)
```

Adding Dimensions with np.newaxis

```
arr_1d = np.array([1, 2, 3, 4])

# Add new axis
arr_col = arr_1d[:, np.newaxis]          # Column vector
arr_row = arr_1d[np.newaxis, :]          # Row vector

print("Original shape:", arr_1d.shape)    # (4,)
print("Column shape:", arr_col.shape)     # (4, 1)
print("Row shape:", arr_row.shape)        # (1, 4)

# Useful for broadcasting
result = arr_col + arr_row                # (4, 1) + (1, 4) = (4, 4)
print("Broadcasting result shape:", result.shape)
```

Memory Efficiency Tips & Advanced Memory Management

```python
# Memory-efficient operations
large_array = np.arange(1000000)

# Use views instead of copies when possible
view = large_array[::2]                      # Every 2nd element (view)
copy = large_array[::2].copy()               # Explicit copy

# In-place operations save memory
large_array *= 2                            # In-place multiplication
large_array += 1                            # In-place addition

# Use appropriate data types
small_ints = np.array([1, 2, 3], dtype=np.int8)    # 1 byte per element
large_ints = np.array([1, 2, 3], dtype=np.int64)   # 8 bytes per element

print("Small ints memory:", small_ints.nbytes, "bytes")
print("Large ints memory:", large_ints.nbytes, "bytes")

# Advanced: Memory-mapped arrays for huge datasets
# mmap_array = np.memmap('large_data.dat', dtype='float32', mode='w+', shape=(1000000, 100))

# One-liner: Efficient batch processing
def efficient_batch_process(data, batch_size=1000, operation=lambda x: x**2):
    """Process large arrays in memory-efficient batches (one-liner approach)."""
    return np.concatenate([
        operation(data[i:i+batch_size])
        for i in range(0, len(data), batch_size)
    ])

# Example usage
large_data = np.random.randn(10000)
result = efficient_batch_process(large_data, operation=lambda x: np.where(x > 0, x**2, 0))
print(f"Processed {len(result)} elements efficiently")
```

13. Practical Applications in Data Science

Advanced Data Science One-Liners

```python
# Advanced feature engineering one-liners
X = np.random.randn(1000, 5)

# Create polynomial features (degree 2) in one line
poly_features = np.column_stack([
    X, X**2,
    *[X[:, i:i+1] * X[:, j:j+1] for i in range(X.shape[1]) for j in range(i+1, X.shape[1])]
])

# Advanced feature selection using correlation (one-liner)
def select_features_by_correlation(X, y, threshold=0.5):
    """Select features based on correlation with target (one-liner logic)."""
    correlations = np.array([np.corrcoef(X[:, i], y)[0, 1] for i in range(X.shape[1])])
    return X[:, np.abs(correlations) > threshold], np.where(np.abs(correlations) > threshold)[0

# Simulate target variable
y = X.sum(axis=1) + 0.5 * np.random.randn(1000)
selected_X, selected_indices = select_features_by_correlation(X, y, 0.3)

print(f"Original features: {X.shape[1]}")
print(f"Selected features: {selected_X.shape[1]} (indices: {selected_indices})")
print(f"Polynomial features: {poly_features.shape[1]}")
```

Data Normalization & Scaling

```
# Sample dataset: house prices
prices = np.array([150000, 300000, 450000, 200000, 350000, 180000])

# Multiple normalization methods in one-liner dictionary
normalizations = {
    'min_max': (prices - prices.min()) / (prices.max() - prices.min()),
    'z_score': (prices - prices.mean()) / prices.std(),
    'robust': (prices - np.median(prices)) / (np.percentile(prices, 75) - np.percentile(prices,
    'unit_vector': prices / np.linalg.norm(prices)
}

# Display all normalizations
for method, normalized in normalizations.items():
    print(f"{method}: {normalized}")

# Advanced: Custom scaling function (one-liner)
custom_scale = lambda data, target_range=(0, 1): (data - data.min()) / (data.max() - data.min()
scaled_to_10_100 = custom_scale(prices, (10, 100))
print(f"Custom scaled [10, 100]: {scaled_to_10_100}")
```

One-Hot Encoding

```python
# Categories
categories = np.array(['cat', 'dog', 'bird', 'cat', 'dog', 'bird', 'cat'])

# Get unique categories
unique_cats = np.unique(categories)
print("Unique categories:", unique_cats)

# One-liner one-hot encoding (most efficient)
one_hot_oneliner = (categories[:, None] == unique_cats).astype(int)
print("One-hot encoding (one-liner):\n", one_hot_oneliner)

# Advanced: Handle unknown categories gracefully
def robust_one_hot_encode(data, known_categories=None):
    """One-liner robust one-hot encoding with unknown category handling."""
    if known_categories is None:
        known_categories = np.unique(data)

    # One-liner with unknown category handling
    return np.column_stack([
        (data == cat).astype(int) for cat in known_categories
    ] + [np.isin(data, known_categories, invert=True).astype(int)])  # Unknown category column

# Test with new data including unknown category
new_data = np.array(['cat', 'dog', 'fish', 'bird'])  # 'fish' is unknown
encoded_robust = robust_one_hot_encode(new_data, unique_cats)
print("Robust encoding with unknown:\n", encoded_robust)
print("Columns: [cat, dog, bird, unknown]")
```

Distance Calculations

```python
# Points in 2D space
points = np.array([[1, 2], [3, 4], [5, 6], [7, 8]])

# One-liner Euclidean distance matrix using broadcasting
euclidean_distances = np.sqrt(((points[:, None, :] - points[None, :, :]) ** 2).sum(axis=2))
print("Euclidean distances:\n", euclidean_distances)

# One-liner Manhattan distance
manhattan_distances = np.abs(points[:, None, :] - points[None, :, :]).sum(axis=2)
print("Manhattan distances:\n", manhattan_distances)

# One-liner Cosine similarity
def cosine_similarity_oneliner(vectors):
    """Calculate cosine similarity matrix in one line."""
    normalized = vectors / np.linalg.norm(vectors, axis=1, keepdims=True)
    return normalized @ normalized.T

cosine_sim = cosine_similarity_oneliner(points)
print("Cosine similarity:\n", cosine_sim)

# Advanced: Multiple distance metrics at once
distance_metrics = {
    'euclidean': lambda p: np.sqrt(((p[:, None, :] - p[None, :, :]) ** 2).sum(axis=2)),
    'manhattan': lambda p: np.abs(p[:, None, :] - p[None, :, :]).sum(axis=2),
    'chebyshev': lambda p: np.abs(p[:, None, :] - p[None, :, :]).max(axis=2)
}

# Calculate all distances in one go
all_distances = {name: metric(points) for name, metric in distance_metrics.items()}

for name, dist_matrix in all_distances.items():
    print(f"\n{name.title()} distances:\n", dist_matrix)
```

Mini-batch Creation for ML

```
def create_mini_batches(X, y, batch_size=32, shuffle=True):
    """Create mini-batches for machine learning with advanced one-liner logic."""
    n_samples = X.shape[0]

    # One-liner shuffling
    indices = np.random.permutation(n_samples) if shuffle else np.arange(n_samples)
    X_shuffled, y_shuffled = X[indices], y[indices]

    # One-liner batch creation using list comprehension
    return [(X_shuffled[i:i+batch_size], y_shuffled[i:i+batch_size])
            for i in range(0, n_samples, batch_size)]

# Advanced: Stratified mini-batches (maintain class distribution)
def create_stratified_batches(X, y, batch_size=32):
    """Create stratified mini-batches maintaining class distribution."""
    unique_classes = np.unique(y)

    # One-liner: get indices for each class
    class_indices = {cls: np.where(y == cls)[0] for cls in unique_classes}

    # Calculate samples per class per batch
    samples_per_class = {cls: max(1, len(indices) * batch_size // len(y))
                         for cls, indices in class_indices.items()}

    batches = []
    max_batches = min(len(indices) // samples_per_class[cls]
                      for cls, indices in class_indices.items())

    for batch_idx in range(max_batches):
        # One-liner: select balanced samples from each class
        batch_indices = np.concatenate([
            class_indices[cls][batch_idx * samples_per_class[cls]:
                               (batch_idx + 1) * samples_per_class[cls]]
            for cls in unique_classes
        ])

        batches.append((X[batch_indices], y[batch_indices]))

    return batches

# Example usage
```

```
X = np.random.randn(100, 5)  # 100 samples, 5 features
y = np.random.randint(0, 3, 100)  # 3 classes

regular_batches = create_mini_batches(X, y, batch_size=16)
stratified_batches = create_stratified_batches(X, y, batch_size=16)

print(f"Regular batches: {len(regular_batches)}")
print(f"Stratified batches: {len(stratified_batches)}")
print(f"First regular batch shapes: X={regular_batches[0][0].shape}, y={regular_batches[0][1].sh
print(f"First stratified batch class distribution: {np.bincount(stratified_batches[0][1])}")
```

Generating Synthetic Datasets

```python
def generate_regression_data(n_samples=100, n_features=1, noise=0.1, random_state=42):
    """Generate synthetic regression dataset with advanced options."""
    np.random.seed(random_state)

    X = np.random.randn(n_samples, n_features)
    true_coefficients = np.random.randn(n_features)

    # One-liner: add polynomial features and noise
    y = (X @ true_coefficients +
         noise * np.random.randn(n_samples) +
         0.1 * np.sum(X**2, axis=1))  # Add non-linear component

    return X, y, true_coefficients

def generate_classification_data(n_samples=100, n_features=2, n_classes=2,
                                 class_sep=1.0, random_state=42):
    """Generate synthetic classification dataset with controlled separation."""
    np.random.seed(random_state)

    # One-liner: create class centers
    centers = np.random.randn(n_classes, n_features) * class_sep * 3

    # One-liner: generate samples and assign to nearest center
    X = np.random.randn(n_samples, n_features)
    y = np.array([np.argmin([np.linalg.norm(x - center) for center in centers])
                  for x in X])

    # Add class-specific offsets to improve separation
    for class_idx in range(n_classes):
        mask = y == class_idx
        X[mask] += centers[class_idx] * class_sep

    return X, y

# Advanced: Generate time series data
def generate_time_series(n_points=1000, noise_level=0.1, trend=0.001,
                         seasonality_period=50, random_state=42):
    """Generate synthetic time series with trend, seasonality, and noise."""
    np.random.seed(random_state)
    t = np.arange(n_points)
```

```python
    # One-liner: combine trend, seasonality, and noise
    series = (trend * t +
              np.sin(2 * np.pi * t / seasonality_period) +
              0.5 * np.sin(2 * np.pi * t / (seasonality_period * 0.3)) +  # Higher frequency
              noise_level * np.random.randn(n_points))

    return t, series

# Generate various datasets
X_reg, y_reg, true_coefs = generate_regression_data(100, 3, noise=0.2)
X_clf, y_clf = generate_classification_data(200, 2, 3, class_sep=1.5)
t, ts_data = generate_time_series(365, noise_level=0.2)

print("Regression data shape:", X_reg.shape, y_reg.shape)
print("Classification data shape:", X_clf.shape, y_clf.shape)
print("Time series shape:", t.shape, ts_data.shape)
print("True coefficients:", true_coefs)
print("Class distribution:", np.bincount(y_clf))

# Advanced: All-in-one dataset generator
def generate_ml_dataset(dataset_type='regression', **kwargs):
    """One-liner dataset generator for different ML tasks."""
    generators = {
        'regression': generate_regression_data,
        'classification': generate_classification_data,
        'time_series': generate_time_series
    }
    return generators[dataset_type](**kwargs)

# Usage examples
datasets = {name: generate_ml_dataset(name, n_samples=100)
           for name in ['regression', 'classification']}

print("\nGenerated datasets:")
for name, data in datasets.items():
    if name == 'time_series':
        print(f"{name}: {len(data)} points")
    else:
        print(f"{name}: X shape {data[0].shape}, y shape {data[1].shape}")
```

Exercises - Advanced Applications

Exercise 1: Implement k-means clustering algorithm using only NumPy and advanced one-liners.

Exercise 2: Create a function that performs cross-validation splits for time series data (no random shuffling).

Exercise 3: Build a complete data preprocessing pipeline using functional programming and one-liners that handles missing values, outliers, scaling, and feature engineering.

Exercise 4: Implement PCA (Principal Component Analysis) using only NumPy with matrix operations in a functional style.

Exercise 5: Create an advanced anomaly detection system using statistical methods and NumPy one-liners.

```python
# Exercise 5 Solution Hint: Advanced Anomaly Detection
def detect_anomalies_advanced(data, methods=['zscore', 'iqr', 'isolation']):
    """Advanced anomaly detection using multiple methods (one-liner approach)."""

    detectors = {
        'zscore': lambda x: np.abs((x - np.mean(x, axis=0)) / np.std(x, axis=0)) > 3,
        'iqr': lambda x: (x < (np.percentile(x, 25, axis=0) - 1.5 *
                              (np.percentile(x, 75, axis=0) - np.percentile(x, 25, axis=0)))) |
                         (x > (np.percentile(x, 75, axis=0) + 1.5 *
                              (np.percentile(x, 75, axis=0) - np.percentile(x, 25, axis=0)))),
        'isolation': lambda x: np.random.rand(*x.shape) < 0.05  # Simplified for demo
    }

    # One-liner: combine multiple detection methods
    anomaly_scores = np.mean([detectors[method](data).astype(int)
                             for method in methods if method in detectors], axis=0)

    return anomaly_scores > 0.5  # Majority vote

# Test the anomaly detector
test_data = np.random.normal(0, 1, (1000, 3))
test_data[50:60] = np.random.normal(5, 1, (10, 3))  # Insert anomalies

anomalies = detect_anomalies_advanced(test_data)
print(f"Detected {np.sum(anomalies)} anomalies out of {len(test_data)} samples")
```

Part 7 – Advanced NumPy Patterns & Best Practices

Advanced Patterns for Production Code

```python
# Pattern 1: Vectorized Operations with Error Handling
def safe_vectorized_operation(data, operation, fill_value=0):
    """Apply operation safely with error handling."""
    try:
        return np.where(np.isfinite(data), operation(data), fill_value)
    except:
        return np.full_like(data, fill_value)

# Pattern 2: Memory-Efficient Chunked Processing
def process_in_chunks(data, chunk_size=1000, operation=lambda x: x):
    """Process large arrays in memory-efficient chunks."""
    return np.concatenate([
        operation(data[i:i+chunk_size])
        for i in range(0, len(data), chunk_size)
    ]) if len(data) > 0 else data

# Pattern 3: Advanced Broadcasting with Shape Validation
def broadcast_safe_operation(a, b, operation=np.add):
    """Perform operations with automatic broadcasting validation."""
    try:
        return operation(a, b)
    except ValueError as e:
        # Attempt to fix common broadcasting issues
        if a.ndim < b.ndim:
            a = a.reshape(*((1,) * (b.ndim - a.ndim)), *a.shape)
        elif b.ndim < a.ndim:
            b = b.reshape(*((1,) * (a.ndim - b.ndim)), *b.shape)
        return operation(a, b)

# Pattern 4: Functional Composition for Data Pipelines
def compose(*functions):
    """Compose multiple functions into a single operation."""
    return lambda x: reduce(lambda acc, f: f(acc), functions, x)

# Example: Complex data processing pipeline
pipeline = compose(
    lambda x: x[~np.isnan(x).any(axis=1)],  # Remove NaN rows
    lambda x: (x - np.mean(x, axis=0)) / np.std(x, axis=0),  # Standardize
```

```
    lambda x: np.clip(x, -3, 3)  # Clip outliers
)

# Usage
sample_data = np.random.normal(0, 1, (100, 5))
sample_data[10:15] = np.nan  # Add some NaN values
processed = pipeline(sample_data)
print(f"Original shape: {sample_data.shape}, Processed shape: {processed.shape}")
```

Performance Optimization Techniques

```python
# Technique 1: Avoid Python loops with vectorization
def bad_approach(arr):
    """Slow approach using Python loops."""
    result = []
    for i in range(len(arr)):
        if arr[i] > 0:
            result.append(arr[i] ** 2)
        else:
            result.append(0)
    return np.array(result)

def good_approach(arr):
    """Fast approach using NumPy vectorization."""
    return np.where(arr > 0, arr ** 2, 0)

# Technique 2: Use views instead of copies when possible
def create_view_not_copy(arr):
    """Create views for memory efficiency."""
    return arr[::2]  # View of every second element

def force_copy_when_needed(arr):
    """Explicitly copy when modification is needed."""
    return arr[::2].copy()

# Technique 3: Leverage NumPy's built-in functions
def efficient_statistics(data):
    """Calculate multiple statistics efficiently."""
    return {
        'mean': np.mean(data, axis=0),
        'std': np.std(data, axis=0),
        'percentiles': np.percentile(data, [25, 50, 75], axis=0),
        'min_max': np.array([np.min(data, axis=0), np.max(data, axis=0)])
    }

# Technique 4: Use appropriate data types
def optimize_memory_usage(data):
    """Optimize memory by using appropriate dtypes."""
    if np.all(data == data.astype(int)):
        if np.all((data >= -128) & (data <= 127)):
            return data.astype(np.int8)
        elif np.all((data >= -32768) & (data <= 32767)):
```

```
            return data.astype(np.int16)
        else:
            return data.astype(np.int32)
    else:
        return data.astype(np.float32) if np.all(np.abs(data) < 1e10) else data

# Performance comparison
large_data = np.random.randn(1000000)

import time

# Time the approaches
start = time.time()
result_bad = bad_approach(large_data[:1000])  # Smaller subset for timing
time_bad = time.time() - start

start = time.time()
result_good = good_approach(large_data[:1000])
time_good = time.time() - start

print(f"Bad approach time: {time_bad:.6f} seconds")
print(f"Good approach time: {time_good:.6f} seconds")
print(f"Speedup: {time_bad/time_good:.1f}x faster")
```

Part 8 – Mini Project: Advanced Sales Data Analysis

Let's apply advanced NumPy techniques and one-liners to analyze sales data for a fictional company.

```python
import numpy as np
from functools import reduce

# Set seed for reproducibility
np.random.seed(42)

class AdvancedSalesAnalyzer:
    """Advanced sales data analyzer using NumPy one-liners and functional programming."""

    def __init__(self):
        self.sales_data = None
        self.product_names = None
        self.product_prices = None

    def generate_advanced_sales_data(self):
        """Generate sophisticated sales data with multiple patterns."""
        n_days, n_products = 365, 5

        # Product configuration (one-liner dictionary)
        product_config = {
            'names': ['Laptop', 'Mouse', 'Keyboard', 'Monitor', 'Webcam'],
            'base_sales': np.array([10, 50, 30, 15, 25]),
            'prices': np.array([1200, 25, 75, 400, 80]),
            'seasonality': np.array([0.3, 0.1, 0.2, 0.25, 0.4])
        }

        days = np.arange(n_days)

        # Advanced pattern generation using functional composition
        pattern_generators = [
            lambda d: 1 + product_config['seasonality'][:, None] * np.sin(2 * np.pi * d / 365),
            lambda d: 1 - 0.2 * np.isin(d % 7, [5, 6]).astype(float),  # Weekend effect
            lambda d: 1 + 0.1 * np.sin(2 * np.pi * d / 30),  # Monthly cycle
            lambda d: 1 + 0.05 * np.sin(2 * np.pi * d / 7)   # Weekly cycle
        ]

        # One-liner: combine all patterns
        combined_pattern = reduce(
            lambda acc, gen: acc * gen(days),
            pattern_generators,
            np.ones((n_products, n_days))
        )
```

```python
        # Generate sales with noise (vectorized)
        base_sales_matrix = product_config['base_sales'][:, None]
        noise_matrix = np.random.normal(0, base_sales_matrix * 0.2, (n_products, n_days))

        # One-liner: final sales calculation
        self.sales_data = np.maximum(0, base_sales_matrix * combined_pattern + noise_matrix).T
        self.product_names = product_config['names']
        self.product_prices = product_config['prices']

        return self

    def calculate_advanced_metrics(self):
        """Calculate comprehensive metrics using advanced NumPy techniques."""

        # One-liner: basic statistics
        basic_stats = {
            metric: getattr(np, f'{metric}')(self.sales_data, axis=0)
            for metric in ['sum', 'mean', 'std', 'min', 'max']
        }

        # Advanced metrics using functional programming
        advanced_metrics = {
            'coefficient_of_variation': basic_stats['std'] / basic_stats['mean'],
            'revenue_total': basic_stats['sum'] * self.product_prices,
            'revenue_share': lambda: (basic_stats['sum'] * self.product_prices) /
                                      np.sum(basic_stats['sum'] * self.product_prices) * 100
        }

        # Execute lambda functions
        for key, value in advanced_metrics.items():
            if callable(value):
                advanced_metrics[key] = value()

        return {**basic_stats, **advanced_metrics}

    def temporal_analysis_advanced(self):
        """Advanced temporal analysis using NumPy broadcasting and vectorization."""

        # One-liner: reshape data for temporal grouping
        n_days = len(self.sales_data)
        daily_totals = np.sum(self.sales_data, axis=1)

        # Advanced temporal patterns
```

```python
    temporal_analyses = {
        # Weekly analysis (reshape and mean)
        'weekly_patterns': np.mean(
            self.sales_data[:n_days//7*7].reshape(-1, 7, len(self.product_names)),
            axis=0
        ),

        # Monthly analysis (broadcasting)
        'monthly_trends': np.array([
            np.mean(self.sales_data[i*30:(i+1)*30], axis=0)
            for i in range(min(12, n_days//30))
        ]),

        # Moving averages (one-liner with convolution)
        'moving_avg_7': np.array([
            np.convolve(self.sales_data[:, i], np.ones(7)/7, mode='valid')
            for i in range(len(self.product_names))
        ]).T,

        # Volatility analysis
        'volatility': np.std(np.diff(self.sales_data, axis=0), axis=0),

        # Best/worst days (argmax/argmin)
        'best_days': np.argmax(self.sales_data, axis=0),
        'worst_days': np.argmin(self.sales_data, axis=0)
    }

    return temporal_analyses

def correlation_and_clustering_analysis(self):
    """Advanced correlation and clustering analysis."""

    # One-liner: correlation matrix
    correlation_matrix = np.corrcoef(self.sales_data.T)

    # Advanced correlation insights
    corr_insights = {
        'correlation_matrix': correlation_matrix,
        'most_correlated_pair': np.unravel_index(
            np.argmax(np.triu(correlation_matrix, k=1)),
            correlation_matrix.shape
        ),
        'least_correlated_pair': np.unravel_index(
```

```
            np.argmin(np.triu(correlation_matrix, k=1) + np.eye(len(correlation_matrix))),
            correlation_matrix.shape
        )
    }

    # Simple clustering using correlation distance
    corr_distance = 1 - np.abs(correlation_matrix)

    return corr_insights, corr_distance

def anomaly_detection_advanced(self):
    """Multi-method anomaly detection."""

    # Multiple anomaly detection methods (one-liners)
    anomaly_detectors = {
        'zscore': lambda x: np.abs((x - np.mean(x, axis=0)) / np.std(x, axis=0)) > 2.5,
        'iqr': lambda x: (x < (np.percentile(x, 25, axis=0) - 1.5 *
                          (np.percentile(x, 75, axis=0) - np.percentile(x, 25, axis=0))
                     (x > (np.percentile(x, 75, axis=0) + 1.5 *
                          (np.percentile(x, 75, axis=0) - np.percentile(x, 25, axis=0))
        'percentile': lambda x: (x < np.percentile(x, 5, axis=0)) | (x > np.percentile(x, 9
    }

    # One-liner: combine detection methods
    anomaly_scores = {
        method: detector(self.sales_data).astype(int)
        for method, detector in anomaly_detectors.items()
    }

    # Consensus anomaly detection (majority vote)
    consensus_anomalies = np.mean(list(anomaly_scores.values()), axis=0) > 0.5

    return anomaly_scores, consensus_anomalies

def predictive_insights(self):
    """Generate predictive insights using trend analysis."""

    # Trend analysis using linear regression (NumPy only)
    days = np.arange(len(self.sales_data))
    X = np.column_stack([np.ones(len(days)), days])  # Design matrix

    # One-liner: calculate trends for all products
    trends = np.array([
```

```python
            np.linalg.lstsq(X, self.sales_data[:, i], rcond=None)[0][1]
            for i in range(len(self.product_names))
        ])

        # Forecast next 30 days (one-liner)
        future_days = np.arange(len(self.sales_data), len(self.sales_data) + 30)
        forecasts = np.array([
            np.mean(self.sales_data[-30:, i]) + trends[i] * np.arange(1, 31)
            for i in range(len(self.product_names))
        ]).T

        return {
            'trends': trends,
            'forecasts': forecasts,
            'trend_direction': np.where(trends > 0, 'Increasing', 'Decreasing'),
            'forecast_revenue': np.sum(forecasts * self.product_prices, axis=1)
        }

    def run_complete_analysis(self):
        """Run comprehensive analysis with all advanced techniques."""

        # Generate data
        self.generate_advanced_sales_data()

        # Run all analyses
        analyses = {
            'basic_metrics': self.calculate_advanced_metrics(),
            'temporal': self.temporal_analysis_advanced(),
            'correlation': self.correlation_and_clustering_analysis(),
            'anomalies': self.anomaly_detection_advanced(),
            'predictions': self.predictive_insights()
        }

        return analyses

    def generate_executive_summary(self, analyses):
        """Generate executive summary using advanced NumPy operations."""

        metrics = analyses['basic_metrics']
        temporal = analyses['temporal']
        predictions = analyses['predictions']

        # One-liner insights
```

```python
        insights = {
            'top_performer': self.product_names[np.argmax(metrics['revenue_total'])],
            'most_volatile': self.product_names[np.argmax(metrics['coefficient_of_variation'])].
            'best_growth': self.product_names[np.argmax(predictions['trends'])],
            'total_revenue': np.sum(metrics['revenue_total']),
            'avg_daily_sales': np.mean(np.sum(self.sales_data, axis=1)),
            'forecast_revenue_30d': np.sum(predictions['forecast_revenue'])
        }

        return insights

# Execute the advanced analysis
analyzer = AdvancedSalesAnalyzer()
results = analyzer.run_complete_analysis()
summary = analyzer.generate_executive_summary(results)

# Display results with advanced formatting
print("="*60)
print("ADVANCED SALES ANALYTICS DASHBOARD")
print("="*60)

print(f"\n🏆 TOP PERFORMER: {summary['top_performer']}")
print(f"📈 BEST GROWTH TREND: {summary['best_growth']}")
print(f"⚡ MOST VOLATILE: {summary['most_volatile']}")
print(f"💰 TOTAL REVENUE: ${summary['total_revenue']:,.0f}")
print(f"📊 AVERAGE DAILY SALES: {summary['avg_daily_sales']:.0f} units")
print(f"🔮 30-DAY FORECAST REVENUE: ${summary['forecast_revenue_30d']:,.0f}")

# Advanced correlation insights
corr_insights, _ = results['correlation']
most_corr = corr_insights['most_correlated_pair']
print(f"\n🔗 MOST CORRELATED PRODUCTS: {analyzer.product_names[most_corr[0]]} & {analyzer.produ

# Anomaly summary
_, consensus_anomalies = results['anomalies']
anomaly_days = np.sum(np.any(consensus_anomalies, axis=1))
print(f"🚨 ANOMALY DAYS DETECTED: {anomaly_days} ({anomaly_days/365*100:.1f}% of year)")

# Advanced trend analysis
trends = results['predictions']['trends']
positive_trends = np.sum(trends > 0)
print(f"📈 PRODUCTS WITH POSITIVE TRENDS: {positive_trends}/{len(analyzer.product_names)}")
```

```python
print("\n" + "="*60)
print("ADVANCED METRICS BY PRODUCT")
print("="*60)

for i, product in enumerate(analyzer.product_names):
    metrics = results['basic_metrics']
    print(f"\n{product}:")
    print(f"  Revenue: ${metrics['revenue_total'][i]:,.0f} ({metrics['revenue_share'][i]:.1f}%)"
    print(f"  Avg Daily Sales: {metrics['mean'][i]:.1f} units")
    print(f"  Volatility (CV): {metrics['coefficient_of_variation'][i]:.2f}")
    print(f"  Trend: {results['predictions']['trend_direction'][i]} ({trends[i]:+.3f} units/day

# Advanced temporal insights
weekly_patterns = results['temporal']['weekly_patterns']
best_weekday = np.argmax(np.sum(weekly_patterns, axis=1))
weekdays = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday']

print(f"\n📅 BEST SALES DAY: {weekdays[best_weekday]}")
print(f"📅 WORST SALES DAY: {weekdays[np.argmin(np.sum(weekly_patterns, axis=1))]}")

print("\n" + "="*60)
print("This analysis demonstrates advanced NumPy techniques:")
print("• Functional programming with reduce() and lambda functions")
print("• Advanced broadcasting and vectorization")
print("• One-liner statistical computations")
print("• Multi-dimensional array manipulation")
print("• Advanced indexing and boolean operations")
print("• Linear algebra for trend analysis")
print("• Memory-efficient data processing")
print("="*60)
```

This advanced mini project demonstrates:

- **Functional programming patterns** with NumPy
- **Advanced one-liners** for complex operations
- **Broadcasting and vectorization** for efficiency
- **Multi-dimensional array manipulation**
- **Advanced statistical analysis**
- **Predictive modeling** using linear algebra
- **Anomaly detection** with multiple methods
- **Memory-efficient processing** techniques
- **Complex data pipelines** using composition

Let's apply multiple NumPy concepts to analyze sales data for a fictional company.

```python
import numpy as np

# Set seed for reproducibility
np.random.seed(42)

# Generate synthetic sales data
def generate_sales_data():
    """Generate 1 year of daily sales data for multiple products."""
    n_days = 365
    n_products = 5

    # Product names
    products = ['Laptop', 'Mouse', 'Keyboard', 'Monitor', 'Webcam']

    # Base daily sales (units) for each product
    base_sales = np.array([10, 50, 30, 15, 25])

    # Create seasonal trends (higher sales in certain months)
    days = np.arange(n_days)
    seasonal_factor = 1 + 0.3 * np.sin(2 * np.pi * days / 365)

    # Add weekly patterns (lower sales on weekends)
    weekly_factor = 1 - 0.2 * np.isin(days % 7, [5, 6]).astype(float)

    # Generate sales data with noise
    sales_data = np.zeros((n_days, n_products))
    for i in range(n_products):
        trend = base_sales[i] * seasonal_factor * weekly_factor
        noise = np.random.normal(0, base_sales[i] * 0.2, n_days)
        sales_data[:, i] = np.maximum(0, trend + noise)  # No negative sales

    # Product prices
    prices = np.array([1200, 25, 75, 400, 80])

    return sales_data, products, prices

# Generate the data
sales_units, product_names, product_prices = generate_sales_data()

print("Sales Data Shape:", sales_units.shape)
print("Product Names:", product_names)
print("Product Prices:", product_prices)
```

```python
# Analysis 1: Basic Statistics
print("\n" + "="*50)
print("BASIC SALES STATISTICS")
print("="*50)

total_units_sold = np.sum(sales_units, axis=0)
avg_daily_sales = np.mean(sales_units, axis=0)
max_daily_sales = np.max(sales_units, axis=0)
std_daily_sales = np.std(sales_units, axis=0)

for i, product in enumerate(product_names):
    print(f"{product}:")
    print(f"  Total units sold: {total_units_sold[i]:.0f}")
    print(f"  Average daily sales: {avg_daily_sales[i]:.1f}")
    print(f"  Max daily sales: {max_daily_sales[i]:.0f}")
    print(f"  Sales volatility (std): {std_daily_sales[i]:.1f}")
    print()

# Analysis 2: Revenue Analysis
print("="*50)
print("REVENUE ANALYSIS")
print("="*50)

# Calculate daily revenue for each product
daily_revenue = sales_units * product_prices

# Total revenue per product
total_revenue = np.sum(daily_revenue, axis=0)

# Revenue share
revenue_share = total_revenue / np.sum(total_revenue) * 100

print("Revenue by Product:")
for i, product in enumerate(product_names):
    print(f"{product}: ${total_revenue[i]:,.0f} ({revenue_share[i]:.1f}%)")

print(f"\nTotal Revenue: ${np.sum(total_revenue):,.0f}")

# Analysis 3: Temporal Patterns
print("\n" + "="*50)
print("TEMPORAL PATTERNS")
print("="*50)
```

```python
# Best and worst sales days
total_daily_sales = np.sum(sales_units, axis=1)
best_day = np.argmax(total_daily_sales)
worst_day = np.argmin(total_daily_sales)

print(f"Best sales day: Day {best_day + 1} with {total_daily_sales[best_day]:.0f} units")
print(f"Worst sales day: Day {worst_day + 1} with {total_daily_sales[worst_day]:.0f} units")

# Monthly analysis (group by 30-day periods)
n_months = 12
monthly_sales = np.zeros((n_months, len(product_names)))

for month in range(n_months):
    start_day = month * 30
    end_day = min((month + 1) * 30, 365)
    monthly_sales[month] = np.sum(sales_units[start_day:end_day], axis=0)

# Find best month for each product
best_months = np.argmax(monthly_sales, axis=0)
print("\nBest month for each product:")
for i, product in enumerate(product_names):
    print(f"{product}: Month {best_months[i] + 1}")

# Analysis 4: Correlation Analysis
print("\n" + "="*50)
print("PRODUCT CORRELATION ANALYSIS")
print("="*50)

# Calculate correlation matrix
def correlation_matrix(data):
    """Calculate correlation matrix manually using NumPy."""
    # Center the data
    centered = data - np.mean(data, axis=0)
    # Calculate covariance matrix
    cov_matrix = (centered.T @ centered) / (data.shape[0] - 1)
    # Calculate standard deviations
    stds = np.sqrt(np.diag(cov_matrix))
    # Calculate correlation matrix
    corr_matrix = cov_matrix / (stds[:, np.newaxis] @ stds[np.newaxis, :])
    return corr_matrix

corr_matrix = correlation_matrix(sales_units)
```

```python
print("Correlation Matrix:")
print("Products:", [name[:8] for name in product_names])
for i, row in enumerate(corr_matrix):
    print(f"{product_names[i][:8]:8}", [f"{val:6.3f}" for val in row])

# Find most correlated product pairs
corr_upper = np.triu(corr_matrix, k=1)  # Upper triangle excluding diagonal
max_corr_idx = np.unravel_index(np.argmax(corr_upper), corr_upper.shape)
max_corr_value = corr_upper[max_corr_idx]

print(f"\nMost correlated products: {product_names[max_corr_idx[0]]} and {product_names[max_corr
print(f"Correlation: {max_corr_value:.3f}")

# Analysis 5: Outlier Detection
print("\n" + "="*50)
print("OUTLIER DETECTION")
print("="*50)

def detect_outliers_iqr(data, multiplier=1.5):
    """Detect outliers using IQR method."""
    q25 = np.percentile(data, 25, axis=0)
    q75 = np.percentile(data, 75, axis=0)
    iqr = q75 - q25

    lower_bound = q25 - multiplier * iqr
    upper_bound = q75 + multiplier * iqr

    outliers = (data < lower_bound) | (data > upper_bound)
    return outliers

outliers = detect_outliers_iqr(sales_units)
outlier_days = np.any(outliers, axis=1)

print(f"Number of days with outliers: {np.sum(outlier_days)}")
print("Outliers by product:")
for i, product in enumerate(product_names):
    product_outliers = np.sum(outliers[:, i])
    print(f"{product}: {product_outliers} outlier days")

# Analysis 6: Performance Metrics
print("\n" + "="*50)
print("PERFORMANCE METRICS")
print("="*50)
```

```python
# Calculate moving averages (7-day and 30-day)
def moving_average(data, window):
    """Calculate moving average."""
    return np.convolve(data, np.ones(window)/window, mode='valid')

# Calculate for total daily sales
ma_7 = moving_average(total_daily_sales, 7)
ma_30 = moving_average(total_daily_sales, 30)

print(f"7-day moving average (last value): {ma_7[-1]:.1f}")
print(f"30-day moving average (last value): {ma_30[-1]:.1f}")

# Growth rate analysis (month-over-month)
monthly_total_sales = np.sum(monthly_sales, axis=1)
growth_rates = np.diff(monthly_total_sales) / monthly_total_sales[:-1] * 100

print("\nMonth-over-month growth rates:")
for i, growth in enumerate(growth_rates):
    print(f"Month {i+1} to {i+2}: {growth:+.1f}%")

print(f"Average monthly growth: {np.mean(growth_rates):+.1f}%")

# Summary Report
print("\n" + "="*60)
print("EXECUTIVE SUMMARY")
print("="*60)

print(f"• Total annual revenue: ${np.sum(total_revenue):,.0f}")
print(f"• Average daily sales: {np.mean(total_daily_sales):.0f} units")
print(f"• Most profitable product: {product_names[np.argmax(total_revenue)]}")
print(f"• Most volatile product: {product_names[np.argmax(std_daily_sales)]}")
print(f"• Days with outlier sales: {np.sum(outlier_days)} ({np.sum(outlier_days)/365*100:.1f}%)'
print(f"• Overall growth trend: {np.mean(growth_rates):+.1f}% per month")
```

This mini project demonstrates:

- Array creation and manipulation
- Statistical analysis with aggregations
- Broadcasting for revenue calculations
- Boolean indexing for outlier detection
- Custom functions using NumPy operations

- Correlation analysis
- Time series analysis with moving averages
- Data reshaping for temporal grouping

Further Reading

Official Documentation

- NumPy Official Documentation
- NumPy User Guide
- NumPy API Reference

Tutorials and Courses

- NumPy Quickstart Tutorial
- Scientific Python Lectures
- Real Python NumPy Tutorial

Advanced Topics

- Advanced NumPy
- NumPy for MATLAB Users
- Broadcasting in NumPy

Related Libraries

- Pandas - Data manipulation and analysis
- Matplotlib - Plotting and visualization
- Scikit-learn - Machine learning
- SciPy - Scientific computing

Books

- "Python for Data Analysis" by Wes McKinney
- "Numerical Python" by Robert Johansson
- "Python Data Science Handbook" by Jake VanderPlas

Final Notes

Congratulations! You've completed the NumPy crash course. You now have the foundational knowledge to:

- Create and manipulate NumPy arrays efficiently
- Perform complex mathematical and statistical operations
- Handle real-world data science tasks
- Optimize your code for performance
- Build data pipelines for machine learning

Next Steps:

1. Practice with real datasets
2. Explore Pandas for data manipulation
3. Learn Matplotlib/Seaborn for visualization
4. Dive into Scikit-learn for machine learning
5. Consider TensorFlow/PyTorch for deep learning

Remember: The best way to master NumPy is through practice. Start applying these concepts to your own projects and datasets!