Lecture 12 Handout

NumPy Fundamentals and Array Operations

The Foundation of Scientific Computing in Python

INF 605 - Introduction to Programming - Python

Prof. Rongyu Lin
Quinnipiac University
School of Computing and Engineering

Fall 2024

Required Reading

Textbook: Chapter 4, Sections 4.1-4.3 (NumPy Basics)
Reference Notebooks: `ch04/04_NumPy_Basics.ipynb` for array fundamentals

Learning Objectives

By the end of this lecture, you will be able to:

1. **Understand NumPy's role** in Python's scientific computing ecosystem and why it's essential for data analysis
2. **Create NumPy arrays** from lists, tuples, and built-in functions with appropriate data types
3. **Master array indexing and slicing** including multi-dimensional arrays and boolean indexing
4. **Perform element-wise operations** and understand the difference between array and list operations
5. **Manipulate array shapes** using reshape, flatten, and transpose operations
6. **Apply mathematical operations** including statistical functions and aggregations
7. **Work with array broadcasting** to perform operations on arrays of different shapes
8. **Create professional data processing pipelines** using NumPy's efficient array operations

Prerequisites Review

Building on Your Complete Programming Foundation:

From your comprehensive foundation in Python programming (Lectures 1-11), you've mastered fundamental data types, control structures, functions, file handling, object-oriented programming with inheritance and polymorphism, and advanced string operations. You've worked with lists, tuples, and dictionaries, understanding how Python stores and manipulates data in memory.

This lecture introduces NumPy, Python's fundamental package for scientific computing. While Python's built-in lists are flexible and easy to use, they're not optimized for numerical computations. NumPy provides a powerful N-dimensional array object that forms the foundation for nearly all scientific and data analysis libraries in Python.

Transformation Goal: Evolve from **using Python lists for basic data storage** to **leveraging NumPy arrays for efficient numerical computing and data analysis**.

1 Part 1: Introduction to NumPy

1.1 Why NumPy?

NumPy (Numerical Python) is the fundamental package for scientific computing in Python. It provides a high-performance multidimensional array object and tools for working with these arrays. Think of NumPy arrays as specialized containers optimized for numerical data - like having a Formula 1 race car instead of a family sedan when you need speed and precision.

The key advantages of NumPy over Python lists include:

- **Performance**: NumPy arrays are stored in contiguous memory and operations are implemented in C, making them 10-100x faster than Python lists for numerical operations
- **Vectorization**: Operations can be applied to entire arrays without writing loops, leading to cleaner and faster code
- **Memory Efficiency**: NumPy arrays use less memory than Python lists for numerical data
- **Broadcasting**: Sophisticated rules for performing operations on arrays of different shapes
- **Ecosystem**: NumPy is the foundation for pandas, scikit-learn, matplotlib, and most scientific Python libraries

```python
# Comparing Python lists vs NumPy arrays
import numpy as np
import time

# Creating large datasets
size = 1000000
python_list = list(range(size))
numpy_array = np.arange(size)

# Timing element-wise multiplication
# Python list approach
start_time = time.time()
python_result = [x * 2 for x in python_list]
python_time = time.time() - start_time

# NumPy array approach
start_time = time.time()
numpy_result = numpy_array * 2
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!")
```

1.2 Installing and Importing NumPy

NumPy is not part of Python's standard library and needs to be installed separately. It's typically installed using pip:

```
# Installation (run in terminal/command prompt)
pip install numpy

# Standard import convention
import numpy as np  # 'np' is the universally accepted alias

# Verify installation
print(f"NumPy version: {np.__version__}")
```

2 Part 2: Creating NumPy Arrays

2.1 Arrays from Python Sequences

The most straightforward way to create NumPy arrays is from Python lists or tuples. The 'np.array()' function converts sequences into arrays, automatically inferring the appropriate data type. Think of this as pouring water (your data) from a flexible container (Python list) into a rigid, optimized container (NumPy array).

```
# Creating 1D arrays from lists
numbers_list = [1, 2, 3, 4, 5]
numbers_array = np.array(numbers_list)
print(f"Array: {numbers_array}")
print(f"Type: {type(numbers_array)}")
print(f"Data type: {numbers_array.dtype}")

# Creating arrays with specific data types
float_array = np.array([1, 2, 3, 4, 5], dtype=np.float64)
print(f"Float array: {float_array}")

# Creating 2D arrays (matrices)
matrix_list = [[1, 2, 3],
               [4, 5, 6],
               [7, 8, 9]]
matrix_array = np.array(matrix_list)
print(f"\n2D Array:\n{matrix_array}")
print(f"Shape: {matrix_array.shape}")  # (rows, columns)
print(f"Dimensions: {matrix_array.ndim}")
print(f"Total elements: {matrix_array.size}")
```

2.2 Array Creation Functions

NumPy provides numerous functions to create arrays with specific patterns or values. These are essential for initializing arrays for computations, creating test data, or setting up mathematical operations.

```
# Common array creation functions
# Zeros - often used for initialization
zeros_1d = np.zeros(5)
zeros_2d = np.zeros((3, 4))  # Note: shape as tuple
print(f"1D zeros: {zeros_1d}")
print(f"2D zeros:\n{zeros_2d}")

# Ones - useful for multiplicative operations
ones_1d = np.ones(5)
ones_2d = np.ones((2, 3), dtype=int)
```

```python
print(f"\n1D ones: {ones_1d}")
print(f"2D ones:\n{ones_2d}")

# Full - create array filled with specific value
full_array = np.full((3, 3), 7)
print(f"\nFull array:\n{full_array}")

# Identity matrix - diagonal ones, zeros elsewhere
identity = np.eye(4)
print(f"\nIdentity matrix:\n{identity}")

# Empty - uninitialized array (faster but contains garbage)
empty_array = np.empty((2, 2))
print(f"\nEmpty array (uninitialized):\n{empty_array}")
```

2.3 Sequential Arrays

Creating arrays with sequential values is common in numerical computing. NumPy provides ‘arange‘ (similar to Python’s range) and ‘linspace‘ for different sequential patterns.

```python
# arange - similar to range but returns array
integers = np.arange(10)  # 0 to 9
evens = np.arange(0, 20, 2)  # Even numbers 0 to 18
floats = np.arange(0, 1, 0.1)  # 0.0 to 0.9 in steps of 0.1

print(f"Integers: {integers}")
print(f"Evens: {evens}")
print(f"Floats: {floats}")

# linspace - evenly spaced values over interval
linear = np.linspace(0, 1, 5)  # 5 points from 0 to 1
angles = np.linspace(0, 2*np.pi, 8)  # For circular calculations

print(f"\nLinspace: {linear}")
print(f"Angles: {angles}")

# Random arrays - essential for simulations
np.random.seed(42)  # For reproducibility
random_uniform = np.random.random((3, 3))  # 0 to 1
random_normal = np.random.randn(5)  # Standard normal
random_integers = np.random.randint(1, 10, size=10)

print(f"\nRandom uniform:\n{random_uniform}")
print(f"Random normal: {random_normal}")
print(f"Random integers: {random_integers}")
```

3 Part 3: Array Indexing and Slicing

3.1 Basic Indexing

NumPy arrays support all the indexing operations you learned with Python lists, plus powerful additional features. For 1D arrays, indexing works exactly like lists. For multi-dimensional arrays, you can index along each dimension separately or simultaneously.

```python
# 1D array indexing
arr_1d = np.array([10, 20, 30, 40, 50])
print(f"Original array: {arr_1d}")
print(f"First element: {arr_1d[0]}")
print(f"Last element: {arr_1d[-1]}")
print(f"Third element: {arr_1d[2]}")
```

```python

# 2D array indexing
arr_2d = np.array([[1, 2, 3],
                   [4, 5, 6],
                   [7, 8, 9]])
print(f"\n2D array:\n{arr_2d}")

# Two ways to index 2D arrays
print(f"Element at row 1, col 2: {arr_2d[1, 2]}")  # Preferred
print(f"Same element: {arr_2d[1][2]}")  # Works but slower

# Accessing entire rows or columns
print(f"Second row: {arr_2d[1]}")  # Complete row
print(f"Second column: {arr_2d[:, 1]}")  # All rows, column 1

# Modifying elements
arr_2d[0, 0] = 100
arr_2d[2] = [70, 80, 90]  # Replace entire row
print(f"\nModified array:\n{arr_2d}")
```

3.2 Advanced Slicing

NumPy extends Python's slicing syntax to work with multiple dimensions. You can slice along each axis independently, creating powerful ways to extract subarrays. Think of this as cutting a cake - you can slice horizontally, vertically, or both to get exactly the piece you want.

```python
# Creating a larger array for slicing demos
big_array = np.arange(24).reshape(4, 6)
print(f"Original array:\n{big_array}")

# Basic slicing
print(f"\nFirst two rows:\n{big_array[:2]}")
print(f"Last two columns:\n{big_array[:, -2:]}")
print(f"Middle section:\n{big_array[1:3, 2:5]}")

# Strided slicing
print(f"\nEvery other row:\n{big_array[::2]}")
print(f"Every other column:\n{big_array[:, ::2]}")
print(f"Reverse rows:\n{big_array[::-1]}")

# Combining slicing techniques
print(f"\nComplex slice - alternating rows, middle columns:\n"
      f"{big_array[::2, 1:4]}")

# Important: Slicing creates views, not copies!
slice_view = big_array[1:3, 2:4]
print(f"\nSlice view:\n{slice_view}")
slice_view[0, 0] = 999
print(f"Original array modified:\n{big_array}")

# Creating a copy instead of view
slice_copy = big_array[1:3, 2:4].copy()
slice_copy[0, 0] = -1
print(f"\nCopy modified, original unchanged:\n{big_array}")
```

3.3 Boolean Indexing

Boolean indexing is one of NumPy's most powerful features, allowing you to select elements based on conditions. This creates a boolean mask array that selects only the elements where the condition is True. It's like using a filter to extract exactly the data points you need.

```python
# Boolean indexing with 1D arrays
temps = np.array([72, 68, 75, 71, 69, 76, 73])
print(f"Temperatures: {temps}")

# Create boolean mask
warm_days = temps > 70
print(f"Warm days mask: {warm_days}")

# Use mask to select elements
warm_temps = temps[warm_days]
print(f"Warm temperatures: {warm_temps}")

# Direct boolean indexing
print(f"Cool temperatures: {temps[temps <= 70]}")

# Modifying elements with boolean indexing
temps[temps > 75] = 75  # Cap at 75
print(f"Capped temperatures: {temps}")

# Boolean indexing with 2D arrays
data = np.random.randint(0, 100, size=(5, 4))
print(f"\nRandom data:\n{data}")

# Multiple conditions
mask = (data > 30) & (data < 70)  # Note: use & not 'and'
print(f"Values between 30 and 70: {data[mask]}")

# Set values based on condition
data[data < 50] = 0
data[data >= 50] = 1
print(f"Binary data:\n{data}")
```

4 Part 4: Array Operations and Mathematics

4.1 Element-wise Operations

NumPy arrays support vectorized operations, meaning you can perform operations on entire arrays without writing loops. These operations are element-wise by default, applying the operation to each corresponding element. This is like having a team of workers each handling one element simultaneously, rather than one worker processing elements sequentially.

```python
# Basic arithmetic operations
a = np.array([1, 2, 3, 4, 5])
b = np.array([10, 20, 30, 40, 50])

print(f"Array a: {a}")
print(f"Array b: {b}")

# Element-wise operations
print(f"\nAddition (a + b): {a + b}")
print(f"Subtraction (b - a): {b - a}")
print(f"Multiplication (a * b): {a * b}")
print(f"Division (b / a): {b / a}")
print(f"Power (a ** 2): {a ** 2}")

# Operations with scalars (broadcasting)
print(f"\nScalar operations:")
print(f"a * 10: {a * 10}")
print(f"b + 5: {b + 5}")
print(f"100 / a: {100 / a}")
```

```python

# Mathematical functions
angles = np.array([0, np.pi/4, np.pi/2, np.pi])
print(f"\nAngles: {angles}")
print(f"Sine: {np.sin(angles)}")
print(f"Cosine: {np.cos(angles)}")

# More math functions
numbers = np.array([1, 4, 9, 16, 25])
print(f"\nSquare root: {np.sqrt(numbers)}")
print(f"Exponential: {np.exp(a[:3])}")  # e^x
print(f"Natural log: {np.log(numbers)}")
```

4.2 Array Aggregations

Aggregation functions compute summary statistics across arrays or along specific axes. These are essential for data analysis, allowing you to quickly understand the characteristics of your data. Think of these as taking a bird's-eye view of your data landscape.

```python
# Creating sample data
data = np.random.randint(1, 100, size=(4, 5))
print(f"Sample data:\n{data}")

# Basic aggregations on entire array
print(f"\nArray-wide statistics:")
print(f"Sum: {np.sum(data)}")
print(f"Mean: {np.mean(data)}")
print(f"Median: {np.median(data)}")
print(f"Standard deviation: {np.std(data):.2f}")
print(f"Min: {np.min(data)}, Max: {np.max(data)}")

# Aggregations along axes
print(f"\nAxis-specific aggregations:")
print(f"Sum along rows (axis=1): {np.sum(data, axis=1)}")
print(f"Sum along columns (axis=0): {np.sum(data, axis=0)}")
print(f"Mean of each row: {np.mean(data, axis=1)}")
print(f"Max of each column: {np.max(data, axis=0)}")

# Finding positions of min/max
print(f"\nPosition information:")
print(f"Position of minimum: {np.argmin(data)}")
print(f"Position of maximum: {np.argmax(data)}")
print(f"Position as (row, col): {np.unravel_index(np.argmax(data), data.shape)}
    ")

# Cumulative operations
arr = np.array([1, 2, 3, 4, 5])
print(f"\nCumulative operations on {arr}:")
print(f"Cumulative sum: {np.cumsum(arr)}")
print(f"Cumulative product: {np.cumprod(arr)}")
```

5 Part 5: Array Shape Manipulation

5.1 Reshaping Arrays

Reshaping is fundamental to NumPy operations, allowing you to change the dimensions of an array without changing its data. Think of it like reorganizing a deck of cards - you can arrange them in different patterns (4 rows of 13, 13 rows of 4, etc.) but you still have the same 52 cards.

```python
# Creating a 1D array
original = np.arange(12)
print(f"Original 1D array: {original}")
print(f"Shape: {original.shape}")

# Reshape to 2D
reshaped_2d = original.reshape(3, 4)
print(f"\nReshaped to 3x4:\n{reshaped_2d}")

# Reshape to different 2D
reshaped_alt = original.reshape(2, 6)
print(f"\nReshaped to 2x6:\n{reshaped_alt}")

# Reshape to 3D
reshaped_3d = original.reshape(2, 2, 3)
print(f"\nReshaped to 2x2x3:\n{reshaped_3d}")

# Using -1 for automatic dimension calculation
auto_reshape = original.reshape(3, -1)  # NumPy calculates columns
print(f"\nAuto reshape (3, -1):\n{auto_reshape}")

# Flattening arrays
print(f"\nFlattening methods:")
print(f"Flatten (copy): {reshaped_2d.flatten()}")
print(f"Ravel (view when possible): {reshaped_2d.ravel()}")

# Important: reshape returns a view when possible
reshaped_view = original.reshape(3, 4)
reshaped_view[0, 0] = 999
print(f"\nOriginal modified through view: {original}")
```

5.2 Transposing and Axis Manipulation

Transposing swaps the axes of an array, which is essential for matrix operations and data alignment. For 2D arrays, this flips rows and columns. For higher dimensions, you can specify exactly how to rearrange the axes.

```python
# Transposing 2D arrays
matrix = np.array([[1, 2, 3],
                   [4, 5, 6]])
print(f"Original matrix:\n{matrix}")
print(f"Shape: {matrix.shape}")

# Simple transpose
transposed = matrix.T
print(f"\nTransposed:\n{transposed}")
print(f"Shape: {transposed.shape}")

# Alternative transpose methods
transposed_alt = np.transpose(matrix)
print(f"\nUsing np.transpose:\n{transposed_alt}")

# Transposing 1D arrays (no effect)
arr_1d = np.array([1, 2, 3, 4])
print(f"\n1D array: {arr_1d}")
print(f"1D transposed: {arr_1d.T}")  # Still 1D!

# For higher dimensions
arr_3d = np.arange(24).reshape(2, 3, 4)
print(f"\n3D array shape: {arr_3d.shape}")
transposed_3d = np.transpose(arr_3d, axes=(1, 0, 2))
```

```python
print(f"Transposed shape: {transposed_3d.shape}")

# Swapping specific axes
swapped = np.swapaxes(arr_3d, 0, 1)
print(f"Swapped axes shape: {swapped.shape}")
```

5.3 Stacking and Splitting

Combining and dividing arrays is essential for data preprocessing and manipulation. NumPy provides various functions to stack arrays together or split them apart, like assembling or disassembling building blocks.

```python
# Creating sample arrays
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])
c = np.array([7, 8, 9])

print(f"Array a: {a}")
print(f"Array b: {b}")
print(f"Array c: {c}")

# Vertical stacking (row-wise)
vstacked = np.vstack([a, b, c])
print(f"\nVertical stack:\n{vstacked}")

# Horizontal stacking (column-wise)
hstacked = np.hstack([a, b, c])
print(f"\nHorizontal stack: {hstacked}")

# Column stacking (1D to 2D columns)
column_stacked = np.column_stack([a, b, c])
print(f"\nColumn stack:\n{column_stacked}")

# Concatenate (general purpose)
concat_axis0 = np.concatenate([a.reshape(1, -1),
                               b.reshape(1, -1),
                               c.reshape(1, -1)], axis=0)
print(f"\nConcatenate along axis 0:\n{concat_axis0}")

# Splitting arrays
big_array = np.arange(12).reshape(3, 4)
print(f"\nArray to split:\n{big_array}")

# Split into equal parts
vsplit_arrays = np.vsplit(big_array, 3)  # 3 equal row groups
print(f"\nVertical split into 3:")
for i, arr in enumerate(vsplit_arrays):
    print(f"Part {i+1}: {arr}")

# Split at specific indices
hsplit_arrays = np.hsplit(big_array, [1, 3])  # Split at columns 1 and 3
print(f"\nHorizontal split at indices [1, 3]:")
for i, arr in enumerate(hsplit_arrays):
    print(f"Part {i+1}:\n{arr}")
```

6 Part 6: Broadcasting

6.1 Understanding Broadcasting Rules

Broadcasting is NumPy's powerful mechanism for performing operations on arrays of different shapes. It follows specific rules to "stretch" smaller arrays across larger ones, enabling element-wise operations without explicitly creating copies. Think of broadcasting like using a stamp - you can apply the same pattern across a larger surface without manually copying it.

Broadcasting rules:

1. Arrays are compatible for broadcasting if their dimensions match or if one dimension is 1
2. Arrays are broadcast together by adding dimensions of size 1 to the left
3. After broadcasting, all dimensions must match

```python
# Broadcasting scalars
array = np.array([[1, 2, 3],
                  [4, 5, 6]])
scalar = 10

print(f"Array:\n{array}")
print(f"Array + scalar: \n{array + scalar}")  # Scalar broadcasts to all elements

# Broadcasting 1D array to 2D
row_array = np.array([10, 20, 30])
print(f"\nRow array: {row_array}")
print(f"Array + row array:\n{array + row_array}")  # Broadcasts across rows

# Broadcasting with reshape for column operations
col_array = np.array([[100], [200]])  # Shape (2, 1)
print(f"\nColumn array:\n{col_array}")
print(f"Array + column array:\n{array + col_array}")  # Broadcasts across columns

# More complex broadcasting
a = np.ones((3, 4))
b = np.arange(4)
c = np.arange(3).reshape(3, 1)

print(f"\nShapes: a={a.shape}, b={b.shape}, c={c.shape}")
print(f"a + b:\n{a + b}")
print(f"a + c:\n{a + c}")

# Broadcasting in practice - normalization
data = np.random.randint(0, 100, size=(5, 3))
print(f"\nOriginal data:\n{data}")

# Normalize each column (subtract mean, divide by std)
col_means = data.mean(axis=0)  # Shape (3,)
col_stds = data.std(axis=0)    # Shape (3,)
normalized = (data - col_means) / col_stds
print(f"\nNormalized data:\n{normalized}")
print(f"Normalized means: {normalized.mean(axis=0)}")  # Should be ~0
print(f"Normalized stds: {normalized.std(axis=0)}")    # Should be ~1
```

6.2 Practical Broadcasting Applications

Broadcasting enables elegant solutions to common data manipulation tasks. Here we explore practical applications that demonstrate the power and efficiency of broadcasting in real-world scenarios.

```
# Creating a multiplication table using broadcasting
rows = np.arange(1, 11).reshape(10, 1)  # Column vector
cols = np.arange(1, 11)                  # Row vector

multiplication_table = rows * cols
print("Multiplication table (1-10):")
print(multiplication_table)

# Distance calculation between points
# Points in 2D space
points = np.array([[0, 0], [1, 0], [0, 1], [1, 1]])
# Calculate distances from origin [0, 0]
origin = np.array([0, 0])
distances = np.sqrt(np.sum((points - origin)**2, axis=1))
print(f"\nDistances from origin: {distances}")

# Image-like data manipulation
# Simulate RGB image data (height=3, width=4, channels=3)
image = np.random.randint(0, 256, size=(3, 4, 3))
print(f"\nOriginal 'image' shape: {image.shape}")

# Adjust brightness by scaling all pixels
brightness_factor = 0.5
dimmed = (image * brightness_factor).astype(int)
print(f"Dimmed image sample:\n{dimmed[0]}")  # First row

# Apply different scaling to each color channel
channel_scales = np.array([1.2, 0.8, 0.9])  # R, G, B scales
adjusted = (image * channel_scales).clip(0, 255).astype(int)
print(f"Channel-adjusted sample:\n{adjusted[0]}")
```

7 Part 7: Practical NumPy Applications

7.1 Data Analysis Pipeline

Let's build a complete data analysis pipeline using NumPy, demonstrating how these concepts work together in practice. We'll analyze temperature data from multiple weather stations.

```
# Simulating temperature data from 5 weather stations over 7 days
np.random.seed(42)
stations = ['Station_A', 'Station_B', 'Station_C', 'Station_D', 'Station_E']
days = ['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun']

# Generate temperature data (Celsius)
# Base temperatures with daily variations
base_temps = np.array([20, 22, 19, 21, 23])  # Per station
daily_variation = np.random.randn(7, 5) * 3  # Random variation
temperatures = base_temps + daily_variation

print("Temperature data (\textdegree{}C):")
print(temperatures)

# Analysis 1: Basic statistics per station
print("\nStation Statistics:")
print(f"Mean temperatures: {temperatures.mean(axis=0)}")
print(f"Max temperatures: {temperatures.max(axis=0)}")
print(f"Min temperatures: {temperatures.min(axis=0)}")
print(f"Temperature ranges: {temperatures.max(axis=0) - temperatures.min(axis
    =0)}")

```

```
# Analysis 2: Find hottest and coldest days
daily_means = temperatures.mean(axis=1)
hottest_day = days[np.argmax(daily_means)]
coldest_day = days[np.argmin(daily_means)]
print(f"\nHottest day: {hottest_day} ({daily_means.max():.1f}\textdegree{}C average)")
print(f"Coldest day: {coldest_day} ({daily_means.min():.1f}\textdegree{}C average)")

# Analysis 3: Identify extreme temperatures
threshold_high = 25
threshold_low = 17
extreme_high = temperatures > threshold_high
extreme_low = temperatures < threshold_low

print(f"\nDays with temperatures above {threshold_high}\textdegree{}C:")
high_days, high_stations = np.where(extreme_high)
for day, station in zip(high_days, high_stations):
    print(f"  {days[day]} at {stations[station]}: {temperatures[day, station]:.1f}\textdegree{}C")

# Analysis 4: Temperature anomalies
station_means = temperatures.mean(axis=0, keepdims=True)
anomalies = temperatures - station_means
print(f"\nLargest positive anomaly: {anomalies.max():.1f}\textdegree{}C")
anomaly_pos = np.unravel_index(np.argmax(anomalies), anomalies.shape)
print(f"  Occurred on {days[anomaly_pos[0]]} at {stations[anomaly_pos[1]]}")
```

7.2 Financial Data Processing

NumPy is extensively used in financial analysis. Here's an example processing stock price data and calculating various financial metrics.

```
# Simulating stock price data
np.random.seed(42)
days = 252  # Trading days in a year
initial_price = 100
daily_returns = np.random.randn(days) * 0.02  # 2% daily volatility

# Calculate price series using cumulative product
price_multipliers = 1 + daily_returns
prices = initial_price * np.cumprod(price_multipliers)

print(f"Stock price statistics:")
print(f"Starting price: ${initial_price:.2f}")
print(f"Ending price: ${prices[-1]:.2f}")
print(f"Max price: ${prices.max():.2f}")
print(f"Min price: ${prices.min():.2f}")

# Calculate moving averages
window_short = 20
window_long = 50

# Simple moving averages
sma_short = np.convolve(prices, np.ones(window_short)/window_short, mode='valid')
sma_long = np.convolve(prices, np.ones(window_long)/window_long, mode='valid')

# Calculate daily returns from prices
price_returns = np.diff(prices) / prices[:-1]

# Risk metrics
```

```python
volatility = np.std(price_returns) * np.sqrt(252)  # Annualized
sharpe_ratio = np.mean(price_returns) * 252 / volatility
max_drawdown = np.min(prices / np.maximum.accumulate(prices) - 1)

print(f"\nRisk Metrics:")
print(f"Annual volatility: {volatility:.1%}")
print(f"Sharpe ratio: {sharpe_ratio:.2f}")
print(f"Maximum drawdown: {max_drawdown:.1%}")

# Find best and worst periods
rolling_returns = np.convolve(price_returns, np.ones(5)/5, mode='valid')
best_week = np.argmax(rolling_returns)
worst_week = np.argmin(rolling_returns)

print(f"\nBest 5-day period: Days {best_week}-{best_week+4}")
print(f"Worst 5-day period: Days {worst_week}-{worst_week+4}")
```

Summary and Best Practices

Key Takeaways:

1. NumPy arrays are the foundation of scientific computing in Python, offering 10-100x performance improvements over lists for numerical operations
2. Arrays can be created from sequences, built-in functions (zeros, ones, arange, linspace), or random generators
3. Indexing and slicing work similarly to lists but extend naturally to multiple dimensions
4. Vectorized operations eliminate the need for explicit loops, making code cleaner and faster
5. Broadcasting enables operations between arrays of different shapes following consistent rules
6. Shape manipulation (reshape, transpose, stack, split) is essential for data preprocessing
7. Always consider memory views vs. copies when slicing and reshaping arrays

Common Pitfalls to Avoid

- Forgetting that slicing creates views, not copies - modifications affect the original array
- Using Python's 'and', 'or', 'not' instead of NumPy's '&', '—', '˜' for boolean operations
- Not understanding broadcasting rules, leading to unexpected results or errors
- Creating unnecessary copies of large arrays, causing memory issues
- Using loops instead of vectorized operations, resulting in slow code
- Mixing NumPy arrays with Python lists without understanding the performance implications

Practice Exercises

Exercise 1: Grade Analysis System
Create a program that analyzes student grades across multiple subjects:

- Generate random grades (60-100) for 30 students across 5 subjects
- Calculate average grade per student and per subject
- Identify students with any failing grades (¡70)
- Find the top 5 performing students
- Calculate grade distribution statistics

Exercise 2: Image Processing Basics
Simulate basic image operations using NumPy:

- Create a 100x100 pixel ”image” with random grayscale values (0-255)
- Apply brightness and contrast adjustments
- Create a simple blur effect using averaging
- Detect edges by calculating pixel differences
- Generate a histogram of pixel intensities

Exercise 3: Sales Data Analysis
Build a sales analysis system:

- Create sales data for 12 months across 10 products
- Calculate month-over-month growth rates
- Identify best and worst performing products
- Find seasonal patterns using moving averages
- Generate a sales forecast based on trends

Next Week Preview

In Lecture 13, we'll explore advanced NumPy features including structured arrays, memory layout optimization, advanced indexing techniques, and integration with pandas for data analysis. We'll also cover performance optimization strategies and real-world applications in data science and machine learning.