Lecture 13 Handout

Advanced NumPy Operations

Linear Algebra, Random Numbers, and Applications

INF 605 - Introduction to Programming - Python

Prof. Rongyu Lin
Quinnipiac University
School of Computing and Engineering

Fall 2024

Required Reading

Textbook: Chapter 4, Sections 4.1.11, 4.2, 4.4, 4.8, 4.9, 4.10
Reference Notebooks: `ch04/04_NumPy_Advanced.ipynb` for advanced operations

Learning Objectives

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

1. **Master matrix operations and transposing** - Apply transpose, swapaxes(), and matrix multiplication with @ operator
2. **Apply broadcasting principles** - Understand broadcasting rules and use them for efficient array operations
3. **Generate and control random numbers** - Use np.random.default_rng() for reproducible random data generation
4. **Use array-oriented programming** - Apply np.where() and np.meshgrid() for vectorized logic
5. **Perform linear algebra operations** - Use matrix inverse, solve linear systems, **fit polynomials with np.polyfit()**, and **calculate correlations with np.corrcoef()**
6. **Save and load NumPy arrays** - Use np.save(), np.load(), and np.savez() for data persistence
7. **Synthesize skills in complex applications** - Implement complete random walk simulations

Prerequisites Review

Building on Your NumPy Foundation:

From Lectures 12-13, you have mastered NumPy fundamentals including array creation (`np.array()`, `np.zeros()`, `np.ones()`, `np.arange()`, `np.linspace()`), array attributes (shape,

dtype, ndim, size), basic indexing and slicing, boolean and fancy indexing, element-wise operations, reshaping operations (`reshape()`, `flatten()`, `ravel()`), universal functions (`np.sqrt()`, `np.exp()`, `np.maximum()`), and statistical methods with the axis parameter (`mean()`, `sum()`, `std()`, `cumsum()`).

This lecture builds on that foundation by introducing advanced NumPy operations that are essential for scientific computing, data analysis, and machine learning. You will learn powerful techniques including transpose operations (building on your understanding of array dimensions), broadcasting rules (extending element-wise operations), random number generation (more powerful than Python's random module), vectorized conditional logic (replacing loops with np.where()), linear algebra operations (solving systems and fitting models), and comprehensive applications that synthesize all your NumPy skills.

Connection to Previous Knowledge: Transpose builds on Lecture 13's axis understanding (axis=0 goes DOWN, axis=1 goes ACROSS). Broadcasting extends Lecture 12's element-wise operations. np.where() extends Lecture 13's boolean indexing to conditional assignment. Linear algebra operations solve mathematical problems using all your array manipulation skills.

1 Section 1: Transposing and Matrix Operations

1.1 Understanding the Transpose Operation

Transpose is a fundamental matrix operation that flips rows and columns - imagine rotating a spreadsheet 90 degrees so that what were rows become columns and vice versa. In NumPy, you use the `.T` attribute (very simple!) to transpose any array, which is essential for many mathematical operations. Think of a class roster where rows are students and columns are test scores - transposing it makes rows become different tests and columns become different students, letting you analyze from a different perspective. The transpose operation is crucial for matrix multiplication because it lets you align dimensions correctly. For a 2D array with shape (3, 5), the transpose has shape (5, 3) - the dimensions swap positions.

```
# Basic transpose with .T attribute
import numpy as np

# Create a 3x5 array
arr = np.arange(15).reshape((3, 5))
print("Original array (3 rows, 5 columns):")
print(arr)
print(f"Shape: {arr.shape}")

# Transpose - swap rows and columns
transposed = arr.T
print("\nTransposed array (5 rows, 3 columns):")
print(transposed)
print(f"Shape: {transposed.shape}")
```

The transpose operation flipped the array - what were 3 rows became 3 columns, and 5 columns became 5 rows. Notice how the shape changed from (3, 5) to (5, 3). The first row [0, 1, 2, 3, 4] became the first column in the transposed version. This operation is extremely fast because NumPy doesn't actually move data in memory - it just changes how the array is viewed.

```
# Real application - analyzing test scores by subject
# Student scores: rows are students, columns are tests
scores = np.array([[85, 92, 78],
                   [90, 88, 82],
                   [75, 80, 85],
                   [88, 92, 90]])

```

```python
print(f"Original shape (students x tests): {scores.shape}")

# Transpose: now rows are tests, columns are students
scores_by_test = scores.T
print(f"Transposed shape (tests x students): {scores_by_test.shape}")

# Now we can easily calculate statistics per test
test_averages = scores_by_test.mean(axis=1)
print(f"\nAverage score per test: {test_averages}")
```

1.2 Matrix Multiplication and Dot Products

Matrix multiplication is a fundamental operation in linear algebra where you combine two matrices following specific rules to produce a new matrix - it is not just multiplying corresponding elements! For two matrices to multiply, the number of columns in the first must equal the number of rows in the second - if A is (2,3) and B is (3,4), then A @ B produces a (2,4) result. In Python, you use the @ operator (the modern way) or `np.dot()` function (the traditional way) to multiply matrices. This operation is crucial for machine learning, computer graphics, solving systems of equations, and analyzing relationships in data. Do not confuse matrix multiplication (@) with element-wise multiplication (*) - they are completely different operations!

```python
# Matrix multiplication with @ operator
arr = np.array([[0, 1, 0],
                [1, 2, -2],
                [6, 3, 2],
                [-1, 0, -1],
                [1, 0, 1]])
# arr is 5x3

print(f"Array shape: {arr.shape}")

# Compute arr.T @ arr (transpose times original)
result = arr.T @ arr
print("Result of arr.T @ arr:")
print(result)
print(f"Result shape: {result.shape}")  # (3, 3)
```

When we multiply arr.T (3x5) by arr (5x3), we get a (3x3) square matrix. This operation (transpose times original) is very common in statistics and machine learning - it computes how each column relates to every other column. The resulting matrix is symmetric, meaning it equals its own transpose.

```python
# Practical example: calculating revenue
# Prices: 2 products in 3 stores
# Rows = products, Columns = stores
prices = np.array([[2.50, 2.75, 2.60],  # Product 1
                   [3.20, 3.00, 3.10]])  # Product 2

# Quantities sold: 3 stores, 2 days
# Rows = stores, Columns = days
quantities = np.array([[100, 120],  # Store 1
                       [150, 140],  # Store 2
                       [80, 90]])    # Store 3

# Calculate revenue: prices @ quantities
revenue = prices @ quantities
print(f"Revenue (products x days):\n{revenue}")
# Result shows total revenue for each product on each day
# First row: Product 1 revenue on day 1 and day 2
# Second row: Product 2 revenue on day 1 and day 2
```

1.3 Swapping Axes in Multidimensional Arrays

For arrays with more than 2 dimensions, `swapaxes()` gives you precise control over which dimensions to swap - it is like transpose but for any two axes you choose. While `.T` reverses all axes (in 2D, swaps rows and columns), `swapaxes()` lets you swap specific axes by number: axis 0, axis 1, axis 2, etc. This is useful for reorganizing 3D data like images (height x width x color channels) or scientific datasets (time x location x measurement). Understanding axis swapping helps you reshape data to match what different functions expect.

```
# Swapping specific axes in 3D array
arr = np.arange(16).reshape((2, 2, 4))
print("Original array (2x2x4):")
print(arr)
print(f"Shape: {arr.shape}")

# Swap axes 0 and 1
swapped = arr.swapaxes(0, 1)
print("\nAfter swapping axes 0 and 1:")
print(swapped)
print(f"Shape: {swapped.shape}")
# Shape stays (2,2,4) but data is reorganized
```

Key takeaway: Transpose (`.T`) reverses all axes, but `swapaxes(i, j)` swaps only the specified axes i and j. For most 2D work, use `.T`. For complex 3D+ arrays, `swapaxes()` gives precise control.

2 Section 2: Broadcasting Principles

2.1 Understanding Broadcasting Rules

Broadcasting is NumPy's powerful mechanism for performing operations on arrays of different shapes without creating copies - it is what makes NumPy code so concise and fast! Think of it like filling a multiplication table: when you multiply a single number by a list of numbers, the single number "broadcasts" to match the list's shape. NumPy automatically stretches smaller arrays to match larger ones during operations, following specific rules: (1) Arrays must have compatible shapes (dimensions match or one is 1), (2) Broadcasting happens along missing dimensions by repeating values. For example, adding a (3,) array to a (4,3) array broadcasts the (3,) array across all 4 rows automatically - no loops needed! Understanding broadcasting is essential for writing efficient NumPy code and avoiding unnecessary memory usage. Most shape errors you encounter are broadcasting compatibility issues!

```
# Broadcasting Compatibility Rules:
#
# Rule 1: Compare dimensions from right to left
# Rule 2: Dimensions are compatible if they are:
#         - Equal, OR
#         - One of them is 1
#
# Examples:
# (3, 4) + (4,)    Compatible! (4,) becomes (1, 4) then broadcasts to (3, 4)
# (3, 4) + (3, 1)  Compatible! (3, 1) broadcasts to (3, 4)
# (3, 4) + (3,)    ERROR! Cannot broadcast (3,) to match (3, 4)
```

```
# Basic broadcasting patterns
# Pattern 1: Scalar broadcasting
arr = np.array([[1, 2, 3],
                [4, 5, 6]])
result = arr + 10  # 10 broadcasts to (2,3)
print("Array + scalar:")
```

```
print(result)

# Pattern 2: 1D array broadcasting to 2D
row = np.array([10, 20, 30])
result = arr + row  # row broadcasts across all rows
print("\nArray + row:")
print(result)
```

In the first example, the scalar 10 broadcasts to match the array's (2,3) shape, adding 10 to every element. In the second, the (3,) row array broadcasts across both rows of the (2,3) array, adding corresponding values to each column. This is equivalent to manually copying the row twice and then adding, but NumPy does it automatically and efficiently.

```
# Broadcasting with columns
arr = np.array([[1, 2, 3],
                [4, 5, 6],
                [7, 8, 9]])

# Add different value to each row (broadcast column)
col = np.array([[10],
                [20],
                [30]])
result = arr + col
print("Array + column:")
print(result)
# [[11, 12, 13],
#  [24, 25, 26],
#  [37, 38, 39]]
```

The column array with shape (3,1) broadcasts across all columns, adding 10 to the first row, 20 to the second row, and 30 to the third row. Notice how the shape (3,1) is compatible with (3,3) because the second dimension is 1.

2.2 Practical Broadcasting - Data Normalization

One of the most common uses of broadcasting is data normalization - transforming data to have mean of 0 and standard deviation of 1 (called "standardization"). This is crucial in machine learning and statistics because it puts all features on the same scale, making comparisons fair. Using broadcasting, you can calculate column means and standard deviations, then subtract means and divide by stds in just two lines - no loops! This is exactly the kind of operation where broadcasting shines: clean, readable, fast code that operates on entire datasets at once. Understanding this pattern will help you with Assignment 4 Problem 6 and many real data analysis tasks.

```
# Normalizing data with broadcasting
# Sample data: rows = observations, columns = features
data = np.array([[1, 200, 3],
                 [4, 250, 6],
                 [7, 300, 9],
                 [10, 350, 12]])

# Calculate mean and std for each column
mean = data.mean(axis=0)  # Shape: (3,)
std = data.std(axis=0)    # Shape: (3,)

# Normalize: broadcasting makes this clean!
normalized = (data - mean) / std

print("Original data:")
print(data)
print(f"\nMeans: {mean}")
```

```
print(f"Stds: {std}")
print("\nNormalized data:")
print(normalized)
```

Broadcasting happened twice: first when subtracting mean (3,) from data (4,3), then dividing by std (3,). Each column now has mean approximately 0 and std approximately 1, making all features comparable despite originally different scales (1-10 vs 200-350). This two-line pattern - `(data - mean) / std` - is fundamental in data science.

3 Section 3: Random Number Generation

3.1 Modern Random Number API - np.random.default_rng()

NumPy's random number generation is dramatically faster than Python's built-in random module - we are talking 10-100 times faster for large arrays! The modern approach uses `np.random.default_rng()` to create a random number generator object, which then has methods for generating different types of random numbers. Think of it like creating a specialized random number machine that you can configure and use repeatedly - you create the generator once, then call its methods many times. This API replaced the old `np.random` functions (which still work but are discouraged) to give better control and reproducibility. Whether you need uniform random numbers, normally distributed data, random integers, or samples from datasets, the Generator object has optimized methods for all of these.

```
# Creating and using the random number generator
# Create random number generator with seed
rng = np.random.default_rng(seed=12345)

# Generate uniform random numbers [0, 1)
uniform = rng.random(5)
print("Uniform [0,1):", uniform)

# Generate standard normal (mean=0, std=1)
normal = rng.standard_normal(10)
print("\nStandard normal:", normal)

# Generate random integers
dice_rolls = rng.integers(1, 7, size=20)  # 20 dice rolls
print("\nDice rolls:", dice_rolls)
```

We created one generator with a seed (12345), then used it multiple times to generate different types of random numbers. The seed makes results reproducible - running this code again with same seed produces identical "random" numbers. This is essential for debugging and sharing results.

3.2 Distributions and Seeding for Reproducibility

Random number generators can produce values from many different probability distributions - uniform (all values equally likely), normal/Gaussian (bell curve), integers (discrete values), and many more. In real work, you almost always want seeded random generation for reproducibility - if you discover interesting results from a simulation, you need to be able to reproduce them exactly! Setting a seed (any number) makes the "random" sequence deterministic and repeatable, which is essential for debugging, sharing results, and scientific reproducibility. Think of the seed as a starting point for the random sequence - same seed always produces same sequence.

```
# Different distributions with seeding
rng = np.random.default_rng(seed=42)

```

```python
# Uniform distribution
uniform_data = rng.random(5)
print("Uniform [0,1):", uniform_data)

# Custom normal distribution (mean=100, std=15)
iq_scores = rng.normal(loc=100, scale=15, size=10)
print("\nSimulated IQ scores:", iq_scores.astype(int))

# Random integers from range
lottery = rng.integers(1, 50, size=6)  # 6 numbers between 1-49
print("\nLottery numbers:", np.sort(lottery))
```

```python
# Demonstrating reproducibility
# Two generators with same seed produce identical results
rng1 = np.random.default_rng(seed=999)
rng2 = np.random.default_rng(seed=999)

sample1 = rng1.random(5)
sample2 = rng2.random(5)

print("Sample 1:", sample1)
print("Sample 2:", sample2)
print("Identical?", np.array_equal(sample1, sample2))  # True!
```

The two generators with identical seeds produced exactly the same ”random” numbers. This is crucial for scientific reproducibility - anyone running your code with the same seed will get identical results.

3.3 Sampling and Permutations

Beyond generating random numbers from distributions, you often need to sample from existing data (picking random elements) or shuffle data randomly. The generator's `choice()` method lets you sample elements from an array, with or without replacement (putting them back), which is perfect for creating random train/test splits or selecting random subsets. The `shuffle()` method randomly reorders an array in place, useful for randomizing data order before processing. These operations are fundamental for machine learning workflows, statistical sampling, and any situation where you need random subsets of your data.

```python
# Sampling and shuffling operations
rng = np.random.default_rng(seed=42)

# Sample without replacement (no duplicates)
names = np.array(["Alice", "Bob", "Charlie", "David", "Eve"])
selected = rng.choice(names, size=3, replace=False)
print("Random sample (3 names):", selected)

# Shuffle array in place
arr = np.arange(10)
print("\nBefore shuffle:", arr)
rng.shuffle(arr)
print("After shuffle:", arr)
# Note: shuffle modifies the original array!
```

Use `choice()` to sample elements (with or without replacement), and `shuffle()` to randomize order. Both are essential for data science workflows like cross-validation and random sampling.

4 Section 4: Array-Oriented Programming

4.1 Vectorized Conditional Logic with np.where()

The `np.where()` function is NumPy's answer to vectorized if-else logic - instead of looping through elements one by one asking "if this, then that", `np.where()` evaluates the entire array at once and assigns values based on conditions. Think of it like Excel's IF function but applied to thousands of cells simultaneously: IF(condition, value_if_true, value_if_false). This is dramatically faster than Python loops or list comprehensions for large arrays, and the code is often more readable too. You can use `np.where()` for simple substitution (replace negatives with zero) or complex conditional transformations. This function is essential for any time you need different operations based on array values - data cleaning, categorization, or conditional calculations.

```
# Basic np.where() usage
xarr = np.array([1.1, 1.2, 1.3, 1.4, 1.5])
yarr = np.array([2.1, 2.2, 2.3, 2.4, 2.5])
cond = np.array([True, False, True, True, False])

# Without np.where() - SLOW with loops
result_slow = [(x if c else y)
               for x, y, c in zip(xarr, yarr, cond)]
print("Loop result:", result_slow)

# With np.where() - FAST and clean!
result_fast = np.where(cond, xarr, yarr)
print("np.where result:", result_fast)
# Same result, much faster!
```

`np.where(cond, xarr, yarr)` says "where condition is True, take from xarr; where False, take from yarr". The result has values from xarr at positions 0, 2, 3 (where cond is True) and values from yarr at positions 1, 4 (where cond is False).

```
# Common np.where() patterns
rng = np.random.default_rng(seed=42)
arr = rng.standard_normal((4, 4))

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

# Pattern 1: Replace positive values with 2, keep negative
result1 = np.where(arr > 0, 2, arr)
print("\nPositive to 2, negative unchanged:")
print(result1)

# Pattern 2: Replace positive with 2, negative with -2
result2 = np.where(arr > 0, 2, -2)
print("\nPositive to 2, negative to -2:")
print(result2)
```

These patterns show how `np.where()` can transform arrays based on conditions. The first pattern keeps negative values unchanged (using arr as the false value), while the second pattern assigns a constant -2 to all negative values.

4.2 Creating Coordinate Grids with np.meshgrid()

The `np.meshgrid()` function creates coordinate matrices from coordinate vectors, which is essential for evaluating functions over 2D grids - imagine creating a checkerboard where you want to calculate something at every square. You provide 1D arrays for x-coordinates and y-coordinates, and meshgrid gives you 2D arrays where every combination of (x,y) is represented.

This is fundamental for plotting 3D surfaces, creating grids for image processing, or evaluating mathematical functions over a region. For example, to evaluate the function $z = \sqrt{x^2 + y^2}$ over a grid, you first create coordinate grids with meshgrid, then apply the function using broadcasting.

```
# Creating and using meshgrid
# Create 1D arrays for coordinates
points = np.arange(-5, 5, 0.1)  # 100 points from -5 to 5
print(f"Points shape: {points.shape}")  # (100,)

# Create 2D coordinate grids
xs, ys = np.meshgrid(points, points)
print(f"xs shape: {xs.shape}")  # (100, 100)
print(f"ys shape: {ys.shape}")  # (100, 100)

# Calculate function over grid: distance from origin
z = np.sqrt(xs**2 + ys**2)
print(f"z shape: {z.shape}")  # (100, 100)
# Now z contains the distance from origin at every (x,y) point
```

Meshgrid converted two 1D arrays (100,) into two 2D arrays (100, 100) where every (x,y) coordinate pair is represented. We then calculated $z = \sqrt{x^2 + y^2}$ over the entire grid using broadcasting - this evaluated the function at 10,000 points without any loops!

5 Section 5: Linear Algebra Operations

5.1 Matrix Inverse and Solving Linear Systems

Linear algebra operations in NumPy solve systems of equations and perform matrix calculations you learned in algebra class - but now working with large matrices automatically! The matrix inverse is like division for matrices: just as dividing by x is the same as multiplying by 1/x, multiplying by a matrix inverse "undoes" the original matrix. NumPy's `np.linalg.inv()` computes matrix inverses, and `np.linalg.solve()` efficiently solves systems of linear equations (Ax = b) without explicitly computing the inverse. These operations are fundamental in many fields: solving circuit equations in engineering, fitting models in statistics, transforming coordinates in graphics, and optimization in machine learning.

```
# Matrix inverse
rng = np.random.default_rng(seed=42)

# Create symmetric positive definite matrix
X = rng.standard_normal((5, 5))
mat = X.T @ X  # This creates a matrix we can invert

# Calculate inverse
inv_mat = np.linalg.inv(mat)

# Verify: matrix times inverse should be identity
identity = mat @ inv_mat
print("mat @ inv(mat) should be identity:")
print(identity)
# Should be very close to identity matrix (1s on diagonal, 0s elsewhere)
```

The matrix multiplied by its inverse produces an identity matrix (like multiplying a number by its reciprocal gives 1). Small rounding errors cause diagonal to be approximately 1.0 instead of exactly 1.0, and off-diagonal approximately 0.0 instead of exactly 0.0.

```
# Solving linear systems
# Solve system of equations:
# 3x + y = 9
```

```python
# x + 2y = 8

# Matrix form: Ax = b
A = np.array([[3, 1],
              [1, 2]])
b = np.array([9, 8])

# Solve for x
x = np.linalg.solve(A, b)
print(f"Solution: x = {x}")

# Verify solution
print(f"Check: A @ x = {A @ x}")  # Should equal b
```

The solution x = [2, 3] satisfies both equations: $3(2) + 3 = 9$ and $2 + 2(3) = 8$. Always use `np.linalg.solve()` instead of computing inverse explicitly - it is more efficient and numerically stable.

5.2 Polynomial Fitting with np.polyfit() - CRITICAL FOR ASSIGNMENT 4

Polynomial fitting is one of the most common data analysis tasks - finding the best polynomial curve that matches your data points. NumPy's `np.polyfit()` does this automatically using least squares fitting (minimizing the squared errors between the curve and data points). You provide x-values, y-values, and the polynomial degree (1 for linear, 2 for quadratic, 3 for cubic, etc.), and it returns the coefficients of the polynomial that best fits your data. This is fundamental for trend analysis, making predictions, and understanding relationships in data - is the relationship linear? curved? exponential-looking? Polynomial fitting is used everywhere: economics (trend forecasting), physics (fitting experimental data), machine learning (polynomial regression), and Assignment 4 Problem 7 explicitly requires using `np.polyfit()`!

```python
# Linear fit (degree 1)
# Sample data with linear trend
x_data = np.array([1, 2, 3, 4, 5])
y_data = np.array([2.1, 3.9, 6.2, 8.1, 9.8])

# Fit polynomial of degree 1 (linear: y = mx + b)
coefficients = np.polyfit(x_data, y_data, deg=1)
slope, intercept = coefficients

print(f"Best fit line: y = {slope:.2f}x + {intercept:.2f}")

# Make predictions
x_new = 6
y_pred = slope * x_new + intercept
print(f"Prediction at x=6: {y_pred:.2f}")
```

`polyfit()` found the line that best fits the data points. The slope approximately 2 and intercept approximately 0 mean the relationship is approximately y = 2x. We can use these coefficients to predict y-values for new x-values not in the original data.

```python
# Quadratic fit (degree 2)
# Data with curved relationship
x_data = np.array([0, 1, 2, 3, 4, 5])
y_data = np.array([1, 2.5, 5.8, 10.9, 17.8, 26.5])

# Fit polynomial of degree 2 (quadratic: y = ax^2 + bx + c)
coefficients = np.polyfit(x_data, y_data, deg=2)
a, b, c = coefficients

```

```
print(f"Best fit curve: y = {a:.2f}x^2 + {b:.2f}x + {c:.2f}")

# Evaluate polynomial at new points
x_new = np.array([6, 7, 8])
y_pred = a*x_new**2 + b*x_new + c
print(f"Predictions: {y_pred}")
```

For curved data, a quadratic polynomial (degree 2) provides a better fit than a straight line. The three coefficients (a, b, c) define the parabola that best matches the data points. This pattern - fitting, extracting coefficients, making predictions - is exactly what Assignment 4 Problem 7 requires.

5.3 Correlation Analysis with np.corrcoef() - CRITICAL FOR ASSIGNMENT 4

Correlation measures how strongly two variables are related - do they increase together? decrease together? or are they unrelated? NumPy's `np.corrcoef()` calculates correlation coefficients between variables, producing a correlation matrix showing all pairwise relationships. Correlation values range from -1 (perfect negative correlation) through 0 (no correlation) to +1 (perfect positive correlation). For example, height and weight have positive correlation (taller people tend to weigh more), while temperature and heating bills have negative correlation (higher temperature means lower bills). Understanding correlation is fundamental for data analysis - identifying related variables, finding redundant features, or discovering hidden relationships. Assignment 4 Problem 7 requires using `np.corrcoef()` to analyze relationships in data!

```
# Basic correlation analysis
# Two variables: study hours and test scores
study_hours = np.array([2, 3, 4, 5, 6, 7, 8])
test_scores = np.array([65, 70, 75, 80, 85, 88, 92])

# Calculate correlation matrix
correlation_matrix = np.corrcoef(study_hours, test_scores)
print("Correlation matrix:")
print(correlation_matrix)

# Extract correlation coefficient (off-diagonal)
correlation = correlation_matrix[0, 1]
print(f"\nCorrelation between study hours and scores: {correlation:.3f}")
# Interpretation: value near 1 means strong positive relationship
```

The correlation matrix is 2x2: diagonal elements are 1.0 (each variable correlates perfectly with itself), and off-diagonal elements show correlation between the two variables. A value near 1.0 indicates strong positive correlation - more study hours are associated with higher scores.

```
# Multiple variables correlation
# Three variables: temperature, ice cream sales, sunglasses sales
temp = np.array([70, 75, 80, 85, 90, 95])
ice_cream = np.array([100, 120, 150, 180, 220, 250])
sunglasses = np.array([50, 55, 65, 75, 90, 100])

# Stack variables as rows
data = np.array([temp, ice_cream, sunglasses])

# Calculate all pairwise correlations
corr_matrix = np.corrcoef(data)
print("Correlation matrix:")
print(corr_matrix)
# All three variables positively correlated (all increase together)
```

With three variables, the correlation matrix is 3x3, showing all nine pairwise correlations. The diagonal is still 1.0, and off-diagonal elements show how each pair of variables relates. All three variables show strong positive correlation because they all tend to increase together with warmer weather.

5.4 Other Linear Algebra Functions

NumPy's linalg module includes many other linear algebra operations you might encounter: determinant (`np.linalg.det`), eigenvalues/eigenvectors (`np.linalg.eig`), QR decomposition (`np.linalg.qr`), and SVD (`np.linalg.svd`). These advanced operations are crucial in machine learning and scientific computing but beyond our current scope - just know they exist for future reference. For this course, focus on matrix multiplication (@), inverse, solve, polyfit, and corrcoef - these cover 90 percent of practical linear algebra needs.

```
# Other operations available (awareness only):
# np.linalg.det(A)        # Determinant
# np.linalg.eig(A)        # Eigenvalues and eigenvectors
# np.linalg.qr(A)         # QR decomposition
# np.linalg.svd(A)        # Singular Value Decomposition
# np.linalg.matrix_rank(A) # Matrix rank

# For this course, focus on:
# - Matrix multiplication: @
# - Solving systems: np.linalg.solve()
# - Polynomial fitting: np.polyfit()
# - Correlation: np.corrcoef()
```

6 Section 6: File I/O for Arrays

6.1 Saving and Loading Single Arrays

After spending hours processing data and creating NumPy arrays, you do not want to lose your work or recompute everything next time! NumPy provides efficient binary file formats (.npy) for saving and loading arrays - much faster and more space-efficient than text formats like CSV. Use `np.save()` to save a single array to disk, and `np.load()` to load it back - NumPy automatically preserves the exact shape, data type, and values. Think of it like saving a Word document - you can close your program, come back later, and pick up exactly where you left off.

```
# Save and load single array
# Create array
arr = np.arange(10)
print("Original array:", arr)

# Save to file (automatically adds .npy extension)
np.save("my_array", arr)
print("Array saved to 'my_array.npy'")

# Load from file
loaded = np.load("my_array.npy")
print("Loaded array:", loaded)
print("Arrays equal?", np.array_equal(arr, loaded))  # True
```

`np.save()` writes the array in NumPy's efficient binary format. `np.load()` reads it back perfectly - same values, same shape, same dtype. The .npy file is much smaller and faster to read than text formats.

6.2 Saving Multiple Arrays with np.savez()

Often you need to save multiple related arrays together - think of input data, processed results, and calculated statistics all belonging to one analysis. NumPy's `np.savez()` creates compressed archives (like zip files) that store multiple arrays with named keys, and `np.savez_compressed()` adds extra compression for large files. You load the archive with `np.load()` and access individual arrays by their names like a dictionary.

```
# Save multiple arrays together
# Create multiple arrays
arr1 = np.arange(10)
arr2 = np.random.randn(5, 3)

# Save multiple arrays with names
np.savez("array_archive.npz", first=arr1, second=arr2)
print("Multiple arrays saved to 'array_archive.npz'")

# Load archive
archive = np.load("array_archive.npz")

# Access by name
loaded_arr1 = archive["first"]
loaded_arr2 = archive["second"]

print("First array:", loaded_arr1)
print("Second array shape:", loaded_arr2.shape)

# For large arrays, use compressed version:
np.savez_compressed("compressed_archive.npz", first=arr1, second=arr2)
```

Use `np.save()` for single arrays, `np.savez()` for multiple arrays, and `np.savez_compressed()` when file size matters. This preserves your work and makes analysis reproducible.

7 Section 7: Comprehensive Example - Random Walks

7.1 Understanding Random Walks

Random walks are a fundamental model in mathematics, physics, finance, and computer science - imagine flipping a coin repeatedly and taking a step left or right based on heads or tails. This simple process models everything from stock prices to molecular motion to gamblers' fortunes! What makes this example amazing is how NumPy lets us simulate not just one walk, but THOUSANDS simultaneously with just a few lines of code. We will start with a Python loop version (slow, many lines), then show the NumPy version (fast, few lines), then simulate 5000 walks at once and analyze them all together. This demonstrates everything we have learned: random numbers, cumsum, boolean operations, axis parameter, argmax, broadcasting - all working together in a real, meaningful application!

7.2 Pure Python Implementation

Let us first implement a random walk the "traditional" Python way using loops - this helps us understand what a random walk is before we accelerate it with NumPy. We will flip a coin 1000 times (random.randint), take steps based on results, and track our position over time. This version is clear but slow - it would take forever to simulate thousands of walks.

```
# Python loop version
import random

# Start at position 0
position = 0
```

```
walk = [position]
nsteps = 1000

# Take 1000 random steps
for _ in range(nsteps):
    step = 1 if random.randint(0, 1) else -1
    position += step
    walk.append(position)

print(f"Final position: {walk[-1]}")
print(f"Maximum position: {max(walk)}")
print(f"Minimum position: {min(walk)}")
```

After 1000 random steps, we ended at some random position. The walk randomly drifts away from origin - sometimes reaching high values, sometimes low. This took Python many loop iterations to compute.

7.3 NumPy Implementation - Single Walk

Now let us implement the same random walk using NumPy - watch how much shorter and faster this is! Instead of a loop, we generate all 1000 random numbers at once, convert them to steps with `np.where()`, then use `cumsum()` to calculate positions. This is 10-100 times faster than the loop version and demonstrates the power of vectorized thinking: generate all random data, transform it, accumulate it - no loops needed!

```
# NumPy version - single walk
rng = np.random.default_rng(seed=12345)
nsteps = 1000

# Generate all random values at once
draws = rng.integers(0, 2, size=nsteps)  # 1000 random 0s and 1s

# Convert to steps: 0 -> +1, 1 -> -1
steps = np.where(draws == 0, 1, -1)

# Calculate cumulative position (this is the walk!)
walk = steps.cumsum()

print(f"Final position: {walk[-1]}")
print(f"Maximum position: {walk.max()}")
print(f"Minimum position: {walk.min()}")

# Find first crossing time for plus/minus 10
crossing_time = (np.abs(walk) >= 10).argmax()
print(f"First crossed +/-10 at step: {crossing_time}")
```

We generated the entire walk in just 4 lines of NumPy code! `cumsum()` calculated all 1000 positions at once. `argmax()` on boolean array finds the first True value - this tells us when the walk first crossed the plus/minus 10 threshold.

7.4 Simulating 5000 Walks Simultaneously

Here is where NumPy truly shines: we can simulate 5000 different random walks SIMULTANEOUSLY using 2D arrays! Instead of a 1D array of 1000 steps, we create a 2D array with shape (5000, 1000) - 5000 walks, each with 1000 steps. We use `cumsum(axis=1)` to accumulate along the steps dimension (axis=1), computing all 5000 walks at once. Then we analyze all walks together: how many crossed plus/minus 30? what was the average crossing time? This would take HOURS with Python loops but runs in seconds with NumPy!

```
# Simulate 5000 walks simultaneously
```

```
nwalks = 5000
nsteps = 1000

# Generate ALL random draws at once: 5000 x 1000 array!
draws = rng.integers(0, 2, size=(nwalks, nsteps))

# Convert all to steps
steps = np.where(draws > 0, 1, -1)

# Calculate ALL 5000 walks simultaneously!
walks = steps.cumsum(axis=1)  # cumsum along steps (axis=1)

print(f"Walks array shape: {walks.shape}")  # (5000, 1000)
print(f"Maximum any walk reached: {walks.max()}")
print(f"Minimum any walk reached: {walks.min()}")
```

```
# Analyze the 5000 walks
# How many walks crossed +/-30?
hits30 = (np.abs(walks) >= 30).any(axis=1)
print(f"\nOut of {nwalks} walks:")
print(f"  {hits30.sum()} walks crossed +/-30")
print(f"  {nwalks - hits30.sum()} walks stayed within +/-30")

# Average crossing time for walks that crossed
crossing_times = (np.abs(walks[hits30]) >= 30).argmax(axis=1)
print(f"  Average crossing time: {crossing_times.mean():.1f} steps")

# Distribution of crossing times
print(f"  Fastest crossing: {crossing_times.min()} steps")
print(f"  Slowest crossing: {crossing_times.max()} steps")
```

We simulated 5 MILLION individual steps (5000 walks times 1000 steps) in under a second! The analysis shows fascinating statistical patterns: roughly 3400 walks crossed plus/minus 30, taking an average of approximately 500 steps to do so. This demonstrates NumPy's incredible power - what would take hours with loops takes seconds with vectorization.

7.5 Synthesis: What We Demonstrated

Key points from the random walks example:

1. **Random generation:** Created 5 million random numbers instantly
2. **Broadcasting:** Applied `np.where()` to entire 2D array at once
3. **Axis operations:** Used `cumsum(axis=1)` to process each walk independently
4. **Boolean operations:** Found crossing events with `any()` along axis=1
5. **Advanced indexing:** Selected only walks that crossed with `walks[hits30]`
6. **Analysis:** Used `argmax()`, `mean()`, `min()`, `max()` to understand results

Real-world applications of random walks:

- **Finance:** Monte Carlo simulation for option pricing
- **Physics:** Brownian motion and diffusion models
- **Biology:** Random mutations and genetic drift
- **Computer Science:** Algorithm analysis and randomized algorithms

- **Statistics:** Understanding random processes and probability

The power of NumPy: Pure Python with loops takes approximately 30-60 seconds for this simulation. NumPy vectorized version takes approximately 0.5-1 seconds. Speed improvement: 30-60 times faster!

This is why NumPy is essential for data science and scientific computing - it lets you work with large datasets efficiently, writing clear, concise code that runs at nearly the speed of compiled languages.

Key Takeaways

Essential Concepts to Remember:

- Transpose is simple: Just use `.T` for 2D arrays
- Broadcasting is powerful: Eliminates loops, makes code clean
- Modern random API: `np.random.default_rng()` is the new standard
- `np.where()` is vectorized if-else: Fundamental for array-oriented thinking
- `np.polyfit()` and `np.corrcoef()`: Essential for Assignment 4 Problem 7!
- Random walks synthesis: Demonstrates everything learned, shows NumPy's power
- Real applications: Connect every technique to practical uses

Connection to Assignment 4

Direct Assignment 4 Support:

- **Problem 4:** Reshaping and transpose operations
- **Problem 5:** Boolean filtering with percentiles
- **Problem 6:** Normalization with broadcasting
- **Problem 7:** Polynomial fitting with `np.polyfit()` and correlation with `np.corrcoef()`
- **Problem 8:** Comprehensive analytics combining all techniques

Preview of Next Lecture

In the next lecture, we will transition from NumPy arrays to pandas DataFrames - pandas builds on NumPy, adding labels, handling mixed types, managing missing data, and providing powerful tools for real-world data analysis. All your NumPy skills transfer directly to pandas!