# Lecture 15 Handout

NumPy Data Manipulation

Filtering, Statistics, and Sorting

INF 605 - Introduction to Programming - Python

Prof. Rongyu Lin
Quinnipiac University
School of Computing and Engineering

Fall 2024

## Required Reading

**Textbook:** Chapter 4, Sections 4.1.9-4.1.10, 4.3, 4.5, 4.6-4.7
**Reference Notebooks:** `ch04/pydata-book_ch04.ipynb` cells 40-90 for data manipulation examples

## Learning Objectives

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

1. **Master Boolean Indexing** for powerful data filtering using comparison operators and logical combinations
2. **Apply Fancy Indexing** for advanced element selection and understand copies vs views
3. **Utilize Universal Functions** efficiently for fast element-wise operations
4. **Perform Statistical Analysis** with axis operations (CRITICAL SKILL for data analysis)
5. **Sort and Find Unique Values** using both in-place and copy-based approaches
6. **Combine Techniques** for real data analysis workflows preparing for Assignment 4

## Prerequisites Review

**Building on Your NumPy Foundation:**

From Lecture 12, you mastered NumPy array creation, basic indexing and slicing, element-wise operations, and array reshaping. You understand how NumPy arrays provide 10-100x speed improvement over Python lists for numerical operations, and you've worked with array attributes like shape, dtype, and ndim.

This lecture advances your NumPy skills to the next level: filtering data with boolean masks, selecting non-contiguous elements with fancy indexing, calculating statistics across dimensions with the axis parameter, and sorting data. These are the fundamental operations for any data analysis task you'll encounter in real-world applications.

**Transformation Goal:** Evolve from **basic array operations** to **sophisticated data filtering, analysis, and manipulation techniques**.

## 1 Part 1: Boolean Indexing - Powerful Data Filtering

### 1.1 Creating Boolean Masks with Comparison Operators

Boolean indexing is one of NumPy's most powerful features for working with real data. Imagine you have a spreadsheet with thousands of rows and you want to see only the rows where sales exceeded 1000 dollars - instead of manually checking each row, you create a filter that automatically shows only matching rows. In NumPy, we create boolean masks using comparison operators (>, <, ==, ! =, >=, <=) which produce True/False arrays that match the original array's shape. Each True value in the mask means "include this element" and each False means "skip this element". This technique is essential for data analysis - filtering datasets, finding outliers, and selecting subsets of data based on conditions.

```python
# Create sample data
import numpy as np
names = np.array(["Bob", "Joe", "Will", "Bob", "Will", "Joe", "Joe"])
data = np.array([[4, 7], [0, 2], [-5, 6], [0, 0], [1, 2], [-12, -4], [3, 4]])

# Create boolean mask - which elements equal "Bob"?
mask = names == "Bob"
print(mask)  # [True False False True False False False]

# Use mask to filter data
bob_data = data[mask]
print(bob_data)
# [[4 7]
#  [0 0]]
```

**Result Explanation:** The comparison operator creates a boolean array showing where "Bob" appears. When we use this mask to index the data array, NumPy returns only the rows where the mask is True - giving us Bob's data rows. Think of this like a spotlight that illuminates only the elements you want to see, leaving everything else in darkness.

```python
# Filtering with numeric comparisons
scores = np.array([78, 92, 85, 67, 95, 73, 88])

# Find passing grades (>= 70)
passing_mask = scores >= 70
passing_scores = scores[passing_mask]
print(f"Passing scores: {passing_scores}")
# Passing scores: [78 92 85 95 73 88]
```

**Practice Exercise:**

```python
# Given temperatures array
temperatures = np.array([68, 75, 82, 79, 71, 85, 88])

# Exercise: Filter temperatures above 80 degrees
# Your solution here:
```

**Solution:**

```python
# Solution
hot_days_mask = temperatures > 80
hot_temperatures = temperatures[hot_days_mask]
print(hot_temperatures)  # [82 85 88]

# Explanation: We create a mask for temperatures > 80,
# then use it to get only the hot day temperatures
```

## 1.2 Combining Conditions with Logical Operators

In real data analysis, you often need to filter based on multiple criteria simultaneously - like finding students who scored between 80 and 90, or temperatures that are either very hot or very cold. NumPy provides logical operators to combine multiple boolean conditions: & (and), | (or), and ∼ (not). Think of it like building complex database queries: ”show me records where condition1 AND condition2” or ”condition1 OR condition2”. **IMPORTANT:** You must use & and | instead of Python’s ”and” and ”or” keywords for NumPy arrays, and you need parentheses around each condition because of operator precedence. This is one of the most common mistakes students make - remember: parentheses are required!

```python
# Combining with OR
names = np.array(["Bob", "Joe", "Will", "Bob", "Will", "Joe", "Joe"])
data = np.array([[4, 7], [0, 2], [-5, 6], [0, 0], [1, 2], [-12, -4], [3, 4]])

mask = (names == "Bob") | (names == "Will")
print(mask)  # [True False True True True False False]
selected_data = data[mask]
print(selected_data)
```

```python
# Combining with AND
scores = np.array([78, 92, 85, 67, 95, 73, 88])

# Both conditions must be true
good_range = (scores >= 80) & (scores <= 90)
range_scores = scores[good_range]
print(f"Scores 80-90: {range_scores}")
# Scores 80-90: [85 88]
```

**Common Pitfall Warning:**

```python
# WRONG - This will cause an error!
mask = scores >= 80 and scores <= 90  # Error: use & not 'and'

# WRONG - Missing parentheses
mask = scores >= 80 & scores <= 90    # Error: wrong precedence

# CORRECT - Use & with parentheses
mask = (scores >= 80) & (scores <= 90)  # Works perfectly!
```

## 1.3 Modifying Values with Boolean Indexing

Boolean indexing isn’t just for filtering - you can also use it to modify specific elements in place based on conditions. This is incredibly useful for data cleaning: replacing negative values with zero, capping outliers at a maximum value, or correcting invalid data. Think of it like a sophisticated find-and-replace that works with any condition you can express. The pattern is simple: `arr[condition] = new_value`, and NumPy will replace all elements where the condition is True. This modifies the original array, so be careful - make a copy first if you need to preserve the original data!

```python
# Replace all negative values with 0
data = np.array([[4, 7], [0, 2], [-5, 6], [0, 0], [1, 2], [-12, -4], [3, 4]])
print("Before:", data)

data[data < 0] = 0
print("After:", data)
# All negative values are now 0
# Cap test scores at maximum of 100
scores = np.array([78, 92, 105, 67, 98, 110, 88])
print(f"Original: {scores}")

scores[scores > 100] = 100
print(f"Capped: {scores}")
# [78 92 100 67 98 100 88]
```

**Advanced Example:**

```python
# Replace error values (-999) with mean of valid data
temps = np.array([72, -999, 85, 90, -999, 78, 82])
valid_temps = temps[temps != -999]
mean_temp = valid_temps.mean()
temps[temps == -999] = mean_temp
print(temps)  # Error values replaced with mean
```

## 2 Part 2: Fancy Indexing - Advanced Element Selection

### 2.1 Integer Array Indexing Basics

Fancy indexing allows you to select multiple elements from an array using an array of indices - it's like having a shopping list of specific positions you want to pick from a shelf. Unlike slicing (which selects contiguous elements), fancy indexing lets you grab elements from any positions in any order: ”give me elements 5, 2, 8, 1” - they don't have to be next to each other! This is incredibly powerful for reordering data, selecting specific rows based on rankings, or picking non-contiguous samples. **IMPORTANT:** Fancy indexing creates a copy of the data, not a view, so modifying the result won't affect the original array - this is different from slicing which creates views.

```python
# Create array with row numbers as values
arr = np.zeros((8, 4))
for i in range(8):
    arr[i] = i

print(arr)

# Select specific rows in specific order
selected = arr[[4, 3, 0, 6]]
print(selected)
# Rows 4, 3, 0, 6 in that order
```

```python
# Selecting from 1D array
scores = np.array([78, 92, 85, 67, 95, 73, 88, 91])

# Select scores at positions 1, 4, 7 (2nd, 5th, 8th students)
indices = np.array([1, 4, 7])
selected_scores = scores[indices]
print(selected_scores)  # [92 95 91]
```

### 2.2 Fancy Indexing in 2D Arrays

Fancy indexing becomes even more powerful with 2D arrays, where you can select specific rows, specific elements, or even create rectangular selections. When you provide two index arrays for a 2D array, NumPy pairs them up: first index array specifies rows, second specifies columns, and it selects elements at (row[0], col[0]), (row[1], col[1]), etc. This is perfect for selecting diagonal elements, specific matrix entries, or creating custom data selections. Understanding the difference between `arr[[1,2,3]]` (selecting entire rows) and `arr[[1,2,3], [0,1,2]]` (selecting specific row,col pairs) is crucial for working with real datasets.

```python
# Create 2D array
arr = np.arange(32).reshape((8, 4))
print(arr)

# Select specific row,col pairs
# Gets elements at (1,0), (5,3), (7,1), (2,2)
result = arr[[1, 5, 7, 2], [0, 3, 1, 2]]
print(result)  # [4 23 29 10]
```

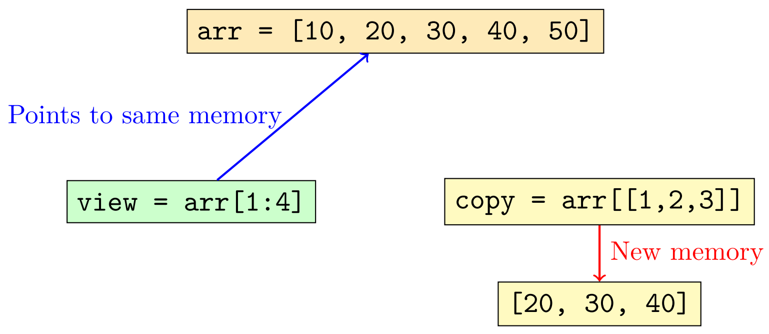
```python
# Rectangular selection - complete rows, then specific columns
arr = np.arange(32).reshape((8, 4))

# Select rows 1, 5, 7, 2, then columns 0, 3, 1, 2 from those rows
result = arr[[1, 5, 7, 2]][:, [0, 3, 1, 2]]
print(result)
```

## 2.3 Understanding Copies vs Views - Memory Management

One of the most important differences between fancy indexing and slicing is how they handle memory: slicing creates a view (a window into the original data), while fancy indexing creates a copy (completely new data in memory). This matters because modifying a view changes the original array, but modifying a copy doesn't affect the original. Think of a view like looking at your room through a window - changes you make through the window affect the actual room. A copy is like taking a photograph - you can draw on the photo but the room stays the same. For data analysis, views are more memory efficient, but copies give you independence to modify without side effects.

```python
arr = np.array([10, 20, 30, 40, 50])

# Slicing creates a view
view = arr[1:4]
view[0] = 999
print(arr)  # [10 999 30 40 50] - original changed!

# Fancy indexing creates a copy
arr = np.array([10, 20, 30, 40, 50])
copy = arr[[1, 2, 3]]
copy[0] = 999
print(arr)  # [10 20 30 40 50] - original unchanged!
```

**Key Memory Diagram:**



**Key Takeaway:** Always remember: slicing creates views (shares memory), fancy indexing creates copies (independent memory). Choose based on whether you want changes to affect the original.

# 3 Part 3: Universal Functions - Fast Element-wise Operations

## 3.1 Unary Universal Functions

Universal functions (ufuncs) are highly optimized C functions that operate on NumPy arrays element by element, making them 10-100 times faster than Python loops for mathematical operations. Think of them as assembly-line workers specialized in one specific task - they're incredibly efficient because they do one thing perfectly. Common unary ufuncs (operating on single arrays) include `np.sqrt()` for square roots, `np.exp()` for exponential function, `np.log()` for natural logarithm, `np.abs()` for absolute value, and `np.sign()` for getting signs of numbers. These are essential for data transformations: normalizing data, calculating distances, working with exponential growth models, and many scientific computations. The speed difference is dramatic - for a million elements, a ufunc might take milliseconds while a Python loop takes seconds!

```python
# Create sample array
arr = np.arange(10)
print(arr)

# Apply square root
print(np.sqrt(arr))

# Apply exponential function
print(np.exp(arr))
```

```python
# Real data transformation
celsius = np.array([0, 10, 20, 30, 40])

# Convert to Fahrenheit: F = C * 9/5 + 32
fahrenheit = celsius * 9/5 + 32
print(f"Celsius: {celsius}")
print(f"Fahrenheit: {fahrenheit}")
```

**Common Unary Ufuncs:**

| Function | Description |
|---|---|
| `np.sqrt(x)` | Square root of each element |
| `np.exp(x)` | Exponential ($e^x$) of each element |
| `np.log(x)` | Natural logarithm of each element |
| `np.abs(x)` | Absolute value of each element |
| `np.sign(x)` | Sign of each element (-1, 0, or 1) |
| `np.sin(x)`, `np.cos(x)` | Trigonometric functions |
| `np.ceil(x)`, `np.floor(x)` | Rounding up/down |

## 3.2 Binary Universal Functions

Binary ufuncs operate on two arrays simultaneously, performing element-wise operations between corresponding elements. The most useful binary ufuncs are `np.maximum()` and `np.minimum()` which compare two arrays element by element and take the larger or smaller value at each position - perfect for capping values, finding element-wise bounds, or combining datasets. Don't confuse `np.maximum()` with `arr.max()` - `np.maximum()` compares two arrays element by element, while `arr.max()` finds the single largest value in an array. These functions are essential for data cleaning (capping outliers) and combining multiple data sources (taking the best of multiple measurements).

```python
# Create two arrays
x = np.array([1.5, 2.3, 0.8, 4.2])
```

```python
y = np.array([2.1, 1.8, 3.0, 3.9])

# Element-wise maximum - compare each pair
result = np.maximum(x, y)
print(result)  # [2.1, 2.3, 3.0, 4.2]
```

```python
# Practical application - data validation
sensor1 = np.array([98.6, 102.5, 99.1, 104.0])
sensor2 = np.array([98.8, 98.5, 99.3, 103.8])

# Take minimum reading (conservative estimate)
readings = np.minimum(sensor1, sensor2)
print(f"Conservative readings: {readings}")
```

### 3.3 The Out Parameter for Memory Efficiency

Most ufuncs support an ’out’ parameter that lets you specify where to store the result, avoiding creation of new arrays and saving memory. Instead of creating a new array for the result, NumPy writes directly into an existing array you provide. This is especially important when working with large datasets where memory is limited - you can reuse arrays instead of creating new ones. Think of it like reusing containers instead of getting new ones every time - more efficient and environmentally friendly! This technique is common in production code where performance and memory usage matter.

```python
# Create arrays
arr = np.array([1.0, 2.0, 3.0, 4.0, 5.0])
result = np.zeros_like(arr)

# Add 10 to arr, store in existing result array
np.add(arr, 10, out=result)
print(result)  # [11. 12. 13. 14. 15.]

# No new array created - memory efficient!
```

## 4 Part 4: Mathematical and Statistical Methods

### 4.1 Basic Aggregation Methods

NumPy arrays have built-in methods for statistical analysis that work on entire arrays or along specific dimensions. These aggregation methods collapse array data into summary statistics: `mean()` calculates the average, `sum()` adds all elements, `std()` finds standard deviation (spread of data), `var()` calculates variance, `min()` and `max()` find extremes. Think of these as asking questions about your data: ”What’s typical?” (mean), ”What’s the total?” (sum), ”How spread out is it?” (std), ”What are the bounds?” (min/max). These are the foundation of data analysis - you’ll use them constantly to understand datasets, compare groups, and identify patterns.

```python
# Student test scores
scores = np.array([78, 92, 85, 67, 95, 73, 88, 91])

print(f"Average score: {scores.mean():.1f}")
print(f"Highest score: {scores.max()}")
print(f"Lowest score: {scores.min()}")
print(f"Score spread (std): {scores.std():.1f}")
```

```python
# Daily website visitors for a week
visitors = np.array([1245, 1532, 1423, 1678, 1891, 2134, 1987])

```

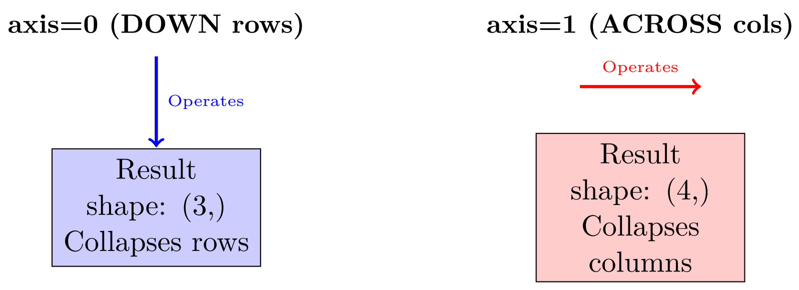
```python
total = visitors.sum()
average = visitors.mean()
visit_range = visitors.max() - visitors.min()

print(f"Total visitors: {total}")
print(f"Average per day: {average:.1f}")
print(f"Range: {visit_range}")
```

## 4.2 The Axis Parameter - CRITICAL CONCEPT

The axis parameter is one of NumPy's most powerful but confusing features - mastering it is essential for data analysis! Think of a 2D array as a table: **axis=0 goes DOWN the rows** (operating on each column), while **axis=1 goes ACROSS the columns** (operating on each row). A good memory trick: "axis=0 collapses rows, leaving columns" and "axis=1 collapses columns, leaving rows". For a table of test scores where rows are students and columns are tests: `arr.mean(axis=0)` gives the average score on each test (across all students), while `arr.mean(axis=1)` gives each student's average score (across all tests). If you don't specify axis, the operation works on the entire array flattened. Take time to understand this - it's crucial for Assignment 4 and all future data analysis!

**Visual Guide - Understanding Axis Operations:**

For array with shape (4 rows, 3 columns):


axis=0 (DOWN rows)
Operates
Result shape: (3,) Collapses rows
axis=1 (ACROSS cols)
Operates
Result shape: (4,) Collapses columns

```python
# Test scores: rows=students, columns=tests
arr = np.array([[85, 92, 78, 88],
                [72, 68, 75, 70],
                [95, 98, 92, 97],
                [88, 85, 90, 87]])

# axis=0: Mean of each column (each test's average across students)
test_averages = arr.mean(axis=0)
print(f"Test averages: {test_averages}")
# [85.0, 85.75, 83.75, 85.5]

# axis=1: Mean of each row (each student's average across tests)
student_averages = arr.mean(axis=1)
print(f"Student averages: {student_averages}")
# [85.75, 71.25, 95.5, 87.5]
```

**Memory Aid Table:**

| Axis | Direction | Result |
|---|---|---|
| `axis=0` | DOWN rows (vertical) | One value per column |
| `axis=1` | ACROSS columns (horizontal) | One value per row |
| `axis=None` | Entire array (flatten) | Single value |

```python
# Sales data: rows=months, columns=products
sales = np.array([[1200, 1500, 1800],
                  [1350, 1600, 1750],
                  [1400, 1550, 1900],
                  [1500, 1650, 1950]])

# Total sales per product (down months)
product_totals = sales.sum(axis=0)
print(f"Product totals: {product_totals}")
# [5450, 6300, 7400]

# Total sales per month (across products)
month_totals = sales.sum(axis=1)
print(f"Month totals: {month_totals}")
# [4500, 4700, 4850, 5100]
```

## 4.3 Cumulative Operations

Cumulative operations build up results progressively: `cumsum()` creates a running total where each element is the sum of all previous elements plus the current one, while `cumprod()` does the same with multiplication. Think of `cumsum()` like a bank account balance where each entry adds to the previous total, or like tracking your total miles walked where each day adds to your cumulative distance. These are perfect for calculating running totals, tracking accumulated values over time, or understanding how quantities build up. In finance, `cumsum()` is used for cumulative returns; in physics, for distance from velocity; in business, for year-to-date sales.

```python
# Daily sales amounts
arr = np.array([100, 150, 200, 175, 225, 250, 200, 180])

# Running total (cumulative sum)
cumulative_sales = arr.cumsum()
print(f"Daily sales: {arr}")
print(f"Cumulative: {cumulative_sales}")
# [100, 250, 450, 625, 850, 1100, 1300, 1480]
```

```python
# Growth rates (as multipliers)
growth = np.array([1.02, 1.03, 1.01, 1.04, 1.02])

# Cumulative growth
cumulative_growth = growth.cumprod()
print(f"Period growth: {growth}")
print(f"Cumulative: {cumulative_growth}")
# Shows compounded growth over time
```

## 4.4 Boolean Array Methods

Boolean arrays have special methods that answer logical questions about the data: `any()` returns True if at least one element is True (asking ”is this true anywhere?”), while `all()` returns True only if every element is True (asking ”is this true everywhere?”). You can also use `sum()` on boolean arrays to count how many True values exist, since Python treats True as 1 and False as 0. These methods are perfect for data validation (checking if any values are invalid), quality control (ensuring all measurements meet standards), and analysis (counting how many items meet criteria). They turn boolean masks from filtering tools into quantitative measures.

```python
# Test scores
scores = np.array([78, 92, 85, 67, 95, 73, 88])

```

```python
# Check for any failing grades (< 60)
has_failures = (scores < 60).any()
print(f"Any failures? {has_failures}")

# Check if all students passed (>= 60)
all_passing = (scores >= 60).all()
print(f"All passing? {all_passing}")

# Count passing grades
num_passing = (scores >= 60).sum()
print(f"Number passing: {num_passing}")
```

```python
# Daily temperatures
temps = np.array([72, 68, 75, 82, 79, 85, 88, 84])

any_hot = (temps > 85).any()
all_warm = (temps > 60).all()
count_hot = (temps > 80).sum()

print(f"Any day above 85? {any_hot}")  # True
print(f"All days above 60? {all_warm}")  # True
print(f"Days above 80: {count_hot}")  # 4
```

## 4.5 Percentiles for Distribution Analysis

Percentiles tell you the value below which a certain percentage of data falls - the 25th percentile means 25 percent of data is below this value, 50th percentile is the median (middle value), and 75th percentile means 75 percent of data is below this value. Think of class rankings: being in the 90th percentile means you scored better than 90 percent of students. NumPy's `np.percentile()` function calculates these values, which are crucial for understanding data distribution, identifying outliers, and setting thresholds. The quartiles (25th, 50th, 75th percentiles) are especially important - they divide your data into four equal groups and help you understand data spread beyond just average and standard deviation.

```python
# Test scores
scores = np.array([67, 72, 73, 78, 82, 85, 88, 91, 92, 95])

# Calculate quartiles
percentile_25 = np.percentile(scores, 25)
percentile_50 = np.percentile(scores, 50)  # Median
percentile_75 = np.percentile(scores, 75)

print(f"25th percentile: {percentile_25}")
print(f"50th percentile (median): {percentile_50}")
print(f"75th percentile: {percentile_75}")
```

```python
# Housing prices in neighborhood (thousands)
prices = np.array([245, 280, 310, 325, 340, 365, 385, 420, 450, 520, 850])

q25 = np.percentile(prices, 25)
q50 = np.percentile(prices, 50)
q75 = np.percentile(prices, 75)

print(f"Lower quartile (25 percent): ${q25}k")
print(f"Median (50 percent): ${q50}k")
print(f"Upper quartile (75 percent): ${q75}k")

# Interpretation: 75 percent of homes cost less than the upper quartile
```

## 5 Part 5: Sorting and Unique Values

### 5.1 Sorting Arrays In-Place vs Creating Copies

NumPy provides two ways to sort arrays: the `.sort()` method sorts the array in-place (modifying the original), while the `np.sort()` function returns a sorted copy (leaving the original unchanged). Think of it like organizing your closet: `.sort()` rearranges your actual clothes permanently, while `np.sort()` is like making a list of how clothes could be arranged without touching them. Choose in-place sorting when you want to permanently reorder data and save memory, but use `np.sort()` when you need both the original order and sorted order. For multidimensional arrays, you can specify which axis to sort along - sorting rows or sorting columns independently.

```python
# In-place sort
arr = np.array([3, 1, 4, 1, 5, 9, 2, 6])
arr.sort()  # Modifies arr directly
print(arr)  # [1 1 2 3 4 5 6 9]

# Sorted copy
arr2 = np.array([3, 1, 4, 1, 5, 9, 2, 6])
sorted_copy = np.sort(arr2)
print(f"Original: {arr2}")  # Unchanged
print(f"Sorted: {sorted_copy}")
```

```python
# Sorting 2D arrays along different axes
arr = np.array([[5, 2, 8],
                [3, 9, 1],
                [7, 4, 6]])

# Sort each column (axis=0)
arr_sorted = arr.copy()
arr_sorted.sort(axis=0)
print("Sorted by columns:")
print(arr_sorted)

# Sort each row (axis=1)
arr_sorted2 = arr.copy()
arr_sorted2.sort(axis=1)
print("Sorted by rows:")
print(arr_sorted2)
```

### 5.2 Indirect Sorting with argsort()

The `argsort()` function doesn't sort the array itself - instead, it returns the indices that would sort the array. This is incredibly useful when you need to sort multiple related arrays together based on one array's values, or when you want to rank items. Think of it like a competition where `argsort()` tells you ”1st place goes to competitor number 5, 2nd place to competitor number 2, 3rd to competitor number 7” - it gives you the ordering of indices. You can use these indices with fancy indexing to reorder any related arrays, keeping data synchronized. This is essential for sorting student names by their scores, or organizing any data where multiple arrays need to stay aligned.

```python
# Scores array
scores = np.array([78, 92, 85, 67, 95])

# Get sorting indices
sort_indices = np.argsort(scores)
print(f"Indices: {sort_indices}")  # [3, 0, 2, 1, 4]

```

```python
# Meaning: smallest value is at index 3 (67),
# next at index 0 (78), then index 2 (85), etc.
```

```python
# Student data
names = np.array(["Alice", "Bob", "Charlie", "David", "Eve"])
scores = np.array([85, 92, 78, 95, 88])

# Get ranking (highest to lowest)
ranking = np.argsort(scores)[::-1]  # Reverse for descending

# Print rankings
print("Rankings:")
for i, idx in enumerate(ranking, 1):
    print(f"{i}. {names[idx]}: {scores[idx]}")
```

### 5.3 Finding Unique Values and Membership Testing

NumPy provides efficient functions for working with unique values: `np.unique()` returns sorted unique elements (removing duplicates), while `np.in1d()` tests if elements from one array appear in another array. Think of `np.unique()` like getting a list of distinct items from a shopping cart (removing duplicates), and `np.in1d()` like checking if items on your wishlist are available in a store. These operations are fundamental for data cleaning (finding distinct categories), analysis (counting unique values), and filtering (selecting elements that match a list). They're much faster than Python's set operations for large arrays.

```python
# Array with duplicates
names = np.array(["Bob", "Will", "Joe", "Bob", "Will", "Joe", "Joe"])
unique_names = np.unique(names)
print(unique_names)  # ['Bob', 'Joe', 'Will'] - sorted, no duplicates

# Numeric example
ints = np.array([3, 3, 3, 2, 2, 1, 1, 4, 4])
print(np.unique(ints))  # [1, 2, 3, 4]
```

```python
# Membership testing
values = np.array([6, 0, 0, 3, 2, 5, 6])
target = [2, 3, 6]

# Check membership
result = np.in1d(values, target)
print(result)  # [True, False, False, True, True, False, True]
```

## 6 Part 6: Comprehensive Practice Example

### 6.1 Putting It All Together: Complete Student Grade Analysis

This comprehensive example combines all the techniques from today's lecture to solve a realistic data analysis problem. We'll use boolean indexing to filter students, statistical methods with axis operations to calculate metrics, `argsort()` for rankings, and `np.percentile()` for grade distribution analysis. This is exactly the kind of workflow you'll use in Assignment 4 and in real data analysis tasks - filtering data, calculating statistics, and generating insights. Pay attention to how the techniques work together to answer multiple questions about the same dataset.

```python
# Student grade analysis
student_names = np.array(["Alice", "Bob", "Charlie", "David", "Eve", "Frank"])
test_scores = np.array([
    [85, 92, 78, 88],  # Alice's 4 test scores
    [72, 68, 75, 70],   # Bob
    [95, 98, 92, 97],   # Charlie
    [88, 85, 90, 87],   # David
    [65, 70, 68, 72],   # Eve
    [90, 88, 92, 89]    # Frank
])

print("=== Student Grade Analysis ===\n")

# 1. Calculate each student's average
student_averages = test_scores.mean(axis=1)
print("Student averages:")
for name, avg in zip(student_names, student_averages):
    print(f"  {name}: {avg:.1f}")
```

```python
# 2. Find top performers (average > 85)
top_performers_mask = student_averages > 85
top_performers = student_names[top_performers_mask]
print(f"\nTop performers (>85 avg): {top_performers}")

# 3. Find students needing help (average < 75)
needs_help_mask = student_averages < 75
needs_help = student_names[needs_help_mask]
print(f"Needs help (<75 avg): {needs_help}")
```

```python
# 4. Rank all students by average
ranking_indices = np.argsort(student_averages)[::-1]
print("\nStudent rankings:")
for i, idx in enumerate(ranking_indices, 1):
    print(f"  {i}. {student_names[idx]}: {student_averages[idx]:.1f}")

# 5. Calculate grade distribution percentiles
q25 = np.percentile(student_averages, 25)
q50 = np.percentile(student_averages, 50)
q75 = np.percentile(student_averages, 75)

print(f"\nGrade distribution:")
print(f"  25th percentile: {q25:.1f}")
print(f"  Median (50th): {q50:.1f}")
print(f"  75th percentile: {q75:.1f}")

# 6. Test average across all students
test_averages = test_scores.mean(axis=0)
print(f"\nTest averages: {test_averages}")
print(f"  Test 1: {test_averages[0]:.1f}")
print(f"  Test 2: {test_averages[1]:.1f}")
print(f"  Test 3: {test_averages[2]:.1f}")
print(f"  Test 4: {test_averages[3]:.1f}")
```

**Synthesis and Key Takeaways:**

This comprehensive example demonstrates the power of combining NumPy techniques for real data analysis. You:

- Filtered students by performance using boolean indexing
- Calculated statistics across different dimensions with the axis parameter
- Ranked students using `argsort()`
- Analyzed grade distribution with percentiles

All of these are fundamental data analysis tasks that you'll use constantly in Assignment 4 and beyond. The key insight is that NumPy's tools work together seamlessly - you can filter data, calculate statistics, sort results, and analyze distributions all in a few lines of efficient code.

## Key Takeaways

### Master These Critical Skills:

1. **Boolean Indexing** is your primary tool for data filtering - remember to use & and | with parentheses!
2. **Fancy Indexing** creates copies, not views - understand when you need each
3. **The axis Parameter** is THE most important concept - axis=0 goes down, axis=1 goes across
4. **Statistical Methods** with axis operations let you analyze data by groups or categories
5. **Sorting with `argsort()`** keeps related data synchronized when reordering
6. **Combining Techniques** creates powerful data analysis workflows

## Connection to Next Topics

In Lecture 15, we'll add even more powerful tools to your NumPy toolkit: random number generation for simulations, linear algebra operations for advanced calculations, conditional logic with `np.where()`, and sophisticated array manipulations. These NumPy skills you've learned today are the foundation for pandas, which we'll use to analyze real datasets with labeled data structures starting soon!

## Assignment 4 Preparation

The techniques in this lecture directly prepare you for Assignment 4:

- **Problem 2**: Basic statistics using aggregation methods
- **Problem 3**: Win/loss counting using boolean array methods
- **Problem 5**: Boolean filtering and percentiles for data analysis

Practice these skills - understanding boolean indexing, the axis parameter, and combining multiple operations will be essential for assignment success!