Tutorial by Dr. Ayes Chinmay
Date: 23/10/2025

Chapter 13: Advanced Uses of NumPy

Complete Tutorial Guide

This comprehensive tutorial covers all sections and subsections of Chapter 13, focusing on advanced NumPy operations, data visualization with Matplotlib, and practical applications.

13.1 Advanced Math Operations with NumPy

NumPy provides an extensive collection of mathematical functions that operate efficiently on arrays. These functions enable complex mathematical computations with simple syntax.

13.1.1 Trigonometric Functions

Trigonometric functions are essential for scientific computing, signal processing, and geometric calculations.

Code Example:

```
import numpy as np

# Create an array of angles in radians
angles = np.array([0, np.pi/6, np.pi/4, np.pi/3, np.pi/2])

# Calculate trigonometric functions
sin_values = np.sin(angles)
cos_values = np.cos(angles)
tan_values = np.tan(angles)

print("Angles (radians):", angles)
print("Sine values:", sin_values)
print("Cosine values:", cos_values)
print("Tangent values:", tan_values)
```

Output:

```
Angles (radians): [0.         0.52359878 0.78539816 1.04719755 1.57079633]
Sine values: [0.         0.5        0.70710678 0.8660254  1.        ]
Cosine values: [1.00000000e+00 8.66025404e-01 7.07106781e-01 5.00000000e-01
 6.12323400e-17]
Tangent values: [0.00000000e+00 5.77350269e-01 1.00000000e+00 1.73205081e+00
 1.63312394e+16]
```

Key Points:

- NumPy trigonometric functions work with radians by default
- Functions are vectorized, operating on entire arrays efficiently
- Common functions include: `np.sin()`, `np.cos()`, `np.tan()`, `np.arcsin()`, `np.arccos()`, `np.arctan()`

13.1.2 Exponential and Logarithmic Functions

Exponential and logarithmic functions are fundamental in mathematics, finance, and data science.

Code Example:

```
import numpy as np

# Exponential and logarithmic functions
x = np.array([1, 2, 3, 4, 5])

# Exponential
```

```
exp_values = np.exp(x)

# Natural logarithm
log_values = np.log(x)

# Base-10 logarithm
log10_values = np.log10(x)

# Square root
sqrt_values = np.sqrt(x)

print("Original array:", x)
print("Exponential (e^x):", exp_values)
print("Natural log (ln(x)):", log_values)
print("Base-10 log:", log10_values)
print("Square root:", sqrt_values)
```

Output:

```
Original array: [1 2 3 4 5]
Exponential (e^x): [  2.71828183   7.3890561   20.08553692  54.59815003 148.4131591 ]
Natural log (ln(x)): [0.         0.69314718 1.09861229 1.38629436 1.60943791]
Base-10 log: [0.         0.30103    0.47712125 0.60205999 0.69897   ]
Square root: [1.         1.41421356 1.73205081 2.         2.23606798]
```

13.1.3 Statistical Functions

NumPy provides comprehensive statistical functions for data analysis.

Code Example:

```
import numpy as np

# Create a sample array
data = np.array([12, 15, 18, 20, 22, 25, 28, 30, 35, 40])

# Statistical functions
mean_val = np.mean(data)
median_val = np.median(data)
std_val = np.std(data)
var_val = np.var(data)
min_val = np.min(data)
max_val = np.max(data)

print("Data:", data)
print("Mean:", mean_val)
print("Median:", median_val)
print("Standard Deviation:", std_val)
print("Variance:", var_val)
print("Minimum:", min_val)
print("Maximum:", max_val)
```

Output:

```
Data: [12 15 18 20 22 25 28 30 35 40]
Mean: 24.5
Median: 23.5
Standard Deviation: 8.417244204607586
Variance: 70.85
Minimum: 12
Maximum: 40
```

13.2 Downloading Matplotlib

Matplotlib is a comprehensive library for creating static, animated, and interactive visualizations in Python.

Installation Methods

Using pip:

```
pip install matplotlib
```

Using conda:

```
conda install matplotlib
```

Verification:

```
import matplotlib
print("Matplotlib version:", matplotlib.__version__)
```

Basic Import Convention

```
import matplotlib.pyplot as plt
import numpy as np
```

The standard convention is to import `matplotlib.pyplot` as `plt` for easier usage throughout your code.

13.3 Plotting Lines with NumPy and Matplotlib

Line plots are fundamental for visualizing trends, relationships, and changes over time.

13.3.1 Basic Line Plot

Code Example:

```
import numpy as np
import matplotlib.pyplot as plt

# Create data
x = np.linspace(0, 10, 100)
y = np.sin(x)

# Create plot
plt.plot(x, y)
plt.xlabel('X-axis')
plt.ylabel('Y-axis')
plt.title('Sine Wave')
plt.grid(True)
plt.show()
```

Description: This creates a smooth sine wave with 100 points between 0 and 10 on the x-axis.

13.3.2 Customizing Line Plots

Code Example:

```
import numpy as np
import matplotlib.pyplot as plt

x = np.linspace(0, 2*np.pi, 100)
y = np.sin(x)
```

```
# Create plot with customization
plt.plot(x, y, color='blue', linewidth=2, linestyle='-',
         marker='o', markersize=3, label='sin(x)')
plt.xlabel('Angle (radians)', fontsize=12)
plt.ylabel('Amplitude', fontsize=12)
plt.title('Customized Sine Wave', fontsize=14, fontweight='bold')
plt.legend()
plt.grid(True, alpha=0.3)
plt.show()
```

Customization Options:

- **color**: Line color (e.g., 'red', 'blue', '#FF5733')
- **linewidth**: Thickness of the line
- **linestyle**: Style ('-', '--', '-.', ':')
- **marker**: Point markers ('o', 's', '^', etc.)
- **label**: Legend label

13.4 Plotting More Than One Line

Comparing multiple datasets on the same plot helps identify relationships and patterns.

Code Example:

```
import numpy as np
import matplotlib.pyplot as plt

# Create data
x = np.linspace(0, 2*np.pi, 100)
y1 = np.sin(x)
y2 = np.cos(x)
y3 = np.sin(x) + np.cos(x)

# Plot multiple lines
plt.plot(x, y1, label='sin(x)', color='blue', linewidth=2)
plt.plot(x, y2, label='cos(x)', color='red', linewidth=2)
plt.plot(x, y3, label='sin(x) + cos(x)', color='green',
         linewidth=2, linestyle='--')

plt.xlabel('Angle (radians)')
plt.ylabel('Value')
plt.title('Multiple Trigonometric Functions')
plt.legend()
plt.grid(True, alpha=0.3)
plt.show()
```

Key Features:

- Each `plt.plot()` call adds a new line
- The `label` parameter enables legends
- Different colors and styles distinguish lines
- `plt.legend()` displays the legend

13.5 Plotting Compound Interest

Visualizing financial growth helps understand the power of compound interest over time.

Code Example:

```
import numpy as np
import matplotlib.pyplot as plt
```

```
# Parameters
principal = 1000  # Initial investment
rates = [0.03, 0.05, 0.07, 0.10]  # Different interest rates
years = np.arange(0, 31)  # 0 to 30 years

# Plot for different rates
plt.figure(figsize=(10, 6))
for rate in rates:
    amount = principal * (1 + rate) ** years
    plt.plot(years, amount, label=f'{rate*100:.0f}% interest',
             linewidth=2)

plt.xlabel('Years', fontsize=12)
plt.ylabel('Amount ($)', fontsize=12)
plt.title('Compound Interest Growth Over Time',
          fontsize=14, fontweight='bold')
plt.legend()
plt.grid(True, alpha=0.3)
plt.show()

# Calculate specific example
principal = 1000
rate = 0.05
years_calc = np.arange(0, 11)
amount = principal * (1 + rate) ** years_calc

print("Year\tAmount")
print("-" * 20)
for year, amt in zip(years_calc, amount):
    print(f"{year}\t${amt:.2f}")
```

Output:

```
Year    Amount
--------------------
0       $1000.00
1       $1050.00
2       $1102.50
3       $1157.63
4       $1215.51
5       $1276.28
6       $1340.10
7       $1407.10
8       $1477.46
9       $1551.33
10      $1628.89
```

Formula: $A = P(1 + r)^t$

- A = Final amount
- P = Principal (initial investment)
- r = Interest rate (as decimal)
- t = Time in years

13.6 Creating Histograms with Matplotlib

Histograms display the distribution of data by grouping values into bins.

Code Example:

```
import numpy as np
import matplotlib.pyplot as plt

# Generate random data (normal distribution)
np.random.seed(42)
data = np.random.randn(1000) * 10 + 50
```

```
# Create histogram
plt.figure(figsize=(10, 6))
plt.hist(data, bins=30, color='skyblue', edgecolor='black',
         alpha=0.7)
plt.xlabel('Value', fontsize=12)
plt.ylabel('Frequency', fontsize=12)
plt.title('Histogram of Random Data', fontsize=14, fontweight='bold')
plt.grid(True, alpha=0.3, axis='y')
plt.show()

# Statistical summary
print("Statistical Summary:")
print(f"Mean: {np.mean(data):.2f}")
print(f"Median: {np.median(data):.2f}")
print(f"Standard Deviation: {np.std(data):.2f}")
print(f"Min: {np.min(data):.2f}")
print(f"Max: {np.max(data):.2f}")
```

Parameters:

- **bins**: Number of histogram bins
- **color**: Bar color
- **edgecolor**: Border color of bars
- **alpha**: Transparency (0-1)

13.7 Circles and the Aspect Ratio

Creating circles requires proper aspect ratio settings to ensure they appear circular rather than elliptical.

Code Example:

```
import numpy as np
import matplotlib.pyplot as plt

# Create circle using parametric equations
theta = np.linspace(0, 2*np.pi, 100)
radius = 5
x = radius * np.cos(theta)
y = radius * np.sin(theta)

# Plot without aspect ratio correction
plt.figure(figsize=(12, 5))

plt.subplot(1, 2, 1)
plt.plot(x, y, linewidth=2)
plt.title('Without Aspect Ratio (Ellipse)')
plt.xlabel('X')
plt.ylabel('Y')
plt.grid(True)

# Plot with aspect ratio correction
plt.subplot(1, 2, 2)
plt.plot(x, y, linewidth=2)
plt.axis('equal')  # Set equal aspect ratio
plt.title('With Aspect Ratio (Circle)')
plt.xlabel('X')
plt.ylabel('Y')
plt.grid(True)

plt.tight_layout()
plt.show()
```

Key Command: `plt.axis('equal')` ensures equal scaling on both axes, making circles appear circular.

13.8 Creating Pie Charts

Pie charts effectively display proportional data as slices of a circle.

Code Example:

```
import numpy as np
import matplotlib.pyplot as plt

# Data
categories = ['Category A', 'Category B', 'Category C',
              'Category D', 'Category E']
values = np.array([30, 25, 20, 15, 10])

# Create pie chart
plt.figure(figsize=(10, 8))
colors = ['#ff9999', '#66b3ff', '#99ff99', '#ffcc99', '#ff99cc']
explode = (0.1, 0, 0, 0, 0)  # Explode 1st slice

plt.pie(values, labels=categories, colors=colors, autopct='%1.1f%%',
        startangle=90, explode=explode, shadow=True)
plt.title('Distribution of Categories', fontsize=14, fontweight='bold')
plt.axis('equal')  # Equal aspect ratio ensures circular pie
plt.show()

# Print percentages
print("Category Distribution:")
for cat, val in zip(categories, values):
    percentage = (val / values.sum()) * 100
    print(f"{cat}: {percentage:.1f}%")
```

Parameters:

- **labels**: Category names
- **autopct**: Format for percentage display
- **startangle**: Rotation angle
- **explode**: Distance to separate slices
- **shadow**: Add shadow effect

13.9 Doing Linear Algebra with NumPy

NumPy provides comprehensive linear algebra operations essential for scientific computing, machine learning, and data analysis.

13.9.1 The Dot Product

The dot product is a fundamental operation in linear algebra, used for vector and matrix multiplication.

Code Example:

```
import numpy as np

# Dot product of two vectors
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])

dot_product = np.dot(a, b)
print("Vector a:", a)
print("Vector b:", b)
print("Dot product (a · b):", dot_product)
print("Calculation: (1*4) + (2*5) + (3*6) =", dot_product)

# Matrix multiplication using dot product
matrix1 = np.array([[1, 2], [3, 4]])
matrix2 = np.array([[5, 6], [7, 8]])
```

```
result = np.dot(matrix1, matrix2)
print("\nMatrix 1:")
print(matrix1)
print("Matrix 2:")
print(matrix2)
print("Matrix multiplication result:")
print(result)
```

Output:

```
Vector a: [1 2 3]
Vector b: [4 5 6]
Dot product (a · b): 32
Calculation: (1*4) + (2*5) + (3*6) = 32

Matrix 1:
[[1 2]
 [3 4]]
Matrix 2:
[[5 6]
 [7 8]]
Matrix multiplication result:
[[19 22]
 [43 50]]
```

Mathematical Formula:
For vectors: $a \cdot b = \sum_{i=1}^{n} a_i b_i$

For matrices: $(AB)_{ij} = \sum_{k} A_{ik} B_{kj}$

13.9.2 The Outer Product Function

The outer product creates a matrix from two vectors.

Code Example:

```
import numpy as np

# Outer product of two vectors
a = np.array([1, 2, 3])
b = np.array([4, 5])

outer_product = np.outer(a, b)
print("Vector a:", a)
print("Vector b:", b)
print("Outer product:")
print(outer_product)
```

Output:

```
Vector a: [1 2 3]
Vector b: [4 5]
Outer product:
[[ 4  5]
 [ 8 10]
 [12 15]]
```

Explanation: Each element (i, j) of the outer product equals $a_i \times b_j$

13.9.3 Other Linear Algebra Functions

NumPy's `linalg` module provides advanced linear algebra operations.

Code Example:

```
import numpy as np

# Create a matrix
matrix = np.array([[1, 2], [3, 4]])
print("Original Matrix:")
print(matrix)

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

# Inverse
inv = np.linalg.inv(matrix)
print("\nInverse Matrix:")
print(inv)

# Eigenvalues and eigenvectors
eigenvalues, eigenvectors = np.linalg.eig(matrix)
print("\nEigenvalues:", eigenvalues)
print("Eigenvectors:")
print(eigenvectors)

# Matrix transpose
transpose = matrix.T
print("\nTranspose:")
print(transpose)

# Verify: Matrix × Inverse = Identity
identity = np.dot(matrix, inv)
print("\nMatrix × Inverse (should be identity):")
print(identity)
```

Output:

```
Original Matrix:
[[1 2]
 [3 4]]

Determinant: -2.0

Inverse Matrix:
[[-2.   1. ]
 [ 1.5 -0.5]]

Eigenvalues: [-0.37228132  5.37228132]
Eigenvectors:
[[-0.82456484 -0.41597356]
 [ 0.56576746 -0.90937671]]

Transpose:
[[1 3]
 [2 4]]
```

Key Functions:

- `np.linalg.det()`: Calculate determinant
- `np.linalg.inv()`: Calculate matrix inverse
- `np.linalg.eig()`: Find eigenvalues and eigenvectors
- `matrix.T`: Transpose matrix
- `np.linalg.solve()`: Solve linear equations

13.10 Three-Dimensional Plotting

Three-dimensional plots visualize data in three dimensions, useful for surface plots, scatter plots, and more.

Code Example:

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# Create data for 3D surface
x = np.linspace(-5, 5, 50)
y = np.linspace(-5, 5, 50)
X, Y = np.meshgrid(x, y)
Z = np.sin(np.sqrt(X**2 + Y**2))

# Create 3D plot
fig = plt.figure(figsize=(12, 5))

# Surface plot
ax1 = fig.add_subplot(121, projection='3d')
ax1.plot_surface(X, Y, Z, cmap='viridis')
ax1.set_xlabel('X axis')
ax1.set_ylabel('Y axis')
ax1.set_zlabel('Z axis')
ax1.set_title('3D Surface Plot')

# Wireframe plot
ax2 = fig.add_subplot(122, projection='3d')
ax2.plot_wireframe(X, Y, Z, color='blue')
ax2.set_xlabel('X axis')
ax2.set_ylabel('Y axis')
ax2.set_zlabel('Z axis')
ax2.set_title('3D Wireframe Plot')

plt.tight_layout()
plt.show()
```

3D Plot Types:

- **Surface plots**: `plot_surface()`
- **Wireframe plots**: `plot_wireframe()`
- **Scatter plots**: `scatter()`
- **Contour plots**: `contour()`

13.11 NumPy Financial Applications

NumPy provides powerful tools for financial calculations and modeling.

Present and Future Value Calculations

Code Example:

```
import numpy as np

# Compound Interest Calculation
principal = 1000  # Initial investment
rate = 0.05  # 5% annual interest rate
years = np.arange(0, 11)  # 0 to 10 years

# Calculate compound interest: A = P(1 + r)^t
amount = principal * (1 + rate) ** years

print("Year\tAmount")
print("-" * 20)
```

```
for year, amt in zip(years, amount):
    print(f"{year}\t${amt:.2f}")

print(f"\nTotal growth: ${amount[-1] - principal:.2f}")
print(f"Percentage increase: {((amount[-1] - principal) / principal * 100):.2f}%")
```

Output:

```
Year    Amount
--------------------
0       $1000.00
1       $1050.00
2       $1102.50
3       $1157.63
4       $1215.51
5       $1276.28
6       $1340.10
7       $1407.10
8       $1477.46
9       $1551.33
10      $1628.89

Total growth: $628.89
Percentage increase: 62.89%
```

Investment Portfolio Analysis

Code Example:

```
import numpy as np

# Portfolio returns (monthly)
returns = np.array([0.02, -0.01, 0.03, 0.015, -0.005,
                    0.025, 0.01, -0.02, 0.04, 0.018])

# Calculate metrics
mean_return = np.mean(returns)
std_return = np.std(returns)
cumulative_return = np.prod(1 + returns) - 1

print("Investment Portfolio Analysis")
print("-" * 40)
print(f"Average Monthly Return: {mean_return*100:.2f}%")
print(f"Return Volatility (Std Dev): {std_return*100:.2f}%")
print(f"Cumulative Return: {cumulative_return*100:.2f}%")
print(f"Annualized Return: {mean_return*12*100:.2f}%")

# Sharpe Ratio (assuming risk-free rate of 2% annually)
risk_free_rate = 0.02 / 12  # Monthly
sharpe_ratio = (mean_return - risk_free_rate) / std_return
print(f"Sharpe Ratio: {sharpe_ratio:.3f}")
```

Financial Formulas:

- Compound Interest: $A = P(1 + r)^t$
- Average Return: $\bar{r} = \frac{1}{n} \sum_{i=1}^{n} r_i$
- Sharpe Ratio: $S = \frac{r_p - r_f}{\sigma_p}$

13.12 Adjusting Axes with xticks and yticks

Customizing axis ticks improves plot readability and presentation.

Code Example:

```
import numpy as np
import matplotlib.pyplot as plt

# Create data
x = np.linspace(0, 2*np.pi, 100)
y = np.sin(x)

# Create plot
plt.figure(figsize=(10, 6))
plt.plot(x, y, linewidth=2, color='blue')

# Custom x-axis ticks
x_ticks = np.array([0, np.pi/2, np.pi, 3*np.pi/2, 2*np.pi])
x_labels = ['0', 'π/2', 'π', '3π/2', '2π']
plt.xticks(x_ticks, x_labels, fontsize=12)

# Custom y-axis ticks
y_ticks = np.array([-1, -0.5, 0, 0.5, 1])
plt.yticks(y_ticks, fontsize=12)

plt.xlabel('Angle', fontsize=12)
plt.ylabel('sin(x)', fontsize=12)
plt.title('Sine Wave with Custom Ticks', fontsize=14, fontweight='bold')
plt.grid(True, alpha=0.3)
plt.show()
```

Customization Options:

```
# Set tick positions
plt.xticks([0, 1, 2, 3, 4, 5])

# Set tick positions and labels
plt.xticks([0, 1, 2, 3], ['A', 'B', 'C', 'D'])

# Rotate labels
plt.xticks(rotation=45)

# Format ticks
plt.xticks(fontsize=10, color='red')
```

13.13 NumPy Mixed-Data Records

Structured arrays allow storing different data types in a single NumPy array, similar to database records.

Code Example:

```
import numpy as np

# Create a structured array with mixed data types
dtype = [('name', 'U20'), ('age', 'i4'), ('salary', 'f8')]

employees = np.array([
    ('John Doe', 30, 50000.0),
    ('Jane Smith', 25, 55000.0),
    ('Bob Johnson', 35, 60000.0),
    ('Alice Brown', 28, 52000.0),
    ('Charlie Wilson', 32, 58000.0)
], dtype=dtype)

print("Employee Records:")
print(employees)
```

```
print("\nNames:", employees['name'])
print("Ages:", employees['age'])
print("Salaries:", employees['salary'])

# Statistical analysis
print("\nStatistical Summary:")
print(f"Average Age: {np.mean(employees['age']):.1f}")
print(f"Average Salary: ${np.mean(employees['salary']):.2f}")
print(f"Min Salary: ${np.min(employees['salary']):.2f}")
print(f"Max Salary: ${np.max(employees['salary']):.2f}")

# Filtering data
high_earners = employees[employees['salary'] &gt; 55000]
print("\nHigh Earners (&gt;$55,000):")
for emp in high_earners:
    print(f"  {emp['name']}: ${emp['salary']:.2f}")
```

Output:

```
Employee Records:
[('John Doe', 30, 50000.) ('Jane Smith', 25, 55000.)
 ('Bob Johnson', 35, 60000.) ('Alice Brown', 28, 52000.)
 ('Charlie Wilson', 32, 58000.)]

Names: ['John Doe' 'Jane Smith' 'Bob Johnson' 'Alice Brown' 'Charlie Wilson']
Ages: [30 25 35 28 32]
Salaries: [50000. 55000. 60000. 52000. 58000.]

Statistical Summary:
Average Age: 30.0
Average Salary: $55000.00
Min Salary: $50000.00
Max Salary: $60000.00

High Earners (&gt;$55,000):
  Bob Johnson: $60000.00
  Charlie Wilson: $58000.00
```

Data Type Codes:

- `'U20'`: Unicode string (max 20 characters)
- `'i4'`: 32-bit integer
- `'f8'`: 64-bit float
- `'S10'`: Fixed-length byte string
- `'?'`: Boolean

13.14 Reading and Writing NumPy Data from Files

NumPy provides efficient methods for saving and loading array data.

Binary Format (.npy and .npz)

Code Example:

```
import numpy as np

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

# Save to binary file (.npy)
np.save('data.npy', data)
print("Data saved to 'data.npy'")

# Load from binary file
loaded_data = np.load('data.npy')
```

```
print("\nLoaded data from 'data.npy':")
print(loaded_data)

# Save multiple arrays (.npz)
array1 = np.array([1, 2, 3])
array2 = np.array([4, 5, 6])
array3 = np.array([[1, 2], [3, 4]])

np.savez('multiple_arrays.npz', first=array1, second=array2,
         third=array3)
print("\nMultiple arrays saved to 'multiple_arrays.npz'")

# Load multiple arrays
loaded_npz = np.load('multiple_arrays.npz')
print("\nFirst array:", loaded_npz['first'])
print("Second array:", loaded_npz['second'])
print("Third array:")
print(loaded_npz['third'])
```

Output:

```
Data saved to 'data.npy'

Loaded data from 'data.npy':
[[1 2 3]
 [4 5 6]
 [7 8 9]]

Multiple arrays saved to 'multiple_arrays.npz'

First array: [1 2 3]
Second array: [4 5 6]
Third array:
[[1 2]
 [3 4]]
```

Text Format (.txt, .csv)

Code Example:

```
import numpy as np

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

# Save to text file (space-separated)
np.savetxt('data.txt', data, fmt='%d')
print("Data saved to 'data.txt'")

# Save to CSV file
np.savetxt('data.csv', data, fmt='%d', delimiter=',',
           header='Column1,Column2,Column3', comments='')
print("Data saved to 'data.csv'")

# Load from text file
loaded_txt = np.loadtxt('data.txt')
print("\nLoaded data from 'data.txt':")
print(loaded_txt)

# Load from CSV file
loaded_csv = np.loadtxt('data.csv', delimiter=',', skiprows=1)
print("\nLoaded data from 'data.csv':")
print(loaded_csv)
```

Advanced File Operations

Code Example:

```
import numpy as np

# Create structured array
dtype = [('name', 'U20'), ('score', 'f8')]
students = np.array([
    ('Alice', 95.5),
    ('Bob', 87.3),
    ('Charlie', 92.1)
], dtype=dtype)

# Save structured array
np.save('students.npy', students)

# Load and display
loaded_students = np.load('students.npy')
print("Student Records:")
for student in loaded_students:
    print(f"  {student['name']}: {student['score']:.1f}")

# Compressed save (.npz with compression)
large_array = np.random.rand(1000, 1000)
np.savez_compressed('large_data.npz', data=large_array)
print("\nCompressed data saved")
```

File Format Comparison:

Format	Extension	Advantages	Use Case
Binary	.npy	Fast, preserves type	Single array
Archive	.npz	Multiple arrays	Related datasets
Text	.txt	Human-readable	Small datasets
CSV	.csv	Spreadsheet compatible	Tabular data

Chapter Summary

Chapter 13 covered advanced NumPy operations and visualization techniques:

Mathematical Operations:

- Trigonometric functions (sin, cos, tan)
- Exponential and logarithmic functions
- Statistical functions (mean, median, std)

Visualization with Matplotlib:

- Line plots for trends and relationships
- Multiple line plots for comparisons
- Histograms for distribution analysis
- Pie charts for proportional data
- 3D plotting for multidimensional data

Linear Algebra:

- Dot product for vector and matrix multiplication
- Outer product for creating matrices
- Matrix operations (determinant, inverse, eigenvalues)

Practical Applications:

- Financial calculations (compound interest)
- Portfolio analysis
- Data visualization techniques

Data Management:

- Structured arrays for mixed data types
- File I/O operations (binary and text formats)
- Efficient data storage and retrieval

Key Takeaways:

1. NumPy provides comprehensive mathematical functions for array operations
2. Matplotlib enables professional data visualization
3. Linear algebra operations are essential for scientific computing
4. Proper axis configuration improves plot readability
5. Structured arrays handle complex data efficiently
6. Multiple file formats serve different purposes

Practice Exercises

Exercise 1: Mathematical Operations

Create an array of 50 evenly spaced values between 0 and 2π, calculate sin(x), cos(x), and plot both on the same graph.

Exercise 2: Financial Analysis

Calculate the future value of monthly investments of $200 for 20 years at 6% annual interest, compounded monthly.

Exercise 3: Data Visualization

Create a histogram showing the distribution of 10,000 random values from a normal distribution with mean=100 and std=15.

Exercise 4: Linear Algebra

Create two 3×3 matrices, calculate their dot product, determinant, and inverse.

Exercise 5: Data Management

Create a structured array storing information about 10 products (name, price, quantity), calculate total inventory value.

Additional Resources

Official Documentation:

- NumPy: https://numpy.org/doc/
- Matplotlib: https://matplotlib.org/

Key Functions Reference:

NumPy Math:

- `np.sin()`, `np.cos()`, `np.tan()`
- `np.exp()`, `np.log()`, `np.sqrt()`
- `np.mean()`, `np.median()`, `np.std()`

Matplotlib Plotting:

- `plt.plot()`: Line plots
- `plt.hist()`: Histograms
- `plt.pie()`: Pie charts
- `plt.scatter()`: Scatter plots

Linear Algebra:

- `np.dot()`: Dot product
- `np.outer()`: Outer product
- `np.linalg.det()`: Determinant
- `np.linalg.inv()`: Matrix inverse

File Operations:

- `np.save()`, `np.load()`: Binary format
- `np.savetxt()`, `np.loadtxt()`: Text format
- `np.savez()`: Multiple arrays

This completes the comprehensive tutorial for Chapter 13: Advanced Uses of NumPy.