Lab 4: Numpy and Matplotlib

CS6.201: Introduction to Software Systems

February 11, 2026

Introduction to Numpy

- Fundamental package for scientific computing with Python.
- Features an N-dimensional array object.
- Provides tools for linear algebra, Fourier transforms, random number capabilities.
- Serves as a building block for other packages (e.g., SciPy, Pandas).
- Works well with plotting libraries (matplotlib, seaborn, plotly etc.).
- Open source.

Why Use NumPy over Python Objects?

- NumPy is implemented in C, making it much faster than native Python lists.
- Uses contiguous memory allocation which improves cache performance.
- Supports vectorized operations (eliminating the need for loops).
- Performs hardware-level optimizations using optimized BLAS and LAPACK libraries.
- More memory-efficient compared to Python lists and tuples.
- Essential for high-performance scientific computing.

Basics

```python
import numpy as np

A = np.array([[1, 2, 3],
              [4, 5, 6]])

print(A)
# Output:
# [[1 2 3]
#  [4 5 6]]

Af = np.array([1, 2, 3], dtype=float)
# Output:
# [1. 2. 3.]
```

Array Creation and Initialization

```
np.arange(0, 1, 0.2)
# array([0., 0.2, 0.4, 0.6, 0.8])

np.linspace(0, 2*np.pi, 4)
# array ([0. 2.09, 4.18, 6.28])

A = np.zeros((2, 3))
# array([[0., 0., 0.],
#        [0., 0., 0.]])
# Also available: np.ones, np.diag

A.shape
# (2, 3)
```

Numpy arrays are mutable

```
A = np.zeros((2, 2))
# array([[ 0., 0.],
#        [ 0., 0.]])

C = A  # Reference assignment
C[0, 0] = 1

print(A)
# [[ 1. 0.]
#  [ 0. 0.]]
```

Note: You can prevent this behavior by using the `np.copy()` function to create an independent copy of the array.

Array Attributes

```
a = np.arange(10).reshape((2,5))

print(a.ndim)
# 2 (dimensions)

print(a.shape)
# (2, 5)

print(a.size)
# 10 (number of elements)

print(a.T)
# Transpose

a.dtype
# Data type
```

Basic Operations

```
a = np.arange(4)
# array ([0, 1, 2, 3])

b = np.array([2,3,2,4])

print(a * b)
# array([0, 3, 4, 12])

print(b - a)
# array([2, 2, 0, 1])

c = [2, 3, 4, 5]
print(a * c) # Works if c is list-like and broadcastable
# array([0, 3, 8, 15])
```

Vector Operations

```
# Example vectors
u = np.array([1, 2, 3])
v = np.array([1, 1, 1])

# Inner Product
np.inner(u, v)
# Output: 6

# Outer Product
np.outer(u, v)
# Output:
# array([[1, 1, 1],
#        [2, 2, 2],
#        [3, 3, 3]])

# Dot Product (Matrix Multiplication)
np.dot(u, v)
# Output: 6
```

Matrix Operations (Functions)

- `np.add(A, B)`: Adds matrices A and B element-wise.
- `np.subtract(A, B)`: Subtracts matrix B from matrix A element-wise.
- `np.multiply(A, B)`: Multiplies matrices A and B element-wise.
- `np.divide(A, B)`: Divides matrix A by matrix B element-wise.
- `np.dot(A, B)`: Computes the dot product of matrices A and B.

Matrix Operations (Linear Algebra)

- `A.transpose()` or `A.T`: Transposes matrix A (swaps rows and columns).
- `np.matmul(A, B)`: Performs matrix multiplication (supports 2D arrays). Similar to `np.dot` for 2D.
- `np.identity(n)`: Creates an $n \times n$ identity matrix.
- `np.trace(A)`: Computes the trace of matrix A (sum of diagonal elements).

Matrix Operations Example

```
# Define matrices
A = np.ones((3,2))
# array([[ 1., 1.],
#        [ 1., 1.],
#        [ 1., 1.]])

A.T
# array([[ 1., 1., 1.],
#        [ 1., 1., 1.]])

B = np.ones((2,3))
```

Matrix Operations Example (Continued)

```
1 np.dot(B.T, A.T)
2 # array([[ 2., 2., 2.],
3 #        [ 2., 2., 2.],
4 #        [ 2., 2., 2.]])
5
6 np.dot(A, B.T)
7 # Error: ValueError: shapes (3,2) and (3,2) not aligned:
8 # 2 (dim 1) != 3 (dim 0)
```

1-D Array Slicing Example

```
# Generate a 1-D array
a = np.array([0.25, 0.56, 0.98, 0.13, 0.72])

# Select elements from index 2 to the end
a[2:]
# array([ 0.98, 0.13, 0.72])

# Select elements from index 1 to 4 (exclusive)
a[1:4]
# array([ 0.56, 0.98, 0.13])

# Select every second element
a[::2]
# array([ 0.25, 0.98, 0.72])
```

1-D Array Slicing Example (Continued)

```
# Select elements in reverse order
a[::-1]
# array([ 0.72, 0.13, 0.98, 0.56, 0.25])

# Select elements with a negative step from index 4 to 0 (exclusive)
a[4:0:-1]
# array ([0.13, 0.98, 0.56, 0.25])

# Select the last element (negative indexing)
a[-1]
# Output: 0.72

# Select the second to last element
a[-2]
# Output: 0.13
```

2-D Array Slicing

```
a = np.array([
    [0.25, 0.56, 0.98, 0.13, 0.72],
    [0.43, 0.15, 0.67, 0.89, 0.24],
    [0.91, 0.78, 0.64, 0.38, 0.55],
    [0.19, 0.82, 0.13, 0.29, 0.71]
])

# Select the third row, all columns
a[2, :]
# Output: array([0.91, 0.78, 0.64, 0.38, 0.55])

# Select the 2nd and 3rd rows, all columns
a[1:3]
```

2-D Array Slicing (Continued)

```
# Select rows 1 through 3, columns 1 through 4
a[1:3, 1:4]
# Output: array([[ 0.15, 0.67, 0.89],
#                [ 0.78, 0.64, 0.38]])

# Select rows in reverse order, every other column
a[::-1, ::2]

# Select the last element from the last row
a[-1, -1]
# Output: 0.71
```

Reshaping Arrays

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

# Reshape the array to 3x4
reshaped = a.reshape(3, 4)
print("\nReshaped array (3x4):")
print(reshaped)
# [[ 0  1  2  3]
#  [ 4  5  6  7]
#  [ 8  9 10 11]]
```

Rules of Reshaping

- The total number of elements in the original array must equal the total number of elements in the reshaped array.
- $OriginalSize = \prod(original\_dims) = \prod(new\_dims)$
- Example 1: 1D array (12 elements) $\rightarrow$ 3×4 array (valid, $3 \times 4 = 12$).
- Example 2: 1D array (12 elements) $\rightarrow$ 3×5 array (invalid, $3 \times 5 \neq 12$).

```
a = np.arange(12)
reshaped = a.reshape(3, 4)        # Valid
invalid = a.reshape(3, 5)         # ValueError
```

Random Sampling with NumPy

- For random sampling we use the `np.random` module.

Random Sampling Functions (1)

- `np.random.rand(d0, d1, ..., dn)`: Generates random values from a uniform distribution in range $[0, 1)$ with specified shape.
- `np.random.randn(d0, d1, ..., dn)`: Generates random values from a standard normal distribution (mean=0, variance=1).
- `np.random.randint(lo, hi, size)`: Generates random integers from range $[lo, hi)$.

Random Sampling Functions (2)

- `np.random.choice(a, size, repl, p)`: Randomly samples elements from array `a`.
  - `size`: Number of elements to sample.
  - `repl=True`: Sampling with replacement (same element can be selected multiple times).
  - `repl=False`: Sampling without replacement (each element selected only once).
  - `p`: Probability distribution for sampling.

Understanding Seeds in Random Sampling

- A seed is an initial value used by a pseudorandom number generator (PRNG) to produce a sequence of numbers.
- A seed ensures that random sampling can be reproduced.
- `np.random.seed(seed)`: Sets the seed for the PRNG.

```
np.random.seed(42)
print(np.random.rand(2,3))
# array([[0.37454012, 0.95071431, 0.73199394],
#        [0.59865848, 0.15601864, 0.15599452]])

# Reset seed
np.random.seed(42)
print(np.random.rand(2,3))
# Generates the same numbers
```

Introduction to Masking

- Masking allows for filtering elements of an array based on conditions.
- Uses Boolean expressions to create a mask (True/False).
- Logical operators: & (AND), | (OR).

Masking Examples

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

# Masking with AND: > 2 AND < 5
mask = (arr > 2) & (arr < 5)
print(arr[mask]) # Output: [3 4]

# Masking with OR: < 2 OR > 5
mask = (arr < 2) | (arr > 5)
print(arr[mask]) # Output: [1 6]

# Masking with AND: > 2 AND even
mask = (arr > 2) & (arr % 2 == 0)
print(arr[mask]) # Output: [4 6]
```

Important NumPy Functions (1)

- `np.concatenate()`: Joins arrays along an axis.
- `np.mean()`, `np.median()`, `np.std()`: Statistics.
- `np.unique()`: Finds unique elements.
- `np.split()`: Splits array into sub-arrays.
- `np.argmax()`, `np.argmin()`: Indices of max/min values.

Important NumPy Functions (2)

- `np.argsort()`: Indices that would sort an array.
- `np.hstack()`: Stacks arrays horizontally (column-wise).
- `np.vstack()`: Stacks arrays vertically (row-wise).
- `np.repeat()`: Repeats elements of an array.
- `np.isnan()`: Tests element-wise for NaNs (Not a Number).
- `np.isin()`: Tests whether elements of an array are in another array.
- `np.newaxis()`: Adds a new axis to an array (used for reshaping).

NumPy Submodules

It doesn't end here...

- `np.linalg`: Linear algebra functions (decompositions, solving linear systems, eigenvalues).
- `np.random`: Suite of functions for generating random numbers and probability distributions.
- `np.fft`: Fast Fourier Transform functions for signal processing.

SIMD (Single Instruction Multiple Data)

- Scalar operations process one data point at a time.
- SIMD operations allow a single instruction to operate on multiple data points simultaneously.
- Commonly used in multimedia, graphics, scientific computing, and ML.
- Modern CPUs and GPUs leverage SIMD for faster parallel computations.

Restrictions of SIMD (Self-Study)

- SIMD only works on uniform operations.
- Cannot process different operations on different data elements simultaneously.
- Data must be organized in a vectorized format.

Advantages of SIMD (Self-Study)

- Faster computations for large datasets.
- Efficient for repetitive tasks like image and audio processing.
- Reduces the number of instructions executed.
- Boosts performance in machine learning algorithms.

Vectorization

- Vectorization refers to replacing explicit loops with optimized, low-level operations.
- Makes use of SIMD to perform multiple operations simultaneously.
- Eliminates the need for loops in Python.

Advantages of Vectorization (Self-Study)

- Improves speed and memory efficiency.
- Especially useful in deep learning for faster model training.
- Reduces the need for explicit loops, making code more concise.
- Utilizes optimized low-level implementations like BLAS and SIMD.
- Enables parallel execution on modern hardware (GPUs, multi-core CPUs).

Vectorization vs For-Loops

```python
import numpy as np
from time import time

NO_OF_ELEMENTS = int(1e6)
w = np.random.rand(NO_OF_ELEMENTS)
x = np.random.rand(NO_OF_ELEMENTS)
c = 0

start_time = time()
# Non-vectorized
for i in range(NO_OF_ELEMENTS):
    c += w[i] * x[i]

# Vectorized (Faster)
c = np.dot(w, x)
```

SIMD, Vectorization, and Broadcasting

Optimizing Performance in NumPy:

- **SIMD:** Hardware technique for processing multiple data points with a single instruction.
- **Vectorization:** Software technique leveraging NumPy to perform operations on entire arrays without explicit loops.
- **Broadcasting:** Powerful feature enabling operations on arrays of different shapes without reshaping, promoting memory efficiency.

Numpy Summary & Next Steps

- Numpy is powerful for arrays, linear algebra, and random sampling.
- **Next Steps:**
  - Pandas (Data manipulation)
  - SciPy (Advanced scientific computing)
  - Scikit-learn (Machine Learning)
  - PyTorch (Deep Learning)
- **Side Project:** Implement a neural network from scratch using NumPy to deepen your understanding (requires backpropagation).

Matplotlib

- A powerful Python library for static, animated, and interactive visualizations.
- Widely used in scientific computing.

Basic Plot

```
1  import matplotlib.pyplot as plt
2
3  x = [1, 2, 3, 4, 5]
4  y = [10, 20, 25, 30, 40]
5
6  plt.plot(x, y, marker='o', linestyle='-', color='b')
7  plt.xlabel('X Axis')
8  plt.ylabel('Y Axis')
9  plt.title('Basic Line Plot')
10 plt.show()
```

Scatter Plot

```
x = np.random.rand(50)
y = np.random.rand(50)
colors = np.random.rand(50)

plt.scatter(x, y, c=colors, alpha=0.5, cmap='viridis')
plt.colorbar()
plt.title('Scatter Plot')
plt.show()
```

Scatter and Line Plot Combined

```
x = np.linspace(0, 10, 100)
y = np.sin(x)
x_scat = np.random.rand(30) * 10
y_scat = np.sin(x_scat) + np.random.randn(30) * 0.1

plt.plot(x, y, label='Line Plot', color='b')
plt.scatter(x_scat, y_scat, label='Scatter Plot', color='r')

plt.legend()
plt.show()
```

Bar Plot

```
categories = ['A', 'B', 'C', 'D']
values = [3, 7, 1, 8]

plt.bar(categories, values, color=['red', 'blue', 'green', 'purple'])
plt.xlabel('Categories')
plt.ylabel('Values')
plt.title('Bar Chart')
plt.show()
# Use plt.barh() for horizontal
```

Side by Side Bar Plot

```
categories = ['A', 'B', 'C']
val1 = [10, 20, 30]
val2 = [15, 25, 35]
width = 0.25

r1 = np.arange(len(categories))
r2 = [x + width for x in r1]

plt.bar(r1, val1, color='b', width=width, label='Series 1')
plt.bar(r2, val2, color='r', width=width, label='Series 2')

plt.xticks([r + width/2 for r in r1], categories)
plt.legend()
plt.show()
```

Histogram

```
data = np.random.randn(1000)

plt.hist(data, bins=30, color='blue', edgecolor='black', alpha=0.7)
plt.xlabel('Value')
plt.ylabel('Frequency')
plt.title('Histogram')
plt.show()
```

Subplots

```
plt.figure(figsize=(8,6))

plt.subplot(2, 1, 1)
plt.plot([1, 2, 3], [4, 5, 6], 'r')
plt.title('First Subplot')

plt.subplot(2, 1, 2)
plt.plot([1, 2, 3], [10, 20, 30], 'b')
plt.title('Second Subplot')

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

Advanced Visualization: subplots()

- **subplot()**: Allows you to manually define grid layout (e.g., 2 rows, 2 columns) and select the plot's position.
- **subplots()**: A more flexible approach, returning a figure and an array of axes, ideal for complex layouts.
- Other plots: Pie Chart, Heatmap, Box Plot.

Questions?

Questions?