Week 1, Lecture 2

Introduction to NumPy

Lecture Overview

NumPy (Numerical Python) is the foundational library for numerical computing in Python. It provides efficient multidimensional arrays and mathematical functions essential for machine learning and data science.

Topics Covered:

- NumPy arrays vs Python lists
- Creating and manipulating arrays
- Array indexing and slicing
- Mathematical operations and broadcasting
- Array methods and functions
- Random number generation

1. Why NumPy?

```
import numpy as np
import time

print("Why NumPy?")
print("="*70)

# Speed comparison: Python list vs NumPy array
size = 1000000

# Python list
python_list = list(range(size))
start = time.time()
result_list = [x * 2 for x in python_list]
time_list = time.time() - start

# NumPy array
numpy_array = np.arange(size)
start = time.time()
result_array = numpy_array * 2
time_numpy = time.time() - start

print(f"Size: {size:,} elements")
print(f"Python list time: {time_list:.4f} seconds")
print(f"NumPy array time: {time_numpy:.4f} seconds")
print(f"Speedup: {time_list/time_numpy:.1f}x faster with NumPy!")

# Memory comparison
import sys
print(f"\nMemory comparison:")
print(f"Python list: {sys.getsizeof(python_list[:100]):,} bytes (100 elements)")
print(f"NumPy array: {numpy_array[:100].nbytes:,} bytes (100 elements)")
print(f"NumPy is more memory efficient!")
```

```
Why NumPy?
======================================================================
Size: 1,000,000 elements
Python list time: 0.0856 seconds
NumPy array time: 0.0034 seconds
Speedup: 25.2x faster with NumPy!

Memory comparison:
Python list: 856 bytes (100 elements)
NumPy array: 800 bytes (100 elements)
NumPy is more memory efficient!
```

2. Creating NumPy Arrays

```
import numpy as np

print("Creating NumPy Arrays")
print("="*70)

# From Python list
list_1d = [1, 2, 3, 4, 5]
arr_1d = np.array(list_1d)
print(f"From list: {arr_1d}")
print(f"Type: {type(arr_1d)}")
print(f"Data type: {arr_1d.dtype}")

# 2D array
list_2d = [[1, 2, 3], [4, 5, 6]]
arr_2d = np.array(list_2d)
print(f"\n2D array:\n{arr_2d}")
print(f"Shape: {arr_2d.shape}")
print(f"Dimensions: {arr_2d.ndim}")
print(f"Size: {arr_2d.size}")

# Special arrays
zeros = np.zeros((2, 3))
ones = np.ones((3, 2))
identity = np.eye(3)

print(f"\nZeros (2x3):\n{zeros}")
print(f"\nOnes (3x2):\n{ones}")
print(f"\nIdentity (3x3):\n{identity}")

# Range arrays
arange_arr = np.arange(0, 10, 2)  # start, stop, step
linspace_arr = np.linspace(0, 1, 5)  # start, stop, num_points

print(f"\narange(0, 10, 2): {arange_arr}")
print(f"linspace(0, 1, 5): {linspace_arr}")

# Random arrays
np.random.seed(42)
random_arr = np.random.rand(2, 3)  # Uniform [0, 1]
normal_arr = np.random.randn(2, 3)  # Standard normal

print(f"\nRandom uniform (2x3):\n{random_arr}")
print(f"\nRandom normal (2x3):\n{normal_arr}")
```

```
Creating NumPy Arrays
======================================================================
From list: [1 2 3 4 5]
Type: <class 'numpy.ndarray'>
Data type: int64

2D array:
[[1 2 3]
 [4 5 6]]
Shape: (2, 3)
Dimensions: 2
Size: 6

Zeros (2x3):
[[0. 0. 0.]
```

```
 [0. 0. 0.]]

Ones (3x2):
[[1. 1.]
 [1. 1.]
 [1. 1.]]

Identity (3x3):
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]

arange(0, 10, 2): [0 2 4 6 8]
linspace(0, 1, 5): [0.   0.25 0.5  0.75 1.  ]

Random uniform (2x3):
[[0.37454012 0.95071431 0.73199394]
 [0.59865848 0.15601864 0.15599452]]

Random normal (2x3):
[[ 0.64768854  1.52302986 -0.23415337]
 [-0.23413696  1.57921282  0.76743473]]
```

3. Array Indexing and Slicing

```
# Indexing and slicing
arr = np.arange(10)
print("Array Indexing and Slicing")
print("="*70)
print(f"Array: {arr}")

# Basic indexing
print(f"\nBasic indexing:")
print(f"First element arr[0]: {arr[0]}")
print(f"Last element arr[-1]: {arr[-1]}")
print(f"Third element arr[2]: {arr[2]}")

# Slicing
print(f"\nSlicing:")
print(f"arr[2:5]: {arr[2:5]}")
print(f"arr[:4]: {arr[:4]}")
print(f"arr[6:]: {arr[6:]}")
print(f"arr[::2]: {arr[::2]}")  # Every other element
print(f"arr[::-1]: {arr[::-1]}")  # Reverse

# 2D indexing
arr_2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print(f"\n2D array:\n{arr_2d}")
print(f"\n2D indexing:")
print(f"Element [1, 2]: {arr_2d[1, 2]}")
print(f"First row: {arr_2d[0, :]}")
print(f"Second column: {arr_2d[:, 1]}")
print(f"Subarray [0:2, 1:3]:\n{arr_2d[0:2, 1:3]}")

# Boolean indexing
arr = np.array([1, 2, 3, 4, 5, 6])
print(f"\nBoolean indexing:")
print(f"Array: {arr}")
mask = arr > 3
print(f"Mask (arr > 3): {mask}")
print(f"Elements > 3: {arr[mask]}")
print(f"Direct: arr[arr > 3] = {arr[arr > 3]}")

# Fancy indexing
indices = [0, 2, 4]
print(f"\nFancy indexing:")
print(f"arr[[0, 2, 4]]: {arr[indices]}")
```

```
Array Indexing and Slicing
======================================================================
Array: [0 1 2 3 4 5 6 7 8 9]

Basic indexing:
First element arr[0]: 0
Last element arr[-1]: 9
Third element arr[2]: 2

Slicing:
arr[2:5]: [2 3 4]
arr[:4]: [0 1 2 3]
arr[6:]: [6 7 8 9]
arr[::2]: [0 2 4 6 8]
arr[::-1]: [9 8 7 6 5 4 3 2 1 0]

2D array:
```

```
[[1 2 3]
 [4 5 6]
 [7 8 9]]

2D indexing:
Element [1, 2]: 6
First row: [1 2 3]
Second column: [2 5 8]
Subarray [0:2, 1:3]:
[[2 3]
 [5 6]]

Boolean indexing:
Array: [1 2 3 4 5 6]
Mask (arr > 3): [False False False  True  True  True]
Elements > 3: [4 5 6]
Direct: arr[arr > 3] = [4 5 6]

Fancy indexing:
arr[[0, 2, 4]]: [1 3 5]
```

4. Array Operations

```
# Mathematical operations
a = np.array([1, 2, 3, 4])
b = np.array([5, 6, 7, 8])

print("Array Operations")
print("="*70)
print(f"a = {a}")
print(f"b = {b}")

# Element-wise operations
print(f"\nElement-wise operations:")
print(f"a + b = {a + b}")
print(f"a - b = {a - b}")
print(f"a * b = {a * b}")
print(f"a / b = {a / b}")
print(f"a ** 2 = {a ** 2}")

# Scalar operations
print(f"\nScalar operations:")
print(f"a + 10 = {a + 10}")
print(f"a * 2 = {a * 2}")
print(f"a / 2 = {a / 2}")

# Mathematical functions
arr = np.array([0, np.pi/2, np.pi])
print(f"\nMathematical functions:")
print(f"Array: {arr}")
print(f"sin(arr): {np.sin(arr)}")
print(f"cos(arr): {np.cos(arr)}")
print(f"exp(arr): {np.exp(arr)}")
print(f"log([1, 2, 4, 8]): {np.log([1, 2, 4, 8])}")

# Aggregation functions
data = np.array([1, 2, 3, 4, 5])
print(f"\nAggregation functions:")
print(f"Data: {data}")
print(f"Sum: {np.sum(data)}")
print(f"Mean: {np.mean(data)}")
print(f"Std: {np.std(data):.4f}")
print(f"Min: {np.min(data)}")
print(f"Max: {np.max(data)}")

# 2D aggregations
matrix = np.array([[1, 2, 3], [4, 5, 6]])
print(f"\n2D aggregations:")
print(f"Matrix:\n{matrix}")
print(f"Sum all: {np.sum(matrix)}")
print(f"Sum axis=0 (cols): {np.sum(matrix, axis=0)}")
print(f"Sum axis=1 (rows): {np.sum(matrix, axis=1)}")
```

```
Array Operations
======================================================================
a = [1 2 3 4]
b = [5 6 7 8]

Element-wise operations:
a + b = [ 6  8 10 12]
a - b = [-4 -4 -4 -4]
a * b = [ 5 12 21 32]
```

```
a / b = [0.2        0.33333333 0.42857143 0.5       ]
a ** 2 = [ 1  4  9 16]

Scalar operations:
a + 10 = [11 12 13 14]
a * 2 = [2 4 6 8]
a / 2 = [0.5 1.  1.5 2. ]

Mathematical functions:
Array: [0.         1.57079633 3.14159265]
sin(arr): [0.0000000e+00 1.0000000e+00 1.2246468e-16]
cos(arr): [ 1.000000e+00  6.123234e-17 -1.000000e+00]
exp(arr): [ 1.          4.81047738 23.14069263]
log([1, 2, 4, 8]): [0.         0.69314718 1.38629436 2.07944154]

Aggregation functions:
Data: [1 2 3 4 5]
Sum: 15
Mean: 3.0
Std: 1.4142
Min: 1
Max: 5

2D aggregations:
Matrix:
[[1 2 3]
 [4 5 6]]
Sum all: 21
Sum axis=0 (cols): [5 7 9]
Sum axis=1 (rows): [ 6 15]
```

5. Broadcasting

```
# Broadcasting - operating on arrays of different shapes
print("Broadcasting")
print("="*70)

# 1D + scalar
a = np.array([1, 2, 3])
b = 10
print(f"Array: {a}")
print(f"Scalar: {b}")
print(f"a + b: {a + b}")

# 2D + 1D
matrix = np.array([[1, 2, 3], [4, 5, 6]])
row = np.array([10, 20, 30])

print(f"\nMatrix (2x3):\n{matrix}")
print(f"Row (3,): {row}")
print(f"matrix + row:\n{matrix + row}")

# Column broadcasting
col = np.array([[10], [20]])
print(f"\nColumn (2x1):\n{col}")
print(f"matrix + col:\n{matrix + col}")

# Broadcasting rules demonstration
print(f"\nBroadcasting rules:")
print(f"(2, 3) + (3,) -> (2, 3)  ✓")
print(f"(2, 3) + (2, 1) -> (2, 3)  ✓")
print(f"(2, 3) + (1, 3) -> (2, 3)  ✓")

# Practical example: normalize columns
data = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
col_mean = np.mean(data, axis=0)
col_std = np.std(data, axis=0)

normalized = (data - col_mean) / col_std
print(f"\nNormalization example:")
print(f"Original:\n{data}")
print(f"Column means: {col_mean}")
print(f"Normalized:\n{normalized}")
```

```
Broadcasting
======================================================================
Array: [1 2 3]
Scalar: 10
a + b: [11 12 13]

Matrix (2x3):
[[1 2 3]
 [4 5 6]]
Row (3,): [10 20 30]
matrix + row:
[[11 22 33]
 [14 25 36]]

Column (2x1):
[[10]
 [20]]
matrix + col:
```

```
[[11 12 13]
 [24 25 26]]

Broadcasting rules:
(2, 3) + (3,) -> (2, 3)  ✓
(2, 3) + (2, 1) -> (2, 3)  ✓
(2, 3) + (1, 3) -> (2, 3)  ✓

Normalization example:
Original:
[[1 2 3]
 [4 5 6]
 [7 8 9]]
Column means: [4. 5. 6.]
Normalized:
[[-1.22474487 -1.22474487 -1.22474487]
 [ 0.          0.          0.        ]
 [ 1.22474487  1.22474487  1.22474487]]
```

Summary

Key Takeaways:

- NumPy arrays are faster and more memory efficient than Python lists
- Arrays support element-wise operations and broadcasting
- Boolean and fancy indexing enable powerful data selection
- Rich set of mathematical and statistical functions
- Foundation for all numerical computing in Python

Practice Exercises

1. Create a 5x5 matrix of random numbers and normalize each row

2. Find all elements in an array that are between 10 and 20

3. Compute the dot product of two matrices

4. Create a checkerboard pattern using NumPy