CS6.201: Introduction to Software Systems
Python Session - 2
`numpy` and `matplotlib`

Prakhar Jain, Amey Karan, Neel Amrutia, Vivek Hruday, Chirag Dhamija

March 5, 2025

Laptops Off!

Real-Time Operating System

This is the answer for whoever asked

Embedded systems almost always run **real-time operating systems**. A real-time system is used when rigid time requirements have been placed on the operation of a processor or the flow of data; thus, it is often used as a control device in a dedicated application. Sensors bring data to the computer. The computer must analyze the data and possibly adjust controls to modify the sensor inputs. Systems that control scientific experiments, medical imaging systems, industrial control systems, and certain display systems are real-time systems. Some automobile-engine fuel-injection systems, home-appliance controllers, and weapon systems are also real-time systems.

A real-time system has well-defined, fixed time constraints. Processing ***must*** be done within the defined constraints, or the system will fail. For instance, it would not do for a robot arm to be instructed to halt ***after*** it had smashed into the car it was building. A real-time system functions correctly only if it returns the correct result within its time constraints. Contrast this system with a traditional laptop system where it is desirable (but not mandatory) to respond quickly.

Examples of Real-Time Operating Systems include Wind River VxWorks, QNX, FreeRTOS, RTLinux, etc.

Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts, 10th Edition*. Wiley, 2018. ISBN: 978-1-118-06333-0. URL: http://os-book.com/OS10/index.html

numpy

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.
- blah, blah, blah...

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** (*covered later*), 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.]])
# np.ones, np.diag

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

Numpy arrays are mutable

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

C = A
C[0, 0] = 1

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

```

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

```python
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)      # array([0, 3, 8, 15])
```

Vector Operations

```python
# Example vectors
u = [1, 2, 3]
v = [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

- `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

- `np.transpose(A)`: Transposes matrix A (swaps rows and columns).
- `np.matmul(A, B)`: Performs matrix multiplication (supports 2D arrays). Performs the same functions as `np.dot`
- `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

```
# 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))
# array([[ 1., 1., 1.],
#        [ 1., 1., 1.]])
```

Matrix Operations

```
# Matrix operations

np.dot(B.T, A.T)
# array([[ 2., 2., 2.],
#        [ 2., 2., 2.],
#        [ 2., 2., 2.]])

np.dot(A, B.T)
# Error: ValueError: shapes (3,2) and (3,2) not aligned: ...
# ... 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)

```python
# 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] # 0.72

# Select the second to last element (negative indexing)
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]
# Output: array([[ 0.43,  0.15,  0.67,  0.89,  0.24],
#                [ 0.91,  0.78,  0.64,  0.38,  0.55]])
```

2-D Array Slicing (Continued)

```
# Select rows 1 through 3 (exclusive of 3), columns 1 through 4
↪  (exclusive of 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]
# Output: array([[ 0.19,  0.13,  0.71],
#                [ 0.91,  0.64,  0.55],
#                [ 0.43,  0.67,  0.24],
#                [ 0.25,  0.98,  0.72]])

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

Reshaping Arrays

```python
# Create a 1D array of size 12
a = np.arange(12)
print("Original array:")
print(a)
# Original array:
# [ 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)

# Reshaped array (3x4):
# [[ 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:

$$\text{Original Size} = \prod(\text{original dimensions}) = \prod(\text{new dimensions})$$

- Example 1: A 1D array with 12 elements can be reshaped into a 3×4 array because $3 \times 4 = 12$.
- Example 2: Attempting to reshape a 1D array with 12 elements into a 3×5 array raises an error because $3 \times 5 \neq 12$.

```python
# Reshape Example
a = np.arange(12)

# Valid reshaping
reshaped = a.reshape(3, 4)

# Invalid reshaping (raises an error)
invalid = a.reshape(3, 5)  # ValueError: cannot reshape array
```

Random Sampling with NumPy

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

Random Sampling Functions - 1

- `np.random.rand(d0, d1, ..., dn)`: Generates random values from a uniform distribution in the range $[0, 1)$ with a specified shape. For example, `rand(2, 3)` will generate a 2x3 array of random values.
- `np.random.randn(d0, d1, ..., dn)`: Generates random values from a standard normal distribution (mean=0, variance=1) with the given shape. For example, `randn(2, 3)` will produce a 2x3 array with values from the standard normal distribution.

Random Sampling Functions - 2

- `np.random.randint(lo, hi, size)`: Generates random integers from the range $[lo, hi)$, where `lo` is the lower bound (inclusive) and `hi` is the upper bound (exclusive). The `size` argument specifies the shape of the output array. For example, `randint(0, 10, (2, 3))` will produce a 2x3 array with integers between 0 and 9.
- `np.random.choice(a, size, repl, p)`: Randomly samples elements from array `a`. The `size` specifies the number of elements to sample. If `repl=True`, the same element can be selected multiple times (sampling with replacement), otherwise, each element can only be selected once (sampling without replacement). The `p` parameter specifies the probability distribution for the sampling.

Understanding Seeds in Random Sampling

- In random sampling, a **seed** is an initial value used by a pseudorandom number generator (PRNG) to produce a sequence of random numbers.
- The seed ensures that random sampling can be reproduced. If you use the same seed value, you will get the same random numbers each time, which is useful for debugging and reproducibility in experiments.

Setting a Seed in NumPy

`np.random.seed(seed)`: Sets the seed for the random number generator. Once the seed is set, all subsequent random number generation will be deterministic and based on that seed.

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

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

# Generates the same numbers every time after setting the seed
```

Introduction to Masking in NumPy

- Masking allows for filtering elements of an array based on specified conditions.
- The result is an array where only the elements satisfying the condition(s) are included.
- Masking uses Boolean expressions to create a mask (True/False values) that is applied to the array.
- Common logical operators used in masking:
  - & (AND) – for multiple conditions that must be true
  - | (OR) – for conditions where at least one must be true

Example 1: Using & (AND) and — (OR)

- Masking with AND condition: values greater than 2 AND less than 5

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

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

- Masking with OR condition: values less than 2 OR greater than 5

```
mask = (arr < 2) | (arr > 5)
masked_array = arr[mask]
print(masked_array)  # Output: [1 6]
```

Example 2: Using & (AND) for Multiple Conditions

- Masking with AND condition: values greater than 2 AND even numbers

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

# Masking with AND condition: greater than 2 AND even
↪ numbers
mask = (arr > 2) & (arr % 2 == 0)
masked_array = arr[mask]
print(masked_array)  # Output: [4 6]
```

Important NumPy Functions

- `np.concatenate()` : Joins two or more arrays along an existing axis.
- `np.mean()` : Computes the mean (average) of array elements.
- `np.median()` : Computes the median of array elements.
- `np.std()` : Computes the standard deviation of array elements.
- `np.unique()` : Finds the unique elements of an array.
- `np.split()` : Splits an array into multiple sub-arrays.
- `np.argmax()` : Returns the indices of the maximum values along an axis.
- `np.argmin()` : Returns the indices of the minimum values along an axis.

Important NumPy Functions

- `np.argsort()` : Returns the indices that would sort an array.
- `np.hstack()` : Stacks arrays in sequence horizontally (column-wise).
- `np.vstack()` : Stacks arrays in sequence 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 or increasing dimensions.

It doesn't end here...

NumPy's submodules:

- `np.linalg`: Linear algebra functions, such as matrix decompositions, solving linear systems, eigenvalues/eigenvectors, and more.
- `np.random`: Provides a suite of functions for generating random numbers, including probability distributions and random sampling.
- `np.fft`: Fast Fourier Transform functions for signal processing, spectral analysis, and efficient computation of discrete Fourier transforms.
- and more...

https://numpy.org/doc/stable/reference/index.html

SIMD

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

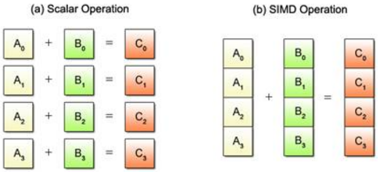

Figure: Scalar vs. SIMD Operations

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.

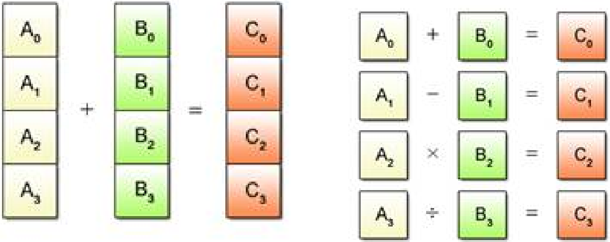

Figure: Processable (Left) vs Unprocessable (Right) SIMD Patterns

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.
- Vectorization makes use of **SIMD** to perform multiple operations simultaneously.
- Eliminates the need for loops in high-level languages like Python.

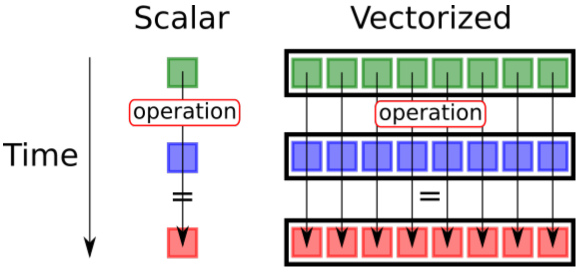

Figure: Vectorization Workflow

Advantages of Vectorization (Self-Study)

- Improves speed and memory efficiency.
- Especially useful in deep learning for faster model training on large datasets.
- Reduces the need for explicit loops, making code more concise and readable.
- Utilizes optimized low-level implementations like BLAS and SIMD instructions.
- Enables parallel execution on modern hardware such as GPUs and multi-core CPUs.
- Minimizes overhead from control flow statements.

Vectorization vs For-Loops

Example: Dot Product

```
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 = np.zeros(NO_OF_ELEMENTS)

start_time = time()
for i in range(NO_OF_ELEMENTS): # Non-vectorized implementation
    c += w[i] * x[i]
c = np.dot(w, x) # Vectorized Implementation
```

- Vectorized version is **faster** due to hardware-level optimizations.
- Most `NumPy` functions support vectorized operations.

SIMD, Vectorization, and Broadcasting

Optimizing Performance in NumPy:

- `SIMD` (Single Instruction, Multiple Data): A technique for processing multiple data points with a single instruction, enabling parallel processing for better performance.
- `Vectorization`: Leveraging NumPy's ability to perform operations on entire arrays without explicit loops, allowing for faster and more efficient computations.
- `Broadcasting`: A powerful feature that enables NumPy to perform operations on arrays of different shapes without needing to reshape them, promoting memory efficiency.

Explore Broadcasting to get a better understanding of how NumPy works!

Summary

- Numpy provides a powerful toolkit for numerical computation.
- Offers support for arrays, linear algebra, Fourier transforms, and random sampling.
- Highly optimized and widely used in scientific computing.
- **Next Steps:**
  - Explore `pandas` for data manipulation.
  - Learn `SciPy` for advanced scientific computing.
  - Use `scikit-learn` for machine learning.
  - Try `XGBoost` for gradient boosting.
  - `PyTorch` for deep learning (Learning `PyTorch` will prepare you for most of the ML/DL courses here).
- **Side Project:** Implement a neural network from scratch using `NumPy` to deepen your understanding. You'll need an understanding of how backpropagation works.

Questions?

`matplotlib`

Matplotlib is a powerful Python library used for creating static, animated, and interactive visualizations. It is widely used for data visualization in scientific computing.

Installation

To install Matplotlib, use the following command:

```
pip install matplotlib
```

Basic Plot

A simple line plot can be created as follows:

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

x = [1, 2, 3, 4, 5]
y = [10, 20, 25, 30, 40]

plt.plot(x, y, marker='o', linestyle='-', color='b')
plt.xlabel('X Axis')
plt.ylabel('Y Axis')
plt.title('Basic Line Plot')
plt.show()
```

Scatter Plot

A scatter plot can be created using:

```
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 Plot and Line Plot on the Same Axes

To combine a scatter plot and a line plot in the same figure, you can use both `plot()` and `scatter()` functions:

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

plt.plot(x, y, label='Line Plot', color='b')  # Line plot
plt.scatter(x_scatter, y_scatter, label='Scatter Plot',
    color='r')  # Scatter plot

plt.xlabel('X Axis')
plt.ylabel('Y Axis')
plt.title('Scatter Plot and Line Plot')
plt.legend()

plt.show()
```

Bar Plot

Creating a bar chart:

```python
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()
```

You can also use `plt.barh()` to create a horizontal chart.

Side by Side Bar Plot

```python
categories = ['A', 'B', 'C']
values1 = [10, 20, 30]
values2 = [15, 25, 35]

barWidth = 0.25

r1 = np.arange(len(categories))  # Positions for the first set
↪   of bars
r2 = [x + barWidth for x in r1]  # Positions for the second set
↪   of bars

plt.bar(r1, values1, color='b', width=barWidth, label='Series
↪   1')
plt.bar(r2, values2, color='r', width=barWidth, label='Series
↪   2')

```

Side by Side Bar Plot (Continued)

```
# Add labels and title
plt.xlabel('Categories')
plt.ylabel('Values')
plt.title('Side by Side Bar Plot')
# Add custom x-axis tick labels
plt.xticks([r + barWidth / 2 for r in r1], categories)

# Add legend
plt.legend()

# Show the plot
plt.show()
```

Histogram

A histogram is useful for visualizing data distributions:

```
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()
```

More plots...

Additionally, `matplotlib` supports other types of plots like:

- Pie Chart
- Heatmap
- Box Plot

Subplots and Figures

Matplotlib provides the `subplot` and `subplots` functions to create multiple plots within a single figure.

Using Subplot

The `subplot` function allows creating multiple plots by specifying the grid layout as `subplot(rows, cols, index)`.

```
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()
```

Now let's see some advanced topics and libraries you can dive into to enhance your data visualization skills.

`subplots()` in Matplotlib

Creating multiple plots within a single figure can be achieved with `subplot()` or `subplots()` functions. These allow you to display several visualizations side by side for better comparison.

- `subplot()` allows you to manually define grid layout (e.g., 2 rows and 2 columns) and select the plot's position.
- `subplots()` provides a more flexible approach, returning a figure and an array of axes, ideal for complex layouts.

3D Plotting with Matplotlib

Matplotlib supports 3D plotting via the `mpl_toolkits.mplot3d` module, which is helpful for visualizing relationships in multivariate data.

- Create 3D scatter plots, surface plots, wireframes, etc.
- Ideal for datasets with more than two variables.

Seaborn Integration

Seaborn, built on top of Matplotlib, simplifies the creation of complex visualizations and offers a high-level interface with built-in themes and color palettes.

- Seaborn handles statistical plots like categorical plots and pair plots with minimal code.
- It integrates well with pandas dataframes, automatically handling data aggregation and statistical analysis.

Interactive Plotting with Plotly

Plotly is a library for creating interactive plots, making it ideal for dynamic data exploration in web applications or Jupyter notebooks.

- Supports various charts, including 3D plots, heatmaps, and geographic maps.
- Features like zooming, hovering, and clicking on data points for detailed information enhance data interactivity.

Questions?


MACHINE LEARNING
Why is it, when something happens, it is always you three?
Numpy
Pandas
Matplotlib

Hands On!