# NumPy Library

Sumedha Kulasekara
Session 2

## NumPy library

The NumPy library is a popular Python library used for scientific computing applications, and is an acronym for "Numerical Python".To make it as fast as possible, NumPy is written in C and Python.

NumPy's operations are divided into three main categories:

1. Fourier Transform and Shape Manipulation,
2. Mathematical and Logical Operations
3. Linear Algebra and Random Number Generation.

## Installation

```
!pip install numpy
```

## Importing NumPy

```
import numpy as np

print(np.__version__)  #Checking NumPy Version
```

## NumPy Arrays

NumPy arrays are the main data structure in NumPy. They are similar to Python lists but more efficient for numerical operations.

**Creating a 1D Array**

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

## Creating a 2D Array

```
arr2d = np.array([[1, 2, 3], [4, 5, 6]])
print(arr2d)
```

## Array Attributes

You can check important properties of a NumPy array, such as its shape, size, data type, and dimension.

```
arr = np.array([1, 2, 3, 4, 5])
print("Shape:", arr.shape)
print("Size:", arr.size)
print("Data Type:", arr.dtype)
print("Dimension:", arr.ndim)
```

## Array Operations

NumPy arrays support vectorized operations, which makes mathematical computations on arrays much faster than using native Python lists.

### Adding two arrays

```
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])
result = arr1 + arr2
print(result)
```

### Scalar multiplication

```
arr = np.array([1, 2, 3])
result = arr * 2
print(result)
```

### Applying Custom Functions

```
def custom_func(x):
    return x**2 + 2*x + 1

a1 = np.array([1, 2, 3, 4])
result = custom_func(a1)
print(result)
```

## Array Indexing and Slicing

You can access elements in a NumPy array just like in Python lists. NumPy arrays also support advanced indexing, which allows you to slice and access sub-arrays.

**Indexing**

```
arr = np.array([10, 20, 30, 40, 50])
print(arr[2])  # Accessing the 3rd element (index 2)

arr2d = np.array([[1,2,3,4,5], [6,7,8,9,10]])
print('2nd element on 1st row: ', arr2d [0, 1])

#Negative Indexing
print('Last element from 2nd dim: ', arr[1, -1])
```

**Exercise** - Access the element on the 2nd row, 4th column ?

## Slicing

```
arr = np.array([10, 20, 30, 40, 50])
print(arr[1:4])  # Slice elements from index 1 to 3
```

**Exercise -** Get third and fourth elements from the following array and add them.

```
arr = np.array([10, 52,33, 84])
```

## Reshaping Arrays

You can change the shape of NumPy arrays using the reshape function

### Reshaping a 1D array to a 2D array

```
arr = np.array([1, 2, 3, 4, 5, 6])
reshaped_arr = arr.reshape(2, 3)  # 2 rows and 3 columns
print(reshaped_arr)
```

## Array Broadcasting

Broadcasting is a powerful feature in NumPy that allows operations between arrays of different shapes.

```
arr = np.array([1, 2, 3])
result = arr + 5  # Adding a scalar to each element of the array
print(result)
```

**Mathematical Functions**

NumPy provides a variety of mathematical functions like sin, cos, log, and others to apply element-wise operations on arrays.

**Trigonometric operations**

```
arr = np.array([0, np.pi/2, np.pi])
print(np.sin(arr))  # Sine of each element in the array
```

**Random Module**

NumPy includes the random module for generating random numbers..

```
random_arr = np.random.random(5)  # Generates 5 random numbers between 0 and 1
print(random_arr)

random_values = np.random.rand(3, 4)  # 3 rows, 4 columns
print(random_values)

#Random Integer Generation
random_integers = np.random.randint(0, 10, size=(2, 5))  # 2 rows, 5 columns
print(random_integers)
```

## Linear Algebra with NumPy

NumPy includes a module called numpy.linalg for performing linear algebra operations such as matrix multiplication, eigenvalues, singular value decomposition (SVD), etc.

### Matrix Multiplication

```
# Creating two 2D arrays (matrices)
A = np.array([[1, 2], [3, 4]])
B = np.array([[5, 6], [7, 8]])

# Matrix multiplication
result = np.dot(A, B)
print(result)

OR

result = A @ B
print(result)
```

## Solving Linear Systems

NumPy provides a function numpy.linalg.solve() to solve systems of linear equations.
Solving a system of linear equations We want to solve for x in the equation Ax = b, where:

**$2x+3y = 5$**
**$x+2y = 4$**

A = [[2, 3], [1, 2]]

b = [5, 4]

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

# Solving for x
x = np.linalg.solve(A, b)
print(x)
```

**Exercise:-**

1)

$3x+y = 9$

$x+2y = 8$

2)

$6x + 3y = 1$

$3x - 4y = 2$

3)

$2x + 4y = 5$

$6x + 8y = 6$.

## Singular Value Decomposition (SVD)

SVD is a method for decomposing a matrix into three other matrices, and it's widely used in tasks like dimensionality reduction.

SVD is a powerful matrix factorization technique used in many areas such as signal processing, statistics, and machine learning. It decomposes a matrix into three distinct matrices:

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

# Singular Value Decomposition
U, S, Vt = np.linalg.svd(A)

print("U matrix:")
print(U)

print("Singular values:")
print(S)

print("Vt matrix:")
print(Vt)
```

**Reproducibility:**

The primary purpose of random.seed() is to ensure reproducibility. In data science, simulations, or any application where random processes are involved, setting the seed allows you to get the same "random" results every time you run your code. This is crucial for debugging, testing, and sharing your work.

```
# Without setting a seed, results will vary
print(np.random.randint(1, 100))
print(np.random.randint(1, 100))

# Setting a seed ensures reproducible results with np.random
np.random.seed(123)
print(np.random.randint(1, 100))


random.seed()    # Reset the seed to system-based randomness
```

## Statistical Functions

Generate random numbers from a normal (Gaussian) distribution using np.random.normal().

**Calculating mean, median, and standard deviation**

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

mean = np.mean(arr) # Mean
print("Mean:", mean)

median = np.median(arr) # Median
print("Median:", median)

std_dev = np.std(arr) # Standard deviation
print("Standard Deviation:", std_dev)
```

**Generating random samples from a normal distribution**

```
mean = 0
std_dev = 1
size = 5

random_normal = np.random.normal(mean, std_dev, size)
print(random_normal)
```

## Generating random samples from a Binomial Distribution

The Binomial Distribution models the number of successes in a fixed number of independent trials, where each trial has only two outcomes: success or failure.

**Key Parameters:**

n: number of trials

p: probability of success in each trial

**Example**:

Flipping a coin 10 times (n = 10), and counting how many times you get heads (p = 0.5).

```
np.random.binomial(n=10, p=0.5, size=5)
```

## Exercise

1) A researcher measures the height of plants grown in a lab. The heights follow a normal distribution with a mean of **100 cm** and a standard deviation of **15 cm**. (use **np.random.seed(42)**)

   a) List the 5 simulated plant heights.
   b) Calculate the sample mean and compare it with the theoretical mean (100 cm).
   c) Why might the sample mean differ from the theoretical mean?

2) A machine produces items with a 50% chance of being defective. Each shift, 10 items are inspected. Simulate 5 shifts:

   a) List the 5 simulated results (number of defective items per shift).
   b) Calculate the mean number of defects and compare your result with the theoretical mean.
   c) Explain why the sample mean may differ from the expected value.

## Answers

1) a)

```
np.random.seed(42)
samples = np.random.normal(loc=100, scale=15, size=5)
print(samples)
mean = np.mean(samples)
```

2) a)

```
samples = np.random.binomial(n=10, p=0.5, size=5)
print(samples)
mean = np.mean(samples)
```

## Correlation matrix and Covariance matrix

NumPy allows you to compute the correlation matrix and covariance matrix for a set of variables.

```
data1 = np.array([1, 2, 3, 4, 5])
data2 = np.array([5, 4, 3, 2, 1])

correlation_coefficient = np.corrcoef(data1, data2)[0, 1]

covariance_matrix = np.cov(data1, data2)
```

NumPy includes functions for random number generation and sampling from various probability distributions.

```
random_data = np.random.randn(20)

binomial_data = np.random.binomial(n=10, p=0.5, size=20)
normal_data = np.random.normal(loc=10, scale=2, size=20)
```

## Advanced Array Operations

### Broadcasting

Broadcasting allows NumPy to perform element-wise operations on arrays of different shapes.

```
# 2D array
arr2d = np.array([[1, 2, 3], [4, 5, 6]])

# 1D array
arr1d = np.array([10, 20, 30])

# Broadcasting: Adding 1D array to 2D array
result = arr2d + arr1d
print(result)
```

### Element-wise Operations

```
arr = np.array([1, 4, 9, 16])

# Applying square root element-wise
sqrt_arr = np.sqrt(arr)
print(sqrt_arr)
```

## Broadcasting with arrays of different shapes

```
arr1 = np.array([1, 2, 3])  # Shape (1,3)
arr2 = np.array([[10], [20], [30]])  # Shape (3,1)

# Broadcasting: The second array is automatically expanded to match the shape of the first
array
result = arr1 + arr2
print(result)
```

## Advanced Indexing Techniques

### Boolean Indexing

You can use boolean conditions to select elements from an array.

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

# Selecting elements greater than 25
selected = arr[arr > 25]
print(selected)
```

### Fancy Indexing

Fancy indexing allows you to select elements based on an array of indices.

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

# Selecting elements at indices 1, 3, and 4
selected = arr[[1, 3, 4]]
print(selected)
```

## Handling Missing Data with NumPy

**NaN (Not a Number)**
NaN is often used to represent missing or undefined values in arrays**.**
**Checking for NaN and replacing NaN values**

```
arr = np.array([1, 2, np.nan, 4, 5])
print(arr)

# Check for NaN values
print(np.isnan(arr))

# Replace NaN with a specific value (e.g., 0)
arr_no_nan = np.nan_to_num(arr, nan=0)
print(arr_no_nan)
```

## Working with Date and Time

You can use NumPy's datetime module to handle date and time operations, although for more advanced tasks, you might prefer using pandas.

```
# Create a date array from string
dates = np.array(['2025-03-01', '2025-03-02', '2025-03-03'], dtype='datetime64[D]')
print(dates)

# Performing date arithmetic (adding days)
new_dates = dates + np.arange(3)
print(new_dates)
```

### Explain …

np.arange(3) creates an array of integers [0, 1, 2], which represent days to be added to each date in the dates array.

The operation dates + np.arange(3) performs element-wise addition of the dates array and the array [0, 1, 2]. This means:

- 2025-03-01 + 0 results in 2025-03-01
- 2025-03-02 + 1 results in 2025-03-03
- 2025-03-03 + 2 results in 2025-03-05

**Matrix Inversion**

Inverting matrices is another fundamental linear algebra operation. It’s useful in solving linear systems of equations and in various other applications.

```
A = np.array([[4, 7], [2, 6]])

# Matrix inversion
A_inv = np.linalg.inv(A)
print(A_inv)
```

## Advanced Array Manipulation

**Reshaping with Multiple Dimensions**
Reshaping allows you to manipulate the dimensionality of an array.

```
arr = np.arange(27)  # Creates an array of values from 0 to 26
reshaped_arr = arr.reshape(3, 3, 3)  # Reshapes it into a 3x3x3 3D array
print(reshaped_arr)
```

## Stacking Arrays

Stacking allows you to join multiple arrays along a new axis. It’s useful when you need to combine datasets.

```
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])

stacked = np.stack((arr1, arr2), axis=0)  # Stack along the first axis (rows)
print(stacked)
```

## Splitting Arrays

You can split arrays into multiple smaller arrays. This can be helpful for tasks like cross-validation in machine learning.

```
arr = np.array([1, 2, 3, 4, 5, 6])
splits = np.split(arr, 2)  # Split into 2 equal parts
print(splits)
```

## Working with Structured Arrays

Structured arrays allow you to define arrays with different data types for each column. This is useful for working with tabular data, similar to pandas DataFrames

```
dt = np.dtype([('name', 'S10'), ('age', 'i4'), ('marks', 'f4')])
students = np.array([('John', 20, 85.5), ('Sara', 22, 92.3)], dtype=dt)

# Accessing fields
print(students['name'])  # Accessing 'name' field
print(students['age'])   # Accessing 'age' field
```

## Universal Functions (ufuncs)

ufuncs are functions that operate element-wise on arrays. These functions are highly optimized and are a central feature of NumPy.

```
arr = np.array([1, 2, 3, 4])
result = np.exp(arr)  # Exponentiation element-wise
print(result)
```

## Polynomials in NumPy

You can work with polynomials using the numpy.poly module. This is useful for polynomial regression, root finding, and fitting problems.

```
# Define a polynomial: 2x^2 + 3x + 1
p = np.poly1d([2, 3, 1])
print(p)
```

## Memory Management and Performance

Understanding how NumPy handles memory, views vs copies, and the use of np.copy() for ensuring data integrity is important for performance optimization..

```
arr = np.array([1, 2, 3])
arr_copy = arr.copy()
arr_copy[0] = 100
print(arr)  # Original array is unchanged
print(arr_copy)  # Copy is modified
```

## Advanced Random Sampling

Beyond simple random numbers, NumPy offers functions like np.random.choice() for sampling with or without replacement, and generating random numbers from specific distributions.

```
arr = np.array([1, 2, 3, 4, 5])
sampled = np.random.choice(arr, size=3, replace=True)
print(sampled)
```

## Performance Comparison: Loop vs. Vectorization

Here, we will calculate the sum of numbers from 0 upto 10 million using %timeit for benchmarking for showing performance differences. Let’s compare the time required to execute a vectorized operation versus an equivalent loop-based operation.

```
import numpy as np
import time

N = 10_000_000
start_time = time.time()
vectorized_sum = np.sum(np.arange(N ))
print("Vectorized sum:", vectorized_sum)
print("Time taken by vectorized sum:", time.time() - start_time)

start_time = time.time()
iterative_sum = sum(range(N ))
print("\nIterative sum:", iterative_sum)
print("Time taken by iterative sum:", time.time() - start_time)
```

Exercise:-

1) Generate a 1000-element array from a normal distribution with mean=10, std=3. Calculate the mean, median, std, and check how many values are above 13.

2) Create a Function for the below equation and find the values when x=10 and y= 9

$$\boxed{f(x,y) = 3x^2 + \sqrt{x^2 + y^2} + e^{\ln(x)}}$$

Exercise:-

1) Generate a 1000-element array from a normal distribution with mean=10, std=3. Calculate the mean, median, std, and check how many values are above 13.

2) Create a Function for the below equation and find the values when x=10 and y= 9

$$f(x, y) = 3x^2 + \sqrt{x^2 + y^2} + e^{ln(x)}$$

```
import numpy as np
def func_ex(x,y):
    f = 3*np.power(x,2) + np.sqrt(np.power(x,2) +
    np.power(y,2)) + np.exp(np.log(x))
    return f

func_ex(10,9)
```