lecture 3

Introduction to Python Libraries

(NumPy, Pandas, Seaborn, Matplotlib)

Eng :Shereen saad

Introduction:

NumPy – The foundation for numerical computations and arrays.

Pandas – For efficient data handling and manipulation.

Matplotlib – For creating simple and customizable visualizations.

Seaborn – For advanced and statistical data visualization.

Python Programming

NumPy Arrays

<u>Eng :Shereen saad</u>

NumPy

- NumPy, short for Numerical Python, is one of the most important foundational packages for numerical computing in Python.

Why NumPy

- An efficient multidimensional array providing fast array-oriented arithmetic operations.
- NumPy-based algorithms are generally 10 to 100 times faster (or more) than their pure Python counterparts and use less memory.
- NumPy forms the basis of many powerful libraries.
- Mathematical functions for fast operations on entire arrays of data without having to write loops.
- NumPy provides linear algebra, random number generation, and Fourier transform capabilities

Why NumPy

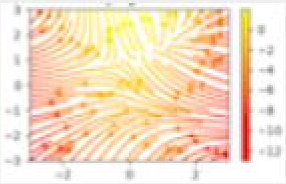
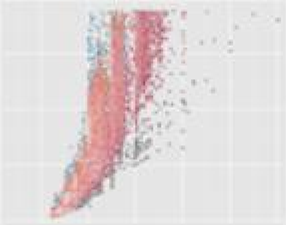
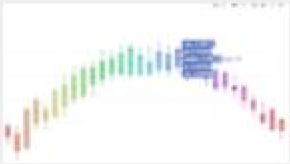
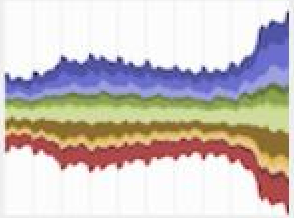
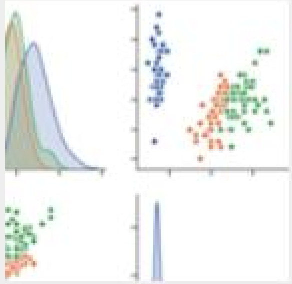
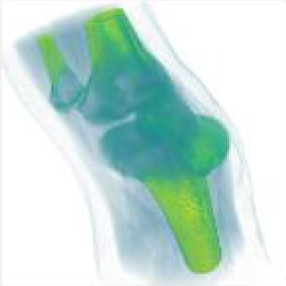
- One of the reasons NumPy is so important for computations in Python is because it is designed for efficiency on large arrays of data.
  - NumPy stores data in a contiguous block of memory.
  - NumPy is faster than regular Python code because its C-based algorithms.
  - NumPy arrays also use much less memory than built-in Python sequences.
  - NumPy operations perform complex computations on entire arrays without the need for Python for loops.

NumPy: Data Analysis

- Fast array-based operations for data cleaning, filtering, transformation, and any other kind of computation.
- Common array algorithms like sorting, unique, and set operations.
- Efficient descriptive statistics and aggregating/summarizing data.
- Expressing conditional logic as array expressions instead of loops with if-else branches.
- Group-wise data manipulations
  aggregation, transformation, and function application
- Relational data manipulations for merging heterogeneous datasets.

NumPy: Visualization

- NumPy is an essential component in the Python visualization landscape, which includes Matplotlib, Seaborn, Plotly, Altair, Bokeh, Holoviz, Vispy, Napari, and PyVista, to name a few.

NumPy: Machine Learning

- NumPy forms the basis of powerful machine learning and deep learning libraries like scikit-learn, SciPy, TensorFlow, Keras, PyTorch, and MXNet.

Uses of NumPy

01 Arithmetic Operation

02 Statistical Operation

03 Bitwise Operators

04 Copying & Viewing Arrays

05 Stacking

06 Matrix Operations

07 Linear Algebra

08 Broadcasting

09 Mathematical Operations

10 Searching, Sorting & Counting

How to import NumPy

- To access NumPy and its functions import it in your Python code

```python
import numpy as np
```

- We shorten the imported name to `np` for better readability of code using NumPy.
- This is a widely adopted convention that you should follow so that anyone working with your code can easily understand it.

Creating Arrays from Existing Data

- The numpy module provides various functions for creating arrays.
- Here we use the array function, which receives as an argument a collection of elements and returns a new array containing the elements.

```
numbers = np.array([2, 3, 5, 7, 11])

print(numbers)
print(type(numbers))

# Output
[2  3  5  7 11]
<class 'numpy.ndarray
```

array Attributes: Determining Element Type

- The `array` function determines an array's element type from its argument's elements.
- You can check the element type with an array's `dtype` attribute.

```
integers = np.array([1, 2, 4, 8, 16, 32, 64])

print(integers)
print(integers.dtype)

# Output
[1  2  4  8 16 32 64]
int32
```

array Attributes: Determining Element Type

- The `array` function **determines an array's element type** from its argument's elements.
- You can **check the element type** with an array's `dtype` attribute.

```
floats = np.array([10.5, 11, 7.25, 4.74, 10])

print(floats)
print(floats.dtype)

# Output
[10.5  11. 7.25  4.74  10.]
float64
```

array Attributes: Determining Number of Elements and Element Size

- You can view an array's total number of elements with the attribute `size` and the number of bytes required to store each element with `itemsize`.

```
integers = np.array([1, 2, 4, 8, 16, 32, 64])

print(integers.size)          # Output: 7
```

- The attribute `ndim` contains an array's number of dimensions and the attribute `shape` contains a tuple specifying an array's dimensions.

```
integers = np.array([1, 2, 4, 8, 16, 32, 64])

print(integers.ndim)          # Output: 1

print(integers.shape)         # Output: (7,)
```

Difference Between 1D, 2D, and 3D Arrays in NumPy

- **1D Array (One-Dimensional Array)**
- A single row of elements, similar to a simple list.
- Shape: (n,) → n is the number of elements.
- Dimensions (ndim): 1
- General structure: [element1, element2, element3, ...]
- **Example:**
- import numpy as np
- arr_1d = np.array([1, 2, 3, 4, 5])
- print(arr_1d.ndim)   # Output: 1
- print(arr_1d.shape)   # Output: (5,)

Difference Between 1D, 2D, and 3D Arrays in NumPy

- 2D Array (Two-Dimensional Array)
- A table-like structure with rows and columns.
- Shape: (rows, columns) → rows is the number of rows, and columns is the number of columns.
- Dimensions (ndim): 2
- General structure: [[row1], [row2], [row3], ...]
- **Example:**

```
arr_2d = np.array([[1, 2, 3],
                   [4, 5, 6]])
print(arr_2d.ndim)   # Output: 2
print(arr_2d.shape)  # Output: (2, 3)
```

Difference Between 1D, 2D, and 3D Arrays in NumPy

3D Array (Three-Dimensional Array) : A collection of multiple 2D arrays, like stacked matrices or RGB images.

- Shape:(depth, rows, columns) → depth represents the number of layers. Dimensions (ndim):3 General structure:[[[matrix1]], [[matrix2]], ...]

- **Example:**

```
arr_3d = np.array([[[1, 2, 3], [4, 5, 6]],
           [[7, 8, 9], [10, 11, 12]]])
print(arr_3d.ndim)   # Output: 3
print(arr_3d.shape)  # Output: (2, 2, 3)
```

- **This array has 2 layers, 2 rows, and 3 columns, so shape = (2, 2, 3) and ndim = 3.**

array Attributes: Determining Dimensions

- The attribute `ndim` contains an array's number of dimensions and the attribute `shape` contains a tuple specifying an array's dimensions.

```
integers = np.array([1, 2, 4, 8, 16, 32, 64])

print(integers.ndim)          # Output: 1

print(integers.shape)         # Output: (7,)
```

array Attributes: Determining Dimensions

- Let's create an array from a 2 × 3 list:

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

print(mat.ndim)                 # Output: 2

print(mat.shape)                # Output: (2, 3)

print(mat.shape[0])             # Output: 2

print(mat.shape[1])             # Output: 3
```

Creating arrays from Ranges

- NumPy provides optimized functions for creating arrays from ranges.

```python
arr = np.arange(5)
print(arr)
# Output: [0 1 2 3 4]

arr = np.arange(5, 10)
print(arr)
# Output: [5 6 7 8 9]

arr = np.arange(2, 10, 2)
print(arr)
# Output: [2 4 6 8]
```

Creating Floating-Point Ranges with `linspace`

- NumPy provides optimized functions for creating arrays from ranges.

```
arr = np.linspace(0, 100, 5)
print(arr)
# Output: [0.   25.   50.   75. 100.]

arr = np.linspace(10, 45, 8)
print(arr)
# Output: [10. 15. 20. 25. 30. 35. 40. 45.]

arr = np.linspace(2, 10, 6)
print(arr)
# Output: [2.   3.6  5.2  6.8  8.4  10.]
```

Reshaping an array

- You also can create an array from a range of elements, then use array method `reshape` to transform the one-dimensional array into a multidimensional array.

```
arr = np.arange(1, 21).reshape((4, 5))
print(arr)

# Output:
[[ 1  2  3  4  5]
 [ 6  7  8  9 10]
 [11 12 13 14 15]
 [16 17 18 19 20]]
```

Filling arrays with Specific Values

- NumPy provides functions `zeros`, `ones` and `full` for creating arrays containing 0s, 1s or a specified value, respectively.

```
arr = np.zeros(5)
print(arr)
# Output: [0. 0. 0. 0. 0.]

arr = np.zeros((3, 5))
print(arr)
# Output:
[[0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]]
```

Filling arrays with Specific Values

- NumPy provides functions `zeros`, `ones` and `full` for creating arrays containing 0s, 1s or a specified value, respectively.

```
arr = np.ones(5)
print(arr)
# Output: [1. 1. 1. 1. 1.]

arr = np.ones((3, 5))
print(arr)
# Output:
[[1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]]
```

Filling arrays with Specific Values

- NumPy provides functions `zeros`, `ones` and `full` for creating arrays containing 0s, 1s or a specified value, respectively.

```
arr = np.full(5, 7)
print(arr)
# Output: [7 7 7 7 7]

arr = np.full((3, 5), 7)
print(arr)
# Output:
[[7 7 7 7 7]
 [7 7 7 7 7]
 [7 7 7 7 7]]
```

Filling arrays with Specific Values

- NumPy provide the function `eye` that creates a square N × N identity matrix (1s on the diagonal and 0s elsewhere).

```
arr = np.eye(5)
print(arr)

# Output:
[[1. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0.]
 [0. 0. 1. 0. 0.]
 [0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 1.]]
```

Filling arrays with Specific Values

- NumPy provide the function `eye` that creates a square N × N identity matrix (1s on the diagonal and 0s elsewhere).

```
arr = np.eye(3, 5)
print(arr)

# Output:
[[1. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0.]
 [0. 0. 1. 0. 0.]]
```

Some Important NumPy Array Creation Functions

Function	Description
`array`	Convert input data (list, tuple, array, or other sequence type) to an ndarray either by inferring a data type or explicitly specifying a data type; copies the input data by default
`asarray`	Convert input to ndarray, but do not copy if the input is already an ndarray
`arange`	Like the built-in `range` but returns an ndarray instead of a list
`ones`, `ones_like`	Produce an array of all 1s with the given shape and data type; `ones_like` takes another array and produces a `ones` array of the same shape and data type
`zeros`, `zeros_like`	Like `ones` and `ones_like` but producing arrays of 0s instead
`empty`, `empty_like`	Create new arrays by allocating new memory, but do not populate with any values like `ones` and `zeros`
`full`, `full_like`	Produce an array of the given shape and data type with all values set to the indicated "fill value"; `full_like` takes another array and produces a filled array of the same shape and data type
`eye`, `identity`	Create a square N × N identity matrix (1s on the diagonal and 0s elsewhere)

Arithmetic Operations with Arrays

- Let's perform element-wise arithmetic with arrays and numeric values by using arithmetic operators .
- The element-wise operations are applied to every element in the array.

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

arr2 = arr + 10
print(arr2)
# Output: [11 12 13 14 15]

arr3 = arr - 4
print(arr3)
# Output: [-3 -2 -1  0  1]
```

Arithmetic Operations with Arrays

- Let's perform element-wise arithmetic with arrays and numeric values by using arithmetic operators .
- The element-wise operations are applied to every element in the array.

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

arr4 = arr * 2
print(arr4)
# Output: [2  4  6  8 10]

arr5 = arr / 2
print(arr5)
# Output: [0.5  1.  1.5  2.  2.5]
```

Arithmetic Operations with Arrays

- Let's perform element-wise arithmetic with arrays and numeric values by using arithmetic operators .
- The element-wise operations are applied to every element in the array.

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

arr6 = arr ** 2
print(arr6)
# Output: [1  4  9 16 25]

arr7 = 2 ** arr
print(arr7)
# Output: [2  4  8 16 32]
```

Arithmetic Operations with Arrays

- Let’s perform element-wise arithmetic with arrays and numeric values by using arithmetic operators .
- The element-wise operations are applied to every element in the array.

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

arr += 2
print(arr)
# Output: [3 4 5 6 7]

arr *= 2
print(arr)
# Output: [6 8 10 12 14]
```

Arithmetic Operations Between Arrays

- You may perform arithmetic operations and augmented assignments between arrays of the same shape.

```
arr1 = np.array([10, 20, 30, 40, 50])
arr2 = np.array([1, 2, 3, 4, 5])

arr = arr1 + arr2
print(arr)

# Output:
[11 22 33 44 55]
```

Arithmetic Operations Between Arrays

- You may perform arithmetic operations and augmented assignments between arrays of the same shape.

```
arr1 = np.array([10, 20, 30, 40, 50])
arr2 = np.array([1, 2, 3, 4, 5])

arr = arr1 - arr2
print(arr)

# Output:
[9 18 27 36 45]
```

Arithmetic Operations Between Arrays

- You may perform arithmetic operations and augmented assignments between arrays of the same shape.

```
arr1 = np.array([10, 20, 30, 40, 50])
arr2 = np.array([1, 2, 3, 4, 5])

arr = arr1 * arr2
print(arr)

# Output:
[10  40  90  160  250]
```

Arithmetic Operations Between Arrays

- You may perform arithmetic operations and augmented assignments between arrays of the same shape.

```
arr1 = np.array([10, 20, 30, 40, 50])
arr2 = np.array([1, 2, 3, 4, 5])

arr = arr1 / arr2
print(arr)

# Output:
[10. 10. 10. 10. 10.]
```

Comparing Arrays

- You can compare arrays with individual values and with other arrays. Comparisons are performed element-wise.

```
arr1 = np.array([5, 7, 4, 3, 5])
arr2 = np.array([1, 2, 4, 8, 16])

arr = arr1 >= arr2
print(arr)

# Output:
[True  True  True  False  False]
```

Comparing Arrays

- You can compare arrays with individual values and with other arrays. Comparisons are performed element-wise.

```
arr1 = np.array([5, 7, 4, 3, 5])
arr2 = np.array([1, 2, 4, 8, 16])

arr = arr1 == arr2
print(arr)

# Output:
[False  False  True  False  False]
```

Comparing Arrays

- You can compare arrays with individual values and with other arrays. Comparisons are performed element-wise.

```
arr1 = np.array([5, 7, 4, 3, 5])
arr2 = np.array([1, 2, 4, 8, 16])

arr = arr1 != arr2
print(arr)

# Output:
[True  True  False  True  True]
```

Calculation Methods

- We can use methods to calculate `sum`, `min`, `max`, `mean`, `std` (standard deviation) and `var` (variance).

```
arr = np.array([4, 7, 2, 10, 22, 15])

print(arr.max())        # Output: 22
print(arr.min())        # Output: 2
print(arr.sum())        # Output: 60
print(arr.mean())       # Output: 10.0
print(arr.var())        # Output: 46.333333333333336
print(arr.std())        # Output: 6.806859285540455
```

Universal Functions

- NumPy offers dozens of standalone universal functions that perform various element-wise operations.

```
arr1 = np.array([10, 20, 30, 40, 50])
arr2 = np.array([1, 2, 3, 4, 5])

arr = arr1 + arr2
print(arr)
# Output: [11 22 33 44 55]

arr = np.add(arr1, arr2)
print(arr)
# Output: [11 22 33 44 55]
```

Universal Functions

- NumPy offers dozens of standalone universal functions that perform various element-wise operations.

```
arr1 = np.array([10, 20, 30, 40, 50])
arr2 = np.array([1, 2, 3, 4, 5])

arr = arr1 - arr2
print(arr)
# Output: [9 18 27 36 45]

arr = np.subtract(arr1, arr2)
print(arr)
# Output: [9 18 27 36 45]
```

Universal Functions

- NumPy offers dozens of standalone universal functions that perform various element-wise operations.

```
arr1 = np.array([10, 20, 30, 40, 50])
arr2 = np.array([1, 2, 3, 4, 5])

arr = arr1 * arr2
print(arr)
# Output: [10  40  90  160  250]

arr = np.multiply(arr1, arr2)
print(arr)
# Output: [10  40  90  160  250]
```

Universal Functions

- NumPy offers dozens of standalone universal functions that perform various element-wise operations.

```
arr1 = np.array([10, 20, 30, 40, 50])
arr2 = np.array([1, 2, 3, 4, 5])

arr = arr1 / arr2
print(arr)
# Output: [10. 10. 10. 10. 10.]

arr = np.divide(arr1, arr2)
print(arr)
# Output: [10. 10. 10. 10. 10.]
```

Universal Functions

- NumPy offers dozens of standalone universal functions that perform various element-wise operations.

```
print(np.sqrt(25))              # Output: 5.0
print(np.exp(1))                # Output: 2.718281828459045
print(np.log2(128))             # Output: 7.0
print(np.log10(1000))           # Output: 3.0
print(np.sin(np.pi/2))          # Output: 1.0
print(np.power(2, 10))          # Output: 1024
print(np.mod(22, 4))            # Output: 2
```

Universal Functions

Function	Description
`abs, fabs`	Compute the absolute value element-wise for integer, floating-point, or complex values
`sqrt`	Compute the square root of each element (equivalent to `arr ** 0.5`)
`square`	Compute the square of each element (equivalent to `arr ** 2`)
`exp`	Compute the exponent e^x of each element
`log, log10, log2, log1p`	Natural logarithm (base *e*), log base 10, log base 2, and log(1 + x), respectively
`sign`	Compute the sign of each element: 1 (positive), 0 (zero), or –1 (negative)
`ceil`	Compute the ceiling of each element (i.e., the smallest integer greater than or equal to that number)
`floor`	Compute the floor of each element (i.e., the largest integer less than or equal to each element)
`rint`	Round elements to the nearest integer, preserving the `dtype`
`modf`	Return fractional and integral parts of array as separate arrays
`isnan`	Return Boolean array indicating whether each value is `NaN` (Not a Number)
`isfinite, isinf`	Return Boolean array indicating whether each element is finite (non-`inf`, non-`NaN`) or infinite, respectively
`cos, cosh, sin, sinh, tan, tanh`	Regular and hyperbolic trigonometric functions
`arccos, arccosh, arcsin, arcsinh, arctan, arctanh`	Inverse trigonometric functions
`logical_not`	Compute truth value of `not x` element-wise (equivalent to `~arr`)

Universal Functions

Function	Description
add	Add corresponding elements in arrays
subtract	Subtract elements in second array from first array
multiply	Multiply array elements
divide, floor_divide	Divide or floor divide (truncating the remainder)
power	Raise elements in first array to powers indicated in second array
maximum, fmax	Element-wise maximum; fmax ignores NaN
minimum, fmin	Element-wise minimum; fmin ignores NaN
mod	Element-wise modulus (remainder of division)
copysign	Copy sign of values in second argument to values in first argument
greater, greater_equal, less, less_equal, equal, not_equal	Perform element-wise comparison, yielding Boolean array (equivalent to infix operators >, >=, <, <=, ==, !=)
logical_and	Compute element-wise truth value of AND (&) logical operation
logical_or	Compute element-wise truth value of OR (|) logical operation
logical_xor	Compute element-wise truth value of XOR (^) logical operation

Indexing and Slicing

- One-dimensional arrays can be indexed and sliced using the same syntax and of lists and tuples.

```
arr = np.array([1, 2, 4, 8, 16, 32, 64])

print(arr[0])           # Output: 1
print(arr[4])           # Output: 16
print(arr[-1])          # Output: 64
print(arr[-2])          # Output: 32

print(arr[:3])          # Output: [1 2 4]
print(arr[3:])          # Output: [8 16 32 64]
print(arr[2:6])         # Output: [4  8 16 32]
```

Indexing and Slicing

- One-dimensional arrays can be indexed and sliced using the same syntax and of lists and tuples.

```
arr = np.array([1, 2, 4, 8, 16, 32, 64])
print(arr[1:6:2])      # Output: [2  8 32]
print(arr[::3])        # Output: [1  8 64]
print(arr[2::2])       # Output: [4 16 64]
```

Indexing and Slicing

- To select an element in a two-dimensional array, specify a tuple containing the element's row and column indices in square brackets.

```
arr = np.array([[87, 96, 70],
                [100, 87, 90],
                [94, 77, 95],
                [100, 81, 82]])

print(arr[0, 0])        # Output: 87
print(arr[1, 2])        # Output: 90
print(arr[2, 1])        # Output: 77
print(arr[2, 2])        # Output: 95
print(arr[3, 1])        # Output: 81
```

Indexing and Slicing

- To select a single row, specify only one index in square brackets.

```
arr = np.array([[87, 96, 70],
               [100, 87, 90],
               [94, 77, 95],
               [100, 81, 82]])

print(arr[0])              # Output: [87 96 70]

print(arr[0:2])            # Output: [[ 87  96  70]
                                      [100  87  90]]
```

Indexing and Slicing

- You can select subsets of the columns by providing a tuple specifying the row(s) and column(s) to select.

```
arr = np.array([[87, 96, 70],
               [100, 87, 90],
               [94, 77, 95],
               [100, 81, 82]])

print(arr[:, 1])          # Output: [96 87 77 81]

print(arr[:, 1:3])        # Output: [[96 70]
                                     [87 90]
                                     [77 95]
                                     [81 82]]
```

Indexing and Slicing

- You can select subsets of columns and rows by specifying the row(s) and column(s) to select.

```
arr = np.array([[87, 96, 70],
                [100, 87, 90],
                [94, 77, 95],
                [100, 81, 82]])

print(arr[1:3, :2])

# Output:
[[100  87]
 [ 94  77]]
```

Linear Algebra Operations: Matrix Multiplication

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

y = np.array([[5, 6],
              [7, 8]])

print(x.dot(y))

# Output:
[[19 22]
 [43 50]]
```

Linear Algebra Operations: Matrix Multiplication

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

y = np.array([[5, 6],
              [7, 8]])

print(y.dot(x))

# Output:
[[23 34]
 [31 46]]
```

Linear Algebra Operations: Matrix Transpose

```
arr = np.arange(1, 11).reshape(2, 5)

print(arr)
# Output:
[[ 1  2  3  4  5]
 [ 6  7  8  9 10]]

print(arr.T)
# Output:
[[ 1  6]
 [ 2  7]
 [ 3  8]
 [ 4  9]
 [ 5 10]]
```

Linear Algebra Operations: Matrix Diagonal

```python
arr = np.array([[1, 2, 3],
                [4, 5, 6],
                [7, 8, 9]])


print(arr.diagonal())

# Output:
[1 5 9]
```

Linear Algebra Operations: Matrix Determinant

```
arr = np.array([[1, 0, 5],
                [2, 1, 6],
                [3, 4, 0]])

print(np.linalg.det(arr))

# Output:
0.9999999999999967
```

Linear Algebra Operations: Matrix Inverse

```
arr = np.array([[1, 0, 5],
                [2, 1, 6],
                [3, 4, 0]])

print(np.linalg.inv(arr))

# Output:
[[-24.  20.  -5.]
 [ 18. -15.   4.]
 [  5.  -4.   1.]]
```

Linear Algebra Operations: Commonly `numpy.linalg` functions

Function	Description
`diag`	Return the diagonal (or off-diagonal) elements of a square matrix as a 1D array, or convert a 1D array into a square matrix with zeros on the off-diagonal
`dot`	Matrix multiplication
`trace`	Compute the sum of the diagonal elements
`det`	Compute the matrix determinant
`eig`	Compute the eigenvalues and eigenvectors of a square matrix
`inv`	Compute the inverse of a square matrix
`pinv`	Compute the Moore-Penrose pseudoinverse of a matrix
`qr`	Compute the QR decomposition
`svd`	Compute the singular value decomposition (SVD)
`solve`	Solve the linear system $Ax = b$ for x, where A is a square matrix
`lstsq`	Compute the least-squares solution to `Ax = b`

Copying NumPy Arrays

- The views are also known as shallow copies.

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

print(arr1)        # Output: [1 2 3 4 5]
print(arr2)        # Output: [1 2 3 4 5]

arr1[0] = 10
print(arr1)        # Output: [10  2  3  4  5]
print(arr2)        # Output: [10  2  3  4  5]
```

Copying NumPy Arrays

- The array method `copy` returns a new array object with a deep copy of the original array object's data.

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

print(arr1)        # Output: [1 2 3 4 5]
print(arr2)        # Output: [1 2 3 4 5]

arr1[0] = 10
print(arr1)        # Output: [10  2  3  4  5]
print(arr2)        # Output: [1   2  3  4  5]
```

NumPy Data Types

Type	Type code	Description
int8, uint8	i1, u1	Signed and unsigned 8-bit (1 byte) integer types
int16, uint16	i2, u2	Signed and unsigned 16-bit integer types
int32, uint32	i4, u4	Signed and unsigned 32-bit integer types
int64, uint64	i8, u8	Signed and unsigned 64-bit integer types
float16	f2	Half-precision floating point
float32	f4 or f	Standard single-precision floating point; compatible with C float
float64	f8 or d	Standard double-precision floating point; compatible with C double and Python `float` object
float128	f16 or g	Extended-precision floating point
complex64, complex128, complex256	c8, c16, c32	Complex numbers represented by two 32, 64, or 128 floats, respectively
bool	?	Boolean type storing `True` and `False` values
object	O	Python object type; a value can be any Python object
string_	S	Fixed-length ASCII string type (1 byte per character); for example, to create a string data type with length 10, use `'S10'`
unicode_	U	Fixed-length Unicode type (number of bytes platform specific); same specification semantics as `string_` (e.g., `'U10'`)

NumPy Data Types

- NumPy supports a much greater variety of numerical types.

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

arr = np.array([1, 2, 3, 4, 5], dtype='float32')
print(arr)
# Output: [1. 2. 3. 4. 5.]

arr = np.array([1, 2, 3, 4, 5], dtype='str')
print(arr)
# Output: ['1' '2' '3' '4' '5']
```

Conversions Between Types

- To convert the type of an array, use the `astype()` method.

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

print(arr)             # Output: [1 2 3 4 5]
print(arr.dtype)       # Output: Int32


arr = arr.astype('float64')

print(arr)             # Output: [1. 2. 3. 4. 5.]
print(arr.dtype)       # Output: float64
```

Pseudorandom Number Generation

- The `numpy.random` module supplements the built-in Python random module with functions for efficiently generating whole arrays of sample values from many kinds of probability distributions.
- The `rand()` method returns a random float between 0 and 1.

```
x = np.random.rand()
print(x)
# Output: 0.37708279136694967

x = np.random.rand()
print(x)
# Output: 0.8653202160879492
```

Pseudorandom Number Generation

- The `rand()` method returns a random float between 0 and 1.

```
x = np.random.rand(4)
print(x)
# Output: [0.92648582 0.24813201 0.01548673 0.38614011]

x = np.random.rand(5, 3)
print(x)
# Output:
[[0.01629089 0.89310838 0.37853944]
 [0.10719307 0.64188753 0.56036435]
 [0.58155728 0.43390515 0.80745081]
 [0.88340205 0.06324677 0.78770121]
 [0.7838058  0.77233733 0.2268093 ]]
```

Pseudorandom Number Generation

- The method `randint()` returns random integers from low (inclusive) to high (exclusive).

```
# Generate a random integer from 1 to 100
x = np.random.randint(1, 101)
print(x)      # Output: 70

x = np.random.randint(1, 101)
print(x)      # Output: 26

x = np.random.randint(1, 101)
print(x)      # Output: 48
```

Pseudorandom Number Generation

- The method `randint()` returns random integers from low (inclusive) to high (exclusive).

```
# Generate a random integer from 10 to 15
x = np.random.randint(10, 16)
print(x)
# Output: 14

# Generate a random binary digit
x = np.random.randint(2)
print(x)
# Output: 1
```

Pseudorandom Number Generation

- Generate a 1D array containing 10 random integers from 0 to 99.

```
x = np.random.randint(0, 100, 10)
print(x)
# Output: [61 39 86 63 85 21 78 78  9  0]
```

- Generate a 1D array containing 8 random binary digits from 0 to 99:

```
x = np.random.randint(0, 2, 8)
print(x)
# Output: [0 1 0 0 0 0 1 1]
```

Pseudorandom Number Generation

- Generate a 2D array with 3 rows, each row containing 5 random integers from 1 to 100.

```
x = np.random.randint(1, 101, (3, 5))

print(x)
# Output:
[[72  6 67 64 24]
 [64 87 39 32 55]
 [38  9 68 90 99]]
```

Pseudorandom Number Generation

- The `choice()` method takes an array as a parameter and randomly returns one of the values.

```
x = np.random.choice([22, 11, 4, 7, 3, 5])
print(x)       # Output: 11

x = np.random.choice([22, 11, 4, 7, 3, 5])
print(x)       # Output: 7

x = np.random.choice([22, 11, 4, 7, 3, 5])
print(x)       # Output: 5
```

Pseudorandom Number Generation

- The `choice()` method takes an array as a parameter and randomly returns one of the values.

```
x = x = np.random.choice([22, 11, 4, 7, 3, 5], size=(3, 5))

print(x)
# Output:
[[ 7 11 22  7  7]
 [ 5  7  4 11  5]
 [ 3  3 11  5  4]]
```

Pseudorandom Number Generation

- The `permutation()` method returns a random permutation of a sequence.

```
x = np.random.permutation([22, 11, 4, 7, 3, 5])
print(x)
# Output
[3  7  5  22  4  11]

x = np.random.permutation([22, 11, 4, 7, 3, 5])
print(x)
# Output
[22  4  11  5  3  7]
```

Introduction to Pandas Library

- **What is Pandas?**

- **Pandas is an open-source Python library used for data analysis and manipulation.**
- **It provides flexible data structures such as Series and DataFrame for handling structured data.**
- **Built on top of NumPy, making it efficient for numerical computations.**

Why is Pandas Important?

- Simplifies working with structured data (CSV, Excel, SQL, JSON).
- Provides tools for data cleaning, transformation, and analysis.
- Essential for Data Science, Machine Learning, and AI applications.
- Enhances data visualization when combined with Matplotlib and Seaborn.

Key Features of Pandas

- **DataFrame & Series:** Efficient data storage structures.
- **Data Cleaning**: Handling missing values, duplicates, and inconsistent data.
- **Data Transformation:** Sorting, filtering, and grouping data.
- **Data Input/Output:** Read & write data from various formats (CSV, Excel, JSON).
- **Statistical Analysis**: Built-in functions for summarizing data.

Introduction to Matplotlib & Seaborn

- **Introduction to Matplotlib**

- **Matplotlib is a popular Python library for creating static, animated, and interactive visualizations.**

- **Provides a MATLAB-like interface.**

- **Highly customizable and supports multiple types of plots.**

Why Use Matplotlib?

- Simple and flexible
- Works well with NumPy and Pandas
- Can create a variety of charts (line, bar, scatter, histogram, etc.)
- Customizable styles and themes.

Basic Plot in Matplotlib

- import matplotlib.pyplot as plt
- x = [1, 2, 3, 4, 5]
- y = [10, 20, 25, 30, 50]
- plt.plot(x, y, linestyle="-", color="b")
- plt.xlabel("X-axis")
- plt.ylabel("Y-axis")
- plt.title("Basic Line Plot")
- plt.show()

Common Matplotlib Plots

- Line Plot
- Bar Chart
- Scatter Plot
- Histogram
- Pie Chart

Introduction to Seaborn

- Seaborn is a data visualization library built on top of Matplotlib.
- Provides a high-level interface for statistical graphics.
- Works well with Pandas DataFrames.
- Includes beautiful default themes.

Basic Plot in Seaborn

- import seaborn as sns
- import matplotlib.pyplot as plt
- tips = sns.load_dataset("tips")
- sns.scatterplot(x="total_bill", y="tip", hue="sex", data=tips)
- plt.title("Total Bill vs Tip")
- plt.show()

Common Seaborn Plots

- **Scatter Plot**
- **Line Plot**
- **Bar Chart**
- **Box Plot**
- **Heatmap**

difference between Matplotlib and Seaborn

- Matplotlib provides a limited set of default styles and color palettes, requiring users to customize their plots manually to achieve a desired look.

- Seaborn, on the other hand, offers a range of default styles and color palettes that are optimized for different types of data and visualizations. This makes it easy for users to create visually appealing plots with minimal customization.

Thank You