Chapter2 .NumPy Library

NumPy is a powerful Python library that is used for scientific computing, primarily working with arrays and matrices of numeric data. It provides support for large multidimensional arrays, and also includes a vast collection of mathematical functions to operate on these arrays efficiently.

Key Features of NumPy:

1. **Array Creation**: NumPy provides the ndarray, an efficient multi-dimensional array object, which is the core of the library.
2. **Mathematical Functions**: Supports many mathematical operations like basic algebra, trigonometric operations, statistics, and linear algebra.
3. **Broadcasting**: Allows arithmetic operations on arrays of different shapes, enabling efficient computation.
4. **Random Sampling**: Includes a suite of random number generation functions.
5. **Performance**: Written in C and optimized for performance, making it much faster for array-based operations compared to Python's native lists.

Installation

Before diving into examples, if you haven't installed NumPy, you can do so with:

pip install numpy

2. Creating Arrays انشاء مصفوفة متعددة الابعاد

NumPy arrays are at the heart of the library. They are faster and more efficient than Python lists for numerical operations.

Example:
python
Copier le code

```
import numpy as np

# Creating a 1D array
arr_1d = np.array([1, 2, 3, 4])
print("1D Array:", arr_1d)

# Creating a 2D array (matrix)
arr_2d = np.array([[1, 2, 3], [4, 5, 6]])
print("2D Array:\n", arr_2d)
```

Output Explanation:

```
1D Array: [1 2 3 4]
2D Array:
 [[1 2 3]
  [4 5 6]]
```

- The 1D array is a simple sequence of numbers.

- The 2D array (or matrix) is like a table with rows and columns.

3. Array Attributes خصائص

NumPy arrays come with several useful attributes to help understand the structure of the data.

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

print("Shape:", arr.shape)    # The shape of the array (rows, columns)
print("Size:", arr.size)      # The total number of elements
print("Data Type:", arr.dtype) # The type of data in the array
```

Output Explanation:

```
Shape: (2, 3)
Size: 6
Data Type: int64
```

- **Shape** tells us the dimensions of the array (2 rows, 3 columns).
- **Size** gives the total number of elements.
- **Data Type (dtype)** indicates the type of data stored in the array (integers in this case).

4. Array Operations

NumPy supports element-wise operations, meaning that you can perform operations on arrays directly, and NumPy will apply them to each element.

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

# Element-wise addition
print("Addition:", arr1 + arr2)

# Element-wise multiplication
print("Multiplication:", arr1 * arr2)

# Scalar multiplication
print("Scalar Multiplication:", arr1 * 2)
```

Output Explanation:

```
Addition: [5 7 9]
Multiplication: [ 4 10 18]
Scalar Multiplication: [2 4 6]
```

- **Addition:** Each element of arr1 is added to the corresponding element of arr2.
- **Multiplication:** Each element of arr1 is multiplied by the corresponding element of arr2.
- **Scalar Multiplication:** Each element of arr1 is multiplied by the scalar value 2.

5. Indexing and Slicing

Like Python lists, you can access specific elements of a NumPy array using indexing, and you can retrieve sub-arrays through slicing.

Example:

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

# Accessing a single element
print("Element at index 2:", arr[2])

# Slicing (extracting elements from index 1 to 3)
print("Slice from index 1 to 4:", arr[1:4])

# 2D Array indexing
arr_2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print("Element at [1,2]:", arr_2d[1, 2])
```

Output Explanation:

```
Element at index 2: 30
Slice from index 1 to 4: [20 30 40]
Element at [1,2]: 6
```

- **Indexing** allows access to specific elements (e.g., arr[2] retrieves the element at index 2).
- **Slicing** extracts a portion of the array. In this case, it takes elements from index 1 to 3 (excluding 4).
- **2D Indexing** retrieves an element from the specified row and column (row 1, column 2 gives 6).

6. Reshaping Arrays

You can change the shape of an array using the reshape() function. This does not alter the data but changes its structure.

Example:

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

# Reshaping into a 2x3 matrix
reshaped = arr.reshape(2, 3)
print("Reshaped Array:\n", reshaped)
```

Output Explanation:

```
Reshaped Array:
 [[1 2 3]
  [4 5 6]]
```

- The array is reshaped from a 1D array of length 6 into a 2D array with 2 rows and 3 columns.

7. Broadcasting

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

Example:
```
arr = np.array([[1, 2, 3], [4, 5, 6]])
scalar = 10

# Adding a scalar to a 2D array
print("Array + Scalar:\n", arr + scalar)
```

Output Explanation:

```
Array + Scalar:
 [[11 12 13]
  [14 15 16]]
```

- The scalar 10 is added to every element in the 2D array, demonstrating broadcasting.

8. Mathematical Operations

NumPy includes various mathematical functions, such as sum, mean, and standard deviation, that work efficiently with arrays.

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

# Sum of elements
print("Sum:", arr.sum())

# Mean of elements
print("Mean:", arr.mean())

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

Output Explanation:

```
Sum: 15
Mean: 3.0
Standard Deviation: 1.4142135623730951
```

- **Sum:** Adds up all the elements in the array.
- **Mean:** Calculates the average of the array's elements.
- **Standard Deviation:** Measures the spread of the array's values from the mean.

9. Matrix Multiplication **المصفوفات ثنائية الابعاد**

NumPy also supports matrix operations such as dot products and matrix multiplication.

Example:
python

```
matrix_1 = np.array([[1, 2], [3, 4]])
matrix_2 = np.array([[5, 6], [7, 8]])

# Matrix multiplication (dot product)
result = np.dot(matrix_1, matrix_2)
print("Matrix Multiplication:\n", result)
```

Output Explanation:

```
Matrix Multiplication:
 [[19 22]
  [43 50]]
```

- This performs the matrix multiplication of matrix_1 and matrix_2.

10. Random Numbers

NumPy provides functionality to generate random numbers using numpy.random.

Example:

```
# Random array of 5 values
random_arr = np.random.rand(5)
print("Random Array:", random_arr)

# Random integers between 1 and 10
random_int_arr = np.random.randint(1, 10, (2, 3))
print("Random Integer Array:\n", random_int_arr)
```

Output Explanation:

```
Random Array: [0.31524553 0.9314619  0.09245596 0.98733145 0.4894387 ]
Random Integer Array:
 [[4 7 8]
  [5 3 6]]
```

- **rand()** generates random float numbers between 0 and 1.
- **randint()** generates random integers in a specified range and shape.

What Is an Array?

An **array** is a data structure that stores a collection of elements **of the same type**.
It can have **one dimension** or **multiple dimensions**.

✓ Types of arrays:

- **1D array** (list):
  Example: [1, 2, 3, 4]
- **2D array** (table):
  Example:
- [[1, 2, 3],

- [4, 5, 6]]
- **n-dimensional array** (3D, 4D, etc.)

➤ **Use of arrays:**

Arrays store data for general purposes:

- names
- scores
- temperature readings
- images (as multidimensional arrays)

In Python, **NumPy arrays** are powerful for numeric work.

✅ **2. What Is a Matrix?**

A **matrix** is a **special type of 2-dimensional array** used specifically in **mathematics**, especially:

- Linear algebra
- Machine learning
- Scientific computing

A matrix has:

- **rows and columns**
- Only **2 dimensions**

Example of a matrix:

[[1, 2],

[3, 4]]

➤ **Use of matrices:**

Matrices support mathematical operations like:

- matrix multiplication
- determinants
- inverse
- eigenvalues

These operations are defined in linear algebra.

Summary (Easy Comparison)

Feature	**Array**	**Matrix**
Dimensions	1D, 2D, 3D, …	Always 2D

Feature	Array	Matrix
Purpose	General data storage	Mathematical computations
Operations	Element-wise	Linear algebra operations
Flexibility	More flexible	More specialized

Simple Example in Python (NumPy)

```
import numpy as np


# Array (1D)
arr = np.array([1, 2, 3])


# Matrix (2D)
mat = np.matrix([[1, 2],
                 [3, 4]])
```

Final Explanation

- All **matrices** are **2D arrays**.
- But **not all arrays** are matrices because arrays can have **any number of dimensions**.
- Arrays are general-purpose.
- Matrices are specifically for **linear algebra and math operations**.

Exercises

BEGINNER

1. Create a 1D array and print its elements

Exercise:
Create a 1D NumPy array: [10, 20, 30, 40] and print each element using a loop.

Solution:

```
import numpy as np

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

for x in arr:

    print(x)
```

2. Create a 2D array (matrix) and print its shape

Exercise:
Create this matrix and print its shape:

[[1, 2, 3],

[4, 5, 6]]

Solution:

```
mat = np.array([[1, 2, 3],

                [4, 5, 6]])

print(mat.shape)
```

Output: (2, 3)

3. Check if an object is an array or a matrix

Exercise:
Create:

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

m = np.matrix([[1, 2], [3, 4]])
```

Print their types.

Solution:

```
print(type(a))   # numpy.ndarray

print(type(m))   # numpy.matrix
```

INTERMEDIATE EXERCISES

4. Add two arrays element-wise

Exercise:
Compute the sum of these arrays:

a = [1, 2, 3]

b = [4, 5, 6]

Solution:

a = np.array([1,2,3])

b = np.array([4,5,6])

c = a + b

print(c)

5. Multiply two matrices (matrix multiplication)

Exercise:
Multiply:

A = [[1, 2],

[3, 4]]

B = [[5, 6],

[7, 8]]

Solution:

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

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

C = A @ B # or np.dot(A, B)

print(C)

Output:

[[19 22]

[43 50]]

6. Find the transpose of a matrix

Exercise:
Find the transpose of:

[[1, 4],

[2, 5],

[3, 6]]

Solution:

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

print(mat.T)

✅ ADVANCED EXERCISES

7. Compute the determinant of a matrix

Exercise:
Compute the determinant of:

[[2, 5],

  [1, 3]]

Solution:

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

det = np.linalg.det(mat)

print(det)
```

Output: 1.0

8. Find the inverse of a matrix

Exercise:
Find the inverse of:

[[4, 7],

  [2, 6]]

Solution:

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

inv = np.linalg.inv(mat)

print(inv)
```

9. Solve a system of equations using matrices

Solve the system:

2x + 3y = 13

4x + y = 11

Solution:

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


b = np.array([13,11])

solution = np.linalg.solve(A, b)

print(solution)  # [x, y]
```

10. Create a 3D array and access a specific slice

Exercise:
Given:

arr = np.arange(1, 25).reshape(2, 3, 4)

Print the 2nd block (index 1).

Solution:

print(arr[1])