NumPy Array Operations and Functions

Basic Array Operations: NumPy provides a variety of operations for manipulating arrays efficiently.

The basic operations which can be performed on arrays are:

1. Slicing an Array
2. Iterating over the elements
3. Modifying Array Values

Let us learn about them.

1. **Slicing an Array:** Array slicing is used to extract a portion of an array. Slicing is a powerful and efficient way to manipulate arrays in Python.

   Slicing can be done in two ways:

   a. Using slice() function
   b. Using indices

   a. Using slice() function: The **slice()** function takes three parameters: **start**, **stop**, and **step**, and returns a slice object that represents the set of indices specified by these parameters.

   Example Code:

```
import numpy as np
a = np.arange(10)
s = slice(2,10,2)
print(a[s])

Output
[2 4 6 8]
```

   The code snippet creates an array named a, with values ranging from 0 to 9. It then defines a slice object s with a start index of 2, stop index of 10, and step size of 2.

   b. Using Indices: Array can be sliced using parameters separated by a colon (start:stop:step).

Example Code 1:

```
import numpy as np
a = np.arange(10)
b = a[2:10:2]
print(b)

Output
[2 4 6 8]
```

The output is an array containing elements from index 2 to 10 (exclusive), with a step size of 2.

Example Code 2:

```
# slice single item
import numpy as np

a = np.arange(30)
b = a[7]
print(b)

Output
7
```

The code slices single item at index 7 from a NumPy array created with values 0 to 29 using arange().

In this code snippet, a NumPy array a is created using np.arange(30), generating an array with values from 0 to 29. The variable b is assigned the value at index 7 of the array a. Since indexing starts from 0, this corresponds to the eighth element in the array. The output of the code will be the value at index 7 of the array a, which is 7. Therefore, running the code will print 7 to the console.

Example Code 3:

```
# slice items starting from index
import numpy as np
a = np.arange(7)
print(a[3:])

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

The code creates a NumPy array a with values 0 to 6. It then prints elements from index 3 onwards using print(a[3:]), resulting in the output [3, 4, 5, 6].

2. **Iterating over Numpy arrays:** The NumPy package includes an efficient multidimensional iterator object called numpy.nditer. It allows iteration over an array using Python's standard Iterator interface.

Example Code:

```
import numpy as np
# Create a 2x3 array using arange() and reshape()
arr = np.arange(0, 30, 5).reshape(2, 3)

print('Original array is:')
print(arr)
print('\n')

print('Modified array is:')
for x in np.nditer(arr):
    print(x)
```

Output

```
Original array is:
[[ 0  5 10]
 [15 20 25]]



Modified array is:
0
5
10
15
20
25
```

The code creates a NumPy array (**arr**) using the **arange** function. Array has values starting from 0, incrementing by 5, and stopping before 30. The resulting 1-dimensional array is then reshaped into a 2x3 matrix using the **reshape** method. **arr**. **nditer** method is used to iterate over each element and then print the whole array.

3. **Modifying Array Values**: The nditer object features an additional optional parameter known as op_flags. By default, it is set to read-only, yet it can be configured to read-write or write-only mode. This allows the modification of array elements through the iterator.

Example Code:

```
import numpy as np
arr = np.arange(0,60,5)
arr = arr.reshape(3,4)
print('Original array is:')
print(arr)
print('\n')

for x in np.nditer(arr, op_flags = ['readwrite']):
   x[...] = 2*x
print('Modified array is:')
print(arr)
```

Output

```
Original array is:
[[ 0  5 10 15]
 [20 25 30 35]
 [40 45 50 55]]


Modified array is:
[[  0  10  20  30]
 [ 40  50  60  70]
 [ 80  90 100 110]]
```

The code creates a 1-dimensional array, reshape it into a 3x4 matrix, and then iterates over the elements of the array using **np.nditer** to modify each element by multiplying it by 2.

However, when dealing with arrays of different shapes, the broadcasting method becomes essential. Let's delve deeper into understanding this powerful technique.

Broadcasting in Numpy: Broadcasting in NumPy allows for the manipulation of arrays with different shapes during arithmetic operations.

Broadcasting is possible under the following conditions:

1. An array with a smaller number of dimensions than the other is extended by adding '1' to its shape.
2. The size in each dimension of the output shape is the maximum of the input sizes in that dimension.
3. An input can be employed in calculations if its size in a specific dimension matches the output size or if its value is precisely 1.
4. If an input has a dimension size of 1, the initial data entry in that dimension is utilized for all calculations along that dimension.

A group of arrays is considered broadcastable if the rules outlined above yield a valid outcome, and one of the following conditions is met:

1. Arrays share identical shapes.
2. Arrays possess the same number of dimensions, and the length of each dimension is either a shared length or 1.
3. An array with fewer dimensions can have its shape extended by adding a dimension of length 1 to fulfill the stated condition.

Example Code:

```
import numpy as np

arr1 = np.array([2,3,4,5])
arr2 = np.array([10,20,30,40])
arr3= arr1 * arr2
print(arr3)

Output
[ 20  60 120 200]
```

Two arrays named arr1 and arr2 are created. Elements of both arrays are multiplied and the results are stored in arr3.

Example 2:

```
import numpy as np
arr1 =
np.array([[10.0,10.0,10.0],[20.0,20.0,20.0],[30.0,30.0,
30.0],[40.0,40.0,40.0]])

arr2 = np.array([1.0,2.0,3.0])

print('First array:')
print(arr1)
print('\n')

print('Second array:')
print(arr2)
print('\n')

print('First Array + Second Array')
print(a + b)
```

Output

```
First array:
[[10. 10. 10.]
 [20. 20. 20.]
 [30. 30. 30.]
 [40. 40. 40.]]


Second array:
[1. 2. 3.]


First Array + Second Array
[[11. 12. 13.]
 [21. 22. 23.]
 [31. 32. 33.]
 [41. 42. 43.]]
```

Two arrays named arr1 and arr2 are created. The array arr1 is a 2D array, and arr2 is a 1D array. Broadcasting is performed adding each element of arr2 to the corresponding elements in each row of arr1.

Broadcasting allows you to perform element-wise operations on arrays of different shapes and sizes by automatically expanding the smaller array to match the shape of the larger one. Let us learn broadcasting iteration.

Broadcasting Iteration: When two arrays are broadcastable, a unified nditer object can iterate over them simultaneously.

For example, if an array 'a' has dimensions 3x4, and another array 'b' has dimensions 1x4, an iterator of the following kind is employed, where array 'b' is broadcasted to the size of 'a'.

Example Code:

```
import numpy as np
arr = np.arange(0,60,5)
arr = arr.reshape(3,4)

print('First array is:')
print(arr)
print('\n')

print('Second array is:')
arr2 = np.array([1, 2, 3, 4], dtype = int)
print(arr2)
print('\n')

print('Modified array is:')
for x,y in np.nditer([arr1,arr2]):
   print("%d:%d" % (x,y)),
```

Output

```
First array is:
[[ 0  5 10 15]
 [20 25 30 35]
 [40 45 50 55]]

Second array is:
[1 2 3 4]

Modified array is:
0:1
```

```
5:2
10:3
15:4
20:1
25:2
30:3
35:4
40:1
45:2
50:3
55:4
```

A 3x4 matrix named **arr** with values ranging from 0 to 55 in steps of 5 is created. Another 1-dimensional array named **arr2** with values [1, 2, 3, 4] is also defined. The code iterates over corresponding elements of both arrays using **np.nditer** and prints them.

Now that you've become adept at iterating over arrays of both similar and varying sizes, let's delve into performing operations on their elements.

In NumPy, a comprehensive set of mathematical functions is available for performing various operations on arrays. These functions enable efficient and vectorized computation, making them essential for scientific computing and data analysis.

Mathematical Functions in NumPy:

Following mathematical functions can be applied on arrays:

1. Basic Mathematical Operation
2. Trigonometric Functions:
3. Exponential and Logarithmic Functions
4. Statistical Functions

1. **Basic Mathematical Operations**: NumPy allows you to perform basic arithmetic operations on arrays like addition, subtraction, multiplication, and division.

Example Code:

```
import numpy as np
# Creating NumPy arrays
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])

# Addition
result_add = np.add(a, b)  # or a + b
print('Result of addition:',result_add)

# Subtraction
result_subtract = np.subtract(a, b)  # or a - b
print('Result of subtraction:',result_subtract)

# Multiplication
result_multiply = np.multiply(a, b)  # or a * b
print('Result of multiplication:',result_multiply)

# Division
result_divide = np.divide(a, b)  # or a / b
print('Result after division:', result_divide)

Output
Result of addition: [5 7 9]
Result of subtraction: [-3 -3 -3]
Result of multiplication: [ 4 10 18]
Result after division: [0.25 0.4  0.5 ]
```

In the code, NumPy arrays a and b are created with values [1, 2, 3] and [4, 5, 6]. The results of element-wise operations are stored in result_add (addition), result_subtract (subtraction using np.subtract or -), result_multiply (multiplication using np.multiply or *), and result_divide (division using np.divide or /).

2. **Trigonometric Functions**: NumPy provides trigonometric functions for working with angles.

```
import numpy as np
angle = np.pi / 4  # 45 degrees in radians
```

```
# Sine function
sin_val = np.sin(angle)
print(sin_val)

# Cosine function
cos_val = np.cos(angle)
print(cos_val)

# Tangent function
tan_val = np.tan(angle)
print(tan_val)


Output
0.7071067811865476
0.7071067811865476
0.9999999999999999
```

The code calculates and prints the sine, cosine, and tangent values of a 45-degree angle (converted to radians).

3. **Exponential and Logarithmic Functions:**
   - np.exp(x) computes the element-wise exponential of an array x.
   - np.log(x) computes the element-wise natural logarithm of an array x.
   - np.log10(x) computes the element-wise logarithm to the base 10 of an array x.

```
import numpy as np
# Exponential function (e^x)
exp_val = np.exp(2)
print(exp_val)

# Logarithm base 10
log10_val = np.log10(100)
print(log10_val)

# Natural logarithm (base e)
log_val = np.log(100)
print(log_val)
```

```
Output
7.38905609893065
2.0
4.605170185988092
```

The code calculates and prints the exponential of 2, the base-10 logarithm of 100, and the natural logarithm of 100 using NumPy.

4. **Statistical Functions**: These are the operations that provide summary statistics and insights into the characteristics of a dataset.

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

# Mean
mean_val = np.mean(data)
print(mean_val)

# Median
median_val = np.median(data)
print(median_val)

# Standard deviation
std_dev = np.std(data)
print(std_dev)

Output
3.0
3.0
1.4142135623730951
```

The code computes and prints the **mean**, **median**, and **standard deviation** of a NumPy array [1, 2, 3, 4, 5].

Linear algebra is a branch of mathematics that deals with vector spaces and linear mappings between these spaces. In the context of data science, machine learning, and scientific computing, linear algebra is widely used for tasks such as solving systems of linear equations, eigenvalue problems, and manipulating vectors and matrices. Let us learn about them.

Linear Algebra with NumPy

NumPy provides a comprehensive set of functions for linear algebra operations.

1. **Matrix Operations:** NumPy supports various matrix operations, such as addition, subtraction, and multiplication.

   Example Code:

```
# Matrix multiplication
matrix_a = np.array([[1, 2], [3, 4]])
matrix_b = np.array([[5, 6], [7, 8]])
result_matrix_multiply = np.dot(matrix_a, matrix_b)
print("Matrix Multiplication:")
print(result_matrix_multiply)

# Transpose of a matrix
result_transpose = np.transpose(matrix_a)
print("\nTranspose of Matrix A:")
print(result_transpose)

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

Transpose of Matrix A:
[[1 3]
 [2 4]]
```

   The code demonstrates matrix multiplication of two arrays (matrix_a and matrix_b) using np.dot() and prints the result. It also computes the transpose of matrix_a using np.transpose() and prints the transposed matrix.

2. **Eigenvalues and Eigenvectors:** NumPy can be used to compute eigenvalues and eigenvectors of a matrix. Eigenvalues and eigenvectors are used to analyse linear transformations represented by matrices.
   - Eigenvalues are scalar values associated with a square matrix, indicating how the matrix scales space in specific directions. They

are solutions to the equation Av = λv, where A is the matrix, λ is the eigenvalue, and v is the eigenvector.

- Eigenvectors are non-zero vectors that only change in scale (scalar multiplication) when a linear transformation is applied to them. They represent the directions in space that remain unchanged under the transformation.

You can learn more about Eigenvalues and Eigenvectors from this video- youtube.com/watch?v=PFDu9oVAE-g

Example Code:

```
# Import NumPy
import numpy as np

# Define a matrix
matrix_a = np.array([[1, 2], [3, 4]])

# Calculate eigenvalues and eigenvectors
eigenvalues, eigenvectors = np.linalg.eig(matrix_a)

# Print the results
print("Eigenvalues:")
print(eigenvalues)

print("\nEigenvectors:")
print(eigenvectors)
```

Output

```
Eigenvalues:
[-0.37228132  5.37228132]

Eigenvectors:
[[-0.82456484 -0.41597356]
 [ 0.56576746 -0.90937671]]
```

The code uses NumPy to calculate the eigenvalues and eigenvectors of a matrix a using np.linalg.eig() function. It then prints the eigenvalues and eigenvectors.

3. **Solving Linear Equations:** NumPy provides a function to solve systems of linear equations.

```
# Import NumPy
import numpy as np

# Define coefficients and constants
coefficients = np.array([[2, 3], [4, 5]])
constants = np.array([6, 7])

# Solve the system of linear equations
solution = np.linalg.solve(coefficients, constants)

# Print the solution
print("Solution to the system of linear equations:")
print(solution)

Output
Solution to the system of linear equations:
[-4.5  5. ]
```

This code defines a 2x2 matrix of coefficients with elements 2, 3, 4, and 5. A 1-dimentional NumPy array representing the constants is also created with values 6 and 7. **np.linalg.solve(coefficients, constants)** function is used to solve a system of linear equations represented by the coefficient matrix (coefficients) and the constant vector (constants). It returns an array representing the solution to the system.