Numpy

Notes by Mannan Ul Haq (BDS-3C)

Numpy Library in Python

NumPy **(short for Numerical Python)** is a fundamental library in Python for scientific computing and data manipulation. It is an essential tool for data scientists, as it provides support for handling large, multi-dimensional arrays and matrices, along with a collection of mathematical functions to operate on these arrays efficiently.

Why Use NumPy?

1. **Efficiency:** NumPy is written in C and optimized for performance. It allows you to perform operations on large datasets much faster than using standard Python lists.
2. **Multi-dimensional arrays:** NumPy introduces the `numpy.ndarray` object, which can represent arrays of any dimension.
3. **Mathematical functions:** NumPy provides a vast collection of mathematical functions that make it easy to perform complex mathematical operations on arrays.

Topic Outline for Learning Basics of NumPy:

1. **Creating Arrays**:
    - Creating arrays using **“array”** function.
    - Creating arrays using **“arange”** function.
    - Creating arrays using other functions like **numpy.zeros**, **numpy.empty**, etc.
    - Check the number of Dimensions.
    - Check the Data Type.
2. **Indexing and Slicing**:
    - Indexing with One-Dimensional Arrays.
    - Slicing One-Dimensional Arrays.
    - Indexing with Multi-Dimensional Arrays.

- Slicing Multi-Dimensional Arrays.

3. **Array Manipulation**:
    - Reshaping arrays.
    - Splitting arrays.
    - Sorting arrays.
    - Reversing arrays.
    - Conditional Elements Selection.
4. **Arithmetic with Arrays:**
    - Element-wise Arithmetic Operations.
    - Arithmetic Operations with Scalars.
    - Comparisons between Arrays.
5. **Mathematical Functions**:
    - Unary Functions.
    - Binary Functions.
    - Mathematical and Statistical Methods.
6. **Expressing Conditional Logic as Array Operations**
7. **Random Numbers and Arrays:**
    - random.rand
    - random.randint

1. Creating Arrays

Creating Arrays Using the `array` Function

1-Dimensional Array:

The easiest way to create arrays in NumPy is by using the `array` function. This function accepts any sequence-like object, including lists, and produces a new NumPy array containing the passed data. Here's how you can create **1D-Array** using the `array` function:

```
import numpy as np  # Import the NumPy library

# Creating an array from a list
data1 = [6, 7.5, 8, 0, 1]
arr1 = np.array(data1)
```

In this example, we start by importing NumPy as `np`. Then, we create an array `arr1` by passing a list `data1` to the `array` function. The resulting array `arr1` will contain the data from the list:

```
print(arr1)
# Output: [6.  7.5 8.  0.  1. ]
```

2-Dimensional Array:

NumPy can also handle nested sequences, such as a list of equal-length lists. When you pass nested sequences to `array`, it will create a **Multidimensional-Array**:

```
data2 = [[1, 2, 3, 4], [5, 6, 7, 8]]
arr2 = np.array(data2)
```

In this case, `arr2` will be a **2D-Array** with a shape inferred from the data:

```
print(arr2)
# Output:
# [[1, 2, 3, 4]
#  [5, 6, 7, 8]]
```

Creating Arrays Using the `arange` Function

In NumPy, the `arange` function is indeed an array-valued version of the built-in Python `range` function. It generates a NumPy array containing a sequence of numbers within a specified range.

```
import numpy as np

# Create a NumPy array with numbers from 0 to 14
result = np.arange(15)
```

```
# Output:
# array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14])
```

The `arange` function can take additional arguments to customize the generated sequence, such as specifying the start, stop, and step values. Here's an example:

```
import numpy as np

# Create an array with numbers from 2 to 10 with a step of 2
custom_range = np.arange(2, 11, 2)

# Output:
# array([ 2,  4,  6,  8, 10])
```

Creating Arrays Using Other Functions

In addition to `np.array`, NumPy provides other functions for creating new arrays. For example, `np.zeros` creates arrays filled with zeros, `np.ones` creates arrays filled with ones, and `np.empty` creates uninitialized arrays. You can specify the shape of the arrays by passing a tuple:

```
np.zeros(10)  # Create an array of 10 zeros
# Output: array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])

np.zeros((3, 6))  # Create a 2-dimensional array of zeros with shape (3, 6)
# Output:
# array([[0., 0., 0., 0., 0., 0.],
#        [0., 0., 0., 0., 0., 0.],
#        [0., 0., 0., 0., 0., 0.]])

np.empty((2, 3, 2))  # Create a 3-dimensional uninitialized array with shape (2, 3, 2)
# Output:
# array([[[0., 0.],
#         [0., 0.],
#         [0., 0.]],
#        [[0., 0.],
#         [0., 0.],
#         [0., 0.]]])
```

In NumPy, the `numpy.eye` function is used to create a 2-dimensional identity matrix, which is a square matrix with ones on the diagonal and zeros elsewhere.

Here's how you can use the `eye` function:

```
import numpy as np

# Create a 3x3 identity matrix
identity_matrix = np.eye(3)

# Output:
# array([[1., 0., 0.],
#        [0., 1., 0.],
#        [0., 0., 1.]])
```

In this example, we import NumPy as `np` and then use `np.eye(3)` to create a 3x3 identity matrix. You can replace `3` with any positive integer to create an identity matrix of the desired size.

Check the number of Dimensions:

You can check the number of dimensions and the shape of an array using the `ndim` and `shape` attributes:

```
arr2.ndim  # Number of dimensions
# Output: 2

arr2.shape  # Shape of the array
# Output: (2, 4)
```

Check the Data Type:

```
arr1.dtype  # Data type of arr1
# Output: dtype('float64')

arr2.dtype  # Data type of arr2
# Output: dtype('int64')
```

2. Indexing and Slicing in NumPy

Indexing with One-Dimensional Arrays:

Suppose we have a one-dimensional NumPy array:

```
import numpy as np

arr = np.arange(10)
```

You can access a single element at a specific position, just like you do with Python lists:

```
element = arr[5]
print(element)  # Output: 5
```

Slicing One-Dimensional Arrays:

Slicing allows you to select a range of elements from an array:

```
# Get a slice of numbers from position 2 to 5
slice_of_numbers = arr[2:6]
print(slice_of_numbers)  # Output: [2, 3, 4, 5]
```

You can even change the slice to a single value:

```
arr[2:6] = 99
print(arr)
# Output: [ 0  1 99 99 99 99  6  7  8  9]
```

Important Thing to Remember:

When you create a slice from an array, it doesn't copy the data. Any changes to the slice affect the original array.

Indexing with Multi-Dimensional Arrays:

Now, let's consider a two-dimensional NumPy array:

```
arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
```

You can access individual elements using their row and column positions:

```
element = arr2d[1, 2]
# OR
element = arr2d[1][2]

print(element)  # Output: 6 (Row 1, Column 2)
```

Slicing Multi-Dimensional Arrays:

Slicing in multi-dimensional arrays allows you to select portions of the data:

```
# Get the first two rows and all columns starting from the second column
slice_of_data = arr2d[:2, 1:]
print(slice_of_data)
# Output:
# [[2 3]
#  [5 6]]
```

Keep in Mind:

Slicing provides a view of the original data. Changes to the slice affect the original array.

3. Array Manipulation

Array manipulation is an important aspect of working with NumPy in Python, and it involves various operations to reshape arrays and split them into smaller arrays or subarrays. Let's explore these array manipulation techniques in detail:

Reshaping Arrays:

Reshaping arrays refers to changing the dimensions or shape of an existing array while preserving the total number of elements. NumPy provides the `reshape()` method to achieve this. Here's how you can reshape an array:

```
import numpy as np

# Create a 1D array with 12 elements
arr = np.arange(12)

# Reshape it into a 2D array with 3 rows and 4 columns
reshaped_arr = arr.reshape(3, 4)

print(reshaped_arr)
```

```
# Output:
# [[ 0  1  2  3]
#  [ 4  5  6  7]
#  [ 8  9 10 11]]
```

It's important to note that the total number of elements in the original array must match the total number of elements in the reshaped array.

Sorting Arrays:

NumPy provides the `sort()` method to sort elements in an array. You can sort arrays along specified axes or globally, and you can choose to sort in ascending or descending order.

Here's how you can sort a 1D array in ascending order:

```
import numpy as np

# Create a 1D array
arr = np.array([4, 2, 8, 1, 6])

# Sort the array in ascending order
sorted_arr = np.sort(arr)

print(sorted_arr)  # Output: [1 2 4 6 8]
```

To sort the array in descending order, you can use the `[::-1]` slicing notation:

```
# Sort the array in descending order
sorted_arr_desc = np.sort(arr)[::-1]

print(sorted_arr_desc)  # Output: [8 6 4 2 1]
```

For 2D arrays, you can specify the axis along which you want to perform the sorting:

```
# Create a 2D array
arr2d = np.array([[4, 2, 8], [1, 6, 3]])

# Sort the rows along axis 1 (columns) in ascending order
sorted_arr2d = np.sort(arr2d, axis=1)

print(sorted_arr2d)
```

```
# Output:
# [[2 4 8]
#  [1 3 6]]
```

Reversing Arrays:

You can reverse the order of elements in an array using slicing. Here's how you can reverse a 1D array:

```
import numpy as np

# Create a 1D array
arr = np.array([1, 2, 3, 4, 5])

# Reverse the array
reversed_arr = arr[::-1]

print(reversed_arr)  # Output: [5 4 3 2 1]
```

Conditional Elements Selection:

In NumPy, you can perform element selection based on conditions. This allows you to select elements from an array that satisfy specific conditions. Here's how to do it:

```
import numpy as np

# Create a sample NumPy array
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])

# Select elements that satisfy the condition
selected_elements = arr[arr % 2 == 0]

print(selected_elements)
```

In this example, we first create a NumPy array `arr`. We then select the elements from the original array that satisfy the condition.

Output (example):

```
[ 2  4  6  8 10]
```

4. Arithmetic with NumPy Arrays

NumPy arrays are versatile data structures that allow you to perform arithmetic operations efficiently on entire arrays, making it possible to express batch operations on data without the need for explicit **for loops**. Let's explore how arithmetic works with NumPy arrays based on the provided examples:

Element-wise Arithmetic Operations:

- When you perform arithmetic operations between equal-size NumPy arrays, the operations are applied element-wise. This means that each element in the resulting array is calculated independently based on the corresponding elements in the input arrays.

Here's an example using a NumPy array `arr`:

```
import numpy as np

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

The `arr` array looks like this:

```
array([[1, 2, 3],
       [4, 5, 6]])
```

Now, let's perform element-wise multiplication (`arr * arr`) on the `arr` array:

```
arr * arr
```

The result is an array where each element is the product of the corresponding elements in `arr`:

```
array([[ 1,  4,  9],
       [16, 25, 36]])
```

Similarly, when you perform subtraction (`arr - arr`), you get element-wise subtraction:

```
array([[0, 0, 0],
       [0, 0, 0]])
```

Arithmetic Operations with Scalars:

- NumPy also supports arithmetic operations between arrays and scalars. When you perform such operations, the scalar argument is propagated to each element in the array, and the operation is applied element-wise.

For example, raising `arr` to the power of `2` (`arr ** 2`) calculates the square of each element in the array:

```
array([[ 1,  4,  9],
       [16, 25, 36]])
```

Comparisons between Arrays:

- When you compare arrays of the same size, NumPy returns Boolean arrays indicating the results of element-wise comparisons.

Here's an example using two arrays, `arr` and `arr2`:

```
arr2 = np.array([[0, 4, 1], [7, 2, 12]])
```

The `arr2` array looks like this:

```
array([[ 0,  4,  1],
       [ 7,  2, 12]])
```

Now, when you perform the comparison `arr2 > arr`, NumPy returns a Boolean array where each element is `True` if the corresponding element in `arr2` is greater than the corresponding element in `arr`, and `False` otherwise:

```
array([[False,  True, False],
       [ True, False,  True]])
```

These examples demonstrate the power and flexibility of NumPy's element-wise arithmetic operations, making it a powerful tool for efficient data manipulation and scientific computations in Python.

5. Mathematical Functions

NumPy provides a wide range of mathematical functions that you can use to perform various mathematical operations on arrays. These functions are categorized into two main types: **unary functions** (operating on a single array) and **binary functions** (operating on two arrays). Let's explore these functions with examples:

Unary Functions:

`abs(x)` and `fabs(x)`: Calculate the absolute values of elements in an array. The only difference is that `fabs` returns the absolute values as floats.

```
import numpy as np

arr = np.array([-2, 3, -4.5])

abs_result = np.abs(arr)
print(abs_result)  # Output: [2.  3.  4.5]

fabs_result = np.fabs(arr)
print(fabs_result)  # Output: [2.  3.  4.5]
```

`sqrt(x)`: Compute the square root of each element in the array.

```
sqrt_result = np.sqrt(arr)
print(sqrt_result)  # Output: [       nan 1.73205081        nan]
```

`square(x)`: Calculate the square of each element in the array.

```
square_result = np.square(arr)
print(square_result)  # Output: [ 4.    9.   20.25]
```

`exp(x)`: Compute the exponential of each element in the array (e^x).

```
exp_result = np.exp(arr)
print(exp_result)  # Output: [0.13533528 20.08553692  0.011108  ]
```

`cos(x)`, `sin(x)`, and `tan(x)`: Calculate the cosine, sine, and tangent of each element in the array, respectively.

```
cos_result = np.cos(arr)
print(cos_result)  # Output: [-0.41614684 -0.9899925  -0.2107958 ]

sin_result = np.sin(arr)
print(sin_result)  # Output: [ 0.90929743  0.14112001 -0.97753012]

tan_result = np.tan(arr)
print(tan_result)  # Output: [-2.18503986 -0.14254654  4.63733248]
```

Binary Functions:

`add(x, y)`: Add corresponding elements of two arrays.

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

add_result = np.add(arr1, arr2)
print(add_result)  # Output: [5 7 9]
```

`subtract(x, y)`: Subtract elements of the second array from the first array.

```
subtract_result = np.subtract(arr1, arr2)
print(subtract_result)  # Output: [-3 -3 -3]
```

`multiply(x, y)`: Multiply corresponding elements of two arrays.

```
multiply_result = np.multiply(arr1, arr2)
print(multiply_result)  # Output: [ 4 10 18]
```

`divide(x, y)`: Divide elements of the first array by elements of the second array (element-wise division).

```
divide_result = np.divide(arr1, arr2)
print(divide_result)  # Output: [0.25 0.4  0.5 ]
```

`floor_divide(x, y)`: Floor division of elements in the first array by elements in the second array.

```
floor_divide_result = np.floor_divide(arr1, arr2)
print(floor_divide_result)  # Output: [0 0 0]
```

`power(x, y)`: Raise elements of the first array to the power of elements in the second array.

```
power_result = np.power(arr1, arr2)
print(power_result)  # Output: [  1  32 729]
```

`maximum(x, y)`: Element-wise maximum between two arrays.

```
maximum_result = np.maximum(arr1, arr2)
print(maximum_result)  # Output: [4 5 6]
```

`minimum(x, y)`: Element-wise minimum between two arrays.

```
minimum_result = np.minimum(arr1, arr2)
print(minimum_result)  # Output: [1 2 3]
```

`mod(x, y)`: Compute element-wise remainder of division.

```
mod_result = np.mod(arr1, arr2)
print(mod_result)  # Output: [1 2 3]
```

Comparison Functions: Functions like `greater`, `greater_equal`, `less`, `less_equal`, `equal`, and `not_equal` are used for element-wise comparison of two arrays. They return Boolean arrays indicating the comparison results.

```
compare_result = np.greater(arr1, arr2)
print(compare_result)  # Output: [False False False]
```

Mathematical and Statistical Methods:

NumPy provides a wide range of mathematical and statistical methods that you can use to analyze and manipulate arrays efficiently. These methods allow you to perform common operations such as calculating the mean, standard deviation, variance, minimum, and maximum values of arrays. Let's explore these methods with examples:

Mathematical Methods:

1. `mean(x)`: Calculate the arithmetic mean (average) of the elements in an array.

```
import numpy as np

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

mean_result = np.mean(arr)
print(mean_result)  # Output: 3.0
```

1. `std(x)`: Compute the standard deviation of the elements in an array, which measures the spread or dispersion of the data.

```
std_result = np.std(arr)
print(std_result)  # Output: 1.4142135623730951
```

1. `var(x)`: Calculate the variance of the elements in an array, which quantifies how much the data values vary from the mean.

```
var_result = np.var(arr)
print(var_result)  # Output: 2.0
```

Statistical Methods:

1. `min(x)`: Find the minimum value in an array.

```
min_result = np.min(arr)
print(min_result)  # Output: 1
```

1. `max(x)`: Find the maximum value in an array.

```
max_result = np.max(arr)
print(max_result)  # Output: 5
```

Multi-Dimensional Arrays:

These methods can also be applied to multi-dimensional arrays, and you can specify the axis along which the calculations should be performed.

```
arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

# Calculate the mean along each row (axis=1)
mean_row = np.mean(arr2d, axis=1)
print(mean_row)
# Output: [2. 5. 8.]

# Calculate the minimum along each column (axis=0)
min_column = np.min(arr2d, axis=0)
print(min_column)
# Output: [1 2 3]
```

6. Expressing Conditional Logic as Array Operations

In NumPy, you can perform conditional logic operations on arrays efficiently using the `numpy.where` function. This function acts like a vectorized version of the "if-else" ternary expression.

Basic Conditional Logic

Suppose you have two arrays, `xarr` and `yarr`, and a Boolean array `cond`. You want to create a new array where each element is taken from `xarr` if the corresponding element in `cond` is `True`, and from `yarr` if it's `False`.

Traditionally, you might use a loop or list comprehension to achieve this, but it's not very efficient and doesn't work well with large arrays. In NumPy, you can do this with a single

function call:

```
result = np.where(cond, xarr, yarr)
```

Here, `cond` is the condition array, `xarr` and `yarr` are the arrays you want to choose from based on the condition, and `result` will be the new array containing the selected values.

Example:

Suppose you have a matrix of random numbers in `arr`, and you want to replace all positive values with 2 and all negative values with -2. You can do this efficiently using `np.where`:

```
result = np.where(arr > 0, 2, -2)
```

Here, `arr > 0` creates a Boolean array where `True` represents positive values. Then, `np.where` replaces `True` values with 2 and `False` values with -2, resulting in a new modified array.

You can also combine scalars and arrays when using `np.where`. For instance, you can set only positive values in `arr` to 2 while keeping negative values as they are:

```
result = np.where(arr > 0, 2, arr)
```

In this case, positive values are replaced with 2, and negative values remain unchanged.

7. Random Numbers and Arrays

NumPy's `random.rand` and `random.randint` functions are useful for generating random numbers and arrays. They are commonly used in various scientific and data analysis tasks. Let's explore these functions in detail:

1. `random.rand`: Generating Random Floats

`random.rand` is used to generate random floating-point numbers between 0 (inclusive) and 1 (exclusive) from a uniform distribution. You can specify the shape of the resulting array.

Here's how to use `random.rand` to generate random floats:

```
import numpy as np

# Generate a single random float between 0 and 1
random_float = np.random.rand()
print(random_float)

# Generate a 1D array of 5 random floats
random_array = np.random.rand(5)
print(random_array)

# Generate a 2D array of random floats with shape (3, 2)
random_2d_array = np.random.rand(3, 2)
print(random_2d_array)
```

Output (example):

```
0.45319867163338076
[0.29311845 0.64552256 0.82291909 0.84194604 0.33511567]
[[0.16871563 0.94722351]
 [0.27622938 0.05858612]
 [0.17510878 0.36191377]]
```

2. `random.randint`: Generating Random Integers

`random.randint` is used to generate random integers within a specified range. You can specify the low (inclusive) and high (exclusive) values, as well as the size or shape of the resulting array.

Here's how to use `random.randint` to generate random integers:

```
import numpy as np

# Generate a single random integer between 1 and 10
random_int = np.random.randint(1, 11)
print(random_int)

# Generate a 1D array of 5 random integers between 0 and 9
```

```
random_int_array = np.random.randint(10, size=5)
print(random_int_array)

# Generate a 2D array of random integers with shape (3, 2) between 100 and 199
random_2d_int_array = np.random.randint(100, 200, size=(3, 2))
print(random_2d_int_array)
```

Output (example):

```
7
[3 6 8 2 9]
[[156 122]
 [129 125]
 [166 122]]
```

These functions are helpful when you need to introduce randomness into your simulations, experiments, or data generation processes.