PW SKILLS

Lesson Plan

NUMPY

NUMPY

NumPy stands for “Numerical Python” and it is the standard Python library used for working with arrays (i.e., vectors & matrices), linear algebra, and other numerical computations. NumPy is written in C, making NumPy arrays faster and more memory efficient than Python lists or arrays.

NumPy is the fundamental package for scientific computing in Python. It is a Python library that provides a multidimensional array object, various derived objects (such as masked arrays and matrices), and an assortment of routines for fast operations on arrays, including mathematical, logical, shape manipulation, sorting, selecting, I/O, discrete Fourier transforms, basic linear algebra, basic statistical operations, random simulation and much more.

At the core of the NumPy package, is the ndarray object. This encapsulates n-dimensional arrays of homogeneous data types, with many operations being performed in compiled code for performance. There are several important differences between NumPy arrays and the standard Python sequences:

- NumPy arrays have a fixed size at creation, unlike Python lists (which can grow dynamically). Changing the size of an ndarray will create a new array and delete the original.
- The elements in a NumPy array are all required to be of the same data type, and thus will be the same size in memory. The exception: one can have arrays of (Python, including NumPy) objects, thereby allowing for arrays of different sized elements.
- NumPy arrays facilitate advanced mathematical and other types of operations on large numbers of data. Typically, such operations are executed more efficiently and with less code than is possible using Python’s built-in sequences.
- A growing plethora of scientific and mathematical Python-based packages are using NumPy arrays; though these typically support Python-sequence input, they convert such input to NumPy arrays prior to processing, and they often output NumPy arrays. In other words, in order to efficiently use much (perhaps even most) of today’s scientific/mathematical Python-based software, just knowing how to use Python’s built-in sequence types is insufficient - one also needs to know how to use NumPy arrays.

Why Numpy faster than Python lists?

- In contrast to lists, NumPy arrays are stored in a single continuous location in memory, making it very easy for programs to access and manipulate them.
- In computer science, this behavior is known as locality of reference.
- This is the primary factor that makes NumPy faster than lists. Additionally, it is enhanced to function with modern CPU architectures.

Numpy Installation

NumPy is fairly simple to install if Python and PIP are already set up on a machine.
Use this command to install it:

```
C:\Users\Your Name>pip install numpy
```

Typically, NumPy is imported via the np alias.

Alias: In Python, an alias is a different name for the same item.

While importing, use the as keyword to create an alias:

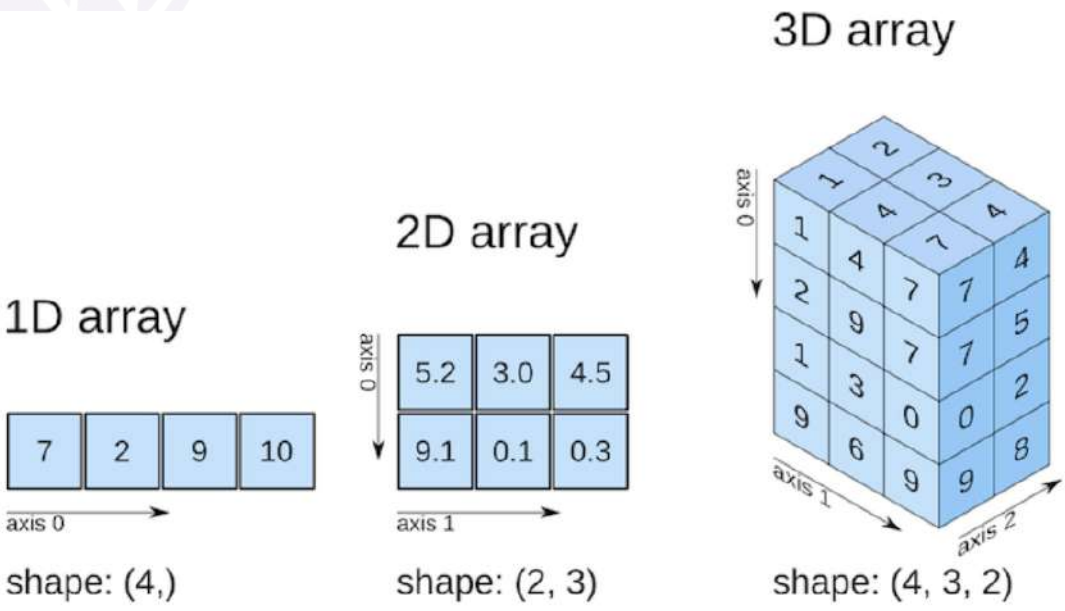
```
import numpy as np
```

Checking NumPy Version

```
import numpy as np
print(np.__version__)
```

Arrays

Arrays are “n-dimensional” data structures that can contain all the basic Python data types, e.g., floats, integers, strings etc, but work best with numeric data. NumPy arrays (“ndarrays”) are homogenous, which means that items in the array should be of the same type.



Ways to create Arrays using NumPy:

Example 1 :

```
import numpy
arr= numpy.array([10,20,30,40,50])  #create array
print(arr)
```

Example 2 :

```
import numpy as np # importing numpy array
type(np.array([1, 2, 3])) # type of numpy array is ndarray
```

Example 3 :

```
# numpy maintains the homogeneous data type
A = np.array([[1.1,2.2,3.3,4.4,5.5],
              [1,2,3,4,5]])
```

Create array using built in functions -

```
zeros_array = np.zeros((3, 4))  # Creates a 3x4 array filled with
zeros
ones_array = np.ones((2, 2))    # Creates a 2x2 array filled with
ones
random_array = np.random.rand(3, 3)  # Creates a 3x3 array with
random values between 0 and 1
```

Dimensions in Arrays:

A dimension in arrays is one level of array depth (nested arrays).

1. 0-D Arrays

0-D arrays, or Scalars, are the elements in an array. Each value in an array is a 0-D array.

```
import numpy as np
arr = np.array(42)
print(arr)
```

2. 1-D Arrays

An array that has 0-D arrays as its elements is called a uni-dimensional or 1-D array.
These are the most common and basic arrays.

Example

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

```
import numpy as np

# Create a 1D array using arange
my_array = np.arange(1, 6)  # Creates an array from 1 to 5
```

```
# Create a 1D array using linspace
my_array = np.linspace(1, 5, 5)  # Creates an array of 5 elements
from 1 to 5
```

3. 2-D Arrays

An array that has 1-D arrays as its elements is called a 2-D array.
These are often used to represent matrix or 2nd order tensors.

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

```
# Create a 2D array from nested Python lists
nested_list = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
my_2d_array = np.array(nested_list)
```

```
# Create a 2D array filled with zeros
zeros_array = np.zeros((3, 4))  # 3 rows, 4 columns
```

```
# Create a 2D array filled with ones
ones_array = np.ones((2, 3))    # 2 rows, 3 columns
```

4. 3-D Arrays

An array that has 1-D arrays as its elements is called a 2-D array.
These are often used to represent matrix or 2nd order tensors.

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

```
# Create a 3D array from nested Python lists
nested_list_3d = [[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11,
12]]]
my_3d_array = np.array(nested_list_3d)
```

```
# Create a 3D array filled with zeros
zeros_array_3d = np.zeros((2, 3, 4))  # 2 matrices, each with 3
rows and 4 columns
```

```
# Create a 3D array filled with ones
ones_array_3d = np.ones((3, 2, 2))    # 3 matrices, each with 2
rows and 2 columns
```

```
# Create a 3D array with random values
random_array_3d = np.random.rand(2, 3, 3)  # 2 matrices, each
with 3 rows and 3 columns
```

3. Higher Dimensional Arrays

An array can have any number of dimensions.

When the array is created, you can define the number of dimensions by using the ndmin argument.

```
arr = np.array([1, 2, 3, 4], ndmin=5)
print(arr)
print('number of dimensions :', arr.ndim)
```

4D Array with zeros function:

```
# Create a 4D array filled with zeros
zeros_array_4d = np.zeros((2, 3, 4, 5))  # 2 sets of 3 matrices,
each with 4 rows and 5 columns

print(zeros_array_4d)
```

5D Array with ones function:

```
# Create a 5D array filled with ones
ones_array_5d = np.ones((2, 2, 3, 4, 2))  # 2 sets of 2 matrices,
each with 3 sets of 4 rows and 2 columns

print(ones_array_5d)
```

6D Array with Random Values:

```
# Create a 6D array with random values
random_array_6d = np.random.rand(2, 2, 2, 3, 4, 3)  # 2 sets of 2
sets of 2 matrices, each with 3 sets of 4 rows and 3 columns

print(random_array_6d)
```

Numpy Data Types

NumPy provides a rich set of data types that can be used to create arrays with different types of elements. The data types in NumPy are similar to, but not exactly the same as, Python data types. Here are some commonly used NumPy data types:

- **Integer Types:**

```
• np.int8, np.int16, np.int32, np.int64
• np.uint8, np.uint16, np.uint32, np.uint64
```

- **Floating-point Types:**

```
• np.float16, np.float32, np.float64
```

- **Complex Types:**

```
• np.complex64, np.complex128
```

- **Boolean Type:**

```
• np.bool
```

- **String Types:**

```
• np.str_, np.unicode_
```

- **Datetime Types:**

```
• np.datetime64
```

- **Object Type:**

```
• np.object
```

- **Fixed-size String Type:**

```
• np.bytes_
```

NumPy has some extra data types, and refers to data types with one character, like i for integers, u for unsigned integers etc. Below is a list of all data types in NumPy and the characters used to represent them:

- `i` - integer
- `b` - boolean
- `u` - unsigned integer
- `f` - float
- `c` - complex float
- `m` - timedelta
- `M` - datetime
- `O` - object
- `S` - string
- `U` - unicode string
- `V` - fixed chunk of memory for other type (void)

1. Checking the Data Type of an Array:

The NumPy array object has a property called dtype that returns the data type of the array:

```
import numpy as np
arr_num = np.array([1, 2, 3, 4])
print(arr_num .dtype)

arr_fruit = np.array(['apple', 'banana', 'cherry'])
print(arr_fruit .dtype)
```

2. Creating Arrays With a Defined Data Type:

We use the array() function to create arrays, this function can take an optional argument: dtype that allows us to define the expected data type of the array elements:

```
import numpy as np
arr = np.array([1, 2, 3, 4], dtype='S')
print(arr)
print(arr.dtype)
```

3. Converting Data Type on Existing Arrays

The best way to change the data type of an existing array, is to make a copy of the array with the astype() method. The astype() function creates a copy of the array, and allows you to specify the data type as a parameter. The data type can be specified using a string, like 'f' for float, 'i' for integer etc. or you can use the data type directly like float for float and int for integer.

```
import numpy as np

arr = np.array([1.1, 2.1, 3.1])
newarr_1 = arr.astype('i')
print(newarr_1)
print(newarr_1.dtype)

arr = np.array([1, 0, 3])
newarr_2 = arr.astype(bool)
print(newarr_2)
print(newarr_2.dtype)
```

Creating Array from Existing Data

"NumPy - Array From Existing Data" refers to the capability of creating NumPy arrays using pre-existing data structures or arrays. NumPy is a powerful numerical computing library in Python, and its arrays (ndarrays) are a fundamental data structure that allows efficient manipulation of numerical data.

Here are some common ways to create NumPy arrays from existing data.

1. **numpy.array:** This function creates an array from an existing iterable, such as a list or tuple.

```
import numpy as np

original_list = [1, 2, 3]
numpy_array = np.array(original_list)
print(numpy_array)

# This example creates a NumPy array from a Python list.
```

```
import numpy as np

# Creating an array from a tuple
tuple_data = (4, 5, 6)
array_from_tuple = np.array(tuple_data)
print(array_from_tuple)
```

```
import numpy as np
# Creating an array from a mixed data type list
mixed_data = [1, 2.5, 'Hello']
array_from_mixed = np.array(mixed_data)
print(array_from_mixed)
```

2. **numpy.asarray:** Similar to numpy.array, numpy.asarray converts input to an array. If the input is already an array, it returns that array; otherwise, it creates a new one.

```
import numpy as np
original_tuple = (4, 5, 6)
numpy_array = np.asarray(original_tuple)
print(numpy_array)
# In this example, a NumPy array is created from a tuple
```

```
import numpy as np
# Creating an array from a list
list_data = [7, 8, 9]
array_from_list = np.asarray(list_data)
print(array_from_list)
```

```
import numpy as np
# Creating an array from a tuple
tuple_data = (10, 11, 12)
array_from_tuple = np.asarray(tuple_data)
print(array_from_tuple)
```

```
import numpy as np
# Creating an array from an existing array
original_array = np.array([13, 14, 15])
array_from_existing = np.asarray(original_array)
print(array_from_existing)
```

3. Slicing and Indexing: You can create a view of an existing array by slicing or indexing.

```
import numpy as np

original_array = np.array([7, 8, 9])

# Creating a view using slicing
sliced_array = original_array[:2]
print(sliced_array)
# Creating a view using indexing
indexed_array = original_array[[0, 2]]
print(indexed_array)
# Slicing and indexing allow you to extract portions of an
existing array, creating a view that refers to the same data.
```

```
import numpy as np

# Creating a view using slicing
original_array = np.array([16, 17, 18])
sliced_array = original_array[:2]
print(sliced_array)
```

```
import numpy as np

# Creating a view using indexing
original_array = np.array([19, 20, 21])
indexed_array = original_array[[0, 2]]
print(indexed_array)
```

```
import numpy as np
# Creating a view with a different shape using slicing
original_array = np.array([[22, 23, 24], [25, 26, 27]])
view_array = original_array[:, 1:]
print(view_array)
```

4. **numpy.copy:** This method creates a deep copy of the array, ensuring that changes made to the copied array do not affect the original array.

```
import numpy as np
original_array = np.array([10, 11, 12])
copied_array = np.copy(original_array)
print(copied_array)
# The numpy.copy function ensures that changes made to
copied_array do not affect original_array.
```

```
import numpy as np
# Creating a deep copy
original_array = np.array([28, 29, 30])
copied_array = np.copy(original_array)
print(copied_array)
```

```
import numpy as np
# Creating a deep copy with a different data type
original_array = np.array([31, 32, 33], dtype=float)
copied_array = np.copy(original_array)
print(copied_array)
```

```
import numpy as np
# Creating a deep copy of a multi-dimensional array
original_array = np.array([[34, 35], [36, 37]])
copied_array = np.copy(original_array)
print(copied_array)
```

5. **numpy.view:** The numpy.view method creates a new array object that looks at the same data. Changes to the view affect the original array, but the shape and strides of the view can be different.

```
import numpy as np
original_array = np.array([13, 14, 15])
view_array = original_array.view()
print(view_array)
# In this example, view_array is a view of the data in
original_array.
```

```
import numpy as np
# Creating a view
original_array = np.array([38, 39, 40])
view_array = original_array.view()
print(view_array)
```

```
import numpy as np
# Creating a view with a different shape
original_array = np.array([41, 42, 43, 44, 45, 46])
view_array = original_array.view(dtype=np.int32).reshape(4, 3)
print(view_array)
```

```
import numpy as np
# Creating a view with a different data type
original_array = np.array([47, 48, 49], dtype=float)
view_array = original_array.view(dtype=np.int16)
print(view_array)
```

6. numpy.concatenate: Concatenating arrays along an existing axis.

```
import numpy as np
array1 = np.array([16, 17, 18])
array2 = np.array([19, 20, 21])
concatenated_array = np.concatenate((array1, array2))
print(concatenated_array)
# The numpy.concatenate function allows you to join two or more
arrays along a specified axis.
```

```
import numpy as np

# Creating two arrays along rows
array1 = np.array([[1, 2, 3], [4, 5, 6]])
array2 = np.array([[7, 8, 9]])

# Concatenating along rows
result_array = np.concatenate((array1, array2), axis=0)

print("Array 1:")
print(array1)
print("\nArray 2:")
print(array2)
print("\nConcatenated Array along Rows:")
print(result_array)
```

```
import numpy as np

# Creating two arrays along columns
array1 = np.array([[1, 2], [3, 4]])
array2 = np.array([[5, 6]])

# Concatenating along columns


result_array = np.concatenate((array1, array2.T), axis=1)

print("Array 1:")
print(array1)
print("\nArray 2:")
print(array2)
print("\nConcatenated Array along Columns:")
print(result_array)
```

```
import numpy as np

# Creating two arrays
array1 = np.array([1, 2, 3])
array2 = np.array([4, 5, 6])

# Concatenating along a new axis
result_array = np.concatenate((array1[np.newaxis, :],
array2[np.newaxis, :]), axis=0)

print("Array 1:")
print(array1)
print("\nArray 2:")
print(array2)
print("\nConcatenated Array along a New Axis:")
print(result_array)
```

Array creation routines

Array creation routines in NumPy are functions or methods that allow you to create arrays with specific properties, shapes, and values. These routines provide a convenient way to initialize NumPy arrays for various purposes. Here's a brief overview of some common array creation routines:

- `numpy.array`: Creates an array from a list or tuple.
- `numpy.zeros`: Creates an array filled with zeros.
- `numpy.ones`: Creates an array filled with ones.
- `numpy.empty`: Creates an array without initializing its values.
- `numpy.arange`: Creates an array with evenly spaced values within a specified range.
- `numpy.linspace`: Creates an array with a specified number of evenly spaced values within a range.
- `numpy.logspace`: Creates an array with values evenly spaced in a logarithmic scale.
- `numpy.eye and numpy.identity`: Create a 2D identity matrix.
- `numpy.random.rand and numpy.random.randn`: Create an array with random values.
- `numpy.full`: Creates an array with a constant value.

These routines offer flexibility in generating arrays for different tasks, such as mathematical computations, data manipulation, and scientific computing. You can choose the appropriate routine based on your requirements, specifying the shape, data type, and values of the resulting array.

For example, numpy.zeros is useful when you want to create an array filled with zeros, numpy.arange when you need a range of values with a specified step, and numpy.random.rand when you want an array with random values between 0 and 1. The array creation routines simplify the process of initializing arrays and are foundational for working with NumPy in various applications.

1. numpy.array

Creates an array from a list or tuple.
The numpy.array function converts a Python list or tuple into a NumPy array. It automatically determines the data type based on the elements of the input.

```
import numpy as np
my_list = [1, 2, 3, 4]
my_array = np.array(my_list)
print(my_array)
import numpy as np
my_list = [10, 20, 30]
my_array = np.array(my_list)
print(my_array)
```

```
import numpy as np
my_tuple = (1.5, 2.5, 3.5)
my_array = np.array(my_tuple)
print(my_array)
```

```
import numpy as np
nested_list = [[1, 2, 3], [4, 5, 6]]
my_array = np.array(nested_list)
print(my_array)
```

2. numpy.zeros

Creates an array filled with zeros.
The numpy.zeros function generates an array of zeros with the specified shape. It is useful when you want to initialize an array with default values.

```
import numpy as np
zeros_array = np.zeros((3, 4))  # Creates a 3x4 array filled with
zeros
print(zeros_array)
```

```
import numpy as np
zeros_array = np.zeros((2, 3))
print(zeros_array)
```

```
import numpy as np
zeros_array = np.zeros((3, 2), dtype=int)
print(zeros_array)
```

```
import numpy as np
zeros_array = np.zeros((1, 5))
print(zeros_array)
```

3. numpy.ones

Creates an array filled with ones.
Similar to numpy.zeros, numpy.ones creates an array with the specified shape but fills it with ones instead.

```
import numpy as np
ones_array = np.ones((2, 2))  # Creates a 2x2 array filled with
ones
print(ones_array)
```

```
import numpy as np
ones_array = np.ones((3, 2))
print(ones_array)
```

```
import numpy as np
ones_array = np.ones((2, 4), dtype=int)
print(ones_array)
```

```
import numpy as np
ones_array = np.ones((1, 3))
print(ones_array)
```

4. numpy.empty

Creates an array without initializing its values.
The numpy.empty function creates an array without setting the values. The content of the array may vary, and it's often used when you plan to fill the array with specific values later.

```
import numpy as np
empty_array = np.empty((2, 3))  # Creates a 2x3 array without
initializing values
print(empty_array)
```

```
import numpy as np
empty_array = np.empty((2, 2))
print(empty_array)
```

```
import numpy as np
empty_array = np.empty((3, 3), dtype=int)
print(empty_array)
```

```
import numpy as np
empty_array = np.empty((1, 4))
print(empty_array)
```

5. numpy.arange

Creates an array with evenly spaced values within a specified range.
numpy.arange generates an array with values ranging from a start value to an end value with a specified step.

```
import numpy as np
arange_array = np.arange(0, 10, 2)  # Creates an array from 0 to
10 with a step of 2
print(arange_array)
```

```
import numpy as np
arange_array = np.arange(0, 10, 2)
print(arange_array)
```

```
import numpy as np
arange_array = np.arange(3)
print(arange_array)
```

```
import numpy as np
arange_array = np.arange(3)
print(arange_array)
```

```
import numpy as np
arange_array = np.arange(5, 10)
print(arange_array)
```

6. numpy.linspace

Creates an array with a specified number of evenly spaced values within a range.
numpy.linspace generates an array with a specified number of values evenly spaced between a start and end value.

```
import numpy as np
linspace_array = np.linspace(0, 1, 5)  # Creates an array with 5
values from 0 to 1
print(linspace_array)
```

```
import numpy as np
linspace_array = np.linspace(0, 120, 5)
print(linspace_array)
```

```
import numpy as np
linspace_array = np.linspace(1, 10, 3)
print(linspace_array)
```

```
import numpy as np
linspace_array = np.linspace(0, 360, 6)
print(linspace_array)
```

7. numpy.logspace

Creates an array with values evenly spaced in a logarithmic scale.
numpy.logspace generates an array with values evenly spaced on a logarithmic scale. The start and end values are the powers of 10.

```
import numpy as np
logspace_array = np.logspace(0, 2, 5)  # Creates an array with 5
values in log scale from 10^0 to 10^2
print(logspace_array)
```

```
import numpy as np
logspace_array = np.logspace(1, 3, 4)
print(logspace_array)
```

```
import numpy as np
logspace_array = np.logspace(-1, 1, 3)
print(logspace_array)
```

8. numpy.eye and numpy.identity

Create a 2D identity matrix.
`numpy.eye` and `numpy.identity` create a square matrix with ones on the main diagonal and zeros elsewhere. It's often used in linear algebra.

```
import numpy as np
identity_matrix = np.identity(4)
print(identity_matrix)
```

```
import numpy as np
identity_matrix = np.identity(4)
print(identity_matrix)
```

```
import numpy as np
identity_matrix = np.eye(2, dtype=int)
print(identity_matrix)
```

9. numpy.random.rand and numpy.random.randn

Create an array with random values.
numpy.random.rand generates random values from a uniform distribution between 0 and 1.
numpy.random.randn generates values from a standard normal distribution with mean 0 and standard deviation 1.

```
import numpy as np
random_array = np.random.rand(3, 3)  # Creates a 3x3 array with
values from a uniform distribution [0, 1)
print(random_array)
```

```
import numpy as np
random_normal_array = np.random.randn(3, 3)  # Creates a 3x3
array with values from a standard normal distribution
print(random_normal_array)
```

```
import numpy as np
random_array = np.random.rand(2, 3)
print(random_array)
```

```
import numpy as np
random_normal_array = np.random.randn(3, 2)
print(random_normal_array)
```

```
import numpy as np
random_array = np.random.rand(1, 4)
print(random_array)
```

10. numpy.full

Create an array with a constant value.

numpy.full creates an array with a specified shape and fills it with a constant value.

These array creation routines provide a foundation for working with NumPy arrays, allowing you to easily generate arrays with specific properties for various applications.

```
import numpy as np
full_array = np.full((2, 2), 7)  # Creates a 2x2 array filled
with the value 7
print(full_array)
```

```
import numpy as np
full_array = np.full((3, 3), 2.5)
print(full_array)
```

```
import numpy as np
full_array = np.full((1, 5), -1)
print(full_array)
```

Broadcasting

NumPy broadcasting is a powerful feature that allows for implicit element-wise operations between arrays of different shapes and sizes. Broadcasting automatically adjusts the shape of smaller arrays to make them compatible with larger arrays, eliminating the need for explicit looping or copying of data. This makes it easier to perform operations on arrays of different shapes, reducing the need for unnecessary duplication of data. The general rule for broadcasting is that the dimensions of the arrays must be compatible, meaning that either the dimensions are equal, or one of them is 1. NumPy automatically performs broadcasting when you perform operations between arrays that satisfy these conditions.

NumPy automatically expands (broadcasts) smaller arrays to match the shape of larger arrays, making element-wise operations possible.

Rules of Broadcasting

1. Array with smaller ndim than the other is prepended with '1' in its shape.
2. Size in each dimension of the output shape is maximum of the input sizes in that dimension.
3. An input can be used in calculation, if its size in a particular dimension matches the output size or its value is exactly 1.
4. If an input has a dimension size of 1, the first data entry in that dimension is used for all calculations along that dimension.

A set of arrays is said to be broadcastable if the above rules produce a valid result and one of the following is true

1. Arrays have exactly the same shape.
2. Arrays have the same number of dimensions and the length of each dimension is either a common length or 1.
3. Array having too few dimensions can have its shape prepended with a dimension of length 1, so that the above stated property is true.

Examples

Example 1: Broadcasting with a Scalar

```
import numpy as np

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

# Adding a scalar to every element in the array
result = arr + 10
print(result)
```

Example 2: Broadcasting with a 1D Array

```
import numpy as np

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

# Multiplying every element in the array by a scalar (1D array)
result = arr * multiplier
print(result)
```

Example 3: Broadcasting with 1D and 2D Arrays

```
import numpy as np

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

# Multiplying every element in the array by a scalar (1D array)
result = arr * multiplier
print(result)
```

Example 4: Broadcasting with Different Dimensions

```
import numpy as np

arr_2d = np.array([[1, 2, 3], [4, 5, 6]])
column_addition = np.array([[10], [20]])

# Adding a column to each column in the 2D array

result = arr_2d + column_addition
print(result)
```

Example 5: Broadcasting with Incompatible Shapes

```
import numpy as np

arr_2d = np.array([[1, 2, 3], [4, 5, 6]])
row_addition = np.array([10, 20, 30, 40])

# Attempting to add incompatible shapes (will result in an error)
try:
    result = arr_2d + row_addition
except ValueError as e:
    print(f"ValueError: {e}")
```

In this example, broadcasting fails because the shapes of the arrays are not compatible. The number of elements in the dimensions must either be equal or one of them must be 1.

Example 6: Broadcasting with Dimension Expansion

```
import numpy as np

arr_2d = np.array([[1, 2, 3], [4, 5, 6]])
column_expansion = np.array([[10], [20]])

# Expanding dimensions to make shapes compatible
result = arr_2d + column_expansion
print(result)
```

Example 7: Broadcasting with Unary Operations

```
import numpy as np

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

# Applying a unary operation (square) to each element in the
array
result = arr ** 2
print(result)
```

Example 8: Broadcasting with Conditional Operations

```
import numpy as np

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

# Creating a boolean mask and broadcasting the condition
condition = arr > threshold

result = arr[condition]
print(result)
```

Example 9: Broadcasting with Mathematical Functions

```
import numpy as np

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

# Applying a mathematical function (e.g., square root) to each
element in the array
result = np.sqrt(arr)
print(result)
```

Example 10: Broadcasting with Reshape

```
import numpy as np

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

# Reshaping the array and broadcasting a scalar
reshaped_arr = arr.reshape((2, 3))
result = reshaped_arr * 10
print(result)
```

Reshaping can be combined with broadcasting to efficiently perform operations on arrays of different shapes.

Advance indexing

NumPy offers more indexing facilities than regular Python sequences. In addition to indexing by integers and slices, as we saw before, arrays can be indexed by arrays of integers and arrays of booleans.

Indexing with Arrays of Indices

```
import numpy as np
a = np.arange(12)**2                    # the first 12 square
numbers
i = np.array( [ 1,1,3,8,5,6 ] )              # an array of
indices
print(a[i] ,"# the elements of a at the positions i")
```

Output –

```
[ 1 1 9 64 25 36] # the elements of a at the positions i

j = np.array( [ [ 3, 4], [ 9, 7 ] ] )      # a bidimensional
array of indices
a[j]                                       # the same shape as j
```

Output –

```
array([[ 9, 16], [81, 49]])
```

When the indexed array a is multidimensional, a single array of indices refers to the first dimension of a. The following example shows this behavior by converting an image of labels into a color image using a palette.

```
palette = np.array( [ [0,0,0],                 # black
                      [255,0,0],               # red
                      [0,255,0],               # green
                      [0,0,255],               # blue
                      [255,255,255] ] )        # white

palette
```

Output -

```
array([[ 0,  0,  0], [255,  0,  0], [ 0, 255,  0], [ 0,  0, 255], [255,
255, 255]])
```

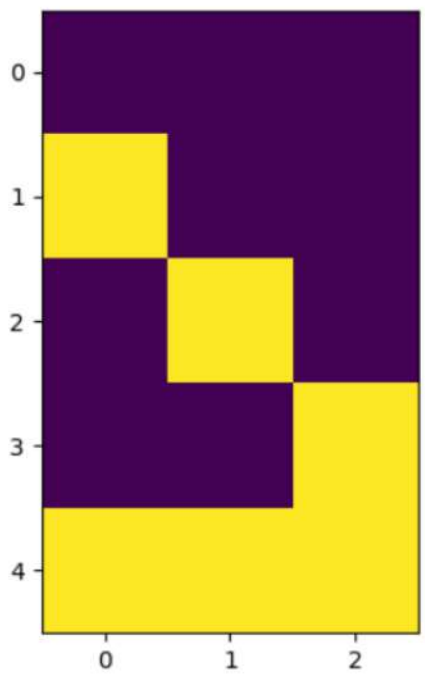
Example:

```
import matplotlib.pyplot as plt
plt.imshow(palette)
```

Output -

```
import matplotlib.pyplot as plt
plt.imshow(palette)
```

```
<matplotlib.image.AxesImage at 0x7a8e4e719ed0>
```



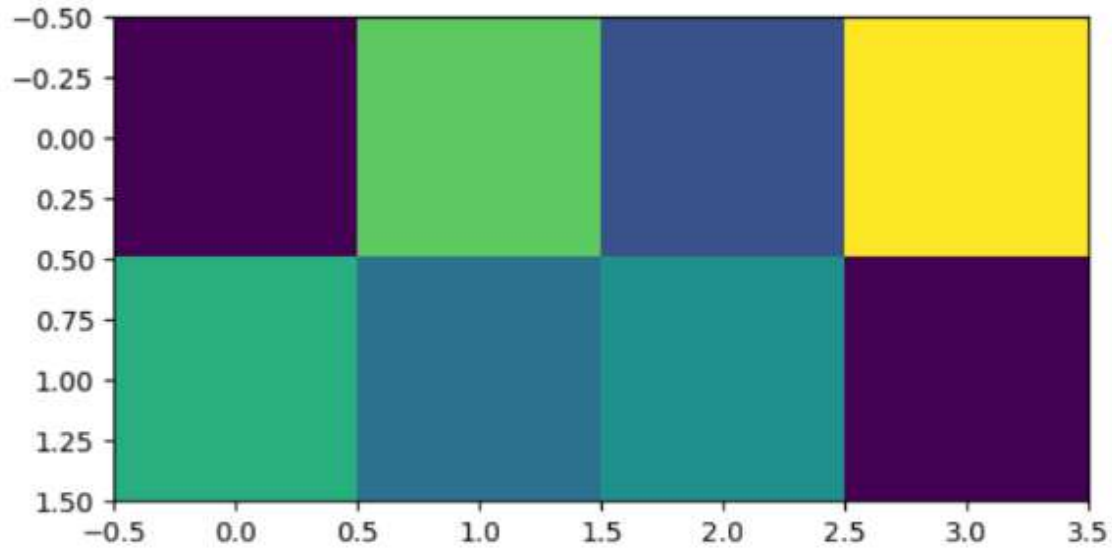

```
image = np.array( [ [ 0, 6, 2, 8 ],            # each value
corresponds to a color in the palette
                    [ 5, 3, 4, 0 ] ] )
print(image)
```

Output -

```
[[0 6 2 8] [5 3 4 0]]
import matplotlib.pyplot as plt
plt.imshow(image)
```

Output -

```
<matplotlib.image.AxesImage at 0x7a8e4c255f30>
```



We can give indexes for more than one dimension. The arrays of indices for each dimension must have the same shape.

```
a = np.arange(12).reshape(3,4);a
```

```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

```
[11] i = np.array( [ [0,1],                    # indices for the first dim of a
                     [1,2] ] )
     i
```

```
array([[0, 1],
       [1, 2]])
```

```
[12] j = np.array( [ [2,1],                    # indices for the second dim
                     [3,3] ] )
     j
```

```
array([[2, 1],
       [3, 3]])
```

```
[13] a[i,j]                                    # i and j must have equal shape
```

```
array([[ 2,  5],
       [ 7, 11]])
```

```
a[i,2]
```

```
array([[ 2,  6],
       [ 6, 10]])
```

```
[16] a[:,j]                                    # i.e., a[ : , j]
```

```
array([[[ 2,  1],
        [ 3,  3]],

       [[ 6,  5],
        [ 7,  7]],

       [[10,  9],
        [11, 11]]])
```

Naturally, we can put i and j in a sequence (say a list) and then do the indexing with the list.

```
[17] l = [i,j]
     l
```

```
[array([[0, 1],
        [1, 2]]),
 array([[2, 1],
        [3, 3]])]
```

```
[18] a[l]
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-18-220ac7e1225b> in <cell line: 1>()
----> 1 a[l]

IndexError: index 3 is out of bounds for axis 0 with size 3
```

SEARCH STACK OVERFLOW

Indexing with Boolean Arrays

When we index arrays with arrays of (integer) indices we are providing the list of indices to pick. With boolean indices the approach is different; we explicitly choose which items in the array we want and which ones we don't.

The most natural way one can think of for boolean indexing is to use boolean arrays that have the same shape as the original array:

```
a = np.arange(12).reshape(3,4)
b = a > 4
b
```

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

```
a[b]  # 1d array with the selected elements
```

```
array([ 5,  6,  7,  8,  9, 10, 11])
```

The ix_() function

The ix_ function can be used to combine different vectors so as to obtain the result for each n-uplet. For example, if you want to compute all the a+b*c for all the triplets taken from each of the vectors a, b and c:

```
a = np.array([2,3,4,5])
b = np.array([8,5,4])
c = np.array([5,4,6,8,3])
ax,bx,cx = np.ix_(a,b,c)
print(ax)
```

```
[[[2]]

 [[3]]

 [[4]]

 [[5]]]
```

```
[24] cx
```

```
array([[[5, 4, 6, 8, 3]]])
```

```
[25] bx
```

```
array([[[8],
        [5],
        [4]]])
```

```
[26] ax.shape, bx.shape, cx.shape
```

```
((4, 1, 1), (1, 3, 1), (1, 1, 5))
```

```
result = ax+bx*cx
result
```

```
array([[[42, 34, 50, 66, 26],
        [27, 22, 32, 42, 17],
        [22, 18, 26, 34, 14]],

       [[43, 35, 51, 67, 27],
        [28, 23, 33, 43, 18],
        [23, 19, 27, 35, 15]],

       [[44, 36, 52, 68, 28],
        [29, 24, 34, 44, 19],
        [24, 20, 28, 36, 16]],

       [[45, 37, 53, 69, 29],
        [30, 25, 35, 45, 20],
        [25, 21, 29, 37, 17]]])
```

```
[28] result[3,2,4]
```

```
17
```

```
[29] a[3]+b[2]*c[4]
```

```
17
```

You could also implement the reduce as follows:

```
[32] def ufunc_reduce(ufct, *vectors):
         vs = np.ix_(*vectors)
         r = ufct.identity
         for v in vs:
             r = ufct(r,v)
         return r
```

```
[33] ufunc_reduce(np.add,a,b,c)
```

```
array([[[15, 14, 16, 18, 13],
        [12, 11, 13, 15, 10],
        [11, 10, 12, 14,  9]],

       [[16, 15, 17, 19, 14],
        [13, 12, 14, 16, 11],
        [12, 11, 13, 15, 10]],

       [[17, 16, 18, 20, 15],
        [14, 13, 15, 17, 12],
        [13, 12, 14, 16, 11]],

       [[18, 17, 19, 21, 16],
        [15, 14, 16, 18, 13],
        [14, 13, 15, 17, 12]]])
```

Numpy Array Indexing and Slicing

Numpy Array Indexing

NumPy array indexing and slicing are powerful features that allow you to access and manipulate elements of arrays. Here's an overview of NumPy array indexing and slicing:

- Accessing an array element is the same as using an array index.
- An array element's index number can be used to access it.
- NumPy arrays' indexes begin with 0, therefore the first element has index 0, the second has index 1, etc.

1. Single Element Indexing:

Indexing allows you to access individual elements of a NumPy array.

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5])
# Accessing the element at index 2
element_at_index_2 = arr[2]
print(element_at_index_2)
# Output: 3
```

```
import numpy as np

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

# Accessing the element at row 1, column 2
element_at_row1_col2 = arr[1, 2]
print(element_at_row1_col2)
# Output: 6
```

```
import numpy as np
arr = np.array([10, 20, 30, 40, 50])
# Accessing the last element using negative indexing
last_element = arr[-1]
print(last_element)
# Output: 50
```

2. Multi-dimensional Indexing:

For multi-dimensional arrays, you can use multiple indices to access specific elements.

```
import numpy as np
arr_2d = np.array([[1, 2, 3], [4, 5, 6]])
# Accessing the element at row 1, column 2
element_at_row1_col2 = arr_2d[1, 2]
print(element_at_row1_col2)
# Output: 6
```

3. Boolean Indexing:
You can use boolean arrays to filter elements based on certain conditions.

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5])
# Creating a boolean mask
mask = arr > 2
# Using the boolean mask to get elements greater than 2
filtered_elements = arr[mask]
print(filtered_elements)
# Output: [3 4 5]
```

```
import numpy as np

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

# Creating a boolean mask for even numbers
even_mask = arr % 2 == 0
```

```
# Using the boolean mask to get even elements
even_elements = arr[even_mask]
print(even_elements)
# Output: [10 20 30 40 50]

import numpy as np
arr = np.array([1, 2, 3, 4, 5])
# Creating a boolean mask for elements equal to 3
equal_to_3_mask = arr == 3
# Using the boolean mask to get elements equal to 3
elements_equal_to_3 = arr[equal_to_3_mask]
print(elements_equal_to_3)
# Output: [3]
```

In this example, we create a boolean mask (mask) where each element is True if the corresponding element in arr is greater than 2. The resulting array contains only elements greater than 2.

Numpy Array Slicing:

- Python's term for extracting elements from one given index and adding them to another is "slicing."
- Instead of passing an index, we use the format [start:end].
- The step can alternatively be defined in the following format: [start:end:step].
- If we fail to pass start, it is regarded as 0.
- If we don't pass the end, the array's length in that dimension is considered.
- If we fail step, it counts as 1.

4. Basic Slicing:
Slicing allows you to extract a range of elements from an array.

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5])
# Slicing from index 1 to 3 (exclusive)
sliced_array = arr[1:3]
print(sliced_array)
# Output: [2 3]
```

```
import numpy as np
arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
# Slicing the first two rows
sliced_2d_array = arr[:2, :]
print(sliced_2d_array)
# Output: [[1 2 3]
#          [4 5 6]]
```

```
import numpy as np
arr = np.array([10, 20, 30, 40, 50])
# Slicing with a step of 2
strided_slice = arr[1:5:2]
print(strided_slice)
# Output: [20 40]
```

5. Multi-dimensional Slicing:

For multi-dimensional arrays, you can use slicing along multiple dimensions

```
import numpy as np
arr_2d = np.array([[1, 2, 3], [4, 5, 6]])
# Slicing rows from index 0 to 1, and columns from index 1 to 2
sliced_2d_array = arr_2d[0:2, 1:3]
print(sliced_2d_array)
# Output: [[2 3]
#          [5 6]]
```

```
import numpy as np
arr_3d = np.random.rand(2, 3, 4)
# Slicing the first dimension (2D array) at index 1
sliced_2d_array = arr_3d[1, :, :]
print(sliced_2d_array)
# Output: A slice of the 2D array at index 1 along the first
dimension
```

```
import numpy as np
arr_3d = np.random.rand(2, 3, 4)
# Slicing along both dimensions
sliced_array = arr_3d[:, 1:3, 2]
print(sliced_array)
# Output: A slice along the first dimension and a range along the
second dimension
```

6. Strided Slicing:

Strided slicing allows you to skip elements while slicing.

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6])
# Slicing with a step of 2
strided_slice = arr[1:5:2]
print(strided_slice)
# Output: [2 4]
```

```
import numpy as np
arr = np.array([10, 20, 30, 40, 50])
# Strided slicing with a step of 2
strided_slice = arr[::2]
print(strided_slice)
# Output: [10 30 50]
```

```
import numpy as np
arr_2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
# Strided slicing along rows with a step of 2
strided_slice = arr_2d[::2, :]
print(strided_slice)
# Output: [[1 2 3]
#          [7 8 9]]
```

7. Conditional Slicing:

Conditional slicing allows you to use boolean conditions to filter elements from an array.

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5])
# Slicing elements greater than 2
condition = arr > 2
conditional_slice = arr[condition]
print(conditional_slice)
# Output: [3 4 5]
```

```
import numpy as np
arr = np.array([10, 20, 30, 40, 50])
# Slicing elements divisible by 20
condition = arr % 20 == 0
conditional_slice = arr[condition]
print(conditional_slice)
# Output: [20 40]
```

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5])
# Slicing elements not equal to 3
condition = arr ≠ 3
conditional_slice = arr[condition]
print(conditional_slice)
# Output: [1 2 4 5]
```

Here, we create a boolean condition where each element is True if it is greater than 2. The resulting array contains only elements that satisfy this condition.

8. Ellipsis (...) Operator:

The ellipsis (...) operator can be used for more concise slicing in higher-dimensional arrays.

```
import numpy as np
arr_3d = np.random.rand(2, 3, 4)
# Slicing using ellipsis to access the entire array along the
last dimension
sliced_3d_array = arr_3d[..., 2]
print(sliced_3d_array)
# Output: A slice of the third column along the last dimension
for all 2D matrices
```

```
import numpy as np
arr_4d = np.random.rand(2, 3, 4, 5)
# Slicing using ellipsis to access the entire array along all
dimensions except the last
sliced_4d_array = arr_4d[..., 2]
print(sliced_4d_array)
# Output: A slice of the third column along the last dimension
for all 3D arrays
```

```
import numpy as np

arr_3d = np.random.rand(2, 3, 4)

# Slicing using ellipsis to access the entire array
sliced_3d_array = arr_3d[...]
print(sliced_3d_array)
# Output: The entire 3D array
```

Negative Slicing

Use the minus operator to refer to an index from the end:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7])
print(arr[-3:-1]) #Output = [5,6]
```

Iterating over NumPy arrays

Iterating over NumPy arrays can be done using various approaches, and the choice of method depends on the specific requirements of your task. NumPy provides efficient ways to iterate over elements, rows, or columns of an array.

1. Iterating Over Elements:
You can use a nested loop to iterate over each element in a NumPy array.

```
import numpy as np

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

# Iterate over each element
```

```
for row in arr:
    for element in row:
        print(element)
```

```
# Using vectorized operation with NumPy functions
result = np.sin(arr)
print(result)
# Output:
# [[ 0.84147098  0.90929743  0.14112001]
#  [-0.7568025  -0.95892427 -0.2794155 ]]
```

```
# Using element-wise condition
condition = arr % 2 == 0
result = arr[condition]
print(result)
# Output: [2 4 6]
```

```
# Using NumPy's unravel_index to get indices for specific values
target_values = [2, 5]
indices = np.unravel_index(np.where(np.isin(arr, target_values)),
arr.shape)
print(indices)
# Output: (array([0, 1]), array([1, 1]))
```

Alternatively, you can use the flat attribute to iterate over each element directly

```
for element in arr.flat:
    print(element)
```

2. Iterating Over Rows: If you want to iterate over rows, you can use a simple loop.

```
for row in arr:
    print(row)
```

```
# Iterating over rows and calculating the row sum
for i, row in enumerate(arr):
    row_sum = np.sum(row)
    print(f"Row {i+1} Sum: {row_sum}")
# Output:
# Row 1 Sum: 6
# Row 2 Sum: 15
```

```
# Iterating over rows and applying a custom function
def custom_function(row):
    return row.mean()

for i, row in enumerate(arr):
    result = custom_function(row)
    print(f"Result for Row {i+1}: {result}")
# Output:
# Result for Row 1: 2.0
# Result for Row 2: 5.0
```

```
# Iterating over rows and finding the minimum value in each row
for i, row in enumerate(arr):
    min_value = np.min(row)
    print(f"Minimum value in Row {i+1}: {min_value}")
# Output:
# Minimum value in Row 1: 1
# Minimum value in Row 2: 4
```

```
# Iterating over rows and checking if all elements are positive
for i, row in enumerate(arr):
    all_positive = np.all(row > 0)
    print(f"All elements in Row {i+1} are positive:
{all_positive}")
# Output:
# All elements in Row 1 are positive: True
# All elements in Row 2 are positive: True
```

3. Iterating Over Columns: To iterate over columns, you can use the transpose function or the T attribute.

```
# Using transpose
for col in arr.transpose():
    print(col)

# Using T attribute
for col in arr.T:
    print(col)
```

```
# Iterating over columns and checking if any element is greater
than 5
for i, col in enumerate(arr.T):
    any_greater_than_5 = np.any(col > 5)
    print(f"Any element in Column {i+1} greater than 5:
{any_greater_than_5}")
# Output:
# Any element in Column 1 greater than 5: False
# Any element in Column 2 greater than 5: False
# Any element in Column 3 greater than 5: True
```

```
# Iterating over columns and applying a custom function
def custom_function(col):
    return col.sum()

for i, col in enumerate(arr.T):
    result = custom_function(col)
    print(f"Result for Column {i+1}: {result}")
# Output:
# Result for Column 1: 5
# Result for Column 2: 7
# Result for Column 3: 9
```

4. Using ndenumerate and ndindex: NumPy provides np.ndenumerate and np.ndindex functions, which are helpful for iterating over array indices.

```
for index, value in np.ndenumerate(arr):
    print(f"Index: {index}, Value: {value}")

# Using ndindex to iterate over indices
for index in np.ndindex(arr.shape):
    print(f"Index: {index}, Value: {arr[index]}")
```

```
# Using np.ndenumerate with element-wise condition
for index, value in np.ndenumerate(arr):
    if value % 2 == 0:
        print(f"Index: {index}, Even Value: {value}")
# Output:
# Index: (0, 1), Even Value: 2
# Index: (0, 2), Even Value: 3
# Index: (1, 0), Even Value: 4
# Index: (1, 1), Even Value: 5
# Index: (1, 2), Even Value: 6
```

```
# Using np.ndindex to iterate over indices and modify the array
for index in np.ndindex(arr.shape):
    arr[index] = arr[index] ** 2
print(arr)
# Output:
# [[ 1  4  9]
#  [16 25 36]]
```

```
# Using np.ndenumerate with a custom function
def custom_function(index, value):
    return f"Index: {index}, Squared Value: {value ** 2}"

for index, value in np.ndenumerate(arr):
    result = custom_function(index, value)
    print(result)
# Output:
# Index: (0, 0), Squared Value: 1
# Index: (0, 1), Squared Value: 4
# Index: (0, 2), Squared Value: 9
# Index: (1, 0), Squared Value: 16
# Index: (1, 1), Squared Value: 25
# Index: (1, 2), Squared Value: 36
```

```
# Using np.ndindex to iterate over indices and calculate the
cumulative sum
cumulative_sum = np.zeros(arr.shape)
for index in np.ndindex(arr.shape):
    cumulative_sum[index] = np.sum(arr[:index[0]+1, :index[1]+1])
print(cumulative_sum)
# Output:
# [[ 1.  5. 14.]
#  [17. 42. 78.]]
```

5. Vectorized Operations: NumPy is designed to perform vectorized operations, and it's generally more efficient to use them than explicit iteration when possible. This avoids the need for explicit loops in Python and leverages optimized C and Fortran code.

```
# Vectorized addition
result = arr + 10
print(result)
```

```
# Vectorized operation with NumPy functions
result = np.cos(arr)
print(result)
# Output:
# [[ 0.54030231 -0.65364362 -0.91113026]
#  [-0.75968791  0.28366219  0.75390225]]
```

```
# Vectorized element-wise condition
condition = arr > 10
result = np.where(condition, arr, 0)
print(result)
# Output:
# [[ 0  0  0]
#  [16 25 36]]
```

```
# Vectorized operation with a custom function
def custom_function(x):
    return x / 2

result = np.vectorize(custom_function)(arr)
print(result)
# Output:
# [[0.5 2.  4.5]
#  [8.  12.5 18. ]]
```

```
# Vectorized operation with NumPy functions
result = np.exp(arr)
print(result)
# Output:
# [[2.71828183e+00 5.45981500e+01 8.10308393e+03]
#  [8.88611052e+06 7.20048993e+10 4.31123155e+15]]
```

6. Using apply_along_axis: The apply_along_axis function can be used for element-wise operations along a specific axis.

```
def my_function(row):
    # Your custom function
    return row * 2

result = np.apply_along_axis(my_function, axis=1, arr=arr)
print(result)
```

Calculate Row Sum

```
import numpy as np

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

# Define a function to calculate row sum
def row_sum(row):
    return np.sum(row)

# Apply the function along the rows (axis=1)
result = np.apply_along_axis(row_sum, axis=1, arr=arr)
print(result)
# Output: [ 6 15]
```

Normalize Each Column

```
# Define a function to normalize each column
def normalize_column(col):
    return col / np.sum(col)

# Apply the function along the columns (axis=0)
result = np.apply_along_axis(normalize_column, axis=0, arr=arr)
print(result)
# Output:
# [[0.2 0.28571429 0.33333333]
#  [0.8 0.71428571 0.66666667]]
```

Calculate Moving Average Along Rows

```
# Define a function to calculate the moving average along rows
def moving_average(row):
    return np.convolve(row, np.ones(3) / 3, mode='valid')

# Apply the function along the rows (axis=1)
result = np.apply_along_axis(moving_average, axis=1, arr=arr)
print(result)
# Output:
# [[2. 3.]
#  [5. 6.]]
```

Custom Operation on Each Column

```
# Define a function to perform a custom operation on each column
def custom_operation(col):
    return np.prod(col)

# Apply the function along the columns (axis=0)
result = np.apply_along_axis(custom_operation, axis=0, arr=arr)
print(result)
# Output: [4 10 18]
```

Numpy array manipulation

NumPy array manipulation refers to a set of operations and functions provided by the NumPy library for modifying the structure, shape, and content of arrays. These operations are essential for tasks such as reshaping arrays, concatenating or splitting arrays, adding or removing elements, and more. Array manipulation in NumPy is crucial for efficiently working with numerical data and performing various mathematical and scientific computations.

Here are some key aspects of NumPy array manipulation:

Reshaping Arrays:

- The reshape function is used to change the shape of an array.
- Reshaping can convert a 1D array to a multi-dimensional array or change the dimensions of an existing array.

Flattening Arrays:

- Flattening refers to converting a multi-dimensional array into a 1D array.
- The flatten and ravel functions are commonly used for this purpose.

Concatenation:

- Concatenation combines multiple arrays along a specified axis.
- Functions like concatenate, hstack, vstack, stack, etc., are used for concatenation.

Splitting Arrays:

- Splitting involves dividing an array into multiple sub-arrays along a specified axis.
- Functions like split, hsplit, vsplit, array_split, etc., are used for splitting.

Adding/Removing Elements:

- Operations like append, insert, and delete are used to add or remove elements from arrays.
- These operations modify the size and content of the arrays.

Transposition:

- Transposition swaps the axes of an array, effectively switching rows with columns.
- The transpose function or the T attribute can be used for transposition.

Changing Data Type:

- NumPy allows changing the data type of an array using functions like astype.
- Changing data types can be important for numerical precision and compatibility.

Vectorization:

- Vectorized operations enable performing element-wise operations on entire arrays without explicit looping.
- This leads to more concise and efficient code.

NumPy's array manipulation capabilities are crucial for working with large datasets, numerical simulations, and scientific computing. These operations provide a high level of flexibility and efficiency when handling numerical data in a Python environment.

1. Reshaping Arrays:

Reshaping changes the shape of an array, either by modifying the shape attribute or using specific functions like reshape.

```
import numpy as np
arr = np.arange(1, 10)
# Reshape 1D array to a 2D array
reshaped_arr = arr.reshape((3, 3))
print(reshaped_arr)
# Output:
# [[1 2 3]
#  [4 5 6]
#  [7 8 9]]
```

```
# Reshape array using newaxis
newaxis_arr = arr[:, np.newaxis]
print(newaxis_arr)
# Output:
# [[1]
#  [2]
#  [3]
#  [4]
#  [5]
#  [6]
#  [7]
#  [8]
#  [9]]
```

```
# Reshape array using -1 for automatic dimension inference
auto_dim_arr = arr.reshape((3, -1))
print(auto_dim_arr)
# Output:
# [[1 2 3]
#  [4 5 6]
#  [7 8 9]]
```

```
# Reshape and order array in Fortran (column-major) style
fortran_arr = np.array(arr, order='F').reshape((3, 3))
print(fortran_arr)
# Output:
# [[1 4 7]
#  [2 5 8]
#  [3 6 9]]
```

```
# Reshape array and flatten it in one step
flat_and_reshaped = np.ravel(arr.reshape((3, 3)))
print(flat_and_reshaped)
# Output: [1 2 3 4 5 6 7 8 9]
```

2. Flattening Arrays:

Flattening an array means converting a multi-dimensional array into a 1D array using the flatten or ravel function.

```
import numpy as np

arr = np.array([[1, 2, 3], [4, 5, 6]])
flattened_arr = reshaped_arr.flatten()
print(flattened_arr)
# Output: [1 2 3 4 5 6 7 8 9]
```

```
# Flatten array using ravel() method
raveled_arr = np.ravel(arr)
```

```
print(raveled_arr)
# Output: [1 2 3 4 5 6]
```

```
# Flatten array and modify original array
arr_flat = arr.flatten()
arr[0, 0] = 99
print(arr_flat)
# Output: [1 2 3 4 5 6]
```

```
# Flatten array using order parameter (Fortran style)
fortran_flatten = np.array(arr, order='F').flatten()
print(fortran_flatten)
# Output: [99 4 2 5 3 6]
```

```
# Flatten array using order parameter (C style)
c_style_flatten = np.array(arr, order='C').flatten()
print(c_style_flatten)
# Output: [99 2 3 4 5 6]
```

4. Concatenation:

Concatenation combines multiple arrays along an existing axis using functions like concatenate, hstack, or vstack.

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

# Concatenate along axis 0 (vertically)
concatenated_arr = np.concatenate((arr1, arr2), axis=0)
print(concatenated_arr)
# Output:
# [[1 2]
#  [3 4]
#  [5 6]]
```

```
# Concatenate along axis 1 (horizontally)
concatenated_arr_h = np.concatenate((arr1, arr2.T), axis=1)
print(concatenated_arr_h)
# Output:
# [[1 2 5]
#  [3 4 6]]
```

```
# Concatenate using vstack
vstack_arr = np.vstack((arr1, arr2))
print(vstack_arr)
# Output:
# [[1 2]
#  [3 4]
#  [5 6]]
```

```
# Concatenate using hstack
hstack_arr = np.hstack((arr1, arr2.T))
print(hstack_arr)
# Output:
# [[1 2 5]
#  [3 4 6]]
```

4. Splitting Arrays:

Splitting divides an array into multiple sub-arrays along a specified axis using functions like split, hsplit, or vsplit.

```
arr3 = np.array([[1, 2], [3, 4], [5, 6]])

# Split along axis 0
split_arr = np.split(arr3, [1])
print(split_arr)
# Output: [array([[1, 2]]), array([[3, 4],
#                                  [5, 6]])]
```

```
# Split along axis 1
split_arr_h = np.split(arr3, [1], axis=1)
print(split_arr_h)
```

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

# Split into equal-sized subarrays
equal_split = np.array_split(arr3, 2)
print(equal_split)
# Output: [array([[1, 2],
#                 [3, 4]]), array([[5, 6]])]
```

```
# Split horizontally using hsplit
hsplit_arr = np.hsplit(arr3, 2)
print(hsplit_arr)
# Output: [array([[1],
#                 [3],
#                 [5]]), array([[2],
#                               [4],
#                               [6]])]
```

```
# Split vertically using vsplit
vsplit_arr = np.vsplit(arr3, 3)
print(vsplit_arr)
# Output: [array([[1, 2]]), array([[3, 4]]), array([[5, 6]])]
```

5. Adding/Removing Elements:

Functions like append, insert, and delete allow adding or removing elements from an array.

```
arr4 = np.array([1, 2, 3])

# Append elements to the end
appended_arr = np.append(arr4, [4, 5])
print(appended_arr)
# Output: [1 2 3 4 5]

# Insert elements at a specific position
inserted_arr = np.insert(arr4, 1, [8, 9])
print(inserted_arr)
```

```
# Output: [1 8 9 2 3]

# Delete elements at specific indices
deleted_arr = np.delete(arr4, [0, 2])
print(deleted_arr)
# Output: [2]
```

6. Transposition:

Transposition switches the axes of an array, effectively swapping rows with columns, using functions like transpose or the T attribute.

```
arr5 = np.array([[1, 2, 3], [4, 5, 6]])

# Transpose the array
transposed_arr = arr5.transpose()
print(transposed_arr)
# Output:
# [[1 4]
#  [2 5]
#  [3 6]]
```

```
# Create a 1D array
arr_1d = np.array([1, 2, 3])

# Transpose the 1D array (effectively converting it to a 2D
column vector)
transposed_1d_arr = arr_1d[:, np.newaxis]

print("Original 1D Array:")
print(arr_1d)
# Output: [1 2 3]

print("\nTransposed 1D Array:")
print(transposed_1d_arr)
# Output:
# [[1]
#  [2]
#  [3]]
```

```
# Create a more complex 2D array
complex_arr = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

# Transpose the array
transposed_complex_arr = np.transpose(complex_arr)

print("Original Complex Array:")
print(complex_arr)
# Output:
# [[1 2 3]
#  [4 5 6]
#  [7 8 9]]

print("\nTransposed Complex Array:")
print(transposed_complex_arr)
# Output:
# [[1 4 7]
#  [2 5 8]
#  [3 6 9]]
```

Changing Data Type:
Changing Data Type of an Array

```
import numpy as np

# Create an array with integers
int_array = np.array([1, 2, 3, 4, 5])

# Change the data type to float
float_array = int_array.astype(float)

print("Original Array with Integer Type:")
print(int_array)
# Output: [1 2 3 4 5]

print("\nArray after Changing Data Type to Float:")
print(float_array)
# Output: [1. 2. 3. 4. 5.]
```

Changing Data Type for Precision

```
# Create an array with floating-point numbers
float_array = np.array([1.23, 2.45, 3.67, 4.89])

# Change the data type to integers for memory efficiency
int_array = float_array.astype(int)

print("Original Array with Float Type:")
print(float_array)
# Output: [1.23 2.45 3.67 4.89]

print("\nArray after Changing Data Type to Integer:")
print(int_array)
# Output: [1 2 3 4]
```

Changing Data Type for Compatibility

```
# Create an array with integers
int_array = np.array([1, 2, 3, 4, 5])

# Change the data type to complex numbers
complex_array = int_array.astype(complex)

print("Original Array with Integer Type:")
print(int_array)
# Output: [1 2 3 4 5]

print("\nArray after Changing Data Type to Complex:")
print(complex_array)
# Output: [1.+0.j 2.+0.j 3.+0.j 4.+0.j 5.+0.j]
```

Vectorization:
Vectorized Addition

```
import numpy as np

# Create two arrays for vectorized addition
arr1 = np.array([1, 2, 3, 4])
arr2 = np.array([5, 6, 7, 8])
```

```
# Perform vectorized addition
result = arr1 + arr2

print("Array 1:")
print(arr1)
# Output: [1 2 3 4]

print("\nArray 2:")
print(arr2)
# Output: [5 6 7 8]

print("\nResult after Vectorized Addition:")
print(result)
# Output: [ 6  8 10 12]
```

Vectorized Multiplication

```
# Create two arrays for vectorized multiplication
arr3 = np.array([1, 2, 3, 4])
arr4 = np.array([2, 3, 4, 5])

# Perform vectorized multiplication
result_mul = arr3 * arr4

print("Array 3:")
print(arr3)
# Output: [1 2 3 4]

print("\nArray 4:")
print(arr4)
# Output: [2 3 4 5]

print("\nResult after Vectorized Multiplication:")
print(result_mul)
# Output: [ 2  6 12 20]
```

Vectorized Exponential Operation

```
# Create an array for vectorized exponential operation
arr5 = np.array([1, 2, 3, 4])

# Perform vectorized exponential operation
result_exp = np.exp(arr5)

print("Original Array for Exponential Operation:")
```

```
print(arr5)
# Output: [1 2 3 4]

print("\nResult after Vectorized Exponential Operation:")
print(result_exp)
# Output: [ 2.71828183  7.3890561  20.08553692 54.59815003]
```

Binary operators

Binary operators in NumPy are operations that involve two arrays, performing element-wise operations between corresponding elements of the arrays. These operators take two input arrays and produce a new array as output, applying the specified operation to each pair of corresponding elements. The term "binary" refers to the fact that these operators operate on two operands.

Here are some commonly used binary operators in NumPy:

- **Addition (+):** Adds corresponding elements of two arrays.
- **Subtraction (-):** Subtracts corresponding elements of the second array from the first.
- **Multiplication (*):** Multiplies corresponding elements of two arrays.
- **Division (/):** Divides corresponding elements of the first array by the second.
- **Floor Division (//):** Performs floor division on corresponding elements.
- **Exponentiation (**):** Raises the elements of the first array to the power of the corresponding elements in the second array.
- **Modulo (%):** Computes the element-wise remainder of the division.

Addition (+):

```
import numpy as np

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

result = arr1 + arr2
print(result)  # Output: [5, 7, 9]


arr3 = np.array([10, 20, 30])
arr4 = np.array([40, 50, 60])
result2 = arr3 + arr4  # Result: [50, 70, 90]
print(result2)
```

```
arr5 = np.array([2.5, 3.5, 4.5])
arr6 = np.array([1.5, 2.5, 3.5])
result3 = arr5 + arr6  # Result: [4.0, 6.0, 8.0]
print(result3)
```

```
arr7 = np.array([-1, -2, -3])
arr8 = np.array([1, 2, 3])
result4 = arr7 + arr8  # Result: [0, 0,
print(result4)
```

Subtraction (-):

```
import numpy as np

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

result = arr1 - arr2
print(result)  # Output: [3, 3, 3]
```

```
result2 = arr3 - arr4  # Result: [-30, -30, -30]
print(result2)
```

```
result3 = arr5 - arr6  # Result: [1.0, 1.0, 1.0]
print(result3)
```

```
result4 = arr7 - arr8  # Result: [-2, -4, -6]
print(result4)
```

Multiplication (*):

```
import numpy as np

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

result = arr1 * arr2
print(result)  # Output: [4, 10, 18]
```

```
result2 = arr3 * arr4  # Result: [400, 1000, 1800]
print(result2)
```

```
result3 = arr5 * arr6  # Result: [3.75, 8.75, 15.75]
print(result3)
```

```
result4 = arr7 * arr8  # Result: [-1, -4, -9]
print(result4)
```

Division (/):

```
import numpy as np

arr1 = np.array([4, 6, 8])
arr2 = np.array([2, 3, 2])

result = arr1 / arr2
print(result)  # Output: [2.0, 2.0, 4.0]
```

```
result2 = arr3 / arr4
print(result2)
```

```
result3 = arr5 / arr6
print(result3)
```

```
result4 = arr7 / arr8
print(result4)
```

Floor Division (//):

```
import numpy as np

arr1 = np.array([5, 7, 10])
arr2 = np.array([2, 3, 3])

result = arr1 // arr2
print(result)  # Output: [2, 2, 3]
```

```
result2 = arr3 // arr4  # Result: [0, 0, 0]

print(result2)
```

```
result3 = arr5 // arr6  # Result: [1.0, 1.0, 1.0]
print(result3)
```

```
result4 = arr7 // arr8  # Result: [-1, -1, -1]
print(result4)
```

Exponentiation (**):

```
import numpy as np

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

result = arr ** 2
print(result)  # Output: [4, 9, 16]
```

```
result2 = arr3 ** arr4
print(result2)
```

```
result3 = arr5 ** arr6
print(result3)
```

```
result4 = arr7 ** 3
print(result4)
```

Modulo (%):

```
import numpy as np

arr1 = np.array([7, 10, 15])
arr2 = np.array([3, 4, 7])

result = arr1 % arr2
print(result)  # Output: [1, 2, 1]
```

```
result2 = arr3 % arr4  # Result: [10, 20, 30]
print(result2)
```

```
result3 = arr5 % arr6  # Result: [0.5, 1.0, 1.0]
print(result3)
```

```
result4 = arr7 % arr8  # Result: [0, 0, 0]
print(result4)
```

Mathematical Functions

Numpy consists of various mathematical functions that can be used. Following are some of the functions:

- **Trigonometric functions:**

Compute Sine	numpy.sin()
Compute Cosine	numpy.cos()
numpy.cos()Compute Tangent	numpy.tan()
Compute Inverse Sine	numpy.arcsin()
Compute Inverse Cosine	numpy.arccos()
Compute Inverse Tangent	numpy.arctan()
Convert angles (radians to degrees)	numpy.degrees()
Convert angles (degrees to radians)	numpy.radians()

- **Rounding functions:**

Evenly round to the given number of decimals.	numpy.round()
Round an array to the given number of decimals.	numpy.around()
Round elements of the array to the nearest integer.	numpy.rint()
Round to nearest integer towards zero.	numpy.fix()
Return the floor of the input, element-wise.	numpy.floor()
Return the ceiling of the input, element-wise.	numpy.ceil()

- **Sum, Products function:**

Product of array elements over a given axis.	numpy.prod()
Sum of array elements over a given axis.	numpy.sum()
Product of array elements over a given axis treating numpy.NaN as ones.	numpy.nanprod()
Sum of array elements over a given axis treating numpy.NaN as zero.	numpy.nansum()
Calculate the n-th discrete difference along the given axis.	numpy.diff()
Return the cross product of two (arrays of) vectors.	numpy.cross()

- **Arithmetic functions:**

Add arguments element-wise.	numpy.add()
Return the reciprocal of the argument, element-wise.	numpy.reciprocal()
Multiply arguments element-wise.	numpy.multiply()
Divide arguments element-wise.	numpy.divide()
Subtract arguments, element-wise.	numpy.subtract()
Return element-wise quotient and remainder simultaneously.	numpy.divmod()

You can refer the following link for reference: **NumPY mathematical functions**