NumPy(Numerical Python)

NumPy(Numerical Python) is a fundamental library for Python **numerical computing**. It provides efficient multi-dimensional array objects and various mathematical functions for handling large datasets making it a critical tool for professionals in fields that require heavy computation.

NumPy (Numerical Python) is a **powerful library** for numerical computing in Python. It is mainly used for **working with arrays, matrices, and performing mathematical operations** efficiently.

Installing NumPy in Python

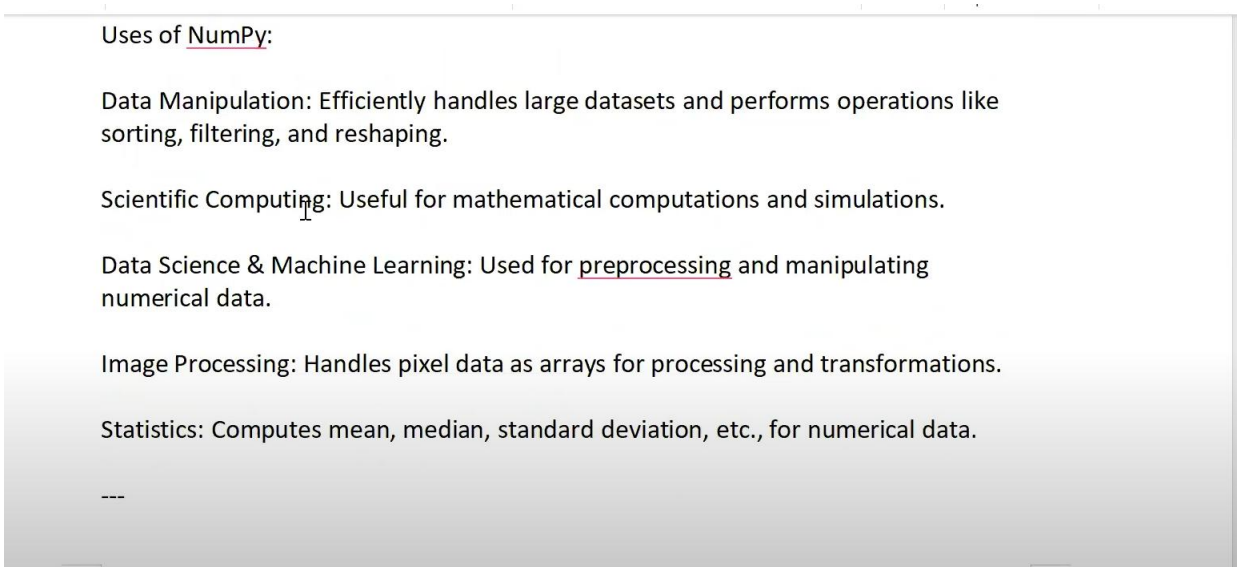
To begin using NumPy, you need to install it first. This can be done through pip command:

pip install numpy

Key Features of NumPy

NumPy has various features that make it popular over lists.

- **N-Dimensional Arrays**: NumPy's core feature is ndarray, a powerful N-dimensional array object that supports homogeneous data types.
- **Arrays with High Performance**: Arrays are stored in contiguous memory locations, enabling faster computations than Python lists (Please see Numpy Array vs Python List for details).
- **Broadcasting:** This allows element-wise computations between arrays of different shapes. It simplifies operations on arrays of **various shapes** by automatically aligning their dimensions without creating new data.
- **Vectorization:** Eliminates the need for explicit Python loops by applying operations directly on entire arrays.
- **Linear algebra**: NumPy contains routines for linear algebra operations, such as matrix multiplication, decompositions, and determinants.



Create a NumPy ndarray Object

NumPy is used to work with arrays. The array object in NumPy is called ndarray.
We can create a NumPy ndarray object by using the array() function.

ex1.py U

Day-16 NumPy > ex1.py > ...

```python
import numpy as np

# Creating a 1D array
x = np.array([1, 2, 3])

# Creating a 2D array
y = np.array([[1, 2], [3, 4]])

# Creating a 3D array
z = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])

print(x)
print(y)
print(z)
```

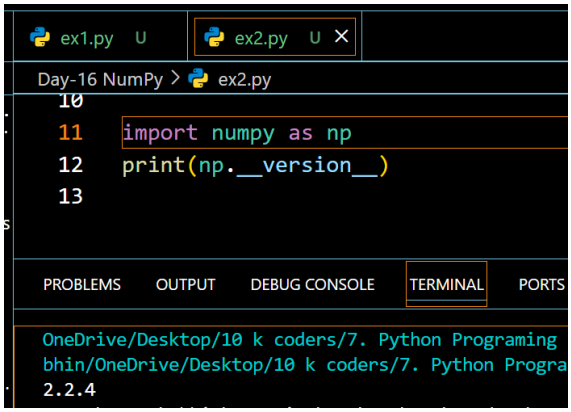
Output

```
[1 2 3]
[[1 2]
 [3 4]]
[[[1 2]
  [3 4]]

 [[5 6]
  [7 8]]]
```

Checking NumPy Version

The version string is stored under __version__ attribute.



Access Array Elements

Array indexing is the same as accessing an array element.

You can access an array element by referring to its index number.

The indexes in NumPy arrays start with 0, meaning that the first element has index 0, and the second has index 1 etc.

Example:

```
import numpy as np
arr = np.array([1, 2, 3, 4])
print(arr[0])    // 1
```

Access 2-D Arrays

To access elements from 2-D arrays we can use comma separated integers representing the dimension and the index of the element.

Think of 2-D arrays like a table with rows and columns, where the dimension represents the row and the index represents the column.

Example

Access the element on the first row, second column:

```
import numpy as np
arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])
print('2nd element on 1st row: ', arr[0, 1])
```

NumPy Array Slicing

Slicing in python means taking elements from one given index to another given index.

We pass slice instead of index like this: [*start*:*end*].
We can also define the step, like this: [*start*:*end*:*step*].

If we don't pass start its considered 0
If we don't pass end its considered length of array in that dimension
If we don't pass step its considered 1

```
import numpy as np

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

print(arr[1:5])
```

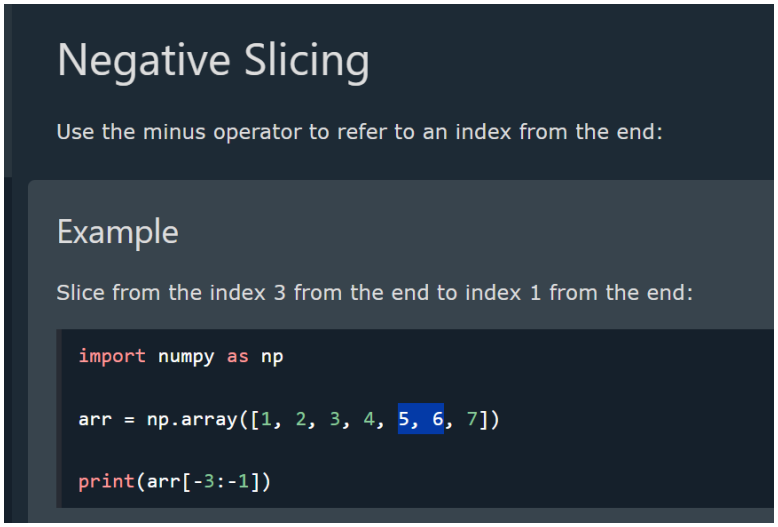
```
[2 3 4 5]
```

```
import numpy as np

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

print(arr[-3:-1])
```

```
[5 6]
```




```
import numpy as np

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

print(arr[1:5:2])
```

```
[2 4]
```

```
import numpy as np

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

print(arr[::2])
```

```
[1 3 5 7]
```

```
#Slicing 2-D Arrays
#Example
#From the second element, slice elements from index 1 to index 4 (not included):

import numpy as np

arr = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]])
print(arr[1, 1:4])
```

```
[7 8 9]
```

Data Types in NumPy

NumPy has some extra data types, and refer 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)

```
import numpy as np

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

print(arr.dtype)
```

```
int64
```

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)
```

```
[b'1' b'2' b'3' b'4']
|S1
```

```
import numpy as np

arr = np.array([1, 2, 3, 4], dtype='i4')

print(arr)
print(arr.dtype)
```

```
[1 2 3 4]
int32
```

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.

```
import numpy as np

arr = np.array([1.1, 2.1, 3.1])

newarr = arr.astype('i')

print(newarr)
print(newarr.dtype)
```

```
[1 2 3]
int32
```

NumPy Array Properties

Day-16 NumPy > array_properties.py > ...

```python
import numpy as np

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

print(arr.shape)  # (2, 3) → 2 rows, 3 columns
print(arr.ndim)   # 2 → Number of dimensions
print(arr.size)   # 6 → Total number of elements
print(arr.dtype)  # int32 → Data type of elements
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
"
(2, 3)
2
6
int64
```

Reshape From 1-D to 2-D

Example

Convert the following 1-D array with 12 elements into a 2-D array.

The outermost dimension will have 4 arrays, each with 3 elements:

```python
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])

newarr = arr.reshape(4, 3)

print(newarr)
```

Reshape From 1-D to 3-D

Example

Convert the following 1-D array with 12 elements into a 3-D array.

The outermost dimension will have 2 arrays that contains 3 arrays, each with 2 elements:

```python
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])

newarr = arr.reshape(2, 3, 2)

print(newarr)
```

```
[[[ 1  2]
  [ 3  4]
  [ 5  6]]

 [[ 7  8]
  [ 9 10]
  [11 12]]]
```

Generating Special Arrays

Day-16 NumPy > special_arrays.py

```python
import numpy as np

print(np.zeros((2, 3)))      # 2x3 matrix filled with 0s
print(np.ones((3, 3)))       # 3x3 matrix filled with 1s
print(np.eye(4))             # 4x4 identity matrix
print(np.arange(1, 10, 2) )  # [1, 3, 5, 7, 9] (like range())
print(np.linspace(0, 5, 10)) # 10 evenly spaced numbers from 0 to 5
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
[[0. 0. 0.]
 [0. 0. 0.]]
[[1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]]
[[1. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 1. 0.]
 [0. 0. 0. 1.]]
[1 3 5 7 9]
[0.         0.55555556 1.11111111 1.66666667 2.22222222 2.77777778
 3.33333333 3.88888889 4.44444444 5.        ]
PS C:\Users\abhin\OneDrive\Desktop\10 k coders\7. Python Programing Language>
```

Array Operations using numPy

Day-16 NumPy > operators_np.py > ...

```python
import numpy as np

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

print(a + b)   # [5 7 9]  → Element-wise addition
print(a - b)   # [-3 -3 -3]  → Element-wise subtraction
print(a * b)   # [4 10 18]  → Element-wise multiplication
print(a / b)   # [0.25 0.4 0.5]  → Element-wise division
print(a ** 2)  # [1 4 9]  → Element-wise exponentiation


A = np.array([[1, 2], [3, 4]])
B = np.array([[5, 6], [7, 8]])
print(A @ B)      # Matrix multiplication
print(A.T)        # Transpose of A
print(np.linalg.inv(A))  # Inverse of A
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
PS C:\Users\abhin\OneDrive\Desktop\10 k coders\7. Python Programing Language> & "c:/Users/abhin/OneDrive/Desktop/10 k coders/7. Python Programing Language/.venv/Scripts/python.exe" "c:/Users/abhin/OneDrive/Desktop/10 k coders/7. Python Programing Language/Day-16 NumPy/operators_np.py"
[5 7 9]
[-3 -3 -3]
[ 4 10 18]
[0.25 0.4  0.5 ]
[1 4 9]
[[19 22]
 [43 50]]
[[1 3]
 [2 4]]
[[-2.   1. ]
[0.25 0.4  0.5 ]
[1 4 9]
[[19 22]
 [43 50]]
[[1 3]
 [2 4]]
[[-2.   1. ]
[1 4 9]
[[19 22]
 [43 50]]
[[1 3]
 [2 4]]
[[-2.   1. ]
[[19 22]
 [43 50]]
[[1 3]
 [2 4]]
[[-2.   1. ]
 [2 4]]
[[-2.   1. ]
[[-2.   1. ]
 [ 1.5 -0.5]]
PS C:\Users\abhin\OneDrive\Desktop\10 k coders\7. Python Programing Language>
```

Logical Operators

NumPy allows **element-wise logical comparisons**.

python Copy Edit

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

print(arr > 25)   # [False False  True  True  True]
print(arr[arr > 25])  # [30 40 50] (Elements greater than 25)
```

- `np.dot()`: Performs matrix multiplication.

```
matrix1 = np.array([[1, 2], [3, 4]])
matrix2 = np.array([[5, 6], [7, 8]])
result = np.dot(matrix1, matrix2)
print(result)
# Output: [[19 22]
#          [43 50]]
```

Aggregate Functions

NumPy provides built-in functions for mathematical operations.

python Copy Edit

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

print(np.sum(arr))    # 15 → Sum of elements
print(np.mean(arr))   # 3.0 → Mean (average)
print(np.median(arr)) # 3.0 → Median
print(np.std(arr))    # 1.41 → Standard deviation
print(np.min(arr))    # 1 → Minimum value
print(np.max(arr))    # 5 → Maximum value
```

NumPy Array Iterating

Iterating means going through elements one by one.
As we deal with multi-dimensional arrays in numpy, we can do this using basic for loop of python.
If we iterate on a 1-D array it will go through each element one by one.

```python
import numpy as np

arr = np.array([1, 2, 3])
for x in arr:
  print(x)
```

```
1
2
3
```

```python
import numpy as np

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

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

```
[1 2 3]
[4 5 6]
```

4. Array Manipulation:
   - `np.reshape()`: Changes the shape of an array.

```
reshaped = np.reshape(data, (1, 5))
```

- `np.concatenate()`: Combines multiple arrays.

```
array3 = np.array([7, 8, 9])
combined = np.concatenate((data, array3))
print(combined)  # Output: [1 2 3 4 5 7 8 9]
```

- `np.split()`: Splits an array into sub-arrays.

```
split_arrays = np.split(data, 5)
print(split_arrays)  # Output: [array([1]), array([2]), ..., array([5])]
```

```
84  list=[1,2,3,4,5,6]
85  # print(np.reshape(list,(2,3)))
86  # print(list)
87  data=np.array(list)
88  # array3 = np.array([7, 8, 9])
89  # combined = np.concatenate((list, array3))
90  # print(combined)
91
92  print(np.split(data,3))
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

[Running] python -u "c:\Users\laksh\OneDrive\Desktop\numpy\num.py"
[array([1, 2]), array([3, 4]), array([5, 6])]
```

Searching Arrays

You can search an array for a certain value, and return the indexes that get a match.

To search an array, use the **where()** method.

```
import numpy as np

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

x = np.where(arr == 4)

print(x)
```

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

Example

Find the indexes where the values are even:

```
import numpy as np

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

x = np.where(arr%2 == 0)

print(x)
```

Search Sorted

There is a method called **searchsorted()** which performs a binary search in the array, and returns the index where the specified value would be inserted to maintain the search order.

```
import numpy as np

arr = np.array([6, 7, 8, 9])

x = np.searchsorted(arr, 7)

print(x)
```

```
1
```

Example

Find the indexes where the value 7 should be inserted, starting from the right:

```
import numpy as np

arr = np.array([6, 7, 8, 9])

x = np.searchsorted(arr, 7, side='right')

print(x)
```

Multiple Values

To search for more than one value, use an array with the specified values.

Example

Find the indexes where the values 2, 4, and 6 should be inserted:

```
import numpy as np

arr = np.array([1, 3, 5, 7])

x = np.searchsorted(arr, [2, 4, 6])

print(x)
```

Sorting Arrays

Sorting means putting elements in an *ordered sequence*.

Ordered sequence is any sequence that has an order corresponding to elements, like numeric or alphabetical, ascending or descending.

The NumPy ndarray object has a function called sort(), that will sort a specified array.

```
import numpy as np

arr = np.array([3, 2, 0, 1])

print(np.sort(arr))
```

```
[0 1 2 3]
```

```
import numpy as np

arr = np.array(['banana', 'cherry', 'apple'])

print(np.sort(arr))
```

```
['apple' 'banana' 'cherry']
```

```python
import numpy as np

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

print(np.sort(arr))
```

```
[[2 3 4]
 [0 1 5]]
```

Filtering Arrays

Getting some elements out of an existing array and creating a new array out of them is called *filtering*.

In NumPy, you filter an array using a *boolean index list*.

If the value at an index is True that element is contained in the filtered array, if the value at that index is False that element is excluded from the filtered array.

Example

Create an array from the elements on index 0 and 2:

```python
import numpy as np

arr = np.array([41, 42, 43, 44])

x = [True, False, True, False]

newarr = arr[x]

print(newarr)
```

The example above will return `[41, 43]`, why?

Random Number Operations

```python
np.random.rand(3, 3)  # 3x3 matrix with random values (0 to 1)
np.random.randint(1, 10, (2, 2))  # Random integers from 1 to 10
```

5. Random Numbers:
  - `np.random.rand()`: Generates random numbers in the range [0, 1].

```
random_array = np.random.rand(3)
print(random_array)  # Output: [0.5488135  0.71518937 0.60276338]
```

  - `np.random.randint()`: Generates random integers within a range.

```
random_integers = np.random.randint(0, 10, 5)
print(random_integers)  # Output: [3 7 9 2 4]
```

  - `np.random.normal()`: Creates a normal distribution.

```
normal_dist = np.random.normal(0, 1, 5)
print(normal_dist)  # Output: [-0.10321885  0.4105985  0.14404357 -1.45427351
0.76103773]
```

```
print(np.random.rand())
otp=""
for i in np.random.randint(1,10,6):
    otp+=str(i)
print(otp)
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS
[Running] python -u "c:\Users\laksh\OneDrive\Desktop\nul
0.44765357711326426
151327
```

Broadcasting in numPy

Broadcasting in NumPy

Broadcasting is a powerful feature in NumPy that allows you to perform operations on arrays of different shapes. It enables arithmetic operations on arrays without explicitly reshaping or replicating them. This feature simplifies coding and makes computations efficient by avoiding unnecessary memory usage.

How Broadcasting Works

When performing operations on two arrays, NumPy compares their shapes element by element. Broadcasting is applied when the following rules are met:

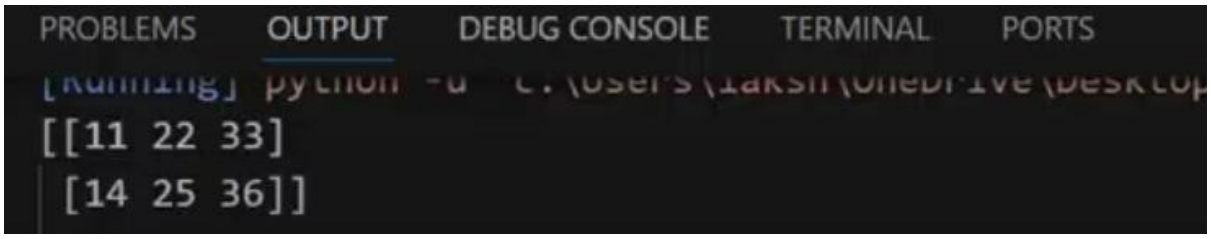
When Broadcasting Fails

Broadcasting fails when the dimensions of the arrays are incompatible. For example:

```
array1 = np.array([1, 2, 3])
array2 = np.array([[1, 2], [3, 4]])
result = array1 + array2  # This will raise a ValueError
```

The shapes (3,) and (2, 2) are incompatible for broadcasting.

```
124    array1 = np.array([[1, 2, 3], [4, 5, 6]])
125    array2 = np.array([10, 20, 30])
126    result = array1 + array2
127    print(result)
```



A scalar value is automatically broadcasted to match the shape of the array.

```
import numpy as np
array = np.array([1, 2, 3])
result = array + 5
print(result)  # Output: [6 7 8]
```

Here, the scalar 5 is "broadcast" across all elements of the array.

Adding Two Arrays of Different Shapes

When one array has a shape (m, n) and the other has a shape (1, n), broadcasting allows the smaller array to expand along the missing dimension.

```
array1 = np.array([[1, 2, 3], [4, 5, 6]])
array2 = np.array([10, 20, 30])
result = array1 + array2
print(result)
# Output:
```