Lab 1.2 – NumPy for Data-Processing

Aim : To understand basics of `Numpy` Python library for Signals and systems Lab.

Completed By ¶

Student Name : Dr. Atul Kumar Dwivedi

Roll Number :

Branch : Electronics and Communication Engineering

Semester : 4

Lab: Signals and Systems (BEC 451)

Date of Completion ....

Numpy

Numpy is a scientific computing tool

Numpy arrays

Numpy arrays offer the following benefits over Python lists for operating on numerical data:

- **Ease of use**: You can write small, concise, and intuitive mathematical expressions.
- **Performance**: Numpy operations and functions are implemented internally in C++, which makes them much faster than using Python statements & loops that are interpreted at runtime Numpy arrays behave like **true numerical vectors**, not ordinary lists. That's why they are used for all mathematical operations, machine learning algorithms, and as basis of Pandas DataFrame for data analytics.

In [1]:
```
from platform import python_version   # show python version
print(python_version())


import numpy as np
print(np.__version__)    # show Numpy version installed
```

In [11]:
```
lst1=[1,2,3,4]   # initialized python list
array1 = np.array(lst1)   #converting list to nd array
```

```
In [12]: type(lst1) # checking data type of list1
Out[12]: list
```

```
In [13]: type(array1) #checking data type of array1
Out[13]: numpy.ndarray
```

```
In [14]: whos  # This command displays all the variables used till now

         No variables match your requested type.
```

```
In [15]: lst2=[10,11,12,13,14]    #Again taking a python list
         array2 = np.array(lst2)  # converted to Numpy Array
```

```
In [16]: print(f"Adding two lists {lst1} and {lst2} together: {lst1+lst2}")

         Adding two lists [1, 2, 3, 4] and [10, 11, 12, 13, 14] together: [1, 2, 3, 4,
         10, 11, 12, 13, 14]
```

```
In [17]: print(f "Adding two numpy arrays {array1} and {array2} together: {array1+array2

           Cell In[17], line 1
             print(f "Adding two numpy arrays {array1} and {array2} together: {array1+
         array2}")
                     ^
         SyntaxError: invalid syntax
```

```
In [18]: myst='Hello'  # String variable
         myst.replace('H','a') # replace H by a
         print(myst)

         Hello
```

```
In [ ]: whos
```

MNumPy Array Attributes

let's discuss some useful array attributes. We'll start by defining three random arrays, a one-dimensional, two-dimensional, and three-dimensional array. We'll use NumPy's random number generator, which we will seed with a set value in order to ensure that the same random arrays are generated each time this code is run:

```
In [ ]: a1=np.array([1,2]) # 1 D array
        #print(a1)
        a2=np.array([(1,2),(3,4)],dtype=float) # 2 D array of loat type
        print(a2)
```

```
In [ ]: np.random.seed(25)  # seed for reproducibility

        x1 = np.random.randint(11, size=6, dtype='int16')  # One-dimensional array
        x2 = np.random.randint(11, size=(5, 4))  # Two-dimensional array
        x3 = np.random.randint(11, size=(3, 2, 5))  # Three-dimensional array
        x4= np.random.randint(11, size=(3, 2, 5,4))  # Four-dimensional array
```

```
In [ ]: print(x1) # One-dimensional array
        print("-" * 10)
        print(x2) # Two-dimensional array
        print("-" * 10)
        print(x3) # Three-dimensional array
        print("-" * 10)
        print(x4) # Four-dimensional array
```

Each array has attributes ndim (the number of dimensions), shape (the size of each dimension), and size (the total size of the array):

```
In [ ]: print("The attributes of x3 are::\n")
        print("x3 ndim: ", x3.ndim)
        print("x3 shape:", x3.shape)
        print("x3 size: ", x3.size)
```

```
In [ ]: print("The attributes of x2 are::\n")
        print("x3 ndim: ", x2.ndim)
        print("x3 shape:", x2.shape)
        print("x3 size: ", x2.size)
```

Another useful attribute is the dtype, the data type of the array.

```
In [ ]: print("dtype:", x3.dtype) # 00000000 00000000 00000000 00000000
```

```
In [ ]: x1.dtype # 00000000 00000000
```

Other attributes include itemsize, which lists the size (in bytes) of each array element, and nbytes, which lists the total size (in bytes) of the array:

```
In [ ]: print("itemsize:", x3.itemsize, "bytes")
        print("nbytes:", x3.nbytes, "bytes")
```

In general, we expect that nbytes is equal to `itemsize * size` .

Array Indexing: Accessing Single Elements

If you are familiar with Python's standard list indexing, indexing in NumPy will feel quite familiar. In a one-dimensional array, the i^{th} value (counting from zero) can be accessed by specifying the desired index in square brackets, just as with Python lists:

```
In [ ]: x1
```

```
In [ ]: x1[4]
```

To index from the end of the array, you can use negative indices:

```
In [ ]: x1[-1]
```

```
In [ ]: x1[-2]
```

In a multi-dimensional array, items can be accessed using a comma-separated tuple of indices:

```
In [ ]: x2
```

```
In [ ]: x2[0][0]
```

```
In [ ]: x2[0, 0]
```

```
In [ ]: x2[2, 0]
```

```
In [ ]: x2[2, -1]
```

Values can also be modified using any of the above index notation:

```
In [ ]: x2[0, 0] = 10
        x2
```

Keep in mind that, unlike Python lists, NumPy arrays have a fixed type. This means, for example, that if you attempt to insert a floating-point value to an integer array, the value will be silently truncated.

```
In [ ]: x1
```

In [ ]:
```
x1[0] = 3.14159  # this will be truncated!

x1
```

Array Slicing: Accessing Subarrays

Just as we can use square brackets to access individual array elements, we can also use them to access subarrays with the *slice* notation, marked by the colon (`:`) character. The NumPy slicing syntax follows that of the standard Python list; to access a slice of an array `x` , use this:

```
x[start:stop:step]
```

If any of these are unspecified, they default to the values `start=0` , `stop=` *`size of dimension`* , `step=1` . We'll take a look at accessing sub-arrays in one dimension and in multiple dimensions.

One-dimensional subarrays

In [ ]:
```
x = np.arange(16)
x
```

In [ ]:
```
x[:10]  # first ten elements
```

In [ ]:
```
x[10:]  # elements after index 10 and incuding 10
```

In [ ]:
```
x[5:10]  # middle sub-array
```

In [ ]:
```
x[::2]  # all elements with step size 2
```

In [ ]:
```
x[1::2]  # all elements with step size 2, starting at index 1
```

In [ ]:
```

# arr = np.arange(0,11)
# print("Array:",arr)
# print('-' *70)
# print("Element at 7th index is:", arr[7])
# print('-' *70)
# print("Elements from 3rd to 5th index are:", arr[3:6])
# print('-' *70)
# print("Elements up to 4th index are:", arr[:4])
# print('-' *70)
# print("Elements from last backwards are:", arr[-1::-1])
# print('-' *70)
# print("3 Elements from last backwards are:", arr[-1:-6:-2])
# print('-' *70)
# print('-' *70)

# arr2 = np.arange(0,21,2)
# print("New array:",arr2)
# print("Elements at 2nd, 4th, and 9th index are:", arr2[[2,4,9]]) # Pass a lis
```

Multi-dimensional subarrays

Multi-dimensional slices work in the same way, with multiple slices separated by commas. For example:

In [ ]:
```
x2
```

In [ ]:
```
x2[:2, :3]  # two rows, three columns
```

In [ ]:
```
x2[:3, ::2]  # all rows, every other column
```

In [ ]:
```
np.random.randint(10,100,15)
```

In [ ]:
```
mat = np.random.randint(10,100,15).reshape(3,5)
print("Matrix of random 2-digit numbers\n",mat)
```

```
In [ ]: print("Double bracket indexing\n")
        print("Element in row index 1 and column index 2:", mat[1][2])

        print("\nSingle bracket with comma indexing\n")
        print("Element in row index 1 and column index 2:", mat[1,2])
        print("\nRow or column extract\n")

        print("Entire row at index 2:", mat[2])
        print("Entire column at index 3:", mat[:,3])
```

Creating copies of arrays

Despite the nice features of array views, it is sometimes useful to instead explicitly copy the data within an array or a subarray. This can be most easily done with the `copy()` method:

```
In [ ]: x2
```

```
In [ ]: x2_sub_copy = x2[:2, :2].copy()
        print(x2_sub_copy)
```

If we now modify this subarray, the original array is not touched:

```
In [ ]: x2_sub_copy[0, 0] = 42
        print(x2_sub_copy)
```

```
In [ ]: print(x2)
```

Mathematical operations with and on Numpy arrays

```
In [ ]: lst2=[10,11,12,13]
        array2 = np.array(lst2)
```

```
In [ ]: print(array1)
        print('-' *30)
        print(array2)
```

```
In [ ]: print("array2 multiplied by array1: ",array1*array2)
        print('-' *70)
        print("array2 divided by array1: ",array2/array1)
        print('-' *70)
        print("array2 raised to the power of array1: ",array2**array1)
```

In [ ]:
```python
# sine function
print("Sine: ",np.sin(array1))
print('-' *70)
print('-' *70)
# logarithm
print("Natural logarithm: ",np.log(array1))

print("Base-10 logarithm: ",np.log10(array1))

print("Base-2 logarithm: ",np.log2(array1))
print('-' *70)
print('-' *70)
# Exponential
print("Exponential: ",np.exp(array1))
```

Easy arrays using Numpy

- `np.zeros`
- `np.ones`
- `np.arange`
- `np.linspace`

In [ ]:
```python
print("A series of zeroes:",np.zeros(7))
print('-' *70)
print("A series of ones:",np.ones(9))
print('-' *70)
print("A series of numbers:",np.arange(5,16))
print('-' *70)
print("Numbers spaced apart by 2:",np.arange(0,11,2))
print('-' *70)
print("Numbers spaced apart by float:",np.arange(0,11,2.5))
print('-' *70)
print("Every 5th number from 30 in reverse order: ",np.arange(30,-1,-5))
print('-' *70)
print("11 linearly spaced numbers between 1 and 5: ",np.linspace(1,5,11))
```

Zeros, Ones, Random, and Identity Matrices and Vectors

In [ ]:
```
print("Vector of zeros:\n",np.zeros(5))
print('-' *70)
print("Matrix of zeros:\n",np.zeros((3,4)))
print('-' *70)
print("Vector of ones:\n",np.ones(4))
print('-' *70)
print("Matrix of ones:\n",np.ones((4,2)))
print('-' *70)
print("Matrix of 9’s:\n",9*np.ones((3,3)))
print('-' *70)
print("Identity matrix of dimension 2:\n",np.eye(2))
print('-' *70)
print("Identity matrix of dimension 4:\n",np.eye(4))
```

Array operations (array-array, array-scalar, universal functions)

In [ ]:
```
mat1 = np.random.randint(1,10,9).reshape(3,3)
mat2 = np.random.randint(1,10,9).reshape(3,3)
print("\n1st Matrix of random single-digit numbers\n",mat1)
print("\n2nd Matrix of random single-digit numbers\n",mat2)
```

In [ ]:
```
print("\nAddition\n", mat1+mat2)
print("\nMultiplication\n", mat1*mat2)
print("\nDivision\n", mat1/mat2)
print("\nLineaer combination: 3*A - 2*B\n", 3*mat1-2*mat2)

print("\nAddition of a scalar (100)\n", 100+mat1)

print("\nExponentiation, matrix cubed here\n", mat1**3)
print("\nExponentiation, sq-root using pow function\n",pow(mat1,0.5))
```

2D Array Operations

Numpy arrays also support *broadcasting*, allowing arithmetic operations between two arrays with different numbers of dimensions but compatible shapes. Let's look at an example to see how it works.

In [ ]:
```
arr2 = np.array([[1, 2, 3, 4],
                 [5, 6, 7, 8],
                 [9, 1, 2, 3]])
```

In [ ]:
```
arr2.shape
```

In [ ]:
```
arr4 = np.array([4, 5, 6, 7])
```

In [ ]:
```
arr4.shape
```

In [ ]:
```
arr2 + arr4
```

When the expression `arr2 + arr4` is evaluated, `arr4` (which has the shape `(4,)`) is replicated three times to match the shape `(3, 4)` of `arr2` . Numpy performs the replication without actually creating three copies of the smaller dimension array, thus improving performance and using lower memory.

Broadcasting only works if one of the arrays can be replicated to match the other array's shape.

In [ ]:
```
arr5 = np.array([7, 8])
```

In [ ]:
```
arr5.shape
```

In [ ]:
```
arr2 + arr5
```

In the above example, even if `arr5` is replicated three times, it will not match the shape of `arr2` . Hence `arr2 + arr5` cannot be evaluated successfully. Learn more about broadcasting here: https://numpy.org/doc/stable/user/basics.broadcasting.html (https://numpy.org/doc/stable/user/basics.broadcasting.html) .

Array Comparison

Numpy arrays also support comparison operations like `==` , `!=` , `>` etc. The result is an array of booleans.

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

In [ ]:
```
print(arr1)
print('-' * 30)
print(arr2)
```

In [ ]:
```
arr1 == arr2
```

In [ ]:
```
arr1 != arr2
```

In [ ]:
```
arr1 >= arr2
```

In [ ]:
```
arr1 < arr2
```

np.transpose (np.T)

It performs transpose of a matrix.

```
In [ ]: # Example 1 - working (change this)
        arr1 = [[1, 2, 3],
                [3, 4.,5]]

        np.transpose(arr1)
```

arr1 is given matrix of order `2 X 3` . Output of transpose(arr1) is a matrix of order `3 X 2` . Same can be achieved by `arr1.T` . But, here we need to convert arr1 as `np.arraay` then apply `.T` to get result.

```
In [ ]: np.array(arr1).T
```

np.polyder

The `numpy.polyder()` method evaluates the derivative of a polynomial with specified order.

- Syntax : `numpy.polyder(p, m)`

Parameters :

p : The polynomial coefficients are given in decreasing order of powers. If the second parameter (root) is set to True then array values are the roots of the polynomial equation.

m : Order of differentiation.

Return: Derivative of polynomial.

```
In [ ]: 
        # Constructing polynomial
        poly1 = np.poly1d([2, 3, 4]) # this method prints polynomial
        poly2 = np.poly1d([4, 9, 5, 4, 5])

        print ("Poly1  : \n\n", poly1)
        print ("\n Poly2 : \n\n", poly2)
```

```
In [ ]: a = np.polyder(poly1, 1)
        print ("\nUsing polyder")
        print ("Poly1 derivative of order = 1 : \n", a)
```

```
In [ ]: 
        b = np.polyder(poly2, 2)
        print ("Poly2 derivative of order = 2 : \n", b)
```

np.intersect1d()

To find only the values that are present in both arrays, use the `intersect1d()` method.

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

newarr = np.intersect1d(arr1, arr2)

print(newarr)
```

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

newarr = np.intersect1d(arr1, arr2)

print(newarr)
```

References

- https://github.com/donnemartin/data-science-ipython-notebooks/tree/master/numpy (https://github.com/donnemartin/data-science-ipython-notebooks/tree/master/numpy)
- A Visual Intro to NumPy and Data Representation (http://jalammar.github.io/visual-numpy/)
- NumPy documentation (https://numpy.org/doc/stable/index.html)

Happy Learning 😊