Week-01 : NumPy

1. Numpy

This is a quick overview of arrays in NumPy. It demonstrates how n-dimensional (n>=2) arrays are represented and can be manipulated. In particular, if you don't know how to apply common functions to n-dimensional arrays (without using for-loops), or if you want to understand axis and shape properties for n-dimensional arrays.

Learning Objectives

You should be able to:

- Understand the difference between one-, two- and n-dimensional arrays in NumPy;
- Understand how to apply some linear algebra operations to n-dimensional arrays without using for-loops;
- Understand axis and shape properties for n-dimensional arrays.

The Basics

NumPy's main object is the homogeneous multidimensional array. It is a table of elements (usually numbers), all of the same type, indexed by a tuple of non-negative integers. In NumPy dimensions are called axes.

For example, the array for the coordinates of a point in 3D space, [1, 2, 1], has one axis. That axis has 3 elements in it, so we say it has a length of 3. In the example pictured below, the array has 2 axes. The first axis has a length of 2, the second axis has a length of 3.

```
[[1., 0., 0.],
 [0., 1., 2.]]
```

NumPy's array class is called ndarray. It is also known by the alias array. Note that numpy.array is not the same as the Standard Python Library class array.array, which only handles one-dimensional arrays and offers less functionality. The more important attributes of an ndarray object are:

ndarray.ndim

The number of axes (dimensions) of the array.

ndarray.shape

The dimensions of the array. This is a tuple of integers indicating the size of the array in each dimension. For a matrix with *n* rows and *m* columns, shape will be (n,m). The length of the shape tuple is therefore the number of axes, ndim.

ndarray.size

The total number of elements of the array. This is equal to the product of the elements of shape.

ndarray.dtype

An object describing the type of the elements in the array. One can create or specify dtype's using standard Python types. Additionally NumPy provides types of its own. numpy.int32, numpy.int16, and numpy.float64 are some examples.

ndarray.itemsize

The size in bytes of each element of the array. For example, an array of elements of type float64 has itemsize 8 (=64/8), while one of type complex32 has itemsize 4 (=32/8). It is equivalent to ndarray.dtype.itemsize.

ndarray.data

The buffer containing the actual elements of the array. Normally, we won't need to use this attribute because we will access the elements in an array using indexing facilities.

Example

```
>>>import numpy as np
>>> a = np.arange(15).reshape(3, 5)
>>> a
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
>>> a.shape
```

```
(3, 5)
>>> a.ndim
2
>>> a.dtype.name
'int64'
>>> a.itemsize
8
>>> a.size
15
>>> type(a)
<class 'numpy.ndarray'>
>>> b = np.array([6, 7, 8])
>>> b
array([6, 7, 8])
>>> type(b)
<class 'numpy.ndarray'>
```

Array Creation
There are several ways to create arrays.
For example, you can create an array from a regular Python list or tuple using the array function. The type of the resulting array is deduced from the type of the elements in the sequences.

```
>>>import numpy as np
>>> a = np.array([2, 3, 4])
>>> a
array([2, 3, 4])
>>> a.dtype
dtype('int64')
>>> b = np.array([1.2, 3.5, 5.1])
>>> b.dtype
dtype('float64')
```

array transforms sequences of sequences into two-dimensional arrays, sequences of sequences of sequences into three-dimensional arrays, and so on.

```
>>>b = np.array([(1.5, 2, 3), (4, 5, 6)])
>>> b
array([[1.5, 2. , 3. ],
       [4. , 5. , 6. ]])
```

The type of the array can also be explicitly specified at creation time:

```
>>>c = np.array([[1, 2], [3, 4]], dtype=complex)
>>> c
array([[1.+0.j, 2.+0.j],
       [3.+0.j, 4.+0.j]])
```

Often, the elements of an array are originally unknown, but its size is known. Hence, NumPy offers several functions to create arrays with initial placeholder content. These minimize the necessity of growing arrays, an expensive operation.

The function zeros creates an array full of zeros, the function ones creates an array full of ones, and the function empty creates an array whose initial content is random and depends on the state of the memory. By default, the dtype of the created array is float64, but it can be specified via the key word argument dtype.

```
>>>np.zeros((3, 4))
array([[0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.]])
>>> np.ones((2, 3, 4), dtype=np.int16)
array([[[1, 1, 1, 1],
        [1, 1, 1, 1],
        [1, 1, 1, 1]],

       [[1, 1, 1, 1],
        [1, 1, 1, 1],
        [1, 1, 1, 1]]], dtype=int16)
>>> np.empty((2, 3))
array([[3.73603959e-262, 6.02658058e-154, 6.55490914e-260],  # may vary
       [5.30498948e-313, 3.14673309e-307, 1.00000000e+000]])
```

uint diff from int

To create sequences of numbers, NumPy provides the arange function which is analogous to the Python built-in range, but returns an array.

```
>>>np.arange(10, 30, 5)
array([10, 15, 20, 25])
>>> np.arange(0, 2, 0.3)  # it accepts float arguments
array([0. , 0.3, 0.6, 0.9, 1.2, 1.5, 1.8])
```

When arange is used with floating point arguments, it is generally not possible to predict the number of elements obtained, due to the finite floating point precision. For this reason, it is usually better to use the function linspace that receives as an argument the number of elements that we want, instead of the step:

```
>>>from numpy import pi
>>> np.linspace(0, 2, 9)              # 9 numbers from 0 to 2
array([0.  , 0.25, 0.5 , 0.75, 1.  , 1.25, 1.5 , 1.75, 2.  ])
>>> x = np.linspace(0, 2 * pi, 100) # useful to evaluate
                                 #function at lots of points
>>> f = np.sin(x)
```

Printing Arrays

When you print an array, NumPy displays it in a similar way to nested lists, but with the following layout:

- the last axis is printed from left to right,
- the second-to-last is printed from top to bottom,
- the rest are also printed from top to bottom, with each slice separated from the next by an empty line.

One-dimensional arrays are then printed as rows, bidimensionals as matrices and tridimensionals as lists of matrices.

```
>>>a = np.arange(6)                   # 1d array
>>> print(a)
[0 1 2 3 4 5]
>>> b = np.arange(12).reshape(4, 3)     # 2d array
>>> print(b)
[[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]]
>>> c = np.arange(24).reshape(2, 3, 4)  # 3d array
>>> print(c)
[[[ 0  1  2  3]
  [ 4  5  6  7]
  [ 8  9 10 11]]

 [[12 13 14 15]
  [16 17 18 19]
  [20 21 22 23]]]
```

If an array is too large to be printed, NumPy automatically skips the central part of the array and only prints the corners:

```
>>>print(np.arange(10000))
[   0    1    2 ... 9997 9998 9999]
>>>
>>> print(np.arange(10000).reshape(100, 100))
[[   0    1    2 ...   97   98   99]
 [ 100  101  102 ...  197  198  199]
 [ 200  201  202 ...  297  298  299]
 ...
 [9700 9701 9702 ... 9797 9798 9799]
 [9800 9801 9802 ... 9897 9898 9899]
 [9900 9901 9902 ... 9997 9998 9999]]
```

To disable this behaviour and force NumPy to print the entire array, you can change the printing options using set_printoptions.

```
>>> np.set_printoptions(threshold=sys.maxsize)  # sys module should be imported
```

Basic Operations

Arithmetic operators on arrays apply *elementwise*. A new array is created and filled with the result.

```
>>> a = np.array([20, 30, 40, 50])
>>> b = np.arange(4)
>>> b
array([0, 1, 2, 3])
>>> c = a - b
>>> c
array([20, 29, 38, 47])
>>> b**2
array([0, 1, 4, 9])
>>> 10 * np.sin(a)
array([ 9.12945251, -9.88031624,  7.4511316 , -2.62374854])
>>> a < 35
```

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

Unlike in many matrix languages, the product operator * operates elementwise in NumPy arrays. The matrix product can be performed using the @ operator (in python >=3.5) or the dot function or method:

```
>>> A = np.array([[1, 1],
...               [0, 1]])
>>> B = np.array([[2, 0],
...               [3, 4]])
>>> A * B     # elementwise product
array([[2, 0],
       [0, 4]])
>>> A @ B     # matrix product
array([[5, 4],
       [3, 4]])
>>> A.dot(B)  # another matrix product
array([[5, 4],
       [3, 4]])
```

Some operations, such as += and *=, act in place to modify an existing array rather than create a new one.

```
>>> rg = np.random.default_rng(1)  # create instance of default random number generator
>>> a = np.ones((2, 3), dtype=int)
>>> b = rg.random((2, 3))
>>> a *= 3
>>> a
array([[3, 3, 3],
       [3, 3, 3]])
>>> b += a
>>> b
array([[3.51182162, 3.9504637 , 3.14415961],
       [3.94864945, 3.31183145, 3.42332645]])
>>> a += b  # b is not automatically converted to integer type
Traceback (most recent call last):
    ...
numpy.core._exceptions._UFuncOutputCastingError: Cannot cast ufunc 'add' output from dtype('float64') to dtype('int64') with casting rule 'same_kind'
```

When operating with arrays of different types, the type of the resulting array corresponds to the more general or precise one (a behavior known as upcasting).

```
>>> a = np.ones(3, dtype=np.int32)
>>> b = np.linspace(0, pi, 3)
>>> b.dtype.name
'float64'
>>> c = a + b
>>> c
array([1.        , 2.57079633, 4.14159265])
>>> c.dtype.name
'float64'
>>> d = np.exp(c * 1j)
>>> d
array([ 0.54030231+0.84147098j, -0.84147098+0.54030231j,
       -0.54030231-0.84147098j])
>>> d.dtype.name
```

```
'complex128'
```

Many unary operations, such as computing the sum of all the elements in the array, are implemented as methods of the ndarray class.

```
>>> a = rg.random((2, 3))
>>> a
array([[0.82770259, 0.40919914, 0.54959369],
       [0.02755911, 0.75351311, 0.53814331]])
>>> a.sum()
3.1057109529998157
>>> a.min()
0.027559113243068367
>>> a.max()
0.8277025938204418
```

Questions

1. Convert a 1-D array into a 2-D array with 3 rows.

Start with:
Assign-1 = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8])
Desired output:
[[0, 1, 2]
[3, 4, 5]
[6, 7, 8]]

2. Replace all odd numbers in the given array with -1

Start with:
Assign-2 = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
Desired output:
[0, -1, 2, -1, 4, -1, 6, -1, 8, -1]

3. Find the positions of:

elements in x where its value is more than its corresponding element in y, and elements in x where its value is equals to its corresponding element in y.

Start with these:
x = np.array([21, 64, 86, 22, 74, 55, 81, 79, 90, 89])
y = np.array([21, 7, 3, 45, 10, 29, 55, 4, 37, 18])

Desired output:
(array([1, 2, 4, 5, 6, 7, 8, 9]),) and (array([0]),)

4. **Extract the first four columns of this 2-D array.**
Start with this:
Assign-4= np.arange(100).reshape(5,-1)

Desired output:
[[0 1 2 3]
[20 21 22 23]
[40 41 42 43]
[60 61 62 63]
[80 81 82 83]]

Additional questions

1. **Generate a 1-D array of 10 random integers. Each integer should be a number between 30 and 40 (inclusive).**
Sample of desired output:
[36, 30, 36, 38, 31, 35, 36, 30, 32, 34]

2. Consider the following matrices :
A= ((1, 2, 3), (4, 5, 6), (7, 8, 10)) and   B = ((7, 8, 10) ,(4, 5, 6), (1, 2, 3))
Write a python program to perform the following using Numeric Python (numpy).
i) Add and Subtract of the Matrix A and B, print the resultant matrix C for add and E for subtract.
ii) Compute the sum of all elements of Matrix A, sum of each column of Matrix B and sum of each row of Matrix C
iii) Product of two matrices A and B, and print the resultant matrix D
iv) Sort the elements of resultant matrix C and print the resultant Matrix E.
v) Transpose the Matrix E and print the result

use np.sort() to sort the elements