NUMPY

Numerical Python Library

Introduction

- NumPy library, short for **Numerical Python** library.

- Used for performing arithmetic , linear algebraic and other mathematical operations on arrays.

- ML packages like Scipy (Scientific Python), Scikit-learn and the data pre-processing library, Pandas are all built on top of Numpy.

Advantages:

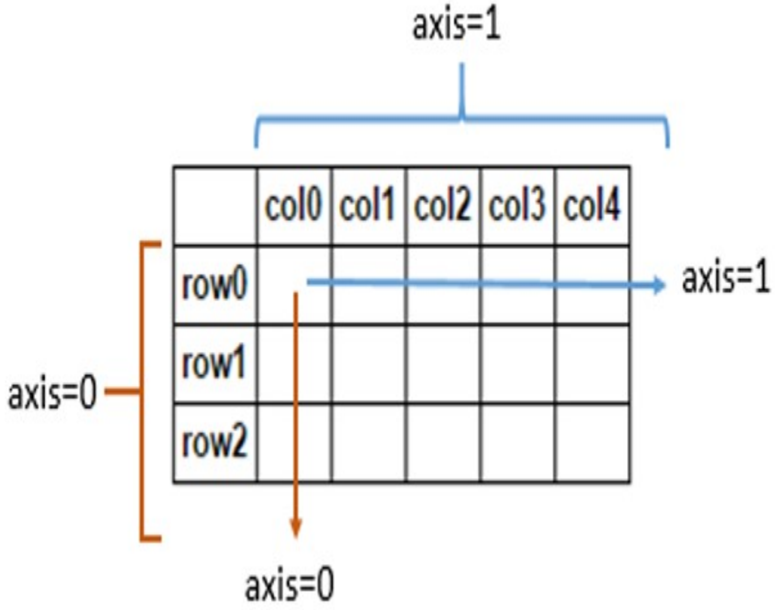
- Speed
- Compact nature of code

What is a NumPy array?

- The most basic object in NumPy is an ndarray or simply an array.
- ***"ndarray"*** means 'n' dimensional array.
- It is a homogeneous array: All the elements of the array have same data type.
- Generally, data type will be numeric in nature (float or integer).

- The most common arrays are: (In Linear Algebra)
  - ➢ One-dimensional *(1-D)* : **Vectors**
  - ➢ _Two dimensional *(2-D)*: ***matrices***

- A typical array looks like this. In NumPy terminology, for 2-D arrays:
  - ➢ axis = 0 refers to the rows
  - ➢ axis = 1 refers to the columns



CREATING AND INSPECTING ARRAYS

Creating a NumPy Array:

- One can create NumPy arrays in multiple ways
  - ➢ Other python data structures like lists, tuples
  - ➢ Using built in functions
  - ➢ Simply by giving the values.

- Before that, We have to load numpy library into jupyter notebook

```
import numpy as np
```

Creating NumPy arrays from Lists and Tuples:

aCubeIT
Institute of Artificial Intelligence

1-D Array:

- Frequently used syntax for creating an array: ***np.array***.

```
In [8]: # Convert lists or tuples to arrays using np.array()
        # Note that np.array(2, 5, 6, 7) will throw an error
        #you need to pass a list or a tuple
        list1 = [2,5,6,7]
        array_from_list = np.array(list1)
        tuple1 = (4,5,8,9)
        array_from_tuple = np.array(tuple1)

        print(array_from_list)
        print(array_from_tuple)

        [2 5 6 7]
        [4 5 8 9]
```

- Converted python **list or tuple to a NumPy array object .**

Creating NumPy arrays from Lists and Tuples:

2-D Array:

- To create a *2-D* array from list or tuple, we should have a *list of lists* or *tuple of tuples or tuple of lists*.

```
In [7]: # Convert lists or tuples to arrays using np.array()

        list1_2d = [[2,5],[6,7]]             # List of Lists
        array_2d_1 = np.array(list1_2d)
        tuple1_2d = ((4,5),(8,9))            # Tuple of Tuples
        array_2d_2 = np.array(tuple1_2d)

        print(array_2d_1,'\n')
        print(array_2d_2)

[[2 5]
 [6 7]]

[[4 5]
 [8 9]]
```

Try to create an array using **list of tuples**

Creating arrays *using built-in functions:*

- *Some functions are:*

np.ones()	*Creates an array of **all ones***
np.zeros()	*Creates an array of **all zeros***
np.arange()	*Creates an array of **range** (Similar to **range()**)*
np.linspace()	*Creates an array **evenly spaced in an interval***
np.eye()	*Creates an **identity matrix***
np.full()	Create a constant array of any number 'n'
np.tile()	Create a new array by repeating an existing array

Creating arrays *using built-in functions:*

np.ones() and np.zeros():

```
In [3]: # Creating a 5 x 3 array of ones
        a = np.ones((5, 3))
        # Notice that, by default, numpy creates data type = float64
        # Can provide dtype explicitly using dtype
        b = np.ones((5, 3), dtype = np.int)
        # Creating array of zeros
        c = np.zeros(4, dtype = np.int)
```

5 rows and 3 columns

Data type explicitly initialized

- ***Tuple (5,3)*** as argument generates a ***2-D array*** of all ones with **5 rows and 3 columns**
- **Data type** can be explicitly mentioned
- **np.zeros()** is a similar function creates an **array of all zeros**.

aCubeIT
Institute of Artificial Intelligence

Creating arrays *using built-in functions:*

np.arange():

```
In [20]: # np.arange()
         # np.arange() is the numpy equivalent of range()
         # Notice that 10 is included, 100 is not, as in standard python lists

         # From 10 to 100 with a step of 5
         numbers = np.arange(10, 100, 5)
         print(numbers)
         numbers.reshape((3,6))
```

reshape() allows to transform the dimensions

```
[10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85 90 95]

Out[20]: array([[10, 15, 20, 25, 30, 35],
                [40, 45, 50, 55, 60, 65],
                [70, 75, 80, 85, 90, 95]])
```

- ***np.arange()*** is similar to python built-in ***range()***
- *Syntax:* ***arange(start,stop,step)*** (step optional)
- **reshape()** helps in changing the dimensions of the existing array

Creating arrays *using built-in functions:*

np.linspace():

```
In [14]: # np.linspace()
         # Sometimes, you know the length of the array, not the step size

         # Array of length 25 between 15 and 18
         np.linspace(15, 18, 25)

Out[14]: array([15.   , 15.125, 15.25 , 15.375, 15.5  , 15.625, 15.75 , 15.875,
                16.   , 16.125, 16.25 , 16.375, 16.5  , 16.625, 16.75 , 16.875,
                17.   , 17.125, 17.25 , 17.375, 17.5  , 17.625, 17.75 , 17.875,
                18.   ])
```

- ***linspace()*** returns numbers evenly spaced over a specified intervals
- It takes the **third argument as the number of data points to be created**

Creating arrays *using built-in functions:*

np.eye(): Creates identity matrix

```
In [18]: # Create a 3 x 3 identity matrix using np.eye()
         np.eye(3, dtype = int)

Out[18]: array([[1, 0, 0],
                [0, 1, 0],
                [0, 0, 1]])
```

- Identity matrix is **religiously used in linear algebra**

Creating random number arrays

np.random.randint():

```
In [19]: # Create a 4 x 4 random array of integers ranging from 0 to 9
         np.random.randint(0, 10, (4,4))

Out[19]: array([[1, 8, 4, 4],
                [7, 2, 4, 6],
                [3, 9, 9, 2],
                [1, 2, 8, 1]])
```

- Random number generator is a **separate package in NumPy.**
- call out ***np.random*** before asking for a particular type of random number to be generated.
- ***randint()*** in the given syntax is distributing numbers 1 to 9 uniformly in the matrix

Creating random number arrays

- Different random number syntaxes are:
  - ➢ np.random.rand()
  - ➢ np.random.randn()
  - ➢ np.random.randint().

Inspecting arrays:

- Typically, any real time data science problem will have thousands to lakhs of rows and hundreds of columns.

- So, it's helpful to inspect the structure of arrays.

- **We cannot make any sense of the data merely by printing the data and it's time consuming too.**

- *There are few built-in functions to quickly inspect the arrays.*
  - ***shape***: No. of rows and columns in a given array
  - ***dtype***: To get the data type of the array.
  - ***ndim***: To get the dimensionality of the array.
  - ***itemsize***: To get the size of the array in 'kB'.

Inspecting arrays:

```
In [25]: # Initialising a random 1000 x 300 array

         rand_array = np.random.randn(1000, 300)

         # Inspecting shape, dtype, ndim and itemsize

         print("Shape:",rand_array.shape)
         print("dtype:",rand_array.dtype)
         print("Dimensions: ",rand_array.ndim)
         print("Item size: ",rand_array.itemsize)

         Shape: (1000, 300)
         dtype: float64
         Dimensions:  2
         Item size:  8
```

- We cannot make sense of data merely by displaying a 1000 x 300 random numbers.
- While pre-processing data in data science projects, it becomes part of the process to inspect data every time we make data transformations.

INDEXING AND OPERATIONS

Array indexing/Slicing:

- Array slicing is similar to other data structures in Python.

- **We pass the index we want and get an element or group of elements out**.

- Similar to regular python, **elements of an array are indexed as (0, n-1)**.

Array indexing/Slicing:

aCubeIT
Institute of Artificial Intelligence

1-D Slicing:

In [2]:
```
# Indexing and slicing one dimensional arrays
array_1d = np.arange(10)
print(array_1d)
```

```
[0 1 2 3 4 5 6 7 8 9]
```

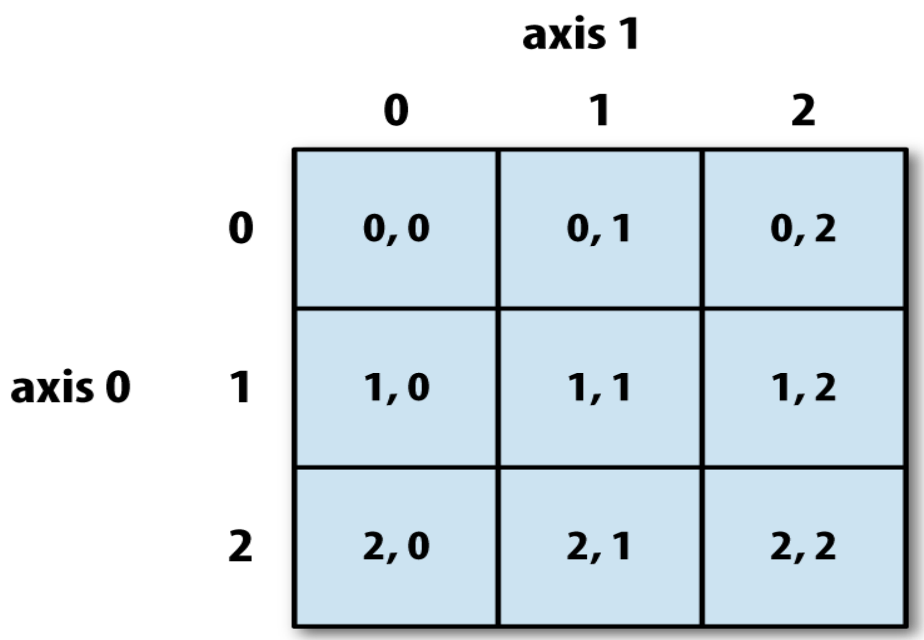
In [24]:
```
# Third element
print(array_1d[2])

# Slice third element onwards
print(array_1d[2:])

# Slice first three elements
print(array_1d[:3])

# Slice third to seventh elements
print(array_1d[2:7])

# Subset starting 0 at increment of 2
print(array_1d[0::2])
```

- ‘ : ’ is used to get a range of values just like in lists.
- Ex: **‘2:5’** is interpreted as a request to pull out elements from **2nd to (5 -1 = 4)th elements.**
- **Try guessing the results for the remaining cells.**

Array indexing/Slicing:

2-D Slicing:

- Multidimensional arrays are indexed using as many indices as the number of dimensions or axes.
- To index a 2-D array, you need two indices - **array[x, y]**
- *In **[x, y]**, x is for rows and y is for columns*.



Array indexing/Slicing:

2-D Slicing:

- ' , ' separates row slicing from column slicing.
- ' : ' without mentioning the range is used to retrieve all the elements of a particular row or column

In [4]:
```
# Creating a 2-D array
array_2d = np.array([[2, 5, 7, 5], [4, 6, 8, 10], [10, 12, 15, 19]])
print(array_2d)
```

```
[[ 2  5  7  5]
 [ 4  6  8 10]
 [10 12 15 19]]
```

In [14]:
```
# Third row second column
print(array_2d[2, 1])

# Slicing the second row, and all columns
# Notice that the resultant is itself a 1-D array
print(array_2d[1, :])

# Slicing all rows and the first three columns
print(array_2d[:, :3])
```

```
12
[ 4  6  8 10]
[[ 2  5  7]
 [ 4  6  8]
 [10 12 15]]
```

Operations on NumPy arrays:

- In NumPy arrays, we can perform more **mathematical and logical operations than one can perform on data structures** like lists and tuples in python.

- On top of that, we can **extensively perform linear algebra and trigonometry calculations** on array objects.

- The learning objectives of this part of the article is broadly classified as
  - ➢ Manipulating arrays
  - ➢ Mathematical and Logical operations on arrays

Manipulating arrays:

aCubeIT
Institute of Artificial Intelligence

Reshaping arrays: *np.reshape()*

- ***reshape():*** transform an array from one dimension to another.
- ***Limitation:*** Example if we have a '(5,4)' array, we can transform that into a new array of these 4 dimensions only:*'(2,10)','(10,2)','(1,20)','(20, 1)' because 5*4 equals 2*10 and so on.*
- **reshape(4,-1)** creates 4 rows and calculates the no of columns by itself.

In [2]:
```python
# Reshape a 1-D array to a 3 x 4 array
some_array = np.arange(0, 12).reshape(3, 4)
print(some_array)
```

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

In [3]:
```python
# Can reshape it further
some_array.reshape(2, 6)
```

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

In [3]:
```python
# If you specify -1 as a dimension, the dimensions are automatically calculated
# -1 means "whatever dimension is needed"
some_array.reshape(4, -1)
```

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

Manipulating arrays:

Stacking arrays: *np.hstack() and np.vstack()*

Arrays should be given as ***'Tuple of Arrays'*** **argument**

```
In [ ]: # Creating two arrays
        array_1 = np.arange(12).reshape(3, 4)
        array_2 = np.arange(12,24).reshape(3, 4)

        print(array_1)
        print("\n")
        print(array_2)
```

```
In [ ]: # Try vstack
        # Note that np.vstack(a, b) throws an error -
        # you need to pass the arrays as a list
        print(np.vstack((array_1, array_2)))
        print('\n')
        # Try hstack
        print(np.hstack((array_1, array_2)))
```

- ***'vstack()'*** places array_2 below array_1 *(Vertically)*
- ***'hstack()'*** places the arrays in the arguments one beside the other.

Logical Operations on arrays:

&(AND), | (OR), <, > and == operators

```
In [14]: array_logical = np.arange(5,15)
         print(array_logical)
         array_logical > 10

[ 5  6  7  8  9 10 11 12 13 14]

Out[14]: array([False, False, False, False, False, False,  True,  True,  True,
                True])
```

```
In [ ]: # try this
        # 1
        bool_arr = array_logical > 10
        array_logical[bool_arr]

        array_logical[array_logical>10] #A shorter way to do what we have just done
        # 2
        array_logical[(array_logical>6) & (array_logical<10)]
```

- *For **array_logical > 10*** , result is a boolean array where it compared each element is greater than or equal to 10.

Mathematical Operations on arrays:

Basic Arithmetic Operations:

```
In [21]: arr = np.arange(1,11).reshape(2,5)
         print(arr * arr,'\n')          #Multiplies each element by itself
         print(arr - arr,'\n')          #Subtracts each element from itself
         print(arr + arr,'\n')          #Adds each element to itself
         print(arr / arr,'\n')          #Divides each element by itself

[[  1   4   9  16  25]
 [ 36  49  64  81 100]]

[[0 0 0 0 0]
 [0 0 0 0 0]]

[[ 2  4  6  8 10]
 [12 14 16 18 20]]

[[1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]]
```

- These are **element-wise operations**.

Mathematical Operations on arrays:

Linear Algebraic Operations:

```
In [22]: # Creating arrays
         a = np.arange(1, 10).reshape(3, 3)
         b= np.arange(1, 13).reshape(3, 4)
         print(a)
         print(b)

[[1 2 3]
 [4 5 6]
 [7 8 9]]
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
```

- np.linalg.inv: Inverse of a matrix
- np.linalg.det: Determinant of a matrix
- np.linalg.eig: Eigenvalues and eigenvectors of a matrix
- np.dot: Multiplication of matrices

Mathematical Operations on arrays:

Linear Algebraic Operations:

```
In [23]: # Inverse
         np.linalg.inv(a)

Out[23]: array([[ 3.15251974e+15, -6.30503948e+15,  3.15251974e+15],
                [-6.30503948e+15,  1.26100790e+16, -6.30503948e+15],
                [ 3.15251974e+15, -6.30503948e+15,  3.15251974e+15]])

In [24]: # Determinant
         np.linalg.det(a)

Out[24]: -9.51619735392994e-16

In [25]: # Eigenvalues and eigenvectors
         np.linalg.eig(a)

Out[25]: (array([ 1.61168440e+01, -1.11684397e+00, -9.75918483e-16]),
          array([[-0.23197069, -0.78583024,  0.40824829],
                 [-0.52532209, -0.08675134, -0.81649658],
                 [-0.8186735 ,  0.61232756,  0.40824829]]))

In [26]: # Multiply matrices
         np.dot(a, b)

Out[26]: array([[ 38,  44,  50,  56],
                [ 83,  98, 113, 128],
                [128, 152, 176, 200]])
```

Mathematical Operations on arrays:

Universal functions:

```
In [ ]: np.sqrt(arr)    #Returns the square root of each element
        np.exp(arr)     #Returns the exponentials of each element
        np.sin(arr)     #Returns the sin of each element
        np.cos(arr)     #Returns the cosine of each element
        np.log(arr)     #Returns the logarithm of each element
        np.sum(arr)     #Returns the sum total of elements in the array
        np.std(arr)     #Returns the standard deviation of in the array
```

```
In [ ]: mat = np.arange(1,26).reshape(5,5)
        mat.sum()          #Returns the sum of all the values in mat
        mat.sum(axis=0)    #Returns the sum of all the columns in mat
        mat.sum(axis=1)    #Returns the sum of all the rows in mat
```

- Among these, functions like sum(), std(), count() are repeatedly used while pre-processing data in data science projects.

BROADCASTING

Broadcasting:

- In general, arrays of different dimensions cannot be added or subtracted.
- NumPy has a smart way to overcome this problem by duplicating the smaller dimension array to be the size of a higher dimension array and then performs the operation. ***It is called broadcasting.***

- Ex: If we want to **add array([3]) to array([1,2,3]).** By simply giving array([3]) + array([1,2,3]), **numpy understands that your idea is to add [3] to every element of [1,2,3].** Immediately, it **duplicates the value [3] as many times as it is in the larger array, in this case, array([3,3,3]) and now performs the addition operation**.

Broadcasting:

- *"The term broadcasting describes how numpy treats arrays with different shapes during arithmetic operations.* ***Subject to certain constraints****, the smaller array is "broadcast" across the larger array so that they have compatible shapes."*

- We can take 3 types of examples to efficiently convey the concept of broadcasting. Let us see examples first and try to interpret them.
  - ➢ *Arithmetic operation on 1-D Array with a scalar number*
  - ➢ *Arithmetic operation on 2-D Array with a scalar number*
  - ➢ *Arithmetic operation on 2-D and with vectors (1-D array)*

Broadcasting:

Arithmetic operation on 1-D and 2-D Arrays with a scalar number:

- value of scalar ‘b’ is **duplicated so that both array dimensions are equal and added.**
- Similarly in the second cell, we added a scalar value to a 2-D array.

```
In [15]: # Adding one dimensional array (a_1d) with a scalar number (b)
         a_1d = np.array([1, 2, 3])
         print(a_1d)
         b = np.array([2])
         print(b)
         c = a_1d + b
         c

         [1 2 3]
         [2]

Out[15]: array([3, 4, 5])

In [16]: # Adding two dimensional array (a_2d) with a scalar number (b)
         a_2d = np.array([[1, 2, 3],[4,5,6]])
         print(a_2d)
         b = np.array([2])
         print(b)
         c = a_2d + b
         c

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

Out[16]: array([[3, 4, 5],
                [6, 7, 8]])
```

Broadcasting:

Arithmetic operation on 1-D and 2-D Arrays with a scalar number:

- The actual purpose of **broadcasting is to add 2 arrays of different dimensions > 1.**
- The first example here is a row-wise broadcasting. **This means each row in ‘a_2d’ is added with the 1-D array ‘b_1d**.’
- This happened because ‘b_1d’ is a unit row vector

```
In [20]: # Adding two dimensional array (a_2d) with unit row array (b_1d)
a_2d = np.array([[1, 2, 3], [1, 2, 3]])
print(a_2d)
b_1d = np.array([1, 2, 3])
print(b_1d)
c = a_2d + b_1d
print('\n c =',c)

[[1 2 3]
 [1 2 3]]
[1 2 3]

 c = [[2 4 6]
 [2 4 6]]
```

```
In [25]: # Adding two dimensional array (a_2d) with a unitcolumn array (b_1d)
a_2d = np.array([[1, 2,3], [1, 2, 3],[1, 2,  3]])
b_1d = np.array([[1],[2],[3]])
c = a_2d + b_1d
print('\n c =',c)

 c = [[2 3 4]
 [3 4 5]
 [4 5 6]]
```

Broadcasting:

Limitation:

```
In [27]: # Adding two dimensional array (a_2d) of shape (2,3) with a unit column array (b_1d) of shape (3,1)
a_2d = np.array([[1, 2, 3], [1, 2, 3]])
b_1d = np.array([[1],[2],[3]])
c = a_2d + b_1d
print('\n c =',c)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-27-a014e71158c8> in <module>()
      2 a_2d = np.array([[1, 2, 3], [1, 2, 3]])
      3 b_1d = np.array([[1],[2],[3]])
----> 4 c = a_2d + b_1d
      5 print('\n c =',c)

ValueError: operands could not be broadcast together with shapes (2,3) (3,1)
```

- Broadcasting expects at least any one dimension (row or column) to be equal in both the arrays.
- neither column nor row dimensions are equal and hence the 'value error'