Numpy and Pandas for Data Manipulation

Numpy

NumPy stands for 'Numerical Python' or 'Numeric Python'. It is an open source module of Python which provides fast mathematical computation on arrays and matrices.

NumPy provides the essential multi-dimensional array-oriented computing functionalities designed for high-level mathematical functions and scientific computation. Numpy can be imported into the notebook using

NumPy brings the computational power of languages like C and Fortran to Python, a language much easier to learn and use.

It is used in every field of science and engineering and is the universal standard for working with numerical data in Python.

Essential tool for beginners as well as experienced researchers doing state-of-the-art scientific and industrial research and development.

NumPy API are use in Pandas, SciPy, Matplotlib, Scikit-learn, Scikit-image, Pytorch, Tensorflow and most other data science and scientific Python packages.

In [1]:
```
import numpy as np
import pandas as pd
```

Define Matrix and Vectors

Convert Python Data Structures to numpy arrays

In [2]:
```
# Define np array using python list

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

print(matrix)
```

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

In [3]:
```
# Define np array using pandas dataframe

df = pd.DataFrame({"A": [1, 2, 3, 4], "B": [2, 6, 7, 8], "C": [9, 10, 11, 12]})

#Convert to numpy array using to_numpy function
matrix = df.to_numpy()

print(df,'\n')
print(matrix)
```

```
   A  B   C
0  1  2   9
1  2  6  10
2  3  7  11
3  4  8  12

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

Define vector, array and tensor

In [4]:
```
# Define a vector from linear algebra course

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

print(vector)
```

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

In [5]:
```
#Define an array from linear algebra course

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

print(matrix)
```

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

In [6]:
```
# Define a tensor from linear algebra course

tensor = np.array([matrix,matrix])
print(tensor.shape)
print(tensor)
```

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

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

Defining Special Arrays

We are going to explore the following functions:

Define an array of 1s: np.ones()

Define an array of 0s: np.zeros()

Define an array of constant: np.full()

Define interval and range arrays: np.interval() and np.range()

The same concepts can be extended to define vectors and tensors

In [7]:
```
# Define an array with zero values
# All these functions can be extended to define any n-dimensional numpy array
zero_arr = np.zeros((3))
print(zero_arr)
```

```
[0. 0. 0.]
```

In [8]:
```
z1=np.zeros((3,3))
print(z1)
```

```
[[0. 0. 0.]
 [0. 0. 0.]
 [0. 0. 0.]]
```

In [9]:
```
z2 = np.zeros((3,4,5))
print(z2)
```

```
[[[0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0.]]

 [[0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0.]]

 [[0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0.]
  [0. 0. 0. 0. 0.]]]
```

In [10]:
```
z = np.zeros((3,4,5,1))
print(z)
```

```
[[[[0.]
   [0.]
   [0.]
   [0.]
   [0.]]

  [[0.]
   [0.]
   [0.]
   [0.]
   [0.]]

  [[0.]
   [0.]
   [0.]
   [0.]
   [0.]]

  [[0.]
   [0.]
   [0.]
   [0.]
   [0.]]]


 [[[0.]
   [0.]
   [0.]
   [0.]
   [0.]]

  [[0.]
   [0.]
   [0.]
   [0.]
   [0.]]

  [[0.]
   [0.]
   [0.]
   [0.]
   [0.]]

  [[0.]
   [0.]
   [0.]
   [0.]
   [0.]]]


 [[[0.]
   [0.]
   [0.]
   [0.]
   [0.]]

  [[0.]
   [0.]
   [0.]
   [0.]
   [0.]]

  [[0.]
   [0.]
   [0.]
   [0.]
   [0.]]

  [[0.]
   [0.]
   [0.]
   [0.]
   [0.]]]]
```

In [11]:
```
# Define an array of 1s

one_arr = np.ones((3,2))

print(one_arr)
```

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

In [12]:
```
# Define a array of constant values:

const_arr = np.full((5,2), 10.)

print(const_arr)
```

```
[[10. 10.]
 [10. 10.]
 [10. 10.]
 [10. 10.]
 [10. 10.]]
```

In [13]:
```
# Define interval and range arrays
range_arr = np.arange(2,9,2)              #start, end and interval
interval_arr = np.linspace(2,8, num=4)    # start, end and number of elements
print("Range array in numpy is:\n {}\n".format(range_arr))
print("Interval array in numpy is:\n {}\n".format(interval_arr))
```

```
Range array in numpy is:
 [2 4 6 8]

Interval array in numpy is:
 [2. 4. 6. 8.]
```

Datatypes :

Let's explore the function astype() to change the datatype in an numpy array.

Following are the datatypes in the numpy arrays:

1. integer or int
2. float
3. String

In [14]:
```
# changing element datatypes in numpy from float to string

interval_arr = np.linspace(2,8, num=4)
print(interval_arr)

print(type(interval_arr[0]))

interval_arr = interval_arr.astype('int64')

print(type(interval_arr[0]))

print(interval_arr)
```

```
[2. 4. 6. 8.]
<class 'numpy.float64'>
<class 'numpy.int64'>
[2 4 6 8]
```

Indexing and Slicing

1. An important concept to understand before going into array indexing is axis of array. Axis can be derived from shape of an array.
2. The ith element in shape() function returns the length of the corresponding axis and all the array operations are performed along that specified axis.

In [17]:
```
#Already covered in list, FOCUS ON AXIS

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

print("original array is \n{}\n".format(arr))

print("Subarray till 2nd row and 2nd to 3rd columns \n{}\n".format(arr[1:2, 1:3]))
```

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

Subarray till 2nd row and 2nd to 3rd columns
[[6 7]]
```

In [18]:
```
#Mix integer indexing with slice indexing

row = arr[:, :-2]
print("Last row is \n{}\n".format(row))
```

```
Last row is
[[ 1  2]
 [ 5  6]
 [ 9 10]]
```

Shape and Size

1. ndarray.shape: Tuple of integers that indicate the number of elements stored along each dimension of the array. If, for example, you have a 2-D array with 2 rows and 3 columns, the shape of your array is (2, 3).
2. ndarray.ndim: Number of dimensions of the array.
3. ndarray.size: Total number of elements of the array. This is the product of the elements of the array's shape.

In [19]:
```
arr = np.array([[[0, 1, 2, 3],
                 [4, 5, 6, 7]],
                [[0, 1, 2, 3],
                 [4, 5, 6, 7]],
                [[0 ,1 ,2, 3],
                 [4, 5, 6, 7]]])

print("Number of dimension of array is \n{}\n".format(arr.ndim))
print("Total number of elements in the array are \n{}\n".format(arr.size))
print("Shape of the array is \n{}\n ".format(arr.shape))
```

```
Number of dimension of array is
3

Total number of elements in the array are
24

Shape of the array is
(3, 2, 4)
```

Array Resizing

In [21]:
```
# Flatten an array using np.ravel()

print("original array is \n{}\n".format(arr))
print("Flattened array is \n{}\n".format(np.ravel(arr)))
print(arr.shape)
```

```
original array is
[[[0 1 2 3]
  [4 5 6 7]]

 [[0 1 2 3]
  [4 5 6 7]]

 [[0 1 2 3]
  [4 5 6 7]]]

Flattened array is
[0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7]

(3, 2, 4)
```

In [22]:
```
# Transpose is another resizing operation performed using np.Transpose or .T

arr = np.arange(6).reshape((2,3))

print("array is \n{}\n".format(arr))

print("Transpose the array is \n{}\n".format(np.transpose(arr)))

print("Transpose the array is \n{}\n".format(arr.T))
```

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

Transpose the array is
[[0 3]
 [1 4]
 [2 5]]

Transpose the array is
[[0 3]
 [1 4]
 [2 5]]
```

Arthematic Operations

1. Addition
2. Array concatination
3. Subtraction
4. Multiplication
5. Division

In [24]:
```
# Shape of an array

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

print(x.shape)
print(x)
```

```
(2, 3)
[[1 2 3]
 [4 5 6]]
```

Axis 0 has length 2 and axis 1 has length 3

1. The default axis for array operations is 0 unless otherwise stated in the function defintion
2. **axis=-1** represents the last axis of an array.

In [26]:
```
# Arthimetic addition of arrays can be performed by simple addition if all the arrays are of same length.

x = np.array([[1, 2], [3, 4]])
y = np.array([[5, 6],[10,11]])

print("x is \n{}\n".format(x))
print("y is \n{}\n".format(y))
print("x+y is \n{}".format(x+y))
```

```
x is
[[1 2]
 [3 4]]

y is
[[ 5  6]
 [10 11]]

x+y is
[[ 6  8]
 [13 15]]
```

In [27]:
```
# Adding using concatenation along 0th axis


print("x is \n{}\n".format(x))
print("y is \n{}\n".format(y))
print("x+y along axis 0 is \n{}".format(np.concatenate((x, y), axis=1)))

# The same can be achieved without providing axis argument
```

```
x is
[[1 2]
 [3 4]]

y is
[[ 5  6]
 [10 11]]

x+y along axis 0 is
[[ 1  2  5  6]
 [ 3  4 10 11]]
```

In [28]:
```
print("x+y along axis 1 is \n{}".format(np.concatenate((x, y), axis=-1)))

# The same can be achieved by providing axis=1
```

```
x+y along axis 1 is
[[ 1  2  5  6]
 [ 3  4 10 11]]
```

In [29]:
```
# Either subtract directly or use np.subtract

print("x-y is \n{}\n".format(x - y))

print("x-y is \n{}\n".format(np.subtract(x, y)))
```

```
x-y is
[[-4 -4]
 [-7 -7]]

x-y is
[[-4 -4]
 [-7 -7]]
```

In [31]:
```
# Hardmond or elementwise product. Either multiply or use np.multiply

print(x)
print('\n')
print(y)
print('\n')
print("Elementwise product is \n{}\n".format(x * y))

print("Elementwise product is \n{}\n".format(np.multiply(x, y)))
```

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


[[ 5  6]
 [10 11]]


Elementwise product is
[[ 5 12]
 [30 44]]

Elementwise product is
[[ 5 12]
 [30 44]]
```

In [32]:
```
# Dot Product can be done by np.dot

print("Matrix multiplication \n{}\n".format(np.dot(x,y)))
```

```
Matrix multiplication
[[25 28]
 [55 62]]
```

In [33]:
```
# Either divide directly or use np.divide

print("Elementwise product is \n{}\n".format(y / x))
print("Elementwise product is \n{}\n".format(np.divide(y, x)))
```

```
Elementwise product is
[[5.         3.        ]
 [3.33333333 2.75      ]]

Elementwise product is
[[5.         3.        ]
 [3.33333333 2.75      ]]
```

In [34]:
```
# Alternatively you can round off the numbers in division using:

print("Elementwise product is \n{}\n".format(y // x))
```

```
Elementwise product is
[[5 3]
 [3 2]]
```

Deletion

In [35]:
```
#Using np.delete

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

print("original array is \n {}\n".format(arr))

print("deleting 2nd item along column \n{}\n".format(np.delete(arr, 2, 1)))

print("deleting 2nd item along row \n{}\n".format(np.delete(arr, 2, 0)))
```

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

deleting 2nd item along column
[[ 1  2  4]
 [ 5  6  8]
 [ 9 10 12]]

deleting 2nd item along row
[[1 2 3 4]
 [5 6 7 8]]
```

Sorting

In [36]:
```
#Use np.sort

arr = np.array([[4,1],[3,2],[2,0]])
print("orginal array is \n{}\n".format(arr))

print("Sorted along the last axis \n{}\n".format(np.sort(arr)))    # sort along the last axis

print("Sorted along the first axis \n{}\n".format(np.sort(arr, axis=0)))    #sort along the first axis

print("Sorted the flattened array \n{}\n".format(np.sort(arr, axis=None)))     #sort the flattened array
```

```
orginal array is
[[4 1]
 [3 2]
 [2 0]]

Sorted along the last axis
[[1 4]
 [2 3]
 [0 2]]

Sorted along the first axis
[[2 0]
 [3 1]
 [4 2]]

Sorted the flattened array
[0 1 2 2 3 4]
```

Conditionals

In [39]:
```
# Conditionals

arr = np.array([[4,1],[3,-2],[-12,0]])
print("orginal array is \n{}\n".format(arr))

ind = np.where(arr>0)
print(ind)
arr1 = arr[ind]
print(arr1)
```

```
orginal array is
[[  4   1]
 [  3  -2]
 [-12   0]]

(array([0, 0, 1], dtype=int64), array([0, 1, 0], dtype=int64))
[4 1 3]
```

Pandas

1. Pandas library provides high perfromance usable data structures and data analysis tools for managing data tables
2. Supports standard functions to create pivot tables, column or row groupings, plotting graphs, joining tables like SQL, etc.

There are three basic data structures in the pandas library:

DataFrame: This is the main data structure and it is a 2D table, similar to excel/spreadsheet tables with columns names and row labels.

Series: It is a 1D array, similar to a column in excel or excel spreadsheet with column name and row labels.

Panel : It is dictionary of dataframes. It is generally not used in practice and we will not be discussing it in the videos.

Pandas Series

1. Similar to python list for definition purpose.
2. For operations purpose, it behaves similar to numpy one-dimensional ndarrays and can be passed directly to numpy functions.

In [41]:
```
#Creating Series
# Using Python list as series objects
s = pd.Series([-1.,-1,21,5])
s
```

Out[41]:
```
0    -1.0
1    -1.0
2    21.0
3     5.0
dtype: float64
```

In [42]:
```
# Using python list as index labels and scalar object values

s = pd.Series(40, ["Chuck", "Darwin", "Elijah"])
s
```

Out[42]:
```
Chuck     40
Darwin    40
Elijah    40
dtype: int64
```

In [43]:
```
# Using dictionary with indecies
dict1 = {"Newton": 6, "Chuck": 3, "Darwin": 8, "Elijah": 9}
s = pd.Series(dict1)
s
```

Out[43]:
```
Newton    6
Chuck     3
Darwin    8
Elijah    9
dtype: int64
```

In [44]:
```
#We can also specify the indicies explicity to control what goes into the series
s = pd.Series(dict1, index = ["Chuck", "Elijah"])
s
```

Out[44]:
```
Chuck     3
Elijah    9
dtype: int64
```

In [47]:
```
# Using numpy array

s = pd.Series(np.array([1,2,34,4]))
s
```

Out[47]:
```
0     1
1     2
2    34
3     4
dtype: int32
```

In [48]:
```
# Giving name to the series as
s = pd.Series([6, -5.4], index=["Charles", "Chuck"], name="heights")
s
```

Out[48]:
```
Charles    6.0
Chuck     -5.4
Name: heights, dtype: float64
```

Operations in Series

In [49]:
```
# Use a series in numpy functions. Here we take exponential of the series
np.abs(s)
```

Out[49]:
```
Charles    6.0
Chuck      5.4
Name: heights, dtype: float64
```

Elementwise arithmetic operations on pandas Series are similar to numpy ndarray's:

In [50]:
```
s + np.array([1000,2000])
```

Out[50]:
```
Charles    1006.0
Chuck      1994.6
Name: heights, dtype: float64
```

Broadcasting is similar to numpy. If you add a single number to Series, it is broadcasted throughout the series.

In [51]:
```
s + 1000
```

Out[51]:
```
Charles    1006.0
Chuck       994.6
Name: heights, dtype: float64
```

The same is true for all binary operations such as multiplication, division, subtraction or evern conditionals operations

Accessing Element in Series

1. Each item in a Series object has a unique identifier called the *index label*.
2. By default, it is the rank of the item in the Series (starting at 0) but you can also set the index labels manually
3. You can also use the series as a dictionary with manually set indexing as well as integer index like regular list

In [52]:
```
s
```

Out[52]:
```
Charles    6.0
Chuck     -5.4
Name: heights, dtype: float64
```

In [53]:
```
s["Chuck"]
```

Out[53]:
```
-5.4
```

In [54]:
```
s.loc["Chuck"]
```

Out[54]:
```
-5.4
```

In [55]:
```
s.iloc[1]
```

Out[55]:
```
-5.4
```

Slicing

In [56]:
```
s2 = pd.Series([1000, 1001, 1002, 1003])
s2
```

Out[56]:
```
0    1000
1    1001
2    1002
3    1003
dtype: int64
```

In [57]:
```
s2_slice = s2[1:]
s2_slice
```
Out[57]:
```
1    1001
2    1002
3    1003
dtype: int64
```

The first element has index label 1. The element with index label 0 is absent from the slice

In [58]:
```
s2_slice[0]
```
```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
~\anaconda3\lib\site-packages\pandas\core\indexes\range.py in get_loc(self, key)
    413             try:
--> 414                 return self._range.index(new_key)
    415             except ValueError as err:

ValueError: 0 is not in range

The above exception was the direct cause of the following exception:

KeyError                                  Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_9776\2969839769.py in <module>
----> 1 s2_slice[0]

~\anaconda3\lib\site-packages\pandas\core\series.py in __getitem__(self, key)
   1038 
   1039         elif key_is_scalar:
-> 1040             return self._get_value(key)
   1041 
   1042         # Convert generator to list before going through hashable part

~\anaconda3\lib\site-packages\pandas\core\series.py in _get_value(self, label, takeable)
   1154 
   1155         # Similar to Index.get_value, but we do not fall back to positional
-> 1156         loc = self.index.get_loc(label)
   1157 
   1158         if is_integer(loc):

~\anaconda3\lib\site-packages\pandas\core\indexes\range.py in get_loc(self, key)
    414                 return self._range.index(new_key)
    415             except ValueError as err:
--> 416                 raise KeyError(key) from err
    417         if isinstance(key, Hashable):
    418             raise KeyError(key)

KeyError: 0
```

But we can access elements by integer location using the iloc attribute.

In [59]:
```
s2_slice.iloc[0]
```
Out[59]:
```
1001
```

So, we should always use loc and iloc to access Series objects.

pd.DataFrame(): Creates a Pandas DataFrame from a list of dictionaries, a list of lists, or a NumPy array.

df.head(): Returns the first five rows of a Pandas DataFrame.

df.tail(): Returns the last five rows of a Pandas DataFrame.

df.shape(): Returns the shape of a Pandas DataFrame.

df.ndim(): Returns the number of dimensions of a Pandas DataFrame.

df.mean(): Calculates the mean of each column in a Pandas DataFrame.

df.median(): Calculates the median of each column in a Pandas DataFrame.

df.std(): Calculates the standard deviation of each column in a Pandas DataFrame.

df.var(): Calculates the variance of each column in a Pandas DataFrame.

df.sum(): Calculates the sum of the elements of each column in a Pandas DataFrame.

df.dot(): Calculates the dot product of two Pandas DataFrames.

In [ ]: