SAMSUNG

Together for Tomorrow!
Enabling People
Education for Future Generations

Samsung Innovation Campus

Artificial Intelligence Course

Together for Tomorrow!
Enabling People
Education for Future Generations

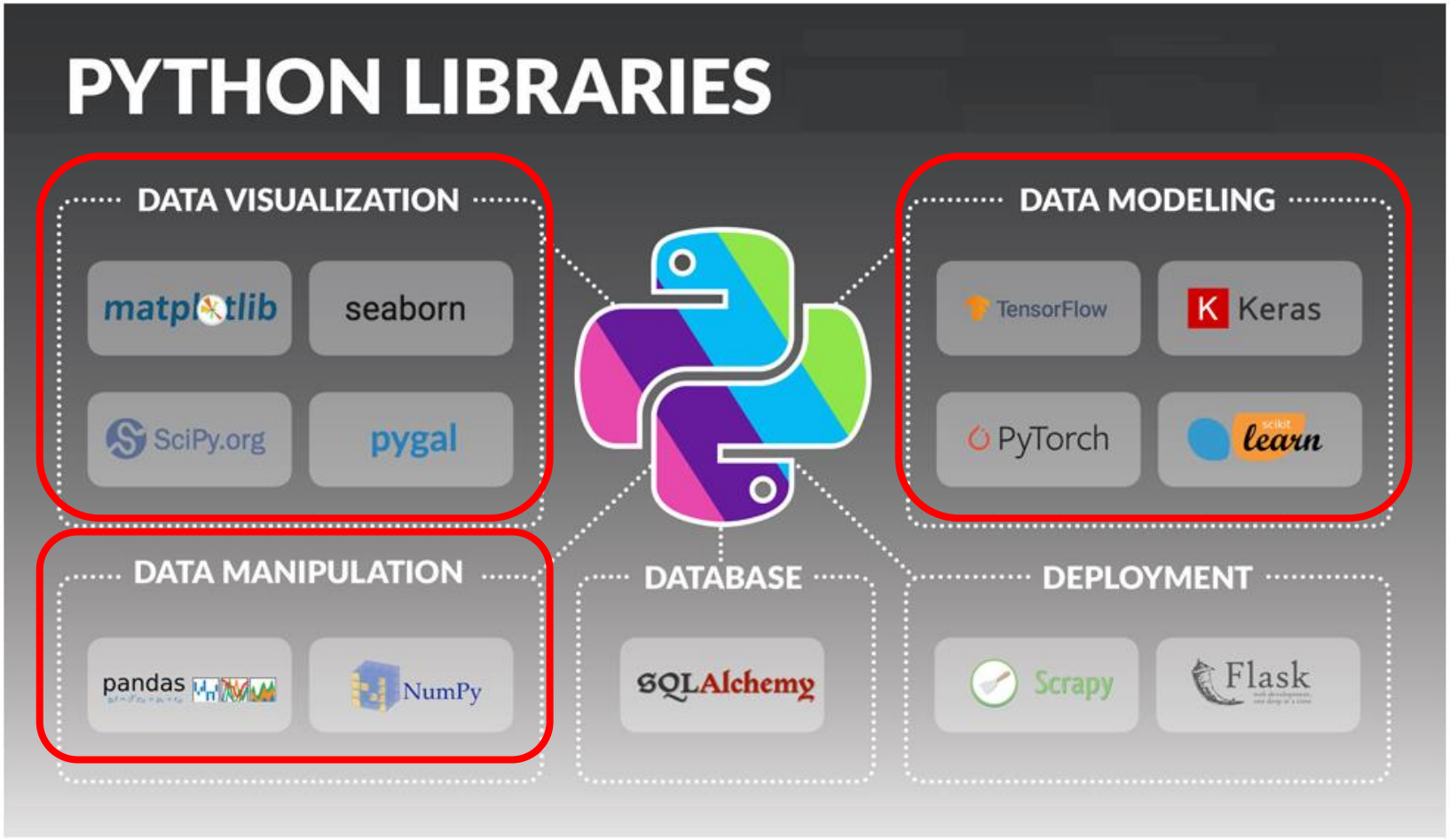
Chapter 3.

Python Libraries

AI Course

Samsung Innovation Campus

Python Libraries



Chapter Description (1/2)

Duration: 22 Hours.

Purpose:

- **Introduction to the NumPy library.**
- Introduction to linear algebra: vector and matrix operations.
- **Introduction to the Pandas library.**
- **Introduction to the Matplotlib and Seaborn libraries: visualization techniques.**

Target Audience:

- This course is intended for those who are willing to go beyond the basic Python programming.
- This course is a prerequisite for machine learning and deep learning.

Prerequisite:

- A student taking this course should be able to **program in Python.**
- A student taking this course should be somewhat familiar with vectors and matrices.
- A student taking this course should be familiar with Excel spreadsheets.

Chapter Description (2/2)

Objectives:

- **Create, manipulate and operate with NumPy arrays.**
- Understand the basics of linear algebra.
- Create, manipulate and operate with Series objects.
- **Create, manipulate and operate with DataFrame objects.**
- **Utilize data summarization, aggregation and transformation techniques.**
- **Create basic as well as advanced visualization types.**
- Carry out exploratory data analysis (EDA).

Together for Tomorrow!
Enabling People
Education for Future Generations

Chapter 3.

Python Libraries

UNIT 1.

NumPy Package

Samsung Innovation Campus

Unit 1.
NumPy Package

What this unit is about:

- This unit is an **introduction to the NumPy** library.
- You will learn how to **create NumPy arrays**.
- You will learn how to **manipulate and operate with NumPy arrays**.
- You will learn how to carry out **linear algebra operations** with NumPy arrays.

Expected outcome:

- Ability to **utilize NumPy arrays for holding structured data**.
- Ability to **transform and summarize data** contained in the NumPy arrays.
- Ability to carry out **linear algebra operations** with the NumPy arrays.

How to check your progress:

- Coding Exercises.
- Quiz.

Python Libraries

NumPy Library (1/2)

What is NumPy?

- An open source Python library that supports **multi-dimensional arrays**.
- Provides **high-level mathematical functions**.
- Supports **fast vectorized operations** on arrays.

More information about the NumPy library can be found at https://numpy.org/

NumPy Library (2/2)

NumPy array compared to Python list:

- **NumPy arrays** are mutable like lists.
- However, unlike lists the elements of a NumPy array should have the **same data type**.
- NumPy array is the standard input/output format for many **machine learning libraries**.
- NumPy arrays come with many **attributes and methods** not found in the Python list.

NumPy array basics (1/19)

Creating NumPy arrays from lists and tuples:

```
In[1] : import numpy as np                  # Import the NumPy package and abbreviate its name as 'np'.
In[2] : np.__version__                      # Currently installed NumPy version.
Out[2]: '1.11.3'
In[3] : arr1 = np.array([1,3,5,7,9])        # Create an array from a list.
In[4] : arr2 = np.array((1,3,5,7,9))        # Create an array from a tuple.
In[5] : type(arr1)
Out[5]: numpy.ndarray
In[6] : arr3 = arr1                         # Simple assignment.
In[7] : id(arr1)
Out[7]: 93269488L
In[8] : id(arr3)
Out[8]: 93269488L                  # arr1 and arr3 refer to the same object.
```

NumPy array basics (2/19)

Making a copy of a NumPy array:

```
In[9]  : arr4 = arr1.copy()              # Make a shallow copy.
In[10] : id(arr1)
Out[10]: 93269488L
In[11] : id(arr4)
Out[11]: 93494048L                           # arr4 is a completely different object.
```

NumPy array basics (3/19)

The elements of a NumPy array **must have the same data type**:

```
In[1] : arr1 = np.array([111, 2.3,  True, False, False])      # Numeric and Boolean values entered.
In[2] : arr1                                                  # Automatic conversion to the float type.
Out[2]: ([111., 2.3,  1., 0., 0.])
In[3] : arr2 = np.array([111, 2.3, 'python', 'abc'])          # Numeric and string values entered.
In[4] : arr2                                                  # Automatic conversion to the string type.
Out[4]: array(['111', '2.3', 'python' , 'abc'],
          dtype='|S32')
In[5] : arr3 = np.array([111, True, 'abc'])                   # Numeric, Boolean and string values entered.
In[6] : arr3                                                  # Automatic conversion to the string type.
Out[6]: array(['111', 'True', 'abc'],
          dtype='|S32')
```

NumPy array basics (4/19)

Data types for NumPy arrays:

Data Type	Explanation
int8, int16, int32, int64, int_ uint8, uint16, uint32, uint64	Integer
float16, float32, float64, float128, float_	Floating point
bool_	Boolean
string_, unicode_	String

More information can be found at https://docs.scipy.org/doc/numpy-1.17.0/reference/arrays.dtypes.html

NumPy array basics (5/19)

Creating NumPy arrays: **arrange()** function.

Creating NumPy array from a **Sequence of numbers**

```
In[1]  : np.arange(10)                                  # Array corresponding to the range 0~9.
Out[1] : array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
In[2]  : np.arange(10,20)                  # Array corresponding to the range 10~19.
Out[2] : array([10, 11, 12, 13, 14, 15, 16, 17, 18, 19])
In[3]  : np.arange(10, 20, 3)                           # Array corresponding to the range10~19, step = 3.
Out[3] : array([10, 13, 16, 19])
In[4]  : arr = np.arange(10)
In[5]  : len(arr)                                       # Returns the length. Same as with Python list.
Out[5] : 10
In[5]  : arr.size                                       # Attribute 'size' equals the number of elements.
Out[5] : 10
```

NumPy array basics (6/19)

Creating NumPy arrays: **linspace()** function.

```
In[1]  : np.linspace(0, 10, 5)                    # Equally spaced 5 numbers between 0 and 10.
Out[1] : array([0.0, 2.5, 5.0, 7.5, 10.0])
```

NumPy array basics (7/19)

Creating NumPy arrays: **zeros()** function.

```
In[1]  : np.zeros(10)
Out[1] : array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])      # 10 zeros.
In[2]  : np.zeros((3,4))                                      # The shape is given as a tuple.
Out[2] :
array([[ 0.,  0.,  0.,  0.],                                 # 3 x 4 matrix.
       [ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.]])
In[3]  : np.zeros(5, dtype ='int64')                          # Specified the data type of the elements.
Out[3] : array([0, 0, 0, 0, 0], dtype=int64)
In[4]  : np.zeros(5, dtype ='int64').dtype
Out[4] : dtype('int64')
```

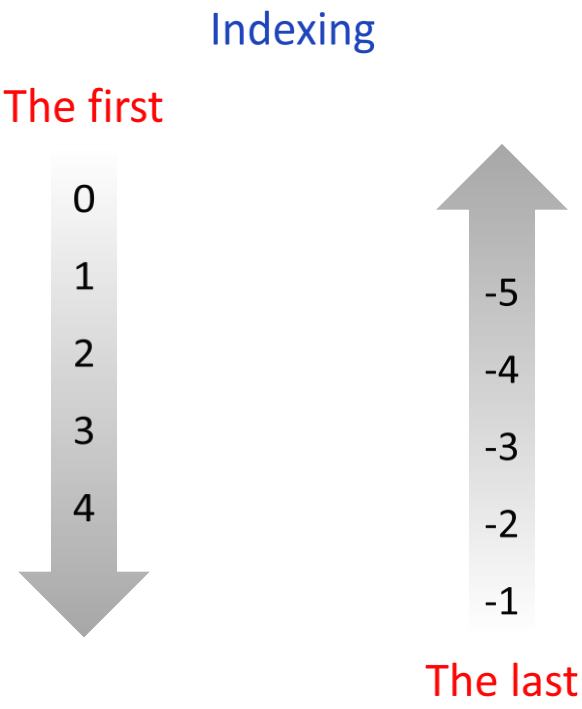
NumPy array basics (8/19)

Creating NumPy arrays: **ones()** function.

```
In[1]  : arr = np.ones((2,3), dtype ='int_')          # The shape given as a tuple. Data type is specified.
In[2]  : arr
Out[2] :
array([[ 1,  1,  1],
       [ 1,  1,  1]])
In[3]  : arr.dtype
Out[3] : dtype('int32')
In[4]  : arr.astype('float32')                        # Re-cast the data type as 'float32'.
Out[4] :
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.]], dtype=float32)
```

NumPy array basics (9/19)

Indexing and slicing NumPy arrays:



NumPy array basics (10/19)

Indexing and slicing NumPy arrays:

```
In[1] : a = np.array([1, 2, 3, 4, 5])      # A one dimensional NumPy array.
In[2] : a[:2]                              # From the index 0 up to the index 1 (excluding 2).
Out[2]: array([1, 2])
In[3] : a[-1]                              # The first element from last.
Out[3]: 5
In[4] : a[:]                               # The whole array.
Out[4]: array([1, 2, 3, 4, 5])
```

NumPy array basics (11/19)

Indexing and slicing NumPy arrays:

```
In[1] : a = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])   # This is a 3 x 3 two dimensional array (matrix).
In[2] : a[1]                                              # The row 1.
Out[2]: array([4, 5, 6])
In[3] : a[-1]                                             # The last row.
Out[3]: array([7, 8, 9])
In[4] : a[:]                                              # The whole array (matrix).
Out[4]:
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
In[5] : a[:2]                                             # From the row 0 to the row 1 (excluding the row 2).
Out[5]:
array([[1, 2, 3],
       [4, 5, 6]])
```

NumPy array basics (12/19)

Indexing and slicing NumPy arrays:

```
In[6]  : a[2][1]                    # The element from the row 2 and the column 1.
Out[6]: 8
In[7]  : a[2, 1]                    # The same as above.
Out[7]: 8
In[8]  : a[[0, 2]]                  # The rows 0 and 2.
Out[8]:
array([[1, 2, 3],
       [7, 8, 9]])
In[9]  : a[1:, 1:]                  # From the row 1 to the end. From the column 1 to the end.
Out[9]:
array([[5, 6],
       [8, 9]])
```

NumPy array basics (13/19)

Fancy indexing:

```
In[1]  : arr = np.arange(100)                          # A sequence from 0 to 99.
In[2]  : arrMask = ((arr % 5) == 0)                    # A Boolean array. True if multiple of 5, else False.
In[3]  : arr[arrMask]                                  # Apply fancy indexing.
Out[3]:
array([ 0,  5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70, 75, 80,85, 90, 95])
In[4]  : arrMask = ( ( (arr % 5) == 0 )  & ( arr >0) )  # A Boolean array. True if multiple of 5 or 0, else False.
In[5]  : arr[arrMask]                                  # Apply fancy indexing.
Out[5]:
array([ 5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70, 75, 80, 85,90, 95])
```

NumPy array basics (14/19)

NumPy array attributes:

```
In[10] : a.size                    # The total number of elements.
Out[10]: 9
In[11] : a.shape                   # The shape: number of rows and columns.
Out[11]: (3, 3)
In[12] : a.ndim                    # The number of dimensions.
Out[12]: 2
```

More information on array attributes and methods can be found at
https://docs.scipy.org/doc/numpy-1.17.0/reference/generated/numpy.ndarray.html

NumPy array basics (15/19)

Reshaping NumPy arrays:

```
In[1]  : a = np.arange(15)                                  # A sequence from 0 to 14. A 1D array.
In[2]  : a.reshape(3,5)                                     # Reshape it as a 3 x 5 matrix (2D array). This is just a view.
Out[2]:
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
In[3]  : a                                                  # 'a' is not reshaped permanently.
Out[3]:  array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14])
In[4]  : a.shape = (3, 5)                                   # Change directly the 'shape' attribute.
In[5]  : a                                                  # The reshaping is now permanent.
Out[5]:
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
```

NumPy array basics (16/19)

Reshaping NumPy arrays:

```
In[1]  : a = np.array([2,5,1,3])
In[2]  : a.shape
Out[2]: (4, )                          # This is a 1D array (rank 1).
In[3]  : a = a.reshape(4,1)            # After reshaping, 'a' is reassigned.
In[4]  : a                             # Now, a single column matrix.
Out[4]:
array([ [2],
        [5],
        [1],
        [3]])
In[5]  : a.shape                       # The reshaping is permanent.
Out[5]: (4,1)
```

NumPy array basics (17/19)

Reshaping NumPy arrays:

```
In[1]  : a = np.arange(10)                    # A sequence from 0 to 9. A 1D array (rank 1).
In[2]  : b = a.reshape(2,5)                   # Reshape as 2 x 5 then assign to 'b'.
In[3]  : a[0] = -999
In[4]  : b
Out[4]:
array([[ -999,  1,  2,  3,  4],              # You can see that 'a' and 'b' share the same memory space!
       [ 5,  6,  7,  8,  9]])
In[5]  : c = a.reshape(2,5).copy()            # Now, make a shallow copy.
In[6]  : c[0, 0] = 0
In[7]  : a
Out[7]:
array([ -999, 1, 2, 3, 4, 5, 6, 7, 8, 9])    # 'a' and 'c' are completely different objects.
```

NumPy array basics (18/19)

Appending elements to NumPy arrays:

```
In[1] : a = np.array([1, 2, 3])                    # A 1D array (rank 1).
In[2] : b = np.append(a, [4, 5, 6])                # Append new values and create a new array object.
In[3] : b
Out[3]: array([1, 2, 3, 4, 5, 6])
```

```
In[1] : a = np.array([[1, 2], [3, 4]])             # A 2D array.
In[2] : b = np.append(a, [[9, 9]], axis=0) # Append a row and create a new array object.
In[3] : c = np.append(a, [[9], [9]] , axis=1)      # Append a column and create a new array object.
```

NumPy array basics (19/19)

Removing elements from NumPy arrays:

```
In[1] : a = np.array([1, 2, 3,4, 5, 6])
In[2] : np.delete(a, 0)                    # Delete the element at position 0. Just a view.
Out[2]: array([2, 3, 4, 5, 6])
```

```
In[3] : np.delete(a, (0, 2, 4))            # Delete the elements at positions 0, 2 and 4. Just a view.
Out[3]: array([2, 4, 6])
```

```
In[4] : np.delete(x, 0, axis = 0)          # Delete the row 0. Just a view.
Out[4]: array([4, 5, 6])
```

```
In[5] : np.delete(x, 1, axis = 1)          # Delete the column 1. Just a view.
Out[5]:
array([[1, 3],
       [4, 6]])
```

Coding Exercise #0201

Follow practice steps on 'ex_0201.ipynb' file

NumPy array basics

Practice Questions

Create a NumPy array. Array should have 25 equally space values from 1 to 50. Display Array data type and shape. Display only those values in the array that are greater than 20. Reshape the array as a 5x5 matrix. Delete 2nd column from the matrix. Append a new row at the end. Display resultant matrix

Python Libraries

NumPy array operations (1/9)

Comparing **Python list and NumPy arrays**: **'+' operator**.

```
In[1] : a = [1, 2, 3]
In[2] : b = [4, 5, 6]
In[3] : a + b                    # For lists, '+' means concatenation.
Out[3]: [1, 2, 3, 4, 5, 6]
```

```
In[1] : a = np.array([1, 2, 3])
In[2] : b = np.array([4, 5, 6])
In[3] : a + b                    # For NumPy arrays, '+' means element-wise addition.
Out[3]: array([5, 7, 9])
```

NumPy array operations (2/9)

Comparing **Python list and NumPy** arrays: **'*' operator**.

```
In[1] : a = [1, 2, 3]
In[2] : 3 * a                                  # For lists, '*' means repetition.
Out[2]: [1, 2, 3, 1, 2, 3, 1, 2, 3]
```

```
In[3] : b = np.array([1, 2, 3])
In[4] : 3 * b                                  # For NumPy arrays, '*' means multiplication of the elements.
Out[4]: array([3, 6, 9])
In[5] : np.array(3 * [1, 2, 3])
Out[5]: array([1, 2, 3, 1, 2, 3, 1, 2, 3])
In[6] : np.repeat(b, 3)                        # Use repeat() function to repeat each element.
Out[6]: array([1, 1, 1, 2, 2, 2, 3, 3, 3])
In[7] : np.tile(b, 3)                          # Use tile() function to repeat the sequence.
Out[7]: array([1, 2, 3, 1, 2, 3, 1, 2, 3])
```

NumPy array operations (3/9)

Operations with NumPy arrays: **Element-wise Operantions**

```
In[1] : a = np.array([1, 2, 3])
In[2] : b = np.array([4, 5, 6])
In[3] : a + b
Out[3]: array([5, 7, 9])
In[4] : b - a
Out[4]: array([3, 3, 3])
In[5] : a * b
Out[5]: array([4, 10, 18])
In[6] : 1.0*a / b
Out[6]: array([0.25, 0.4, 0.5])
```

NumPy array operations (4/9)

Universal functions:

- A universal function takes in an array and applies to each element.
- In NumPy, universal functions are instances of the numpy.ufunc class.

```
In[1] : x = np.array([0, 1, 2, 3])
In[2] : x**3
Out[2]: array([ 0,  1,  8, 27])
In[3] : np.sqrt(x)                                  # Sqrt Operation.
Out[3]: array([ 0., 1., 1.41421356, 1.73205081])
In[4] : np.exp(x)
Out[4]: array([ 1. , 2.71828183, 7.3890561 , 20.08553692])
In[5] : np.sin(x)
Out[5]: array([0. , 0.84147098, 0.90929743, 0.14112001])
In[6] : type(np.sin)                                # Check the type.
Out[6]: numpy.ufunc
```

NumPy array operations (5/9)

NumPy functions:

Function	Explanation	Universal function?
sin, cos, tan	Trigonometric functions.	Yes
arcsin, arccos, arctan	Inverse trigonometric functions.	Yes
round	Round to a given number of decimals.	Yes
floor	Returns the nearest **smaller** integer.	Yes
ceil	Returns the nearest **greater** integer.	Yes
fix	Returns the nearest integer closer to 0.	Yes
prod	Returns the product of the array elements.	**No**
cumsum	Returns the cumulative sums of an array.	**No**
sum, mean, var, std, median	Statistical functions.	**No**
exp, log	Exponential and logarithmic functions.	Yes
unique	Returns the unique values of an array.	**No**
min, max, argmax, argmin	Minimum, maximum and the corresponding indices.	**No**

More information can be found at https://docs.scipy.org/doc/numpy-1.17.0/reference/ under “routines”

NumPy array operations (6/9)

Statistical methods of NumPy arrays:

Method	Explanation
mean	Average.
var	Variance.
std	Standard deviation.
sum	Total sum.
cumsum	The cumulative sums of an array.
max, min	The maximum and the minimum of an array.
argmax, argmin	Indices of the maximum and the minimum.

More information can be found at https://docs.scipy.org/doc/numpy-1.17.0/reference/

NumPy array operations (7/9)

Statistical methods of NumPy arrays:

```
In[1] : x = np.arange(1,11)
In[2] : x.sum()
Out[2]: 55                                                    # Sum of the elements.
In[3] : x.mean()
Out[3]: 5.5                                                   # Average.
In[4] : x.std()
Out[4]: 2.87228                                               # Standard deviation.
In[5] : x.var()
Out[5]: 8.25                                                  # Variance.
In[6] : x.cumsum()
Out[6]: array([1, 3, 6, 10, 15, 21, 28, 36, 44, 55], dtype=int32)   # Cumulative sums given as an array.
```

NumPy array operations (8/9)

Statistical methods of NumPy arrays:

```
In[1] : x = np.arange(1,10)
In[2] : x = x.reshape((3,-1))                    # Shape (3,-1) means (3,3).
In[3] : print(x)
Out[3]:
[ [1  2  3]
  [4  5  6]
  [7  8  9]]
In[4] : print(x.mean(axis=0))                    # Column averages.
Out[4]:
[ 4. 5. 6.]
In[5] : print(x.mean(axis=1))                    # Row averages.
Out[5]:
[ 2. 5. 8.]
```

NumPy array operations (9/9)

Statistical methods of NumPy arrays:

```
In[1] : np.random.seed(123)
In[2] : x = np.random.randint(10, size=1000)          # 1000 random integers from the range 0~9.
In[3] : x.max()
Out[3]: 9
In[4] : x.min()
Out[4]: 0
In[5] : (x > 5).sum()                                 # Count the number of elements larger than 5.
Out[5]: 401
```

Coding Exercise #0202

Follow practice steps on 'ex_0202.ipynb' file

NumPy array operations

Practice Questions:

Create two NumPy arrays. One array should contain Student Marks in Quiz 1 (Out of 15). Second Array should contain student marks in Assignment 1 (Out of 50).
Display maximum obtained marks both in Quiz and Assignment. Count number of students that have marks greater than 10 in Quiz. Count number of students that have marks greater than 30 in assignment. Compute and display average and standard deviation of both quiz and assignment marks. Scale both quiz and assignment marks to 10 and round the values. Add both the marks.

Chapter 3.

Python Libraries

Linear Algebra (1/25)

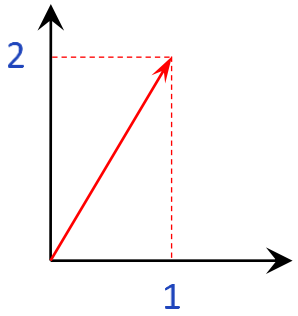
Scalars and Vectors:

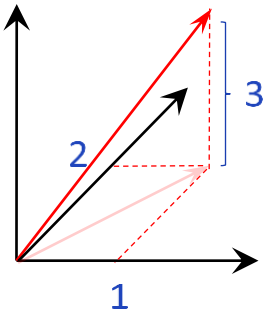
- Scalars are given as single numerical values.

  Ex). 7 3.141 −2 9.083

- Vectors have direction as well as magnitude. They are represented by arrays.

  Ex). (1, 2) (1, 2, 3)





Linear Algebra (2/25)

Vector operations: addition and subtraction.

Ex).

$$(6, 5, 4) + (3, 2, 1) = (6 + 3, 5 + 2, 4 + 1) = (9, 7, 5)$$

$$(6, 5, 4) - (3, 2, 1) = (6 - 3, 5 - 2, 4 - 1) = (3, 3, 3)$$

Linear Algebra (3/25)

Vector operations: element-wise multiplication.

- The '*' operator means element-wise multiplication.
- This is not the inner product of linear algebra.

Ex).

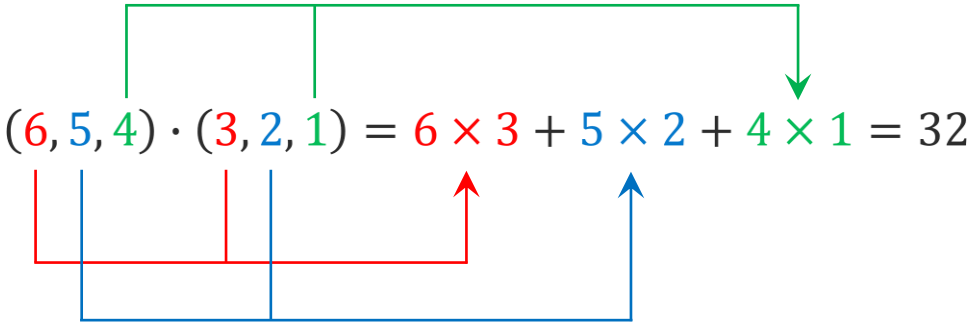
$$(6, 5, 4) * (3, 2, 1) = (6 \times 3, 5 \times 2, 4 \times 1) = (18, 10, 4)$$

Linear Algebra (4/25)

Vector operations: inner product.

- The inner product of linear algebra is carried out in the following way.

Ex).



- Use the dot() method or function of NumPy.

Linear Algebra (5/25)

Vector operations: multiplication and division by a scalar.

Ex).

$$2 \times (1, 2, 3) = (2 \times 1, 2 \times 2, 2 \times 3) = (2, 4, 6)$$

$$(2, 4, 6)/2 = (2/2, 4/2, 6/2) = (1, 2, 3)$$

Linear Algebra (6/25)

Vector operations with NumPy arrays:

```
In[1] : x = np.array([1, 3, 5])          # An array (vector) of length 3.
In[2] : y = np.array([2, 4, 6])          # An array (vector) of length 3.
In[3] : x + y                            # Vector addition.
Out[3]: array([ 3, 7, 11])
In[4] : x – y                            # Vector subtraction.
Out[4]: array([ -1, -1, -1])
In[5] : x * y                            # Element-wise multiplication.
Out[5]: array([ 2, 12, 30])
In[6] : np.dot(x, y)                     # Vector inner product with the NumPy dot() function.
Out[6]: 44
In[7] : x.dot(y)                         # Vector inner product with the dot() method.
Out[7]: 44
```

Linear Algebra (7/25)

Matrix:

- Matrix is a rectangular array of numbers arranged in rows and columns:

  Ex).

$$\begin{bmatrix} 3 \\ 7 \\ 5 \end{bmatrix} \qquad \begin{bmatrix} 2 & 3 \\ 7 & 1 \end{bmatrix} \qquad \begin{bmatrix} 1 & 5 & 2 \\ 7 & 6 & 4 \\ 8 & 0 & 10 \end{bmatrix}$$

- Flip a matrix over the diagonal, exchanging the row and the column indices. This is the **transpose**:

  Ex).

$$\begin{bmatrix} 5 \\ 3 \\ 1 \end{bmatrix}^t = \begin{bmatrix} 5 & 3 & 1 \end{bmatrix} \qquad \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}^t = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

Linear Algebra (8/25)

NumPy array as **matrix**:

```
In[1] : np.zeros((2,3))                    # Create a matrix filled with zero. The shape is given as a tuple.
Out[1]:
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
In[2] : np.ones((2,3))                     # Create a matrix filled with one. The shape is given as a tuple.
Out[2]:
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
```

Linear Algebra (9/25)

NumPy array as matrix:

```
In[1] : np.random.seed(123)                    # Initialize the random seed (123=seed).
In[2] : np.random.random((2,2))                # A matrix with values drawn from the uniform distribution.
Out[2]:
array([[ 0.60050608,  0.0590288 ],
       [ 0.00072301,  0.72163516]])
In[3] : np.random.randn(2,2)                   # A matrix with values drawn from the standard normal distribution.
Out[3]:
array([[-1.3059906 ,  0.98549986],
       [-0.97344165, -0.89474788]])
```

Linear Algebra (10/25)

NumPy array as matrix:

```
In[1] : m = np.diag([1,2,3])                    # Create a diagonal matrix.
In[2] : m
Out[2]:
array([[1, 0, 0],
       [0, 2, 0],
       [0, 0, 3]])
In[3] : np.diag(m)                              # Extract the diagonal elements from a matrix.
Out[3]: array([1, 2, 3])
```

Linear Algebra (11/25)

Matrix operations: addition and subtraction.

Ex).

$$\begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix} + \begin{bmatrix} 5 & 7 \\ 6 & 8 \end{bmatrix} = \begin{bmatrix} 1+5 & 3+7 \\ 2+6 & 4+8 \end{bmatrix} = \begin{bmatrix} 6 & 10 \\ 8 & 12 \end{bmatrix}$$

$$\begin{bmatrix} 6 & 10 \\ 8 & 12 \end{bmatrix} - \begin{bmatrix} 5 & 7 \\ 6 & 8 \end{bmatrix} = \begin{bmatrix} 6-5 & 10-7 \\ 8-6 & 12-8 \end{bmatrix} = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$$

Linear Algebra (12/25)

Matrix operations: addition and subtraction.

```
In[1] : m1 = np.array([[1, 2, 3],[4, 5, 6]])        # A 2 x 3 matrix.
In[2] : m2 = np.array([[6, 5, 4],[3, 2, 1]])        # Another 2 x 3 matrix.
In[3] : m1 + m2                                     # Matrix addition.
Out[3]:
array([[7, 7, 7],
       [7, 7, 7]])
In[4] : m1 – m2                                     # Matrix subtraction.
Out[4]:
array([[-5, -3, -1],
       [ 1,  3,  5]])
```

Linear Algebra (13/25)

NumPy array operations: element-wise **multiplication and division**.

- The '*' and '/' operate element-wise.
- The '*' is not the usual matrix multiplication of linear algebra.
- There is no equivalent to the '/' operator in linear algebra.

Ex).

$$\begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix} * \begin{bmatrix} 7 & 9 & 11 \\ 8 & 10 & 12 \end{bmatrix} = \begin{bmatrix} 1 \times 7 & 3 \times 9 & 5 \times 11 \\ 2 \times 8 & 4 \times 10 & 6 \times 12 \end{bmatrix} = \begin{bmatrix} 7 & 27 & 55 \\ 16 & 40 & 72 \end{bmatrix}$$

Ex).

$$\begin{bmatrix} 7 & 27 & 55 \\ 16 & 40 & 72 \end{bmatrix} / \begin{bmatrix} 7 & 9 & 11 \\ 8 & 10 & 12 \end{bmatrix} = \begin{bmatrix} 7/7 & 27/9 & 55/11 \\ 16/8 & 40/10 & 72/12 \end{bmatrix} = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix}$$

Linear Algebra (14/25)

NumPy array operations: element-wise **multiplication and division**.

- The '*' and '/' operate element-wise.
- The '*' is not the usual matrix multiplication of linear algebra.
- There is no equivalent to the '/' operator in linear algebra.

```
In[1] : m1 = np.array([[1, 2, 3],[4, 5, 6]])        # A 2 x 3 matrix.
In[2] : m2 = np.array([[6, 5, 4],[3, 2, 1]])        # Another 2 x 3 matrix.
In[3] : m1 * m2
Out[3]:
array([[ 6, 10, 12],
       [12, 10,  6]])
```

Linear Algebra (15/25)

Matrix operations: multiplication.

- The matrix multiplication of linear algebra is carried out in the following way.

Ex).

$$\begin{bmatrix} 4 & 5 & 6 \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} = 4 \times 1 + 5 \times 2 + 6 \times 3 = 32$$

Linear Algebra (16/25)

Matrix operations: multiplication.

- The matrix multiplication of linear algebra is carried out in the following way.

Ex).

$$\begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix} \cdot \begin{bmatrix} 7 & 8 \\ 9 & 10 \\ 11 & 12 \end{bmatrix} = \begin{bmatrix} 1 \times 7 + 3 \times 9 + 5 \times 11 & 1 \times 8 + 3 \times 10 + 5 \times 12 \\ 2 \times 7 + 4 \times 9 + 6 \times 11 & 2 \times 8 + 4 \times 10 + 6 \times 12 \end{bmatrix}$$

$$= \begin{bmatrix} 89 & 98 \\ 116 & 128 \end{bmatrix}$$

Linear Algebra (17/25)

Matrix operations: multiplication.

- The matrix multiplication of linear algebra is carried out in the following way.

Ex).

$$\begin{bmatrix}4\\5\\6\end{bmatrix} \cdot \begin{bmatrix}1 & 2 & 3\end{bmatrix} = \begin{bmatrix}4\times1 & 4\times2 & 4\times3\\5\times1 & 5\times2 & 5\times3\\6\times1 & 6\times2 & 6\times3\end{bmatrix} = \begin{bmatrix}4 & 8 & 12\\5 & 10 & 15\\6 & 12 & 18\end{bmatrix}$$

Linear Algebra (18/25)

Matrix operations: multiplication.

- Again, use the dot() method or function of NumPy to do the matrix multiplication.
- The number of columns of the left matrix = the number of rows of the right matrix.

```
In[1] : m1 = np.array([[1, 2, 3],[4, 5, 6]]) # A 2 x 3 matrix.
In[2] : m2 = np.array([[6, 5, 4],[3, 2, 1]]) # Another 2 x 3 matrix.
In[3] : np.transpose(m2)                     # Transpose a 2 x 3 matrix to get another of shape 3 x 2.
Out[3]:
array([[6, 3],
       [5, 2],
       [4, 1]])
In[4] : np.dot(m1 , np.transpose(m2))                # Multiply 2 x 3 matrix by another of shape 3 x 2.
Out[4]:
array([[28, 10],                                     # The result is a matrix of shape 2 x 2.
       [73, 28]])
```

Linear Algebra (19/25)

Matrix operations: multiplication and division by a scalar.

Ex).

$$2 \times \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} = \begin{bmatrix} 2 \times 1 & 2 \times 2 & 2 \times 3 \\ 2 \times 4 & 2 \times 5 & 2 \times 6 \end{bmatrix} = \begin{bmatrix} 2 & 4 & 6 \\ 8 & 10 & 12 \end{bmatrix}$$

$$\begin{bmatrix} 2 & 4 & 6 \\ 8 & 10 & 12 \end{bmatrix} /2 = \begin{bmatrix} 2/2 & 4/2 & 6/2 \\ 8/2 & 10/2 & 12/2 \end{bmatrix} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

Linear Algebra (20/25)

Matrix operations: multiplication and division by a scalar.

```
In[1] : m = np.array([[1, 2],[3, 4]])
In[2] : 3 * m                                   # Multiply a matrix by a scalar value.
Out[2]:
array([[3, 6],
       [9, 12]])
In[3] : m / 2.0                                 #  Divide a matrix by a scalar value.
Out[3]:
array([[0.5, 1.0],
       [1.5, 2.0]])
```

Linear Algebra (21/25)

Inverse matrix:

- The inverse of a matrix $\boldsymbol{A}$ is such that when multiplied to itself gives the **identity matrix**.
- The inverse matrix does not always exist: only square non-singular matrices can be inverted.

Ex).

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix} \cdot \begin{bmatrix} -2 & 1 \\ 1.5 & -0.5 \end{bmatrix} = \begin{bmatrix} -2+3 & 1-1 \\ -6+6 & 3-2 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

A A^{-1} I

Inverse matrix of A

Linear Algebra (22/25)

Inverse matrix:

```
In[1] : m = np.array([[1, 2],[3, 4]])          # A square matrix of shape 2 x 2.
In[2] : minv = np.linalg.inv(m)                # Inverse of 'm'.
In[3] : minv
Out[3]:
array([[-2. ,  1. ],
       [ 1.5, -0.5]])
In[4] : mres = np.dot(m, minv)                 # Do matrix multiplication for a check.
In[5] : np.round(mres, 2)                      # The result is the 2 x 2 identity matrix.
Out[5]:
array([[1., 0.],
       [0., 1.]])
```

Linear Algebra (23/25)

System of linear equations:

- Suppose that you have m equations and n unknowns.

$$a_{11}x_1 + a_{12}x_2 + a_{13}x_3 \dots a_{1n}x_n = b_1$$
$$a_{21}x_1 + a_{22}x_2 + a_{23}x_3 \dots a_{2n}x_n = b_2$$
$$\vdots$$
$$a_{m1}x_1 + a_{m2}x_2 + a_{m3}x_3 \dots a_{mn}x_n = b_m$$

- There can be three cases:

1). There are same number of equations as unknowns (m=n): if there is a solution, it is unique.

2). There are more equations than unknowns (m>n): you can get an approximate solution by least squares method. This case is known as “overdetermined”.

3). There are more unknowns than equations (m<n): you can get an infinity of nontrivial solutions if b_i=0. This case is known as “underdetermined”.

Linear Algebra (24/25)

System of linear equations when the number of equations equals the number of unknowns:

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 \ldots a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 \ldots a_{2n}x_n &= b_2 \\ &\vdots \\ a_{m1}x_1 + a_{m2}x_2 + a_{m3}x_3 \ldots a_{mn}x_n &= b_m \end{aligned}$$

- Let us use matrix notation:

$$\boldsymbol{A\,x = b}$$

$$\boldsymbol{A} = \begin{bmatrix} a_{11} & \cdots & a_{1m} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mm} \end{bmatrix} \qquad \boldsymbol{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix} \qquad \boldsymbol{b} = \begin{bmatrix} b_1 \\ \vdots \\ b_m \end{bmatrix}$$

- The solution can be calculated using the inverse matrix:

$$\boldsymbol{x = A^{-1}b}$$

Linear Algebra (25/25)

System of linear equations when the number of equations equals the number of unknowns:

```
In[1] : A = np.array([[5, 8],[6, 4]])          # The matrix 'A' which is of square shape.
In[2] : b = np.array([[30], [25]])
In[3] : Ainv = np.linalg.inv(A)                # The inverse of 'A'.
In[4] : np.dot(Ainv,b)                         # The solution by matrix multiplication.
Out[4]:
array([[ 2.85714286],
       [ 1.96428571]])
In[5] : np.linalg.solve(A, b)                  # The solution by using np.linalg.solve() function.
Out[5]:
array([[ 2.85714286],
       [ 1.96428571]])
```

Coding Exercise #0203

Follow practice steps on 'ex_0203.ipynb' file