Machine Learning Diploma

Session2: Numpy & Scipy

AMIT

Agenda

- ➔ Numpy.
- ➔ Revisiting matrix arithmetics in numpy.
- ➔ Types of matrices in numpy.
- ➔ Matrix Operations.
- ➔ Sparse Matrices
- ➔ Tensors & tensors arithmetic
- ➔ Matrix Factorization
- ➔ Eigen decomposition

1. Numpy

Numpy:

➔ Arrays are the main data structure used in machine learning. In Python, arrays from the NumPy library, called N-dimensional arrays or the ndarray, are used as the primary data structure for representing data.

➔ NumPy is a Python library that can be used for scientific and numerical applications and is the tool to use for linear algebra operations.

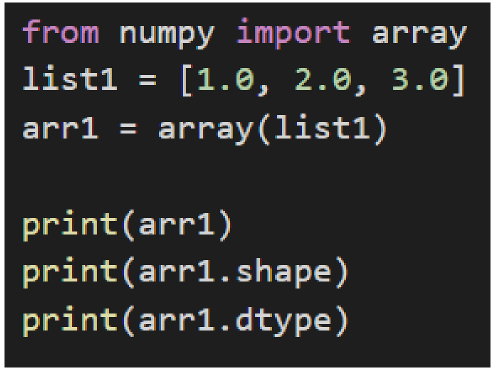
➔ When working with NumPy, data in an ndarray is simply referred to as an array. It is a fixed-sized array in memory that contains data of the same type, such as integers or floating point values

Numpy:

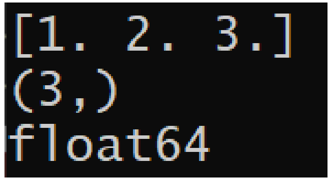
➔ The data type supported by an array can be accessed via the dtype attribute on the array.

➔ The dimensions of an array can be accessed via the shape attribute that returns a tuple describing the length of each dimension.

➔ To create an array from data or simple Python data structures like a list is to use the array() function.



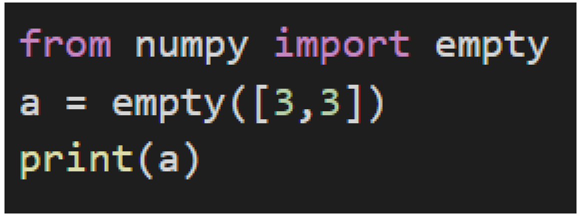
```
from numpy import array
list1 = [1.0, 2.0, 3.0]
arr1 = array(list1)

print(arr1)
print(arr1.shape)
print(arr1.dtype)
```



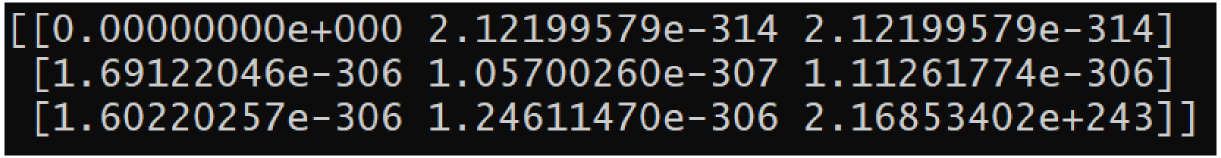
```
[1. 2. 3.]
(3,)
float64
```

Numpy functions: empty()

- ➔ The empty() function will create a new array of the specified shape.
- ➔ The argument to the function is an array or tuple that specifies the length of each dimension of the array to create.
- ➔ It will give you random numbers each time.



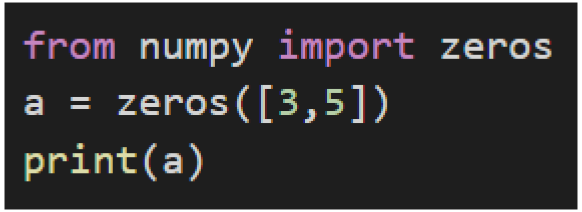
```
from numpy import empty
a = empty([3,3])
print(a)
```



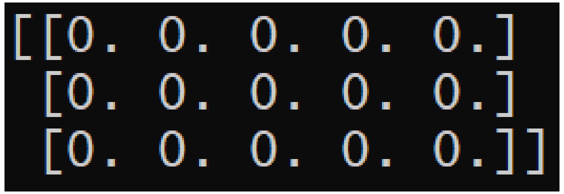
```
[[0.00000000e+000 2.12199579e-314 2.12199579e-314]
 [1.69122046e-306 1.05700260e-307 1.11261774e-306]
 [1.60220257e-306 1.24611470e-306 2.16853402e+243]]
```

Numpy functions: zeros()

- ➔ The zeros() function will create a new array of the specified size with the contents filled with zero values
- ➔ The argument to the function is an array or tuple that specifies the length of each dimension of the array to create.



```python
from numpy import zeros
a = zeros([3,5])
print(a)
```

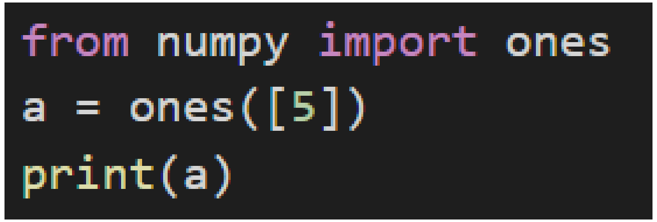


```
[[0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0.]]
```

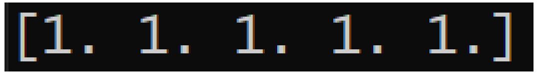
Numpy functions: ones()

➔ The ones() function will create a new array of the specified size with the contents filled with one values.

➔ The argument to the function is an array or tuple that specifies the length of each dimension of the array to create.



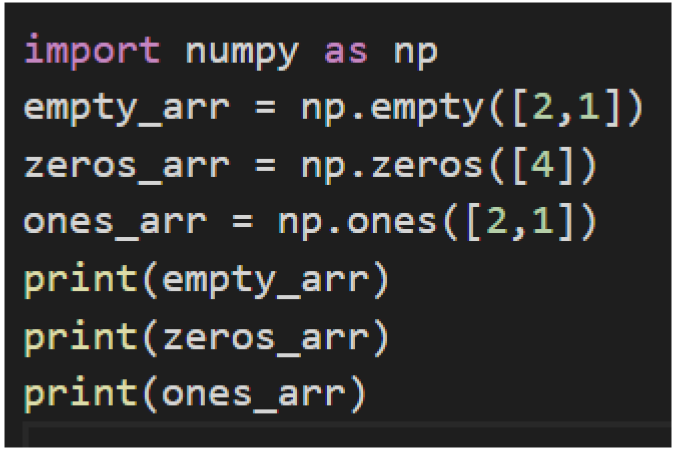
```python
from numpy import ones
a = ones([5])
print(a)
```



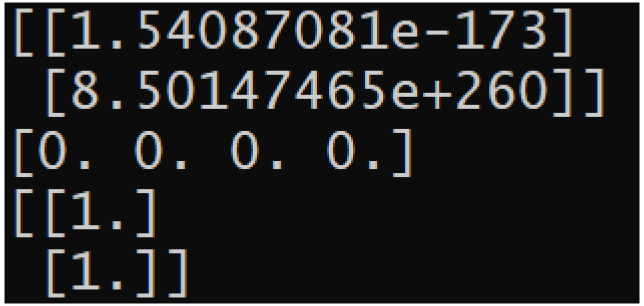
```
[1. 1. 1. 1. 1.]
```

Numpy functions:

➔ Of course, instead of importing the method only, we can import the numpy library and use alias to access its methods



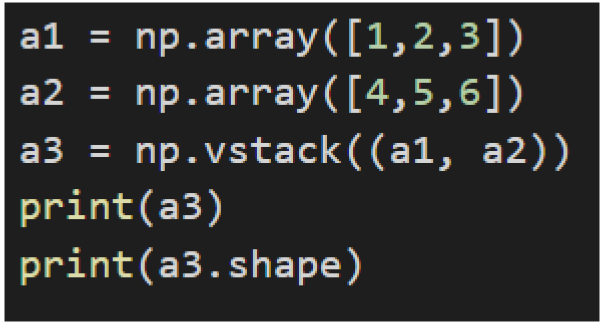
```
import numpy as np
empty_arr = np.empty([2,1])
zeros_arr = np.zeros([4])
ones_arr = np.ones([2,1])
print(empty_arr)
print(zeros_arr)
print(ones_arr)
```



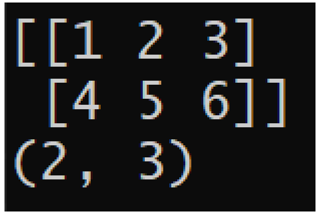
```
[[1.54087081e-173]
 [8.50147465e+260]]
[0. 0. 0. 0.]
[[1.]
 [1.]]
```

Numpy functions vstack(), hstack():

- ➔ NumPy provides many functions to create new arrays from existing arrays.
- ➔ Given two or more existing arrays, you can stack them vertically using the vstack() function. For example, given two one-dimensional arrays, you can create a new two-dimensional array with two rows by vertically stacking them.



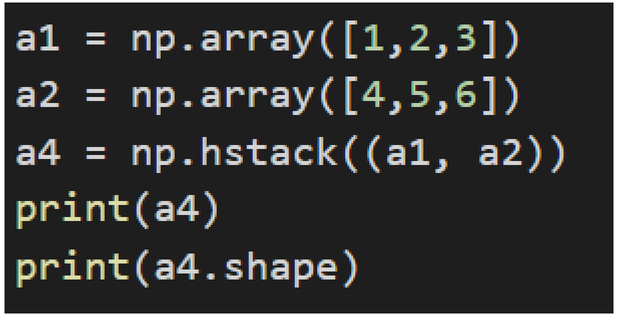
```
a1 = np.array([1,2,3])
a2 = np.array([4,5,6])
a3 = np.vstack((a1, a2))
print(a3)
print(a3.shape)
```



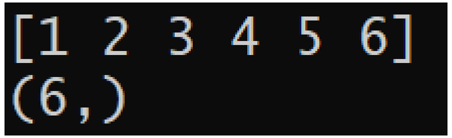
```
[[1 2 3]
 [4 5 6]]
(2, 3)
```

Numpy functions vstack(), hstack():

➔ Given two or more existing arrays, you can stack them horizontally using the hstack() function. For example, given two one-dimensional arrays, you can create a new one-dimensional array or one row with the columns of the first and second arrays concatenated.



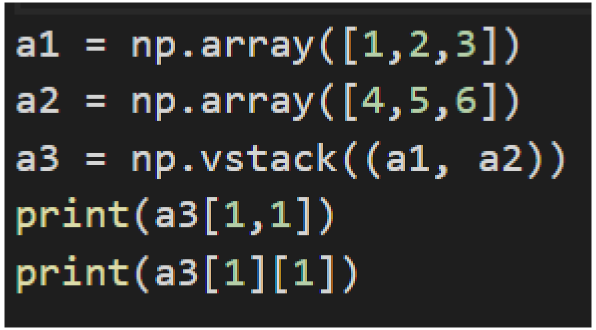
```
a1 = np.array([1,2,3])
a2 = np.array([4,5,6])
a4 = np.hstack((a1, a2))
print(a4)
print(a4.shape)
```



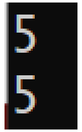
```
[1 2 3 4 5 6]
(6,)
```

Numpy array indexing & slicing:

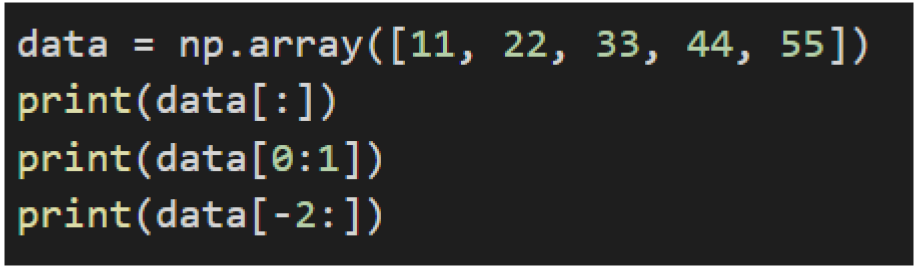
➔ You can index and slice just like ordinary lists.



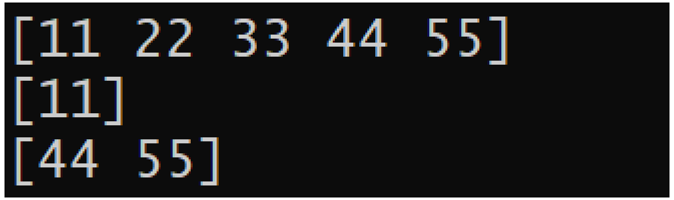
```python
a1 = np.array([1,2,3])
a2 = np.array([4,5,6])
a3 = np.vstack((a1, a2))
print(a3[1,1])
print(a3[1][1])
```



```
5
5
```



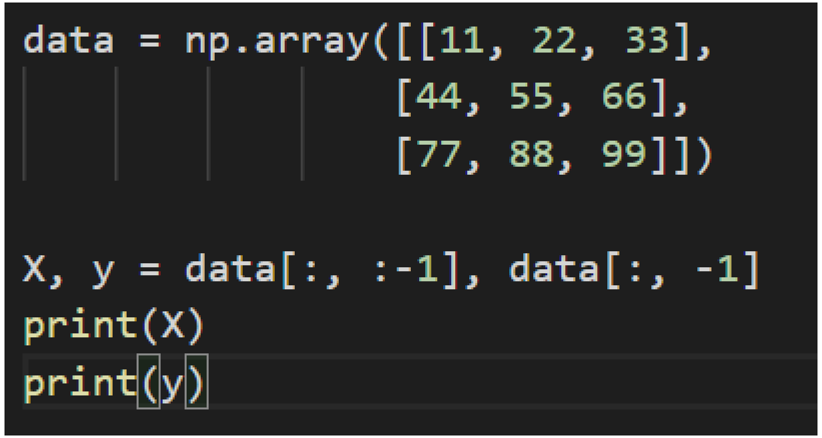
```python
data = np.array([11, 22, 33, 44, 55])
print(data[:])
print(data[0:1])
print(data[-2:])
```



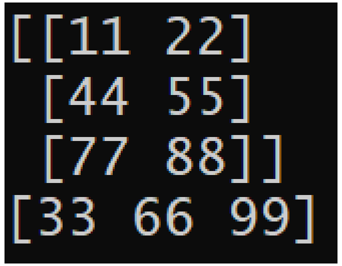
```
[11 22 33 44 55]
[11]
[44 55]
```

Numpy array Splitting data into X,y:

- ➔ It is common to split your loaded data into input variables (X) and the output variable (y).
- ➔ We can do this by slicing all rows and all columns up to, but before the last column, then separately indexing the last column.



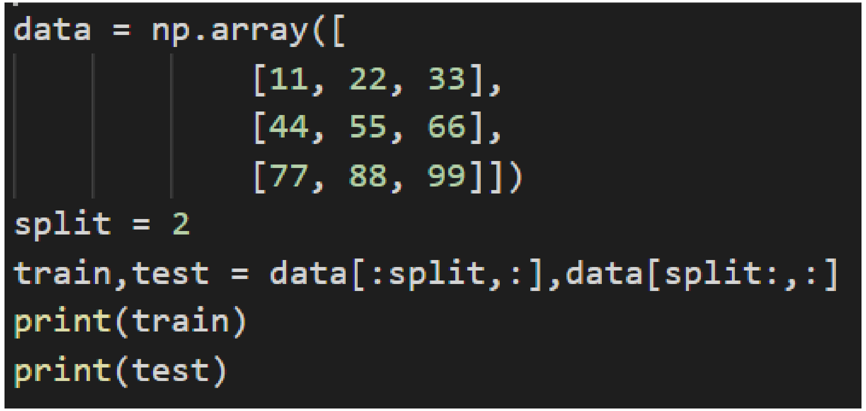
```
data = np.array([[11, 22, 33],
                 [44, 55, 66],
                 [77, 88, 99]])

X, y = data[:, :-1], data[:, -1]
print(X)
print(y)
```



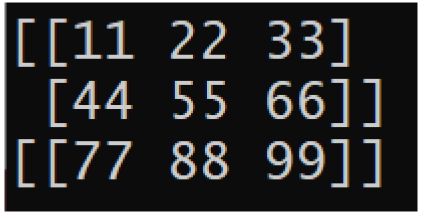
```
[[11 22]
 [44 55]
 [77 88]]
[33 66 99]
```

Numpy array Splitting data into train & test:

- ➔ It is common to split a loaded dataset into separate train and test sets.
- ➔ This is a splitting of rows where some portion will be used to train the model and the remaining portion will be used to estimate the skill of the trained model.



```
data = np.array([
            [11, 22, 33],
            [44, 55, 66],
            [77, 88, 99]])
split = 2
train,test = data[:split,:],data[split:,:]
print(train)
print(test)
```



```
[[11 22 33]
 [44 55 66]]
[[77 88 99]]
```

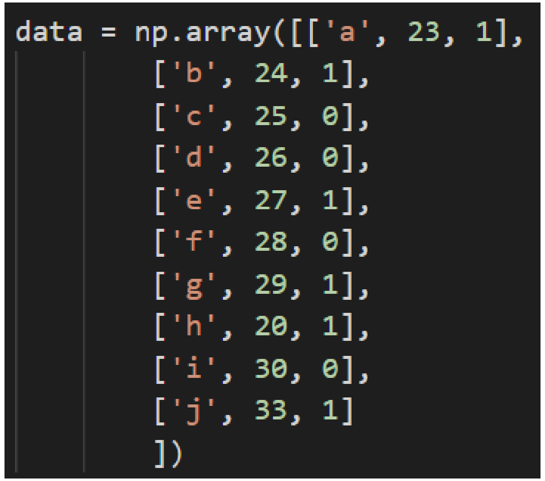
Quiz:

➔ Write a function that takes data and percent of the train set. Splits the data into train and test sets based on the passed percent. Splits both train and test into input and output data.

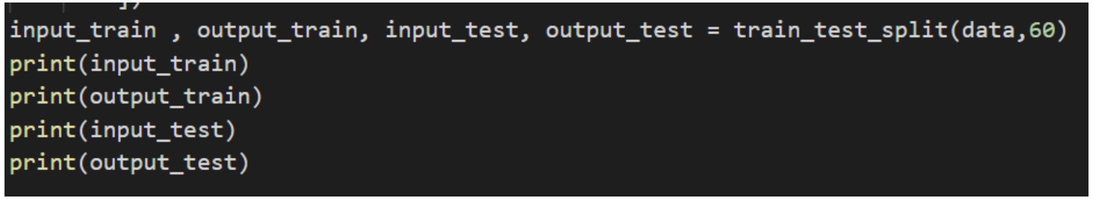
- Input: data, train_percent
- Outputs: X_train, y_train, X_test, y_test

  For example you need a data like this ->

  And call the function like this:



```
data = np.array([['a', 23, 1],
                 ['b', 24, 1],
                 ['c', 25, 0],
                 ['d', 26, 0],
                 ['e', 27, 1],
                 ['f', 28, 0],
                 ['g', 29, 1],
                 ['h', 20, 1],
                 ['i', 30, 0],
                 ['j', 33, 1]
                 ])
```



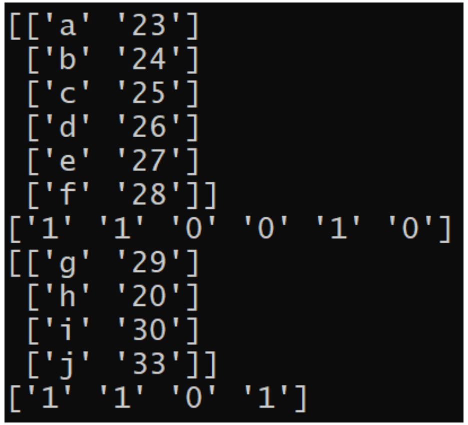
```
input_train , output_train, input_test, output_test = train_test_split(data,60)
print(input_train)
print(output_train)
print(input_test)
print(output_test)
```

Quiz:

➔ Write a function that takes data and percent of the train set. Splits the data into train and test sets based on the passed percent. Splits both train and test into input and output data.

- Input: data, train_percent
- Outputs: X_train, y_train, X_test, y_test

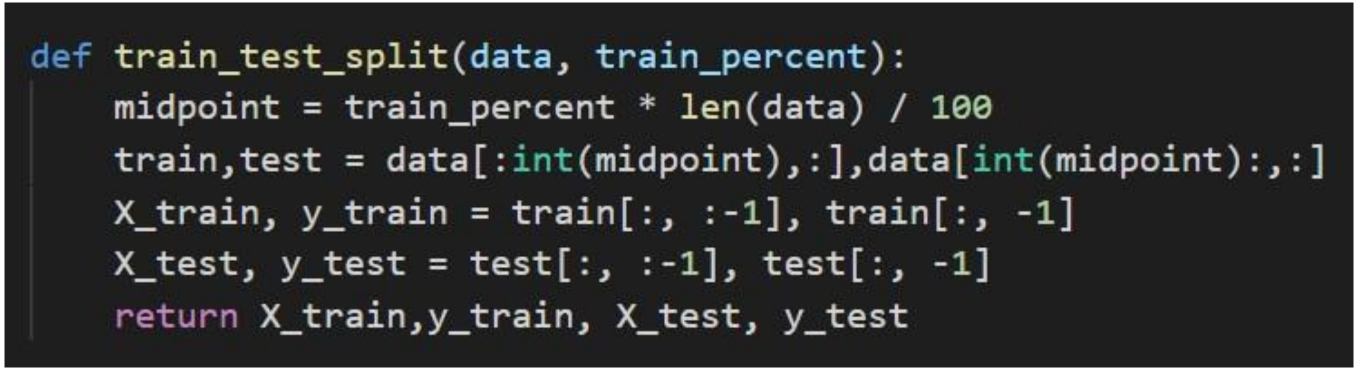
Expected output is:



```
[['a' '23']
 ['b' '24']
 ['c' '25']
 ['d' '26']
 ['e' '27']
 ['f' '28']]
['1' '1' '0' '0' '1' '0']
[['g' '29']
 ['h' '20']
 ['i' '30']
 ['j' '33']]
['1' '1' '0' '1']
```

Quiz (Solution):

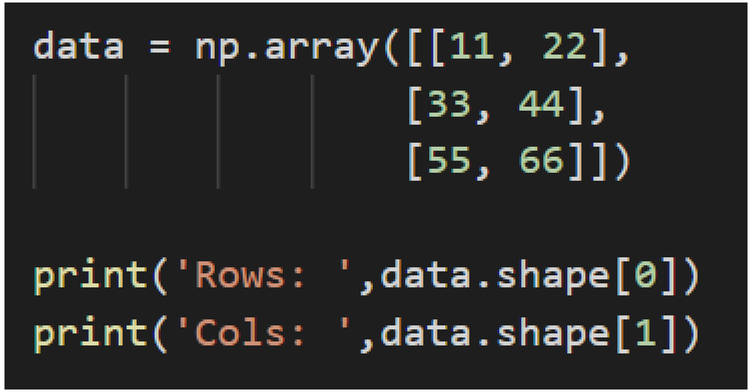
- ➔ Write a function that takes data and percent of the train set. Splits the data into train and test sets based on the passed percent. Splits both train and test into input and output data.
  - Input: data, train_percent
  - Outputs: X_train, y_train, X_test, y_test



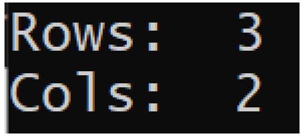
```python
def train_test_split(data, train_percent):
    midpoint = train_percent * len(data) / 100
    train,test = data[:int(midpoint),:],data[int(midpoint):,:]
    X_train, y_train = train[:, :-1], train[:, -1]
    X_test, y_test = test[:, :-1], test[:, -1]
    return X_train,y_train, X_test, y_test
```

Array Reshaping:

➔ You can use the size of your array dimensions in the shape dimension, such as specifying parameters. The elements of the tuple can be accessed just like an array, with the 0th index for the number of rows and the 1st index for the number of columns.



```
data = np.array([[11, 22],
                 [33, 44],
                 [55, 66]])

print('Rows: ',data.shape[0])
print('Cols: ',data.shape[1])
```



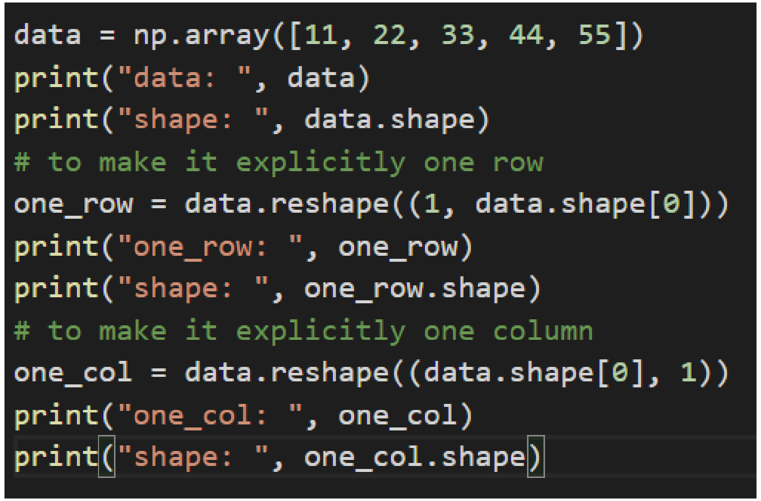
```
Rows:   3
Cols:   2
```

Array Reshaping:

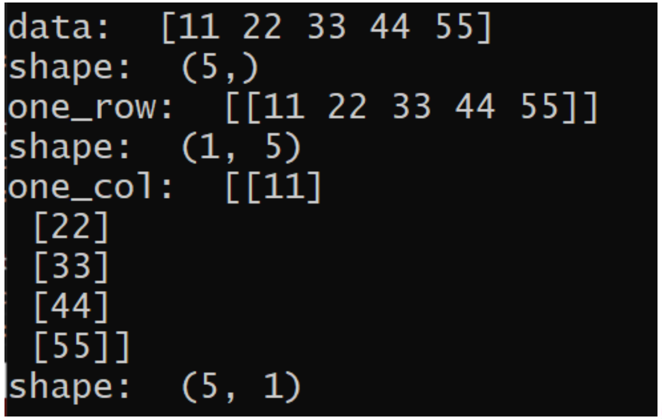
- ➔ It is common to need to reshape a one-dimensional array into a two-dimensional array with one column and multiple arrays.
- ➔ NumPy provides the reshape() function on the NumPy array object that can be used to reshape the data.
- ➔ The reshape() function takes a single argument that specifies the new shape of the array.

Array Reshaping 1D to 2D:

➔ In the case of reshaping a one-dimensional array into a two-dimensional array with one column, the tuple would be the shape of the array as the first dimension (data.shape[0]) and 1 for the second dimension, and vise versa..



```python
data = np.array([11, 22, 33, 44, 55])
print("data: ", data)
print("shape: ", data.shape)
# to make it explicitly one row
one_row = data.reshape((1, data.shape[0]))
print("one_row: ", one_row)
print("shape: ", one_row.shape)
# to make it explicitly one column
one_col = data.reshape((data.shape[0], 1))
print("one_col: ", one_col)
print("shape: ", one_col.shape)
```



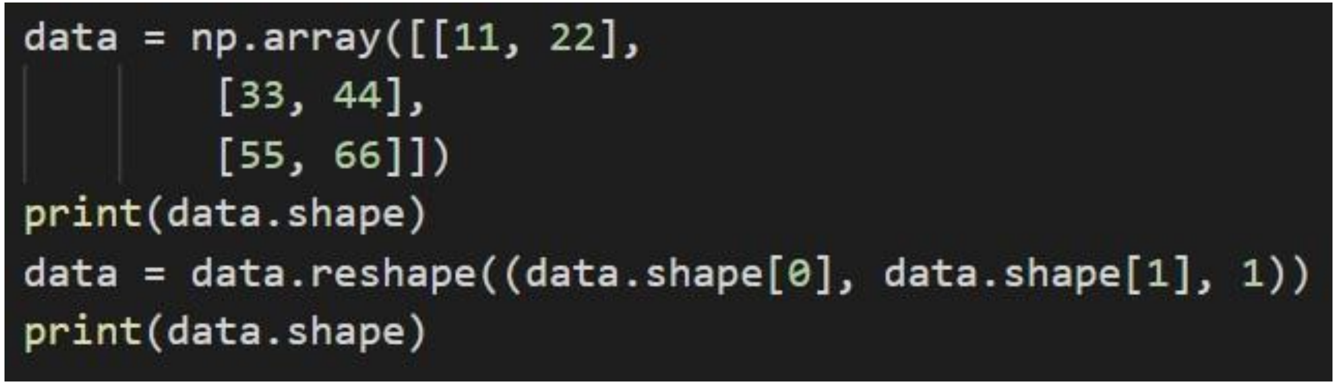
```
data:  [11 22 33 44 55]
shape:  (5,)
one_row:  [[11 22 33 44 55]]
shape:  (1, 5)
one_col:  [[11]
 [22]
 [33]
 [44]
 [55]]
shape:  (5, 1)
```

Array Reshaping 2D to 3D:

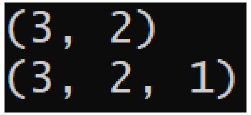
- ➔ It is common to need to reshape two-dimensional data where each row represents a sequence into a three-dimensional array for algorithms that expect multiple samples of one or more time steps and one or more features.
- ➔ A good example is the LSTM recurrent neural network model in the Keras deep learning library.
- ➔ This is clear with an example where each sequence has multiple time steps with one observation (one voice note) (feature) at each time step.

Array Reshaping 2D to 3D:

➔ The reshape function can be used directly, specifying the new dimensionality.



```
data = np.array([[11, 22],
                 [33, 44],
                 [55, 66]])
print(data.shape)
data = data.reshape((data.shape[0], data.shape[1], 1))
print(data.shape)
```

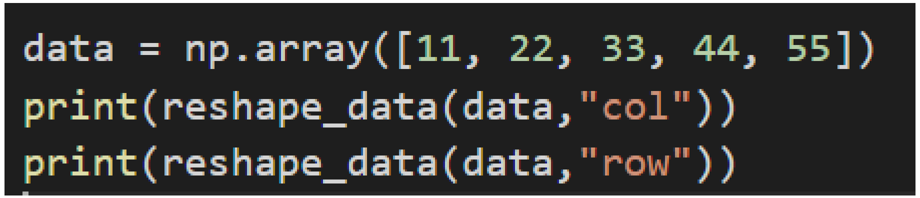


```
(3, 2)
(3, 2, 1)
```

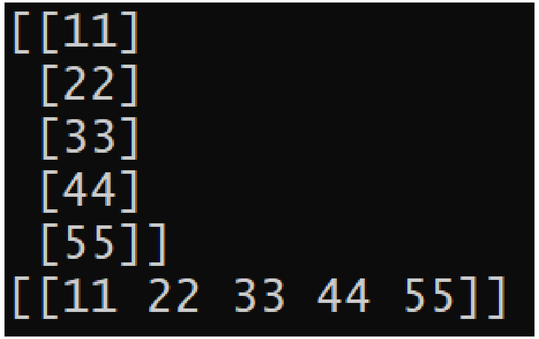
Quiz:

→ Write a function that takes 1D array and reshape it based on a flag, col or row and gives back the array reshaped. Make it default to reshape it as column array

- Input: data, flag
- Output: data reshaped



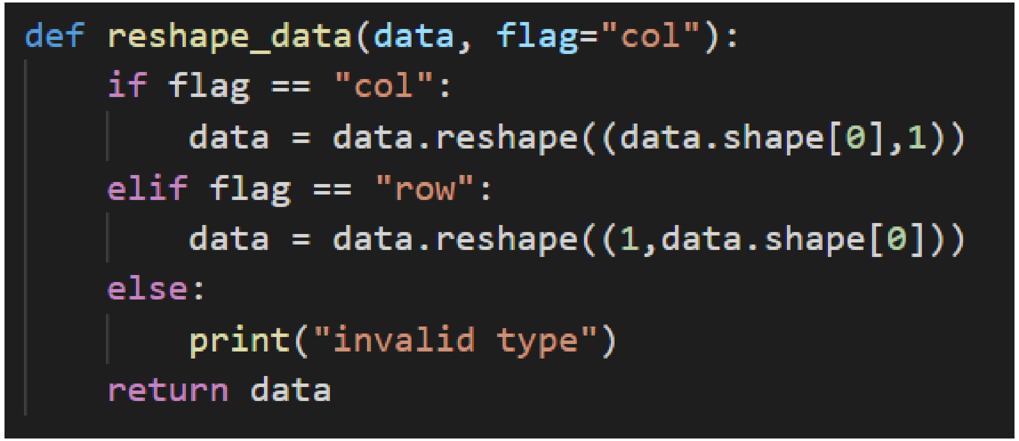
```
data = np.array([11, 22, 33, 44, 55])
print(reshape_data(data,"col"))
print(reshape_data(data,"row"))
```



```
[[11]
 [22]
 [33]
 [44]
 [55]]
[[11 22 33 44 55]]
```

Quiz (Solution):

➔ Write a function that takes 1D array and reshape it based on a flag, col or row and gives back the array reshaped. Make it default to reshape it as column array



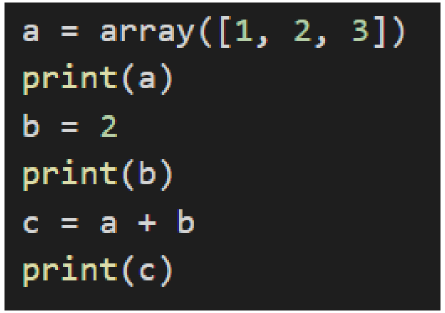
```python
def reshape_data(data, flag="col"):
    if flag == "col":
        data = data.reshape((data.shape[0],1))
    elif flag == "row":
        data = data.reshape((1,data.shape[0]))
    else:
        print("invalid type")
    return data
```

NumPy Array Broadcasting

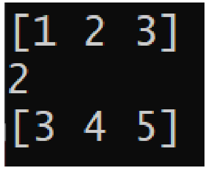
➔ Arithmetic operations may only be performed on arrays that have the same dimensions and dimensions with the same size.

➔ Broadcasting is the name given to the method that NumPy uses to allow array arithmetic between arrays with a different shape or size.

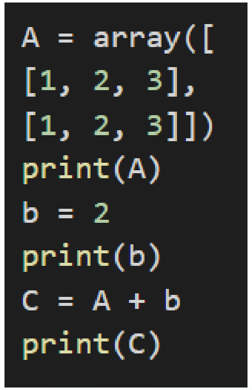
➔ Although the technique was developed for NumPy, it has also been adopted more broadly in other numerical computational libraries, such as Theano, TensorFlow, and Octave.



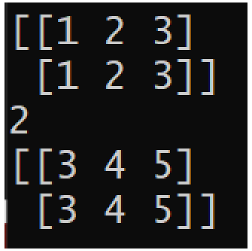
```
a = array([1, 2, 3])
print(a)
b = 2
print(b)
c = a + b
print(c)
```



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



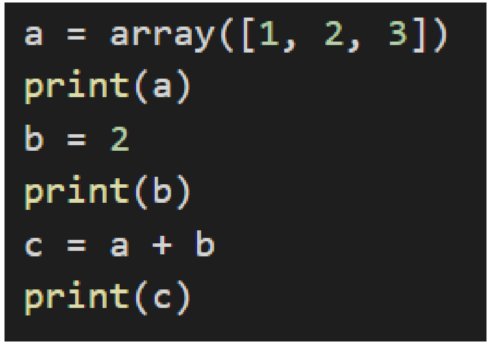
```
A = array([
[1, 2, 3],
[1, 2, 3]])
print(A)
b = 2
print(b)
C = A + b
print(C)
```



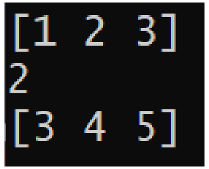
```
[[1 2 3]
 [1 2 3]]
2
[[3 4 5]
 [3 4 5]]
```

Broadcasting a scalar with 1D & 2D:

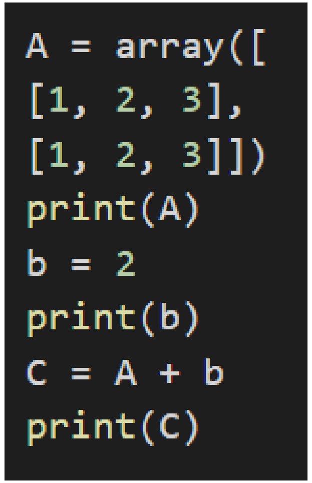
➔ Python does it automatically.



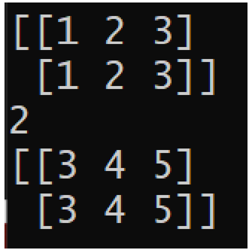
```
a = array([1, 2, 3])
print(a)
b = 2
print(b)
c = a + b
print(c)
```



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



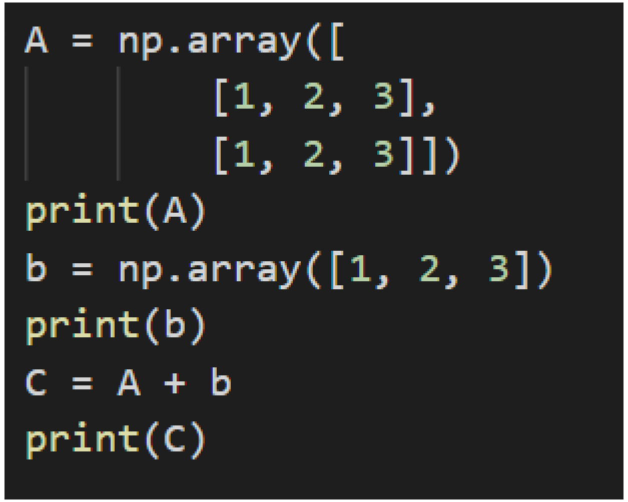
```
A = array([
[1, 2, 3],
[1, 2, 3]])
print(A)
b = 2
print(b)
C = A + b
print(C)
```



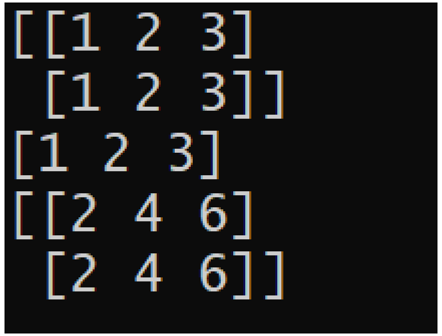
```
[[1 2 3]
 [1 2 3]]
2
[[3 4 5]
 [3 4 5]]
```

Broadcasting a 1D with 2D:

➔ Python does it automatically.



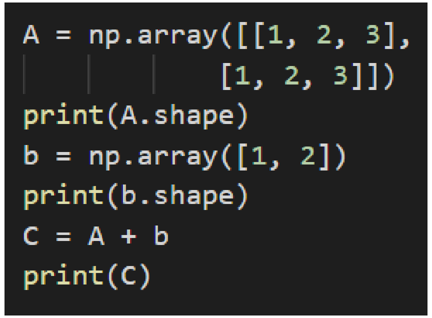
```
A = np.array([
            [1, 2, 3],
            [1, 2, 3]])
print(A)
b = np.array([1, 2, 3])
print(b)
C = A + b
print(C)
```



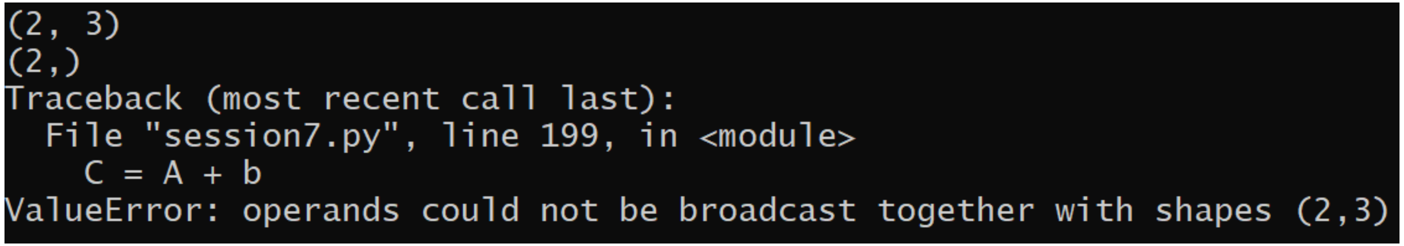
```
[[1 2 3]
 [1 2 3]]
[1 2 3]
[[2 4 6]
 [2 4 6]]
```

Broadcasting Limitation:

- ➔ Limitation of broadcasting is that column numbers should be equal.
- ➔ Here we are attempting to broadcast a 1D array (1x2) and a 2D array (2x3)



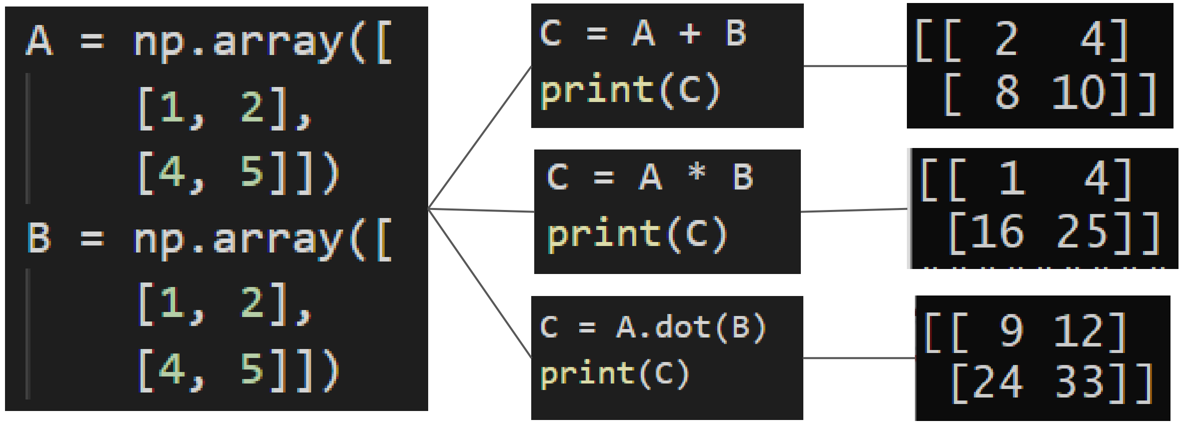
```python
A = np.array([[1, 2, 3],
              [1, 2, 3]])
print(A.shape)
b = np.array([1, 2])
print(b.shape)
C = A + b
print(C)
```



```
(2, 3)
(2,)
Traceback (most recent call last):
  File "session7.py", line 199, in <module>
    C = A + b
ValueError: operands could not be broadcast together with shapes (2,3)
```

Revisiting matrix arithmetics in numpy.

➔It’s much simpler with numpy.



```
A = np.array([
    [1, 2],
    [4, 5]])
B = np.array([
    [1, 2],
    [4, 5]])
```

```
C = A + B
print(C)
```

```
[[ 2  4]
 [ 8 10]]
```

```
C = A * B
print(C)
```

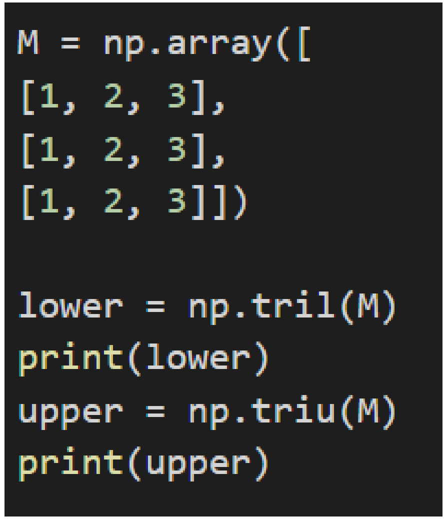
```
[[ 1  4]
 [16 25]]
```

```
C = A.dot(B)
print(C)
```

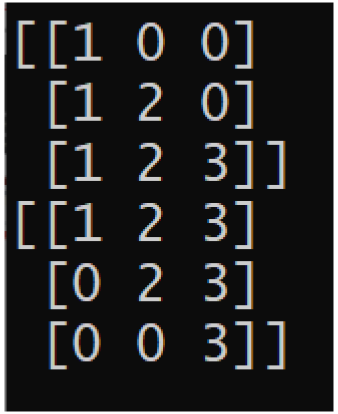
```
[[ 9 12]
 [24 33]]
```

Types of matrices in numpy (Triangular)

➔ We have some built-in methods to construct the popular types of matrices.



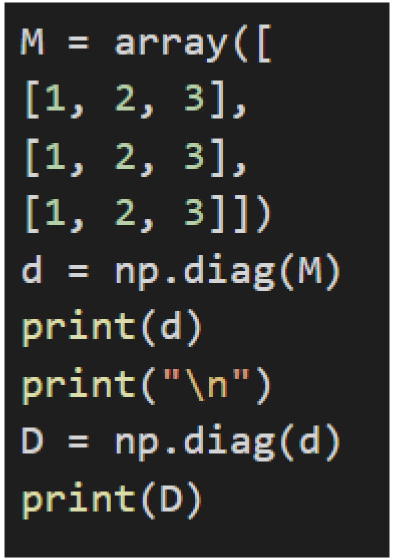
```
M = np.array([
[1, 2, 3],
[1, 2, 3],
[1, 2, 3]])

lower = np.tril(M)
print(lower)
upper = np.triu(M)
print(upper)
```



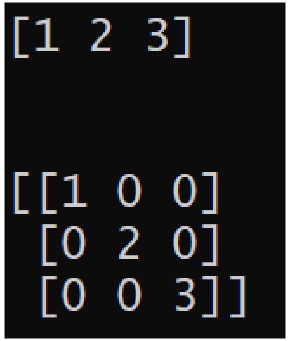
```
[[1 0 0]
 [1 2 0]
 [1 2 3]]
[[1 2 3]
 [0 2 3]
 [0 0 3]]
```

Types of matrices in numpy (Diagonal)

➔ We have some built-in methods to construct the popular types of matrices.



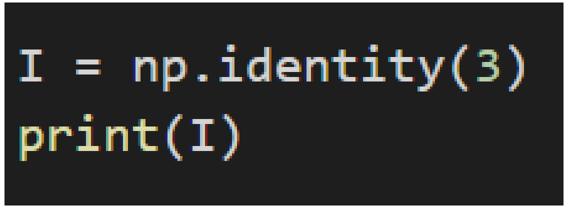
```
M = array([
[1, 2, 3],
[1, 2, 3],
[1, 2, 3]])
d = np.diag(M)
print(d)
print("\n")
D = np.diag(d)
print(D)
```



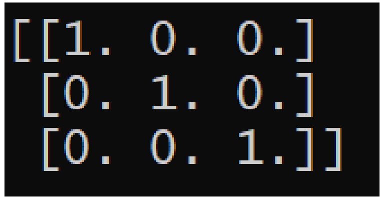
```
[1 2 3]


[[1 0 0]
 [0 2 0]
 [0 0 3]]
```

Types of matrices in numpy (Identity)

➔ We have some built-in methods to construct the popular types of matrices.



```
I = np.identity(3)
print(I)
```



```
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
```

2. Matrix Operations

Matrix Operations:

- ➔ Transpose
- ➔ Inverse
- ➔ Trace
- ➔ Determinant
- ➔ Rank

Matrix Operations (Transpose):

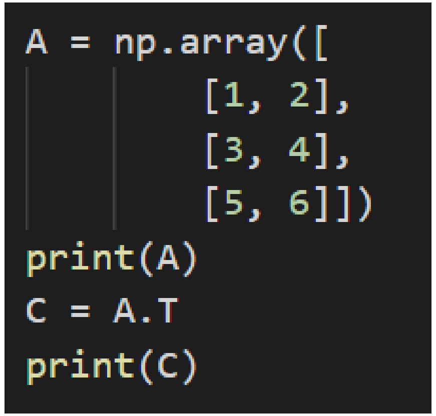
- ➔ A defined matrix can be transposed, which creates a new matrix with the number of columns and rows flipped. This is denoted by the superscript T next to the matrix AT.
- ➔ The columns of AT are the rows of A.

$$C = A^T$$

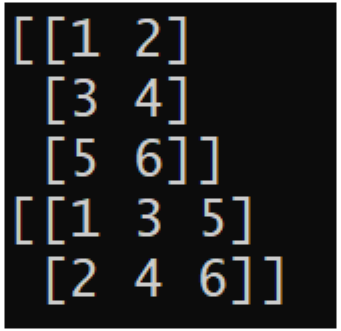
$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix} \qquad A^T = \begin{pmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{pmatrix}$$

Matrix Operations (Transpose):

➔ We can transpose a matrix in NumPy by calling the T attribute.



```
A = np.array([
            [1, 2],
            [3, 4],
            [5, 6]])
print(A)
C = A.T
print(C)
```



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

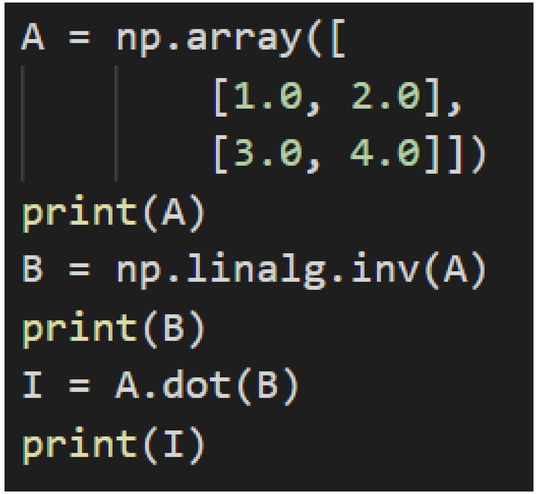
Matrix Operations (Inverse):

- ➔ Matrix inversion is a process that finds another matrix that when multiplied with the matrix, results in an identity matrix.
- ➔ Given a matrix A, find matrix B, such that: $AB = BA = I^n$
- ➔ The operation of inverting a matrix is indicated by a −1 superscript next to the matrix; for example, A−1

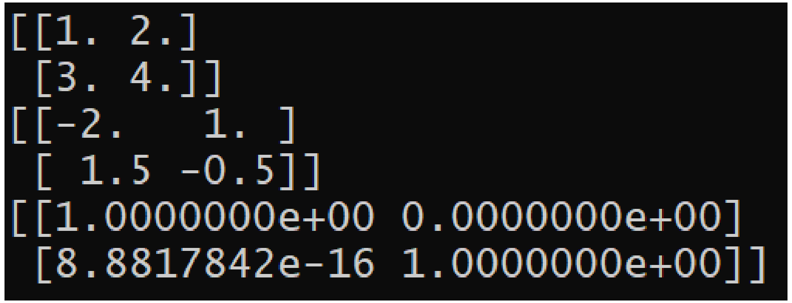
Whatever A does, A^{-1} undoes.

Matrix Operations (Inverse):

→ First, we define a small 2 × 2 matrix, then calculate the inverse of the matrix, and then confirm the inverse by multiplying it with the original matrix to give the identity matrix.



```
A = np.array([
        [1.0, 2.0],
        [3.0, 4.0]])
print(A)
B = np.linalg.inv(A)
print(B)
I = A.dot(B)
print(I)
```



```
[[1. 2.]
 [3. 4.]]
[[-2.   1. ]
 [ 1.5 -0.5]]
[[1.0000000e+00 0.0000000e+00]
 [8.8817842e-16 1.0000000e+00]]
```

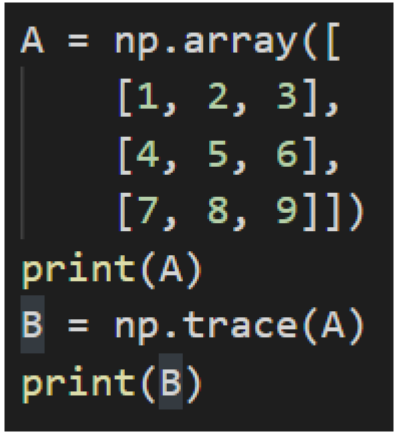
Matrix Operations (Trace):

- ➔ A trace of a square matrix is the sum of the values on the main diagonal of the matrix (top-left to bottom-right).
- ➔ The trace operator gives the sum of all of the diagonal entries of a matrix.
- ➔ The operation of calculating a trace on a square matrix is described using the notation tr(A) where A is the square matrix on which the operation is being performed.
- ➔ The trace is calculated as the sum of the diagonal values; for example, in the case of a 3 × 3 matrix:

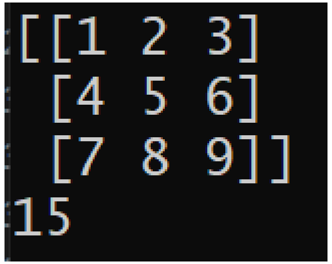
$$tr(A) = a_{1,1} + a_{2,2} + a_{3,3}$$

Matrix Operations (Trace):

➔ First, a 3 × 3 matrix is created and then the trace is calculated



```
A = np.array([
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]])
print(A)
B = np.trace(A)
print(B)
```



```
[[1 2 3]
 [4 5 6]
 [7 8 9]]
15
```

Matrix Operations (Determinant):

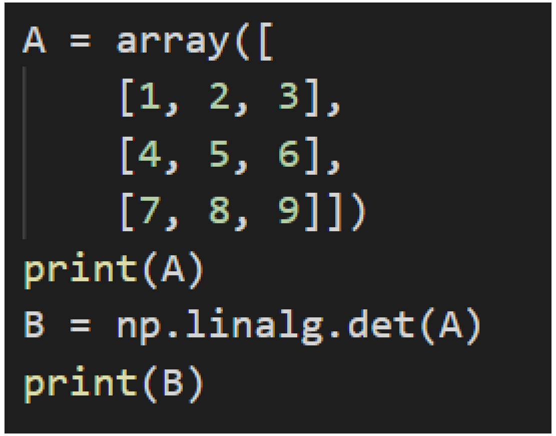
- ➔ The determinant of a square matrix is a scalar representation of the volume of the matrix.
- ➔ It is denoted by the det(A) notation or |A|, where A is the matrix on which we are calculating the determinant.

$$det(A)$$

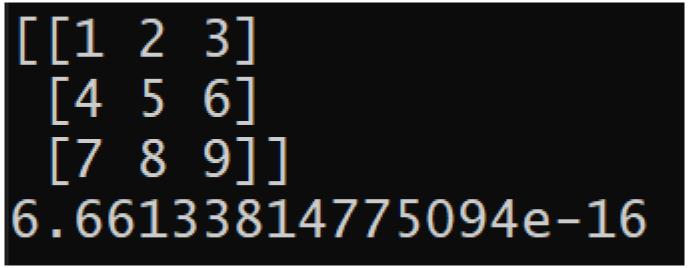
- ➔ The determinant of a square matrix is a single number.

Matrix Operations (Determinant):

➔ First, a 3 × 3 matrix is defined, then the determinant of the matrix is calculated.



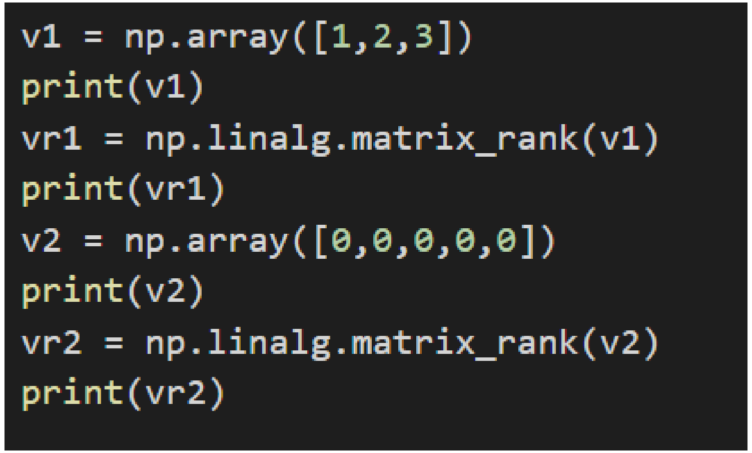
```
A = array([
      [1, 2, 3],
      [4, 5, 6],
      [7, 8, 9]])
print(A)
B = np.linalg.det(A)
print(B)
```



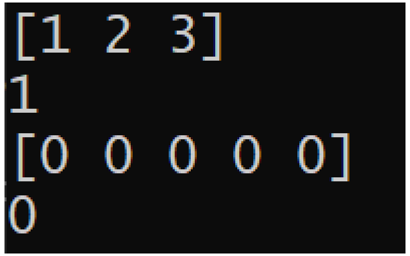
```
[[1 2 3]
 [4 5 6]
 [7 8 9]]
6.66133814775094e-16
```

Matrix Operations (Rank):

➔ The rank of a matrix is the estimate of the number of linearly independent rows or columns in a matrix. The rank of a matrix M is often denoted as the function rank().



```
v1 = np.array([1,2,3])
print(v1)
vr1 = np.linalg.matrix_rank(v1)
print(vr1)
v2 = np.array([0,0,0,0,0])
print(v2)
vr2 = np.linalg.matrix_rank(v2)
print(vr2)
```



```
[1 2 3]
1
[0 0 0 0 0]
0
```

3. Sparse Matrices

Sparse Matrices:

- ➔ Matrices that contain mostly zero values are called sparse.
- ➔ Matrices where most of the values are non-zero, called dense.
- ➔ Large sparse matrices are common in general and especially in applied machine learning, such as in data that contains counts, data encodings that map categories to counts, and even in whole subfields of machine learning such as natural language processing
- ➔ In this part:
    - Sparse Matrix
    - Problems with Sparsity
    - Sparse Matrices in Machine Learning
    - Working with Sparse Matrices
    - Sparse Matrices in Python

Sparse Matrices:

➔ The sparsity of a matrix can be quantified with a score, which is the number of zero values in the matrix divided by the total number of elements in the matrix.

$$\text{sparsity} = \frac{\text{count of non-zero elements}}{\text{total elements}}$$

➔ For the below matrix, sparsity score is 13/18 = 0.722 or about 72%.

$$A = \begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 1 \\ 0 & 0 & 0 & 2 & 0 & 0 \end{pmatrix}$$

Sparse Matrices Problems:

- ➔ Space Complexity: In practice, most large matrices are sparse —almost all entries are zeros.
- ➔ A word or term occurrence matrix for words in one book against all known words in English, the matrix contained is sparse with many more zero values than data values.
- ➔ The problem with representing these sparse matrices as dense matrices is that memory is required and must be allocated for each 32-bit or even 64-bit zero value in the matrix.
- ➔ This is clearly a waste of memory resources as those zero values do not contain any information.

Sparse Matrices Problems:

- ➔ Time Complexity: , if the matrix contains mostly zero-values, i.e. no data, then performing operations across this matrix may take a long time where the bulk of the computation performed will involve adding or multiplying zero values together.
- ➔ This is a problem of increased time complexity of matrix operations that increases with the size of the matrix.

Sparse Matrices in ML:

- ➔ In raw Data, in observations that record the occurrence or count of an activity.
  - ○ Whether or not a user has watched a movie in a movie catalog.
  - ○ Whether or not a user has purchased a product in a product catalog.
  - ○ Count of the number of listens of a song in a song catalog
- ➔ Data Preparation, in encoding schemes used in the preparation of data.
  - ○ One hot encoding, used to represent categorical data as sparse binary vectors.
  - ○ Count encoding, used to represent the frequency of words in a vocabulary for a document.
- ➔ NLP, Recommender systems and Computer vision.

Sparse Matrices Solution:

➔ We need to transform it into another data structure. Where the zero values can be ignored and only the data or non-zero values are stored:

- Compressed Sparse Row. The sparse matrix is represented using three one-dimensional arrays for the non-zero values, the extents of the rows, and the column indexes.
- Compressed Sparse Column. The same as the Compressed Sparse Row method except the column indices are compressed and read first before the row indices.

Scipy:

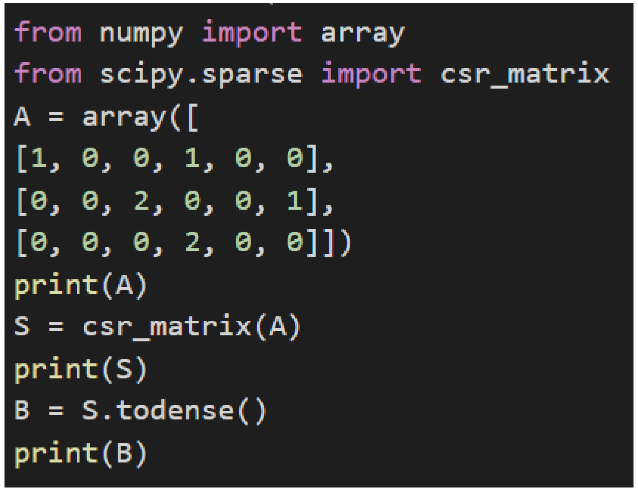
➔ SciPy provides tools for creating sparse matrices using multiple data structures.

➔ And tools for converting a dense matrix to a sparse matrix.

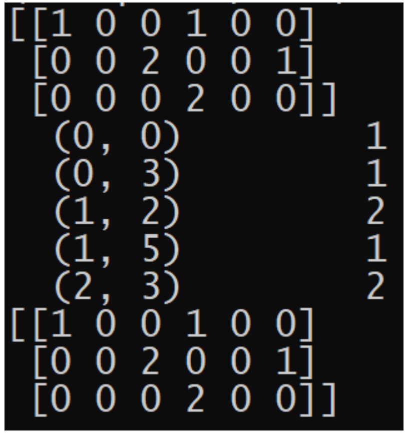
➔ SciPy sparse arrays are special data structure in scipy.

SciPy Sparse Matrix:

➔ A dense matrix stored in a NumPy array can be converted into a sparse matrix using the CSR representation by calling the csr matrix() function.



```
from numpy import array
from scipy.sparse import csr_matrix
A = array([
[1, 0, 0, 1, 0, 0],
[0, 0, 2, 0, 0, 1],
[0, 0, 0, 2, 0, 0]])
print(A)
S = csr_matrix(A)
print(S)
B = S.todense()
print(B)
```



```
[[1 0 0 1 0 0]
 [0 0 2 0 0 1]
 [0 0 0 2 0 0]]
  (0, 0)        1
  (0, 3)        1
  (1, 2)        2
  (1, 5)        1
  (2, 3)        2
[[1 0 0 1 0 0]
 [0 0 2 0 0 1]
 [0 0 0 2 0 0]]
```

Sparsity of Matrix:

- ➔ NumPy does not provide a function to calculate the sparsity of a matrix.
- ➔ we can calculate it easily by first finding the density of the matrix and subtracting it from one.
- ➔ The number of non-zero elements in a NumPy array can be given by the count_nonzero() function and the total number of elements in the array can be given by the size property of the array.

```
sparsity = 1.0 - count_nonzero(A) / A.size
```

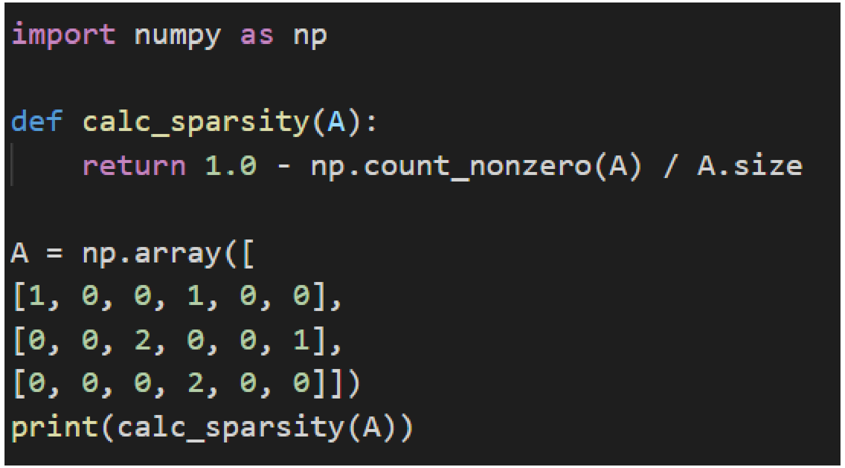
Quiz:

- ➔ Write a function to take a matrix (Sparse Matrix) and calculates its sparsity.
    - Inputs: A matrix as np array
    - Output: sparsity value.
    - Using this equation:

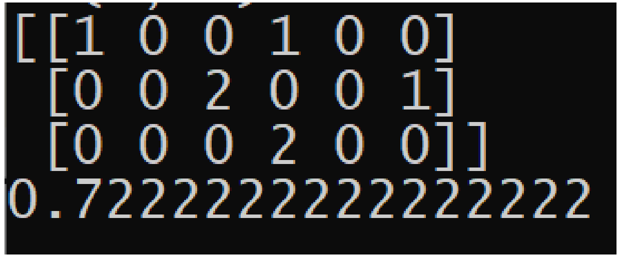
```
sparsity = 1.0 - count_nonzero(A) / A.size
```

Quiz (Solution):

➔ Write a function to take a matrix (Sparse Matrix) and calculates its sparsity.



```python
import numpy as np

def calc_sparsity(A):
    return 1.0 - np.count_nonzero(A) / A.size

A = np.array([
[1, 0, 0, 1, 0, 0],
[0, 0, 2, 0, 0, 1],
[0, 0, 0, 2, 0, 0]])
print(calc_sparsity(A))
```

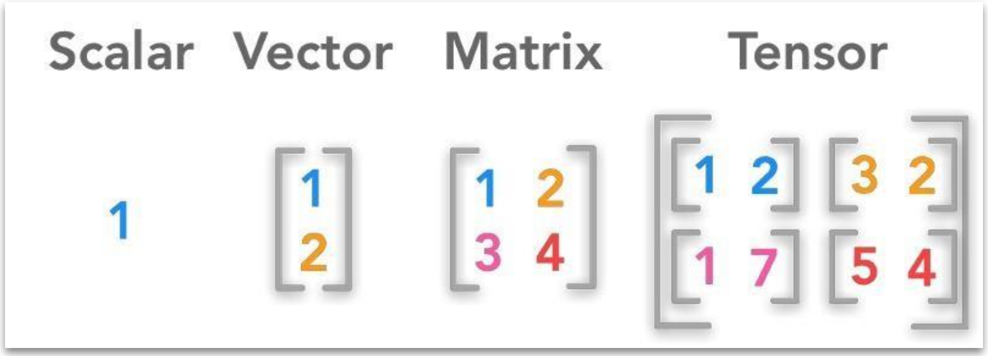


```
[[1 0 0 1 0 0]
 [0 0 2 0 0 1]
 [0 0 0 2 0 0]]
0.7222222222222222
```

4. Tensors & tensors arithmetic

Tensors:

- ➔ In deep learning it is common to see a lot of discussion around tensors as the cornerstone data structure.
- ➔ Tensor even appears in name of Google's flagship machine learning library: TensorFlow.
- ➔ Tensors are a type of data structure used in linear algebra, and like vectors and matrices, you can calculate arithmetic operations with tensors.



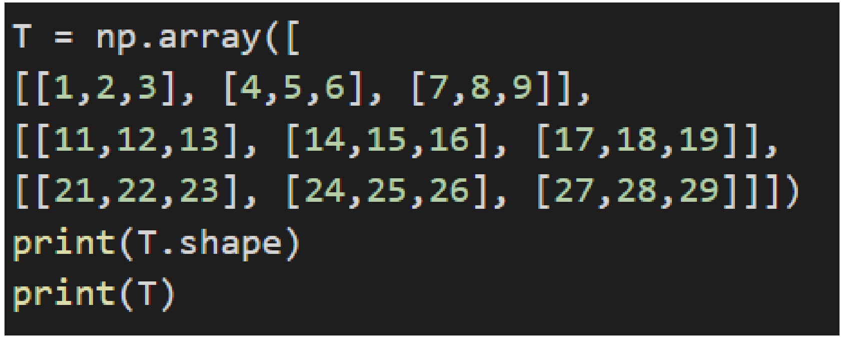
Tensors:

- ➔ A tensor is a generalization of vectors and matrices and is easily understood as a multidimensional array.
- ➔ A vector is a one-dimensional or first order tensor and a matrix is a two-dimensional or second order tensor.
- ➔ Example below defines a 3 × 3 × 3 three-dimensional tensor.

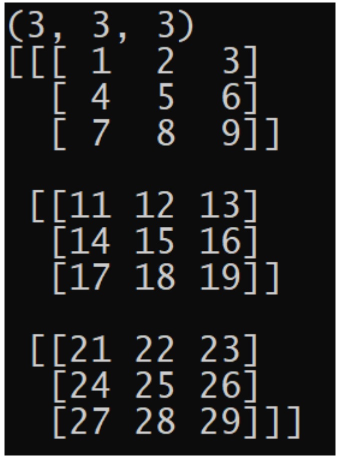
$$T = \begin{pmatrix} t_{1,1,1} & t_{1,2,1} & t_{1,3,1} \\ t_{2,1,1} & t_{2,2,1} & t_{2,3,1} \\ t_{3,1,1} & t_{3,2,1} & t_{3,3,1} \end{pmatrix}, \begin{pmatrix} t_{1,1,2} & t_{1,2,2} & t_{1,3,2} \\ t_{2,1,2} & t_{2,2,2} & t_{2,3,2} \\ t_{3,1,2} & t_{3,2,2} & t_{3,3,2} \end{pmatrix}, \begin{pmatrix} t_{1,1,3} & t_{1,2,3} & t_{1,3,3} \\ t_{2,1,3} & t_{2,2,3} & t_{2,3,3} \\ t_{3,1,3} & t_{3,2,3} & t_{3,3,3} \end{pmatrix}$$

Tensors:

- ➔ A tensor can be defined in-line to the constructor of array() as a list of lists.
- ➔ For this 3D tensor, axis 0 specifies the level (like height), axis 1 specifies the column, and axis 2 specifies the row.



```
T = np.array([
[[1,2,3], [4,5,6], [7,8,9]],
[[11,12,13], [14,15,16], [17,18,19]],
[[21,22,23], [24,25,26], [27,28,29]]])
print(T.shape)
print(T)
```



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

 [[11 12 13]
  [14 15 16]
  [17 18 19]]

 [[21 22 23]
  [24 25 26]
  [27 28 29]]]
```

Tensor Arithmetic Addition/Multiplication:

➔ Just like multidimensional arrays, you can add and subtract tensors.

```
A = array([
[[1,2,3], [4,5,6], [7,8,9]],
[[11,12,13], [14,15,16], [17,18,19]],
[[21,22,23], [24,25,26], [27,28,29]]])
```

```
B = array([
[[1,2,3], [4,5,6], [7,8,9]],
[[11,12,13], [14,15,16], [17,18,19]],
[[21,22,23], [24,25,26], [27,28,29]]])
```

```
C = A + B
```

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

 [[22 24 26]
  [28 30 32]
  [34 36 38]]

 [[42 44 46]
  [48 50 52]
  [54 56 58]]]
```

```
C = A * B
```

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

 [[121 144 169]
  [196 225 256]
  [289 324 361]]

 [[441 484 529]
  [576 625 676]
  [729 784 841]]]
```

Tensor Arithmetic Tensor product:

- ➔ Given a tensor A with q dimensions and tensor B with r dimensions, the product of these tensors will be a new tensor with the order of q + r
- ➔ The tensor product is not limited to tensors, but can also be performed on matrices and vectors.
- ➔ Vector tensor product:

$$a = \begin{pmatrix} a_1 \\ a_2 \end{pmatrix}$$

$$b = \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}$$

$$C = a \otimes b$$

$$C = \begin{pmatrix} a_1 \times \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} \\ a_2 \times \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} \end{pmatrix}$$

$$C = \begin{pmatrix} a_1 \times b_1 & a_1 \times b_2 \\ a_2 \times b_1 & a_2 \times b_2 \end{pmatrix}$$

Tensor Arithmetic Tensor product:

➔ Matrix tensor product:

$$A = \begin{pmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{pmatrix}$$

$$B = \begin{pmatrix} b_{1,1} & b_{1,2} \\ b_{2,1} & b_{2,2} \end{pmatrix}$$

$$C = A \otimes B$$

$$C = \begin{pmatrix} a_{1,1} \times \begin{pmatrix} b_{1,1}, b_{1,2} \\ b_{2,1}, b_{2,2} \end{pmatrix} & a_{1,2} \times \begin{pmatrix} b_{1,1}, b_{1,2} \\ b_{2,1}, b_{2,2} \end{pmatrix} \\ a_{2,1} \times \begin{pmatrix} b_{1,1}, b_{1,2} \\ b_{2,1}, b_{2,2} \end{pmatrix} & a_{2,2} \times \begin{pmatrix} b_{1,1}, b_{1,2} \\ b_{2,1}, b_{2,2} \end{pmatrix} \end{pmatrix}$$

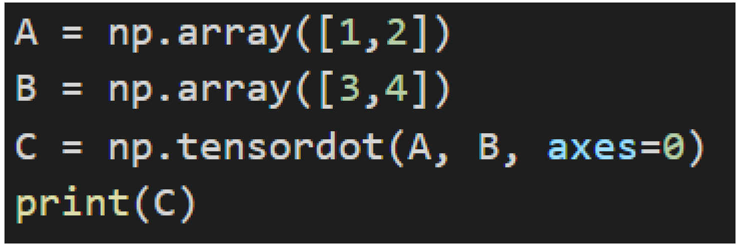
$$C = \begin{pmatrix} a_{1,1} \times b_{1,1} & a_{1,1} \times b_{1,2} & a_{1,2} \times b_{1,1} & a_{1,2} \times b_{1,2} \\ a_{1,1} \times b_{2,1} & a_{1,1} \times b_{2,2} & a_{1,2} \times b_{2,1} & a_{1,2} \times b_{2,2} \\ a_{2,1} \times b_{1,1} & a_{2,1} \times b_{1,2} & a_{2,2} \times b_{1,1} & a_{2,2} \times b_{1,2} \\ a_{2,1} \times b_{2,1} & a_{2,1} \times b_{2,2} & a_{2,2} \times b_{2,1} & a_{2,2} \times b_{2,2} \end{pmatrix}$$

Tensor product in numpy:

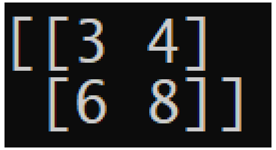
➔ The tensor product can be implemented in NumPy using the tensordot() function.

➔ The function takes as arguments the two tensors to be multiplied and the axis on which to sum the products over, called the sum reduction.

➔ In the example below, we define two order-1 tensors (vectors) with and calculate the tensor product.



```
A = np.array([1,2])
B = np.array([3,4])
C = np.tensordot(A, B, axes=0)
print(C)
```



```
[[3 4]
 [6 8]]
```

Task:

➔ Search the documentation for the definition of the third argument of the tensordot() method and investigate the difference between tensor products and tensor dot product with examples.

5. Matrix Factorization

Matrix Factorization:

- ➔ A matrix decomposition is a way of reducing a matrix into its constituent parts.
- ➔ It is an approach that can simplify more complex matrix operations that can be performed on the decomposed matrix rather than on the original matrix itself.
- ➔ A common analogy for matrix decomposition is the factoring of numbers, such as the factoring of 10 into 2 × 5
- ➔ There are a range of different matrix decomposition techniques.
- ➔ Two simple and widely used matrix decomposition methods are the LU matrix decomposition and the QR matrix decomposition

Matrix Factorization LU Decomposition:

➔ The LU decomposition is for square matrices and decomposes a matrix into L and U components.

$$A = L \cdot U$$

➔ Where A is the square matrix that we wish to decompose, L is the lower triangle matrix and U is the upper triangle matrix.

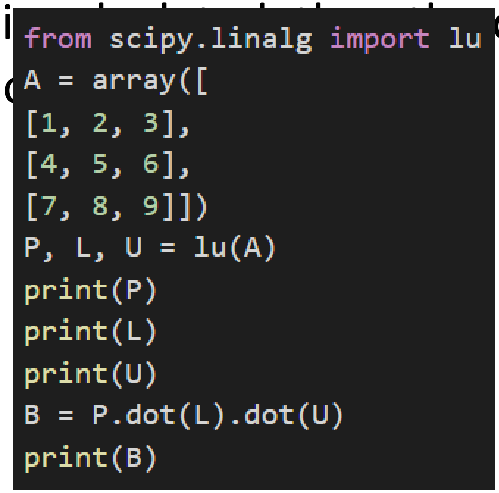
➔ A variation of this decomposition that is numerically more stable to solve in practice is called the LUP decomposition, or the LU decomposition with partial pivoting.

$$A = L \cdot U \cdot P$$

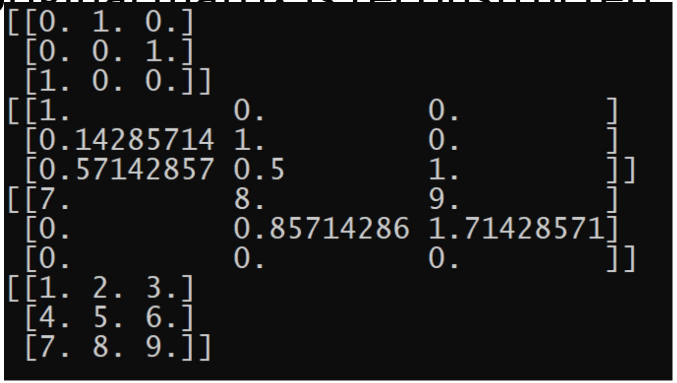
Matrix Factorization LU Decomposition:

➔ The LU decomposition can be implemented in Python with the lu() function. More specifically, this function calculates an LPU decomposition.

➔ The example below first defines a 3×3 square matrix. The LU decomposition i original matrix is reconstructed from the compo



```
from scipy.linalg import lu
A = array([
[1, 2, 3],
[4, 5, 6],
[7, 8, 9]])
P, L, U = lu(A)
print(P)
print(L)
print(U)
B = P.dot(L).dot(U)
print(B)
```



```
[[0. 1. 0.]
 [0. 0. 1.]
 [1. 0. 0.]]
[[1.         0.         0.        ]
 [0.14285714 1.         0.        ]
 [0.57142857 0.5        1.        ]]
[[7.         8.         9.        ]
 [0.         0.85714286 1.71428571]
 [0.         0.         0.        ]]
[[1. 2. 3.]
 [4. 5. 6.]
 [7. 8. 9.]]
```

Matrix Factorization QR Decomposition:

➔ The QR decomposition is for n × m matrices (not limited to square matrices) and decomposes a matrix into Q and R components.

$$A = Q \cdot R$$

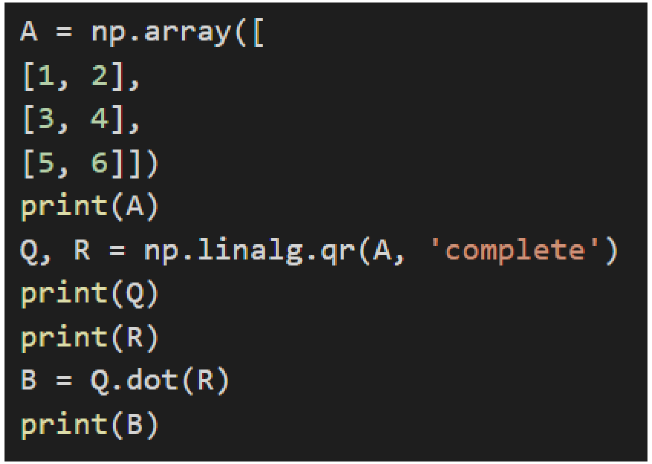
➔ Where A is the matrix that we wish to decompose, Q a matrix with the size m × m, and R is an upper triangular matrix with the size m × n.

➔ the QR decomposition is often used to solve systems of linear equations, although is not limited to square matrices

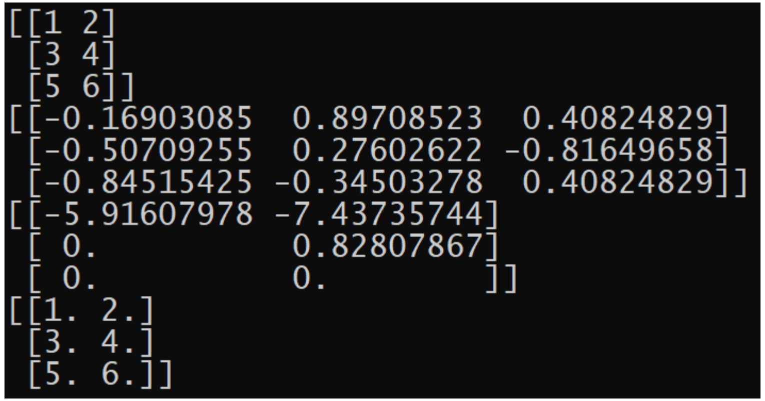
Matrix Factorization QR Decomposition:

➔ The QR decomposition can be implemented in NumPy using the qr() function.

➔ the function returns the Q and R matrices with smaller or reduced dimensions that is more economical.



```
A = np.array([
[1, 2],
[3, 4],
[5, 6]])
print(A)
Q, R = np.linalg.qr(A, 'complete')
print(Q)
print(R)
B = Q.dot(R)
print(B)
```



```
[[1 2]
 [3 4]
 [5 6]]
[[-0.16903085  0.89708523  0.40824829]
 [-0.50709255  0.27602622 -0.81649658]
 [-0.84515425 -0.34503278  0.40824829]]
[[-5.91607978 -7.43735744]
 [ 0.          0.82807867]
 [ 0.          0.        ]]
[[1. 2.]
 [3. 4.]
 [5. 6.]]
```

Quiz:

➔ Write a function that decompose a matrix into PLU or QR based on the passed string:

- Inputs: matrix A, String “LUP” or “QR”
- Outputs: the L, U and P matrices in case of “LUP” or the Q and R in case of “QR”.
- Assert that the decomposed matrices are correct by using the dot function to validate the answer.

6. Eigendecomposition

Eigendecomposition:

- ➔ The most used type of matrix decomposition is the eigendecomposition that decomposes a matrix into eigenvectors and eigenvalues.
- ➔ This decomposition also plays a role in methods used in machine learning, such as in the Principal Component Analysis method or PCA
- ➔ Eigendecomposition of a matrix is a type of decomposition that involves decomposing a square matrix into a set of eigenvectors and eigenvalues.
- ➔ Certain matrix calculations, like computing the power of the matrix, become much easier when we use the eigendecomposition of the matrix.

Eigendecomposition:

- ➔ A vector is an eigenvector of a matrix if it satisfies the following equation.

$$A \cdot v = \lambda \cdot v$$

- ➔ This is called the eigenvalue equation, where A is the parent square matrix that we are decomposing, v is the eigenvector of the matrix, and λ is the lowercase Greek letter lambda and represents the eigenvalue scalar.
- ➔ A matrix could have one eigenvector and eigenvalue for each dimension of the parent matrix.
- ➔ Not all square matrices can be decomposed into eigenvectors and eigenvalues.

Eigendecomposition:

- ➔ The parent matrix can be shown to be a product of the eigenvectors and eigenvalues.

$$A = Q \cdot \Lambda \cdot Q^T$$

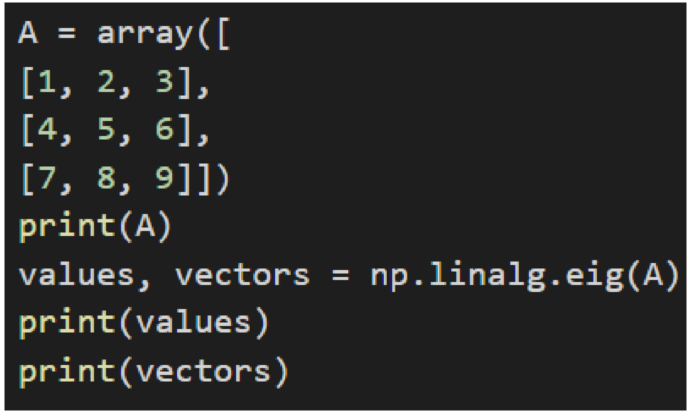
- ➔ Where Q is a matrix comprised of the eigenvectors, Λ is the uppercase Greek letter lambda and is the diagonal matrix comprised of the eigenvalues, and QT is the transpose of the matrix comprised of the eigenvectors.

Eigendecomposition Calculation:

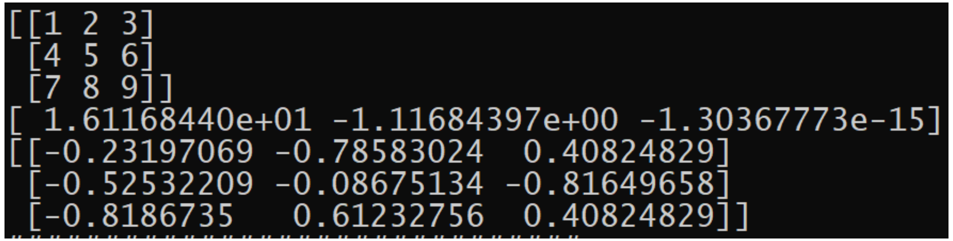
➔ An eigendecomposition is calculated on a square matrix using an efficient iterative algorithm, of which we will not go into the details.

➔ The eigendecomposition can be calculated in NumPy using the eig() function.

➔ The example below first defines a 3 × 3 square matrix.



```
A = array([
[1, 2, 3],
[4, 5, 6],
[7, 8, 9]])
print(A)
values, vectors = np.linalg.eig(A)
print(values)
print(vectors)
```



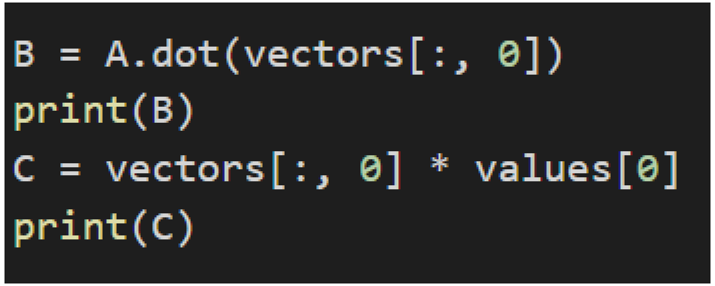
```
[[1 2 3]
 [4 5 6]
 [7 8 9]]
[ 1.61168440e+01 -1.11684397e+00 -1.30367773e-15]
[[-0.23197069 -0.78583024  0.40824829]
 [-0.52532209 -0.08675134 -0.81649658]
 [-0.8186735   0.61232756  0.40824829]]
```

Confirm an Eigenvector and Eigenvalue:

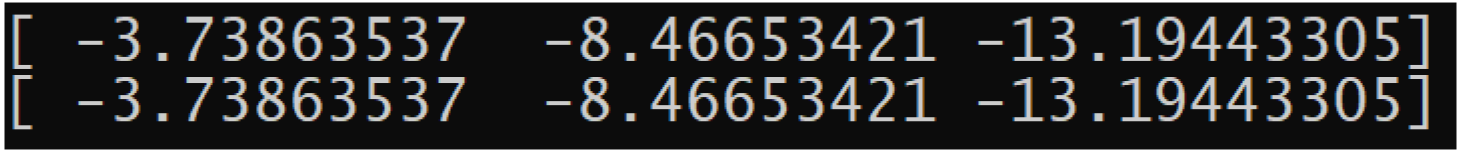
- ➔ We can confirm that a vector is indeed an eigenvector of a matrix.
- ➔ We do this by multiplying the candidate eigenvector by the value vector and comparing the result with the eigenvalue.
- ➔ The eigenvectors are returned as a matrix with the same dimensions as the parent matrix, where each column is an eigenvector.
- ➔ The first eigenvector is vectors[:, 0].
- ➔ Eigenvalues are returned as a list, where value indices in the returned array are paired with eigenvectors by column index.
- ➔ The first eigenvalue at values[0] is paired with the first eigenvector at vectors[:, 0]

Confirm an Eigenvector and Eigenvalue:

➔ The example multiplies the original matrix with the first eigenvector and compares it to the first eigenvector multiplied by the first eigenvalue.



```
B = A.dot(vectors[:, 0])
print(B)
C = vectors[:, 0] * values[0]
print(C)
```



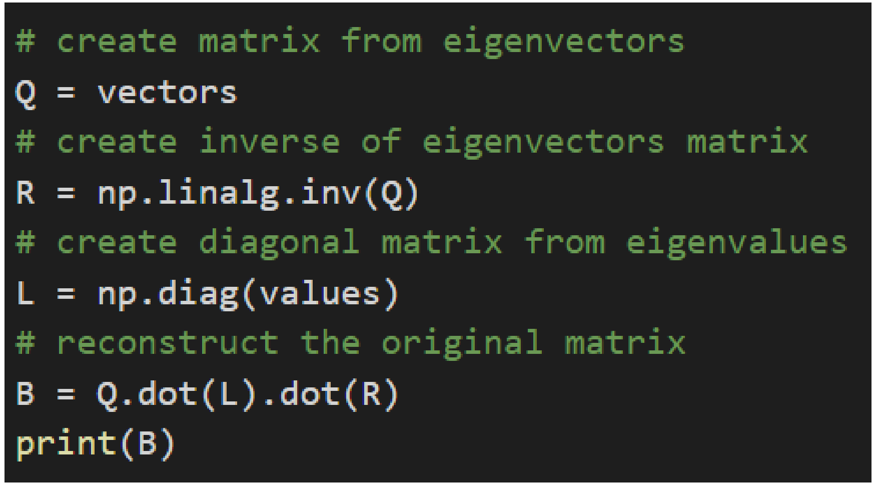
```
[ -3.73863537  -8.46653421 -13.19443305]
[ -3.73863537  -8.46653421 -13.19443305]
```

Reconstruct Matrix:

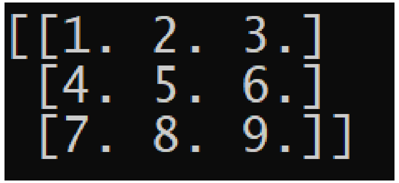
- ➔ We can reverse the process and reconstruct the original matrix given only the eigenvectors and eigenvalues.
- ➔ First, the list of eigenvectors must be taken together as a matrix, where each vector becomes a row.
- ➔ The eigenvalues need to be arranged into a diagonal matrix.
- ➔ Next, we need to calculate the inverse of the eigenvector matrix, which we can achieve with the inv() NumPy function.
- ➔ Finally, these elements need to be multiplied together with the dot() function.

Reconstruct Matrix:

➔ We can reverse the process and reconstruct the original matrix given only



```
# create matrix from eigenvectors
Q = vectors
# create inverse of eigenvectors matrix
R = np.linalg.inv(Q)
# create diagonal matrix from eigenvalues
L = np.diag(values)
# reconstruct the original matrix
B = Q.dot(L).dot(R)
print(B)
```



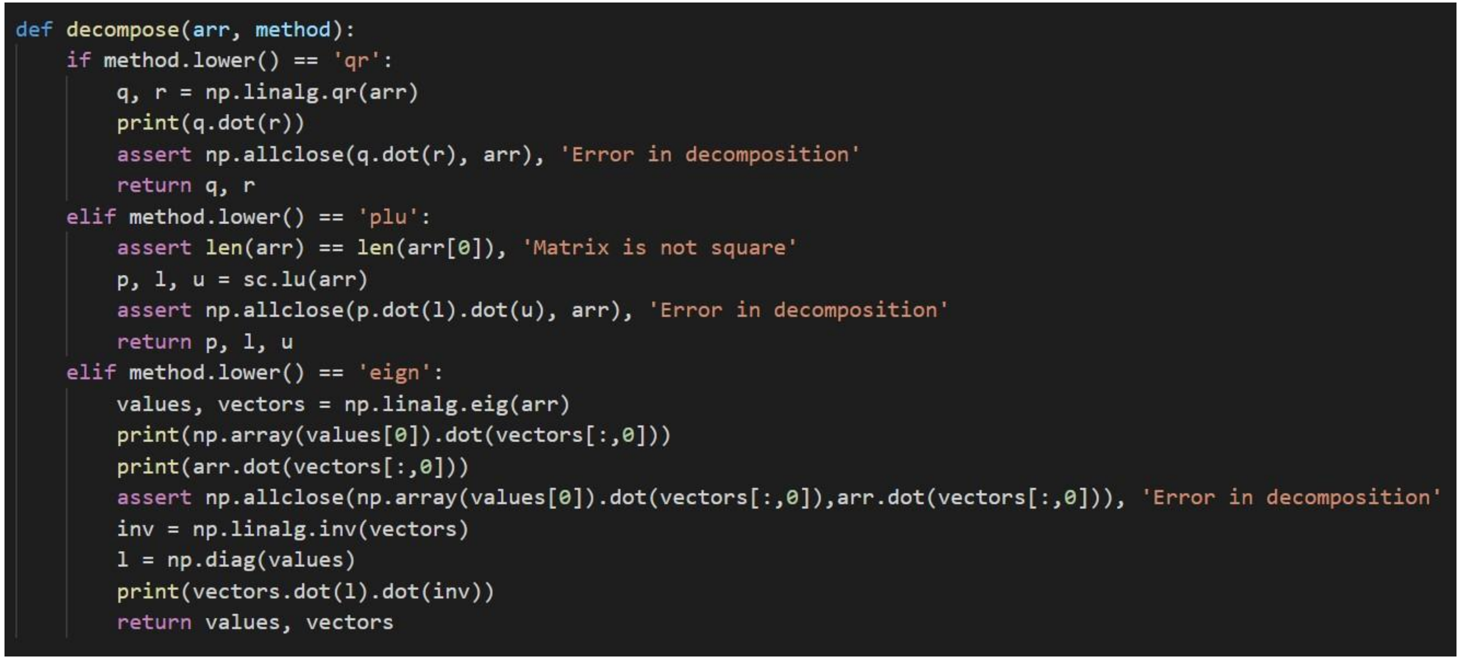
```
[[1. 2. 3.]
 [4. 5. 6.]
 [7. 8. 9.]]
```

Quiz:

- ➔ Add the Eigendecomposition method to the past decomposition function. When passing “eign”, it calculates the eigenvalues and vectors and returns them. Output will be a matrix and a vector.
  - Bonus: assert for the matrix and vector to be correct.

Quiz (Solution):

➔ Solution by: Eng. Ahmed SakrAI-6



```python
def decompose(arr, method):
    if method.lower() == 'qr':
        q, r = np.linalg.qr(arr)
        print(q.dot(r))
        assert np.allclose(q.dot(r), arr), 'Error in decomposition'
        return q, r
    elif method.lower() == 'plu':
        assert len(arr) == len(arr[0]), 'Matrix is not square'
        p, l, u = sc.lu(arr)
        assert np.allclose(p.dot(l).dot(u), arr), 'Error in decomposition'
        return p, l, u
    elif method.lower() == 'eign':
        values, vectors = np.linalg.eig(arr)
        print(np.array(values[0]).dot(vectors[:,0]))
        print(arr.dot(vectors[:,0]))
        assert np.allclose(np.array(values[0]).dot(vectors[:,0]),arr.dot(vectors[:,0])), 'Error in decomposition'
        inv = np.linalg.inv(vectors)
        l = np.diag(values)
        print(vectors.dot(l).dot(inv))
        return values, vectors
```

Any Questions?

THANK YOU!

AMIT