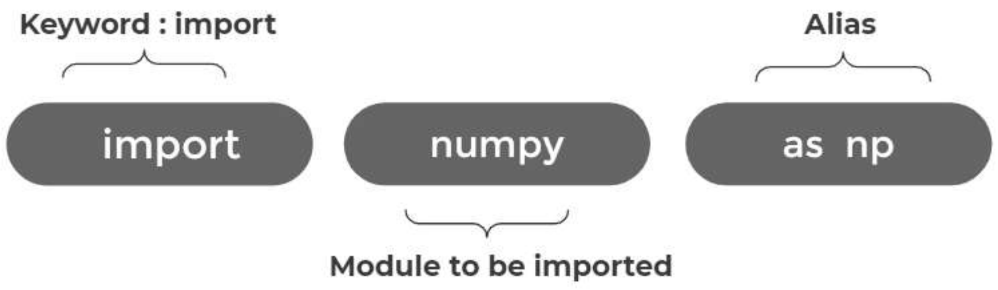
NumPy is a Python package. It stands for '**Numerical Python**'. It is a library consisting of multidimensional array objects and a collection of routines for processing of array. The NumPy package is the workhorse of data analysis, machine learning, and scientific computing in the python ecosystem.

Let's import the NumPy library in our environment

```
# Statement to import the numpy library
import numpy as np
```



The most important object defined in NumPy is an N-dimensional array type called `ndarray`. It describes the collection of items of the same type. Items in the collection can be accessed using a zero-based index.

Every item in an ndarray takes the same size of block in the memory. Each element in ndarray is an object of data-type object also called as **dtype**.

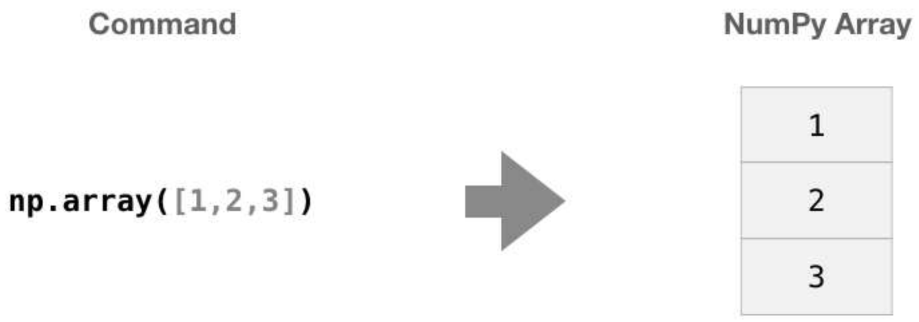
An array is a collection of items stored at continuous memory locations, all of the same data type.

Faster than Python lists for numerical operations.

Less memory usage.

You can do math on all items at once.

Essential in data science, machine learning, image processing, etc.



```
# One-dimensional array of some prime numbers
prime_array = np.array([2,3,5,7,11])
print(prime_array)
```

```
[ 2  3  5  7 11]
```

```
type(prime_array)
```

```
numpy.ndarray
```

ndarray → an object representing an N-dimensional array.

```
prime_array.shape
```

```
(5,)
```

2D ARRAY

```
# Create a two-dimensional array
another_array = np.array([[1,2],[3,4]])
print(another_array)
```

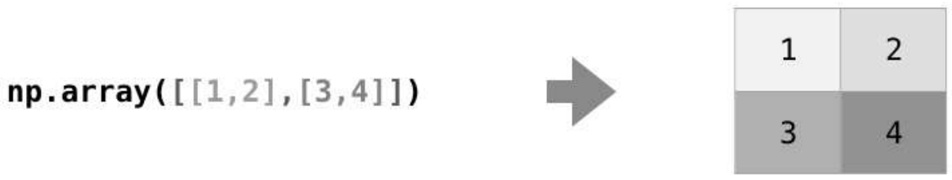
```
[[1 2]
 [3 4]]
```

```
another_array.shape
```

```
(2, 2)
```

```
another_array
```

```
array([[1, 2],
       [3, 4]])
```



One [ ] → 1D

Two [[ ]] → 2D

Three [[[ ]]] → 3D

3D ARRAY

```
# Create a three-dimensional array
threeD_array = np.array([[[1,2],[3,4]],
                        [[5,6],[7,8]]])
print(threeD_array)
print(threeD_array.shape)
```

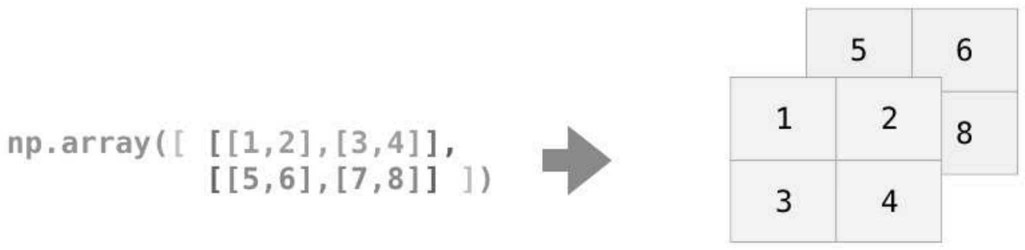
```
[[[1 2]
  [3 4]]

 [[5 6]
  [7 8]]]
(2, 2, 2)
```

🧱 2 layers

📃 Each layer has 2 rows

🔢 Each row has 2 columns



```
array_3d = np.array([
    [  # Layer 1
        [1, 2, 3, 4],
        [5, 6, 7, 8],
        [9, 10, 11, 12]
    ],
    [  # Layer 2
        [13, 14, 15, 16],
        [17, 18, 19, 20],
        [21, 22, 23, 24]
    ]
])

print(array_3d)
print("Shape:", array_3d.shape)
```

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

 [[13 14 15 16]
  [17 18 19 20]
  [21 22 23 24]]]
Shape: (2, 3, 4)
```

🧱 2 layers

📃 Each layer has 3 rows

🔢 Each row has 4 columns

What is a 4D Array?

A 4D array has 4 axes (dimensions). Its shape is typically represented as:

```
array_4d = np.random.randint(1, 10, size=(2, 3, 4, 5))  # shape: (2, 3, 4, 5)
print(array_4d)
print("Shape:", array_4d.shape)
```

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

  [[3 1 8 5 5]
   [8 3 7 2 3]
   [6 5 7 8 4]
   [8 6 7 7 6]]

  [[8 3 3 6 1]
   [9 4 2 1 8]
   [3 3 8 9 8]
   [3 2 3 7 8]]]


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

  [[5 6 5 1 7]
   [4 1 9 7 7]
   [5 6 3 5 6]
   [4 1 7 2 6]]

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

🔍 Explanation of Shape (2, 3, 4, 5): 2 → batches

3 → depth (e.g., channels or layers)

4 → rows

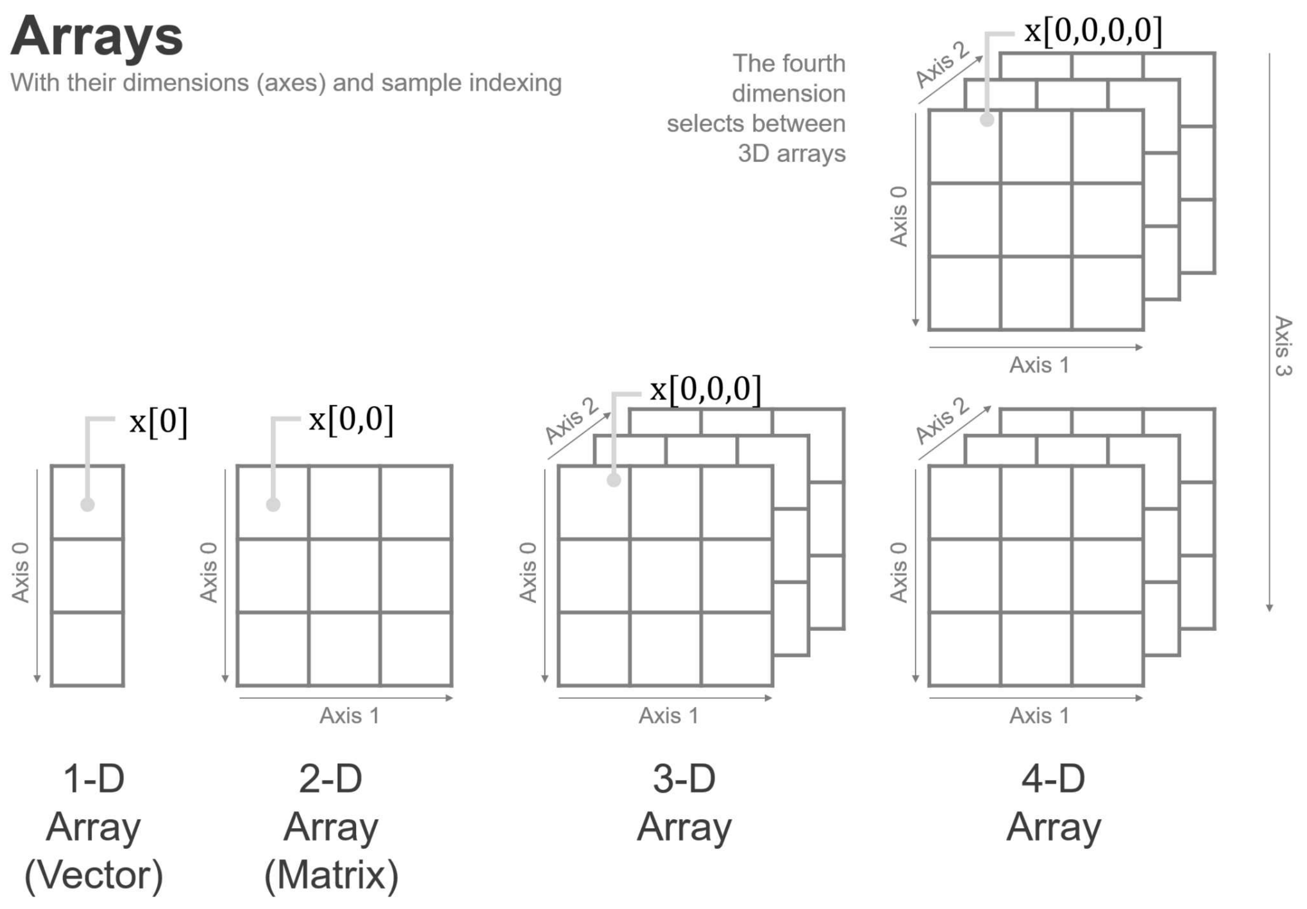
5 → columns

So, it's like:

2 blocks

each block has 3 matrices

each matrix is 4×5



LIST V/S ARRAY

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

# Python list
print(py_list * 2)        # [1, 2, 3, 1, 2, 3]

# NumPy array
print(np_array * 2)       # [2 4 6]
```

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

Array Inspection

A numpy array has following attributes

1. `ndarray.shape`
2. `ndarray.ndim`
3. `ndarray.size`
4. `ndarray.dtype`
5. `ndarray.astype`

1. `ndarray.shape`

This returns a tuple consisting of array dimensions and can be used to resize the array

```
alma_array = np.array([[1,2,3],[4,5,6]])
print(alma_array.shape)
```

```
(2, 3)
```

```
alma_array = np.array([[1,2,3,4],[4,5,9,10]])
print(alma_array.shape)   #(2, 4) is a 2D array with 2 rows and 4 columns (matrix-like).
```

```
(2, 4)
```

```
alma_array = np.array([[[1,2,3],[4,5,6]],
                       [[1,2,3],[4,5,6]],
                       [[1,2,3],[4,5,6]],
                       [[1,2,3],[4,5,6]]])
# print(alma_array)
print(alma_array.shape)      #(4, 2, 3) is a 3D array with 4 blocks, each of shape (2, 3) — like 4 matrices stacked tog
```

```
(4, 2, 3)
```

2. `ndarray.ndim`

This returns the number of array dimensions

```
array_of_integers = np.array([0,1,2,3,4,5,6,7,8,9])
print("Dimensions of 'array_of_integers' = ", array_of_integers.ndim)

sample_2d_array = np.array([[1,2,3],[4,5,6]])
print("Dimensions of 'sample_2d_array' = ", sample_2d_array.ndim)
```

```
Dimensions of 'array_of_integers' =  1
Dimensions of 'sample_2d_array' =  2
```

3. `ndarray.size`

This returns the total number of elements in the array

```
# Print the size of array
array_of_integers = np.array([0,1,2,3,4,5,6,7,8,9])
print(f"The total number of elements in our array 'array_of_integers' =  {array_of_integers.size}")

sample_2d_array = np.array([[1,2,3],[4,5,6]])
print(f"The total number of elements in our array 'sample_2d_array' =  {sample_2d_array.size}")
```

```
The total number of elements in our array 'array_of_integers' =  10
The total number of elements in our array 'sample_2d_array' =  6
```

4. `ndarray.dtype`

This returns the data type of the array

```
# Print the dtype of array
my_array = np.array([1,2,3,4,5])
print(my_array.dtype)
```

```
int64
```

```
# Print the dtype of another array
#An array of unsigned 32-bit integers.
#This means: Unicode string with max length 32 characters
#All elements are now strings in the array.


my_array = np.array([1,2.0,3,4,5,'abc'])
print(my_array.dtype)
```

```
<U32
```

5. `ndarray.astype`

This changes the data type of an array

```
# Create an array
a = np.array([2,3,4])
print(a)
print(a.dtype)

# Change the datatype of the array to float
a_ = a.astype('float64')
print(a_)
print(a_.dtype)
```

```
[2 3 4]
int64
[2. 3. 4.]
float64
```

Why Numpy?

Numpy data structures perform better in:

- Memory - Numpy data structures take up less space
- Performance - they have a need for speed and are faster than lists
- Functionality - SciPy and NumPy have optimized functions such as linear algebra operations built in.

Memory

```
import numpy as np
import sys

# declaring a list of 10 elements
S = [0,1,2,3,4,5,6,7,8,9]
print("Size of each element of list in bytes: ",sys.getsizeof(S))
print("Size of the whole list in bytes: ",sys.getsizeof(S)*len(S))

# declaring a Numpy array of 10 elements
D= np.array([0,1,2,3,4,5,6,7,8,9] )
print("Size of each element of the Numpy array in bytes: ",D.itemsize)
print("Size of the whole Numpy array in bytes: ",D.size*D.itemsize)
```

```
Size of each element of list in bytes:  144
Size of the whole list in bytes:  1440
Size of each element of the Numpy array in bytes:  8
Size of the whole Numpy array in bytes:  80
```

Performance

```
import numpy as np
import time

# size of arrays and lists
size = 100000

# declaring lists
list1 = range(size)
list2 = range(size)

# declaring arrays
array1 = np.arange(size)
array2 = np.arange(size)
```

```
# capturing time before the multiplication of Python lists
initialTime = time.time()

# multiplying  elements of both the lists and stored in another list
resultantList = [(a * b) for a, b in zip(list1, list2)]

# calculating execution time
print("Time taken by Lists to perform multiplication:",
      (time.time() - initialTime),
      "seconds")

# capturing time before the multiplication of Numpy arrays
initialTime = time.time()

# multiplying  elements of both the Numpy arrays and stored in another Numpy array
resultantArray = array1 * array2

# calculating execution time
print("Time taken by NumPy Arrays to perform multiplication:",
      (time.time() - initialTime),
      "seconds")
```

```
Time taken by Lists to perform multiplication: 0.012225151062011719 seconds
Time taken by NumPy Arrays to perform multiplication: 0.0007731914520263672 seconds
```

Different ways of creating an ndarray

Uninitialized

Initialized with zeros

```
# Intializing the array with zeros
zeros_array = np.zeros((3,3)) # Default dtype = float
another_zeros_array = np.zeros((4,3) , dtype=int)
print(zeros_array)
print(another_zeros_array)
```

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

Initialized with ones

```
# Intializing the array with ones
ones_array = np.ones((3,3)) # Default dtype = float
another_ones_array = np.ones((2,3) , dtype=int)
print(ones_array)
print(another_ones_array)
```

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

```
bunch_of_ones = np.ones((4,3))
bunch_of_ones
```

```
array([[1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.]])
```

```
# Trying to change the shape of our array
bunch_of_ones.shape = (2,6)
```

```
bunch_of_ones
```

```
array([[1., 1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1., 1.]])
```

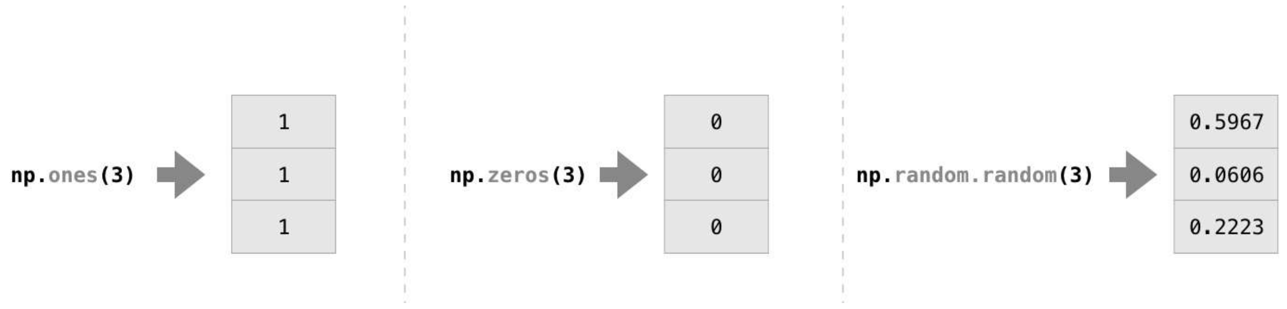
Array from numerical ranges

```
# Array with values from 0 to N
array_with_range = np.arange(5)
array_with_range
```

```
array([0, 1, 2, 3, 4])
```

```
another_array_with_range = np.arange(10,20,2)
another_array_with_range
```

```
array([10, 12, 14, 16, 18])
```



Array from specified numbers

```
# To get an array with specified input element of desired shape
# Parameters for the function - np.full(shape,value)
a = np.full((3,3),30)
print(a)
```

```
[[30 30 30]
 [30 30 30]
 [30 30 30]]
```

```
a = np.ones((3,3))
print(a)
```

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

```
# to get an identity matrix
a = np.identity(4)
print(a)
```

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

Creating ndarray with existing data

- List to Ndarray
- Tuple to Ndarray

For both above conversions, we use a single function

```
# convert list to ndarray
typical_list = [1,2,3]
corresponding_array = np.asarray(typical_list)
corresponding_array
```

```
array([1, 2, 3])
```

```
# convert tuple to ndarray
typical_tuple = (1,2,3)
corresponding_array = np.asarray(typical_tuple)
corresponding_array
```

```
array([1, 2, 3])
```

Indexing and Slicing Arrays

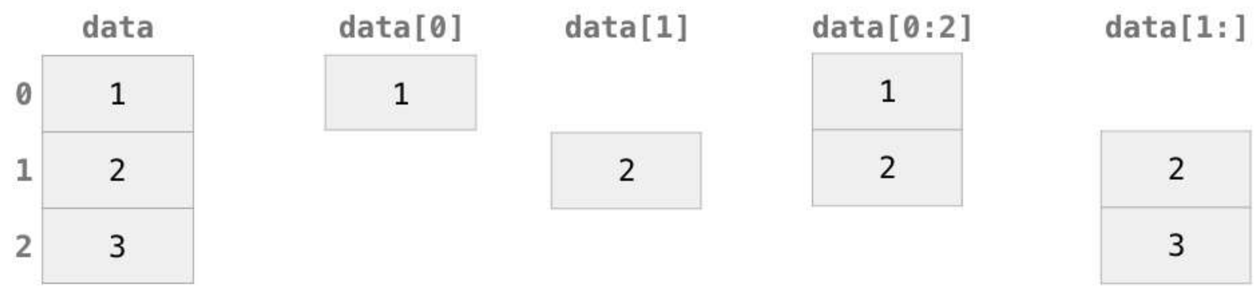
Contents of ndarray object can be accessed and modified by indexing or slicing. Lets see how to extract specific values from the array using these techniques.

Slicing means retrieving elements from one index to another index. All we have to do is to pass the starting and ending point in the index. Slicing includes the starting index but excludes the ending index.

`[start:end]` -> Used to slice arrays without a gap

`[start:end:step_size]` -> Used to slice arrays with a fixed interval

Slicing 1-D Arrays



```
# slice items between indexes
a = np.arange(10)
a[2:5]
```

```
array([2, 3, 4])
```

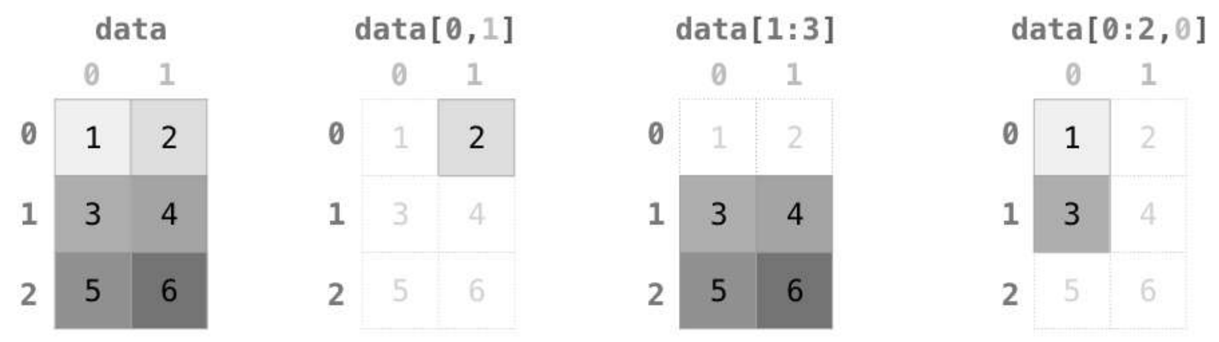
```
# slice items between indexes in steps
a = np.random.random(10)
a
```

```
array([0.66387291, 0.90176839, 0.65985137, 0.87397291, 0.58489737,
       0.74395716, 0.33026795, 0.65324659, 0.7408184 , 0.85595203])
```

```
a[::2]
```

```
array([0.66387291, 0.65985137, 0.58489737, 0.33026795, 0.7408184 ])
```

Slicing 2-D Arrays



```
# retrieving elements from a 2-D array
a = np.array([[1,2,3],[4,5,6]])
print(a[1,2])
# print(a[1,2])              #a[1,2] means:

                             #Row index 1 → [4, 5, 6]

                             #Column index 2 in that row → 6
```

```
6
```

```
# getting the entire array
a = np.array([[1,2,3],[4,5,6],[7,8,9]])
print(a)
print(a[0:2])
print(a[0:2,-2:])      #0:2 → selects both rows: [[1, 2, 3], [4, 5, 6]]

                       #-2: → selects the last 2 columns (columns at index -2 and -1, i.e., columns 1 and 2)
```

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

```
# getting the 1st row
a = np.array([[1,2,3],[4,5,6],[7,8,9]])
print(a[0:1,:])
```

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

Slicing 3-D Arrays

```
# For 3-D array slicing the format is array[index,row,column]
a = np.array([[[1,2],[3,4],[5,6]],
             [[7,8],[9,10],[11,12]],
             [[13,14],[15,16],[17,18]]])
print(a)
print(a[0:2])
# print(a[0:2,1:])
print(a[0:2,1:,1:])
```

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

 [[ 7  8]
  [ 9 10]
  [11 12]]

 [[13 14]
  [15 16]
  [17 18]]]
[[[ 1  2]
  [ 3  4]
  [ 5  6]]

 [[ 7  8]
  [ 9 10]
  [11 12]]]
```

```
[[[ 4]
  [ 6]]

 [[10]
  [12]]]
```

Step-by-step: 0:2 → Selects the first two blocks (index 0 and 1)

1: → From row index 1 onwards → includes rows 1 and 2

1: → From column index 1 onwards → includes the last column only

Let's extract manually: From Block 0 ([[1,2],[3,4],[5,6]])

Rows 1 and 2 → [[3,4], [5,6]]

Columns 1 onwards → [[4], [6]]

From Block 1 ([[7,8],[9,10],[11,12]])

Rows 1 and 2 → [[9,10], [11,12]]

Columns 1 onwards → [[10], [12]]

As of now, we are familiar with the basics of NumPy. It is time to delve a bit deeper into some operations in NumPy

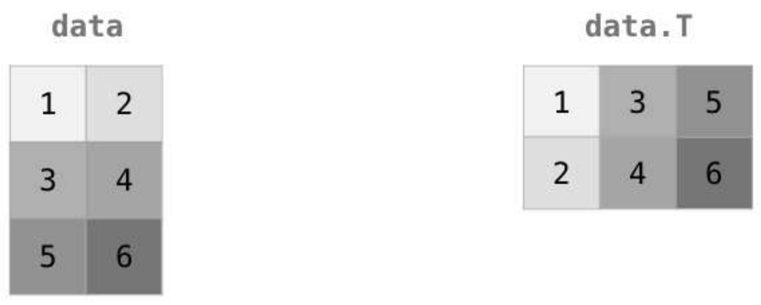
```
# Load the Numpy
import numpy as np
```

Array Manipulation

- `np.transpose(array)` - To transpose array
- `np.reshape(array,shape)` - To Change shape of an array
- `np.resize(array,shape)` - To return new array with shape
- `np.flatten(array)` - To flatten the array

`np.transpose()`

Transpose is defined as an operator which flips a matrix over its diagonal; that is, it switches the row and column indices of the matrix A by producing another matrix, often denoted by A^T



```
# Create a 2 D Array
arravali = np.array(np.random.randint(10,45,(8,2)))
print(f"The defined array is \n{arravali}")
print(f" \n The shape of the array is {arravali.shape}")
```

```
The defined array is
[[15 39]
 [33 39]
 [32 35]
 [14 38]
 [30 16]
 [36 24]
 [30 35]
 [19 43]]

 The shape of the array is (8, 2)
```

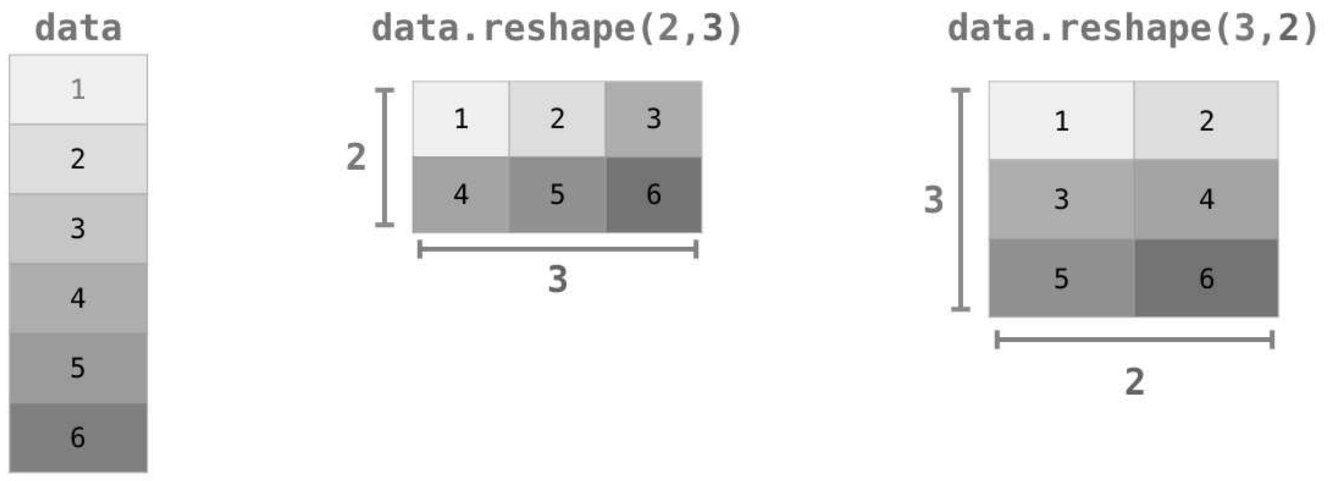
```
# Storing the transpose in another matrix object
nilgiri =np.transpose(arravali)
print(f" The transpose of the array is \n{nilgiri}")
print(f" \n The shape of the array is {nilgiri.shape}")
```

```
 The transpose of the array is
[[15 33 32 14 30 36 30 19]
 [39 39 35 38 16 24 35 43]]

 The shape of the array is (2, 8)
```

np.reshape(shape)

Gives a new shape to an array without changing its data.



```
# Create a two-dimensional array
rand = np.array(np.random.randint(10,45,(3,8))) # 2D
print(f"The defined array is \n{rand}")
print(f" \n The shape of the array is {rand.shape}")
```

```
The defined array is
[[40 28 43 38 27 40 21 27]
 [34 15 23 12 31 30 30 36]
 [33 16 11 29 41 26 43 41]]

 The shape of the array is (3, 8)
```

```
# Now we can reshape this to 1-D,2-D or 3-D array
new_shape = (2,4,3)
reshaped_rand = np.reshape(rand,new_shape)
print(f"The reshaped array is \n{reshaped_rand}")
print(f" \n The shape of the array is {reshaped_rand.shape}")
```

```
The reshaped array is
[[[40 28 43]
  [38 27 40]
  [21 27 34]
  [15 23 12]]

 [[31 30 30]
  [36 33 16]
  [11 29 41]
  [26 43 41]]]

 The shape of the array is (2, 4, 3)
```

np.resize(shape)

Changes shape and size of array in-place.

```
# Intialize a 3D array
random_3D_cohort = np.array(np.random.randint(10,45,(2,4,6)))
print(f"The defined array is \n{random_3D_cohort}")
print(f" \n The shape of the array is {random_3D_cohort.shape}")
```

```
The defined array is
[[[19 40 26 31 31 33]
  [27 24 18 28 43 41]
  [31 20 34 40 43 23]
  [21 20 31 12 18 40]]

 [[18 41 20 32 22 40]
  [39 22 16 25 43 15]
  [27 23 14 25 28 31]
  [31 16 27 44 27 27]]]

 The shape of the array is (2, 4, 6)
```

```
# Resize the created 3D array
new_shape = (4,3,6)
resized_random_3D_cohort = np.resize(random_3D_cohort,new_shape)
print(f"The resized array is \n{resized_random_3D_cohort}")
print(f" \n The shape of the array is {resized_random_3D_cohort.shape}")
```

```
The resized array is
[[[19 40 26 31 31 33]
  [27 24 18 28 43 41]
  [31 20 34 40 43 23]]

 [[21 20 31 12 18 40]
  [18 41 20 32 22 40]
  [39 22 16 25 43 15]]

 [[27 23 14 25 28 31]
  [31 16 27 44 27 27]
  [19 40 26 31 31 33]]

 [[27 24 18 28 43 41]
  [31 20 34 40 43 23]
  [21 20 31 12 18 40]]]

 The shape of the array is (4, 3, 6)
```

`np.flatten()`

Returns flattened 1D arrays

```
# Intialize a 3D array
random_3D_cohort = np.array(np.random.randint(10,45,(2,4,6)))
print(f"The defined array is \n{random_3D_cohort}")
print(f" \n The shape of the array is {random_3D_cohort.shape}")
```

```
The defined array is
[[[29 42 31 35 28 38]
  [32 39 24 24 44 32]
  [28 27 26 20 19 38]
  [11 32 32 25 21 35]]

 [[28 39 15 42 35 16]
  [31 32 33 43 42 44]
  [35 24 13 14 22 16]
  [40 24 29 44 23 43]]]

 The shape of the array is (2, 4, 6)
```

```
# Flatten the 3D array
random_3D_cohort.flatten()
```

```
array([29, 42, 31, 35, 28, 38, 32, 39, 24, 24, 44, 32, 28, 27, 26, 20, 19,
       38, 11, 32, 32, 25, 21, 35, 28, 39, 15, 42, 35, 16, 31, 32, 33, 43,
       42, 44, 35, 24, 13, 14, 22, 16, 40, 24, 29, 44, 23, 43])
```