greatlearning
Learning for Life

Python NumPy

greatlearning
Learning for Life

Agenda

- Introduction to Numpy
- Introduction to Array
  - Creating Numpy Array
  - Attributes of Array
  - Array Methods
  - Indexing and Slicing
  - Comparison and Arithmetic Operations/Function on Array
  - Concatenating and Stacking Array
  - Splitting Array
  - Iterating through Arrays

What is NumPy?

What is NumPy?

greatlearning
Learning for Life

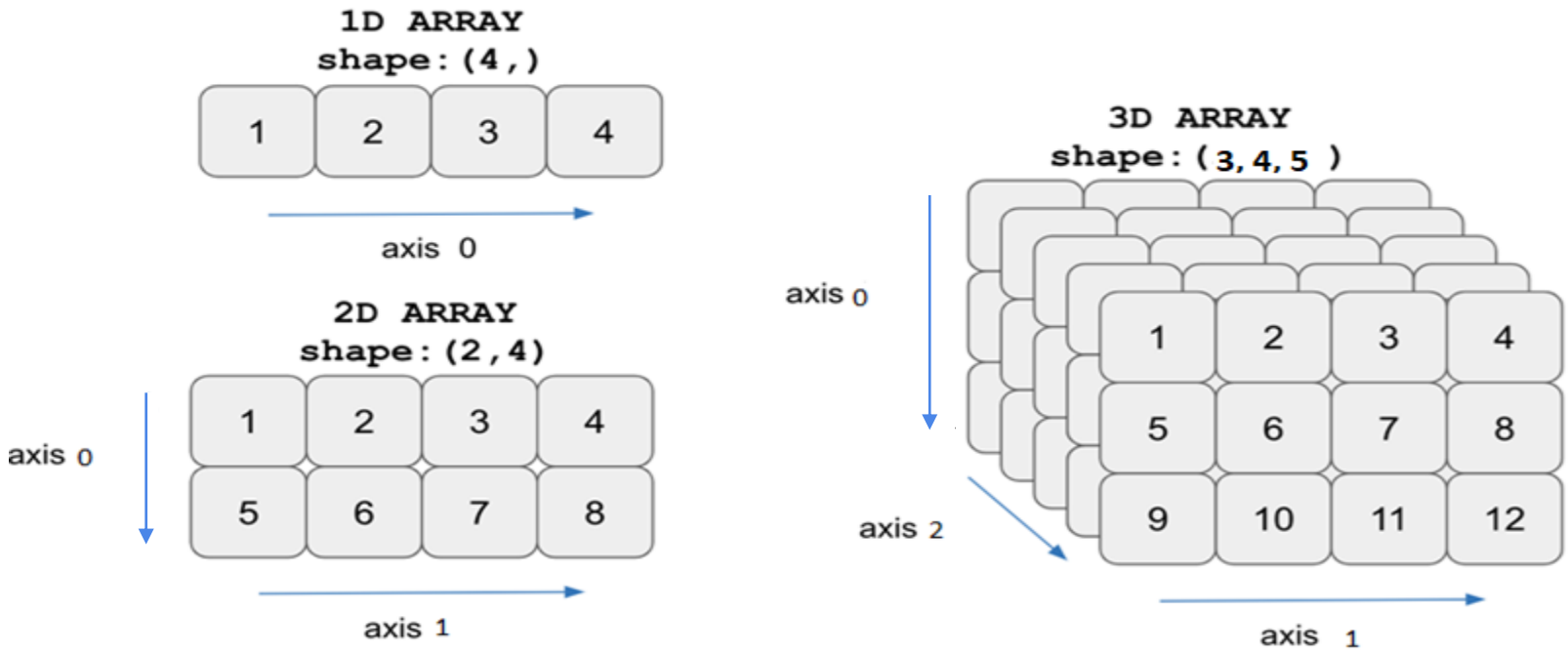
- A package in python which stands for ‘Number Python’
- It is used for mathematical and scientific computations, which contains multi-dimensional arrays and matrices
- NumPy also provides a module called ‘linalg’ which contains various functions (such as det, eig, norm) to apply linear algebra on NumPy array
- NumPy array is a central structure of the NumPy library. It is an n-dimensional array object containing rows and columns

What is an array?

What is an array?

- A numpy array looks similar to a list
- An array is a grid of values, indexed by a tuple of positive integers
- It usually contains numeric values. However it can contain string values
- They work faster than lists
- An array can be n-dimensional

1D, 2D and 3D NumPy array



Creating NumPy Array

Create NumPy array using list

greatlearning
Learning for Life

Use np.array() to create a numpy array from a list

```
# list
weight = [57,62,73,64]

# check type of 'weight'
print('Type of weight:', type(weight))

# create an array
weight_array = np.array(weight)
print('weight_array:', weight_array)

# check type of 'weight_array'
print('Type of weight_array:', type(weight_array))
```

```
Type of weight: <class 'list'>
weight_array: [57 62 73 64]
Type of weight_array: <class 'numpy.ndarray'>
```

Create NumPy array of random numbers

Create an array of 10 random numbers using random()

```
np.random.random(size = 10)
```

```
array([0.37484319, 0.29868527, 0.71745869, 0.65446704, 0.70739843,
       0.60204932, 0.37089291, 0.26806262, 0.39622109, 0.61836867])
```

The random() returns random numbers from uniform distribution over the half-open interval [0.0, 1.0). The half-open interval includes 0 but excludes 1. The required number of random numbers is passed through the ‘size’ parameter

Create NumPy array of random numbers

The rand() creates an array of the given shape and populates it with random variables derived from a uniform distribution between (0, 1)

```
np.random.rand(5)
```

```
array([0.92815896, 0.41897795, 0.37806823, 0.00307337, 0.73693317])
```

Create NumPy array of random numbers

The randn() returns a value (or a set of values) from the "Standard Normal" distribution, unlike rand() which is from a uniform distribution. A standard Normal Distribution has mean 0 and standard deviation 1

```
np.random.randn(5)
```

```
array([ 0.21347567, -0.52623156, -0.66624803,  0.280878  ,  0.72793944])
```

Create NumPy array of random numbers

The randint() returns random integers from low (inclusive) to high (exclusive)

```
np.random.randint(5,20)       # Returns one rand integer between the values 5 & 19(20 is excluded)
```

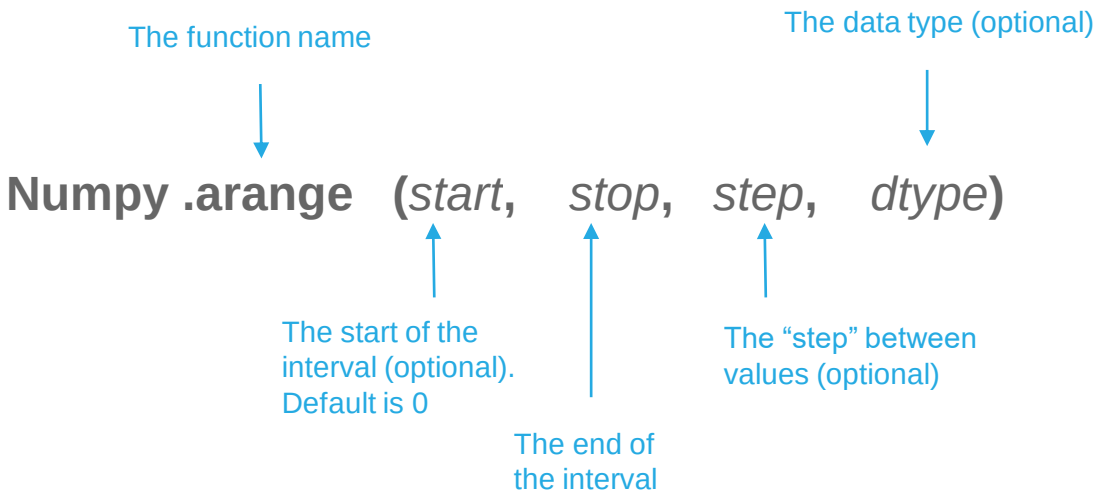
```
16
```

```
np.random.randint(20,50,5)  # Returns  5 rand integers between 20 & 49(50 is excluded)
```

```
array([32, 44, 30, 41, 41])
```

Create NumPy array using arange()

- The np.arange() creates a NumPy array
- The numbers generated have the same difference
- The function generates as many possible numbers in the given range



Create NumPy array using arange()

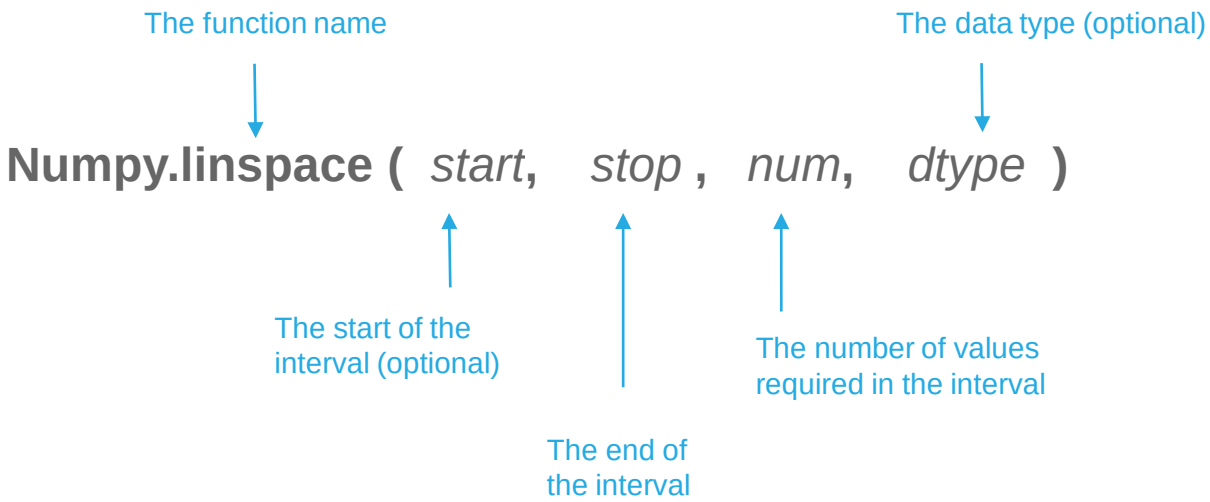
```
# create an array of even integers between 10 to 20
# 'start' is inclusive and 'stop' is exclusive
# 'step' returns values with specified step size
even_num = np.arange(start = 10, stop = 20, step = 2)
even_num
```

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

The np.arange() produces a sequence of evenly spaced values from 10 to 18 with a difference of 2, stored as an ndarray object (i.e., a NumPy array)

Create NumPy array using linspace()

- The function generates a specified number of values in the desired range
- The syntax for np.linspace() is as follows:



Create NumPy array using linspace()

```
# create an array using linspce
# 'start' is the initial value of the sequence
# 'stop' is the end value of the sequence
# 'num' is the number of values to be
np.linspace(start = 0, stop = 50, num = 20)
```

```
array([ 0.        ,  2.63157895,  5.26315789,  7.89473684, 10.52631579,
       13.15789474, 15.78947368, 18.42105263, 21.05263158, 23.68421053,
       26.31578947, 28.94736842, 31.57894737, 34.21052632, 36.84210526,
       39.47368421, 42.10526316, 44.73684211, 47.36842105, 50.        ])
```

The np.linspace() produces a sequence of 20 evenly spaced values from 0 to 50, stored as an ndarray object (i.e., a NumPy array)

Create NumPy array

- Create 1D numpy array of zeros using np.zeros()

```
np.zeros(shape = 4)
```

```
array([0., 0., 0., 0.])
```

- Create 1D numpy array of ones using np.ones()

```
# 'shape' returns array of given dimension
np.ones(shape = 5)
```

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

Create NumPy array

- Create an empty matrix using np.empty()
- It returns the matrix with arbitrary (uninitialized) values of given shape and data type
- ‘dtype = object’ returns None values

```
# create a 2x2 empty matrix
matrix_empty = np.empty((2,2), dtype=float)
matrix_empty
```

```
array([[9.71814759e-313, 7.60339646e-315],
       [7.60339709e-315, 7.60339772e-315]])
```

Create NumPy array

- Create a full matrix using np.full()
- Returns the matrix of given shape with the value passed through the 'fill_value' parameter

```
# create a 2x2 full matrix
matrix_full = np.full((2,2), fill_value=10)
matrix_full
```

```
array([[10, 10],
       [10, 10]])
```

Create NumPy array

Create an identity matrix using np.identity()

```
np.identity(3)
```

```
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

Create NumPy array

- Create a matrix using np.eye()
- Returns the NxM matrix with value ‘1’ on the k-th diagonal and remaining entries as zero
- K = 0 represents main diagonal
  K > 0 represents upper diagonal
  K < 0 represents lower diagonal

```
# create a 4x3 matrix
np.eye(N = 4, M = 3, k=0)
```

```
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.],
       [0., 0., 0.]])
```

Create NumPy array

Numpy arrays can contain strings as well

```
1  array_string = np.array(["Python", "for","Data","Science"])
2  array_string
```

```
array(['Python', 'Data', 'R', 'Great Learning'], dtype='<U14')
```

Attributes of Array

Attributes of array

Attributes are the features/characteristics of an object that describes the object.
Attributes do not have parentheses following them

A few attributes of the numpy array:

- size
- shape
- ndim
- dtype

Attributes of array

The size returns the number of elements in an array

```
num_array = np.array([1,4,-2,8,7.4,3,9,-8,5])
print('No. of elements:', num_array.size)
```

```
No. of elements: 9
```

Attributes of array

- The shape returns the number of rows and columns of the array respectively

```
num_array = np.array([(4,3,2),(7,5,6)])
print('Shape:', num_array.shape)
```

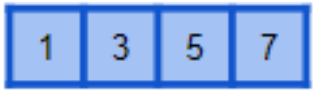
```
Shape: (2, 3)
```

- The ndim returns the number of axes (dimension) of the array

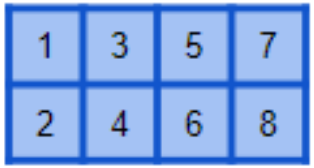
```
num_array = np.array([(4,3,2),(7,5,6)])
print('Dimension:', num_array.ndim)
```

```
Dimension: 2
```

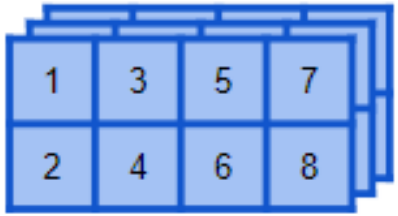
Attributes of array - ndim



Vector (1D array)
Dimension = 1



Matrix (2D array)
Dimension = 2



3D array
Dimension = 3

Attributes of array

The dtype returns the type of the data along with the size in bytes

```
num_array = np.array([4,1.57,3.8])
print('dtype:', num_array.dtype)
```

```
dtype: float64
```

In the example, the array consists of 64-bit floating-point numbers. Thus, the dtype of the array is float64

greatlearning
Learning for Life

Numpy Array Methods

NumPy array methods

Methods are functions stored in the object's class that takes parameters in the parentheses and returns the modified object

For example, the reshape() changes the number of rows and columns of the original array, without changing the data

```
original_array = np.array([(4,9),(7,2),(5,6)])
print("original array:", original_array)

# pass the new shape to change the shape of original_array
reshaped_array = original_array.reshape(2,3)
print("reshaped array:", reshaped_array)
```

```
original array: [[4 9]
 [7 2]
 [5 6]]
reshaped array: [[4 9 7]
 [2 5 6]]
```

Reshape an array with reshape()

In the example, the reshape(2, 2) is reshaping a 1 X 4 array to 2 X 2 array

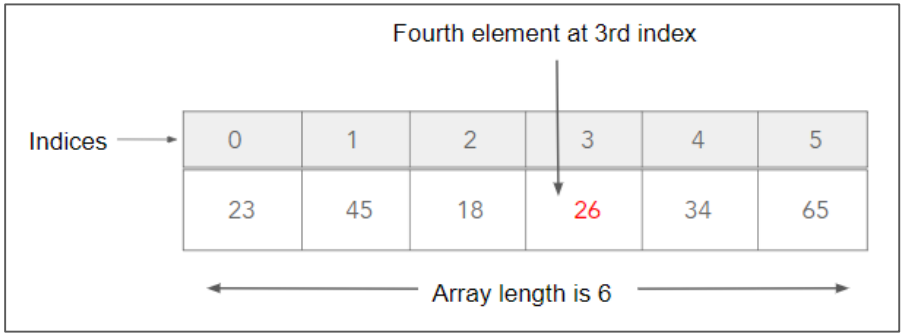
```
# create a 2x2 matrix from a sequence
num_matrix = np.arange(4)
print('Original array:', num_matrix)

print('Reshaped array: \n', num_matrix.reshape(2,2))
```

```
Original array: [0 1 2 3]
Reshaped array:
 [[0 1]
 [2 3]]
```

Indexing of an Array

Index 1D array

Each element in the array can be accessed by passing the positional index of the element. The index for an array starts at 0 from left. It starts at -1 from the right.

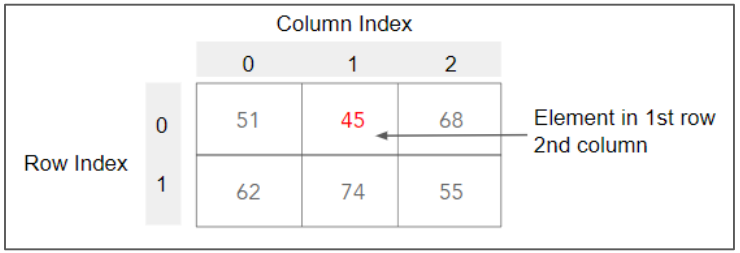



```
# consider a 1D array
age = np.array([23,45,18,26,34,65])

# retrieve the 1st element
print('First element:', age[0])

# retrieve the 4th element
print('Fourth element:', age[3])

# retrieve the last element
print('Last element:', age[-1])
```

```
First element: 23
Fourth element: 26
Last element: 65
```

Index 2D array

Element in 2D array can be accessed by the row and column indices. We can also select a specific row or column by passing the respective index




```
# consider a 2D array
weight = np.array([(51,45,68),(62,74,55)])

# retrieve the element in the 1st row and 2nd column
a12 = weight[0][1]
print('a12 :', a12)

# retrieve the elements in 2nd row for all the columns
print('second row:', weight[1,:])
```

```
a12 : 45
second row: [62 74 55]
```

Slicing of an Array

Slice arrays

	Expression	Shape	Output
10 20 30 40 50 60 70 80 90	array[1: , :2]	(2,2)	([[40,50], [70,80]])
10 20 30 40 50 60 70 80 90	array[1] array[1:2, :]	(3,) (1,3)	([40,50,60])
10 20 30 40 50 60 70 80 90	array[0,1:] array[0:1, 1:]	(2,) (1,2)	([20,30])

Slice 1D array

greatlearning
Learning for Life

Indexing allows us to extract a single element from the array; while, slicing can be used to access more than one element

```
# consider a 1D array
age = np.array([23,45,18,26,34,65])

# retrieve the 3rd and 4th element
# pass the range of positional indices as [2:4]
# last number in range is always exclusive
print('3rd and 4th element:', age[2:4])

# retrieve every 2nd element from 45
# pass the range of indices as [1::2]
# 2 is the step_size and (1:) returns all elements from 45
print('Every 2nd element:', age[1::2])

# retrieve all the elements before 34
# range [:4] returns all the elements upto 3rd positional index
print('Elements before 34:', age[:4])
```

```
3rd and 4th element: [18 26]
Every 2nd element: [45 26 65]
Elements before 34: [23 45 18 26]
```

Slice 2D array

greatlearning
Learning for Life

Slicing can be used to return a sub-matrix of the original matrix

```
# consider a 2D array
weight = np.array([(51,45,68),(62,74,55)])

# retrieve the elements in the 2nd row and first two columns
print('selected elements:', weight[1,0:2])

# retrieve the elements in 2nd and 3rd column
print('elements in 2nd and 3rd column:', weight[:,1:])
```

```
selected elements: [62 74]
elements in 2nd and 3rd column: [[45 68]
 [74 55]]
```

greatlearning
Learning for Life

Comparison Operations on Array

Comparison operations on array

Comparison operators can be used to validate specific condition on each element of the array. By default, the output of the comparison statement is boolean

```
# consider a 1D array
age = np.array([13,45,22,6,14,25])

# check if the age is greater than 18 or not
# the below comparison will return the boolean output
print('boolean output:', age > 18)

# pass the boolean output to retrieve the age greater than 18
print('age greater than 18:',age[age > 18])
```

```
boolean output: [False  True  True False False  True]
age greater than 18: [45 22 25]
```

Arithmetic Operations on Array

Arithmetic operations on array

Addition and Subtraction of 1D array

```
# consider odd_num and even_num
odd_num = np.array([3,7,5,9])
print('odd_num:',odd_num)
even_num = np.array([8,10,4,2])
print('even_num:',even_num)

# add odd_num and even_num element-wise
sum = odd_num + even_num
print('Addition:', sum)

# subtract the even_num from odd_num
sub = odd_num - even_num
print('Subtraction:', sub)
```

```
odd_num: [3 7 5 9]
even_num: [ 8 10  4  2]
Addition: [11 17  9 11]
Subtraction: [-5 -3  1  7]
```

Arithmetic operations on array

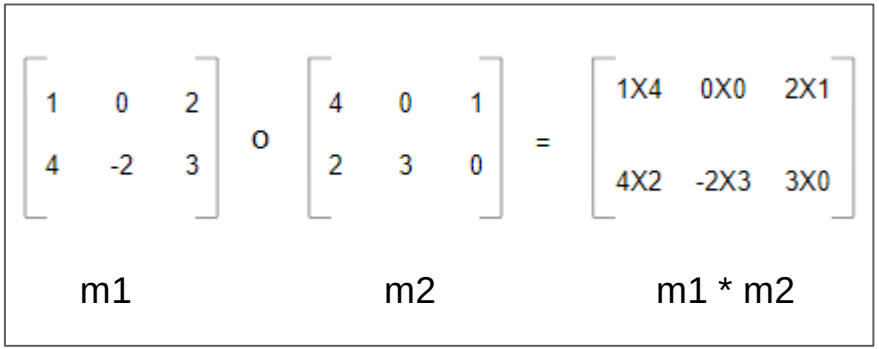
Multiply each element in the array by 2

```
num_array = np.array([1,2,3,4,5])

print('updated_array:', num_array*2)
```

```
updated_array: [ 2  4  6  8 10]
```

Arithmetic operations on array

Element-wise multiplication of two 3x3 matrices



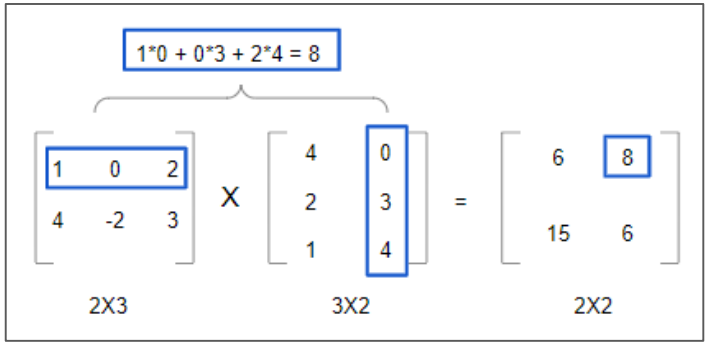

```
num_matrix_1 = np.array([[1,0,2],[4,-2,3]])
num_matrix_2 = np.array([[4,0,1],[2,3,0]])

# element-wise multiplication
prod = num_matrix_1*num_matrix_2
print('element-wise multiplication:\n',prod)
```

```
element-wise multiplication:
 [[ 4  0  2]
 [ 8 -6  0]]
```

This product is also known as ‘Hadamard Product’

Arithmetic operations on array

Matrix multiplication of two 3x3 matrices




```
num_matrix_1 = np.array([[1,0,2],[4,-2,3]])
num_matrix_2 = np.array([[4,0],[2,3],[1,4]])

# dot() returns the matrix multiplication of the matrices
multi = num_matrix_1.dot(num_matrix_2)
print('matrix multiplication:\n',multi)
```

```
matrix multiplication:
 [[ 6  8]
 [15  6]]
```

Arithmetic Functions on Array

Arithmetic functions on array

The min() returns the minimum value in the array

```
age = np.array([14, 74, 84, 26, 56])
print("Minimum:", age.min())
```

```
Minimum: 14
```

The max() returns the maximum value in the array

```
age = np.array([14, 74, 84, 26, 56])
print("Maximum:", age.max())
```

```
Maximum: 84
```

Arithmetic functions on array

- The sum() returns the sum of all the elements in the array

```
age = np.array([14, 74, 84, 26, 56])
print("Sum of all elements:", age.sum())
```

```
Sum of all elements: 254
```

- The mean() returns the mean of the values in the array

```
age = np.array([14, 74, 84, 26, 56])
print("Mean:", age.mean())
```

```
Mean: 50.8
```

Arithmetic functions on array

- The var() returns the variance of all the elements in the array

```
age = np.array([14, 74, 84, 26, 56])
print("variance:", age.var())
```

```
variance: 727.36
```

- The std() returns the standard deviation of all the elements in the array

```
age = np.array([14, 74, 84, 26, 56])
print("Standard deviation:", age.std())
```

```
Standard deviation: 26.969612529660118
```

Arithmetic functions on array

- The np.square() returns the square of the elements

```
age = np.array([14, 74, 84, 26, 56])
print("Square:", np.square(age))
```

```
Square: [ 196 5476 7056  676 3136]
```

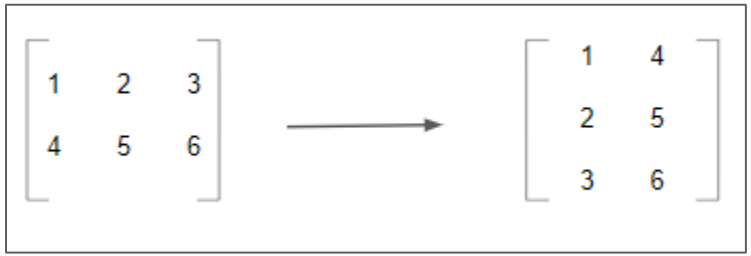
- The np.power() is used to raise the numbers in the array to the given value

```
age = np.array([14, 74, 84, 26, 56])
print("Cube:", np.power(age,3))
```

```
Cube: [  2744 405224 592704  17576 175616]
```

Transpose of an array

The np.transpose() permutes the dimension of an array. By default, it reverses the dimension of the array




```
# create a 2x3 matrix
num_matrix = np.array([[1,2,3],[4,5,6]])
print('Original matrix:\n', num_matrix)

# transpose the matrix
print('Transposed matrix:\n', np.transpose(num_matrix))
```

```
Original matrix:
 [[1 2 3]
 [4 5 6]]
Transposed matrix:
 [[1 4]
 [2 5]
 [3 6]]
```

Concatenate the Arrays

Concatenate 1D array

Two or more arrays will get joined along existing (first) axis, provided they have the same shape

```
# create three 1D array
order_1 = np.array([1,4,5,7])
order_2 = np.array([6,9,3])
order_3 = np.array([2,7,5,9])

# concatenate the arrays
np.concatenate([order_1,order_2,order_3])
```

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

Note: We can not concatenate the arrays with different dimensions (i.e. we can not concatenate a 1D array with a 2D array)

Concatenate 2D array

We can concatenate 2D arrays either along rows (axis = 0) or columns (axis = 1), provided they have same shape

```
# create two 2D array
odd_array = np.array([[1, 3],
                      [5, 7]])

even_array = np.array([[2, 4],
                       [6, 8]])

# concatenate arrays along rows (axis = 0)
# by default concatenate() function concatenate along rows
print('concatenate row-wise:', np.concatenate([odd_array,even_array]))

# concatenate arrays along columns (axis = 1)
print('concatenate column-wise:', np.concatenate([odd_array,even_array], axis = 1))
```

```
concatenate row-wise: [[1 3]
 [5 7]
 [2 4]
 [6 8]]
concatenate column-wise: [[1 3 2 4]
 [5 7 6 8]]
```

Stacking Arrays

Stack arrays with stack()

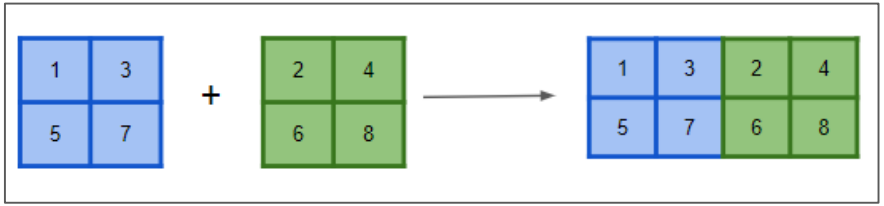
The stack() is used to join two or more arrays of same shape along the specified axis. We can create higher-dimensional arrays by stacking two or more lower dimensional arrays

```
# create two 2D arrays
odd_array = np.array([[1, 3],
                      [5, 7]])

even_array = np.array([[2, 4],
                       [6, 8]])

# by default arrays will stack along axis = 0
print('stacked array: \n',np.stack([odd_array,even_array]))

# dimension of resultant array is 3
print('dimension:',np.ndim(np.stack([odd_array,even_array])))
```

```
stacked array:
 [[[1 3]
  [5 7]]

 [[2 4]
  [6 8]]]
dimension: 3
```

Stack arrays horizontally with hstack()

This is equivalent to concatenation along the second axis (for 2D arrays, axis = 1)

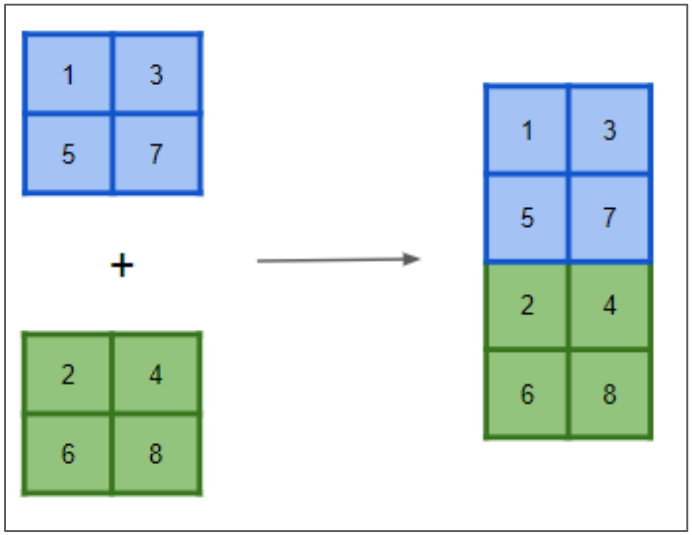



```
# create two 2D array
odd_array = np.array([[1, 3],
                      [5, 7]])

even_array = np.array([[2, 4],
                       [6, 8]])

# hstack() stacks the arrays column-wise
np.hstack([odd_array,even_array])
```

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

Stack arrays vertically with vstack()

This is equivalent to concatenation along the first axis (for 2D arrays, axis = 0)

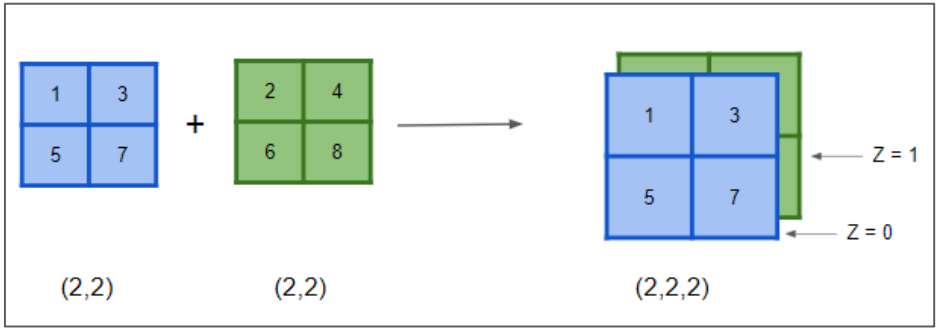



```
# create two 2D array
odd_array = np.array([[1, 3],
                      [5, 7]])

even_array = np.array([[2, 4],
                       [6, 8]])

# vstack() stacks the arrays row-wise
np.vstack([odd_array,even_array])
```

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

Stack arrays depth-wise using dstack()

The dstack() is used to stack the arrays depth-wise. We can create higher-dimensional arrays using dstack()

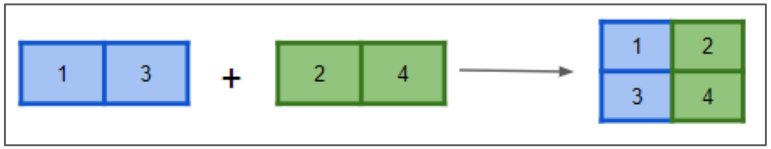



```
# create two 2D array
odd_array = np.array([[1, 3],
                      [5, 7]])

even_array = np.array([[2, 4],
                       [6, 8]])

# dstack() stacks the arrays depth-wise
print('stacked array: \n',np.dstack([odd_array,even_array]))

# dimension of stacked array
print('dimension:',np.ndim(np.dstack([odd_array,even_array])))
```

```
stacked array:
 [[[1 2]
  [3 4]]

 [[5 6]
  [7 8]]]
dimension: 3
```

column_stack() for 1D array

The column_stack() stacks the 1D array as columns into 2D array

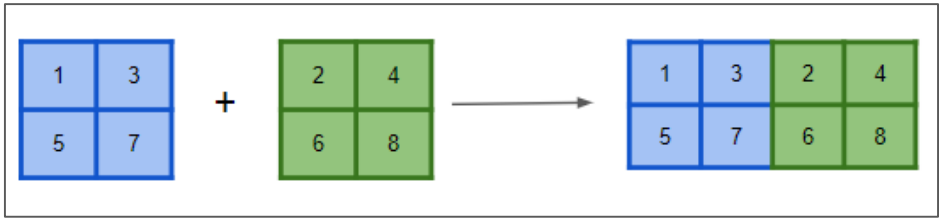



```
# create two 1D array
odd_array = np.array([1, 3])

even_array = np.array([2, 4])

# column_stack() stacks the 1D array to 2D array
np.column_stack([odd_array,even_array])
```

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

column_stack() for 2D array

For 2D arrays, column_stack() is equivalent to hstack()




```
# create two 2D array
odd_array = np.array([[1, 3],
                      [5, 7]])

even_array = np.array([[2, 4],
                       [6, 8]])

# stacking 2D array
np.column_stack([odd_array,even_array])
```

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

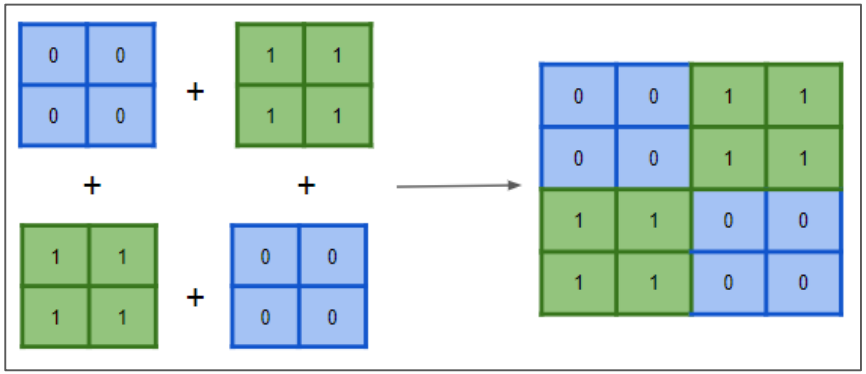
Comparison between different stacking methods

greatlearning
Learning for Life

hstack()	vstack()	dstack()	column_stack
Stacks arrays horizontally (column-wise)	Stacks arrays vertically (row-wise)	Stacks arrays depth-wise (along the 3rd axis)	Stacks 1D arrays as columns into 2D array. Stacking a 2D arrays is same as hstack()

Assemble arrays using block()

The block() is used to assemble array from a nested list of blocks.



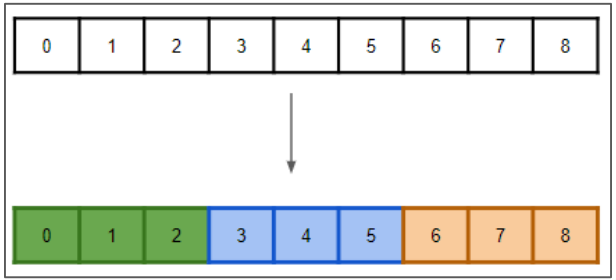

```
# use block() to create a block matrix

# create 4 block matrices
m1 = np.zeros((2,2))
m2 = np.ones((2,2))
m3 = np.ones((2,2))
m4 = np.zeros((2,2))

# use 'block()' to assemble above matrices in 2D array
np.block([[m1, m3],
          [m2, m4]])
```

```
array([[0., 0., 1., 1.],
       [0., 0., 1., 1.],
       [1., 1., 0., 0.],
       [1., 1., 0., 0.]])
```

Splitting Arrays

Split array with split()

The split() is used to split the array into multiple sub-arrays

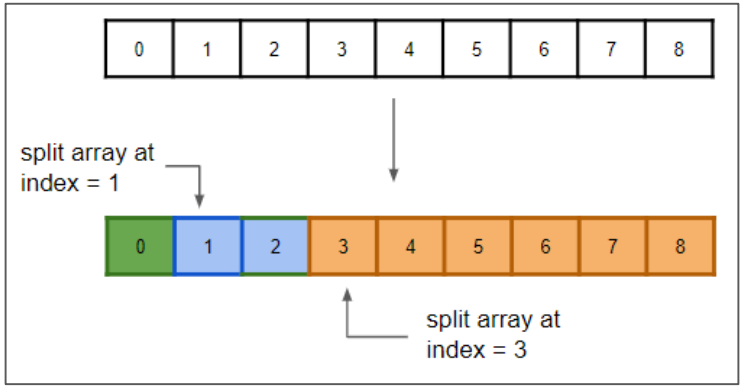



```
# consider 1D array
num_array = np.arange(9)

# split the array into 3 sub-arrays
print('3 sub-arrays of equal length:',np.split(num_array, 3))
```

```
3 sub-arrays of equal length: [array([0, 1, 2]), array([3, 4, 5]), array([6, 7, 8])]
```

Split array with split()

split() can be used to split the array at the specific indices




```
# consider 1D array
num_array = np.arange(9)

# split the array at specified indices
print('splitting of an array at specific indices:', np.split(num_array, [1,3]))
```

```
splitting of an array at specific indices: [array([0]), array([1, 2]), array([3, 4, 5, 6, 7, 8])]
```

Disadvantage of split()

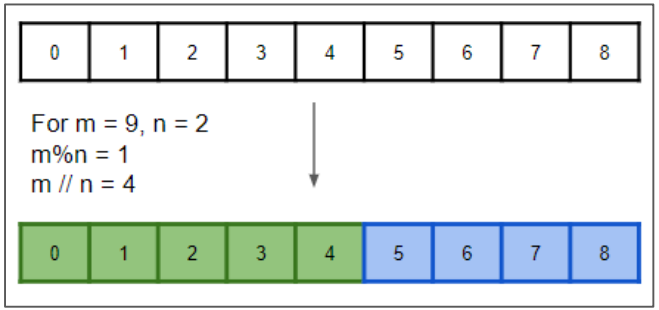
The split() does not allow an integer (N) as a number of splits, if N does not divide the array into equal length of sub-arrays

i.e. If we try to split the array of nine elements into two sub-arrays, the split() throws an error; as the nine elements can not be split into two arrays of equal length

Split with array_split()

The array_split() returns multiple sub-arrays of the passed array

For an array of length ‘m’ that should be split into ‘n’ sub-arrays, array_split() returns m%n sub-arrays of size m//n + 1 and remaining of size m//n

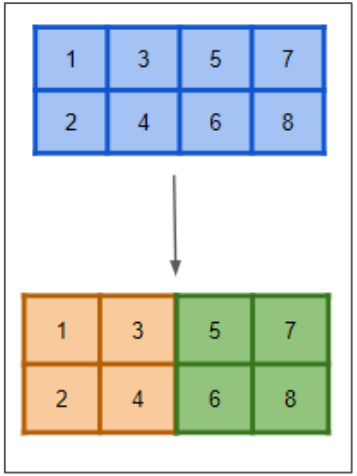



```
# consider 1D array
num_array = np.arange(9)

# split the array into 2 sub-arrays
# here, m = 9, n = 2
# 1st sub-array is of size 5 and 2nd sub-array is of size 4
print('2 sub-arrays:',np.array_split(num_array, 2))
```

```
2 sub-arrays: [array([0, 1, 2, 3, 4]), array([5, 6, 7, 8])]
```

Split array horizontally with hsplit()

The hsplit() It is used to split an array horizontally (column-wise)



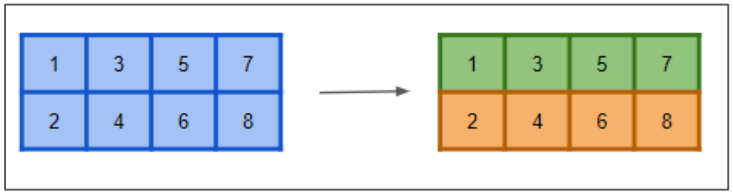

```
# create 2D array
num_array = np.array([[1, 3, 5, 7],[2, 4, 6, 8]])

# split the array into 2 sub-arrays
np.hsplit(num_array, 2)
```

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

The hsplit() creates sub-arrays with equal number of rows

Split array vertically with vsplit()

The vsplit() is used to split an array vertically (row-wise)



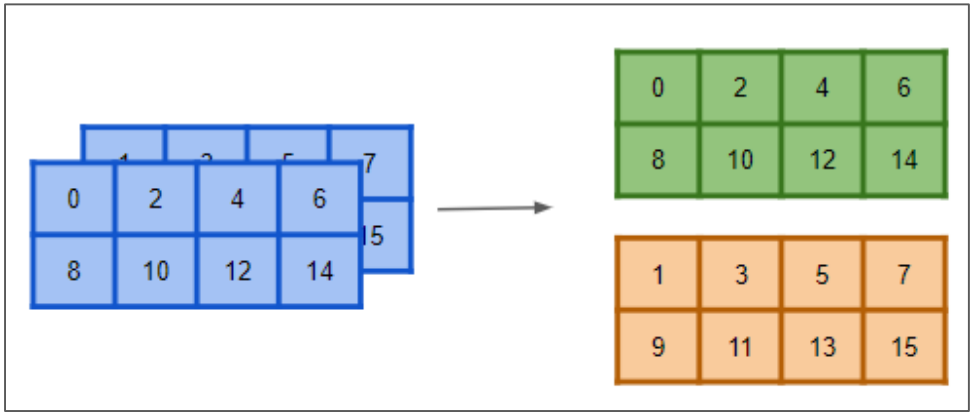

```
# create 2D array
num_array = np.array([[1, 3, 5, 7],[2, 4, 6, 8]])

# split the array into 2 sub-arrays
np.vsplit(num_array, 2)
```

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

The vsplit() creates sub-arrays with equal number of columns

Split array depth-wise with dsplit()

The dsplit() is used to split an array along the third axis (depth-wise)




```
# create 3D array
num_array = np.arange(16).reshape(2,4,2)
# split the array into 2 sub-arrays
print('splitted array', np.dsplit(num_array, 2))
```

```
splitted array [array([[[ 0],
        [ 2],
        [ 4],
        [ 6]],

       [[ 8],
        [10],
        [12],
        [14]]]), array([[[ 1],
        [ 3],
        [ 5],
        [ 7]],

       [[ 9],
        [11],
        [13],
        [15]]])]
```

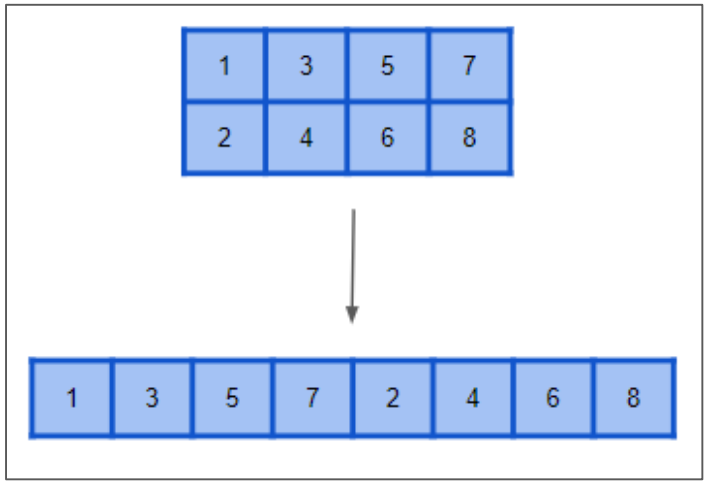
Comparison between different splitting methods

greatlearning
Learning for Life

hsplit()	vsplit()	dsplit()
Split the array horizontally. It is equivalent to split() with axis = 1	Split the array vertically. It is equivalent to split() with axis = 0	Split the array depth-wise. It is equivalent to split() with axis = 2

Flatten the array

The flatten() function collapses the original array into a single dimension




```
# create 2D array
num_array = np.array([[1, 3, 5, 7],[2, 4, 6, 8]])
print('Original array:\n', num_array)

# flatten the array into 1D
print('Flattened array:\n', np.ndarray.flatten(num_array))
```

```
Original array:
 [[1 3 5 7]
 [2 4 6 8]]
Flattened array:
 [1 3 5 7 2 4 6 8]
```

greatlearning
Learning for Life

Iterating Arrays

Iterate 1D array

The for loop is used to iterate the elements in the array

```
# create 1D array
age = np.array([13,45,22,16,14,25])

# print each of the observation using for loop
for i in age:
    print(i)
```

```
13
45
22
16
14
25
```

Iterate 2D array

```
# create 2D array
num_array = np.array([[1, 3, 5, 7],[2, 4, 6, 8]])

# add 5 to each element in the 2D array
for i in num_array:
    print(i+5)
```

```
[ 6  8 10 12]
[ 7  9 11 13]
```

Iterate 2D array with nested loop

greatlearning
Learning for Life

To print each element in the 2D array, use nested for loop

```
# create 2D array
num_array = np.array([[1, 3, 5, 7],[2, 4, 6, 8]])

# print each element in the 2D array
for i in num_array:
    for j in i:
        print(j)
```

```
1
3
5
7
2
4
6
8
```

greatlearning
Learning for Life

Thank You