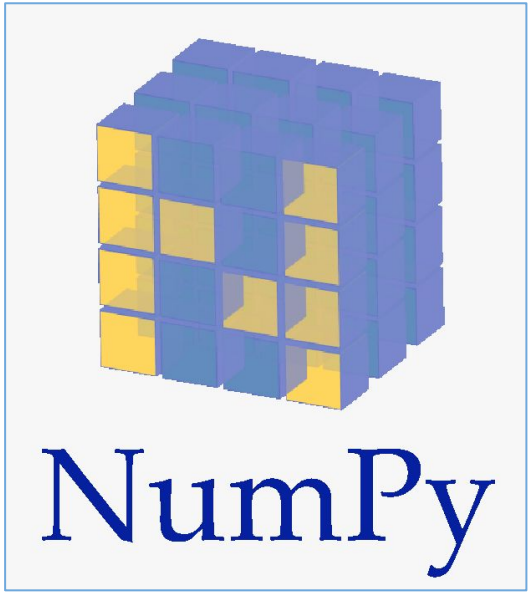
Python NumPy



What is NumPy?

What is NumPy?

- 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 (det, eig, norm, and so on) 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 NumPy?

- How to install NumPy?
  In jupyter notebook, use the command:

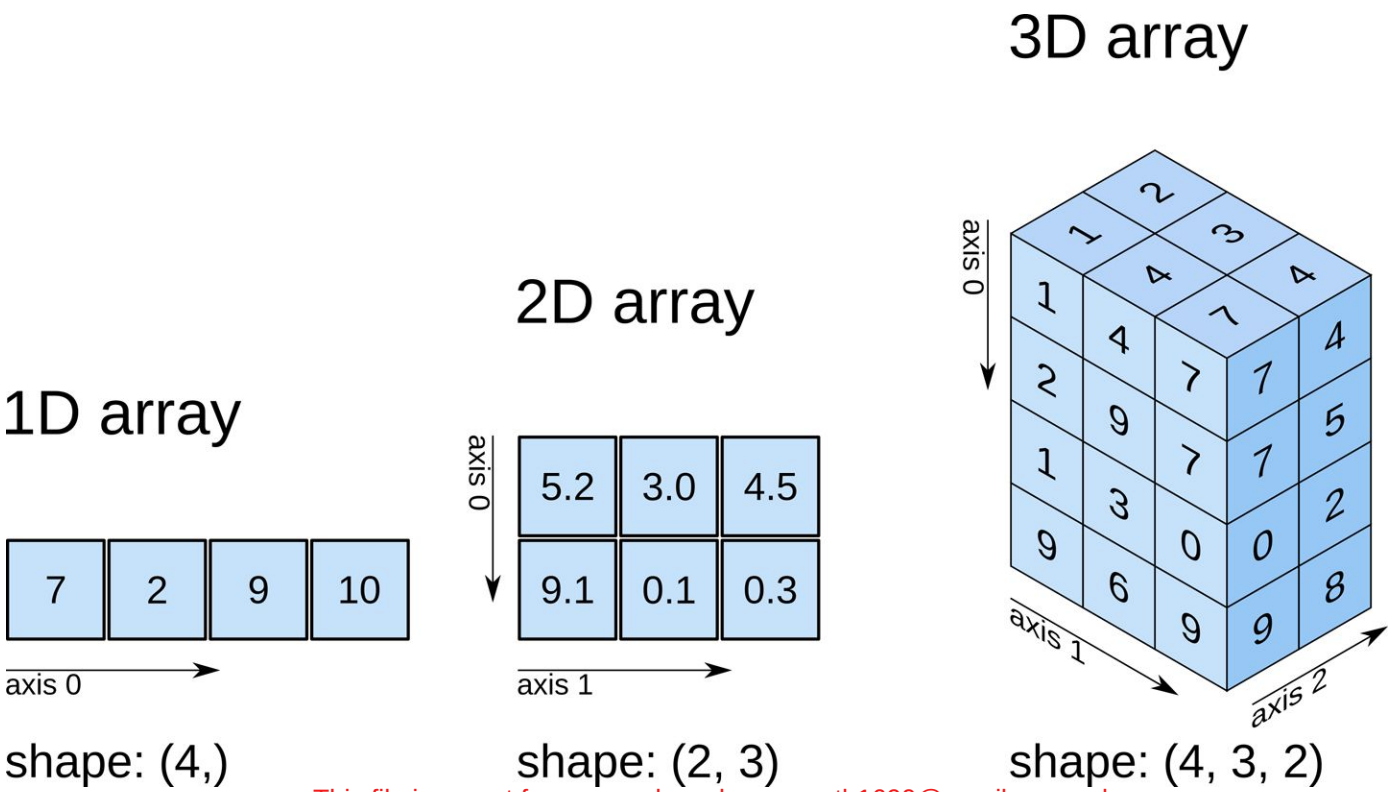
```
# installing NumPy
!pip install numpy
```

- Import the library as:

```
# importing the library
import numpy as np
```

Creating NumPy Array

1D, 2D and 3D NumPy array



Creating NumPy array using list

Use 'array()' function to create a numpy array from a list

```
# list
list1 = [1,2,3,4]

# check type of 'list1'
print('Output:', type(list1))
```

```
Output: <class 'list'>
```

create an array of 'list1'

```
# list
list1 = [1,2,3,4]

# create an array
arr1 = np.array(list1)
print('Output:')
print('Array:', arr1)

# check type of 'arr1'
type(arr1)
```

```
Output:
Array: [1 2 3 4]

numpy.ndarray
```

Creating NumPy array

- Create 1D numpy array of zeros

  Use ‘zeros()’ function to create an array of zeros

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

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

- Create 1D numpy array of ones

  Use ‘ones()’ function to create an array of ones

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

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

Creating NumPy array of random numbers

Create an array of 10 random numbers using random() function

```
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() function returns random numbers from uniform distribution over the half-open interval [0.0, 1.0). The required number of random numbers is passed through parameter 'size'. One can also use the 'rand()' function

Creating NumPy array of random numbers

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

```
In [1]: import numpy as np
        np.random.rand(5)
Out[1]: array([0.97203315, 0.60106205, 0.13302437, 0.52632467, 0.52019637])
```

- randn returns a variable (or a set of variables) from the "Standard Normal" distribution. Unlike rand which is from a uniform distribution. A standard Normal Distribution has mean 0 and SD of 1 as we know

```
In [2]: np.random.randn(5)
Out[2]: array([-0.45191007,  0.92876969,  1.02000334,  1.19583545,  0.40234378])
```

Creating NumPy array of random numbers

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

```
In [3]: np.random.randint(5,20)       # Returns one rand integer between the values 5 & 19(20 is excluded)
Out[3]: 16

In [4]: np.random.randint(20,50,5)  # Returns  5 rand integers between 20 & 49(50 is excluded)
Out[4]: array([32, 44, 30, 41, 41])
```

Creating 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
num = np.arange(start = 10, stop = 20, step = 2)
num
```

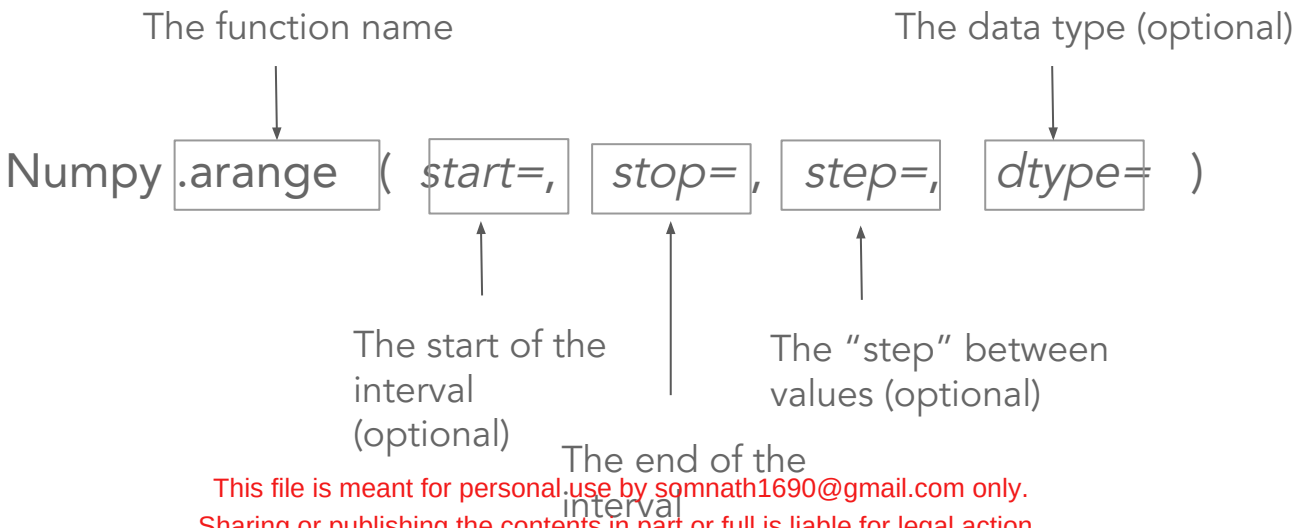
```
array([10, 12, 14, 16, 18])
```

What happened here?

- The np.arange function produced a sequence of 5 evenly spaced values from 10 to 18, stored as an ndarray object (i.e., a NumPy array)
- Having said that, what's actually going on here is a little more complicated, so to fully understand the np.arange function, we need to examine the syntax

The syntax of numpy arange()

- The syntax for NumPy arange is pretty straightforward
- Like essentially all of the NumPy functions, you call the function name and then there are a set of parameters that enable you to specify the exact behavior of the function



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

Let us explore some basic attributes of the numpy array:

- size
- itemsize
- shape
- ndim
- dtype

Attributes of array

- size: returns the number of elements in an array

```
array1 = np.array([1,4,-2,8,7.4,3,9,-8,5])
print('No. of elements:', array1.size)
```

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

- itemsize: returns length of each element of array in bytes

```
arr =  np.array([14,74,84,26,56])
print('itemsize (in bytes):',arr.itemsize)
```

```
itemsize (in bytes): 4
```

Attributes of array

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

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

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

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

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

```
Dimension: 2
```

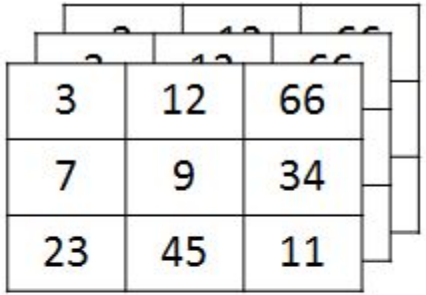
Attributes of array - ndim()

1	5	18	23

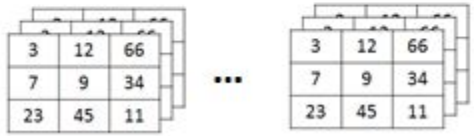
Vector (1D array)
Dimension = 1
(1 index required)

3	12	66
7	9	34
23	45	11

Matrix (2D array)
Dimension = 2
(2 indexes required)



3D array (3rd order Tensor)
Dimension = 3
(3 indexes required)



ND array
Dimension = N
(N indexes required)

Attributes of array

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

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

```
dtype: float64
```

In the example, there are two 64-bit floating-point numbers. Thus, the dtype of the array is ‘float64’

Numpy Array Method

NumPy array method

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

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]]
```

Indexing of an Array

Indexing of 1D array

Each element in the array can be accessed by passing the positional index of the element. The index for an array always starts from '0'

```
# 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
```

Indexing 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

Slicing 1D array

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]
```

Slicing 2D array

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]]
```

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 array1 and array2
array1 = np.array([4,7,5,6])
print('array1',array1)
array2 = np.array([8,9,5,2])
print('array2',array2)

# add array1 and array2 element-wise
sum = array1 + array2
print('Addition:', sum)

# subtract the array2 from array1
sub = array1 - array2
print('Subtraction:', sub)
```

```
array1 [4 7 5 6]
array2 [8 9 5 2]
Addition: [12 16 10  8]
Subtraction: [-4 -2  0  4]
```

Arithmetic operations on array

Element-wise multiplication of two 3x3 matrices:

```
m1 = np.array([[1,0,2],[4,-2,3],[0,5,1]])
m2 = np.array([[4,0,1],[2,3,0],[1,4,-3]])

# element-wise multiplication
prod = m1*m2
print('element-wise multiplication:',prod)
```

```
element-wise multiplication: [[ 4  0  2]
 [ 8 -6  0]
 [ 0 20 -3]]
```

Multiply each element in the array by 2:

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

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

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

Arithmetic operations on array

Matrix multiplication of two 3x3 matrices:

```
m1 = np.array([[1,0,2],[4,-2,3],[0,5,1]])
m2 = np.array([[4,0,1],[2,3,0],[1,4,-3]])

# dot() returns the matrix multiplication of the matrices m1 and m2
multi = m1.dot(m2)
print('matrix multiplication:',multi)
```

```
matrix multiplication: [[ 6  8 -5]
 [15  6 -5]
 [11 19 -3]]
```

Note that the dot product is calculated as
[1*4+0*2+2*1, 1*0+0*3+2*4, 1*1+0*0+2*-3],[4*4-2*2+3*1, 4*0-2*3+3*4, 4*1-2*0+3*-3],[0*4+5*2+1*1,0*0+5*3+1*4, 0*1+5*0-1*3]]

Arithmetic Functions on Array

Arithmetic functions on array

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

```
arr =  np.array([14,74,84,26,56])
print('sum of all the elements:',arr.sum())
```

```
sum of all the elements: 254
```

- min(): returns the minimum value in the array

```
arr =  np.array([14,74,84,26,56])
print('minimum:',arr.min())
```

```
minimum: 14
```

Arithmetic functions on array

power(): It is used to raise the numbers in the array to the given value

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

# raise each of the element to 2nd power (square)
print('Squared_array:',np.power(num, 2))
```

```
Squared_array: [ 1  4  9 16]
```

Concatenation

Concatenate 1D array

Two or more arrays will get joined along existing (first) axis, provided they have the same shape, except in the dimension corresponding to the axis (i.e. for 1D array, the shape can be different along the first axis)

```
# create three 1D arrays
x = np.array([1,4,5,7])
y = np.array([6,9,3])
z = np.array([2,7,5,9])

# concatenate x, y, and z array
np.concatenate([x,y,z])
```

```
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
array_1 = np.array([[1, 3],
                    [5, 7]])

array_2 = np.array([[2, 4],
                    [6, 8]])

# concatenate array_1 and array_2 along rows (axis = 0)
# by default concatenate() function concatenate along rows
print('concatenate row-wise:', np.concatenate([array_1,array_2]))

# concatenate array_1 and array_2 along columns (axis = 1)
print('concatenate column-wise:', np.concatenate([array_1,array_2], 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

Stack arrays with stack()

stack() function 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
array_1 = np.array([[1, 3],
                    [5, 7]])

array_2 = np.array([[2, 4],
                    [6, 8]])

# by default arrays will stack along axis = 0
print('stacked array: \n',np.stack([array_1,array_2]))

# dimension of resultant array is 3
print('dimension:',np.ndim(np.stack([array_1,array_2])))
```

```
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
array_1 = np.array([[1, 3],
                    [5, 7]])

array_2 = np.array([[2, 4],
                    [6, 8]])

# hstack() stacks the arrays column-wise
np.hstack([array_1,array_2])
```

```
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
array_1 = np.array([[1, 3],
                    [5, 7]])

array_2 = np.array([[2, 4],
                    [6, 8]])

# vstack() stacks the arrays row-wise
np.vstack([array_1,array_2])
```

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

Stack arrays depth-wise using dstack()

dstack() is used to stack the arrays depth-wise. It converts a single array as a column in the stacked array. We can create higher-dimensional arrays using dstack()

```
# create two 2D array
array_1 = np.array([[1, 3],
                    [5, 7]])

array_2 = np.array([[2, 4],
                    [6, 8]])

# dstack() stacks the arrays depth-wise
print('stacked array: \n',np.dstack([array_1,array_2]))

# dimension of stacked array
print('dimension:',np.ndim(np.dstack([array_1,array_2])))
```

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

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

column_stack() for 1D array

column_stack() stacks the 1D array as 2D array, thus we can use column_stack() to create the higher dimensional arrays

```
# create two 1D array
array_1 = np.array([1, 3])

array_2 = np.array([2, 4])

# column_stack() stacks the 1D array to 2D array
np.column_stack([array_1,array_2])
```

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

column_stack() for 2D array

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

```
# create two 2D array
array_1 = np.array([[1, 3],
                    [5, 7]])

array_2 = np.array([[2, 4],
                    [6, 8]])

# stacking 2D array
np.column_stack([array_1,array_2])
```

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

block()

block() function is used to assemble array from a nested list of blocks. The block matrix can be created using this function

```
# 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

Splitting array with split()

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

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

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

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

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

Disadvantage of split()

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

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

# split the array into 2 sub-arrays
print('2 sub-arrays:',np.split(arr, 2))

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

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
C:\ProgramData\Anaconda3\lib\site-packages\numpy\lib\shape_base.py in split(ary, indices_or_sections, axis)
    842     try:
--> 843         len(indices_or_sections)
    844     except TypeError:

TypeError: object of type 'int' has no len()

During handling of the above exception, another exception occurred:

ValueError                                Traceback (most recent call last)
<ipython-input-134-b37db852de28> in <module>()
      3
      4 # split the array into 2 sub-arrays
----> 5 print('2 sub-arrays:',np.split(arr, 2))
      6
      7 # split() does not allow an integer(N) as a number of splits, if the integer does not divide the array into equal lengt
h of

C:\ProgramData\Anaconda3\lib\site-packages\numpy\lib\shape_base.py in split(ary, indices_or_sections, axis)
    847         if N % sections:
    848             raise ValueError(
--> 849                 'array split does not result in an equal division')
    850     res = array_split(ary, indices_or_sections, axis)
    851     return res
ValueError: array split does not result in an equal division
```

Splitting with array_split()

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
arr = 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(arr, 2))
```

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

Split array horizontally with hsplit()

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

```
# create 2D array
array_1 = np.array([[1, 3, 5, 7],[2, 4, 6, 8]])

# split the array into 2 sub-arrays
np.hsplit(array_1, 2)
```

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

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

Split array vertically with vsplit()

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

```
# create 2D array
array_1 = np.array([[1, 3, 5, 7],[2, 4, 6, 8]])

# split the array into 2 sub-arrays
np.vsplit(array_1, 2)
```

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

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

Split array depth-wise with dsplit()

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

```
# create 3D array
array_1 = np.arange(16).reshape(2,4,2)
print(array_1)

# split the array into 2 sub-arrays
print('splitted array', np.dsplit(array_1, 2))
```

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

 [[ 8  9]
  [10 11]
  [12 13]
  [14 15]]]
splitted array [array([[[ 0],
        [ 2],
        [ 4],
        [ 6]],

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

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

Iterating Arrays

Iterating 1D array

'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
```

Iterating 2D array

```
# create 2D array
array_1 = np.array([[1, 3, 5, 7],[2, 4, 6, 8]])

# add 5 to each element in the 2D array
for i in array_1:
    print(i+5)
```

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

Iterating 2D array with nested loop

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

```
# create 2D array
array_1 = np.array([[1, 3, 5, 7],[2, 4, 6, 8]])

# print each element in the 2D array
for i in array_1:
    for j in i:
        print(j)
```

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

Iterating arrays with nditer()

NumPy provides an iterator object 'nditer()' to iterate the elements of an array

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

# replace values in the array with corresponding squares
# 'op_flags=['readwrite']' operand will read and change the values
# value[...] stores modified values
for value in np.nditer(num, op_flags=['readwrite']):
    value[...] = value**2

print('modfied array:', num)
```

```
modfied array: [ 1 16  4 36  9 25]
```

Thank You