Numpy

Introduction

```
import numpy as np

# creating numpy array ( which is same as python list )
array = np.array([1,2,3,4,5])

# Accessing multi-dimension list in easier way, note your array must consistent in terms of each sequence.
multi = np.array([[1,2,3],[4,5,6],[7,8,9]])
# this type of accessing list doesn't have
print(multi[2,1])


# ARRAY  ATTRIBUTES  [SHAPE,DIMENSIONS, SIZE, DATATYPE]
get the shape of the array aka list
multi_shape = np.array([
    [1,2,3],
    [4,5,6],
    [7,8,9],
    [10,11,12]
    ])
print(multi_shape.shape)  # (4/3) 4 sub array each has 3 elements

mult_of = np.array([
    [[1,2,3],
    [4,5,6],
    [7,8,9]],
    [['a','b', 'c'],
     ['d','e', 'f'],
     ['f','h', 'i']]
    ],
    )
print(mult_of.shape)  # (2, 3 /3) 2 sub array has  3 sub array each has 3 elements

# see the dimension of array and how deep it goes
print(multi_shape.ndim) # 2  it list inside another list
print(mult_of.ndim) # 3 it's list inside list inside another list

# see the size of array = comes from multiplication of shapes
print(multi_shape.size) # 12 (4x3)
print(mult_of.size)     # 18 (3x3x2)




# DATA TYPE ATTRIBUTES  AND DATATYPES HOW HOW THEY WORK

# access the datatype of array ( why do we care datatype since we using python  cause pandas is written in c and is the reason
# why is so fast and so optimized   and it's statically typed )
print(multi_shape.dtype) # int64
print(mult_of.dtype)  # you will see result of <U21 that means it's string with
# less then 21 characters Since NumPy arrays require a consistent data type for all elements (why ? because it uses c ),
# NumPy converts all elements to common data type that can accommodate both integers and strings. In this case, it choose a Unicode
#  string data type with a maximum length of 21  characters to accommodate the longest string in your array.

# to check it
print(type(mult_of[0,0,0])) # <class 'numpy.str_'> so if the array doesn't have
# any string it will stay as int

# Specify dtype during array creation to prevent element conversion to strings
# change_of = np.array(
#     [[1,2,3],
#     [4,5,6],
#     [7,'hello',9]], dtype=np.int64)  # this won't work cause hello can't be changed into int.
# but what you can is use string number like "4" "5" and change it into int64

change_of = np.array(
    [[1,2,3],
    [4,5,'10'],
    [7,'4',9]], dtype=np.int64)

print(change_of.dtype) # int64 but if you don't type cast it will keep string
```

```
 #  you can type cast int float
change_of = np.array(
    [[1,2,3],
    [4,5,'10'],
    [7,'4',9]], dtype=np.float64)  #cast int float
print(change_of[1,1].dtype) #float64, if you don't type cast it will keep string

# if you mix something can't be changed into int nor string it will be casted into object for example mix the list of dictionary

dic = {1: 'name'}
mix_mult = np.array(
    [[1,2,dic],
    [4,5,'10'],
    [7,'4','hello']])
print(mix_mult.dtype) # object
# print(mix_mult[1,1].dtype)  # 'int' object has no attribute 'dtype'so when you use object each element will have it's original datatype
# for example use string and mix into array (which has object like dict) and that string will be  str datatype ( won't be changed )
# same as the int will have it's int
print(type(mix_mult[1,1])) # <class 'int'>
print(type(mix_mult[2,2]))  # <class 'str'>

# you can specify number to be string and change them
int_as_str = np.array(
    [[1,2,3],
    [4,5,'10'],
    [7,'4',9]], dtype='U2')

print(int_as_str.dtype)   # <U2
print(type(int_as_str[1,1]))  # <class 'numpy.str_'>
```

Filtering Array

```
# FILLING ARRAY   = creating array with default values

# numpy.full(shape, fill_value, dtype=None, order='C'): This function creates an array with the specified shape and fills it
# with a specified fill_value.

fill_array = np.full((2,3,4), 9)
print(fill_array)  #  you will get 2 list consisting 3 lists with 4 elements each

# you can use what ever  shape you like
fill_array = np.full((10,10,10,10), 10)
# print(fill_array)


# np.zeros(shape, dtype=float, order='C')
# This function creates an array filled with zeros. You specify the shape of the array and an optional data type. For example
zeros_array = np.zeros((3, 3))  # Creates a 3x3 array filled with zeros.


# np.ones(shape, dtype=float, order='C')
# This function creates an array filled with ones. It's similar to numpy.zeros but fills the array with ones instead of zeros.
ones_array = np.ones((3, 3))  # Creates a 3x3 array filled with ones.



# np.empty(shape, dtype=float, order='C')
#This function creates an array without initializing its values. it just allocates the space, The content of the array is undefined or
# whatever data was previously stored in that region of memory.. It's faster than numpy.zeros or numpy.ones because it doesn't
# set any values explicitly.
empty_array = np.empty((3, 3))  # Creates a 3x3 array filled with undefined values.
print(empty_array)  # The numbers you're seeing in the array are not meaningful values but rather arbitrary or uninitialized data that
# happened to be in the memory locations allocated for the NumPy array.


# np.arange(start, stop, step, dtype=None)
# This function creates an array with regularly spaced values within a specified range. You specify the start, stop, and step values.

range_array = np.arange(0,100, 5)
```

```
print(range_array) # [ 0  5 10 15 20 25 30 35 40..... 75 80 85 90 95] if you wanna have upto 100 chage the stop into above 100


# numpy.linspace(start, stop, num=50, endpoint=True, retstep=False, dtype=None)
# This function creates an array with evenly spaced values over a specified range. You specify the start and stop values and
# optionally the number of elements in the array.

# the number is how many numbers do we generate from this range like 0 to 1000 generate 2 number [0,1000] if generate 3 it will
# [0,500, 1000]

linsapce_array = np.linspace(0,50, 6) # [ 0. 10. 20. 30. 40. 50.] you have remember  it starts D start
print(linsapce_array)

linsapce_array = np.linspace(0,1000, 1001) # [   0.    1.    2. ...  998.  999. 1000.]
print(linsapce_array)
```

NAN & INF

```
# Nan not a number & inf infinity   (they don't have so much use )
# ------------------------------- DOCS

# NaN is used to represent the result of undefined or unpresentable mathematical operations. For example, dividing zero by zero or
# taking the square root of a negative number results in NaN.


# Inf represents positive infinity, and it is used to indicate that a result exceeds the representable range of the data type.
# For example, dividing a positive number by zero results in positive infinity, while dividing a negative number by zero results in
# negative infinity


# why Nan
# In data analysis and scientific computing, datasets may have gaps or missing values. Using NaN to denote missing data allows you to
# perform computations on the available data while easily identifying and handling missing values.

# why inf
# When performing calculations, especially in scientific and engineering applications, it's possible to encounter results that are too
# large to be represented within the bounds of the chosen data type. In such cases, the result is set to positive infinity instead of
# throwing error

#  you can create positive and negative infinity using np.inf and -np.inf, respectively.

# ---------------------------------------------------

# to check if item is Nan and inf

print(np.isnan(np.nan))
print(np.isinf(np.inf))

print('----------------------------------')
# real world example
result = np.sqrt(-4)  # RuntimeWarning: invalid value encountered in sqrt but you can still show the students
# print(np.isnan(result)) # True , also  you will get Runtime warning not error

numbers = np.array([23,0,32])
# print(np.isinf(numbers/ 0))  # [ True False  True] also you will get  RuntimeWarning: divide by zero encountered in divide

print(result) # nan ,     you will still see the runtime warning
print(numbers/0) #  [inf nan inf] ,  you will still see the runtime warning
```

DID PERFORMING MATHEMATICAL OPERATION IN NUMPY

```
# PERFORMING MATHEMATICAL OPERATION IN NUMPY

# we going to perform mathematical operations to compare the difference of python list and numpy
```

```
l1 = [1,2,3,4,5]
l2 = [6,7,8,9,0]

a1= np.array(l1)
a2= np.array(l2)


# the first Operation  Scaler
# Suppose you have a scenario where you want to multiply each element in a list by 5. Traditionally, without NumPy, you might approach
# this task by looping through the list or using a map function. Here's how you might do it in a typical Python list:

# solution 1
result = []
for num in l1:
    result.append(num * 5)

# solution 2
result = [num * 5 for num in l1]

# solution 3
result = list(map(lambda x: x * 5, l1))


# now let's see how using NumPy, which simplifies the operation significantly:

print(a1 * 5)

# if you  try to do this in python list
print(l1 * 5)  # [1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5]


# it workrs  in all operation

# print(l1 + 5) # you will get type error TypeError: can only concatenate list (not "int") to list
print(a1 + 5) # [ 6  7  8  9 10]

# print(l1 / 5) # you will get type error TypeError: unsupported operand type(s) for /: 'list' and 'int'
print(a1 / 5) # [0.2 0.4 0.6 0.8 1. ]

# print(l1 - 5) # you will get type error TypeError: unsupported operand type(s) for -: 'list' and 'int'
print(a1 - 5) # [-4 -3 -2 -1  0]


# The Second operation is adding list

# Let's consider a scenario where you have two lists, list1 and list2, and you want to add their corresponding elements together to
# create a new list or array

# . Here's how you can do this using both Python lists:

result_list = [x + y for x, y in zip(l1, l2)]


# but  this is how you can do it numpy
print(a1 + a2 ) #  [ 7  9 11 13  5]

# if you try this in python list you will get this

print(l1 + l2 ) # [1, 2, 3, 4, 5, 6, 7, 8, 9, 0]



# it workrs  in all operation


print(a1- a2)  # [-5 -5 -5 -5  5]
# print(l1 - l2)  TypeError: unsupported operand type(s) for -: 'list' and 'list'


print(a1/a2 ) #  [0.16666667 0.28571429 0.375   0.44444444   inf] you will get one infinity cause you have 0 in the second array
# print(l1/l2) # TypeError: unsupported operand type(s) for /: 'list' and 'list'

print(a1* a2) # [ 6 14 24 36  0]
# print(l1 * l2)  # TypeError: can't multiply sequence by non-int of type 'list'



# you can add two numpy arrays if the have same shape but different dimension ( it doesn't matter how deep it goes you just need the two
# array must have the same shape aka elements in the last array(inner array ))
```

```
print('------------------')
b1= np.array([1,2,3,4])
b2 = np.array([[1,2,3,5]])
print(b1 + b2) #  [[2 4 6 9]]

b1= np.array([1,2,3,4])
b2 = np.array([[1,2,3,5],[32,4,3,1]])
print(b1 + b2) # [[ 2  4  6  9]     [33  6  6  5]]
```

MATHEMATICAL OPERATIONS NUMPY PROVIDES

```
# # Mathematical and Statistical Methods:
print('------------------------')

pit = np.array([
    [1, 2, 3, 4, 5, 6],   # split one            |   these two can be one
    [7, 8, 9, 10, 11, 12]])


#     mean(): Compute the mean (average) of array elements.
print(np.mean(pit))  # Output: 6.5

#     median(): Compute the median  of array elements.
print(np.median(pit))  # Output: 6.5


#     sum(): Compute the sum of array elements.
print(np.sum(pit)) # Output: 78

#     min(): Find the minimum value in the array.
print(np.min(pit)) # Output: 1

#     max(): Find the maximum value in the array.
print(np.max(pit)) # Output: 12

#     std(): Compute the standard deviation of array elements.
print(np.std(pit))  # Output: 3.452052529534663

#     var(): Compute the variance of array elements.
print(np.var(pit))  # Output: 11.916666666666666



# Numpy provide mathematical operations
# These  NumPy's mathematical functions are highly optimized for performance and can handle large arrays efficiently, making it a
# valuable tool for numerical computing and scientific applications.


    # Power and Square Root Functions:
    #     np.power(): Raise elements in an array to a specified power.
    #     np.sqrt(): Compute the square root of elements in an array.


two_dim_array = np.array([[1, 4, 9], [16, 25, 36], [49, 64, 81]])
print(np.sqrt(two_dim_array))  # [  [1. 2. 3.]   [4. 5. 6.]   [7. 8. 9.]   ] array of squareroot with same dimensions
print(np.power(two_dim_array,2))#[  [1 16  81]  [ 256 625 1296]   [2401 4096 6561] ] array of power 2 with same dimensions


    # Rounding and Absolute Value Functions:
    #     np.round(): Round elements in an array to the nearest integer or a specified number of decimals.
    #     np.floor(): Round elements in an array down to the nearest integer.
    #     np.ceil(): Round elements in an array up to the nearest integer.
```

```
    #     np.abs(): Compute the absolute value of elements in an array.



    # Statistical Functions:
    #     np.mean(): Compute the mean (average) of elements in an array.
    #     np.median(): Compute the median of elements in an array.
    #     np.std(): Compute the standard deviation of elements in an array.
    #     np.var(): Compute the variance of elements in an array.

    # Trigonometric Functions:
    #     np.sin(): Compute the sine of elements in an array.
    #     np.cos(): Compute the cosine of elements in an array.
    #     np.tan(): Compute the tangent of elements in an array.
    #     np.arcsin(): Compute the arcsine (inverse sine) of elements in an array.
    #     np.arccos(): Compute the arccosine (inverse cosine) of elements in an array.
    #     np.arctan(): Compute the arctangent (inverse tangent) of elements in an array.

    # Exponential and Logarithmic Functions:
    #     np.exp(): Compute the exponential of elements in an array.
    #     np.log(): Compute the natural logarithm of elements in an array.
    #     np.log10(): Compute the base-10 logarithm of elements in an array.
    #     np.log2(): Compute the base-2 logarithm of elements in an array.



    # Trigonometric and Hyperbolic Functions:
    #     np.sinh(): Compute the hyperbolic sine of elements in an array.
    #     np.cosh(): Compute the hyperbolic cosine of elements in an array.
    #     np.tanh(): Compute the hyperbolic tangent of elements in an array.

    # Special Functions:
    #     np.gamma(): Compute the gamma function for elements in an array.
    #     np.erf(): Compute the error function for elements in an array.
    #     np.isnan(): Check for NaN (Not-a-Number) values in an array.

    # Statistical Functions:
    #     np.mean(): Compute the mean (average) of elements in an array.
    #     np.median(): Compute the median of elements in an array.
    #     np.std(): Compute the standard deviation of elements in an array.
    #     np.var(): Compute the variance of elements in an array.

#     # Set Operations:
#     #     unique(): Find unique elements in the array.
#     #     intersect1d(): Find the intersection of two arrays.
#     #     union1d(): Find the union of two arrays.
#     #     setdiff1d(): Find the set difference between two arrays.
```

ARRAY METHODS

```
    # ARRAY METHODS

# NumPy provides a wide range of methods for performing various operations on arrays
# INSERTING , DELETING , UPDATING , ARRANGING

# One of the key advantages of NumPy and pandas, and many other Python libraries for data manipulation and numerical computing,
# is that they typically return new objects or copies of data when you perform operations on existing objects, rather than modifying
# the original data in place. This behavior is known as "immutability


# modifying
# append
arr = np.array([1, 2, 3,4, 5, 6])

print(np.append(arr,[6,7,8]))  # it won't change the default  array
```

```
# insertion   use insert(array, position, value, axis)
print(np.insert(arr, 2,100)) #  it won't change the default  array
print(np.insert(arr,6, [7,8,9] )) #  it won't change the default  array


# deletion  delete(arr, position_to_delete, axis)
print(np.delete(arr, 0))

# deleting array with multi dimensional
multi = np.array([
    [1,2,3],
    [4,5,6],
    [7,8,9]
    ])

# if you don't specify axis it will fltten the array and delete that index
print(np.delete(multi, 7)) # [1 2 3 4 5 6 7 9] deleted the 8 in index of 7

# in 2 dimension array you have two axis (0,10 when you use axis=0, it means you're specifying rows. Operations like deletion, slicing,
# and calculations will be performed along the rows of the array.

# When you use axis=1, it means you're specifying columns. Operations will be performed along the columns of the array.

# but if you do specify axis it won't flatten and won't go inside
# print(np.delete(multi, 7)) # this won't work since you have list with three list and each has three items (so 7 is out of index) you
#  only have 0,1,2(total 3 )
print(np.delete(multi, 1,0)) #     This line deletes the element at index 1 along axis 0 (rows). It removes the entire second row in
#  the multi array.

print(np.delete(multi, 0, 1)) # This line deletes the element at index 1 along axis 1 (columns). It removes the entire second column
# in the multi array.




# reshape(self,): Change the shape of the array.

#  note When you reshape array, the total number of elements in the original array must match the total number of elements in the
# desired shape.
# the following array contains 24 elements so you shape must be 24 after multiplied all..
structure_array = np.array([
    [
        [1, 2, 3, 4],
        [5, 6, 7, 8],
        [9, 10, 11, 12]
    ],
    [
        [13, 14, 15, 16],
        [17, 18, 19, 20],
        [21, 22, 23, 24]
    ]
])
print(structure_array.shape)
print(np.reshape(structure_array, (2,12)))
# output [[ 1  2  3  4  5  6  7  8  9 10 11 12]
#  [13 14 15 16 17 18 19 20 21 22 23 24]]


# if you don't specify two dimensions and you type one dimension it will put all the elements in 1D as the per example below
print(np.reshape(structure_array, (24,))) # [ 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24]

# if you do something like (30,1) that is two dimensions which is array containing 30 elements which each has only 1 element
print(np.reshape(structure_array, (24,1)))
# [[ 1] [ 2][ 3][ 4][ 5][ 6][ 7][ 8][ 9][10][11][12][13][14][15][16][17] [18] [19][20][21][22][23][24]]


# shapes you can make from that (12x2)  (4X3x2)  (2x6X2) (2x2x3x2)
print('----------------')
print(np.reshape(structure_array, (2,2,3,2)))
print(structure_array.reshape(1,1,1,1,24))


# resize  ----- > resize is like the shape but has little bit differences
# one :
    # reshape returns a new array with the specified shape but leaves the original array unchanged.
    # resize modifies the original array in place and doesn't return a new array.
# two :
    #  reshape requires that the requested shape is compatible with the size of the original array. The total number of elements in the
#  original and new shapes must be the same.
```

```
    # resize allows you to change the size of the array, and it will fill or truncate elements as needed to match the new shape and it
# fills with zeros only when you expand the size of the array



# you can't use inside print resize cause it returns None
structure_array.resize(5,10)
print(structure_array)

structure_array.resize(5,2)
print(structure_array)



# #     flatten(): Flatten the array into a 1D array.
flatten_array = np.array([[[ 1, 2 , 3, 4,  5,  6,  7,  8,  9, 10],
 [11, 12, 13, 14, 15, 16, 17, 18, 19, 20]]])
print(flatten_array.flatten())  # [ 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20]


# ravel
print(flatten_array.ravel())  # [ 1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20]

#  so what is the difference b/w flatten and ravel ?
# flatten always returns a copy.
# ravel returns a view of the original array whenever possible but if you modify the array returned by ravel, it may modify the
# entries in the original array. If you modify the entries in an array returned from flatten this will never happen. ravel will often be
#  faster since no memory is copied, but you have to be more careful about modifying the array it returns.

# let's see

array = np.array([[1,2,3], [4,5,6]])
var1 = array.flatten()
var1[4] = 3432
print(var1) # [   1    2    3    4 3432    6]
print(array)
# [[1 2 3]       as you can see it didn't change
#  [4 5 6]


var = array.ravel()
var[4] = 3432
print(var) # [   1    2    3    4 3432    6]
print(array)
# [[1 2 3]       as you can see it did change
#  [3432 5 6]

# so to get fully understand ravel doesn't return copy of the original array , and it doesn't return new array which is flattened  it
#  returns the same array (the original ) but with flatten view on it , so you just have the original array but with flattened view so
# any changes you make it will change the original array

# ravel is often more memory-efficient and potentially faster than flatten  it doesn't create a new copy of the data in memory.
# This can save memory and improve performance, especially when working with large arrays.


#     transpose(): Transpose the array (swap rows and columns).
tranpose = np.array([[1,2,3,], [4,5,6],[7,8,9]])
print(tranpose.transpose())

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

# you can also you T and swap axes to achieve the same goal
print(tranpose.T)
print(tranpose.swapaxes(0,1))

# difference b/w swap axes and transpose
# Transpose: The transpose operation swaps all the axes of a multi-dimensional array
# Swapaxes: The swapaxes operation specifically swaps only the two axes that you specify, while keeping the other dimensions in their
#  original order. It allows you to exchange the positions of two specific dimensions while leaving the rest unchanged.


print("----------------------------------------------------------------------")
print("----------------------------------------------------------------------")
print("----------------------------------------------------------------------")
# Create a sample 4-dimensional array (2, 3, 2, 5)
original_arr = np.array([[[[ 1,  2,  3,  4,  5],
                  [ 6,  7,  8,  9, 10]],
```

```
                 [[11, 12, 13, 14, 15],
                  [16, 17, 18, 19, 20]],

                 [[21, 22, 23, 24, 25],
                  [26, 27, 28, 29, 30]]],

                [[[31, 32, 33, 34, 35],
                  [36, 37, 38, 39, 40]],

                 [[41, 42, 43, 44, 45],
                  [46, 47, 48, 49, 50]],

                 [[51, 52, 53, 54, 55],
                  [56, 57, 58, 59, 60]]]])

# Transpose the first and second dimensions (axes 1 and 2)
transposed_arr = original_arr.transpose(0, 2, 1, 3)
print(transposed_arr)
# Swap axes 1 and 2
print('----')
swapped_arr = original_arr.swapaxes(1, 2)
print(swapped_arr)



#     # Array Comparison Methods:
#     #     all(): Check if all elements in the array evaluate to True.
#     #     any(): Check if any elements in the array evaluate to True.


#     # Indexing and Slicing Methods:
#     #     item(): Get an individual element from the array.
#     #     slice(): Create a slice of the array.
#     #     take(): Return elements at specified indices.
```

Concatenating, Stacking , splitting

```
#   Concatenating, Stacking , splitting

# concatenate joins two or more arrays along an existing axis. It can concatenate arrays vertically or horizontally.
# Syntax: np.concatenate((array1, array2, ...), axis=0)
arr1 = np.array([[1, 2, 3],[4, 5, 6]])
arr2 = np.array([[7,8,9],[10,11,12]])

concatenated_array = np.concatenate((arr1, arr2), axis=0)  # Concatenate horizontally'
print(concatenated_array)  # the two arrays will be joined as one array

# [[ 1  2  3]
#  [ 4  5  6]
#  [ 7  8  9]               joined the two arrays horizontally
#  [10 11 12]]

concatenated_array = np.concatenate((arr1, arr2), axis=1)  # Concatenate vertically'
print(concatenated_array) # still it will join the two array but now horizontally

# [[ 1  2  3  7  8  9]          joined the two array vertically
#  [ 4  5  6 10 11 12]]


# stack = makes new dimension and stacks (it means it takes the first array and then the second array and makes upper [] for them and
# that will makes new dimension)
# Syntax: np.stack(arrays, axis=0)
stacked_array = np.stack((arr1, arr2), axis=0)  # Stack vertically
print(stacked_array)

# [[[ 1  2  3]
#   [ 4  5  6]]
                      # stacked the two arrays on top of each other and made them go inside new upper [] in horizontally
#  [[ 7  8  9]
#   [10 11 12]]]
```

```
stacked_array = np.stack((arr1, arr2), axis=1)  # Stack vertically
print(stacked_array)

# [[[ 1  2  3]
#   [ 7  8  9]]
                        # stacked the two arrays on top of each other and made them go inside new upper [] in vertically
#  [[ 4  5  6]
#   [10 11 12]]]



# np.split divides an array into multiple sub-arrays along a specified axis.
# Syntax: np.split(array, indices_or_sections, axis=0)
print('------------------------')

split = np.array([
    [1, 2, 3, 4, 5, 6],   # split one            |   these two can be one
    [7, 8, 9, 10, 11, 12],  # split two          |
    [13, 14, 15, 16, 17, 18], # split three              |
    [19, 20, 21, 22, 23, 24]   # split four              |  these two can one
])

sub_arrays = np.split(split, 4)   # four splits each one group
sub_arrays = np.split(split, 2)   # two splits each one contains two groups ( the above two and the two at the bottom)
print(sub_arrays)


print(np.split(split, 6, axis=1)) # array1 will be 1,7,13,19 that vertical array 2 will be the next vertical 2,8,14,20
```

Random & Save

```
# Randoms

# Generate Random Number


print(np.random.randint(100))  # Generate a random integer from 0 to 100:

# Generate Random Float
print(np.random.rand())  # Generate a random float from 0 to 1:

# The randint() method takes a size parameter where you can specify the shape of an array.

# Generate Random Array
print(np.random.randint(100, size=(5)))  # Generate a 1-D array containing 5 random integers from 0 to 100:

print(np.random.randint(90,100, size=(3, 5))) # Generate a 2-D array with 3 rows, each row containing 5 random integers from 90 to 100:

print(np.random.rand(3, 5)) # Generate a 2-D array with 3 rows, each row containing 5 random numbers:




# Generate Random Number From Array

# The choice() method allows you to generate a random value based on an array of values.
# The choice() method takes an array as a parameter and randomly returns one of the values.

print(np.random.choice([3, 5, 7, 9]))   # Return one of the values in an array:

print(np.random.choice([3, 5, 7, 9], size=(3, 5))) # Generate a 2-D array that consists of the values in the array parameter (3, 5, 7, 9):




# save and load numpy
```

```
# BINARY
# Create a NumPy arra
arr = np.array([1, 2, 3, 4, 5])

# Save the array to a file
# np.save('my_array.npy', arr)


# Load the array from the file
loaded_arr = np.load('my_array.npy')

print(loaded_arr)




# TXT CSV FILE
data = np.array([[1, 2, 3],
                 [4, 5, 6],
                 [7, 8, 9]])

# Save as CSV (Comma-Separated Values)
# np.savetxt('my_data.csv', data, delimiter=',')   # ',' Will be separated


# lod CSV
data_csv = np.genfromtxt('my_data.csv', delimiter=',')
print(data_csv)


# PLAIN TEXT

data = np.array([[1, 2, 3],
                 [4, 5, 6],
                 [7, 8, 9]])

# Save as plain text
np.savetxt('my_data.txt', data, delimiter=' ', fmt='%s')


# load the text
data_txt = np.loadtxt('my_data.txt', delimiter=' ', dtype=int)

print("Data from Plain Text:")
print(data_txt)




# . It is useful when you want to exchange two specific dimensions of an array.
#     #     Creation Methods:
#     #     np.array(): Create an array from a Python list or tuple.
#     #     np.zeros(): Create an array filled with zeros.
#     #     np.ones(): Create an array filled with ones.
#     #     np.empty(): Create an empty array (uninitialized values).
#     #     np.full(): Create an array filled with a specified value.
#     #     np.arange(): Create an array with regularly spaced values.
#     #     np.linspace(): Create an array with evenly spaced values within a range.

#     # Array Attributes:
#     #     shape: Get the dimensions of the array (e.g., (3, 4) for a 2D array).
#     #     dtype: Get the data type of the elements in the array (e.g., int64, float32).
#     #     size: Get the total number of elements in the array.
#     #     ndim: Get the number of dimensions (axes) in the array.
#     #     itemsize: Get the size (in bytes) of each array element.
#     #     nbytes: Get the total memory usage of the array in bytes.
```