# numpy-assignment

August 31, 2024

1. Create a NumPy array 'arr' of integers from 0 to 5 and print its data type:

[1]:
```
# Answer

import numpy as np
arr = np.array([0, 1, 2, 3, 4, 5])
print(arr.dtype)
```

```
int32
```

2. Given a NumPy array 'arr', check if its data type is float64:

```
arr = np.array([1.5, 2.6, 3.7])
```

[2]:
```
# Answer

arr = np.array([1.5, 2.6, 3.7])
if arr.dtype == np.float64:
    print("Data type is float64")
else:
    print("Data type is not float64")
```

```
Data type is float64
```

3. Create a NumPy array 'arr' with a data type of complex128 containing three complex numbers:

[3]:
```
# Answer

arr = np.array([1+2j, 3+4j, 5+6j], dtype=np.complex128)
print(arr)
```

```
[1.+2.j 3.+4.j 5.+6.j]
```

4. Convert an existing NumPy array 'arr' of integers to float32 data type:

[4]:
```
# Answer

arr = np.array([1, 2, 3, 4, 5])
arr_float32 = arr.astype(np.float32)
print(arr_float32)
```

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

5. Given a NumPy array 'arr' with float64 data type, convert it to float32 to reduce decimal precision:

[5]:
```
# Answer

arr = np.array([1.123456789, 2.123456789, 3.123456789], dtype=np.float64)
arr_float32 = arr.astype(np.float32)
print(arr_float32)
```

```
[1.1234568 2.1234567 3.1234567]
```

6. Write a function array_attributes that takes a NumPy array as input and returns its shape, size, and data type:

[11]:
```
# Answer

import numpy as np

def array_attributes(arr):

    return arr.shape, arr.size, arr.dtype

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

```
((2, 3), 6, dtype('int32'))
```

7. Create a function array_dimension that takes a NumPy array as input and returns its dimensionality:

[9]:
```
import numpy as np

def array_dimension(arr):

    return arr.ndim

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

```
2
```

8. Design a function item_size_info that takes a NumPy array as input and returns the item size and the total size in bytes:

[14]:
```
# Answer

def item_size_info(arr):
    return arr.itemsize, arr.nbytes
```

```
arr8 = np.array([1, 2, 3], dtype=np.float64)
print(" Item size and total size:", item_size_info(arr8))
```

```
Item size and total size: (8, 24)
```

9. Create a function array_strides that takes a NumPy array as input and returns the strides of the array.

[16]:
```
def array_strides(arr):
    return arr.strides

arr9 = np.array([1.5, 2.6, 3.7])
print(" Strides of arr9:", array_strides(arr9))
```

```
Strides of arr9: (8,)
```

10. Design a function shape_stride_relationship that takes a NumPy array as input and returns the shape and strides of the array.

[19]:
```
def shape_stride_relationship(arr):
    return arr.shape, arr.strides

print(" Shape and strides of arr6:", shape_stride_relationship(arr))
```

```
Shape and strides of arr6: ((2, 3), (12, 4))
```

11. Create a function `create_zeros_array` that takes an integer `n` as input and returns a NumPy array of zeros with `n` elements.

[22]:
```
def create_zeros_array(n):
    return np.zeros(n)

print(" 11.Zeros array with 5 elements:", create_zeros_array(5))
```

```
11.Zeros array with 5 elements: [0. 0. 0. 0. 0.]
```

12. Write a function `create_ones_matrix` that takes integers `rows` and `cols` as inputs and generates a 2D NumPy array filled with ones of size `rows x cols`.

[21]:
```
def create_ones_matrix(rows, cols):
    return np.ones((rows, cols))

print("12. Ones matrix of size 3x4:", create_ones_matrix(3, 4))
```

```
12. Ones matrix of size 3x4: [[1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]]
```

13. Write a function `generate_range_array` that takes three integers start, stop, and step as arguments and creates a NumPy array with a range starting from `start`, ending at stop

(exclusive), and with the specified `step`.

[23]:
```
def generate_range_array(start, stop, step):
    return np.arange(start, stop, step)

print("13. Range array from 0 to 10 with step 2:", generate_range_array(0, 10,␣
↪2))
```

```
13. Range array from 0 to 10 with step 2: [0 2 4 6 8]
```

14. Design a function `generate_linear_space` that takes two floats `start`, `stop`, and an integer `num` as arguments and generates a NumPy array with num equally spaced values between `start` and `stop` (inclusive).

[24]:
```
def generate_linear_space(start, stop, num):
    return np.linspace(start, stop, num)

print("14. Linear space between 0 and 1 with 5 values:",␣
↪generate_linear_space(0, 1, 5))
```

```
14. Linear space between 0 and 1 with 5 values: [0.   0.25 0.5  0.75 1.  ]
```

15. Create a function `create_identity_matrix` that takes an integer `n` as input and generates a square identity matrix of size `n x n` using `numpy.eye`.

[25]:
```
def create_identity_matrix(n):
    return np.eye(n)

print("15. Identity matrix of size 3x3:", create_identity_matrix(3))
```

```
15. Identity matrix of size 3x3: [[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
```

16. Write a function that takes a Python list and converts it into a NumPy array.

[26]:
```
def list_to_numpy_array(lst):
    return np.array(lst)

print("16. List [1, 2, 3] to numpy array:", list_to_numpy_array([1, 2, 3]))
```

```
16. List [1, 2, 3] to numpy array: [1 2 3]
```

17. Create a NumPy array and demonstrate the use of `numpy.view` to create a new array object with the same data.

[27]:
```
arr17 = np.array([1, 2, 3, 4, 5])
arr17_view = arr17.view()
print("17. Original array and its view:", arr17, arr17_view)
```

```
17. Original array and its view: [1 2 3 4 5] [1 2 3 4 5]
```

18. Write a function that takes two NumPy arrays and concatenates them along a specified axis.

[28]:
```
def concatenate_arrays(arr1, arr2, axis=0):
    return np.concatenate((arr1, arr2), axis=axis)

arr18_1 = np.array([1, 2, 3])
arr18_2 = np.array([4, 5, 6])
print("18. Concatenated arrays:", concatenate_arrays(arr18_1, arr18_2))
```

```
18. Concatenated arrays: [1 2 3 4 5 6]
```

19. Create two NumPy arrays with different shapes and concatenate them horizontally using `numpy.concatenate`.

[30]:
```
arr19_1 = np.array([[1, 2, 3],[0,0,0]])
arr19_2 = np.array([[4, 5, 6], [7, 8, 9]])
print("19. Horizontally concatenated arrays:", concatenate_arrays(arr19_1,␣
 ↪arr19_2, axis=1))
```

```
19. Horizontally concatenated arrays: [[1 2 3 4 5 6]
 [0 0 0 7 8 9]]
```

20. Write a function that vertically stacks multiple NumPy arrays given as a list.

[31]:
```
def stack_arrays_vertically(arrays):
    return np.vstack(arrays)

print("20. Vertically stacked arrays:", stack_arrays_vertically([arr19_1,␣
 ↪arr19_2]))
```

```
20. Vertically stacked arrays: [[1 2 3]
 [0 0 0]
 [4 5 6]
 [7 8 9]]
```

21. Write a Python function using NumPy to create an array of integers within a specified range (inclusive) with a given step size.

[32]:
```
def create_range_array(start, stop, step):
    return np.arange(start, stop + 1, step)

print("21. Range array from 0 to 10 with step 2:", create_range_array(0, 10, 2))
```

```
21. Range array from 0 to 10 with step 2: [ 0  2  4  6  8 10]
```

22. Write a Python function using NumPy to generate an array of 10 equally spaced values between 0 and 1 (inclusive).

[39]:
```
def generate_equally_spaced_array():
    return np.linspace(0, 1, 10)
```

```
print("22. Equally spaced array between 0 and 1:
  ↪",generate_equally_spaced_array())
```

```
22. Equally spaced array between 0 and 1: [0.         0.11111111 0.22222222
0.33333333 0.44444444 0.55555556
 0.66666667 0.77777778 0.88888889 1.        ]
```

23. Write a Python function using NumPy to create an array of 5 logarithmically spaced values between 1 and 1000 (inclusive).

[40]:
```
def generate_log_space_array():
    return np.logspace(0, 3, 5)

print("23. Logarithmically spaced array between 1 and 1000:",␣
  ↪generate_log_space_array())
```

```
23. Logarithmically spaced array between 1 and 1000: [   1.
5.62341325   31.6227766   177.827941   1000.        ]
```

24. Create a Pandas DataFrame using a NumPy array that contains 5 rows and 3 columns, where the values are random integers between 1 and 100.

[42]:
```
import pandas as pd
data = np.random.randint(1, 101, size=(5, 3))
df = pd.DataFrame(data, columns=['A', 'B', 'C'])
print("24. DataFrame with random integers:\n", df)
```

```
24. DataFrame with random integers:
    A   B   C
0  49  73  65
1  78  11  94
2  55   6  22
3  85  35  52
4  90  75  93
```

25. Write a function that takes a Pandas DataFrame and replaces all negative values in a specific column with zeros. Use NumPy operations within the Pandas DataFrame.

[44]:
```
def replace_negatives_with_zero(df, column_name):
    df[column_name] = np.where(df[column_name] < 0, 0, df[column_name])
    return df

df25 = pd.DataFrame({
    'A': [1, -2, 3, -4, 5],
    'B': [-1, 2, -3, 4, -5]
})
print("25. DataFrame after replacing negatives in column 'A':\n",␣
  ↪replace_negatives_with_zero(df25, 'A'))
```

```
25. DataFrame after replacing negatives in column 'A':
```

```
   A  B
0  1 -1
1  0  2
2  3 -3
3  0  4
4  5 -5
```

26. Access the 3rd element from the given NumPy array.

```
arr = np.array([10, 20, 30, 40, 50])
```

[46]:
```
arr26 = np.array([10, 20, 30, 40, 50])
print("26. 3rd element of arr26:", arr26[2])
```

```
26. 3rd element of arr26: 30
```

27. Retrieve the element at index (1, 2) from the 2D NumPy array.

```
arr_2d = np.array([[1, 2, 3],
                   [4, 5, 6],
                   [7, 8, 9]])
```

[48]:
```
arr27 = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print("27. Element at index (1, 2) in arr27:", arr27[1, 2])
```

```
27. Element at index (1, 2) in arr27: 6
```

28. Using boolean indexing, extract elements greater than 5 from the given NumPy array.

```
arr = np.array([3, 8, 2, 10, 5, 7])
```

[49]:
```
arr28 = np.array([3, 8, 2, 10, 5, 7])
print("28. Elements greater than 5 in arr28:", arr28[arr28 > 5])
```

```
28. Elements greater than 5 in arr28: [ 8 10  7]
```

29. Perform basic slicing to extract elements from index 2 to 5 (inclusive) from the given NumPy array.

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

[50]:
```
arr29 = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9])
print("29. Sliced elements from index 2 to 5 in arr29:", arr29[2:6])
```

```
29. Sliced elements from index 2 to 5 in arr29: [3 4 5 6]
```

30. Slice the 2D NumPy array to extract the sub-array `[[2, 3], [5, 6]]` from the given array.

```
arr_2d = np.array([[1, 2, 3],
                   [4, 5, 6],
                   [7, 8, 9]])
```

[51]:
```
arr30 = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
sub_array = arr30[0:2, 1:3]
print("30. Sub-array extracted from arr30:", sub_array)
```

```
30. Sub-array extracted from arr30: [[2 3]
 [5 6]]
```

31. Write a NumPy function to extract elements in specific order from a given 2D array based on indices provided in another array.

[52]:
```
def extract_elements_by_indices(arr, indices):
    return arr[indices]

indices = np.array([[0, 0], [1, 1], [2, 2]])
arr31 = arr30
print("31. Extracted elements by indices:", extract_elements_by_indices(arr31,␣
 ↪indices))
```

```
31. Extracted elements by indices: [[[1 2 3]
  [1 2 3]]

 [[4 5 6]
  [4 5 6]]

 [[7 8 9]
  [7 8 9]]]
```

32. Create a NumPy function that filters elements greater than a threshold from a given 1D array using boolean indexing.

[53]:
```
def filter_elements(arr, threshold):
    return arr[arr > threshold]

arr32 = np.array([1, 2, 3, 4, 5])
print("32. Elements greater than 3:", filter_elements(arr32, 3))
```

```
32. Elements greater than 3: [4 5]
```

33. Develop a NumPy function that extracts specific elements from a 3D array using indices provided in three separate arrays for each dimension.

[54]:
```
def extract_3d_elements(arr, idx1, idx2, idx3):
    return arr[idx1, idx2, idx3]

arr33 = np.random.randint(1, 10, size=(3, 3, 3))
idx1, idx2, idx3 = np.array([0, 1, 2]), np.array([1, 2, 0]), np.array([2, 0, 1])
print("33. Extracted 3D elements:", extract_3d_elements(arr33, idx1, idx2,␣
 ↪idx3))
```

```
33. Extracted 3D elements: [8 6 8]
```

34. Write a NumPy function that returns elements from an array where both two conditions are satisfied using boolean indexing.

[55]:
```
def filter_two_conditions(arr, cond1, cond2):
    return arr[(cond1) & (cond2)]

arr34 = np.array([10, 15, 20, 25, 30])
print("34. Elements satisfying both conditions:", filter_two_conditions(arr34,␣
 ↪arr34 > 10, arr34 < 30))
```

```
34. Elements satisfying both conditions: [15 20 25]
```

35. Create a NumPy function that extracts elements from a 2D array using row and column indices provided in separate arrays.

[56]:
```
def extract_by_indices(arr, row_indices, col_indices):
    return arr[row_indices, col_indices]

row_indices = np.array([0, 1, 2])
col_indices = np.array([2, 1, 0])
arr35 = arr30
print("35. Extracted elements by row and column indices:",␣
 ↪extract_by_indices(arr35, row_indices, col_indices))
```

```
35. Extracted elements by row and column indices: [3 5 7]
```

36. Given an array arr of shape (3, 3), add a scalar value of 5 to each element using NumPy broadcasting.

[57]:
```
arr36 = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print("36. Array after adding 5:", arr36 + 5)
```

```
36. Array after adding 5: [[ 6  7  8]
 [ 9 10 11]
 [12 13 14]]
```

37. Consider two arrays arr1 of shape (1, 3) and arr2 of shape (3, 4). Multiply each row of arr2 by the corresponding element in arr1 using NumPy broadcasting.

[58]:
```
arr37_1 = np.array([[1, 2, 3]])
arr37_2 = np.array([[4, 5, 6, 7], [8, 9, 10, 11], [12, 13, 14, 15]])
result = arr37_2 * arr37_1.T
print("37. Result of broadcasting multiplication:", result)
```

```
37. Result of broadcasting multiplication: [[ 4  5  6  7]
 [16 18 20 22]
 [36 39 42 45]]
```

38. Given a 1D array arr1 of shape (1, 4) and a 2D array arr2 of shape (4, 3), add arr1 to each row of arr2 using NumPy broadcasting.

[59]:
```
arr38_1 = np.array([[1, 2, 3, 4]])
arr38_2 = np.array([[5, 6, 7], [8, 9, 10], [11, 12, 13], [14, 15, 16]])
result = arr38_2 + arr38_1.T
print("38. Result of broadcasting addition:", result)
```

```
38. Result of broadcasting addition: [[ 6  7  8]
 [10 11 12]
 [14 15 16]
 [18 19 20]]
```

39. Consider two arrays arr1 of shape (3, 1) and arr2 of shape (1, 3). Add these arrays using NumPy broadcasting.

[60]:
```
arr39_1 = np.array([[1], [2], [3]])
arr39_2 = np.array([[4, 5, 6]])
result = arr39_1 + arr39_2
print("39. Result of broadcasting addition:", result)
```

```
39. Result of broadcasting addition: [[5 6 7]
 [6 7 8]
 [7 8 9]]
```

40. Given arrays arr1 of shape (2, 3) and arr2 of shape (2, 2), perform multiplication using NumPy broadcasting. Handle the shape incompatibility.

[61]:
```
arr40_1 = np.array([[1, 2, 3], [4, 5, 6]])
arr40_2 = np.array([[7, 8], [9, 10]])

result = arr40_1[:, :2] * arr40_2
print("40. Result of broadcasting multiplication (handling shape␣
 ↪incompatibility):", result)
```

```
40. Result of broadcasting multiplication (handling shape incompatibility): [[ 7
16]
 [36 50]]
```

41. Calculate column-wise mean for the given array:

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

[62]:
```
arr41 = np.array([[1, 2, 3], [4, 5, 6]])
column_mean = np.mean(arr41, axis=0)
print("41. Column-wise mean:", column_mean)
```

```
41. Column-wise mean: [2.5 3.5 4.5]
```

42. Find maximum value in each row of the given array:

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

[63]:
```
row_max = np.max(arr41, axis=1)
print("42. Row-wise maximum:", row_max)
```

```
42. Row-wise maximum: [3 6]
```

43. For the given array, find indices of maximum value in each column.

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

[64]:
```
column_max_indices = np.argmax(arr41, axis=0)
print("43. Indices of maximum values in each column:", column_max_indices)
```

```
43. Indices of maximum values in each column: [1 1 1]
```

44. For the given array, apply custom function to calculate moving sum along rows.

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

[66]:
```
def moving_sum(arr, window_size):
    return np.convolve(arr, np.ones(window_size), 'valid')

result = np.apply_along_axis(moving_sum, 1, arr41, 2)
print("44. Moving sum along rows:", result)
```

```
44. Moving sum along rows: [[ 3.  5.]
 [ 9. 11.]]
```

45. In the given array, check if all elements in each column are even.

```
arr = np.array([[2, 4, 6], [3, 5, 7]])
```

[67]:
```
arr45 = np.array([[2, 4, 6], [3, 5, 7]])
all_even_columns = np.all(arr45 % 2 == 0, axis=0)
print("45. Are all elements in each column even?", all_even_columns)
```

```
45. Are all elements in each column even? [False False False]
```

46. Given a NumPy array arr, reshape it into a matrix of dimensions `m` rows and `n` columns. Return the reshaped matrix.

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

[69]:
```
def reshape_matrix(arr, m, n):
    return arr.reshape(m, n)

arr46 = np.array([1, 2, 3, 4, 5, 6])
reshaped = reshape_matrix(arr46, 2, 3)
print("46. Reshaped matrix:", reshaped)
```

```
46. Reshaped matrix: [[1 2 3]
 [4 5 6]]
```

47. Create a function that takes a matrix as input and returns the flattened array.

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

[70]:
```
def flatten_matrix(matrix):
    return matrix.flatten()

matrix47 = np.array([[1, 2, 3], [4, 5, 6]])
flattened = flatten_matrix(matrix47)
print("47. Flattened matrix:", flattened)
```

```
47. Flattened matrix: [1 2 3 4 5 6]
```

48. Write a function that concatenates two given arrays along a specified axis.

```
array1 = np.array([[1, 2], [3, 4]])
array2 = np.array([[5, 6], [7, 8]])
```

[71]:
```
def concatenate_arrays(arr1, arr2, axis=0):
    return np.concatenate((arr1, arr2), axis=axis)

arr48_1 = np.array([[1, 2], [3, 4]])
arr48_2 = np.array([[5, 6], [7, 8]])
concatenated = concatenate_arrays(arr48_1, arr48_2, axis=0)
print("48. Concatenated arrays along axis 0:", concatenated)
```

```
48. Concatenated arrays along axis 0: [[1 2]
 [3 4]
 [5 6]
 [7 8]]
```

49. Create a function that splits an array into multiple sub-arrays along a specified axis.

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

[72]:
```
def split_array(arr, sections, axis=0):
    return np.array_split(arr, sections, axis=axis)

arr49 = np.array([1, 2, 3, 4, 5, 6])
split_arr = split_array(arr49, 3)
print("49. Split array into 3 sub-arrays:", split_arr)
```

```
49. Split array into 3 sub-arrays: [array([1, 2]), array([3, 4]), array([5, 6])]
```

50. Write a function that inserts and then deletes elements from a given array at specified indices.

```
original_array = np.array([1, 2, 3, 4, 5])
indices_to_insert = [2, 4]
values_to_insert = [10, 11]
indices_to_delete = [1, 3]
```

[73]:
```
def insert_and_delete(arr, indices_to_insert, values_to_insert,␣
↪indices_to_delete):
```

```
    arr = np.insert(arr, indices_to_insert, values_to_insert)
    arr = np.delete(arr, indices_to_delete)
    return arr

arr50 = np.array([1, 2, 3, 4, 5])
indices_to_insert = [2, 4]
values_to_insert = [10, 11]
indices_to_delete = [1, 3]
result = insert_and_delete(arr50, indices_to_insert, values_to_insert,␣
 ↪indices_to_delete)
print("50. Array after insert and delete:", result)
```

```
50. Array after insert and delete: [ 1 10  4 11  5]
```

51. Create a NumPy array `arr1` with random integers and another array `arr2` with integers from 1 to 10. Perform element-wise addition between `arr1` and `arr2`.

[74]:
```
arr51_1 = np.random.randint(0, 10, size=10)
arr51_2 = np.arange(1, 11)
result = arr51_1 + arr51_2
print("51. Element-wise addition:", result)
```

```
51. Element-wise addition: [ 9  4  9  7  8 11 13 11  9 15]
```

52. Generate a NumPy array `arr1` with sequential integers from 10 to 1 and another array `arr2` with integers from 1 to 10. Subtract `arr2` from `arr1` element-wise.

[75]:
```
arr52_1 = np.arange(10, 0, -1)
arr52_2 = np.arange(1, 11)
result = arr52_1 - arr52_2
print("52. Element-wise subtraction:", result)
```

```
52. Element-wise subtraction: [ 9  7  5  3  1 -1 -3 -5 -7 -9]
```

53. Create a NumPy array `arr1` with random integers and another array `arr2` with integers from 1 to 5.Perform element-wise multiplication between `arr1` and `arr2`.

[76]:
```
arr53_1 = np.random.randint(1, 10, size=5)
arr53_2 = np.arange(1, 6)
result = arr53_1 * arr53_2
print("53. Element-wise multiplication:", result)
```

```
53. Element-wise multiplication: [ 6 14  3 16  5]
```

54. Generate a NumPy array `arr1` with even integers from 2 to 10 and another array `arr2` with integers from 1 to 5. Perform element-wise division of `arr1` by `arr2`

[77]:
```
arr54_1 = np.arange(2, 11, 2)
arr54_2 = np.arange(1, 6)
result = arr54_1 / arr54_2
```

```
print("54. Element-wise division:", result)
```

```
54. Element-wise division: [2. 2. 2. 2. 2.]
```

55. Create a NumPy array `arr1` with integers from 1 to 5 and another array `arr2` with the same numbers reversed. Calculate the exponentiation of `arr1` raised to the power of `arr2` element-wise.

[78]:
```
arr55_1 = np.arange(1, 6)
arr55_2 = arr55_1[::-1]
result = arr55_1 ** arr55_2
print("55. Element-wise exponentiation:", result)
```

```
55. Element-wise exponentiation: [ 1 16 27 16  5]
```

56. Write a function that counts the occurrences of a specific substring within a NumPy array of strings.

```
arr = np.array(['hello', 'world', 'hello', 'numpy', 'hello'])
```

[79]:
```
def count_substring(arr, substring):
    return np.char.count(arr, substring)

arr56 = np.array(['hello', 'world', 'hello', 'numpy', 'hello'])
result = count_substring(arr56, 'hello')
print("56. Count of 'hello':", result)
```

```
56. Count of 'hello': [1 0 1 0 1]
```

57. Write a function that extracts uppercase characters from a NumPy array of strings.

```
arr = np.array(['Hello', 'World', 'OpenAI', 'GPT'])
arr = np.array(['Hello', 'World', 'OpenAI', 'GPT'])
```

[81]:
```
import numpy as np

def extract_uppercase(arr):

    return np.array([''.join([char for char in string if char.isupper()]) for␣
 ↪string in arr])


arr = np.array(['Hello', 'World', 'OpenAI', 'GPT'])
uppercase_chars = extract_uppercase(arr)
print(uppercase_chars)
```

```
['H' 'W' 'OAI' 'GPT']
```

58. Write a function that replaces occurrences of a substring in a NumPy array of strings with a new string.

```
arr = np.array(['apple', 'banana', 'grape', 'pineapple'])
```

[82]:
```
def replace_substring(arr, old, new):
    return np.char.replace(arr, old, new)

arr58 = np.array(['Hello', 'World', 'OpenAI', 'GPT'])
result = replace_substring(arr58, 'World', 'Universe')
print("58. Replaced substring:", result)
```

```
58. Replaced substring: ['Hello' 'Universe' 'OpenAI' 'GPT']
```

59. Write a function that concatenates strings in a NumPy array element-wise.

```
arr1 = np.array(['Hello', 'World'])
 arr2 = np.array(['Open', 'AI'])
```

[83]:
```
def concatenate_strings(arr1, arr2):
    return np.char.add(arr1, arr2)

arr59_1 = np.array(['Hello', 'World'])
arr59_2 = np.array(['Open', 'AI'])
result = concatenate_strings(arr59_1, arr59_2)
print("59. Concatenated strings element-wise:", result)
```

```
59. Concatenated strings element-wise: ['HelloOpen' 'WorldAI']
```

60. Write a function that finds the length of the longest string in a NumPy array.

```
  arr = np.array(['apple', 'banana', 'grape', 'pineapple'])
arr = np.array(['apple', 'banana', 'grape', 'pineapple'])
```

[84]:
```
def find_longest_string(arr):
    return max(np.char.str_len(arr))

arr60 = np.array(['apple', 'banana', 'grape', 'pineapple'])
result = find_longest_string(arr60)
print("60. Length of longest string:", result)
```

```
60. Length of longest string: 9
```

61. Create a dataset of 100 random integers between 1 and 1000. Compute the mean, median, variance, and standard deviation of the dataset using NumPy's functions.

[85]:
```
arr61 = np.random.randint(1, 1001, size=100)
mean = np.mean(arr61)
median = np.median(arr61)
variance = np.var(arr61)
std_dev = np.std(arr61)
print(f"61. Mean: {mean}, Median: {median}, Variance: {variance}, Std Dev:␣
 ↪{std_dev}")
```

```
61. Mean: 455.31, Median: 423.0, Variance: 77277.4739, Std Dev:
277.9882621622719
```

62. Generate an array of 50 random numbers between 1 and 100. Find the 25th and 75th percentiles of the dataset.

[86]:
```
arr62 = np.random.randint(1, 101, size=50)
percentile_25 = np.percentile(arr62, 25)
percentile_75 = np.percentile(arr62, 75)
print(f"62. 25th Percentile: {percentile_25}, 75th Percentile: {percentile_75}")
```

```
62. 25th Percentile: 23.25, 75th Percentile: 81.5
```

63. Create two arrays representing two sets of variables. Compute the correlation coefficient between these arrays using NumPy's `corrcoef` function.

[87]:
```
arr63_1 = np.random.randint(1, 100, size=50)
arr63_2 = np.random.randint(1, 100, size=50)
correlation = np.corrcoef(arr63_1, arr63_2)
print("63. Correlation coefficient:\n", correlation)
```

```
63. Correlation coefficient:
 [[ 1.         -0.20946034]
 [-0.20946034  1.        ]]
```

64. Create two matrices and perform matrix multiplication using NumPy's `dot` function.

[89]:
```
mat64_1 = np.random.randint(1, 10, size=(3, 2))
mat64_2 = np.random.randint(1, 10, size=(2, 3))
result = np.dot(mat64_1, mat64_2)
print("64. Matrix multiplication result:\n", result)
```

```
64. Matrix multiplication result:
 [[ 8 20 26]
 [30 78 69]
 [28 70 91]]
```

65. Create an array of 50 integers between 10 and 1000. Calculate the 10th, 50th (median), and 90th ercentiles along with the first and third quartiles.

[90]:
```
arr65 = np.random.randint(10, 1001, size=50)
percentile_10 = np.percentile(arr65, 10)
median = np.percentile(arr65, 50)
percentile_90 = np.percentile(arr65, 90)
first_quartile = np.percentile(arr65, 25)
third_quartile = np.percentile(arr65, 75)
print(f"65. 10th Percentile: {percentile_10}, Median: {median}, 90th Percentile:
 ↪ {percentile_90}, Q1: {first_quartile}, Q3: {third_quartile}")
```

```
65. 10th Percentile: 132.2, Median: 524.0, 90th Percentile: 917.1, Q1: 330.75,
Q3: 771.0
```

66. Create a NumPy array of integers and find the index of a specific element.

[91]:
```
arr66 = np.array([10, 20, 30, 40, 50])
index = np.where(arr66 == 30)
print("66. Index of 30 in arr66:", index)
```

```
66. Index of 30 in arr66: (array([2], dtype=int64),)
```

67. Generate a random NumPy array and sort it in ascending orde

[92]:
```
arr67 = np.random.randint(1, 100, size=10)
sorted_arr = np.sort(arr67)
print("67. Sorted array:", sorted_arr)
```

```
67. Sorted array: [10 14 27 46 52 59 76 76 81 84]
```

68. Filter elements >20 in the given NumPy array.

```
arr = np.array([12, 25, 6, 42, 8, 30])
```

[93]:
```
arr68 = np.array([12, 25, 6, 42, 8, 30])
filtered_arr = arr68[arr68 > 20]
print("68. Elements greater than 20:", filtered_arr)
```

```
68. Elements greater than 20: [25 42 30]
```

69. Filter elements which are divisible by 3 from a given NumPy array.

```
arr = np.array([1, 5, 8, 12, 15])
```

[95]:
```
arr69 = np.array([1, 5, 8, 12, 15,9,53])
filtered_arr = arr69[arr69 % 3 == 0]
print("69. Elements divisible by 3:", filtered_arr)
```

```
69. Elements divisible by 3: [12 15  9]
```

70. Filter elements which are  20 and  40 from a given NumPy array.

```
arr = np.array([10, 20, 30, 40, 50])
```

[96]:
```
import numpy as np

arr = np.array([10, 20, 30, 40, 50])

filtered_arr = arr[(arr >= 20) & (arr <= 40)]

print(filtered_arr)
```

```
[20 30 40]
```

71. For the given NumPy array, check its byte order using the `dtype` attribute byteorder.

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

[101]:
```
import numpy as np

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

byte_order = arr.dtype.byteorder

print(f'Byte order: {byte_order}')
```

```
Byte order: =
```

72. For the given NumPy array, perform byte swapping in place using `byteswap()`.

```
arr = np.array([1, 2, 3], dtype=np.int32)
```

[102]:
```
import numpy as np

arr = np.array([1, 2, 3], dtype=np.int32)

arr.byteswap(inplace=True)

print(arr)
```

```
[16777216 33554432 50331648]
```

73. For the given NumPy array, swap its byte order without modifying the original array using `newbyteorder()`.

```
arr = np.array([1, 2, 3], dtype=np.int32)
```

[103]:
```
import numpy as np

# Given array
arr = np.array([1, 2, 3], dtype=np.int32)

swapped_arr = arr.newbyteorder()

print("Original array:", arr)
print("Swapped byte order array:", swapped_arr)
```

```
Original array: [1 2 3]
Swapped byte order array: [16777216 33554432 50331648]
```

74. For the given NumPy array and swap its byte order conditionally based on system endianness using `newbyteorder()`.

```
arr = np.array([1, 2, 3], dtype=np.int32)
```

[104]:
```
import numpy as np
import sys

# Given array
```

```
arr = np.array([1, 2, 3], dtype=np.int32)

system_endianness = sys.byteorder

if system_endianness == 'little':
    swapped_arr = arr.newbyteorder('>')
elif system_endianness == 'big':
    swapped_arr = arr.newbyteorder('<')
else:
    raise ValueError("Unknown system endianness")

print("Original array:", arr)
print("Swapped byte order array:", swapped_arr)
```

```
Original array: [1 2 3]
Swapped byte order array: [16777216 33554432 50331648]
```

75. For the given NumPy array, check if byte swapping is necessary for the current system using `dtype` attribute `byteorder`.

    ```
    arr = np.array([1, 2, 3], dtype=np.int32)
    ```

[105]:
```
import numpy as np
import sys


arr = np.array([1, 2, 3], dtype=np.int32)

system_endianness = sys.byteorder


array_byte_order = arr.dtype.byteorder


if array_byte_order == '=':

    print("Byte swapping is not necessary.")
elif (system_endianness == 'little' and array_byte_order == '>') or \
     (system_endianness == 'big' and array_byte_order == '<'):
    print("Byte swapping is necessary.")
else:
    print("Byte swapping is not necessary.")
```

```
Byte swapping is not necessary.
```

76. Create a NumPy array `arr1` with values from 1 to 10. Create a copy of `arr1` named `copy_arr` and modify an element in `copy_arr`. Check if modifying `copy_arr` affects `arr1`.

[106]:
```
import numpy as np

arr1 = np.arange(1, 11)

copy_arr = arr1.copy()

copy_arr[0] = 99

print("Original array (arr1):", arr1)
print("Modified copy array (copy_arr):", copy_arr)
```

```
Original array (arr1): [ 1  2  3  4  5  6  7  8  9 10]
Modified copy array (copy_arr): [99  2  3  4  5  6  7  8  9 10]
```

77. Create a 2D NumPy array `matrix` of shape (3, 3) with random integers. Extract a slice `view_slice` from the matrix. Modify an element in `view_slice` and observe if it changes the original `matrix`

[107]:
```
import numpy as np

matrix = np.random.randint(1, 10, size=(3, 3))

view_slice = matrix[1:3, 0:2]

print("Original matrix:\n", matrix)
print("View slice before modification:\n", view_slice)

view_slice[0, 0] = 99

print("\nModified view slice:\n", view_slice)
print("Original matrix after modification:\n", matrix)
```

```
Original matrix:
 [[9 9 3]
 [8 3 9]
 [8 6 7]]
View slice before modification:
 [[8 3]
 [8 6]]

Modified view slice:
 [[99  3]
 [ 8  6]]
Original matrix after modification:
 [[ 9  9  3]
 [99  3  9]
 [ 8  6  7]]
```

78. Create a NumPy array `array_a` of shape (4, 3) with sequential integers from 1 to 12. Extract

a slice `view_b` from `array_a` and broadcast the addition of 5 to view_b. Check if it alters the original `array_a`.

[108]:
```
import numpy as np

array_a = np.arange(1, 13).reshape(4, 3)

view_b = array_a[1:4, 0:2]

print("Original array (array_a):\n", array_a)
print("Slice (view_b) before broadcasting:\n", view_b)

view_b += 5

print("\nSlice (view_b) after broadcasting:\n", view_b)
print("Original array (array_a) after broadcasting:\n", array_a)
```

```
Original array (array_a):
 [[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]
Slice (view_b) before broadcasting:
 [[ 4  5]
 [ 7  8]
 [10 11]]

Slice (view_b) after broadcasting:
 [[ 9 10]
 [12 13]
 [15 16]]
Original array (array_a) after broadcasting:
 [[ 1  2  3]
 [ 9 10  6]
 [12 13  9]
 [15 16 12]]
```

79. Create a NumPy array `orig_array` of shape (2, 4) with values from 1 to 8. Create a reshaped view `reshaped_view` of shape (4, 2) from orig_array. Modify an element in `reshaped_view` and check if it reflects changes in the original `orig_array`.

[109]:
```
import numpy as np

orig_array = np.arange(1, 9).reshape(2, 4)

reshaped_view = orig_array.reshape(4, 2)

print("Original array (orig_array):\n", orig_array)
```

```
print("Reshaped view (reshaped_view) before modification:\n", reshaped_view)

reshaped_view[0, 0] = 99

print("\nReshaped view (reshaped_view) after modification:\n", reshaped_view)
print("Original array (orig_array) after modification:\n", orig_array)
```

```
Original array (orig_array):
 [[1 2 3 4]
 [5 6 7 8]]
Reshaped view (reshaped_view) before modification:
 [[1 2]
 [3 4]
 [5 6]
 [7 8]]

Reshaped view (reshaped_view) after modification:
 [[99  2]
 [ 3  4]
 [ 5  6]
 [ 7  8]]
Original array (orig_array) after modification:
 [[99  2  3  4]
 [ 5  6  7  8]]
```

80. Create a NumPy array `data` of shape (3, 4) with random integers. Extract a copy `data_copy` of elements greater than 5. Modify an element in `data_copy` and verify if it affects the original `data`.

[111]:
```
import numpy as np

data = np.random.randint(1, 10, size=(3, 4))

data_copy = data[data > 5].copy()

print("Original array (data):\n", data)
print("Extracted copy (data_copy) before modification:\n", data_copy)

data_copy[0] = 99

print("\nExtracted copy (data_copy) after modification:\n", data_copy)
print("Original array (data) after modification:\n", data)
```

```
Original array (data):
 [[8 8 8 2]
 [3 8 3 2]
 [5 4 9 2]]
Extracted copy (data_copy) before modification:
```

```
 [8 8 8 8 9]

Extracted copy (data_copy) after modification:
 [99  8  8  8  9]
Original array (data) after modification:
 [[8 8 8 2]
 [3 8 3 2]
 [5 4 9 2]]
```

81. Create two matrices A and B of identical shape containing integers and perform addition and subtraction operations between them.

[112]:
```
import numpy as np

A = np.array([[1, 2, 3], [4, 5, 6]])
B = np.array([[6, 5, 4], [3, 2, 1]])

addition_result = A + B

subtraction_result = A - B

print("Matrix A:\n", A)
print("Matrix B:\n", B)
print("Addition result (A + B):\n", addition_result)
print("Subtraction result (A - B):\n", subtraction_result)
```

```
Matrix A:
 [[1 2 3]
 [4 5 6]]
Matrix B:
 [[6 5 4]
 [3 2 1]]
Addition result (A + B):
 [[7 7 7]
 [7 7 7]]
Subtraction result (A - B):
 [[-5 -3 -1]
 [ 1  3  5]]
```

82. Generate two matrices `C` (3x2) and `D` (2x4) and perform matrix multiplication.

[113]:
```
import numpy as np

C = np.random.randint(1, 10, size=(3, 2))
D = np.random.randint(1, 10, size=(2, 4))

result = np.matmul(C, D)

print("Matrix C (3x2):\n", C)
```

```
print("Matrix D (2x4):\n", D)
print("Matrix multiplication result (C @ D):\n", result)
```

```
Matrix C (3x2):
 [[5 6]
 [9 6]
 [4 2]]
Matrix D (2x4):
 [[7 6 1 7]
 [9 5 5 2]]
Matrix multiplication result (C @ D):
 [[ 89  60  35  47]
 [117  84  39  75]
 [ 46  34  14  32]]
```

83. Create a matrix `E` and find its transpose.

[114]:
```
import numpy as np

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

E_transpose = E.T

print("Matrix E:\n", E)
print("Transpose of matrix E:\n", E_transpose)
```

```
Matrix E:
 [[1 2 3]
 [4 5 6]
 [7 8 9]]
Transpose of matrix E:
 [[1 4 7]
 [2 5 8]
 [3 6 9]]
```

84. Generate a square matrix `F` and compute its determinant.

[115]:
```
import numpy as np

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

determinant = np.linalg.det(F)

print("Matrix F:\n", F)
print("Determinant of matrix F:", determinant)
```

```
Matrix F:
 [[4 2 1]
 [3 5 2]
 [1 1 3]]
Determinant of matrix F: 36.000000000000014
```

85. Create a square matrix `G` and find its inverse.

[116]:
```python
import numpy as np

G = np.array([[1, 2, 3],
              [0, 1, 4],
              [5, 6, 0]])

try:
    G_inverse = np.linalg.inv(G)
    print("Matrix G:\n", G)
    print("Inverse of matrix G:\n", G_inverse)
except np.linalg.LinAlgError:
    print("Matrix G is not invertible.")
```

```
Matrix G:
 [[1 2 3]
 [0 1 4]
 [5 6 0]]
Inverse of matrix G:
 [[-24.  18.   5.]
 [ 20. -15.  -4.]
 [ -5.   4.   1.]]
```