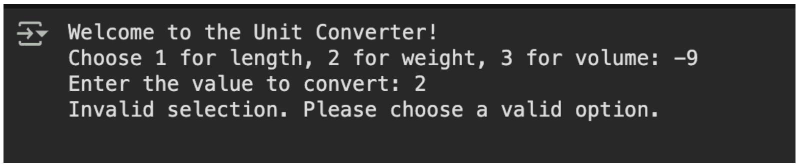
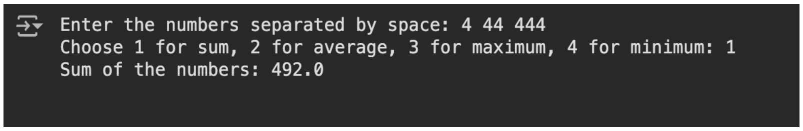
Task 1:



```
Welcome to the Unit Converter!
Choose 1 for length, 2 for weight, 3 for volume: -9
Enter the value to convert: 2
Invalid selection. Please choose a valid option.
```

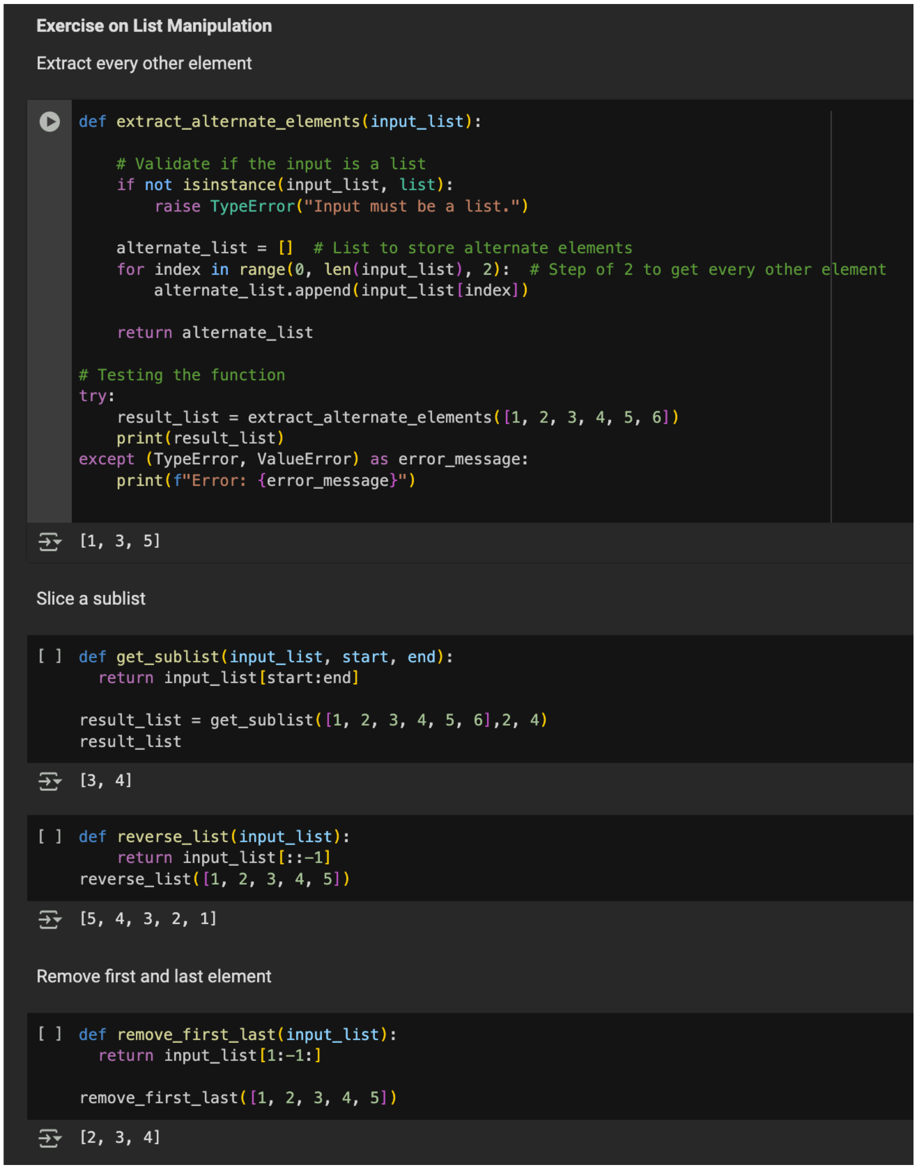
Task 2:



```
Enter the numbers separated by space: 4 44 444
Choose 1 for sum, 2 for average, 3 for maximum, 4 for minimum: 1
Sum of the numbers: 492.0
```

Exercise on List Manipulation

Extract every other element:



Exercise on List Manipulation

Extract every other element

```
def extract_alternate_elements(input_list):

    # Validate if the input is a list
    if not isinstance(input_list, list):
        raise TypeError("Input must be a list.")

    alternate_list = []  # List to store alternate elements
    for index in range(0, len(input_list), 2):  # Step of 2 to get every other element
        alternate_list.append(input_list[index])

    return alternate_list

# Testing the function
try:
    result_list = extract_alternate_elements([1, 2, 3, 4, 5, 6])
    print(result_list)
except (TypeError, ValueError) as error_message:
    print(f"Error: {error_message}")
```

```
[1, 3, 5]
```

Slice a sublist

```
[ ] def get_sublist(input_list, start, end):
      return input_list[start:end]

    result_list = get_sublist([1, 2, 3, 4, 5, 6],2, 4)
    result_list
```

```
[3, 4]
```

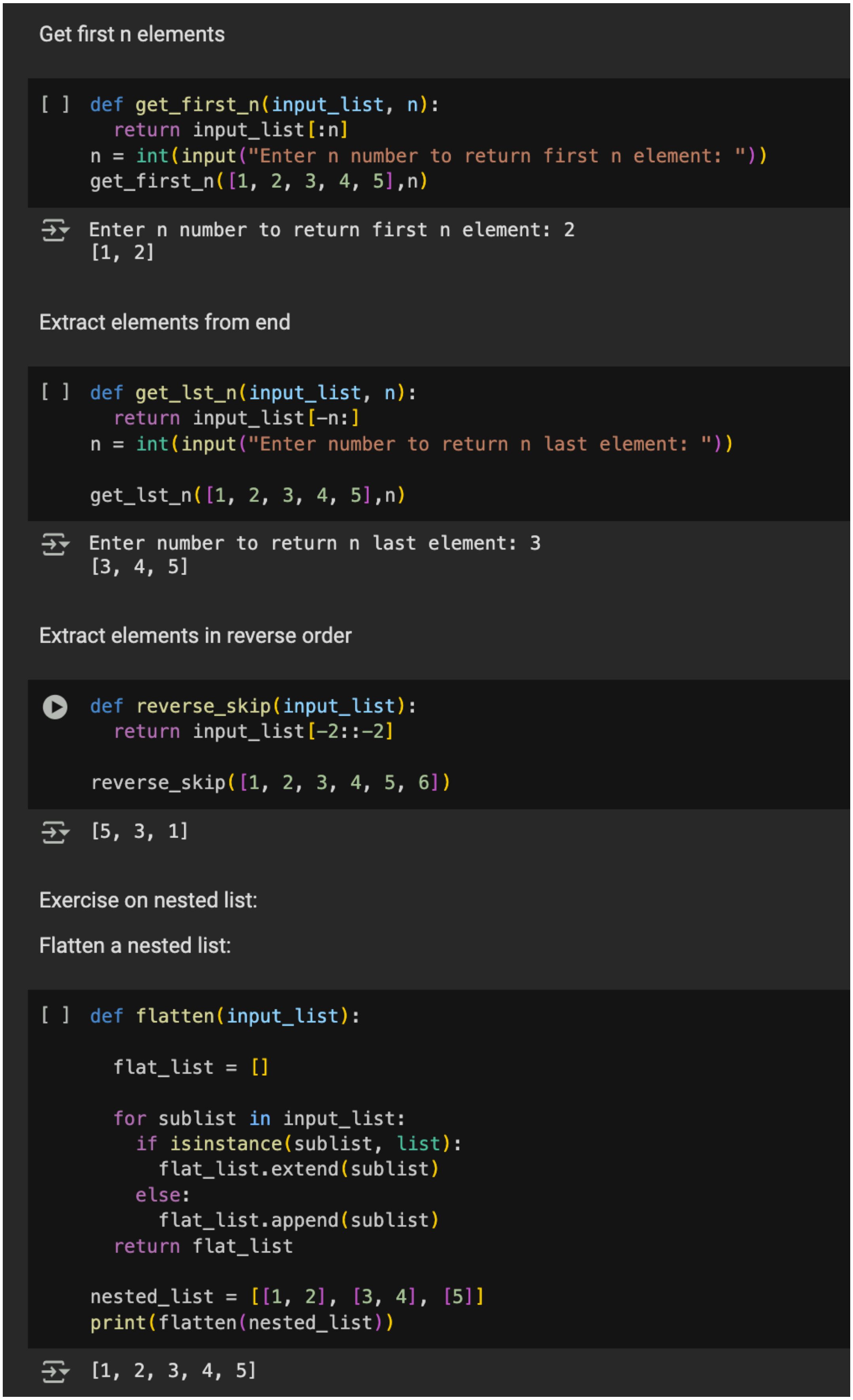
```
[ ] def reverse_list(input_list):
        return input_list[::-1]
    reverse_list([1, 2, 3, 4, 5])
```

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

Remove first and last element

```
[ ] def remove_first_last(input_list):
      return input_list[1:-1:]

    remove_first_last([1, 2, 3, 4, 5])
```

```
[2, 3, 4]
```



Get first n elements

```
def get_first_n(input_list, n):
  return input_list[:n]
n = int(input("Enter n number to return first n element: "))
get_first_n([1, 2, 3, 4, 5],n)
```

```
Enter n number to return first n element: 2
[1, 2]
```

Extract elements from end

```
def get_lst_n(input_list, n):
  return input_list[-n:]
n = int(input("Enter number to return n last element: "))

get_lst_n([1, 2, 3, 4, 5],n)
```

```
Enter number to return n last element: 3
[3, 4, 5]
```

Extract elements in reverse order

```
def reverse_skip(input_list):
  return input_list[-2::-2]

reverse_skip([1, 2, 3, 4, 5, 6])
```

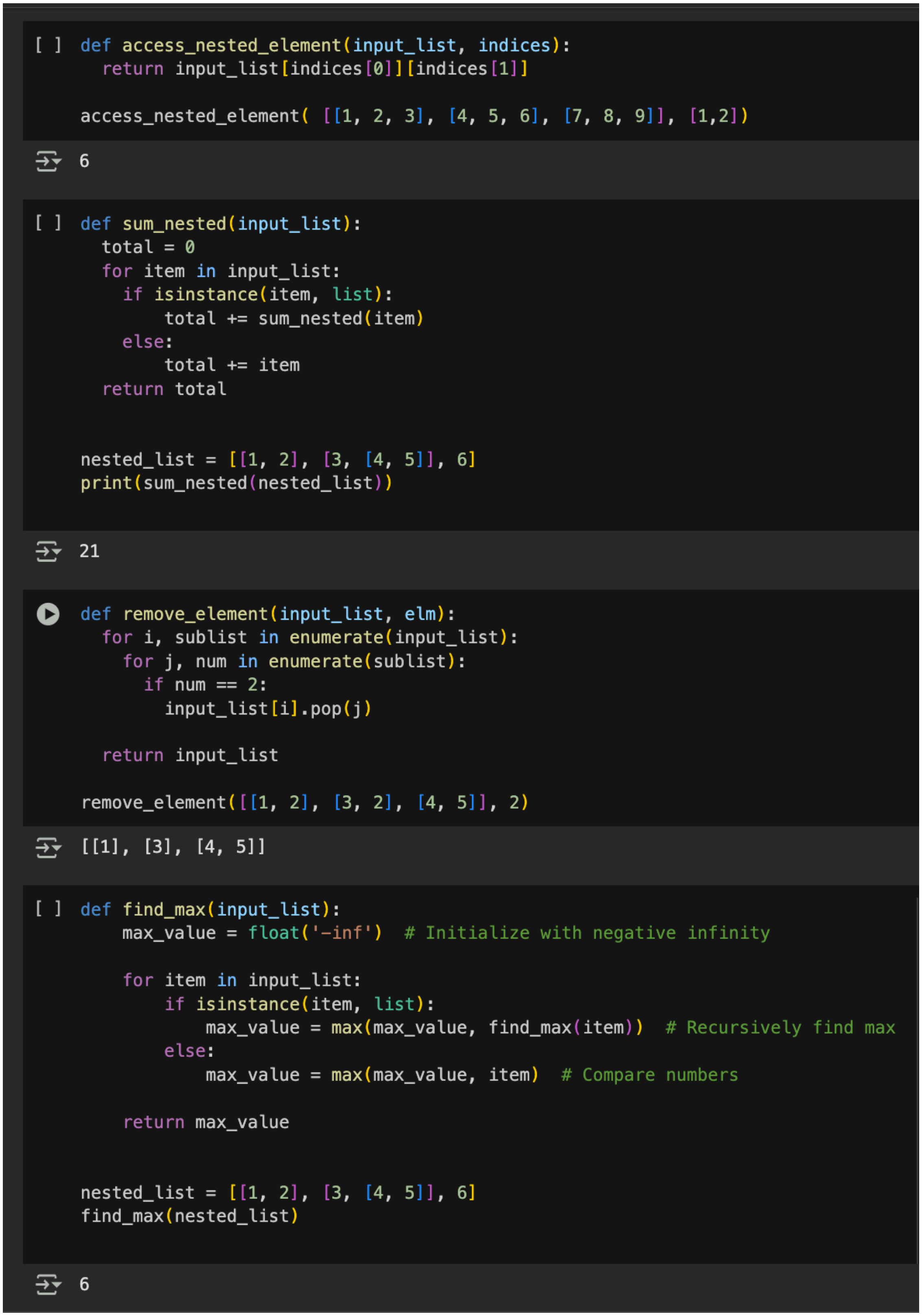
```
[5, 3, 1]
```

Exercise on nested list:

Flatten a nested list:

```
def flatten(input_list):

  flat_list = []

  for sublist in input_list:
    if isinstance(sublist, list):
      flat_list.extend(sublist)
    else:
      flat_list.append(sublist)
  return flat_list

nested_list = [[1, 2], [3, 4], [5]]
print(flatten(nested_list))
```

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



```
def access_nested_element(input_list, indices):
  return input_list[indices[0]][indices[1]]

access_nested_element( [[1, 2, 3], [4, 5, 6], [7, 8, 9]], [1,2])
```

```
6
```

```
def sum_nested(input_list):
  total = 0
  for item in input_list:
    if isinstance(item, list):
        total += sum_nested(item)
    else:
        total += item
  return total


nested_list = [[1, 2], [3, [4, 5]], 6]
print(sum_nested(nested_list))
```

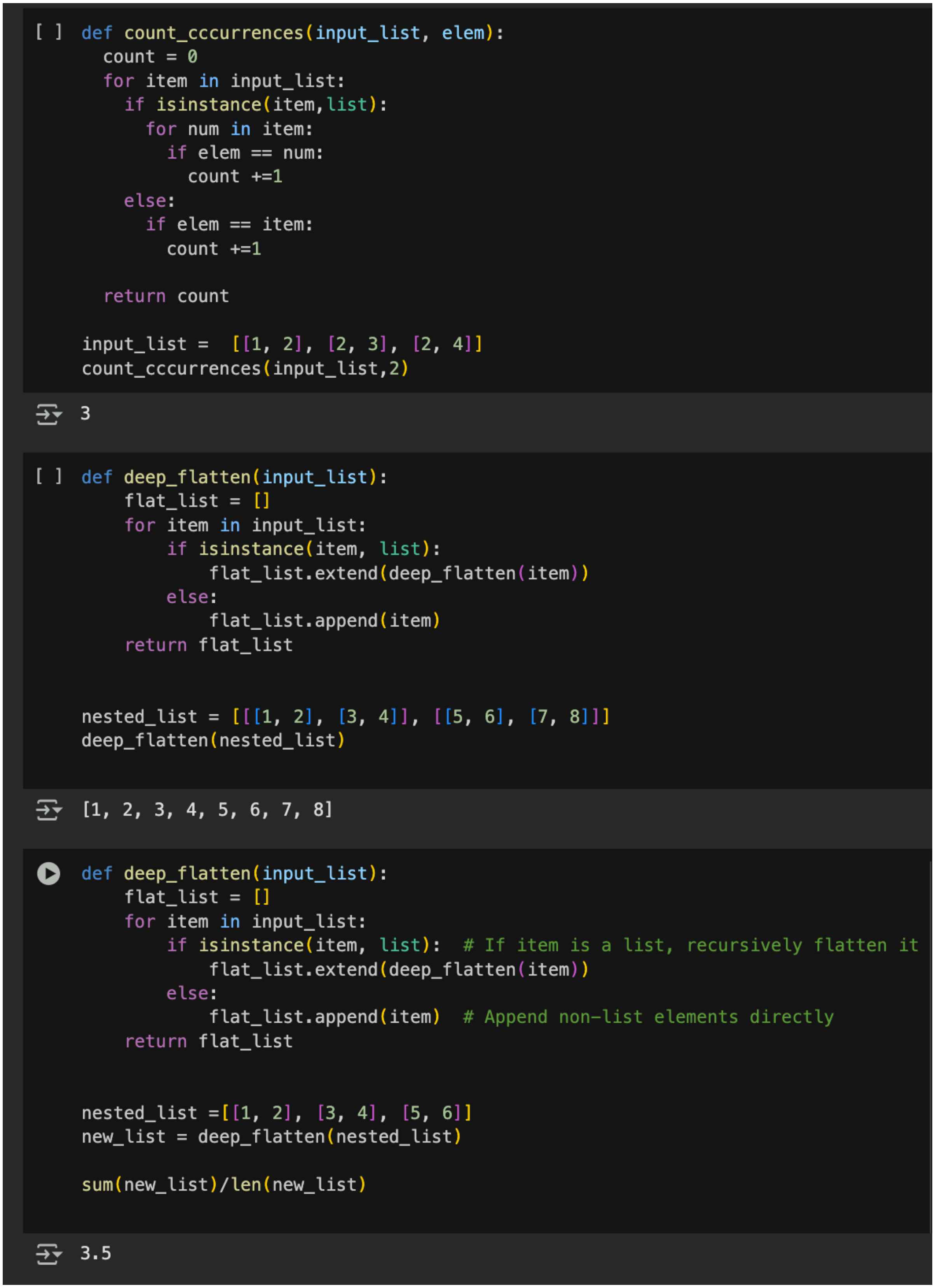
```
21
```

```
def remove_element(input_list, elm):
  for i, sublist in enumerate(input_list):
    for j, num in enumerate(sublist):
      if num == 2:
        input_list[i].pop(j)

  return input_list

remove_element([[1, 2], [3, 2], [4, 5]], 2)
```

```
[[1], [3], [4, 5]]
```

```
def find_max(input_list):
    max_value = float('-inf')  # Initialize with negative infinity

    for item in input_list:
        if isinstance(item, list):
            max_value = max(max_value, find_max(item))  # Recursively find max
        else:
            max_value = max(max_value, item)  # Compare numbers

    return max_value


nested_list = [[1, 2], [3, [4, 5]], 6]
find_max(nested_list)
```

```
6
```



```
def count_cccurrences(input_list, elem):
  count = 0
  for item in input_list:
    if isinstance(item,list):
      for num in item:
        if elem == num:
          count +=1
    else:
      if elem == item:
        count +=1

  return count

input_list =  [[1, 2], [2, 3], [2, 4]]
count_cccurrences(input_list,2)
```

```
3
```

```
def deep_flatten(input_list):
    flat_list = []
    for item in input_list:
        if isinstance(item, list):
            flat_list.extend(deep_flatten(item))
        else:
            flat_list.append(item)
    return flat_list


nested_list = [[[1, 2], [3, 4]], [[5, 6], [7, 8]]]
deep_flatten(nested_list)
```

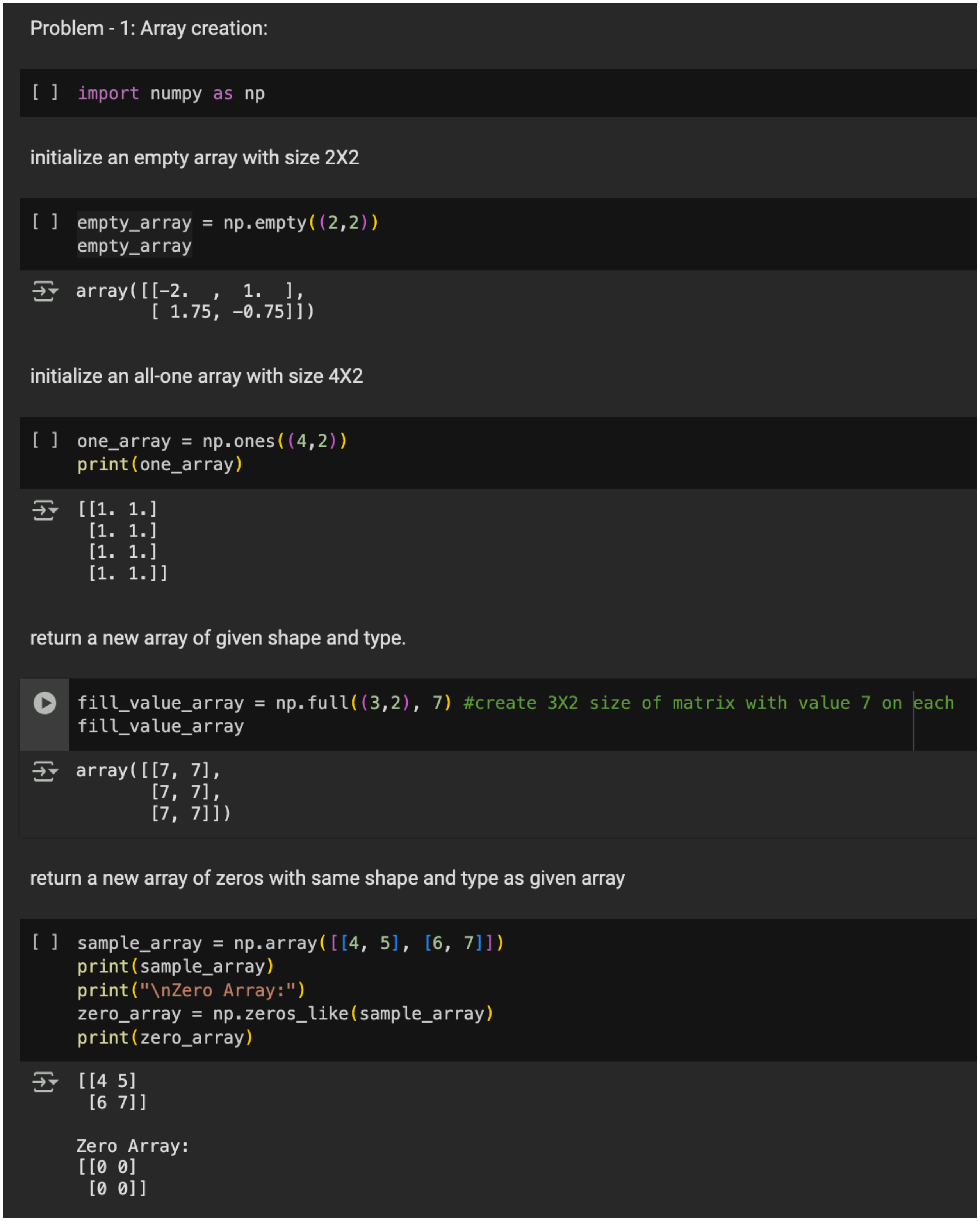
```
[1, 2, 3, 4, 5, 6, 7, 8]
```

```
def deep_flatten(input_list):
    flat_list = []
    for item in input_list:
        if isinstance(item, list):  # If item is a list, recursively flatten it
            flat_list.extend(deep_flatten(item))
        else:
            flat_list.append(item)  # Append non-list elements directly
    return flat_list


nested_list =[[1, 2], [3, 4], [5, 6]]
new_list = deep_flatten(nested_list)

sum(new_list)/len(new_list)
```

```
3.5
```

NumPy

Basic vector and matrix operations with numpy



Problem - 1: Array creation:

```
import numpy as np
```

initialize an empty array with size 2X2

```
empty_array = np.empty((2,2))
empty_array
```

```
array([[-2.  ,  1.  ],
       [ 1.75, -0.75]])
```

initialize an all-one array with size 4X2

```
one_array = np.ones((4,2))
print(one_array)
```

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

return a new array of given shape and type.

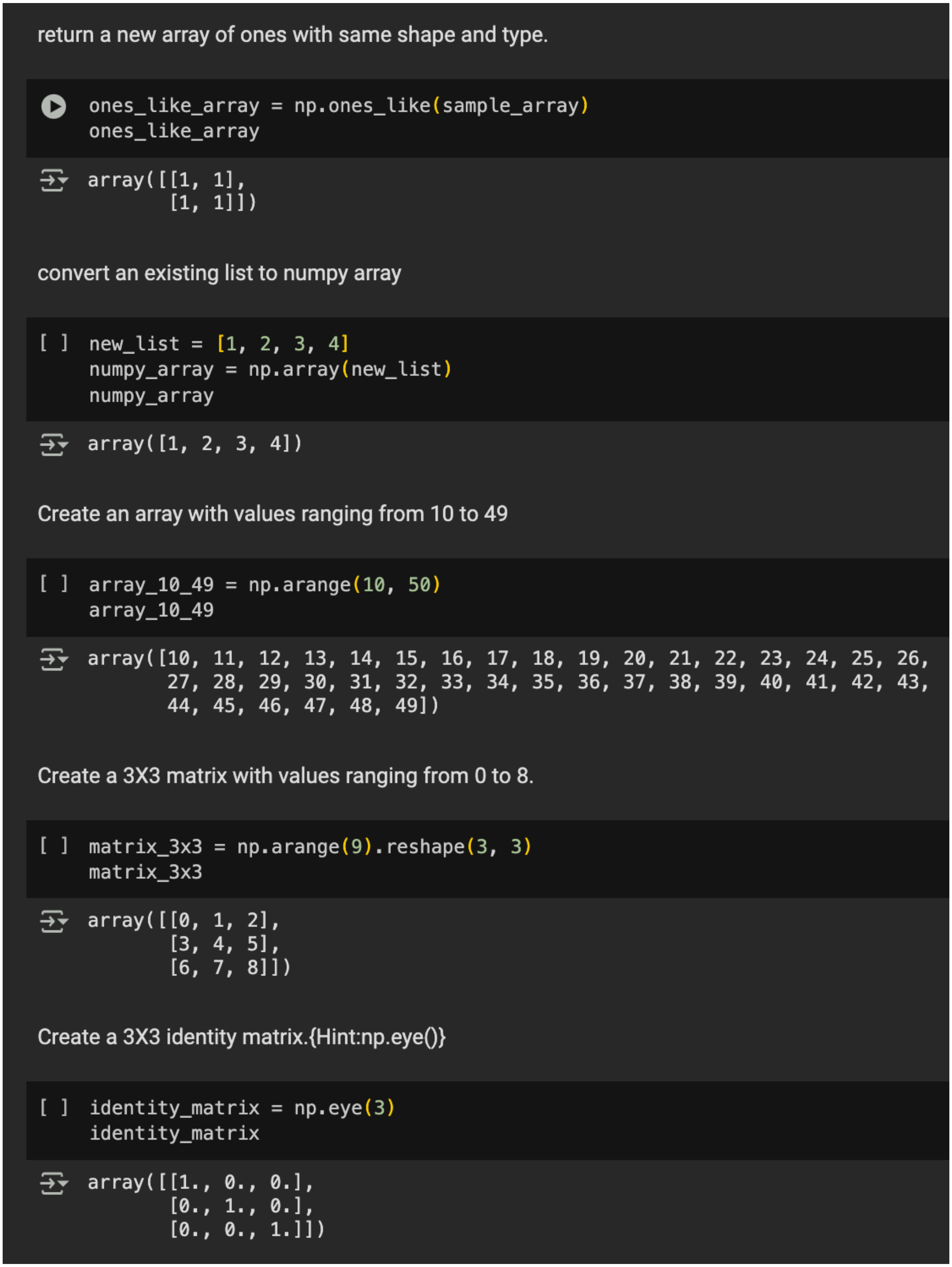
```
fill_value_array = np.full((3,2), 7) #create 3X2 size of matrix with value 7 on each
fill_value_array
```

```
array([[7, 7],
       [7, 7],
       [7, 7]])
```

return a new array of zeros with same shape and type as given array

```
sample_array = np.array([[4, 5], [6, 7]])
print(sample_array)
print("\nZero Array:")
zero_array = np.zeros_like(sample_array)
print(zero_array)
```

```
[[4 5]
 [6 7]]

Zero Array:
[[0 0]
 [0 0]]
```



return a new array of ones with same shape and type.

```
ones_like_array = np.ones_like(sample_array)
ones_like_array
```

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

convert an existing list to numpy array

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

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

Create an array with values ranging from 10 to 49

```
array_10_49 = np.arange(10, 50)
array_10_49
```

```
array([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])
```

Create a 3X3 matrix with values ranging from 0 to 8.

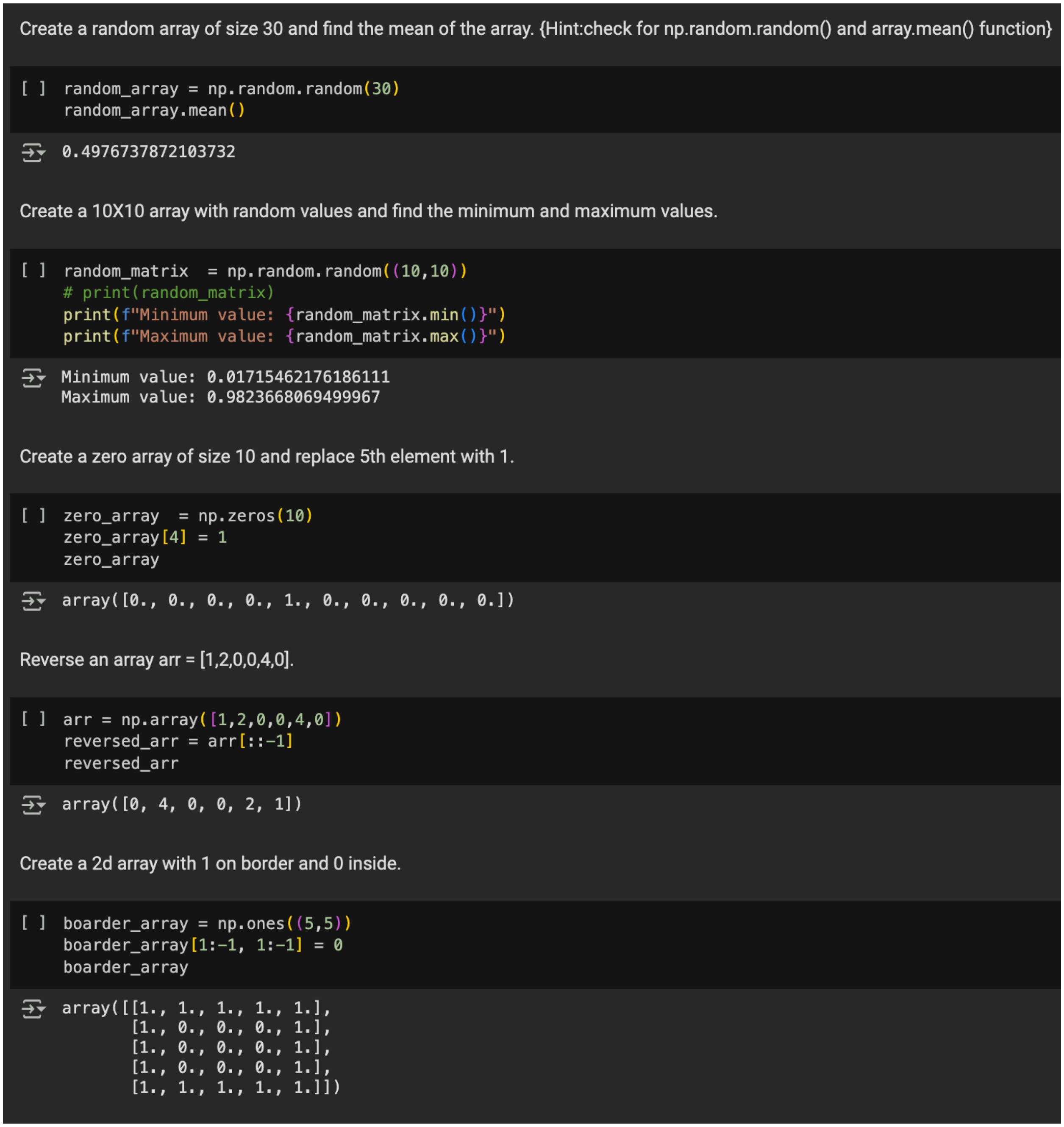
```
matrix_3x3 = np.arange(9).reshape(3, 3)
matrix_3x3
```

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

Create a 3X3 identity matrix.{Hint:np.eye()}

```
identity_matrix = np.eye(3)
identity_matrix
```

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



Create a random array of size 30 and find the mean of the array. {Hint:check for np.random.random() and array.mean() function}

```
random_array = np.random.random(30)
random_array.mean()
```

```
0.4976737872103732
```

Create a 10X10 array with random values and find the minimum and maximum values.

```
random_matrix  = np.random.random((10,10))
# print(random_matrix)
print(f"Minimum value: {random_matrix.min()}")
print(f"Maximum value: {random_matrix.max()}")
```

```
Minimum value: 0.01715462176186111
Maximum value: 0.9823668069499967
```

Create a zero array of size 10 and replace 5th element with 1.

```
zero_array  = np.zeros(10)
zero_array[4] = 1
zero_array
```

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

Reverse an array arr = [1,2,0,0,4,0].

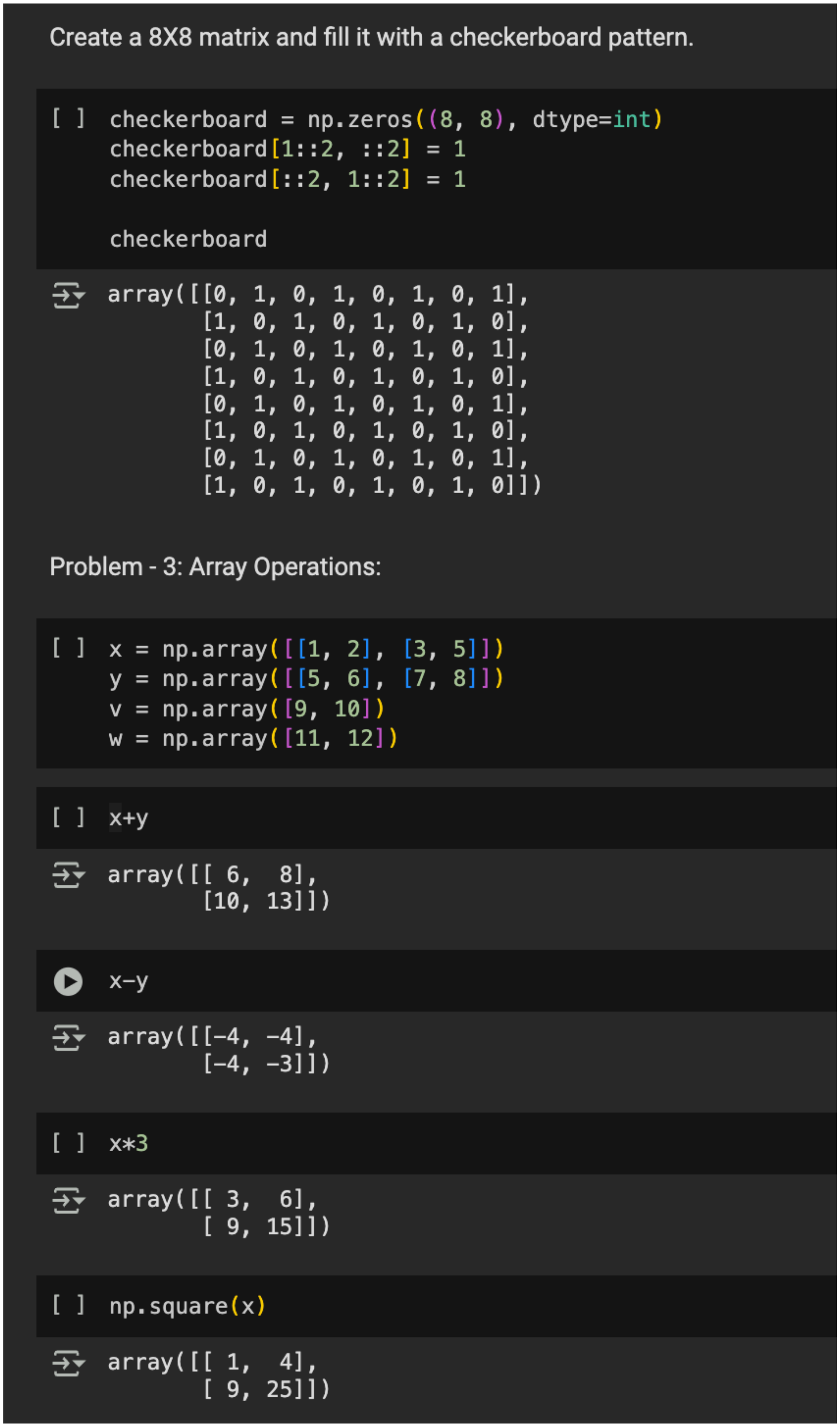
```
arr = np.array([1,2,0,0,4,0])
reversed_arr = arr[::-1]
reversed_arr
```

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

Create a 2d array with 1 on border and 0 inside.

```
boarder_array = np.ones((5,5))
boarder_array[1:-1, 1:-1] = 0
boarder_array
```

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



Create a 8X8 matrix and fill it with a checkerboard pattern.

```
[ ]  checkerboard = np.zeros((8, 8), dtype=int)
     checkerboard[1::2, ::2] = 1
     checkerboard[::2, 1::2] = 1

     checkerboard
```

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

Problem - 3: Array Operations:

```
[ ]  x = np.array([[1, 2], [3, 5]])
     y = np.array([[5, 6], [7, 8]])
     v = np.array([9, 10])
     w = np.array([11, 12])
```

```
[ ]  x+y
```

```
array([[ 6,  8],
       [10, 13]])
```

```
x-y
```

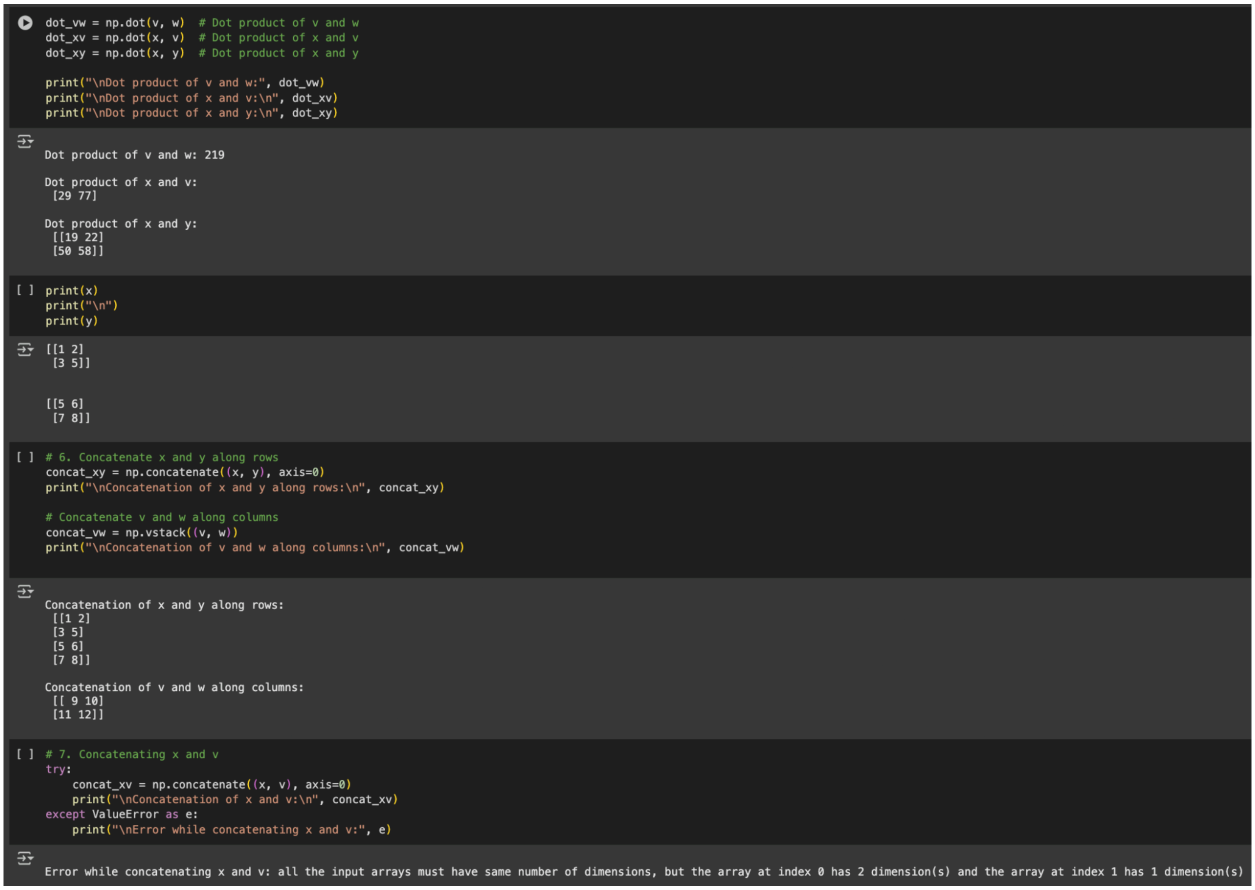
```
array([[-4, -4],
       [-4, -3]])
```

```
[ ]  x*3
```

```
array([[ 3,  6],
       [ 9, 15]])
```

```
[ ]  np.square(x)
```

```
array([[ 1,  4],
       [ 9, 25]])
```



```
dot_vw = np.dot(v, w)  # Dot product of v and w
dot_xv = np.dot(x, v)  # Dot product of x and v
dot_xy = np.dot(x, y)  # Dot product of x and y

print("\nDot product of v and w:", dot_vw)
print("\nDot product of x and v:\n", dot_xv)
print("\nDot product of x and y:\n", dot_xy)
```

```

Dot product of v and w: 219

Dot product of x and v:
 [29 77]

Dot product of x and y:
 [[19 22]
 [50 58]]
```

```
print(x)
print("\n")
print(y)
```

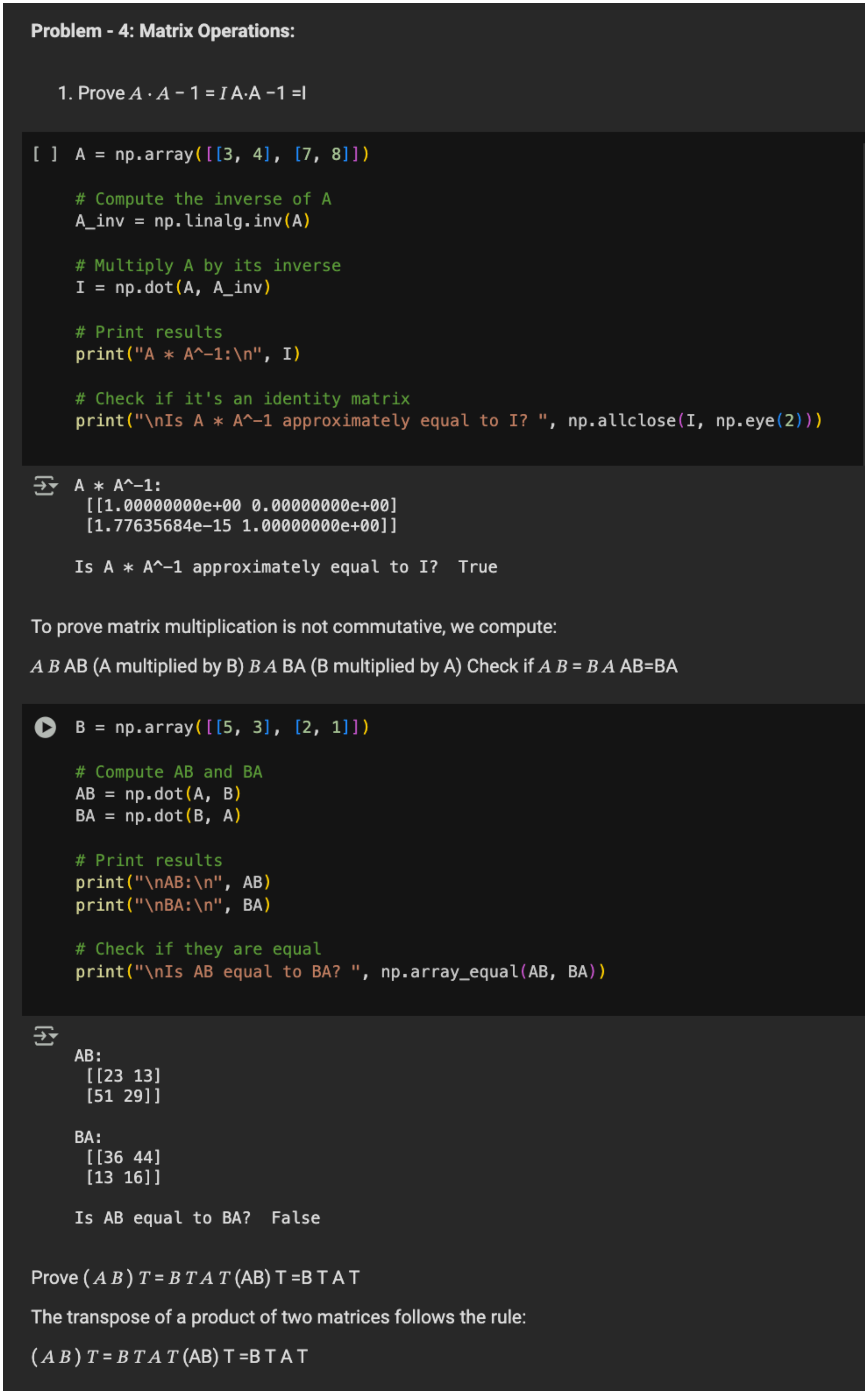
```
[[1 2]
 [3 5]]


[[5 6]
 [7 8]]
```

```
# 6. Concatenate x and y along rows
concat_xy = np.concatenate((x, y), axis=0)
print("\nConcatenation of x and y along rows:\n", concat_xy)

# Concatenate v and w along columns
concat_vw = np.vstack((v, w))
print("\nConcatenation of v and w along columns:\n", concat_vw)
```

```

Concatenation of x and y along rows:
 [[1 2]
 [3 5]
 [5 6]
 [7 8]]

Concatenation of v and w along columns:
 [[ 9 10]
 [11 12]]
```

```
# 7. Concatenating x and v
try:
    concat_xv = np.concatenate((x, v), axis=0)
    print("\nConcatenation of x and v:\n", concat_xv)
except ValueError as e:
    print("\nError while concatenating x and v:", e)
```

```

Error while concatenating x and v: all the input arrays must have same number of dimensions, but the array at index 0 has 2 dimension(s) and the array at index 1 has 1 dimension(s)
```



Problem - 4: Matrix Operations:

1. Prove *A* · *A* − 1 = *I* A·A −1 =I

```
[ ] A = np.array([[3, 4], [7, 8]])

    # Compute the inverse of A
    A_inv = np.linalg.inv(A)

    # Multiply A by its inverse
    I = np.dot(A, A_inv)

    # Print results
    print("A * A^-1:\n", I)

    # Check if it's an identity matrix
    print("\nIs A * A^-1 approximately equal to I? ", np.allclose(I, np.eye(2)))
```

```
A * A^-1:
 [[1.00000000e+00 0.00000000e+00]
 [1.77635684e-15 1.00000000e+00]]

Is A * A^-1 approximately equal to I?  True
```

To prove matrix multiplication is not commutative, we compute:

A B AB (A multiplied by B) *B A* BA (B multiplied by A) Check if *A B* = *B A* AB=BA

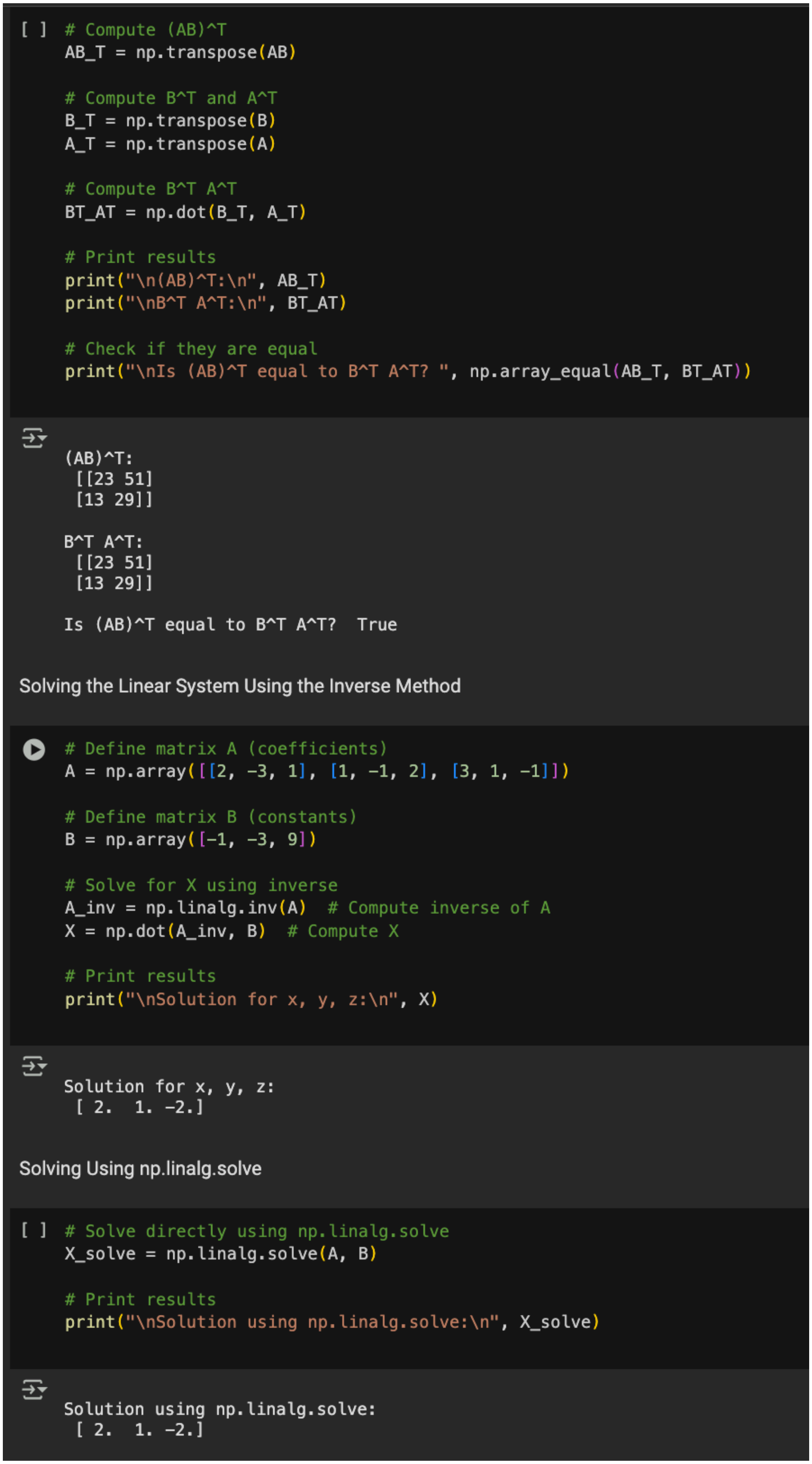
```
B = np.array([[5, 3], [2, 1]])

# Compute AB and BA
AB = np.dot(A, B)
BA = np.dot(B, A)

# Print results
print("\nAB:\n", AB)
print("\nBA:\n", BA)

# Check if they are equal
print("\nIs AB equal to BA? ", np.array_equal(AB, BA))
```

```

AB:
 [[23 13]
 [51 29]]

BA:
 [[36 44]
 [13 16]]

Is AB equal to BA?  False
```

Prove (*A B*) *T* = *B T A T* (AB) T =B T A T

The transpose of a product of two matrices follows the rule:

(*A B*) *T* = *B T A T* (AB) T =B T A T



```
# Compute (AB)^T
AB_T = np.transpose(AB)

# Compute B^T and A^T
B_T = np.transpose(B)
A_T = np.transpose(A)

# Compute B^T A^T
BT_AT = np.dot(B_T, A_T)

# Print results
print("\n(AB)^T:\n", AB_T)
print("\nB^T A^T:\n", BT_AT)

# Check if they are equal
print("\nIs (AB)^T equal to B^T A^T? ", np.array_equal(AB_T, BT_AT))
```

```
(AB)^T:
 [[23 51]
 [13 29]]

B^T A^T:
 [[23 51]
 [13 29]]

Is (AB)^T equal to B^T A^T?  True
```

Solving the Linear System Using the Inverse Method

```
# Define matrix A (coefficients)
A = np.array([[2, -3, 1], [1, -1, 2], [3, 1, -1]])

# Define matrix B (constants)
B = np.array([-1, -3, 9])

# Solve for X using inverse
A_inv = np.linalg.inv(A)  # Compute inverse of A
X = np.dot(A_inv, B)  # Compute X

# Print results
print("\nSolution for x, y, z:\n", X)
```

```
Solution for x, y, z:
 [ 2.  1. -2.]
```

Solving Using np.linalg.solve

```
# Solve directly using np.linalg.solve
X_solve = np.linalg.solve(A, B)

# Print results
print("\nSolution using np.linalg.solve:\n", X_solve)
```

```
Solution using np.linalg.solve:
 [ 2.  1. -2.]
```