sarthakshrestha-worksheet0

March 7, 2025

#4.1 Exercise on Functions:

Task - 1: Create a Python program that converts between different units of measurement.

[ ]:
```python
def convert_units(value, from_unit, to_unit):
    """
    Converts a value from one unit to another.

    Parameters:
        value (float): The value to be converted.
        from_unit (str): The unit to convert from.
        to_unit (str): The unit to convert to.

    Returns:
        float: The converted value.
    """
    # Length conversions
    if from_unit == "m" and to_unit == "ft":
        return value * 3.28084
    elif from_unit == "ft" and to_unit == "m":
        return value / 3.28084

    # Weight conversions
    elif from_unit == "kg" and to_unit == "lbs":
        return value * 2.20462
    elif from_unit == "lbs" and to_unit == "kg":
        return value / 2.20462

    # Volume conversions
    elif from_unit == "L" and to_unit == "gal":
        return value * 0.264172
    elif from_unit == "gal" and to_unit == "L":
        return value / 0.264172

    else:
        raise ValueError("Unsupported conversion type.")

def main():
```

```
    print("Unit Conversion Program")
    print("Supported conversion types:")
    print("1. Length: meters (m) to feet (ft) and vice versa")
    print("2. Weight: kilograms (kg) to pounds (lbs) and vice versa")
    print("3. Volume: liters (L) to gallons (gal) and vice versa")

    try:
        # Prompt user for conversion type
        conversion_type = input("Enter the conversion type (length, weight,␣
↪volume): ").strip().lower()

        if conversion_type not in ["length", "weight", "volume"]:
            print("Error: Invalid conversion type.")
            return

        # Prompt user for input value
        value = float(input("Enter the value to convert: "))

        # Prompt user for units
        if conversion_type == "length":
            from_unit = input("Convert from (m/ft): ").strip().lower()
            to_unit = input("Convert to (m/ft): ").strip().lower()
        elif conversion_type == "weight":
            from_unit = input("Convert from (kg/lbs): ").strip().lower()
            to_unit = input("Convert to (kg/lbs): ").strip().lower()
        elif conversion_type == "volume":
            from_unit = input("Convert from (L/gal): ").strip().lower()
            to_unit = input("Convert to (L/gal): ").strip().lower()

        # Perform conversion
        result = convert_units(value, from_unit, to_unit)
        print(f"{value} {from_unit} is equal to {result:.2f} {to_unit}")

    except ValueError as e:
        print(f"Error: {e}")
    except Exception as e:
        print(f"An unexpected error occurred: {e}")

if __name__ == "__main__":
    main()
```

```
Unit Conversion Program
Supported conversion types:
1. Length: meters (m) to feet (ft) and vice versa
2. Weight: kilograms (kg) to pounds (lbs) and vice versa
3. Volume: liters (L) to gallons (gal) and vice versa
Enter the conversion type (length, weight, volume): length
```

```
Enter the value to convert: 500
Convert from (m/ft): m
Convert to (m/ft): ft
500.0 m is equal to 1640.42 ft
```

Task - 2: Create a Python program that performs various mathematical operations on a list of numbers.

[ ]:
```python
def calculate_sum(numbers):
    """
    Calculate the sum of a list of numbers.

    Parameters:
        numbers (list): A list of numeric values.

    Returns:
        float: The sum of the numbers.
    """
    return sum(numbers)

def calculate_average(numbers):
    """
    Calculate the average of a list of numbers.

    Parameters:
        numbers (list): A list of numeric values.

    Returns:
        float: The average of the numbers.
    """
    return sum(numbers) / len(numbers)

def find_maximum(numbers):
    """
    Find the maximum value in a list of numbers.

    Parameters:
        numbers (list): A list of numeric values.

    Returns:
        float: The maximum value in the list.
    """
    return max(numbers)

def find_minimum(numbers):
    """
    Find the minimum value in a list of numbers.
```

```
    Parameters:
        numbers (list): A list of numeric values.

    Returns:
        float: The minimum value in the list.
    """
    return min(numbers)

def main():
    print("Mathematical Operations Program")
    print("Supported operations:")
    print("1. Sum")
    print("2. Average")
    print("3. Maximum")
    print("4. Minimum")

    try:
        # Prompt user for operation choice
        operation = input("Enter the operation (sum, average, maximum, minimum):
 ↪ ").strip().lower()

        if operation not in ["sum", "average", "maximum", "minimum"]:
            print("Error: Invalid operation.")
            return

        # Prompt user for list of numbers
        input_numbers = input("Enter a list of numbers separated by spaces: ").
 ↪strip()
        if not input_numbers:
            raise ValueError("Empty input. Please enter at least one number.")

        # Convert input to a list of floats
        numbers = list(map(float, input_numbers.split()))

        # Perform the selected operation
        if operation == "sum":
            result = calculate_sum(numbers)
            print(f"The sum of the numbers is: {result}")
        elif operation == "average":
            result = calculate_average(numbers)
            print(f"The average of the numbers is: {result}")
        elif operation == "maximum":
            result = find_maximum(numbers)
            print(f"The maximum number is: {result}")
        elif operation == "minimum":
            result = find_minimum(numbers)
            print(f"The minimum number is: {result}")
```

```
    except ValueError as e:
        print(f"Error: {e}")
    except ZeroDivisionError:
        print("Error: Cannot calculate average of an empty list.")
    except Exception as e:
        print(f"An unexpected error occurred: {e}")

if __name__ == "__main__":
    main()
```

```
Mathematical Operations Program
Supported operations:
1. Sum
2. Average
3. Maximum
4. Minimum
Enter the operation (sum, average, maximum, minimum): sum
Enter a list of numbers separated by spaces: 22 4
The sum of the numbers is: 26.0
```

#4.2 Exercise on List Manipulation:

1. Extract Every Other Element: Write a Python function that extracts every other element from a list, starting from the first element.

[ ]:
```
def extract_every_other(lst):
    """
    Extracts every other element from a list, starting from the first element.

    Parameters:
        lst (list): The input list.

    Returns:
        list: A new list containing every other element from the original list.
    """
    return lst[::2]

# Example usage
input_list = [1, 2, 3, 4, 5, 6]
result = extract_every_other(input_list)
print(result)  # Output: [1, 3, 5]
```

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

2. Slice a Sublist: Write a Python function that returns a sublist from a given list, starting from a specified index and ending at another specified index.

```
[ ]: def get_sublist(lst, start, end):
         """
         Returns a sublist from a given list, starting from a specified index and␣
      ↪ending at another specified index (inclusive).

         Parameters:
             lst (list): The input list.
             start (int): The starting index (inclusive).
             end (int): The ending index (inclusive).

         Returns:
             list: The sublist from start to end (inclusive).
         """
         return lst[start:end + 1]

     # Example usage
     input_list = [1, 2, 3, 4, 5, 6]
     start_index = 2
     end_index = 4
     result = get_sublist(input_list, start_index, end_index)
     print(result)  # Output: [3, 4, 5]
```

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

3. Reverse a List Using Slicing: Write a Python function that reverses a list using slicing.

```
[ ]: def reverse_list(lst):
         """
         Reverses a list using slicing.

         Parameters:
             lst (list): The input list.

         Returns:
             list: The reversed list.
         """
         return lst[::-1]

     # Example usage
     input_list = [1, 2, 3, 4, 5]
     result = reverse_list(input_list)
     print(result)  # Output: [5, 4, 3, 2, 1]
```

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

4. Remove the First and Last Elements: Write a Python function that removes the first and last elements of a list and returns the resulting sublist.

[ ]:
```python
def remove_first_last(lst):
    """
    Removes the first and last elements of a list and returns the resulting␣
 ↪sublist.

    Parameters:
        lst (list): The input list.

    Returns:
        list: The list without the first and last elements.
    """
    return lst[1:-1]

# Example usage
input_list = [1, 2, 3, 4, 5]
result = remove_first_last(input_list)
print(result)  # Output: [2, 3, 4]
```

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

5. Get the First n Elements: Write a Python function that extracts the first n elements from a list.

[ ]:
```python
def get_first_n(lst, n):
    """
    Extracts the first n elements from a list using slicing.

    Parameters:
        lst (list): The input list.
        n (int): The number of elements to extract.

    Returns:
        list: A list containing the first n elements of the input list.
    """
    return lst[:n]

# Example usage
input_list = [1, 2, 3, 4, 5]
n = 3
result = get_first_n(input_list, n)
print(result)  # Output: [1, 2, 3]
```

```
[1, 2, 3]
```

6. Extract Elements from the End: Write a Python function that extracts the last n elements of a list using slicing.

[ ]:
```python
def get_last_n(lst, n):
    """
    Extracts the last n elements from a list using slicing.

    Parameters:
        lst (list): The input list.
        n (int): The number of elements to extract from the end.

    Returns:
        list: A list containing the last n elements of the input list.
    """
    return lst[-n:]

# Example usage
input_list = [1, 2, 3, 4, 5]
n = 2
result = get_last_n(input_list, n)
print(result)  # Output: [4, 5]
```

```
[4, 5]
```

7. Extract Elements in Reverse Order: Write a Python function that extracts a list of elements in reverse order starting from the second-to-last element and skipping one element in between.

[ ]:
```python
def reverse_skip(lst):
    """
    Extracts a list of elements in reverse order starting from the␣
 ↪second-to-last element
    and skipping one element in between.

    Parameters:
        lst (list): The input list.

    Returns:
        list: A new list containing every second element starting from the␣
 ↪second-to-last,
              moving backward.
    """
    return lst[-2::-2]

# Example usage
input_list = [1, 2, 3, 4, 5, 6]
result = reverse_skip(input_list)
print(result)  # Output: [5, 3, 1]
```

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

#4.3 Exercise on Nested List:

1. Flatten a Nested List: Write a Python function that takes a nested list and flattens it into a single list, where all the elements are in a single dimension.

[ ]:
```
def flatten(lst):
    """
    Flattens a nested list into a single list, where all the elements are in a␣
 ↪single dimension.

    Parameters:
        lst (list): The input nested list.

    Returns:
        list: A flattened version of the list.
    """
    flattened_list = []
    for sublist in lst:
        if isinstance(sublist, list):
            flattened_list.extend(flatten(sublist))  # Recursively flatten if␣
 ↪the element is a list
        else:
            flattened_list.append(sublist)  # Append the element if it's not a␣
 ↪list
    return flattened_list

# Example usage
nested_list = [[1, 2], [3, 4], [5]]
result = flatten(nested_list)
print(result)  # Output: [1, 2, 3, 4, 5]
```

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

2. Accessing Nested List Elements: Write a Python function that extracts a specific element from a nested list given its indices.

[ ]:
```
def access_nested_element(lst, indices):
    """
    Extracts an element from a nested list using a list of indices.

    Parameters:
        lst (list): The nested list.
        indices (list): A list of integers representing the indices to access␣
 ↪the element.

    Returns:
        The element at the specified position in the nested list.
    """
    for index in indices:
        lst = lst[index]
```

```
    return lst

# Example usage
nested_list = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
indices = [1, 2]
result = access_nested_element(nested_list, indices)
print(result)  # Output: 6
```

```
6
```

3. Sum of All Elements in a Nested List: Write a Python function that calculates the sum of all the numbers in a nested list (regardless of depth).

[ ]:
```
def sum_nested(lst):
    """
    Calculates the sum of all the numbers in a nested list (regardless of␣
 ↪depth).

    Parameters:
        lst (list): The nested list.

    Returns:
        int/float: The sum of all the elements in the nested list.
    """
    total = 0
    for element in lst:
        if isinstance(element, list):
            total += sum_nested(element)  # Recursively sum nested lists
        else:
            total += element  # Add the element if it's a number
    return total

# Example usage
nested_list = [[1, 2], [3, [4, 5]], 6]
result = sum_nested(nested_list)
print(result)  # Output: 21
```

```
21
```

4. Remove Specific Element from a Nested List: Write a Python function that removes all occurrences of a specific element from a nested list.

[ ]:
```
def remove_element(lst, elem):
    """
    Removes all occurrences of a specific element from a nested list.

    Parameters:
        lst (list): The input nested list.
```

```
        elem: The element to remove.

    Returns:
        list: The modified list with all occurrences of `elem` removed.
    """
    return [[item for item in sublist if item != elem] for sublist in lst]

# Example usage
input_list = [[1, 2], [3, 2], [4, 5]]
element_to_remove = 2
result = remove_element(input_list, element_to_remove)
print(result)  # Output: [[1], [3], [4, 5]]
```

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

5. Find the Maximum Element in a Nested List: Write a Python function that finds the maximum element in a nested list (regardless of depth).

[ ]:
```
def find_max(lst):
    """
    Finds the maximum element in a nested list, regardless of depth.

    Parameters:
        lst (list): The input nested list.

    Returns:
        int or float: The maximum element in the nested list.
    """
    max_value = float('-inf')  # Initialize with negative infinity

    def traverse(nested):
        nonlocal max_value
        for item in nested:
            if isinstance(item, list):
                traverse(item)  # Recursively traverse nested lists
            else:
                if item > max_value:
                    max_value = item  # Update max_value if a larger element is␣
 ↪found

    traverse(lst)
    return max_value

# Example usage
input_list = [[1, 2], [3, [4, 5]], 6]
result = find_max(input_list)
print(result)  # Output: 6
```

6

6. Count Occurrences of an Element in a Nested List: Write a Python function that counts how many times a specific element appears in a nested list.

[ ]:
```python
def count_occurrences(lst, elem):
    """
    Counts how many times a specific element appears in a nested list.

    Parameters:
        lst (list): The input nested list.
        elem (int or any): The element whose occurrences are to be counted.

    Returns:
        int: The number of times elem appears in the nested list.
    """
    count = 0  # Initialize a counter

    def traverse(nested):
        nonlocal count
        for item in nested:
            if isinstance(item, list):
                traverse(item)  # Recursively traverse nested lists
            else:
                if item == elem:
                    count += 1  # Increment count when elem is found

    traverse(lst)
    return count

# Example usage
input_list = [[1, 2], [2, 3], [2, 4]]
element_to_count = 2
result = count_occurrences(input_list, element_to_count)
print(result)  # Output: 3
```

3

7. Flatten a List of Lists of Lists: Write a Python function that flattens a list of lists of lists into a single list, regardless of the depth.

[ ]:
```python
def deep_flatten(lst):
    """
    Flattens a deeply nested list into a single list.

    Parameters:
        lst (list): The input deeply nested list.

    Returns:
```

```
        list: The flattened list.
    """
    flattened = []  # Initialize an empty list to store the flattened elements

    for item in lst:
        if isinstance(item, list):  # Check if the item is a list
            flattened.extend(deep_flatten(item))  # Recursively flatten the␣
↪nested list and extend the result
        else:
            flattened.append(item)  # If the item is not a list, add it␣
↪directly to the result list

    return flattened

# Example usage
input_list = [[[1, 2], [3, 4]], [[5, 6], [7, 8]]]
result = deep_flatten(input_list)
print(result)  # Output: [1, 2, 3, 4, 5, 6, 7, 8]
```

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

8. Nested List Average: Write a Python function that calculates the average of all elements in a nested list.

[ ]:
```
def average_nested(lst):
    """
    Calculates the average of all elements in a nested list.

    Parameters:
        lst (list): The input nested list.

    Returns:
        float: The average of all the elements in the nested list.
    """
    total_sum = 0  # Initialize a variable to store the sum of all elements
    total_count = 0  # Initialize a variable to count the total number of␣
↪elements

    def traverse(nested):
        nonlocal total_sum, total_count
        for item in nested:
            if isinstance(item, list):  # If the item is a list, recursively␣
↪traverse it
                traverse(item)
            else:
                total_sum += item  # Add the item to the sum
                total_count += 1  # Increment the count of elements
```

```
    traverse(lst)

    # Calculate the average by dividing the total sum by the total count
    if total_count == 0:  # Avoid division by zero
        return 0
    return total_sum / total_count

# Example usage
input_list = [[1, 2], [3, 4], [5, 6]]
result = average_nested(input_list)
print(result)  # Output: 3.5
```

```
3.5
```

#10.1 Basic Vector and Matrix Operation with Numpy.

Problem - 1: Array Creation:

[ ]:
```
import numpy as np

# Task 1: Initialize an empty array with size 2x2
empty_array = np.empty((2, 2))
print("Empty array (2x2):")
print(empty_array)

# Task 2: Initialize an all-ones array with size 4x2
ones_array = np.ones((4, 2))
print("\nAll ones array (4x2):")
print(ones_array)

# Task 3: Return a new array filled with the fill value
filled_array = np.full((3, 3), 7, dtype=int)
print("\nArray filled with 7 (3x3):")
print(filled_array)

# Task 4: Return a new array of zeros with same shape and type as the given␣
 ↪array
existing_array = np.array([[1, 2], [3, 4]])
zeros_like_array = np.zeros_like(existing_array)
print("\nArray of zeros with same shape as existing array:")
print(zeros_like_array)

# Task 5: Return a new array of ones with same shape and type as the given array
ones_like_array = np.ones_like(existing_array)
print("\nArray of ones with same shape as existing array:")
print(ones_like_array)

# Task 6: Convert list to NumPy array
```

```
new_list = [1, 2, 3, 4]
numpy_array = np.array(new_list)
print("\nConverted NumPy array from list:")
print(numpy_array)
```

```
Empty array (2x2):
[[3.47695828e-316 0.00000000e+000]
 [2.12199579e-314 6.36598737e-314]]

All ones array (4x2):
[[1. 1.]
 [1. 1.]
 [1. 1.]
 [1. 1.]]

Array filled with 7 (3x3):
[[7 7 7]
 [7 7 7]
 [7 7 7]]

Array of zeros with same shape as existing array:
[[0 0]
 [0 0]]

Array of ones with same shape as existing array:
[[1 1]
 [1 1]]

Converted NumPy array from list:
[1 2 3 4]
```

Problem - 2: Array Manipulation: Numerical Ranges and Array indexing:

[ ]:
```
import numpy as np

# Task 1: Create an array with values ranging from 10 to 49
array_10_to_49 = np.arange(10, 50)
print("Array from 10 to 49:")
print(array_10_to_49)

# Task 2: Create a 3x3 matrix with values ranging from 0 to 8
matrix_3x3 = np.arange(9).reshape(3, 3)
print("\n3x3 Matrix with values from 0 to 8:")
print(matrix_3x3)

# Task 3: Create a 3x3 identity matrix
identity_matrix = np.eye(3)
print("\n3x3 Identity Matrix:")
```

```
print(identity_matrix)

# Task 4: Create a random array of size 30 and find the mean of the array
random_array = np.random.random(30)
mean_value = random_array.mean()
print("\nRandom Array of size 30:")
print(random_array)
print(f"Mean of the array: {mean_value}")

# Task 5: Create a 10x10 array with random values and find the minimum and␣
 ↪maximum values
random_10x10 = np.random.random((10, 10))
min_value = random_10x10.min()
max_value = random_10x10.max()
print("\n10x10 Random Array:")
print(random_10x10)
print(f"Minimum value: {min_value}, Maximum value: {max_value}")

# Task 6: Create a zero array of size 10 and replace the 5th element with 1
zero_array = np.zeros(10)
zero_array[4] = 1
print("\nZero Array with 5th element replaced with 1:")
print(zero_array)

# Task 7: Reverse an array arr = [1, 2, 0, 0, 4, 0]
arr = np.array([1, 2, 0, 0, 4, 0])
reversed_arr = arr[::-1]
print("\nReversed Array:")
print(reversed_arr)

# Task 8: Create a 2D array with 1 on the border and 0 inside
border_array = np.ones((5, 5))
border_array[1:-1, 1:-1] = 0
print("\n2D Array with 1 on the border and 0 inside:")
print(border_array)

# Task 9: Create an 8x8 matrix and fill it with a checkerboard pattern
checkerboard_pattern = np.zeros((8, 8))
checkerboard_pattern[1::2, ::2] = 1  # Fill alternate positions with 1
checkerboard_pattern[::2, 1::2] = 1  # Fill alternate positions with 1
print("\n8x8 Checkerboard Pattern:")
print(checkerboard_pattern)
```

```
Array from 10 to 49:
[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]
```

```
3x3 Matrix with values from 0 to 8:
[[0 1 2]
 [3 4 5]
 [6 7 8]]

3x3 Identity Matrix:
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]

Random Array of size 30:
[0.89598712 0.97470465 0.9549587  0.26093627 0.42989871 0.75484546
 0.99349741 0.38932325 0.4576409  0.8251537  0.84345272 0.34496704
 0.91185882 0.53751927 0.4237168  0.58934062 0.20762817 0.08831161
 0.94399667 0.76598509 0.31703074 0.57342813 0.43459599 0.36503932
 0.1763216  0.40912289 0.58327549 0.84232978 0.79640031 0.43283182]
Mean of the array: 0.5841366363350037

10x10 Random Array:
[[0.46848588 0.43344872 0.0919746  0.20931367 0.69476786 0.66930291
  0.67785107 0.36425701 0.042375   0.2906841 ]
 [0.14483943 0.92084452 0.39714222 0.85053663 0.16338179 0.88540291
  0.84745496 0.68454811 0.30242743 0.71547249]
 [0.13880941 0.51887243 0.8895643  0.18807477 0.12028063 0.16180169
  0.69150668 0.67548226 0.49615375 0.51752095]
 [0.7417203  0.71432573 0.41785118 0.16317433 0.31613005 0.60651158
  0.32886023 0.74708538 0.67784893 0.16929819]
 [0.91932835 0.67890481 0.97640292 0.55692573 0.36801643 0.20950605
  0.32951244 0.55540494 0.85826361 0.31693699]
 [0.8728047  0.26556756 0.4334198  0.78875894 0.74344837 0.95153328
  0.11283053 0.86256555 0.91098683 0.91299796]
 [0.93407311 0.08908077 0.60732356 0.90442359 0.29785583 0.43672457
  0.73286994 0.78071983 0.47810922 0.04479622]
 [0.36387279 0.96146902 0.93324227 0.13957841 0.58683799 0.37573172
  0.94833534 0.92861695 0.16710311 0.16014549]
 [0.47403678 0.82583058 0.48142398 0.89449479 0.0129244  0.32841105
  0.54113342 0.92074402 0.83060267 0.91469457]
 [0.81039437 0.59099395 0.79319585 0.46514288 0.84747802 0.33193031
  0.9067252  0.12581532 0.61178473 0.469329  ]]
Minimum value: 0.012924395049267234, Maximum value: 0.9764029173411379

Zero Array with 5th element replaced with 1:
[0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]

Reversed Array:
[0 4 0 0 2 1]

2D Array with 1 on the border and 0 inside:
```

```
[[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.]]

8x8 Checkerboard Pattern:
[[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:

[ ]:
```python
import numpy as np

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

# 1. Add the two arrays
add_result = x + y
print("1. Addition of x and y:")
print(add_result)

# 2. Subtract the two arrays
subtract_result = x - y
print("\n2. Subtraction of x and y:")
print(subtract_result)

# 3. Multiply the arrays with any integers of your choice
multiply_x = x * 2
multiply_y = y * 3
print("\n3. Multiplication of x by 2 and y by 3:")
print(multiply_x)
print(multiply_y)

# 4. Find the square of each element of the array
square_x = np.square(x)
square_y = np.square(y)
print("\n4. Square of each element in x and y:")
print(square_x)
```

```
print(square_y)

# 5. Find the dot product between v and w, x and v, x and y
dot_vw = np.dot(v, w)
dot_xv = np.dot(x, v)
dot_xy = np.dot(x, y)
print("\n5. Dot Products:")
print("v . w =", dot_vw)
print("x . v =", dot_xv)
print("x . y =", dot_xy)

# 6. Concatenate x and y along rows and v and w along columns
concat_xy_row = np.concatenate((x, y), axis=0)  # Concatenate along rows
concat_vw_col = np.concatenate((v.reshape(-1, 1), w.reshape(-1, 1)), axis=1)  #␣
 ↪Concatenate v and w along columns
print("\n6. Concatenate x and y along rows and v and w along columns:")
print(concat_xy_row)
print(concat_vw_col)

# 7. Concatenate x and v
try:
    concat_xv = np.concatenate((x, v), axis=0)
    print("\n7. Concatenate x and v:")
    print(concat_xv)
except ValueError as e:
    print("\n7. Concatenation of x and v failed with error:", e)
```

```
1. Addition of x and y:
[[ 6  8]
 [10 13]]

2. Subtraction of x and y:
[[-4 -4]
 [-4 -3]]

3. Multiplication of x by 2 and y by 3:
[[ 2  4]
 [ 6 10]]
[[15 18]
 [21 24]]

4. Square of each element in x and y:
[[ 1  4]
 [ 9 25]]
[[25 36]
 [49 64]]
```

```
5. Dot Products:
v . w = 219
x . v = [29 77]
x . y = [[19 22]
 [50 58]]

6. Concatenate x and y along rows and v and w along columns:
[[1 2]
 [3 5]
 [5 6]
 [7 8]]
[[ 9 11]
 [10 12]]

7. Concatenation of x and v failed with error: 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:

[ ]:
```python
import numpy as np

# Matrices A and B
A = np.array([[3, 4], [7, 8]])
B = np.array([[5, 3], [2, 1]])

# Identity Matrix I (for A^-1)
I = np.eye(2)  # 2x2 identity matrix

# 1. Prove A * A^(-1) = I
A_inv = np.linalg.inv(A)
result_1 = np.dot(A, A_inv)
print("1. A * A^(-1):")
print(result_1)

# 2. Prove AB  BA
AB = np.dot(A, B)
BA = np.dot(B, A)
print("\n2. AB != BA:")
print("AB =\n", AB)
print("BA =\n", BA)

# 3. Prove (AB)^T = B^T * A^T
AB_T = np.transpose(AB)
B_T_A_T = np.dot(np.transpose(B), np.transpose(A))
print("\n3. (AB)^T = B^T * A^T:")
print("AB^T =\n", AB_T)
print("B^T * A^T =\n", B_T_A_T)
```

```python
# 4. Solve the System of Linear Equations Using Inverse Method

# Coefficient matrix A (3x3)
A_sys = np.array([[2, -3, 1], [1, -1, 2], [3, 1, -1]])

# Constant matrix B (3x1)
B_sys = np.array([-1, -3, 9])

# Solve for X using the inverse of A (X = A_inv * B)
A_inv_sys = np.linalg.inv(A_sys)

# Solve the system
X = np.dot(A_inv_sys, B_sys)

print("\n4. Solution to the system of linear equations (using Inverse method):")
print(f"x = {X[0]}, y = {X[1]}, z = {X[2]}")
```

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

2. AB != BA:
AB =
 [[23 13]
 [51 29]]
BA =
 [[36 44]
 [13 16]]

3. (AB)^T = B^T * A^T:
AB^T =
 [[23 51]
 [13 29]]
B^T * A^T =
 [[23 51]
 [13 29]]

4. Solution to the system of linear equations (using Inverse method):
x = 2.0, y = 1.0, z = -2.0
```

#10.2 Experiment: How Fast is Numpy?

1. Element-wise Addition:

• Using Python Lists, perform element-wise addition of two lists of size 1, 000, 000. Measure and Print the time taken for this operation.

[ ]:
```python
import time

# Create two lists of size 1,000,000
list1 = [i for i in range(1, 1000001)]
list2 = [i for i in range(1000001, 2000001)]

# Measure the time before the operation
start_time = time.time()

# Perform element-wise addition using list comprehension
result = [list1[i] + list2[i] for i in range(1000000)]

# Measure the time after the operation
end_time = time.time()

# Print the time taken
print(f"Time taken for element-wise addition: {end_time - start_time} seconds")
```

```
Time taken for element-wise addition: 0.13124370574951172 seconds
```

Using Numpy Arrays, Repeat the calculation and measure and print the time taken for this operation.

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

# Create two NumPy arrays of size 1,000,000
array1 = np.arange(1, 1000001)
array2 = np.arange(1000001, 2000001)

# Measure the time before the operation
start_time = time.time()

# Perform element-wise addition using NumPy
result = array1 + array2

# Measure the time after the operation
end_time = time.time()

# Print the time taken
print(f"Time taken for element-wise addition with NumPy: {end_time -␣
↪start_time} seconds")
```

```
Time taken for element-wise addition with NumPy: 0.009052038192749023 seconds
```

2. Element-wise Multiplication

• Using Python Lists, perform element-wise multiplication of two lists of size 1, 000, 000. Measure and Print the time taken for this operation.

[ ]:
```python
import time

# Create two Python lists of size 1,000,000
list1 = list(range(1, 1000001))
list2 = list(range(1000001, 2000001))

# Measure the time before the operation
start_time = time.time()

# Perform element-wise multiplication using list comprehension
result = [list1[i] * list2[i] for i in range(len(list1))]

# Measure the time after the operation
end_time = time.time()

# Print the time taken
print(f"Time taken for element-wise multiplication with Python lists: {end_time␣
 ↪- start_time} seconds")
```

```
Time taken for element-wise multiplication with Python lists:
0.08447122573852539 seconds
```

- Using Numpy Arrays, Repeat the calculation and measure and print the time taken for this operation.

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

# Create two NumPy arrays of size 1,000,000
array1 = np.arange(1, 1000001)
array2 = np.arange(1000001, 2000001)

# Measure the time before the operation
start_time = time.time()

# Perform element-wise multiplication using NumPy
result = array1 * array2

# Measure the time after the operation
end_time = time.time()

# Print the time taken
print(f"Time taken for element-wise multiplication with NumPy arrays: {end_time␣
 ↪- start_time} seconds")
```

```
Time taken for element-wise multiplication with NumPy arrays:
0.005049467086791992 seconds
```

3. Dot Product

• Using Python Lists, compute the dot product of two lists of size 1, 000, 000. Measure and Print the time taken for this operation.

[ ]:
```
import time

# Create two lists of size 1,000,000
list1 = [i for i in range(1, 1000001)]
list2 = [i for i in range(1000001, 2000001)]

# Measure the time before the operation
start_time = time.time()

# Compute the dot product using Python lists
dot_product = sum(a * b for a, b in zip(list1, list2))

# Measure the time after the operation
end_time = time.time()

# Print the time taken
print(f"Time taken for dot product using Python lists: {end_time - start_time}␣
 ↪seconds")
```

```
Time taken for dot product using Python lists: 0.08272886276245117 seconds
```

• Using Numpy Arrays, Repeat the calculation and measure and print the time taken for this operation.

[ ]:
```
import numpy as np
import time

# Create two NumPy arrays of size 1,000,000
array1 = np.arange(1, 1000001)
array2 = np.arange(1000001, 2000001)

# Measure the time before the operation
start_time = time.time()

# Compute the dot product using NumPy arrays
dot_product = np.dot(array1, array2)

# Measure the time after the operation
end_time = time.time()

# Print the time taken
print(f"Time taken for dot product using NumPy arrays: {end_time - start_time}␣
 ↪seconds")
```

```
Time taken for dot product using NumPy arrays: 0.0020294189453125 seconds
```

4. Matrix Multiplication

• Using Python lists, perform matrix multiplication of two matrices of size 1000x1000. Mea- sure and print the time taken for this operation.

[ ]:
```
import time

# Function to perform matrix multiplication using Python lists
def matrix_multiply(A, B):
    # Initialize the result matrix with zeros
    result = [[0] * len(B[0]) for _ in range(len(A))]

    # Perform matrix multiplication
    for i in range(len(A)):
        for j in range(len(B[0])):
            for k in range(len(B)):
                result[i][j] += A[i][k] * B[k][j]

    return result

# Generate two 1000x1000 matrices with random values
matrix_A = [[1] * 1000 for _ in range(1000)]  # 1000x1000 matrix filled with 1s
matrix_B = [[1] * 1000 for _ in range(1000)]  # 1000x1000 matrix filled with 1s

# Measure the time before the operation
start_time = time.time()

# Perform matrix multiplication
result_matrix = matrix_multiply(matrix_A, matrix_B)

# Measure the time after the operation
end_time = time.time()

# Print the time taken
print(f"Time taken for matrix multiplication using Python lists: {end_time -␣
 ↪start_time} seconds")
```

```
Time taken for matrix multiplication using Python lists: 135.38863229751587
seconds
```

• Using NumPy arrays, perform matrix multiplication of two matrices of size 1000x1000. Measure and print the time taken for this operation.

[ ]:
```
import numpy as np
import time

# Generate two 1000x1000 matrices with random values
matrix_A = np.ones((1000, 1000))  # 1000x1000 matrix filled with 1s
matrix_B = np.ones((1000, 1000))  # 1000x1000 matrix filled with 1s
```

```
# Measure the time before the operation
start_time = time.time()

# Perform matrix multiplication using NumPy
result_matrix = np.dot(matrix_A, matrix_B)

# Measure the time after the operation
end_time = time.time()

# Print the time taken
print(f"Time taken for matrix multiplication using NumPy: {end_time -␣
 ↪start_time} seconds")
```

```
Time taken for matrix multiplication using NumPy: 0.07257533073425293 seconds
```