Question 1:

Define the following metrics and perform the following operations:

i) Write a Python program using Python Lists
ii) Write a Python program using NumPy

Matrices:

- **Matrix A:** [[3.7827, 3.3454, 3.2341], [2.2122, 3.5678, 3.9087], [1.1234, 2.8934, 5.9087]]
- **Matrix B:** [[3.1234, 3.0987, 3.1234], [2.1111, 3.2222, 3.3333], [1.0987, 1.3456, 5.1234]]
- **Matrix C:** [[3.1243, 3.0989, 3.1256], [2.6721, 3.6785, 3.9017], [1.1254, 2.8956, 5.9187]]

---

i): Write a Python program using Python Lists

In this approach, an empty matrix matrix_c is initialized with zeros, and then a nested loop is used to perform element-wise addition of matrix_a and matrix_b.

The results are stored in matrix_c.

Code:

```
# Defining Matrix A, B
matrix_a = [
    [3.7827, 3.3454, 3.2341],
    [2.2122, 3.5678, 3.9087],
    [1.1234, 2.8934, 5.9087]
]

matrix_b = [
    [3.1234, 3.0987, 3.1234],
    [2.1111, 3.2222, 3.3333],
    [1.0987, 1.3456, 5.1234]
]
```

```
# Define matrix_c as an empty matrix with the same dimensions as
matrix_a and matrix_b
matrix_c = [[0, 0, 0], [0, 0, 0], [0, 0, 0]]

# Perform element-wise addition
for i in range(len(matrix_a)):  # iterate over rows
    for j in range(len(matrix_a[0])):  # iterate over columns
        matrix_c[i][j] = matrix_a[i][j] + matrix_b[i][j]

# Display the result
print("Matrix C (A + B):")
for row in matrix_c:
    print(row)
```

Output:

```
Matrix C (A + B):
[6.9061, 6.4441, 6.3575]
[4.3233, 6.79, 7.242]
[2.2221, 4.239, 11.0321]
```

---

ii): Write a Python program using NumPy

Using the NumPy's built-in np.add() function to perform matrix addition.

Code:

```
import numpy as np

# Defining Matrix A and B using np.array
matrix_a = np.array([
    [3.7827, 3.3454, 3.2341],
    [2.2122, 3.5678, 3.9087],
    [1.1234, 2.8934, 5.9087]
])

matrix_b = np.array([
    [3.1234, 3.0987, 3.1234],
```

```
    [2.1111, 3.2222, 3.3333],
    [1.0987, 1.3456, 5.1234]
])

# Perform matrix addition
matrix_c = np.add(matrix_a, matrix_b)

# Display the result
print("Matrix C (A + B):")
print(matrix_c)
```

Output:

```
Matrix C (A + B):
[[ 6.9061  6.4441  6.3575]
 [ 4.3233  6.79    7.242 ]
 [ 2.2221  4.239  11.0321]]
```

Question 2:

Write a Python Program and perform the following operations:

I. Create a List {1225, 4986, 6789, 7890, 2345, 6783, 0987, 1234, 8765, 3456}
II. Iterate using a for loop
III. Iterate using a for loop and range
IV. List Comprehension
V. Enumerate
VI. Iter function and next function
VII. Map function
VIII. Using zip
IX. Using NumPy Module

I: Create a List

Code:

Here, the list is created using the list() constructor, which converts a tuple into a list.

```
# List creation using a list constructor
my_list = list((1225, 4986, 6789, 7890, 2345, 6783, 987, 1234, 8765,
3456))
print("List:", my_list)
```

Output:

```
List: [1225, 4986, 6789, 7890, 2345, 6783, 987, 1234, 8765, 3456]
```

II: Iterate using a for loop

Code:

Using a traditional for loop, a list comprehension is used here to iterate through the list and print each element.

```
# Using a list comprehension for iteration and printing
[print(item) for item in my_list]
```

Output:

```
1225
4986
6789
7890
2345
6783
987
1234
8765
3456
```

III: Iterate using a for loop and range

Code:

This code loops each element is accessed using range(len(my_list)) along with an index to print a more detailed message that includes the index of the element.

```
# Using enumerate with for loop and range for printing
for i in range(len(my_list)):
    print(f"Element at index {i} is {my_list[i]}")
```

Output:

```
Element at index 0 is 1225
Element at index 1 is 4986
Element at index 2 is 6789
Element at index 3 is 7890
Element at index 4 is 2345
Element at index 5 is 6783
Element at index 6 is 987
Element at index 7 is 1234
Element at index 8 is 8765
Element at index 9 is 3456
```

IV: List Comprehension

Code:

This code creates a new list where 100 is subtracted from each element of my_list.

It shows how list comprehensions can be used for more than just basic operations like squaring.

```
# Create a new list by subtracting a constant from each element
modified_list = [x - 100 for x in my_list]
print("Modified List (subtracting 100):", modified_list)
```

Output:

```
Modified List (subtracting 100): [1125, 4886, 6689, 7790, 2245, 6683,
887, 1134, 8665, 3356]
```

V: Enumerate

Code:

This enumerate code is combined with a condition to print only elements at even indices, demonstrating the versatility of enumerate in iteration.

```
# Using enumerate with condition to print only even indices
for index, item in enumerate(my_list):
    if index % 2 == 0:
        print(f"Index {index}: {item} (even index)")
```

Output:

```
Index 0: 1225 (even index)
Index 2: 6789 (even index)
Index 4: 2345 (even index)
Index 6: 987 (even index)
Index 8: 8765 (even index)
```

VI: Iter function and next function

Code:

This code uses an infinite loop with next() to fetch elements from the iterator until StopIteration is raised, and how iteration can be controlled manually.

```
list_iter = iter(my_list)

try:
    while True:
# Keep fetching elements until StopIteration is raised
```

```
        print(next(list_iter))
except StopIteration:
    print("End of list reached.")
```

Output:

```
1225
4986
6789
7890
2345
6783
987
1234
8765
3456
End of list reached.
```

VII: Map function

Code:

This code squares each element in the list instead of doubling them.

```
# Map function with squaring each element
squared_list = list(map(lambda x: x**2, my_list))
print("Squared List:", squared_list)
```

Output:

```
Squared List: [1500625, 24860296, 46090041, 62252100, 5490025,
45911289, 974169, 1522756, 76861625, 11953336]
```

VIII: Using zip

Code:

Taking the two lists list1 and list2 are of different lengths.

zip truncates the result to the length of the shorter list.

```
list1 = [1, 2, 3]
list2 = [4, 5, 6, 7]
zipped_list = list(zip(list1, list2))
print("Zipped List:", zipped_list)
```

Output:

```
Zipped List: [(1, 4), (2, 5), (3, 6)]
```

IX: Using NumPy Module

Code:

This code doubles each element using NumPy's array operations.

```
import numpy as np

np_array = np.array(my_list) * 2  # Doubling each element using NumPy
print("Doubled NumPy Array:", np_array)
```

Output:

```
Doubled NumPy Array: [ 2450  9972 13578 15780  4690 13566  1974  2468
17530  6912]
```

Question 3:

For a List A, B, C, D, E write a Python program to compute all the combinations and permutations

Code:

This code generates permutations of length 3 and combinations of length 4, demonstrating how the length parameter can be adjusted to customize the output.

```
from itertools import permutations, combinations

# List A, B, C, D, E
elements = ['A', 'B', 'C', 'D', 'E']

# Computing permutations of length 3
perm = list(permutations(elements, 3))
print("Permutations (Length 3):")
for p in perm:
    print(p)

# Computing combinations of length 4
comb = list(combinations(elements, 4))
print("\nCombinations (Length 4):")
for c in comb:
    print(c)
```

Output:

```
Permutations (Length 3):
('A', 'B', 'C')
('A', 'B', 'D')
...

Combinations (Length 4):
('A', 'B', 'C', 'D')
('A', 'B', 'C', 'E')
...
```

Question 4:

Using the same list, use itertools to compute permutations and combinations

Code:

This code computes all possible permutations of the list and combinations of length 3.

```
from itertools import permutations, combinations
```

```
# List A, B, C, D, E
elements = ['A', 'B', 'C', 'D', 'E']

# Computing all permutations
perm = list(permutations(elements))
print("All Permutations:")
for p in perm:
    print(p)

# Computing combinations of length 3
comb = list(combinations(elements, 3))
print("\nCombinations (Length 3):")
for c in comb:
    print(c)
```

Output:

```
All Permutations:
('A', 'B', 'C', 'D', 'E')
('A', 'B', 'C', 'E', 'D')
...

Combinations (Length 3):
('A', 'B', 'C')
('A', 'B', 'D')
...
```