**Lab Exercise 4- : List Comprehensions, Generator Expressions, and Generator Functions**

In this lab, you’ll explore **list comprehensions**, **generator expressions**, and **generator functions** to work with iterables efficiently and concisely.

**1. List Comprehensions**

List comprehensions provide a concise way to create lists using an iterable, optionally applying conditions.

**Task 1: Basic List Comprehension**

**Task**: Given a list of numbers, use list comprehension to create a new list containing only the squares of the even numbers.

# List of numbers

numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# List comprehension to get squares of even numbers

squared\_evens = [x\*\*2 for x in numbers if x % 2 == 0]

# Print the result

print(squared\_evens)

**Exercise Breakdown**:

* The list comprehension [x\*\*2 for x in numbers if x % 2 == 0] applies a condition (if x % 2 == 0) to select even numbers and then squares them.

**2. Nested List Comprehension**

Nested list comprehensions allow you to handle more complex structures, such as 2D arrays or matrices.

**Task 2: Flatten a 2D Matrix**

**Task**: Given a 2D list (matrix), use a nested list comprehension to flatten it into a 1D list.

# 2D matrix

matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

# Flatten the matrix using nested list comprehension

flattened = [item for row in matrix for item in row]

# Print the result

print(flattened)

**Exercise Breakdown**:

* The comprehension [item for row in matrix for item in row] iterates over each row in the matrix, then each element (item) in the row to flatten the matrix.

**3. Generator Expressions**

Generator expressions are similar to list comprehensions, but they create an iterator instead of a list, which is more memory efficient for large datasets.

**Task 3: Creating a Generator Expression**

**Task**: Use a generator expression to generate cubes of numbers from 1 to 10 and print them one by one using a for loop.

# Generator expression to generate cubes

cubes = (x\*\*3 for x in range(1, 11))

# Iterate over the generator and print each cube

for cube in cubes:

print(cube)

**Exercise Breakdown**:

* The generator expression (x\*\*3 for x in range(1, 11)) creates an iterator that generates cubes of numbers from 1 to 10. Unlike list comprehensions, it doesn't generate all values at once, saving memory.

**4. Generator Functions**

Generator functions are functions that yield values one at a time, allowing for efficient iteration over large datasets.

**Task 4: Creating a Simple Generator Function**

**Task**: Write a generator function fibonacci that yields the Fibonacci sequence up to a certain number of terms.

# Generator function to yield Fibonacci sequence

def fibonacci(n):

a, b = 0, 1

for \_ in range(n):

yield a

a, b = b, a + b

# Test the generator function

fib\_gen = fibonacci(10)

# Print Fibonacci sequence

for num in fib\_gen:

print(num)

**Exercise Breakdown**:

* The fibonacci(n) generator function uses yield to return the next number in the sequence, one at a time. It generates the first n Fibonacci numbers efficiently without storing them all in memory.