**1. How enumerate function works**

->The enumerate() function in Python is used to iterate over a sequence (such as a list, tuple or string) while keeping track of the index of each item during iteration. It returns an iterator of pairs containing the index and the corresponding item from the sequence.

x = ["eat", "sleep", "repeat"]

enumerate(x,start=0)

print(x)

**Result:**

['eat', 'sleep', 'repeat']

l1: This is the sequence list that you want to iterate over and enumerate.

start: This is an optional parameter that specifies the starting index value. The default value is 0.

# printing the tuples in object directly

for ele in enumerate(x):

print (ele)

**Result:**

(0, 'eat')

(1, 'sleep')

(2, 'repeat')

# changing index and printing separately

for count, ele in enumerate(x, 100):

print (count, ele)

**Result:**

100 eat

101 sleep

102 repeat

# getting desired output from tuple

for count, ele in enumerate(x):

print(count)

print(ele)

**Result:**

0

eat

1

sleep

2

repeat

**2.how zip function function works internally**

->The zip() function in Python is used to combine multiple sequences (like lists, tuples, etc.) into a single iterable of tuples. Each tuple contains elements from the input sequences at the corresponding index. It stops creating tuples when the shortest input sequence is exhausted. Internally, the zip() function works by creating an iterator that produces tuples containing elements from the input sequences.

names = ['Alice', 'Bob', 'Charlie',]

ages = [25, 30, 22]

zipped = zip(names, ages)

for item in zipped:

print(item)

**Result:**

('Alice',25)

('Bob',30)

('Charlie',22)

**3.what happens if size of any list is not equal to each other in zip?**

->It stops creating tuples when the shortest input sequence is exhausted.

names = ['Alice', 'Bob', 'Charlie', 'Rita']

ages = [25, 30, 22]

zipped = zip(names, ages)

for item in zipped:

print(item)

**Result:**

('Alice',25)

('Bob',30)

('Charlie',22)

**4.execution time of list comprehension and tradition approach**

->list comprehension takes less time than the tradition approach. For Example:

import time

# For loop

start\_time = time.time()

numbers = [i for i in range(0,1000)]

result = []

for num in numbers:

if num % 2 == 1:

result.append(num \*\* 2)

end\_time = time.time()

for\_loop\_time = end\_time - start\_time

# List comprehension

start\_time = time.time()

result = [num \*\* 2 for num in numbers if num % 2 == 1]

end\_time = time.time()

list\_comp\_time = end\_time - start\_time

print("For loop time:", for\_loop\_time)

print("List comprehension time:", list\_comp\_time)

print("Difference:", for\_loop\_time - list\_comp\_time)

**Result:**

For loop time: 0.0009965896606445312

List comprehension time: 0.0

Difference: 0.0009965896606445312

Here, List-comprehension avoids the overhead of calling result.append() and uses the optimized C code under the hood of list comprehension.

**5. Memory size of each index in a list according to different type of data**

-> Calculation:

Memory size = Size of Integer + List Overhead + Pointer Size

For Integer value, each index in a list:

On a 32-bit system, this would be roughly 4 bytes (integer) + 24 bytes (overhead and pointer) = 28 bytes per index.

On a 64-bit system, this would be roughly 8 bytes (integer) + 24 bytes (overhead and pointer) = 32 bytes per index.

For Float value, each index in a list:

On a 32-bit system, this would be roughly 12 bytes (integer) + 24 bytes (overhead and pointer) = 36 bytes per index.

On a 64-bit system, this would be roughly 24 bytes (integer) + 24 bytes (overhead and pointer) = 48 bytes per index.