Enumerate Function:

The *enumerate()* function in Python helps us go through a list of things (like items in a shopping list) and keeps track of their order. Imagine we have a list of fruits: apple, banana, and cherry. *enumerate()* let us loop through this list while also knowing the number or position of each fruit.

Working Steps of *enumerate():*

We give *enumerate()* the list of things us want to go through. *enumerate()* sets up a special counter that starts from 0. As we start going through the list using a loop, *enumerate()* gives us the current thing (fruit) and its position (counter value). After we are done with one thing, the counter goes up by 1, so we are ready for the next thing. This continues until we have gone through all the things in the list. When we have finished, the loop stops.

For example:

```
fruits = ['apple', 'banana', 'cherry']

for index, fruit in enumerate(fruits):
    print(index, fruit)
```

Output:

```
0 apple
1 banana
2 cherry
```

But *enumerate()* internally works as below code. Where indexing of list has been used.

```
fruits = ['apple', 'banana', 'cherry']
length = len(fruits)

for i in range(length):
    fruit = fruits[i]
    print(i, fruit)
```

Output:

```
0 apple
1 banana
2 cherry
```

Zip Function:

As per above discussion , this code also works by indexing method with for loop. It also can be done by using next().

```
names = ['Alice', 'Bob', 'Charlie']
scores = [85, 92, 78]
fruits = ['apple', 'banana', 'cherry']

for i in range(len(names)):
    name = names[i]
    score = scores[i]
    fruit = fruits[i]
    print(name, score, fruit)
```

Output:

```
Alice 85 apple
Bob 92 banana
Charlie 78 cherry
```

What happens if size of any list is not equal to each other in zip

It will throw an ‘IndexError’ by printing a message “list index out of range”

The execution time of list comprehensions versus traditional approaches can vary depending on f actors like the complexity of the operation being performed, the size of the data, and the specific implementation details. In general, list comprehensions tend to be more concise and, in some cas es, more efficient than traditional approaches using loops, especially for simple operations.

```
import timeit

# Using list comprehension
numbers = list(range(1, 10001))
start_time = timeit.default_timer()
squared_lc = [x ** 2 for x in numbers]
end_time = timeit.default_timer()
lc_time = end_time - start_time

# Using traditional loop
numbers = list(range(1, 10001))
start_time = timeit.default_timer()
squared_loop = []
for x in numbers:
    squared_loop.append(x ** 2)
end_time = timeit.default_timer()
loop_time = end_time - start_time

print("Time taken by list comprehension:", lc_time)
print("Time taken by traditional loop:", loop_time)
```

Output:

```
Time taken by list comprehension: 0.003491999999823747
Time taken by traditional loop: 0.004892400000244379
```