enumerate() is used to iterate over a sequence (like a list or a tuple) & get both index & the value of each item during iteration.
It return tuple where 1st ~~inde~~ element is index.
2nd element is ~~elem~~ item.

```
items = ['apple', 'banana', 'cherry']
for index, item in enumerate(items):
    print(index, item)
```

#output
0 apple
1 banana
2 cherry

zip() is used to combine 2 or more sequence (list, tuples, etc.) element-wise into a single iterator of tuples.
Each tuple contains one element from each sequence at the same position.

```
names = ['Alice', 'Ash', 'Bob']
scores = [85, 90, 88]
for name, score in zip(names, scores):
    print(name, score)
```

#output
Alice 85
Ash 90
Bob 88

my_list = [1, 2, 3, 4, 5]

1) Update an element in a list

my_list[2] = 10 $\xrightarrow{\text{output}}$ [1, 2, 10, 4, 5]

2 Add element to a list

append

my_list.append(6)
print(mylist) → [1, 2, 10, 4, 5, 6]

insert → add element at specific index
my_list.insert(2, 8) #Insert 8 at Index 2
print(mylist) → [1, 2, 8, 10, 4, 5, 6]

extend :- add multiple element at end of index.

my_list.extend([7, 8])
print(my_list) → [1, 2, 8, 10, 4, 5, 6, 7, 8]

3) Remove an Element

remove() removes the first occurence of a value:

my_list.remove(10)
print(my_list) → [1, 2, 8, 4, 5, 6, 7, 8]

pop :- remove element by index (or remove the last ~~ele~~element if no index is provided)

```
popped_value = my_list.pop(3)
print(popped_value) → 4
print(my_list) → [1,2,8,5,6,7,8]
```

del :- removes an element or a slice of Element.

```
del my_list[1] # Deletes the element at index 1
print(my_list) # [1,8,5,6,7,8]
```

4) Sort the list

#sort()

```
my_list.sort()       #sort list in ascending order
print(my_list) → [1,5,6,7,8,8]
```

#sorted()

```
my_list.sorted()  # sort the list in decending order
print(my_list) → [8,8,7,6,5,1]
```

(5) Reversing the list

You can reverse the list using reverse() or reversed()

reverse() :- It modifies the list <u>in place</u> & does not return a new list.
It changes the original list & returns <u>none</u>.

```
my_list = [1,5,2,3]
my_list.reverse()
print(my_list)     # [3,2,5,1]
```

reversed() :- It does not modify the original iterable; instead it returns an iterator.
Can be used with list, tuples, strings etc.

```
my_list = [1,5,2,3]
~~my~~ reversed_list = reversed(my_list)

print(reversed_list) -> [3,2,5,1]
print(my_list)  -> [1,5,2,3]
```

apsara

Date: __.__.__

6 Remove all Elements.

clear () to remove all element from list.

```
my_list.clear()
print(my_list) ⟶ [ ]
```

7 List Comprehesion

```
my_list = [1,2,3,4,5]
# square each number in the list

my_list = [x**2 for x in my_list]
print(my_list) → [1,4,9,16,25]
```

apsara

Date: ___

A dictionary in python is mutable, unordered collection of data that store key-value pairs.
Each key in dictionary is unique & associated value can be any datatype (ex:- integer, string, list, tuple, dict etc.)

Syntax of dictionary:-

{key: value, key: value, ---}

ex:-

my_dict = {"name": "Alice", "age": 25, ~~"city": "Ja"~~ "city": "Jaipur" }

Here name, age & city are keys
Alice, 25, Jaipur are values

Dictionary is Unordered, Mutable.

apsara

Date: __.__.__

Time complexity refers to the measure of the amount of time an algorithm takes to run as a function of the size of the input.

Big O notation is the most common way to express time complexity.
It describe the upper bond of an algo. runtime for worst case.

Time complexity typically depended upon the ~~num~~ size of the input which is denoted by n.

Best case :- When algo. perform the least number of operation.

~~Wo~~

Worst case : The scenario where the algo. perform the maximum number of operation.

Average case :- The expected number of operation for typical input.

1) O(1) - Constant Time.

Ex:-
```
arr = [1, 2, 3]
print (arr [2]) # O(1) operation
```

2) O(log n) - Logarithmic Time.

reduce the problem size by a constant factor (usually halving it) at each step.
Common in divide-and-conquer algo.

Ex:- Binary Search in sorted array.

③ O(n) - Linear Time:

Number of operation is directly propotion to the input size n.

Ex:- Linear Search

4) $O(n \log n)$ - Linearithmic Time

Algo with $O(n\log n)$ time complexity faster the $O(n^2)$ but slower then $O(n)$.

Ex: sorting algo. of Merge & Quick sort.

5) $O(n^2)$ - Quadratic Time.

time proportional to the square of the input size. Most common in nested loops.

Ex: Bubble sort, Selection sort,

6 $O(2^n)$ - Exponential Time :.

An algo. with time complexity $O(2^n)$ doubles its work with every additional input element. It is often ~~so~~ very inefficient & often used in brute-force solution

Ex: Recursive Fibonacci algoritm. (inefficient implementation).

(7) O(n!) - Factorial time.

An algo with time complexity O(n!) grows even faster than exponential time.

It appears mostly in all permutation & combinations of n elements.

like solving the traveling salesman problem by brute-method.

Ex:- Generating all permutation of a list. (by brute-force)