1. Accessing List Theory:

❖Understanding how to create and access elements in a list.

What Is a List ?

1. Ordered – retains the order of elements
2. Mutable – elements can be changed, added, or deleted
3. Heterogeneous – can contain mixed types (ints, strings, objects)
4. Dynamic – size grows/shrinks as elements are added or removed

Lists are implemented as dynamic arrays, offering flexible storage and indexing.

Creating a List :

- **Literal syntax:**

```
my_list = [1, 2, 3, "apple"]
```

- **Constructor syntax, from any iterable:**

```
my_list = list((1, 2, 3))
```

- **Repetition syntax:**

```
zeros = [0] * 5  # [0, 0, 0, 0, 0]
```

Accessing Elements:

1. Indexing

- **Basic indexing (0-based):**

```
my_list[0]    # 1st element
my_list[2]    # 3rd element
```

- **Negative indexing:**

```
my_list[-1]   # last element
my_list[-2]   # second-last element
```

2. Slicing :

Syntax:

```
slice_result = my_list[start:stop:step]
```

- start: inclusive index (defaults to 0)
- stop: exclusive index (defaults to len(my_list))
- step: step size (defaults to 1)

3. Nested Lists (2D Access) :

You can access multi-dimensional data by chaining indices:

```
matrix = [
    [1, 2, 3],
    [4, 5, 6],
]

matrix[1][2]   # 6 — row 1, column 2
```

❖Indexing in lists (positive and negative indexing).

Single-Element Indexing

```
my_list = ['a', 'b', 'c', 'd']
```

Positive Indexing (0-based):

- my_list[0] → 'a'
- my_list[2] → 'c'

Negative Indexing (from the end):

- my_list[-1] → 'd' (last element)
- my_list[-2] → 'c' (second-to-last)

Slicing (list[start:stop:step]) :

Syntax format:

```
my_list[start:stop:step]
```

- start: inclusive (default 0 if omitted)
- stop: exclusive (defaults to end if omitted)
- step: optional stride (default 1)

List .index() Method :

Syntax:

list.index(element, start, end)

- element: required
- start, end: optional range window

Common Pitfalls :

- **IndexError**: Accessing a[-len(a)-1], a[len(a)], or out-of-range slices.
- **Slices don’t wrap**: e.g., a[-1:1] returns []. Use a[-1:1:-1] to reverse part.
- .index() throws ValueError if the element is missing.

❖Slicing a list: accessing a range of elements.

Basic Syntax :

```
subset = my_list[start:stop:step]
```

- start: inclusive starting index; defaults to 0
- stop: exclusive ending index; defaults to len(list)
- step: interval between elements; defaults to 1

Semantics & Behavior :

- Slicing operates on *all sequence types* (lists, tuples, strings, etc.).
- The slice indices define positions *between elements*, selecting elements at those positions.
- If start > stop with positive step, you get an empty sequence.

Negative Indices & Stride :

- Negative indices count from the end, with -1 being the last element
    - a[-3:] returns the last three items.

    - a[:-2] returns all but the last two
- Negative step (step < 0): reverses direction
    - a[::-1] copies the list in reverse order
    - Defaults: if step < 0, start defaults to last index, stop to before first

Under-the-Hood: slice Objects :

- Internally, a[b:c:d] is equivalent to a[slice(b, c, d)].
- slice.start, .stop, and .step store the slicing params.
- .indices(length) adjusts slice bounds for a specific sequence length.

Copy vs. View :

- Built-in Python sequences (lists, strings, tuples): slicing returns a shallow copy.
    - The new list has its own container but contains references to the same elements.
- Other array types : slicing often produces a view sharing memory.

Slice vs. islice :

Complexity & Limitations :