Thanks for sharing the screenshot! You're demonstrating a very important concept in Python: **reference vs copy** for mutable objects like **lists**.

👇 Let me explain what's happening in your code, step by step:

✅ Case 1: `l1 = [1, 2, 3]`, `l2 = l1`

```python
l1 = [1, 2, 3] l2 = l1
```

- You did **not** create a new list for `l2`.
- Both `l1` and `l2` point to the **same list** in memory.

Then you did:

```python
l1[0] = 77
```

This **modifies the original list**, so now:

```python
l1 → [77, 2, 3] l2 → [77, 2, 3] ✅ (because they are the same object)
```

✅ Case 2: `p1 = [1, 2, 3]`, `p2 = [1, 2, 3]`

```python
p1 = [1, 2, 3] p2 = [1, 2, 3]
```

Here, you created **two different lists** with the same values. So `p1` and `p2` point to **different objects**.

Then you changed:

```python
p1[0] = 77
```

Now:

```python
p1 → [77, 2, 3] p2 → [1, 2, 3] ✅ (unchanged because it's a separate object)
```

🧠 Summary:

Statement	Points to Same Object?	Changes Affect Both?
`l2 = l1`	✅ Yes	✅ Yes
`p2 = [1, 2, 3]`	❌ No	❌ No

💡 Key Concept:

- `l2 = l1` is a **reference assignment**.
- `p2 = [1, 2, 3]` is a **new list with same content**.
- To make a **copy** of a list (not just a reference), use:

```python
new_list = old_list.copy() # or import copy new_list = copy.deepcopy(old_list) # for nested lists
```

NOTE:
- number above is an immutable so cannot be change
  so refrence change hudaina.
- as list is mutable so refrence can be change
  so simply can be understand from above figure.