UNIT 6: LISTS

• 1. What is a List?

A list is used to store multiple items in a single variable. Items in a list are ordered, changeable (mutable), and allow duplicate values.

• 2. Definition

An ordered, mutable collection of items enclosed in square brackets [ ].

Real-world example:

A shopping list or a list of students in a class.

• 3. Examples of Lists

- List with Integers:
  `numbers = [1, 2, 3, 4, 5]`
- List with Strings:
  `fruits = ["apple", "banana", "cherry"]`
- List with Mixed Data Types:
  `mixed = [10, "hello", 3.14, True]`

Remember:

Lists are Mutable !

4. Pro Tip

You can change, add, or remove items in a list after it is created.

* Common Operations:

Append(), Insert(), Remove(), Pop(), Sort(), len()

Topic : Creating Lists

☆ 1. Creating Lists

In Python, a list is a collection of items (values) stored in a particular order. Lists are created using square brackets [ ].

Examples :

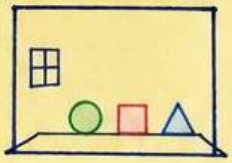
```
• empty_list = []                    # an empty list
• numbers = [1, 2, 3]                # list of integers
• mixed = [1, "Hello", 3.4]          # list with different data types
```

Common Mistake

Forgetting the commas between items! ✗

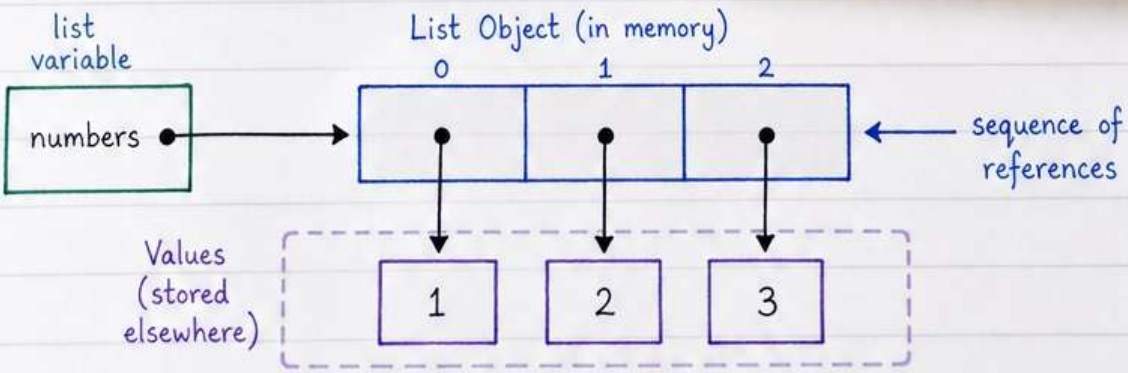
Memory Trick

Square brackets [ ] are like the walls of a room holding your items!



2. How is a List Stored in Memory?

A list stores references to its items in a sequence.



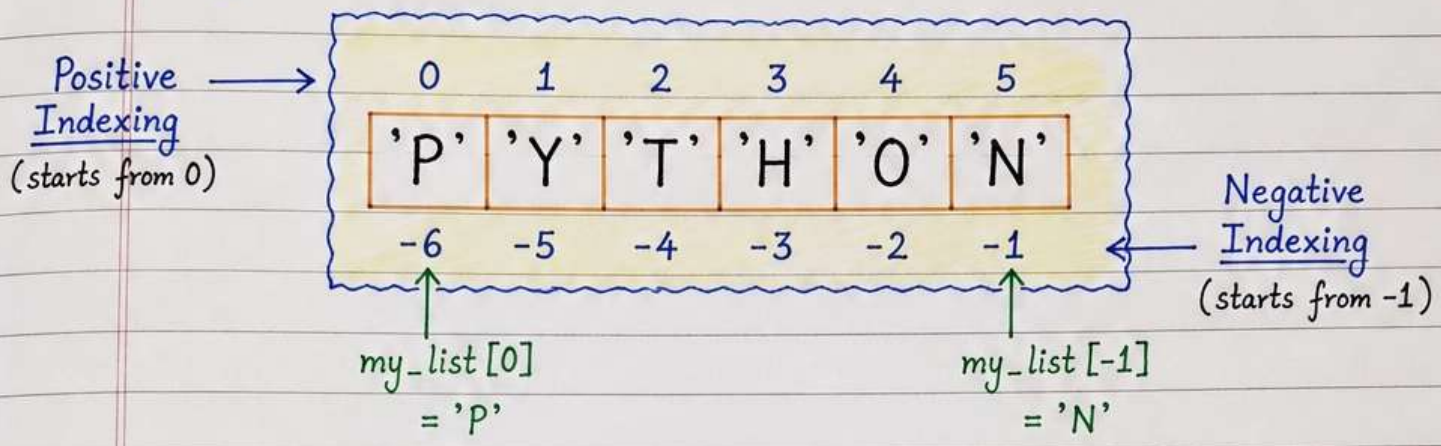
Note : The list holds references (addresses) to the actual values, not the values directly. ☺

S.P Group of Institute

Date : 24 / 05 / 2024
Page No. : 27

Topic : Accessing Elements - Indexing

In Python, elements of a list (or string) can be accessed using their index number.



☆ 1. Positive Indexing (starts from 0)

The first element has index 0, second has index 1 and so on.

Example : my_list [0] gives the first element.

☆ 2. Negative Indexing (starts from -1)

The last element has index -1, second last has index -2 and so on.

Example : my_list [-1] gives the last element.

Remember:
-0 is not negative zero!

Code Examples :

```
my_list = ['P', 'Y', 'T', 'H', 'O', 'N']
print (my_list [0])    # first element
print (my_list [-1])   # last element
```

→

Output :
P
N

☆ Quick Revision ☆

Rule	Type	Start	Direction	Example	Accesses
Positive Indexing	Forward Indexing	0	Left to Right →	my_list [0] my_list [3]	First element Fourth element
Negative Indexing	Backward Indexing	-1	Right to Left ←	my_list [-1] my_list [-4]	Last element Fourth last element

☆ Practice makes perfect! ☆

Topic : List Slicing

Slicing : Taking a Piece of the List !

Slicing allows us to extract a portion (or "slice") of a list using a simple and powerful syntax.

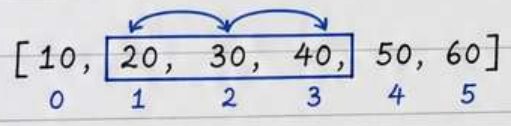
Syntax : `list[start : stop : step]`

- **start** → The index to start from (inclusive).
  Defaults to 0 if not provided.
- **stop** → The index to stop at (exclusive).
  Defaults to length of the list if not provided.
- **step** → The interval or "jump" between elements.
  Defaults to 1 if not provided.
  Can be any positive or negative integer (except 0).

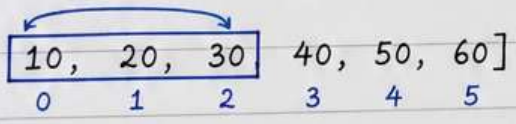
Examples : `nums = [10, 20, 30, 40, 50, 60]`

Index : 0 1 2 3 4 5

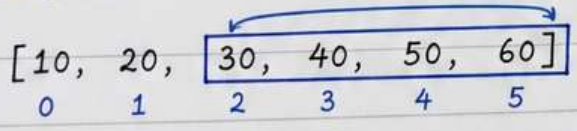
(1) `nums[1:4]` → `[20, 30, 40]`



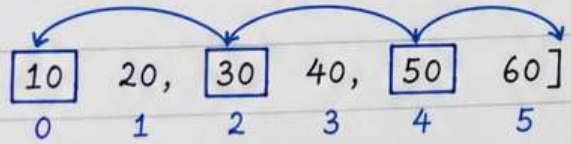
(2) `nums[:3]` → `[10, 20, 30]`



(3) `nums[2:]` → `[30, 40, 50, 60]`



(4) `nums[::2]` → `[10, 30, 50]` (step = 2)



Jump of 2 steps.

Pro Tip

You can reverse a list using slicing with a negative step !

`nums[::-1]`

→ Reverses the list.

Example :

`nums[::-1]` → `[60, 50, 40, 30, 20, 10]`

Mini-Exercise

What will `nums[1:5:2]` give for `[10, 20, 30, 40, 50, 60]` ?

.....

(Answer in small text at bottom)

(Ans: [20, 40])

S.P Group of Institute

Date : 16 / 05 / 2025
Page No. : 23

[+] Adding
★ Important

Topic : List Methods - Adding Elements

List Methods (Part 1) - Adding Items

1. **append()** — Adds a single element at the end of the list.

```
# Example
users = ['Alice', 'Bob']
users.append('Charlie')
print(users)
```

Output:
['Alice', 'Bob', 'Charlie']

★ append() adds only one item at the end.

---

2. **extend()** — Adds multiple elements (from another iterable) at the end of the list.

```
# Example
users = ['Alice', 'Bob']
users.extend(['Charlie', 'David'])
print(users)
```

Output:
['Alice', 'Bob', 'Charlie', 'David']

★ extend() iterates over the given iterable and adds each element to the list.

---

3. **insert()** — Inserts an element at the specified index.

```
# Example
users = ['Alice', 'Bob', 'David']
users.intend(1, 'Charlie')
print(users)
```

Output:
['Alice', 'Charlie', 'Bob', 'David']

★ insert(index, item) shifts the existing elements to the right.

Real-world example: Adding a new user to a database list

Suppose we have a list of users in a system. When a new user registers, we can add their name using these methods.

```
users = ['Alice', 'Bob']                # Existing users
users.append('Charlie')                 # Add at end
users.extend(['David', 'Eve'])          # Add multiple users
users.insert(1, 'Frank')                # Add at specific position
```

Result:
['Alice', 'Frank', 'Bob', 'Charlie', 'David', 'Eve']

★ Choosing the right method helps keep your list organized and your code efficient! ☺

List Methods (Part 2) – Removing Items

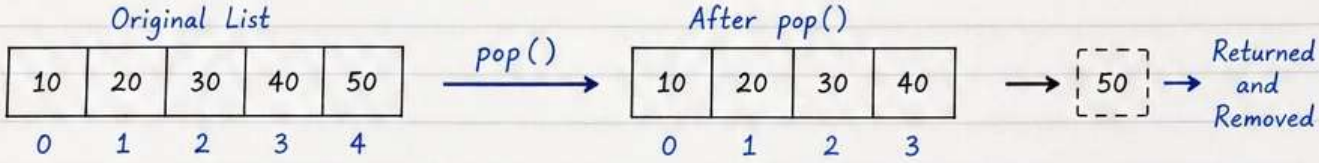
Python provides several methods to remove items from a list.
Let's explore them with examples.

Methods Covered:

① remove() → Removes the first occurrence of a specific value.
② pop() → Removes and returns an item at a given index.
③ clear() → Removes all items from the list.
④ del keyword → Deletes an item or the entire list using index or name.

---

* pop() removes the last item by default (if no index is specified).



① remove() – Removes first occurrence of value

```
lst = [10, 20, 30, 20, 40]
lst.remove(20)
print(lst)
```

Output: [10, 30, 20, 40]

② pop() – Removes item at index (default last)

```
lst = [10, 20, 30, 40]
item = lst.pop()    # default removes last item
print(item)
print(lst)
```

Output: 40
[10, 20, 30]

pop() with index:

```
lst = [10, 20, 30, 40]
item = lst.pop(1)    # removes item at index 1
print(item)
print(lst)
```

Output: 20
[10, 30, 40]

③ clear() – Removes all items from the list

```
lst = [10, 20, 30, 40]
lst.clear()
print(lst)
```

Output: []

④ del keyword – Deletes item by index

```
lst = [10, 20, 30, 40]
del lst[2]    # deletes item at index 2
print(lst)
```

Output: [10, 20, 40]

Deleting the entire list:

```
lst = [10, 20, 30]
del lst      # deletes the entire list
print(lst)   # This will raise an error
```

Output:
NameError: name 'lst' is not defined

✗ Common Mistake

Trying to remove an item that doesn't exist raises a ValueError!

Example:

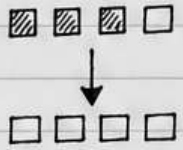
```
lst = [1, 2, 3]
lst.remove(5)    # ValueError
```

Pro Tip ☆

pop() returns the removed item, while remove() doesn't!

Topic: List Methods - Searching & Sorting

List Methods (Part 3) - Searching & Sorting



Python provides powerful built-in methods to search and sort elements in a list.

1. index(x) → Returns the index of the first occurrence of x
2. count(x) → Returns the number of times x appears in the list
3. sort() → Sorts the list in ascending order (in-place)
4. reverse() → Reverses the elements of the list (in-place)

✱ sort() vs sorted()

Method	sort()	sorted()
What it does	Sorts the list in-place (changes the original list)	Returns a new sorted list (original list remains unchanged)
Returns	None	New sorted list

Example	`nums = [3, 1, 2]`
	`nums.sort()`
	`# nums is now [1, 2, 3]`
		`nums = [3, 1, 2]`
		`new_nums = sorted(nums)`
		`# new_nums is [1, 2, 3]`
		`# nums is still [3, 1, 2]`

★ Sorting in Descending Order

- Use the parameter reverse = True

Using sort()	Using sorted()

`nums = [5, 2, 9, 1, 5]`
`nums.sort(reverse=True)`
`print(nums)`
`# Output: [9, 5, 5, 2, 1]`
	`nums = [5, 2, 9, 1, 5]`
	`new_nums = sorted(nums, reverse=True)`
	`print(new_nums)`
	`# Output: [9, 5, 5, 2, 1]`
	`print(nums)`
	`# Output: [5, 2, 9, 1, 5] (unchanged)`

★ Pro Tip ★

Sort() only works if all items are of the same type!

Note: Sorting is case-sensitive for strings.
Uppercase letters come before lowercase letters in ASCII.

☆ Nested Lists : Lists within Lists!

→ A nested list is a list where one or more elements are also lists.

Visualizing a 2D List (Matrix)

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

⇒

	0	1
0	1	2 → matrix[0][1]
1	3	4

Row index = 0
Column index = 1

Accessing elements using double indexing

```
matrix[row_index][col_index]
```

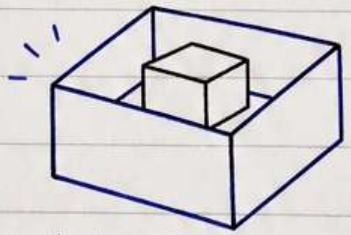
Code Examples </>

```
# Creating a nested list
matrix = [[1, 2], [3, 4]]

# Accessing elements
print(matrix[0][0])    # Output: 1
print(matrix[0][1])    # Output: 2
print(matrix[1][0])    # Output: 3
print(matrix[1][1])    # Output: 4

# Modifying elements
matrix[1][1] = 10
print(matrix)          # Output: [[1, 2], [3, 10]]
```

☆ Real-world example

Representing a game board (like Tic-Tac-Toe) or a table of data.

X	O	X
O	X	O
X	O	X



A box inside a box! ☺

☆ Key Takeaway

Nested lists help us represent grids, tables, or any multi-dimensional data in Python!

Think in rows & columns!

S.P Group of Institute

Page No.: 01
Date: 28/05/2024

Topic : List Comprehension

List Comprehension : The Pythonic Way!

➤ Syntax :

```
[expression for item in iterable if condition]
```

- The expression to evaluate and include in the list.
- The variable that takes each value from the iterable.
- The iterable (list, tuple, range, etc.) to loop over.
- Optional condition to filter items. Only items that satisfy the condition are included.

➤ Examples :

1) Create a list of squares of numbers from 1 to 5.

Traditional :

```
squares = []
for i in range(1, 6):
    squares.append(i*i)
print(squares)  # Output: [1, 4, 9, 16, 25]
```

List Comprehension :

```
squares = [i*i for i in range(1,6)]
print(squares)  # Output: [1, 4, 9, 16, 25]
```

Pro Tip

List comprehensions are faster and more concise than for loops!

2) Create a list of even numbers from 1 to 10.

Traditional :

```
evens = []
for i in range(1, 11):
    if i % 2 == 0:
        evens.append(i)
print(evens)  # Output: [2, 4, 6, 8, 10]
```

List Comprehension :

```
evens = [i for i in range(1,11) if i % 2 == 0]
print(evens)  # Output: [2, 4, 6, 8, 10]
```

➤ Comparison :

Aspect	Traditional For Loop	List Comprehension
Syntax	Longer and more verbose	Shorter and more concise
Readability	More lines, harder to read	Cleaner and easier to read
Performance	Slower	Faster (optimized in Python)
Use Case	Complex logic, multiple operations	Simple transformations, filtering
Example	`for i in range(1, 6):` `  squares.append(i*i)`	`[i*i for i in range(1, 6)]`

➤ Summary : Use list comprehensions for clean, efficient and Pythonic code!

S.P Group of Institute

Date : 24/05/2024
Page No. : 23

Topic: List Operations & Functions

List Operations & Built-in Functions

→ List Operations

1. Concatenation (+) : Combines two lists.
   Example: [1, 2] + [3, 4] → [1, 2, 3, 4]

2. Repetition (*) : Repeats the list multiple times.
   Example: [1, 2] * 3 → [1, 2, 1, 2, 1, 2]

3. Membership (in, not in): Checks if an element exists in a list.
   Example: 3 in [1, 2, 3, 4] → True
   Example: 5 not in [1, 2, 3, 4] → True

→ Built-in Functions

- len() : Returns the number of items in the list.
- max() : Returns the largest item in the list.
- min() : Returns the smallest item in the list.
- sum() : Returns the sum of all numeric items in the list.

Code Examples:

```
lst = [4, 7, 2, 9, 5]
print (len(lst))      # Output: 5
print (max(lst))      # Output: 9
print (min(lst))      # Output: 2
print (sum(lst))      # Output: 27

a = [1, 2, 3]
b = [4, 5]
print (a + b)         # Output: [1, 2, 3, 4, 5]
print (a * 2)         # Output: [1, 2, 3, 1, 2, 3]
print (2 in a)        # Output: True
print (6 not in a)    # Output: True
```

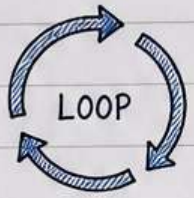
Quick Revision

Function	What it Does
len(lst)	Returns number of items
max(lst)	Returns largest item
min(lst)	Returns smallest item
sum(lst)	Returns sum of items

Memory Trick

max() and min() are like the tallest and shortest students in a class!

Iterating Through Lists



Lists are ordered collections in Python. We can access items one by one using loops. Here are three common ways to iterate through a list.

1 Using for loop (for item in my_list)

```
my_list = [10, 20, 30, 40]
for item in my_list:
    print (item)
```

→ Output:

```
10
20
30
40
```

2 Using range and len (for i in range(len(my_list)))

```
my_list = [10, 20, 30, 40]
for i in range(len(my_list)):
    print (my_list[i])
```

→ Output:

```
10
20
30
40
```

3 Using enumerate() to get both index and value

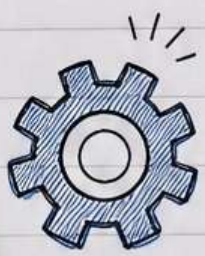
```
my_list = [10, 20, 30, 40]
for index, value in enumerate(my_list):
    print (f"Index {index} → Value {value}")
```

Output:

```
Index 0 → Value 10
Index 1 → Value 20
Index 2 → Value 30
Index 3 → Value 40
```

Pro Tip

Use enumerate() when you need both the item and its position!



Mini-Exercise

Write a loop to find the sum of all even numbers in a list.

Example:

Input : [1, 2, 3, 4, 5, 6, 7, 8]

Output : 20

"Code is like humor. When you have to explain it, it's bad." – Cory House

S.P Group of Institute

Date: 24/05/2024
Page No.: 27

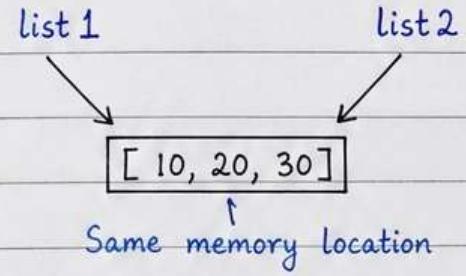
Topic : List Aliasing vs Cloning

Aliasing vs Cloning: Be Careful!

1. What's the Difference?

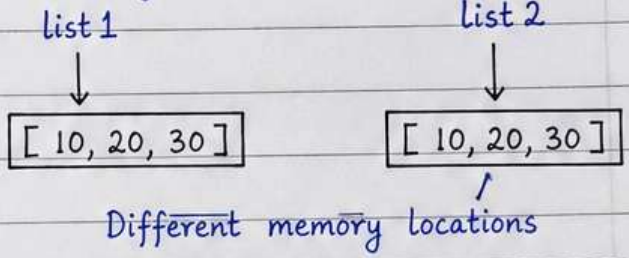
Aliasing (list2 = list1)

Both variables point to the same list in memory.



Cloning (list2 = list1.copy() or list1[:])

A new list is created. Both point to different memory locations.



2. Let's See in Code!

Aliasing Example

```
list1 = [10, 20, 30]
list2 = list1  # alias
list2.append(40).
print(list1)  # [10, 20, 30, 40]
print(list2)  # [10, 20, 30, 40]
```

↳ Both changed!

Cloning Example

```
list1 = [10, 20, 30]
list2 = list1.copy()  # or list1[:]
list2.append(40)
print(list1)  # [10, 20, 30]
print(list2)  # [10, 20, 30, 40]
```

↳ Original not changed!

Common Mistake

Modifying an alias by mistake!

```
list1 = [1, 2, 3]
list2 = list1  # alias (not a copy)
list2[0] = 99  # OOPS!
print(list1)  # [99, 2, 3] ← Unexpected!
```

Always use .copy() or [:] when you need a separate list.

Memory Trick

An alias is like a nickname for the same person, while a clone is a twin!

S.P Group of Institute

Date 24/05/2025
Page No.: 1

Practice Questions (Set 1)

Practice Questions - Level: Beginner

1) Find the largest number in a list without `max()`.

> **Hint:**
> Assume the first element is largest, then compare with the rest.

2) Remove duplicates from a list.

> **Hint:**
> Use a set to track seen elements.

3) Swap the first and last elements of a list.

> **Hint:**
> Use indexing. Think about `list[0]` and `list[-1]`.

4) Count how many times an item appears in a list.

> **Hint:**
> Initialize a counter and iterate through the list.

5) Check if a list is a palindrome (reads the same forwards and backwards).

> **Hint:**
> Compare the list with its reverse.

6) Find the second largest number in a list.

> **Hint:**
> Track the largest and second largest as you iterate.

Challenge

Given a list of integers, find all pairs that sum up to a target value.

(Example: [2, 7, 11, 15], target = 9 → Output: (2, 7))

S.P Group of Institute

Date : 20/05/2024
Page No. : 01

Topic : Interview Questions (Part 1)

Top Interview Questions on Lists

1. Difference between append() and extend()?

- append() adds a single element to the end of the list.
- extend() adds all elements of an iterable (list, tuple, etc.) to the end of the list.
- Example:

```
lst = [1, 2]
lst.append([3, 4])  → [1, 2, [3, 4]]
lst.extend([3, 4])  → [1, 2, 3, 4]
```

2. What is list comprehension?

- List comprehension is a concise way to create lists.
- It consists of an expression followed by a for clause, optionally followed by one or more if clauses.
- Example: `squares = [x*x for x in range(5)]` → `[0, 1, 4, 9, 16]`

3. How to reverse a list in one line?

- Use slicing with step -1.
- Example: `reversed_list = lst[::-1]`
- This returns a new list in reverse order.

4. How to remove duplicates from a list?

- Convert the list to a set and back to a list.
- Example: `unique_list = list(set(lst))`
- Note: This method does not preserve order.

5. What is the difference between sort() and sorted()?

- sort() sorts the list in place and returns None.
- sorted() returns a new sorted list without modifying the original list.
- Example:

```
lst.sort()                  # Original list is changed
new_lst = sorted(lst)       # Returns a new sorted list
```

Pro Tip

Interviewers love to ask about the difference between sort() and sorted()!

Topic: Real-world Examples

✧ Lists in the Real World ✧

Lists are used everywhere in programming to store ordered collections of items. Here are some real-world examples!

1 Managing a To-Do list

```
tasks = ["Study", "Lab Report", "Gym"]
tasks.append("Read Book")
tasks.remove("Gym")
print(tasks)          # ['Study', 'Lab Report', 'Read Book']
```

Why it matters:
Helps users add, remove, and track tasks in order of priority. Essential in productivity apps.

2 Storing sensor data over time

```
temperature_readings = []
temperature_readings.append(23.4)
temperature_readings.append(24.1)
temperature_readings.append(22.8)
print(temperature_readings)  # [23.4, 24.1, 22.8]
```

Why it matters:
Sensors generate data continuously. Lists store this data in sequence for analysis and visualization.

3 Implementing a simple stack or queue

Stack (LIFO)

```
stack = []
stack.append(1)
stack.append(2)
stack.pop()      # returns 2
print(stack)     # [1]
```

Queue (FIFO)

```
from collections import deque
queue = deque([])
queue.append(1)
queue.append(2)
queue.popleft()          # returns 1
print(list(queue))       # [2]
```

Why it matters:
Stacks and queues are used in scheduling, task processing, undo operations, and more.

⚠ Common Mistake

Modifying a list while iterating over it can lead to unexpected behavior.

Bad Example:

```
nums = [1, 2, 3, 4, 5]
for n in nums:
    if n % 2 == 0:
        nums.remove(n)    # Skips elements!
```

Better Approach:

```
nums = [1, 2, 3, 4, 5]
for n in nums[:]:    # iterate over a copy
    if n % 2 == 0:
        nums.remove(n)
```

Safe and predictable!

Takeaway: Lists are powerful, flexible, and used in countless real-world applications. Use them wisely! ☺