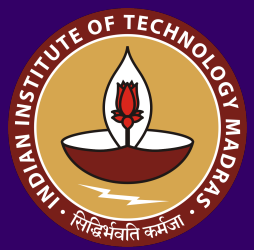
Scientific Computing & Advanced Programming Lab

SAURAV SAMANTARAY

Department of Mathematics
IIT Madras

Lecture-Slides-2



sauravsray@iitm.ac.in

Lists in Python

- A **list** is an ordered collection of objects.
- it can store values of the same or different data types.

Creating Lists

```
numbers = [2, 4, 6, 8]
names = ["Alice", "Bob", "Charlie"]
mixed = [3, "Python", 4.5]
```

Accessing Elements

```
print(numbers[0])     # 2
print(numbers[2])     # 6
print(names[-1])      # Charlie
```

- Lists are one of the most commonly used data structures in Python.

sauravsray@iitm.ac.in

Working with Lists

- Unlike strings, lists are **mutable**.
- Their contents can be changed after creation.

Examples

```
numbers = [2, 4, 6, 8]
numbers[1] = 10
print(numbers)
numbers.append(12)
print(numbers)
```

Output

```
[2, 10, 6, 8]
[2, 10, 6, 8, 12]
```

Lists and `for` Loops: Example

- Suppose we want to store the first five square numbers.

Step 1: Create the list

```
squares = [0, 0, 0, 0, 0]
```

Step 2: Use a `for` loop to initialise the list

```
for i in range(5):
    squares[i] = (i + 1)**2
```

Step 3: Access the elements

```
for value in squares:
    print(value)
```

Step 4: Modify the list Multiply every element by 2.

```
for i in range(len(squares)):
    squares[i] *= 2
```

Creating a List Dynamically: A Common Mistake

```
for i in range(5):
    squares[i] = (i + 1)**2
```

```
NameError:
name 'squares' is not defined
```

Even if we create an empty list,

```
squares = []
for i in range(5):
    squares[i] = (i + 1)**2
```

Python reports

```
IndexError:
list assignment index out of range
```

- **Why?** An empty list has no elements. Before we can assign to `squares[i]`, that position must already exist.

The Correct Approach: Growing a List

- Start with an empty list and use `append()` to add one element at a time.

```
squares = []
for i in range(5):
    squares.append((i + 1)**2)
print(squares)
```

Output

```
[1, 4, 9, 16, 25]
```

How the list grows

$$[] \longrightarrow [1] \longrightarrow [1,4] \longrightarrow [1,4,9] \longrightarrow [1,4,9,16] \longrightarrow [1,4,9,16,25]$$

Key Idea:

- The `append()` method creates a new element at the end of the list, allowing the list to grow dynamically.

Operations on Lists: Combining Lists

- Lists can be combined using the `+` operator.

```
L1 = [1, 2, 3]
L2 = [4, 5, 6]
L3 = L1 + L2
print(L3)
```

Output:

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

- **Adding several elements: `extend()`**

```
L = [1, 2, 3]
L.extend([4, 5, 6])
print(L)
```

Output:

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

Difference:

- `L1 + L2` creates a **new list**.
- `L.extend(...)` modifies `L` itself.

Operations on Lists: Modify and Remove

- Lists are **mutable**: individual elements can be changed.
- **Modify an element**

```
L = [10, 20, 30, 40]
L[1] = 25
```

```
print(L)
```

```
[10, 25, 30, 40]
```

- **Remove an element by index**

gives

```
del(L[2])
```

```
[10, 25, 40]
```

- **Remove an element by value**

gives

```
L.remove(25)
```

```
[10, 40]
```


sauravsray@iitm.ac.in

Removing Elements with `pop()`

- `pop()` removes an element **and returns it**.

```
L = [10, 20, 30, 40]
x = L.pop(2)
print(x)
print(L)
```

Output:

```
30
[10, 20, 40]
```

If no index is specified, the last element is removed.

```
x = L.pop()
```

`del(L[i])`	Remove element at index i
`L.remove(x)`	Remove first occurrence of value x
`L.pop(i)`	Remove and return element at index i
`L.pop()`	Remove and return the last element

Converting Between Lists and Strings

- A string can be split into a list using `split()`.

```
text = "Newton Raphson Method"
words = text.split()
print(words)
```

Output: `['Newton', 'Raphson', 'Method']`

- A list of strings can be joined into a single string using `join()`.

```
words = ["Numerical", "Linear", "Algebra"]
text = " ".join(words)
print(text)
```

Output: `Numerical Linear Algebra`

Key idea:

`string.split()` $\longleftrightarrow$ `separator.join(list)`

Other Useful List Operations

```
L = [4, 1, 7, 2, 5]
len(L)              # Number of elements
sum(L)              # Sum of elements
min(L)              # Smallest element
max(L)              # Largest element
L.sort()            # Sort the list
L.reverse()         # Reverse the list
print(7 in L)       # Check membership
print(3 not in L)
```

Example

```
L.sort()
print(L)
```

produces

```
[1, 2, 4, 5, 7]
```

- Methods such as `sort()`, `reverse()`, `extend()`, `append()`, `remove()`, and `pop()` modify the list.

Aliasing: Two Names, One List

```
L = [10, 20, 30]
M = L
```

What happens to `L`?

```
print(L)
print(M)
```

Now modify `M`:

```
M[0] = 100
```

Output:

```
[100, 20, 30]
[100, 20, 30]
```

- **Why?** `L` and `M` refer to the **same list**:
  $\boxed{L \longrightarrow [10, 20, 30] \longleftarrow M}$
- After the mutation: $\boxed{L \longrightarrow [100, 20, 30] \longleftarrow M}$
- **Key idea:** `M = L` does **not** create a new list.

How and Why Does Aliasing Happen?

- In Python, a variable is a **name that refers to an object**.

```
L = [10, 20, 30]
M = L
```

- Python does **not** create a new list.
- Both names refer to the same object:

Why is this useful?

- Avoids unnecessary copying of large objects.
- Functions can modify mutable objects supplied to them.

How: This is similar to a reference variable in C++:

- `x` and `y` refer to the same mem loc

```
int x = 10;
int& y = x;
```

- Python does not actually have C++-style reference variables. Python variables are names bound

Cloning a List

- What if we want a **separate copy** of a list?

Use slicing:

```
L = [10, 20, 30]
M = L[:]
M[0] = 100
```

```
print(L)
print(M)
```

produces

```
[10, 20, 30]
[100, 20, 30]
```

Other ways to clone a list:

```
M = L.copy()
M = list(L)
```

Key idea:

- `M = L` $\neq$ copy
- `M = L[:]` = new list

Tuples in Python

- A **tuple** is an ordered collection of objects, similar to a list.

Creating Tuples

```
point = (3, 4)
vector = (2.5, -1.0, 0.8)
student = ("Alice", 101, 8.9)
```

- Elements are accessed exactly like lists.

```
print(point[0])       # 3
print(vector[2])      # 0.8
```

Key Difference

- **Lists can be modified. Tuples cannot.**

sauravsray@iitm.ac.in

Why Use Tuples?

- A tuple is useful when the values naturally belong together and should not change.

Examples

- A point in the plane: (x, y)
- A point in three-dimensional space: (x, y, z)
- A student's record: $(\text{Name}, \text{ Roll Number}, \text{ CGPA})$
- Attempting to modify a tuple produces an error.

```
point = (3, 4)
point[0] = 10
```

```
TypeError:
'tuple' object does not support item assignment
```

Using Tuples to Return Multiple Values

- A function may need to return more than one value.
- A tuple provides a convenient way to group these values together.

Example

```
def quadratic_roots(a, b, c):
    # Compute the two roots
    ...

    return (r1, r2)
roots = quadratic_roots(1, -5, 6)
print(roots)
```

Output

```
(2.0, 3.0)
```