Python Programming

Unit 02 – Lecture 01: Lists Fundamentals and Comprehensions

Tofik Ali

School of Computer Science, UPES Dehradun

February 14, 2026

Repository: `https://github.com/tali7c/Python-Programming`

Quick Links

Core Concepts Demo Interactive Summary

Agenda

Learning Outcomes

- Create lists and access elements using indexing and slicing

Learning Outcomes

- Create lists and access elements using indexing and slicing
- Apply core list methods for insertion, deletion, and sorting

Learning Outcomes

- Create lists and access elements using indexing and slicing
- Apply core list methods for insertion, deletion, and sorting
- Write list comprehensions for concise transformations

Learning Outcomes

- Create lists and access elements using indexing and slicing
- Apply core list methods for insertion, deletion, and sorting
- Write list comprehensions for concise transformations
- Explain and use nested list structures

What is a List?

- Ordered collection of items

What is a List?

- Ordered collection of items
- Mutable (can change after creation)

What is a List?

- Ordered collection of items
- Mutable (can change after creation)
- Allows duplicates and mixed types

What is a List?

- Ordered collection of items
- Mutable (can change after creation)
- Allows duplicates and mixed types
- Ideal for sequences you need to update

Creating Lists

- Literal: `numbers = [2, 4, 6]`

Creating Lists

- Literal: `numbers = [2, 4, 6]`
- Constructor: `letters = list("python")`

Creating Lists

- Literal: `numbers = [2, 4, 6]`
- Constructor: `letters = list("python")`
- From range: `nums = list(range(5))`

Creating Lists

- Literal: `numbers = [2, 4, 6]`
- Constructor: `letters = list("python")`
- From range: `nums = list(range(5))`
- Mixed types: `items = ["A", 3.5, True]`

Indexing

- Zero-based indexing: first element at index 0

```
scores = [80, 92, 75, 88]
print(scores[0])   # 80
print(scores[-1])  # 88
```

Indexing

- Zero-based indexing: first element at index 0
- Negative indexing counts from the end

```
scores = [80, 92, 75, 88]
print(scores[0])   # 80
print(scores[-1])  # 88
```

Indexing

- Zero-based indexing: first element at index 0
- Negative indexing counts from the end
- Example: `scores = [80, 92, 75, 88]`

```
scores = [80, 92, 75, 88]
print(scores[0])   # 80
print(scores[-1])  # 88
```

Slicing

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

```
nums = [1, 2, 3, 4, 5, 6]
print(nums[1:4])    # [2, 3, 4]
print(nums[::2])    # [1, 3, 5]
```

Slicing

- Syntax: `list[start:stop:step]`
- Stop index is excluded

```
nums = [1, 2, 3, 4, 5, 6]
print(nums[1:4])    # [2, 3, 4]
print(nums[::2])    # [1, 3, 5]
```

Slicing

- Syntax: `list[start:stop:step]`
- Stop index is excluded
- Useful for sublists and stepping

```
nums = [1, 2, 3, 4, 5, 6]
print(nums[1:4])    # [2, 3, 4]
print(nums[::2])    # [1, 3, 5]
```

Core List Methods

Method	Purpose
`append(x)`	Add to end
`extend(iter)`	Add all items
`insert(i, x)`	Insert at index
`remove(x)`	Remove first match
`pop(i)`	Remove by index

- Most methods mutate the list in place

Sorting: sort vs sorted

- sort() modifies the list in place

```
nums = [3, 1, 2]
nums.sort()                          # nums becomes [1, 2, 3]
new_nums = sorted(nums, reverse=True)
```

Sorting: sort vs sorted

- sort() modifies the list in place
- sorted(list) returns a new list

```
nums = [3, 1, 2]
nums.sort()                          # nums becomes [1, 2, 3]
new_nums = sorted(nums, reverse=True)
```

Sorting: sort vs sorted

- sort() modifies the list in place
- sorted(list) returns a new list
- Use sorted when you must preserve the original

```
nums = [3, 1, 2]
nums.sort()                          # nums becomes [1, 2, 3]
new_nums = sorted(nums, reverse=True)
```

List Operations

- Concatenation: `[1,2] + [3,4]`

List Operations

- Concatenation: `[1,2] + [3,4]`
- Repetition: `[0] * 3`

List Operations

- Concatenation: `[1,2] + [3,4]`
- Repetition: `[0] * 3`
- Membership: `5 in nums`

List Operations

- Concatenation: `[1,2] + [3,4]`
- Repetition: `[0] * 3`
- Membership: `5 in nums`
- Length: `len(nums)`

Pitfall: Aliasing vs Copy

```
a = [1, 2, 3]
b = a              # alias (same object)
b.append(99)
print(a)           # [1, 2, 3, 99]
```

- Fix: create an independent list copy

```
b = a.copy()    # or: a[:]
```

List Comprehension

- Compact way to build lists

```
squares = [x*x for x in range(6)]
```

List Comprehension

- Compact way to build lists
- Pattern: `[expr for item in iterable if cond]`

```
squares = [x*x for x in range(6)]
```

Comprehension Example

```
nums = [3, 4, 7, 8, 10]
result = [n*n for n in nums if n % 2 == 0]
print(result)  # [16, 64, 100]
```

- Combine filtering and transformation in one line

Worked Example: Marks Analyzer (Core Steps)

```
marks = [72, 88, 95, 64, 51, 43, 77]

top_three = sorted(marks, reverse=True)[:3]
pass_mark = 50
passing = [m for m in marks if m >= pass_mark]
```

- Use sorted to avoid mutating the input list

Worked Example: Marks Analyzer (Core Steps)

```
marks = [72, 88, 95, 64, 51, 43, 77]

top_three = sorted(marks, reverse=True)[:3]
pass_mark = 50
passing = [m for m in marks if m >= pass_mark]
```

- Use `sorted` to avoid mutating the input list
- Use a comprehension to filter by a rule

Nested Lists

- Lists can contain other lists (2D structures)

```
matrix = [[1, 2, 3], [4, 5, 6]]
print(matrix[1][2])  # 6
```

Nested Lists

- Lists can contain other lists (2D structures)
- Access with two indices: `matrix[row][col]`

```
matrix = [[1, 2, 3], [4, 5, 6]]
print(matrix[1][2])  # 6
```

Pitfall: Modify While Iterating

- Removing items while iterating can skip elements

```
nums = [1, 2, 3, 4, 5]
nums = [n for n in nums if n % 2 == 1]
print(nums)  # [1, 3, 5]
```

Pitfall: Modify While Iterating

- Removing items while iterating can skip elements
- Prefer building a new list (filter) or iterate over a copy

```
nums = [1, 2, 3, 4, 5]
nums = [n for n in nums if n % 2 == 1]
print(nums)  # [1, 3, 5]
```

Demo: Student Marks Analyzer

- Load a list of marks

Script: `demo/list_marks_analyzer.py`

Demo: Student Marks Analyzer

- Load a list of marks
- Compute average and top 3 scores

Script: `demo/list_marks_analyzer.py`

Demo: Student Marks Analyzer

- Load a list of marks
- Compute average and top 3 scores
- Filter passing marks using a comprehension

Script: `demo/list_marks_analyzer.py`

Demo: Student Marks Analyzer

- Load a list of marks
- Compute average and top 3 scores
- Filter passing marks using a comprehension
- Extension: compute median as a challenge

Script: `demo/list_marks_analyzer.py`

Checkpoint 1

Which of these returns a new list instead of mutating?

1. `append`

Checkpoint 1

Which of these returns a new list instead of mutating?

1. `append`
2. `sort`

Checkpoint 1

Which of these returns a new list instead of mutating?

1. `append`
2. `sort`
3. `sorted`

Checkpoint 2

Write a list comprehension to square only even numbers from `nums = [1,2,3,4,5,6]`.

Think-Pair-Share

When is a **for-loop** clearer than a list comprehension? Give a short example and justify your choice.

Summary

- Lists are ordered, mutable collections

Summary

- Lists are ordered, mutable collections
- Indexing/slicing enable flexible access

Summary

- Lists are ordered, mutable collections
- Indexing/slicing enable flexible access
- Methods like `append` and `sort` mutate lists

Summary

- Lists are ordered, mutable collections
- Indexing/slicing enable flexible access
- Methods like `append` and `sort` mutate lists
- Comprehensions build lists concisely

Exit Question

What slice returns the last three items of a list `data`?