
📘 Python List Complete Guide

Introduction to Lists

A **list** is a collection of ordered, mutable, and heterogeneous elements. Lists are defined using **square brackets []** and can hold numbers, strings, booleans, other lists, or mixed data types.

my_list = [1, "apple", 3.5, True]

print(my_list) # [1, "apple", 3.5, True]

Properties of Lists

- **Ordered** – Items have defined index positions, allowing you to access them by their location.
- **Mutable** – You can change, add, or remove items after the list is created.
- **Allow Duplicates** – The same value can appear multiple times within a single list.
- **Heterogeneous** – Lists can store elements of different data types (e.g., a number and a string in the same list).

List Operations

Concatenation (+)

Combines two lists into a new list.

a = [1, 2]

b = [3, 4]

print(a + b) # [1, 2, 3, 4]

Repetition (*)

Repeats the list a specified number of times.

nums = [1, 2]

print(nums * 3) # [1, 2, 1, 2, 1, 2]

Membership (in, not in)p

Checks if an item exists in the list.

fruits = ["apple", "banana"]

print("apple" in fruits) # True

print("grape" not in fruits) # True

Indexing & Slicing

Accesses or extracts specific parts of the list using indices.

nums = [1, 2, 3, 4, 5]

print(nums[0]) # 1

print(nums[-1]) # 5

print(nums[1:4]) # [2, 3, 4]

Updating Elements

Changes the value of an element at a specific index.

nums = [10, 20, 30]

nums[1] = 25

print(nums) # [10, 25, 30]

Deleting Elements (del)

Removes an element at a specific index.

nums = [1, 2, 3, 4]

del nums[2]

print(nums) # [1, 2, 4]

List Methods (with Example & Behavior)

1. **append(x):** Adds a single element to the end of the list.

lst = [1, 2]

lst.append(3)

```
print(lst) # [1, 2, 3]
```

2. **extend(iterable):** Adds all elements from an iterable (like another list) to the end of the list.

```
lst = [1, 2]

lst.extend([3, 4])

print(lst) # [1, 2, 3, 4]
```

3. **insert(i, x):** Inserts an element x at a specified index i.

```
lst = [1, 3]

lst.insert(1, 2)

print(lst) # [1, 2, 3]
```

4. **remove(x):** Removes the first occurrence of a specified value x.

```
lst = [1, 2, 2, 3]

lst.remove(2)

print(lst) # [1, 2, 3]
```

5. **pop(i):** Removes and returns the element at a specified index i. If no index is given, it removes and returns the last element.

```
lst = [1, 2, 3]

print(lst.pop()) # 3

print(lst) # [1, 2]
```

6. **clear():** Removes all items from the list, making it empty.

```
lst = [1, 2, 3]

lst.clear()

print(lst) # []
```

7. **index(x**): Returns the index of the first occurrence of value x.

```
lst = [10, 20, 30]

print(lst.index(20)) # 1
```

8. **count(x):** Returns the number of times a value x appears in the list.

```
lst = [1, 2, 2, 3]
```

```
print(lst.count(2)) # 2
```

9. **sort():** Sorts the list in place (modifies the original list).

```
lst = [3, 1, 2]
lst.sort()
print(lst) # [1, 2, 3]
```

10. **reverse**(): Reverses the elements of the list in place.

```
lst = [1, 2, 3]
lst.reverse()
print(lst) # [3, 2, 1]
```

11. **copy():** Creates a shallow copy of the list.

```
lst = [1, 2, 3]
copy_lst = lst.copy()
print(copy_lst) # [1, 2, 3]
```

Built-in Functions with Lists

```
nums = [1, 2, 3, 4, 5]
print(len(nums)) # 5
print(sum(nums)) # 15
print(max(nums)) # 5
print(min(nums)) # 1
```

List Comprehension

List comprehension provides a concise way to create lists.

Syntax: [expression for item in iterable if condition]

Examples

```
squares = [x**2 for x in range(5)]
print(squares) # [0, 1, 4, 9, 16]
```

```
evens = [x for x in range(10) if x % 2 == 0]
print(evens) # [0, 2, 4, 6, 8]
```

Additional Points

Shallow vs. Deep Copy

The list.copy() method creates a **shallow copy**. This means it creates a new list object, but if the original list contains nested objects (like other lists), the new list will still refer to the *same* nested objects.

- Shallow Copy Example:

```
nested_list = [[1, 2], [3, 4]]
shallow_copy = nested_list.copy()
shallow_copy[0][0] = 99
print(nested_list) # [[99, 2], [3, 4]] (Original list is also changed!)
```

- **Deep Copy:** To create a completely independent copy of a list and its nested objects, you need to use the copy module's deepcopy() function.

```
import copy
nested_list = [[1, 2], [3, 4]]
deep_copy = copy.deepcopy(nested_list)
deep_copy[0][0] = 99
print(nested_list) # [[1, 2], [3, 4]] (Original list is unchanged)
```

Iterating Over a List

You can iterate through a list using a for loop.

- Simple Iteration:

```
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
print(fruit)
```

- Iteration with Index: Use the enumerate() function to get both the index and the value.

```
fruits = ["apple", "banana", "cherry"]
```

for index, fruit in enumerate(fruits):

print(f"Index {index}: {fruit}")

List as Stack and Queue

Lists can be used to implement fundamental data structures:

- Stack (LIFO - Last-In, First-Out): Use append() to add items and pop() to remove them from the end.

stack = [1, 2, 3]

stack.append(4) # [1, 2, 3, 4]

last_item = stack.pop() # last_item is 4, stack is [1, 2, 3]

- Queue (FIFO - First-In, First-Out): Use append() to add items and pop(0) to remove them from the beginning.

queue = [1, 2, 3]

queue.append(4) # [1, 2, 3, 4]

first_item = queue.pop(0) # first_item is 1, queue is [2, 3, 4]

Note: pop(0) is inefficient for large lists as it requires shifting all elements. A collections.deque is a better choice for queues.

Exercises

Basic Level

- Create a list of the first 10 natural numbers and print their squares.
- Find the largest and smallest number in a list.
- Count even and odd numbers in a list.
- Reverse a list without using the reverse() method.
- Take 5 student names as input and store them in a list.

Intermediate Level

- Merge two lists and remove duplicates.
- Find the second largest element in a list.
- Rotate a list by k steps.

- Generate prime numbers between 1–50 using list comprehension.
- Remove all vowels from a list of characters.

Advanced Level

- Flatten a nested list [[1,2],[3,4],[5,6]].
- Find common elements between two lists without using sets.
- Create a 3x3 matrix using list comprehension and print its diagonals.
- Generate Pascal's Triangle using lists.
- Implement stack and queue data structures using lists.

Interview-Oriented Questions

- **Difference between append() and extend()?**
  - append() adds a single element (the entire object) to the end of the list.
  - extend() adds all elements from an iterable to the end of the list.
- **Time complexity of insert() and pop()?**
  - insert(i, x) is **O(n)** because all subsequent elements must be shifted.
  - pop(i) is **O(n)** for the same reason, while pop() (no index) is **O(1)**.
- **How is sort() different from sorted()?**
  - list.sort() sorts the list **in place** and returns None. It only works on lists.
  - sorted() returns a **new sorted list** and leaves the original list unchanged. It works on any iterable.
- **Why use list comprehension over for loops?**
  - List comprehensions are generally **more concise, readable**, and often **faster** for creating a new list from an existing iterable.
- **Difference between copy() and = assignment?**
  - new_list = old_list creates a **reference**; both variables point to the same list in memory. Changes to one will affect the other.

- new_list = old_list.copy() creates a **shallow copy**, a new list object with a new memory address. Changes to the new list won't affect the original.