Python Fundamentals: Data Structures - Lists

by Bîcu Andrei Ovidiu

Table of contents

Introduction

Lists in Python are one of the most versatile and commonly used data structures. They are ***ordered, mutable,*** *and hold* ***duplicate values****, and can store* ***different data types*** *in a single collection.*

Think of a list as a flexible container that can change its size and content at any time.

```
my_list = [10, "apple", True]
print(my_list)  # Output: [10, 'apple', True]
```

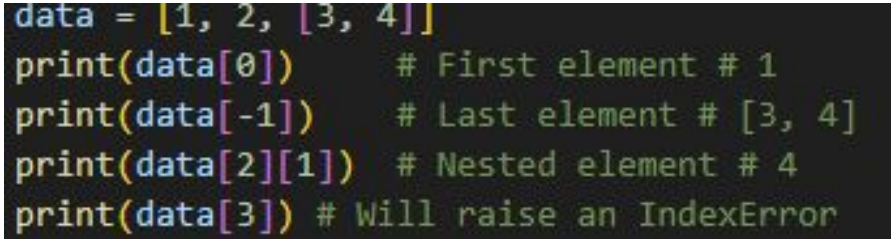
Creating Lists

A list can be created using square brackets or the list() constructor. You can start with an empty list, or initialize it with values. Lists can also be built from other iterables like strings or ranges, and through list comprehensions.

```
empty_list = [] # []
numbers = list(range(5)) # [0, 1, 2, 3 ,4]
chars = list("abc") #  ['a','b','c']
squares = [x**2 for x in range(5)] # [0, 1, 4 ,9, 16]
```

Accessing Elements

Elements are accessed by their index, starting at zero. Negative indexes count from the end. You can also access elements inside nested lists. Trying to access an index outside the list's range will cause an error.



```
data = [1, 2, [3, 4]]
print(data[0])     # First element # 1
print(data[-1])    # Last element # [3, 4]
print(data[2][1])  # Nested element # 4
print(data[3]) # Will raise an IndexError
```

Modifying Elements

Because lists are mutable, you can change elements directly. You can also replace multiple elements at once using slicing assignment.

```
items = ["a", "b", "c"]
items[1] = "z"  # ['a', 'z', 'c']
items[0:2] = ["x", "y"] # ['x', 'y', 'c']
```

Adding Elements

Python provides multiple methods to add elements:

- append() adds a single item to the end.
- extend() adds multiple items from another iterable.
- insert() adds an item at a specific position

```
lst = [1, 2]
lst.append([3, 4])   # Adds list as a single element -> [1, 2, [3, 4]]
lst.extend([5, 6])   # Adds elements individually -> [1, 2, [3, 4], 5, 6]
lst.insert(1, 10)    # Inserts at index 1 -> [1, 10, 2, [3, 4], 5, 6]
```

Removing Elements

Elements can be removed in several ways:

- remove() deletes the first matching value.
- pop() removes by index (last item by default).
- clear() empties the list entirely.
- The del statement deletes by index or slice

```
itms = ["apple", "banana", "cherry", 'avocado', 'tomato']
itms.remove("banana")  # ["apple", "cherry", 'avocado', 'tomato']
itms.pop()  # ["apple", "cherry", 'avocado']
del itms[0] # ["cherry", 'avocado']
itms.clear() # []
itms = ["apple", "banana", "cherry", 'avocado', 'tomato']
del itms[0:2] #  ["cherry", 'avocado', 'tomato']
```

Slicing

Slicing lets you extract parts of a list using [start:stop:step]. Omitting start or stop uses the list's beginning or end. A negative step reverses the list. You can also assign to a slice to replace multiple elements at once.

```
nums = [0, 1, 2, 3, 4, 5]
print(nums[1:4]) # [1, 2, 3]
print(nums[::-1]) # [5, 4, 3, 2, 1, 0]
nums[2:4] = [8, 9]
print(nums) # [0, 1, 8, 9, 4 ,5]
```

Bîcu Andrei Ovidiu

Searching in Lists

You can check if a value exists in a list with the in keyword. The index() method returns the position of the first match, and count() tells you how many times a value appears.

```
colors = ["red", "blue", "red"]
print("red" in colors) # True
print(colors.index("blue")) # 1
print(colors.count("red")) # 2
```

Bîcu Andrei Ovidiu

Iterating Over Lists

Lists can be looped over with for or while loops. Using enumerate() gives you both the index and the value in each iteration.

```
names = ["Alice", "Bob"]
for name in names:
    print(name)
'''
Alice
Bob
'''
for idx, name in enumerate(names):
    print(idx, name)
'''
0 Alice
1 Bob'''
```

Copying Lists

Assigning one list to another doesn't make a real copy, it just creates a new reference. To copy, use slicing or copy(). For nested lists, use copy.deepcopy() to avoid shared references

```
import copy
a = [[1, 2], [3, 4]]
b = a.copy()
c = copy.deepcopy(a)
d = a

print(a is b) # False
print(a is c) # False
print(a is d) # True
```

Sorting & Reversing

Lists can be sorted in-place with sort() or returned sorted as a new list with sorted(). The reverse() method flips the order, and you can customize sorting with a key function.

```
nums = [3, 1, 2]
nums.sort()
print(sorted(nums, reverse=True)) # [3, 2, 1]
words = ["avocado", "apple", "banana"]
words.sort(key=len)
print(words) # ['apple', 'banana', 'avocado']
```

Built-in Functions with Lists

Several built-in functions work with lists:

- len() counts elements.
- min() and max() find smallest and largest values.
- sum() adds numeric values.
- any() returns True if any element is truthy.
- all() returns True if all are truthy.

```
numbers = [1, 2, 3]
print(len(numbers), min(numbers), max(numbers), sum(numbers)) # 3 1 3 6 / 3 - len, 1 - min, 3 - max, 6 - sum
print(all([True, True]), any([False, True])) # True True
```

List Comprehensions

List comprehensions provide a concise way to build lists. You can add conditions and even nest them for more complex operations.

```
squares = [x**2 for x in range(5)]
evens = [x for x in range(10) if x % 2 == 0]
matrix = [[i*j for j in range(3)] for i in range(3)]

print(squares) # [0, 1, 4, 9, 16]
print(evens) # [0, 2, 4, 6, 8]
print(matrix) # [[0, 0, 0], [0, 1, 2], [0, 2, 4]]
```

Nested Lists

Lists can store other lists, forming multi-dimensional structures like matrices. Be careful when using multiplication (*) to initialize nested lists, it may cause shared references.

```
grid = [[0]*3 for _ in range(3)]
print(grid) # [[0, 0, 0], [0, 0, 0], [0, 0, 0]]
grid[0][0] = 1
print(grid) # [[1, 0, 0], [0, 0, 0], [0, 0, 0]]
```

Memory & Mutability

Lists are mutable, meaning they can be changed after creation. Assigning one list to another variable doesn't copy the data, it just points to the same memory location.

```
a = [1, 2]
b = a
b.append(3)
print(a) # [1, 2, 3]
print(b is a) # True
```

Common Pitfalls

Modifying a list while iterating over it can lead to unexpected results. Also, using mutable objects like lists as default function arguments can cause bugs.

```python
# Pitfall: Modifying a list while iterating
nums = [1, 2, 3, 4]
for n in nums:
    if n % 2 == 0:
        nums.remove(n)
print(nums)
# Output: [1, 3] — BUT it skipped 4 because indexes shifted!

# Safe approach: iterate over a copy or use list comprehension
nums = [1, 2, 3, 4]
nums = [n for n in nums if n % 2 != 0]
print(nums)  # [1, 3]
```

Lists vs Tuples vs Sets

Lists are ordered and mutable. Tuples are ordered but immutable. Sets are unordered and contain unique elements. Choosing the right one depends on your use case.

```python
# List: ordered, mutable, allows duplicates
lst = [1, 2, 2]
lst.append(3)  # Can modify
print(lst)     # [1, 2, 2, 3]

# Tuple: ordered, immutable, allows duplicates
tpl = (1, 2, 2)
# tpl[0] = 10  # Error: cannot modify
print(tpl)     # (1, 2, 2)

# Set: unordered, unique elements only
st = {1, 2, 2}
print(st)      # {1, 2} - duplicates removed
```

Real-World Use Cases

*Lists are used in **data storage**, **implementing stacks** and **queues**, or holding intermediate results in data processing.*

Why Lists Work Well Here:

- **Maintain order** of items
- **Easy modification** with .append() / .remove()
- **Compatible** with loops, comprehensions, and most built-in functions

```
'''Storing & Processing Data'''
temperatures = [22.5, 19.0, 23.7]
avg_temp = sum(temperatures) / len(temperatures)
print(f"Average: {avg_temp:.1f}°C")
# Great for datasets where order matters

'''Representing Game Inventories'''
inventory = ["sword", "shield", "potion"]
inventory.append("bow")
print(inventory)
# Easily add/remove items during gameplay

'''Single Data Pipelines'''
words = ["apple", "banana", "cherry"]
capitalized = [w.capitalize() for w in words]
print(capitalized)
# Ideal for quick transformations with list comprehensions
```

Lists as Stacks (LIFO)

*A stack follows the **Last In, First Out** principle. In Python, you can use a list and append() to push items, and pop() to remove the last item. Lists make it easy to implement stacks without extra libraries*

```
stack = []
stack.append("A")  # Push: Add 'A' to the top of the stack
stack.append("B")  # Push: Add 'B' on top of 'A'
print(stack)       # Output: ['A', 'B']

print(stack.pop()) # Pop: Removes and returns 'B' (last in)
print(stack)       # Output: ['A']
```

Lists as Queues (FIFO)

*A **queue** follows the **First in, First Out** rule, meaning the first element added is the first to be removed. You can simulate a queue with lists by appending items and removing them from the front using pop(0). However, this operation is inefficient for large lists because it shifts all the remaining elements.*

```
queue = []
queue.append("A")   # Enqueue: Add 'A' to the end of the queue
queue.append("B")   # Enqueue: Add 'B' behind 'A'
print(queue)        # Output: ['A', 'B']

print(queue.pop(0)) # Dequeue: Removes and returns 'A' (first in)
print(queue)        # Output: ['B']
```

Best Practices

When working with lists, following best practices helps keep your code clean, efficient, and easy to maintain. Here are some important guidelines:

1. ***Use List Comprehensions for Clarity and Performance***

   List comprehensions are not only concise but also often faster than traditional loops.

2. ***Avoid Modifying Lists While Iterating***
3. ***Use copy() and deepcopy() to Avoid Shared References***
4. ***Choose the Right Data Structure***
   - Immutable collections: tuples
   - Unique elements: sets
   - Efficient queues: collections.deque
5. ***Prefer collections.deque for Queues and Stacks When Performance Matters***

   deque provides fast appends and pops from both ends

```python
'''Use List Comprehensions for Clarity and Performance'''
nums = [1, 2, 3, 4, 5]

# Instead of using a loop to create squares:
squares = []
for n in nums:
    squares.append(n**2)  # Append the square of n

# Use a list comprehension for the same result:
squares = [n**2 for n in nums]  # Cleaner and more Pythonic

print(squares)  # Output: [1, 4, 9, 16, 25]

'''Avoid Modifying Lists While Iterating'''
nums = [1, 2, 3, 4]

# Problematic way: modifying list during iteration
for n in nums:
    if n % 2 == 0:
        nums.remove(n)  # Removing items shifts indices and causes skips

print(nums)  # Output might be unexpected: [1, 3]

#  Safe way: create a new filtered list instead
nums = [1, 2, 3, 4]
filtered = [n for n in nums if n % 2 != 0]  # Keep only odd numbers
print(filtered)  # Output: [1, 3]

'''Use copy() or deepcopy() to Avoid Shared References'''
import copy

original = [[1, 2], [3, 4]]
shallow_copy = original.copy()
deep_copy = copy.deepcopy(original)

shallow_copy[0].append(99)  # Affects original because inner lists are shared
print(original)  # Output: [[1, 2, 99], [3, 4]]

deep_copy[1].append(100)    # Does NOT affect original
print(original)  # Output remains: [[1, 2, 99], [3, 4]]

'''Prefer collections.deque for Queues and Stacks When Performance Matters'''
from collections import deque

queue = deque()
queue.append('A')  # Enqueue
queue.append('B')
print(queue.popleft())  # Dequeue 'A' efficiently
```

Bîcu Andrei Ovidiu

Glossary

Name	Definition
Mutable	An object that can be changed after creation. Lists are mutable.
Immutable	An object that cannot be changed after creation. Tuples are immutable.
Indexing	Accessing elements in a list by their position, starting at zero.
Negative Indexing	Accessing elements from the end of a list using negative numbers (e.g., -1 for the last item).
Slicing	Extracting a part of a list using [start:stop:step] syntax.
List Comprehension	A concise way to create lists using an expression, optionally with conditions and nesting.
Shallow Copy	Copying the outer list only; inner objects still reference the original elements.
Deep Copy	Copying the list and all objects inside it, so changes don't affect the original.
LIFO	Last In, First Out — the last added item is the first to be removed; used in stacks.
FIFO	First In, First Out — the first added item is the first to be removed; used in queues.

Name	Definition
append()	List method to add a single element at the end.
extend()	List method to add multiple elements from an iterable to the end.
pop()	List method to remove and return an element by index; defaults to the last element.
insert()	List method to add an element at a specified position.
remove()	List method to delete the first occurrence of a value.
clear()	List method that removes all items from the list.
Aliasing	When two or more variables reference the same list in memory, causing changes in one to affect the others.
Iterable	Any Python object capable of returning its elements one at a time (e.g., lists, strings, tuples).
Nested List	A list containing other lists as elements, allowing multi-dimensional data structures.
Mutable Default Argument	A common pitfall where a mutable object like a list is used as a default function parameter, causing unexpected shared state.

References & Further Reading

Official Documentation

- Python Docs: Lists
- Python Docs: list Class

Practice

- HackerRank: Python Challenges

Online Tutorials

- Real Python: Lists and Tuples
- W3Schools: Python Lists