Python & Leetcode Review

Beh Qian Jun

May 11, 2025

Contents

Chapter 1

Python Fundamentals

1.1 Scope of this Appendix

These condensed notes cover core Python concepts and ideas of solving some LeetCode problems when I was preparing for interviews. The notes assume that you already have some degree of familiarity with Python and these are just meant to be a recap. They are not exhaustive and do not cover all Python features or LeetCode problems (of course not). Please feel free to let me know if you think certain parts need to be revised, or any concepts/ problems should be added!

1.2 Lists vs Tuples

Lists Mutable, variable-length: `[1, 2, 3]`. Supports in-place modification (`append`, `__setitem__`, etc.). Unhashable by design.

Tuples Immutable, fixed-size: `(1, 2, 3)`. Once allocated, element pointers never change. Therefore tuples can safely cache their `__hash__` after the first call:

```
>>> t = (1, 2, 3)
>>> hash(t)          # hash computed & cached
>>> hash(t)          # O(1) field read, no recomputation
```

Why Immutability Helps

Because the tuple's contents cannot change, the cached hash stays valid forever, allowing tuples (and frozensets) to serve as dictionary keys or set members without risk of table corruption.

Memory Allocation: `malloc`

Low-level memory for every Python object ultimately comes from the C function `malloc` ("memory allocate"). CPython layers an object allocator on top (small-object pools, arenas) but still calls `malloc`/`realloc` for large blocks.

Lazy Hash Caching

When `hash()` is invoked on an immutable container for the first time, CPython computes the hash and stores it in an internal slot. Subsequent calls are constant-time field reads, which is crucial for performance in hash-table heavy workloads (e.g., counting keys in large dictionaries).

1.3 Dictionaries

`dict.get(key, default)` returns the stored value if `key` exists; otherwise it yields `default` without raising `KeyError`.

```python
d = {'a': 1, 'b': 2}
print(d.get('d', 4))  # -> 4
```

1.4 Object-Oriented Programming (OOP) in Python

1.4.1 Class vs Instance Attributes

- **Class attribute**: defined directly in the class body. Shared by all instances unless shadowed.
- **Instance attribute**: stored in the object's `__dict__`; unique per instance.
- Methods are class attributes that are functions; the descriptor protocol turns them into *bound methods* when accessed via an instance.

```python
class Animal:
    species = 'Unknown'   # class attribute
    def __init__(self, name):
        self.name = name  # instance attribute

dog = Animal('Buddy')
print(dog.species)  # -> 'Unknown'
dog.species = 'Dog'       # shadows class attribute for this instance
print(dog.species)  # -> 'Dog'
```

1.4.2 Inheritance Syntax `class Child(Parent)`

Declaring `class Child(Parent):` puts `Parent` into the new type's method-resolution order (MRO). Attribute look-up proceeds *instance* → *Child* → *Parent* → … until found or `AttributeError` is raised.

1.5 Shallow vs Deep Copy

1. **Shallow copy** (`copy.copy`): duplicates the outer container, reusing references to inner objects.
2. **Deep copy** (`copy.deepcopy`): recursively duplicates the entire object graph.

Use deep copies when you must mutate nested content without altering the original; otherwise prefer the cheaper shallow copy.

Revision checklist for interview prep

- Understand reference vs value semantics in CPython.
- Be able to predict side-effects of slicing, copying, and in-place operations.
- Explain how Python's data model (dunder methods) enables operator overloading and custom containers.
- Know common pitfalls: mutable default arguments, late binding in closures, iterator exhaustion.

Practice prompt: Implement a function that deep-copies a nested list *without* using `copy.deepcopy`. Analyse time/space complexity and potential stack-overflow issues.

Chapter 2

Leetcode Problems

2.1 Scope of this Appendix

The section below covers some of the **many** LeetCode problems I solved while preparing for interviews, or when I feel like I need to give myself a brainstorm. I will not include any actual code here, but rather the thought process that I had when solving the problems.

Problem 1: Two Sum

- Difficulty: `Easy`
- Topics: `Array`, `Hash Table`

> Given an array of integers `nums` and an integer `target`, return indices of the two numbers such that they add up to `target`. You may assume that each input would have exactly one solution, and you may not use the same element twice. You can return the answer in any order.

- Initialise a `hashmap` (or `dictionary` in Python), store the element of `nums` as key and index as value if its difference with the target is not already in.
- If the difference is in the hashmap, extract the indices.

Problem 21: Merge Two Sorted Lists

- Difficulty: `Easy`
- Topics: `Linked List`, `Recursion`

> Merge two sorted linked lists and return it as a new sorted list. The new list should be made by splicing together the nodes of the first two lists.

- Initialise a `dummy` node to store the merged list.
- Initialise a `curr` pointer to the `dummy` node.
- Iterate through both lists, comparing the values of the nodes.
- Append the smaller node to the `curr` pointer and move the `head` pointer of the respective list to the next node.
- Move the `curr` pointer to the next node.
- If one of the lists is not empty, append the remaining nodes to the `curr` pointer.
- Return the `next` of the `dummy` node.
- The time complexity is $\mathcal{O}(n + m)$ where n and m are the lengths of the two lists.
- The space complexity is $\mathcal{O}(1)$ since we are not using any extra space.

Problem 49: Group Anagrams

- Difficulty: `Medium`
- Topics: `Array`, `Hash Table`, `String`, `Sorting`

> Given an array of strings `strs`, group the anagrams together. You can return the answer in any order.

- The key idea here is to find a way to hash each word in a way such that only anagrams share the same hash. Using `ord()` is one way of doing it.
- Intialise a `dict()` to store the hash as key and the list of words as value.
- Iterate through the list of words. For each word, initialise a `list` of 26 zeros, then increment the corresponding index by 1 for each character in the word.
- Convert the `list` to a `tuple`, and that would be your key for that particular anagram.
- Append the word to the `dict()` using the hash as key.
- Return the values of the `dict()`.

Problem 104: Maximum Depth of Binary Tree

- Difficulty: `Easy`
- Topics: `Tree`, `Depth-First Search`, `Binary Tree`, `Breadth-First Search`

> Given the `root` of a binary tree, return its maximum depth. A binary tree's maximum depth is the number of nodes along the longest path from the root node down to the farthest leaf node.

- The idea is to use a depth-first search (DFS) to traverse the tree.
- If the current node is `None`, return 0.
- Otherwise, traverse the left and right subtrees with depth + 1.
- Return the maximum of the two depths.
- The time complexity is $\mathcal{O}(n)$ where n is the number of nodes in the tree.
- The space complexity is $\mathcal{O}(h)$ where h is the height of the tree.

Problem 121: Best Time to Buy and Sell Stock

- Difficulty: `Easy`
- Topics: `Array`, `Dynamic Programming`

> You are given an array `prices` where `prices[i]` is the price of a given stock on the i-th day. You want to maximize your profit by choosing a single day to buy one stock and choosing a different day in the future to sell that stock. Return the maximum profit you can achieve from this transaction. If you cannot achieve any profit, return 0.

- The idea is to keep track of the minimum price seen so far and the maximum profit.
- Iterate through the list of prices, updating the minimum price and maximum profit.
- Return the maximum profit.
- The time complexity is $\mathcal{O}(n)$ where n is the number of days.
- The space complexity is $\mathcal{O}(1)$.

Problem 217: Contains Duplicate

- Difficulty: `Easy`
- Topics: `Array`, `Hash Table`, `Sorting`

> Given an integer array `nums`, return `true` if any value appears at least twice in the array, and return `false` if every element is distinct.

- Initialise a `hashset`, if the current value is not in, add it into `hashset`.
- If the current value is in, return `True`.
- Return `False` when the loop ends.