Python Coding Interview Cheatsheet

Essential Patterns & Quick Reference

Core Data Structures & Operations

Lists - O(1) append, O(n) insert/delete

```
lst = [1, 2, 3]
lst.append(4)           # O(1)
lst.insert(1, 'x')      # O(n)
lst.pop()               # O(1) last
lst.pop(0)              # O(n) first
lst.reverse()           # O(n) in-place
lst.sort()              # O(n log n)
lst.sort(key=lambda x: -x)  # descending

# List slicing - creates new list
lst[1:3]    # [2, 3]
lst[::-1]   # reverse copy
lst[::2]    # every 2nd element
```

Strings - Immutable

```
s = "hello world"
s.split()               # ['hello', 'world']
s.split('l')            # ['he', '', 'o wor', 'd']
''.join(['a','b','c'])  # 'abc'
s.replace('l', 'x')     # 'hexxo worxd'
s.find('wor')           # 6 (index)
s.count('l')            # 3

# String formatting
f"Value: {x}"           # f-strings (preferred)
"Value: {}".format(x)

# Character operations
ord('a')                # 97
chr(97)                 # 'a'
s.isalnum(), s.isdigit(), s.isalpha()
```

Sets - O(1) average operations

```
s = {1, 2, 3}
s.add(4)
s.remove(2)             # KeyError if not found
s.discard(2)            # No error if not found
s1.union(s2)            # s1 | s2
s1.intersection(s2)     # s1 & s2
s1.difference(s2)       # s1 - s2
```

Dictionaries - O(1) average operations

```
d = {'a': 1, 'b': 2}
d.get('c', 0)           # default value
d.setdefault('c', [])   # set if not exists
d.pop('a', None)        # remove with default

# Iteration
for k, v in d.items():
    print(k, v)
```

Essential Collections Module

```
from collections import Counter, defaultdict, deque

# Counter - frequency counting
count = Counter("hello")    # {'l': 2, 'h': 1, ...}
count.most_common(2)        # [('l', 2), ('h', 1)]
count['x']                  # 0 (no KeyError)

# defaultdict - auto-initialize
dd = defaultdict(list)
dd['key'].append(1)         # auto-creates list

dd = defaultdict(int)
dd['count'] += 1            # auto-creates 0

# deque - efficient ends operations
dq = deque([1, 2, 3])
dq.appendleft(0)            # O(1)
dq.append(4)                # O(1)
dq.popleft()                # O(1)
dq.pop()                    # O(1)
```

Heap / Priority Queue

```
import heapq

# Min heap (default)
h = []
heapq.heappush(h, 3)
heapq.heappush(h, 1)
min_val = heapq.heappop(h)  # 1

# Max heap (negate values)
max_heap = []
heapq.heappush(max_heap, -5)
max_val = -heapq.heappop(max_heap)  # 5

# Heap from list
lst = [3, 1, 4, 1, 5]
heapq.heapify(lst)          # O(n)

# Top K elements
heapq.nlargest(3, lst)
heapq.nsmallest(3, lst)
```

Binary Search & Bisect

```
import bisect

# Standard binary search
def binary_search(arr, target):
    left, right = 0, len(arr) - 1
    while left <= right:
        mid = (left + right) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return -1

# Using bisect module
arr = [1, 3, 5, 7, 9]
idx = bisect.bisect_left(arr, 5)    # leftmost position
idx = bisect.bisect_right(arr, 5)   # rightmost position
bisect.insort(arr, 6)               # insert maintaining
    order
```

Essential Algorithm Patterns

Two Pointers

```
# Two sum on sorted array
def two_sum_sorted(arr, target):
    left, right = 0, len(arr) - 1
    while left < right:
```

```
        curr_sum = arr[left] + arr[right]
        if curr_sum == target:
            return [left, right]
        elif curr_sum < target:
            left += 1
        else:
            right -= 1
    return []

# Remove duplicates from sorted array
def remove_duplicates(arr):
    if not arr:
        return 0
    write = 1
    for read in range(1, len(arr)):
        if arr[read] != arr[read-1]:
            arr[write] = arr[read]
            write += 1
    return write
```

Sliding Window

```
# Fixed size window
def max_sum_subarray(arr, k):
    window_sum = sum(arr[:k])
    max_sum = window_sum

    for i in range(k, len(arr)):
        window_sum += arr[i] - arr[i-k]
        max_sum = max(max_sum, window_sum)
    return max_sum

# Variable size window
def longest_substring_k_distinct(s, k):
    char_count = {}
    left = max_length = 0

    for right in range(len(s)):
        char_count[s[right]] = char_count.get(s[right],
            0) + 1

        while len(char_count) > k:
            char_count[s[left]] -= 1
            if char_count[s[left]] == 0:
                del char_count[s[left]]
            left += 1

        max_length = max(max_length, right - left + 1)
    return max_length
```

Fast & Slow Pointers (Floyd's Algorithm)

```
# Detect cycle in linked list
def has_cycle(head):
    slow = fast = head
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next
        if slow == fast:
            return True
    return False

# Find middle of linked list
def find_middle(head):
    slow = fast = head
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next
    return slow
```

Tree & Graph Algorithms

Tree Traversals

```
# Binary tree node
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

# Recursive traversals
def inorder(root):
    return inorder(root.left) + [root.val] + inorder(root
        .right) if root else []

def preorder(root):
    return [root.val] + preorder(root.left) + preorder(
        root.right) if root else []

# Iterative inorder
def inorder_iterative(root):
    stack, result = [], []
    current = root

    while stack or current:
        while current:
            stack.append(current)
            current = current.left
        current = stack.pop()
        result.append(current.val)
        current = current.right
    return result

# Level order (BFS)
def level_order(root):
    if not root:
        return []

    result = []
    queue = deque([root])

    while queue:
        level_size = len(queue)
        level = []

        for _ in range(level_size):
            node = queue.popleft()
            level.append(node.val)

            if node.left:
                queue.append(node.left)
            if node.right:
                queue.append(node.right)

        result.append(level)
    return result
```

Graph DFS & BFS

```
# Graph as adjacency list
graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F'],
    'D': [], 'E': [], 'F': []
}

# DFS recursive
def dfs(graph, node, visited=None):
    if visited is None:
        visited = set()

    visited.add(node)
    print(node)

    for neighbor in graph[node]:
        if neighbor not in visited:
            dfs(graph, neighbor, visited)

# DFS iterative
def dfs_iterative(graph, start):
    visited = set()
    stack = [start]

    while stack:
        node = stack.pop()
        if node not in visited:
            visited.add(node)
            print(node)
            stack.extend(graph[node])
```

```python
# BFS
def bfs(graph, start):
    visited = set()
    queue = deque([start])

    while queue:
        node = queue.popleft()
        if node not in visited:
            visited.add(node)
            print(node)
            queue.extend(graph[node])
```

Dynamic Programming Patterns

1D DP

```python
# Fibonacci
def fib(n):
    if n <= 1:
        return n
    dp = [0] * (n + 1)
    dp[1] = 1
    for i in range(2, n + 1):
        dp[i] = dp[i-1] + dp[i-2]
    return dp[n]

# House robber
def rob(nums):
    if len(nums) <= 2:
        return max(nums) if nums else 0

    dp = [0] * len(nums)
    dp[0] = nums[0]
    dp[1] = max(nums[0], nums[1])

    for i in range(2, len(nums)):
        dp[i] = max(dp[i-1], dp[i-2] + nums[i])
    return dp[-1]
```

2D DP

```python
# Unique paths in grid
def unique_paths(m, n):
    dp = [[1] * n for _ in range(m)]

    for i in range(1, m):
        for j in range(1, n):
            dp[i][j] = dp[i-1][j] + dp[i][j-1]
    return dp[m-1][n-1]

# Longest common subsequence
def lcs(text1, text2):
    m, n = len(text1), len(text2)
    dp = [[0] * (n + 1) for _ in range(m + 1)]

    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if text1[i-1] == text2[j-1]:
                dp[i][j] = dp[i-1][j-1] + 1
            else:
                dp[i][j] = max(dp[i-1][j], dp[i][j-1])
    return dp[m][n]
```

Backtracking

```python
# Generate all permutations
def permute(nums):
    result = []

    def backtrack(path):
        if len(path) == len(nums):
            result.append(path[:])  # copy
            return

        for num in nums:
            if num not in path:
                path.append(num)
                backtrack(path)
                path.pop()          # backtrack

    backtrack([])
    return result
```

```python
# Generate all subsets
def subsets(nums):
    result = []

    def backtrack(start, path):
        result.append(path[:])      # add current subset

        for i in range(start, len(nums)):
            path.append(nums[i])
            backtrack(i + 1, path)
            path.pop()

    backtrack(0, [])
    return result
```

String Algorithms

```python
# Check palindrome
def is_palindrome(s):
    left, right = 0, len(s) - 1
    while left < right:
        if s[left] != s[right]:
            return False
        left += 1
        right -= 1
    return True

# Check anagrams
def is_anagram(s1, s2):
    return Counter(s1) == Counter(s2)
    # or: return sorted(s1) == sorted(s2)

# Longest palindromic substring
def longest_palindrome(s):
    def expand_around_center(left, right):
        while left >= 0 and right < len(s) and s[left] ==
            s[right]:
            left -= 1
            right += 1
        return right - left - 1

    start = max_len = 0
    for i in range(len(s)):
        len1 = expand_around_center(i, i)        # odd
            length
        len2 = expand_around_center(i, i + 1)    # even
            length
        curr_max = max(len1, len2)

        if curr_max > max_len:
            max_len = curr_max
            start = i - (curr_max - 1) // 2

    return s[start:start + max_len]
```

Bit Manipulation

```python
# Common operations
x & 1            # check if odd
x >> 1           # divide by 2
x << 1           # multiply by 2
x & (x-1)        # remove rightmost set bit
x | (1 << i)     # set i-th bit
x & ~(1 << i)    # clear i-th bit
x ^ (1 << i)     # toggle i-th bit

# Count set bits
def count_bits(n):
    count = 0
    while n:
        count += 1
        n &= n - 1  # remove rightmost set bit
    return count

# Check power of 2
def is_power_of_2(n):
    return n > 0 and (n & (n-1)) == 0
```

Sorting & Searching

```python
# Custom sorting
arr.sort(key=lambda x: (x[1], -x[0]))  # by 2nd elem asc, 1st desc

# Binary search variations
def find_first_occurrence(arr, target):
    left, right = 0, len(arr) - 1
    result = -1

    while left <= right:
        mid = (left + right) // 2
        if arr[mid] == target:
            result = mid
            right = mid - 1      # continue searching left
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return result

# Search in rotated sorted array
def search_rotated(nums, target):
    left, right = 0, len(nums) - 1

    while left <= right:
        mid = (left + right) // 2

        if nums[mid] == target:
            return mid

        # Left half is sorted
        if nums[left] <= nums[mid]:
            if nums[left] <= target < nums[mid]:
                right = mid - 1
            else:
                left = mid + 1
        # Right half is sorted
        else:
            if nums[mid] < target <= nums[right]:
                left = mid + 1
            else:
                right = mid - 1
    return -1
```

Advanced Sliding Window

```python
# Longest substring without repeating characters
def length_of_longest_substring(s):
    char_set = set()
    left = max_length = 0

    for right in range(len(s)):
        while s[right] in char_set:
            char_set.remove(s[left])
            left += 1

        char_set.add(s[right])
        max_length = max(max_length, right - left + 1)

    return max_length

# Minimum window substring
def min_window(s, t):
    if not s or not t:
        return ""

    dict_t = Counter(t)
    required = len(dict_t)
    formed = 0
    window_counts = {}

    left = right = 0
    ans = float("inf"), None, None

    while right < len(s):
        char = s[right]
        window_counts[char] = window_counts.get(char, 0) + 1

        if char in dict_t and window_counts[char] == dict_t[char]:
            formed += 1

        while left <= right and formed == required:
            char = s[left]

            if right - left + 1 < ans[0]:
                ans = (right - left + 1, left, right)

            window_counts[char] -= 1
            if char in dict_t and window_counts[char] < dict_t[char]:
                formed -= 1

            left += 1

        right += 1

    return "" if ans[0] == float("inf") else s[ans[1]:ans[2] + 1]
```

Common Interview Tricks

Array Manipulation

```python
# Dutch flag problem (3-way partition)
def sort_colors(nums):
    red, white, blue = 0, 0, len(nums) - 1

    while white <= blue:
        if nums[white] == 0:
            nums[red], nums[white] = nums[white], nums[red]
            red += 1
            white += 1
        elif nums[white] == 1:
            white += 1
        else:
            nums[white], nums[blue] = nums[blue], nums[white]
            blue -= 1

# Rotate array
def rotate(nums, k):
    k %= len(nums)
    nums[:] = nums[-k:] + nums[:-k]

# Product of array except self
def product_except_self(nums):
    result = [1] * len(nums)

    for i in range(1, len(nums)):
        result[i] = result[i-1] * nums[i-1]

    right_product = 1
    for i in range(len(nums)-1, -1, -1):
        result[i] *= right_product
        right_product *= nums[i]

    return result
```

Linked List Patterns

```python
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

# Reverse linked list
def reverse_list(head):
    prev = None
    current = head

    while current:
        next_temp = current.next
        current.next = prev
        prev = current
        current = next_temp

    return prev

# Merge two sorted lists
def merge_two_lists(l1, l2):
    dummy = ListNode(0)
    current = dummy

    while l1 and l2:
        if l1.val <= l2.val:
            current.next = l1
            l1 = l1.next
        else:
            current.next = l2
            l2 = l2.next
        current = current.next

    current.next = l1 or l2
    return dummy.next
```

Math & Utility Functions

```python
import math

# GCD and LCM
math.gcd(12, 15)                    # 3
def lcm(a, b):
    return abs(a * b) // math.gcd(a, b)

# Prime checking
def is_prime(n):
    if n < 2:
        return False
    for i in range(2, int(math.sqrt(n)) + 1):
        if n % i == 0:
            return False
    return True

# Generate primes (Sieve of Eratosthenes)
def sieve_of_eratosthenes(n):
    primes = [True] * (n + 1)
    primes[0] = primes[1] = False

    for i in range(2, int(math.sqrt(n)) + 1):
        if primes[i]:
            for j in range(i*i, n + 1, i):
                primes[j] = False

    return [i for i in range(2, n + 1) if primes[i]]
```

Time & Space Complexity Quick Reference

```python
# Array operations
arr[i]          # O(1)
arr.append(x)   # O(1) amortized
arr.insert(0,x) # O(n)
arr.pop()       # O(1)
arr.pop(0)      # O(n)

# Dictionary operations
d[key]          # O(1) average
d.get(key)      # O(1) average

# Set operations
x in s          # O(1) average
s.add(x)        # O(1) average

# Sorting
sorted(arr)     # O(n log n)
arr.sort()      # O(n log n) in-place

# Common algorithms
# Binary search: O(log n)
# DFS/BFS: O(V + E)
# Heap operations: O(log n)
```

Interview-Specific Python Tips

```python
# Initialize 2D array (avoid [[0]*n]*m)
matrix = [[0] * n for _ in range(m)]

# Infinity values
float('inf'), float('-inf')

# Multiple assignment
a, b = b, a             # swap
a, b = divmod(x, y)     # quotient, remainder

# List comprehensions with conditions
[x for x in arr if x > 0]
[x if x > 0 else 0 for x in arr]

# Dictionary comprehensions
{k: v for k, v in items if condition}

# Enumerate with custom start
for i, val in enumerate(arr, 1):
    print(f"Position {i}: {val}")

# Zip for parallel iteration
for a, b in zip(list1, list2):
    print(a, b)

# All/Any for conditions
all(x > 0 for x in arr)
any(x < 0 for x in arr)

# String multiplication
"a" * 5                 # "aaaaa"

# Set operations shorthand
set1 | set2             # union
set1 & set2             # intersection
set1 - set2             # difference
```

Common Edge Cases to Remember

```python
# Always check:
# - Empty input: [], "", None
# - Single element: [1]
# - Negative numbers
# - Integer overflow (use float('inf'))
# - Duplicate elements
# - Already sorted input

# Template for edge case handling
def solve(arr):
    if not arr:
        return []           # or appropriate default

    if len(arr) == 1:
        return arr[0]       # or appropriate single-
            element result

    # Main logic here
    pass
```