

DSA Patterns Cheat Sheet: 17 Python Templates Beyond Arrays

Reusable Python templates for strings, stacks, linked lists, trees, graphs, heaps, backtracking and dynamic programming. For each: when to use it, the template, its complexity, the classic bug, and LeetCode problems to practise.

Free

Free PDF

11 pages

Intermediate

By wins.solutions team

Updated 2 Oct 2026

Short answer

Most coding-round questions are one of a few patterns in disguise: sliding window, two pointers, stacks, binary search on the answer, linked-list pointers, tree DFS and BFS, grid BFS, topological sort, Union-Find, Dijkstra, heaps, backtracking and dynamic programming. Learn the signal in the question that points to each, keep one tested template per pattern, and adapt it instead of starting from zero.

WINS.SOLUTIONS · DSA

DSA Patterns Cheat Sheet: 17 Python Templates Beyond Arrays

Most coding-round questions are one of a small number of patterns in disguise. Learn to recognise the signal in the question, then adapt the template instead of starting from zero. This sheet picks up where our **array interview questions** leave off: two pointers on arrays, prefix sums and intervals are covered there.

Every template below was run against test cases, including edge cases such as empty input, before this sheet was published. The templates assume these imports:

```
import heapq
from collections import Counter, defaultdict, deque
from math import ceil
```

How to practise with this sheet

Read the "use it when" line and the template, close the sheet, then solve the first linked problem from memory. Come back to the template only if you're stuck for 20 minutes. Two days later, solve the second problem without looking.

#	Pattern	Signal in the question
1	Sliding window on a string	Longest or shortest substring with a condition
2	Two pointers from both ends	Compare the two ends and move inward
3	Group by a signature	Group items that match after a transformation
4	Stack for matching	Openers must close in reverse order
5	Monotonic stack	Next or previous greater or smaller element
6	Binary search on the answer	Minimum value that makes something feasible
7	Reverse, middle and cycle	Reverse, find the middle, detect a cycle
8	Depth-first and breadth-first	Combine subtree results, or work level by level
9	Validate a BST with bounds	A rule that must hold against all ancestors
10	Grid BFS (flood fill)	Count or fill connected regions in a grid
11	Topological sort (Kahn's algorithm)	Order tasks with prerequisites, detect cycles
12	Union-Find	Merge groups and ask if two items connect
13	Dijkstra's shortest path	Cheapest path with non-negative weights
14	Top-k with a heap	k largest, k most frequent, or the kth item
15	Subsets and permutations	Generate every subset or arrangement
16	1-D DP	Answer for n builds on smaller answers
17	2-D DP on two strings	Compare two sequences prefix by prefix

Strings and stacks

1. Sliding window on a string

Use it when: You need the longest or shortest contiguous substring that satisfies a condition, such as no repeats or at most k changes.

```
def longest_unique_substring(s):
    last = {}
    left = best = 0
    for right, ch in enumerate(s):
        if ch in last and last[ch] ≥ left:
            left = last[ch] + 1
        last[ch] = right
        best = max(best, right - left + 1)
    return best
```

Complexity: $O(n)$ time, $O(k)$ space for k distinct characters.

Classic bug: Move `left` only forward. If you ever move it backward, the window is not valid for this pattern.

Practise: Longest Substring Without Repeating Characters, Longest Repeating Character Replacement, Minimum Window Substring

2. Two pointers from both ends

Use it when: The answer depends on comparing the two ends of a string or sorted array and moving inward.

```
def is_palindrome(s):
    i, j = 0, len(s) - 1
    while i < j:
        while i < j and not s[i].isalnum(): i += 1
        while i < j and not s[j].isalnum(): j -= 1
        if s[i].lower() != s[j].lower(): return False
        i += 1; j -= 1
    return True
```

Complexity: $O(n)$ time, $O(1)$ space.

Classic bug: Skip characters you should ignore inside the loop, and re-check `i < j` after each skip.

Practise: Valid Palindrome, Valid Palindrome II

3. Group by a signature

Use it when: Items that are equal under some transformation (sorted letters, counts, a pattern) must be grouped or matched.

```
def group_anagrams(words):
    groups = defaultdict(list)
    for w in words:
        groups["".join(sorted(w))].append(w)
    return list(groups.values())
```

Complexity: $O(n \times k \log k)$ for n words of length k . Using a 26-letter count tuple as the key makes it $O(n \times k)$.

Classic bug: The key must be hashable: use a string or tuple, not a list.

Practise: Group Anagrams, Valid Anagram

4. Stack for matching

Use it when: Every opener must be closed in reverse order: brackets, tags, nested expressions, undo history.

```
def valid_brackets(s):
```

```

pairs = {"(": ")", "[": "]", "{": "}"
stack = []
for ch in s:
    if ch in "([{":
        stack.append(ch)
    elif ch in pairs:
        if not stack or stack.pop() != pairs[ch]: return False
return not stack

```

Complexity: $O(n)$ time, $O(n)$ space.

Classic bug: Check for an empty stack before popping, and check that the stack is empty at the end.

Practise: Valid Parentheses, Min Stack

5. Monotonic stack

Use it when: For each element you need the next (or previous) greater or smaller element.

```

def daily_temperatures(t):
    ans = [0] * len(t)
    stack = [] # indices, temperatures decreasing
    for i, x in enumerate(t):
        while stack and t[stack[-1]] < x:
            j = stack.pop()
            ans[j] = i - j
        stack.append(i)
    return ans

```

Complexity: $O(n)$ time: each index is pushed and popped once. $O(n)$ space.

Classic bug: Store indices, not values, so you can compute distances.

Practise: Daily Temperatures, Next Greater Element I, Largest Rectangle in Histogram

6. Binary search on the answer

Use it when: You're asked for the minimum (or maximum) value that makes something feasible, and feasibility only changes once as the value grows.

```
def min_eating_speed(piles, h):
    lo, hi = 1, max(piles)
    while lo < hi:
        mid = (lo + hi) // 2
        if sum(ceil(p / mid) for p in piles) ≤ h:
            hi = mid
        else:
            lo = mid + 1
    return lo
```

Complexity: $O(n \log M)$, where M is the largest possible answer.

Classic bug: Write the feasibility check as its own function first; the search is then always the same eight lines.

Practise: Koko Eating Bananas, Capacity To Ship Packages Within D Days

Linked lists

7. Reverse, middle and cycle

Use it when: Linked-list questions: reversing all or part of a list, finding the middle, detecting a cycle.

```
class Node:
    def __init__(self, val, nxt=None): self.val, self.next = val, nxt

def reverse_list(head):
    prev = None
    while head:
        head.next, prev, head = prev, head, head.next
    return prev

def middle(head):
    slow = fast = head
    while fast and fast.next:
        slow, fast = slow.next, fast.next.next
    return slow

def has_cycle(head):
    slow = fast = head
    while fast and fast.next:
        slow, fast = slow.next, fast.next.next
        if slow is fast: return True
    return False
```

Complexity: $O(n)$ time, $O(1)$ space for all three.

Classic bug: Draw three boxes for prev, current and next before you write the reversal. Most bugs are a lost pointer.

Practise: Reverse Linked List, Middle of the Linked List, Linked List Cycle

Trees

8. Depth-first and breadth-first

Use it when: DFS when the answer combines results from subtrees (depth, sums, paths). BFS when the question mentions levels or the shortest number of steps.

```
class T:
    def __init__(self, val, left=None, right=None): self.val, self.left, self.right = val, left, right

def max_depth(root):
    if not root: return 0
    return 1 + max(max_depth(root.left), max_depth(root.right))

def level_order(root):
    if not root: return []
    out, q = [], deque([root])
    while q:
        level = []
        for _ in range(len(q)):
            node = q.popleft()
            level.append(node.val)
            if node.left: q.append(node.left)
            if node.right: q.append(node.right)
        out.append(level)
    return out
```

Complexity: $O(n)$ time. DFS uses $O(h)$ stack space for height h ; BFS uses $O(w)$ for the widest level.

Classic bug: In BFS, take `len(q)` before the inner loop so each pass handles exactly one level.

Practise: Maximum Depth of Binary Tree, Binary Tree Level Order Traversal

9. Validate a BST with bounds

Use it when: A property must hold for a node relative to all its ancestors, not just its parent.

```
def is_valid_bst(root, lo=float("-inf"), hi=float("inf")):
    if not root: return True
    if not (lo < root.val < hi): return False
    return is_valid_bst(root.left, lo, root.val) and is_valid_bst(root.right, root.val, hi)
```

Complexity: $O(n)$ time, $O(h)$ space.

Classic bug: Checking only `left.val < node.val` is the classic wrong answer. Pass the allowed range down instead.

Practise: Validate Binary Search Tree, Kth Smallest Element in a BST

Graphs

10. Grid BFS (flood fill)

Use it when: A 2-D grid where you count, fill or measure connected regions, or find the shortest path in steps.

```
def num_islands(grid):
    if not grid: return 0
    rows, cols = len(grid), len(grid[0])
    seen, count = set(), 0
    for r in range(rows):
        for c in range(cols):
            if grid[r][c] == "1" and (r, c) not in seen:
                count += 1
                q = deque([(r, c)]); seen.add((r, c))
                while q:
                    x, y = q.popleft()
                    for dx, dy in ((1, 0), (-1, 0), (0, 1), (0, -1)):
                        nx, ny = x + dx, y + dy
                        if 0 ≤ nx < rows and 0 ≤ ny < cols and grid[nx][ny] == "1" and (nx, ny) not in seen:
                            seen.add((nx, ny)); q.append((nx, ny))
    return count
```

Complexity: $O(\text{rows} \times \text{cols})$ time and space.

Classic bug: Mark a cell as seen when you add it to the queue, not when you pop it, or cells get added twice.

Practise: Number of Islands, Rotting Oranges

11. Topological sort (Kahn's algorithm)

Use it when: Tasks with prerequisites: can they all be done, and in what order? Also detects cycles in a directed graph.

```
def can_finish(n, prereqs):
    graph, indeg = defaultdict(list), [0] * n
    for course, pre in prereqs:
        graph[pre].append(course); indeg[course] += 1
    q = deque(i for i in range(n) if indeg[i] == 0)
    done = 0
    while q:
        u = q.popleft(); done += 1
        for v in graph[u]:
            indeg[v] -= 1
            if indeg[v] == 0: q.append(v)
    return done == n
```

Complexity: $O(V + E)$ time and space.

Classic bug: If fewer than V nodes come out of the queue, the graph has a cycle.

Practise: Course Schedule, Course Schedule II

12. Union-Find

Use it when: You need to merge groups and repeatedly ask whether two items are connected, especially as edges arrive one by one.

```
class DSU:
    def __init__(self, n): self.parent, self.size = list(range(n)), [1] * n
    def find(self, x):
        while self.parent[x] != x:
            self.parent[x] = self.parent[self.parent[x]]
            x = self.parent[x]
        return x
    def union(self, a, b):
        ra, rb = self.find(a), self.find(b)
        if ra == rb: return False
        if self.size[ra] < self.size[rb]: ra, rb = rb, ra
        self.parent[rb] = ra; self.size[ra] += self.size[rb]
        return True

def count_components(n, edges):
    d = DSU(n)
    return n - sum(d.union(a, b) for a, b in edges)
```

Complexity: Nearly $O(1)$ per operation with path compression and union by size.

Classic bug: `union` returning `False` means the two items were already connected: that edge would form a cycle.

Practise: Number of Provinces, Redundant Connection

13. Dijkstra's shortest path

Use it when: Shortest path from one source when edge weights are non-negative.

```
def dijkstra(n, edges, src):
    graph = defaultdict(list)
    for u, v, w in edges: graph[u].append((v, w))
    dist = [float("inf")] * n; dist[src] = 0
    pq = [(0, src)]
    while pq:
        d, u = heapq.heappop(pq)
        if d > dist[u]: continue
        for v, w in graph[u]:
            if d + w < dist[v]:
                dist[v] = d + w; heapq.heappush(pq, (dist[v], v))
    return dist
```

Complexity: $O((V + E) \log V)$ with a binary heap.

Classic bug: Skip stale heap entries with `if d > dist[u]: continue`. For negative weights use Bellman-Ford instead.

Practise: Network Delay Time, Path With Minimum Effort

Heaps

14. Top-k with a heap

Use it when: The k largest, smallest or most frequent items, or the kth one, without fully sorting.

```
def top_k_frequent(nums, k):
    return [x for x, _ in heapq.nlargest(k, Counter(nums).items(), key=lambda p: p[1])]

def kth_largest(nums, k):
    heap = []
    for x in nums:
        heapq.heappush(heap, x)
        if len(heap) > k: heapq.heappop(heap)
    return heap[0]
```

Complexity: $O(n \log k)$ time, $O(k)$ space for the size-k heap.

Classic bug: Python's `heapq` is a min-heap. To keep the k largest, keep a min-heap of size k and pop the smallest.

Practise: Top K Frequent Elements, Kth Largest Element in an Array

Backtracking

15. Subsets and permutations

Use it when: Generate every combination, subset, permutation or arrangement that meets a rule.

```
def subsets(nums):
    out, path = [], []
    def go(i):
        if i == len(nums): out.append(path[:]); return
        path.append(nums[i]); go(i + 1); path.pop()
        go(i + 1)
    go(0)
    return out

def permutations(nums):
    out, path, used = [], [], [False] * len(nums)
    def go():
        if len(path) == len(nums): out.append(path[:]); return
        for i, x in enumerate(nums):
            if not used[i]:
                path.append(x)
                used[i] = True
                go()
                path.pop()
                used[i] = False
```

```

        if used[i]: continue
        used[i] = True; path.append(x)
        go()
        path.pop(); used[i] = False
    go()
    return out

```

Complexity: Subsets $O(n \times 2^n)$. Permutations $O(n \times n!)$.

Classic bug: Append a copy (`path[:]`), never `path` itself, or every result ends up empty.

Practise: Subsets, Permutations, Combination Sum

Dynamic programming

16. 1-D DP

Use it when: The answer for size n depends on answers for smaller sizes, and choices overlap: take or skip, which coin, how many steps.

```

def house_robber(nums):
    take = skip = 0
    for x in nums:
        take, skip = skip + x, max(take, skip)
    return max(take, skip)

def coin_change(coins, amount):
    INF = amount + 1
    dp = [0] + [INF] * amount
    for a in range(1, amount + 1):
        for c in coins:
            if c <= a: dp[a] = min(dp[a], dp[a - c] + 1)
    return dp[amount] if dp[amount] != INF else -1

```

Complexity: House robber $O(n)$ time, $O(1)$ space. Coin change $O(\text{amount} \times \text{coins})$ time, $O(\text{amount})$ space.

Classic bug: Write the recurrence in words first, for example "best up to house i = max(skip i , take i + best up to $i - 2$)".

Practise: Climbing Stairs, House Robber, Coin Change

17. 2-D DP on two strings

Use it when: Two sequences are compared and the answer for prefixes builds the answer for the whole: LCS, edit distance, matching.

```

def lcs(a, b):
    dp = [[0] * (len(b) + 1) for _ in range(len(a) + 1)]
    for i in range(1, len(a) + 1):

```

```

for j in range(1, len(b) + 1):
    dp[i][j] = dp[i-1][j-1] + 1 if a[i-1] == b[j-1] else max(dp[i-1][j], dp[i][j-1])
return dp[-1][-1]

```

Complexity: $O(m \times n)$ time and space; you can reduce space to $O(n)$ by keeping two rows.

Classic bug: Give the table one extra row and column for empty prefixes, so the base cases are all zero.

Practise: Longest Common Subsequence, Edit Distance

Picking the pattern in an interview

If the question says...	Try
"longest / shortest substring or subarray"	Sliding window
"next greater", "previous smaller"	Monotonic stack
"minimum possible maximum", "at least X in Y days"	Binary search on the answer
"levels", "minimum steps", "nearest"	BFS
"prerequisites", "order of tasks", "dependencies"	Topological sort
"are these connected?", edges arriving one at a time	Union-Find
"k largest", "k most frequent", "kth"	Heap
"all combinations", "all arrangements", "generate every"	Backtracking
"number of ways", "minimum cost", choices that repeat	Dynamic programming
weighted edges, "cheapest", "fastest route"	Dijkstra

Say the pattern out loud before you code: "This looks like a sliding window because we need the longest substring with a condition." Interviewers score the reasoning, not just the final code.