Python Cheat Sheet

Capital One CodeSignal Assessment — Complete Quick Reference

1. Core Data Types & Variables

```
x = 10          # int
pi = 3.14        # float
flag = True      # bool  (True/False — capitalized!)
name = "Sean"    # str
nothing = None   # NoneType

# Type conversion
int("42")  # 42       str(42)   # "42"
float("3.14")         bool(0)   # False
list("abc")  # ['a','b','c']
tuple([1,2]) # (1, 2)
set([1,1,2]) # {1, 2}
```

***Tip:** Python is dynamically typed. **type(x)** returns the type. **isinstance(x, int)** for checking.*

2. Operators

Category	Operators	Notes
Arithmetic	+ - * / // % **	// = floor div, ** = power
Comparison	== != < > <= >=	Returns bool
Logical	and or not	Short-circuit evaluation
Bitwise	& | ^ ~ << >>	& = AND, | = OR, ^ = XOR
Identity	is is not	Checks same object in memory
Membership	in not in	Works on str, list, dict, set
Assignment	+= -= *= //= **= %=	Augmented assignment

***Tip: a // b** always floors: **7 // 2 = 3**, **-7 // 2 = -4**. Use **int(a / b)** for truncation toward zero.*

3. Strings (Immutable)

```
s = "hello world"
s[0]          # 'h'         s[-1]       # 'd'
s[0:5]        # 'hello'     s[::-1]     # 'dlrow olleh'
len(s)        # 11

# Essential methods
s.split()          # ['hello', 'world']
s.split(',')       # split on delimiter
' '.join(['a','b'])# 'a b'
s.strip()          # remove leading/trailing whitespace
s.replace('o','0') # 'hell0 w0rld'
s.find('world')    # 6  (-1 if not found)
s.count('l')       # 3
s.startswith('he') # True
s.upper()  /  s.lower()  /  s.title()
s.isdigit() / s.isalpha() / s.isalnum()

# f-strings (formatted)
name, age = "Sean", 30
f"{name} is {age}"        # "Sean is 30"
```

```
f"{3.14159:.2f}"          # "3.14"
f"{42:08b}"               # "00101010" (binary)
```

***Tip:** Strings are **immutable**. Building with **+=** in a loop is O(n^2). Use **list + join** instead.*

4. Lists (Mutable, Ordered)

```
a = [1, 2, 3, 4, 5]
a[0]          # 1          a[-1]         # 5
a[1:3]        # [2, 3]     a[::-1]       # [5,4,3,2,1]

# Modify
a.append(6)             # [1,2,3,4,5,6]
a.insert(0, 0)          # [0,1,2,3,4,5,6]
a.extend([7, 8])        # appends multiple
a.pop()                 # removes & returns last
a.pop(0)                # removes & returns index 0
a.remove(3)             # removes first occurrence of 3
del a[1]                # delete by index

# Search & sort
a.index(4)              # index of first 4
a.count(2)              # count of 2's
a.sort()                # in-place ascending
a.sort(reverse=True)    # in-place descending
sorted(a)               # returns NEW sorted list
a.reverse()             # in-place reverse

# List comprehension
squares = [x**2 for x in range(10)]
evens   = [x for x in range(20) if x % 2 == 0]
flat    = [x for row in matrix for x in row]  # flatten 2D
```

***Tip: sort(key=lambda x: ...)** is critical. E.g., **a.sort(key=lambda x: x[1])** sorts by 2nd element.*

5. Dictionaries (Mutable, Key-Value)

```
d = {"a": 1, "b": 2, "c": 3}
d["a"]                    # 1
d.get("z", 0)             # 0 (default if missing)
d["d"] = 4                # add / update
del d["a"]                # delete key
d.pop("b", None)          # remove & return (with default)

# Iteration
d.keys()                  # dict_keys(['a','b','c'])
d.values()                # dict_values([1,2,3])
d.items()                 # dict_items([('a',1),('b',2),...])
for k, v in d.items():    # unpack key-value

# Dict comprehension
freq = {ch: s.count(ch) for ch in s}
inv  = {v: k for k, v in d.items()}  # invert

# defaultdict & Counter (from collections)
from collections import defaultdict, Counter
dd = defaultdict(list)    # dd[key] auto-creates []
dd = defaultdict(int)     # dd[key] auto-creates 0
cnt = Counter("aabbc")    # Counter({'a':2,'b':2,'c':1})
cnt.most_common(2)        # [('a',2),('b',2)]
```

***Tip:** **Counter** and **defaultdict** are your best friends for frequency counting problems.*

6. Sets (Mutable, Unordered, Unique)

```
s = {1, 2, 3}
s.add(4)                  # {1,2,3,4}
s.discard(2)              # safe remove (no error if missing)
s.remove(3)               # raises KeyError if missing

# Set operations
a = {1,2,3}; b = {2,3,4}
a | b          # union        {1,2,3,4}
a & b          # intersection {2,3}
a - b          # difference   {1}
a ^ b          # symmetric    {1,4}
a.issubset(b)  # False
# Set comprehension
unique = {x % 10 for x in nums}
```

***Tip:** Use sets for O(1) lookups. Converting a list to a set: **seen = set(nums)**.*

7. Tuples (Immutable, Ordered)

```
t = (1, 2, 3)
t[0]            # 1
a, b, c = t     # unpacking
a, *rest = t    # a=1, rest=[2,3]

# As dict keys (hashable)
grid = {}
grid[(0, 0)] = "start"

# Named tuples
from collections import namedtuple
Point = namedtuple('Point', ['x', 'y'])
p = Point(3, 4)
p.x  # 3
```

8. Control Flow

```
# if / elif / else
if x > 0:
    print("positive")
elif x == 0:
    print("zero")
else:
    print("negative")

# Ternary
result = "even" if x % 2 == 0 else "odd"

# for loops
for i in range(5):          # 0,1,2,3,4
for i in range(2, 10, 3):   # 2,5,8
for i, val in enumerate(lst):  # index + value
for a, b in zip(list1, list2): # parallel iteration

# while
while condition:
    ...
    if done: break
    if skip: continue

# for-else (runs if loop completed without break)
for x in lst:
    if x == target:
        break
else:
    print("not found")
```

9. Functions

```
def greet(name, greeting="Hello"):
    """Docstring: describe function."""
    return f"{greeting}, {name}!"

# *args, **kwargs
def f(*args, **kwargs):
    print(args)    # tuple of positional
    print(kwargs)  # dict of keyword

# Lambda
square = lambda x: x ** 2
add    = lambda a, b: a + b

# Map, Filter, Reduce
list(map(lambda x: x*2, [1,2,3]))       # [2,4,6]
list(filter(lambda x: x>2, [1,2,3,4]))  # [3,4]
from functools import reduce
reduce(lambda a,b: a+b, [1,2,3,4])      # 10

# Nested functions / closures
def outer(x):
    def inner(y):
        return x + y
    return inner
add5 = outer(5)
add5(3)  # 8
```

10. Sorting Patterns

```
# Built-in sort
nums.sort()                        # in-place, ascending
nums.sort(reverse=True)            # in-place, descending
sorted_nums = sorted(nums)         # returns new list

# Custom key
words.sort(key=len)                # by length
words.sort(key=str.lower)          # case-insensitive
pairs.sort(key=lambda x: x[1])    # by second element

# Multiple criteria
data.sort(key=lambda x: (x[0], -x[1]))  # asc by [0], desc by [1]

# Sort dict by value
sorted_d = dict(sorted(d.items(), key=lambda x: x[1]))

# Custom comparator (rare, but available)
from functools import cmp_to_key
def compare(a, b):
    return -1 if a < b else 1 if a > b else 0
nums.sort(key=cmp_to_key(compare))
```

***Tip:** Sorting with **key=lambda** covers 90% of interview sort problems. Negate numeric values for reverse within tuples.*

11. Classes & OOP

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

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

```
        self.val = val
        self.next = next

# Dataclass (clean alternative)
from dataclasses import dataclass

@dataclass
class Point:
    x: float
    y: float

    def distance(self):
        return (self.x**2 + self.y**2) ** 0.5

# Dunder methods
class Fraction:
    def __init__(self, num, den):
        self.num = num
        self.den = den
    def __repr__(self):
        return f"{self.num}/{self.den}"
    def __lt__(self, other):     # enables sorting
        return self.num * other.den < other.num * self.den
    def __eq__(self, other):
        return self.num * other.den == other.num * self.den
    def __hash__(self):
        return hash((self.num, self.den))
```

12. Stacks & Queues

```
# Stack (LIFO) — use a list
stack = []
stack.append(1)          # push
stack.append(2)
stack.pop()              # 2 — O(1)
stack[-1]                # peek top

# Queue (FIFO) — use deque
from collections import deque
q = deque()
q.append(1)              # enqueue right
q.appendleft(0)          # enqueue left
q.popleft()              # dequeue left — O(1)
q[0]                     # peek front

# Monotonic stack pattern (Next Greater Element)
def next_greater(nums):
    result = [-1] * len(nums)
    stack = []  # indices
    for i, num in enumerate(nums):
        while stack and nums[stack[-1]] < num:
            result[stack.pop()] = num
        stack.append(i)
    return result
```

13. Heaps (Priority Queue)

```
import heapq

# Min-heap (default in Python)
h = []
heapq.heappush(h, 5)
heapq.heappush(h, 1)
heapq.heappush(h, 3)
heapq.heappop(h)          # 1 (smallest)
h[0]                       # peek min

# Max-heap: negate values
heapq.heappush(h, -val)   # push
-heapq.heappop(h)         # pop (negate back)

# Heapify a list in O(n)
nums = [5, 3, 1, 4, 2]
heapq.heapify(nums)       # now a min-heap

# Top-K pattern
heapq.nlargest(3, nums)   # [5, 4, 3]
heapq.nsmallest(3, nums)  # [1, 2, 3]

# Heap with tuples (sorts by first element)
h = []
heapq.heappush(h, (dist, node))  # priority, data
```

***Tip:** For max-heap, always negate: push **-val**, pop **-heappop(h)**. Tuples sort by first element.*

14. Hash Maps — Common Patterns

```
# Two-Sum pattern
def two_sum(nums, target):
    seen = {}
    for i, num in enumerate(nums):
        comp = target - num
        if comp in seen:
```

```
            return [seen[comp], i]
        seen[num] = i

# Frequency count
from collections import Counter
freq = Counter(nums)
# or manually:
freq = {}
for x in nums:
    freq[x] = freq.get(x, 0) + 1

# Group anagrams
from collections import defaultdict
groups = defaultdict(list)
for word in words:
    key = tuple(sorted(word))
    groups[key].append(word)

# Prefix sum with hashmap (Subarray Sum = K)
def subarray_sum(nums, k):
    count, curr_sum = 0, 0
    prefix = {0: 1}
    for num in nums:
        curr_sum += num
        count += prefix.get(curr_sum - k, 0)
        prefix[curr_sum] = prefix.get(curr_sum, 0) + 1
    return count
```

15. Sliding Window

```
# Fixed-size window
def max_sum_subarray(nums, k):
    window_sum = sum(nums[:k])
    max_sum = window_sum
    for i in range(k, len(nums)):
        window_sum += nums[i] - nums[i - k]
        max_sum = max(max_sum, window_sum)
    return max_sum

# Variable-size window (shrink when invalid)
def longest_unique_substring(s):
    seen = {}
    left = max_len = 0
    for right, ch in enumerate(s):
        if ch in seen and seen[ch] >= left:
            left = seen[ch] + 1
        seen[ch] = right
        max_len = max(max_len, right - left + 1)
    return max_len
```

16. Two Pointers

```
# Opposite ends (sorted array)
def two_sum_sorted(nums, target):
    lo, hi = 0, len(nums) - 1
    while lo < hi:
        s = nums[lo] + nums[hi]
        if s == target:   return [lo, hi]
        elif s < target:  lo += 1
        else:             hi -= 1

# Same direction (remove duplicates)
def remove_dups(nums):
    if not nums: return 0
    slow = 0
    for fast in range(1, len(nums)):
        if nums[fast] != nums[slow]:
            slow += 1
            nums[slow] = nums[fast]
    return slow + 1

# Three-pointer / 3Sum
def three_sum(nums):
    nums.sort()
    result = []
    for i in range(len(nums) - 2):
        if i > 0 and nums[i] == nums[i-1]: continue
        lo, hi = i+1, len(nums)-1
        while lo < hi:
            s = nums[i] + nums[lo] + nums[hi]
            if s == 0:
                result.append([nums[i], nums[lo], nums[hi]])
                while lo < hi and nums[lo] == nums[lo+1]: lo += 1
                while lo < hi and nums[hi] == nums[hi-1]: hi -= 1
                lo += 1; hi -= 1
            elif s < 0: lo += 1
            else:       hi -= 1
    return result
```

17. Binary Search

```
# Standard binary search
def binary_search(nums, target):
```

```
    lo, hi = 0, len(nums) - 1
    while lo <= hi:
        mid = lo + (hi - lo) // 2     # avoids overflow
        if nums[mid] == target:   return mid
        elif nums[mid] < target:  lo = mid + 1
        else:                     hi = mid - 1
    return -1  # not found

# Left bisect (first position >= target)
def bisect_left(nums, target):
    lo, hi = 0, len(nums)
    while lo < hi:
        mid = (lo + hi) // 2
        if nums[mid] < target:  lo = mid + 1
        else:                   hi = mid
    return lo

# Binary search on answer (minimize/maximize)
def min_days(bloomDay, m, k):
    lo, hi = min(bloomDay), max(bloomDay)
    while lo < hi:
        mid = (lo + hi) // 2
        if can_make(bloomDay, m, k, mid):
            hi = mid
        else:
            lo = mid + 1
    return lo

# bisect module
import bisect
bisect.bisect_left(sorted_list, target)
bisect.bisect_right(sorted_list, target)
bisect.insort(sorted_list, val)  # insert maintaining order
```

Tip: *Use* ***bisect*** *module for clean code.* ***bisect_left*** *= first >=,* ***bisect_right*** *= first >.*

18. Trees — Traversals & Patterns

```
# Definition (usually given)
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

# DFS — Recursive
def inorder(root):      # Left, Root, Right
    if not root: return []
    return inorder(root.left) + [root.val] + inorder(root.right)

def preorder(root):     # Root, Left, Right
    if not root: return []
    return [root.val] + preorder(root.left) + preorder(root.right)

def postorder(root):    # Left, Right, Root
    if not root: return []
    return postorder(root.left) + postorder(root.right) + [root.val]

# BFS — Level Order
from collections import deque
def level_order(root):
    if not root: return []
    result, queue = [], deque([root])
    while queue:
        level = []
        for _ in range(len(queue)):
            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

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

19. Graphs — BFS & DFS

```
# Adjacency list representation
graph = defaultdict(list)
for u, v in edges:
    graph[u].append(v)
    graph[v].append(u)     # undirected

# BFS (shortest path in unweighted graph)
def bfs(graph, start):
    visited = {start}
    queue = deque([start])
    dist = {start: 0}
    while queue:
        node = queue.popleft()
        for neighbor in graph[node]:
            if neighbor not in visited:
                visited.add(neighbor)
                dist[neighbor] = dist[node] + 1
                queue.append(neighbor)
    return dist

# DFS (iterative)
```

```
def dfs(graph, start):
    visited = set()
    stack = [start]
    while stack:
        node = stack.pop()
        if node in visited: continue
        visited.add(node)
        for neighbor in graph[node]:
            if neighbor not in visited:
                stack.append(neighbor)

# DFS (recursive) — useful for connected components
def dfs_recursive(node, visited, graph):
    visited.add(node)
    for neighbor in graph[node]:
        if neighbor not in visited:
            dfs_recursive(neighbor, visited, graph)

# Number of connected components
def count_components(n, edges):
    graph = defaultdict(list)
    for u, v in edges:
        graph[u].append(v); graph[v].append(u)
    visited = set()
    count = 0
    for i in range(n):
        if i not in visited:
            dfs_recursive(i, visited, graph)
            count += 1
    return count
```

20. Dynamic Programming

```
# Template: identify state, recurrence, base case

# 1D DP — Fibonacci / Climbing Stairs
def climb_stairs(n):
    if n <= 2: return n
    dp = [0] * (n + 1)
    dp[1], dp[2] = 1, 2
    for i in range(3, n + 1):
        dp[i] = dp[i-1] + dp[i-2]
    return dp[n]

# Space-optimized 1D
def climb_stairs_opt(n):
    a, b = 1, 2
    for _ in range(3, n + 1):
        a, b = b, a + b
    return b

# 2D DP — Grid paths
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]

# Knapsack (0/1)
def knapsack(weights, values, capacity):
    n = len(weights)
    dp = [[0]*(capacity+1) for _ in range(n+1)]
    for i in range(1, n+1):
        for w in range(capacity+1):
            dp[i][w] = dp[i-1][w]
            if weights[i-1] <= w:
                dp[i][w] = max(dp[i][w],
                    dp[i-1][w-weights[i-1]] + values[i-1])
    return dp[n][capacity]
```

21. Backtracking

```
# General template
def backtrack(candidates, path, result, start):
    if is_valid(path):          # base case / goal
        result.append(path[:])
        return
    for i in range(start, len(candidates)):
        path.append(candidates[i])          # choose
        backtrack(candidates, path, result, i + 1)  # explore
        path.pop()                           # un-choose

# Permutations
def permutations(nums):
    result = []
    def bt(path, used):
        if len(path) == len(nums):
            result.append(path[:])
            return
        for i in range(len(nums)):
            if used[i]: continue
            used[i] = True
            path.append(nums[i])
            bt(path, used)
            path.pop()
```

```
            used[i] = False
    bt([], [False]*len(nums))
    return result

# Subsets
def subsets(nums):
    result = []
    def bt(start, path):
        result.append(path[:])
        for i in range(start, len(nums)):
            path.append(nums[i])
            bt(i + 1, path)
            path.pop()
    bt(0, [])
    return result
```

22. Matrix / Grid Patterns

```
# Dimensions
rows, cols = len(grid), len(grid[0])

# 4-directional movement
directions = [(0,1),(0,-1),(1,0),(-1,0)]
for dr, dc in directions:
    nr, nc = r + dr, c + dc
    if 0 <= nr < rows and 0 <= nc < cols:
        # valid neighbor

# BFS on grid (flood fill / island count)
def num_islands(grid):
    rows, cols = len(grid), len(grid[0])
    visited = set()
    count = 0
    def bfs(r, c):
        queue = deque([(r, c)])
        visited.add((r, c))
        while queue:
            cr, cc = queue.popleft()
            for dr, dc in [(0,1),(0,-1),(1,0),(-1,0)]:
                nr, nc = cr+dr, cc+dc
                if (0 <= nr < rows and 0 <= nc < cols
                    and (nr,nc) not in visited
                    and grid[nr][nc] == "1"):
                    visited.add((nr, nc))
                    queue.append((nr, nc))
    for r in range(rows):
        for c in range(cols):
            if grid[r][c] == "1" and (r,c) not in visited:
                bfs(r, c)
                count += 1
    return count

# Rotate matrix 90 degrees clockwise
def rotate(matrix):
    n = len(matrix)
    # Transpose
    for i in range(n):
        for j in range(i+1, n):
            matrix[i][j], matrix[j][i] = matrix[j][i], matrix[i][j]
    # Reverse each row
    for row in matrix:
        row.reverse()
```

23. Linked Lists

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

# Reverse a linked list
def reverse(head):
    prev, curr = None, head
    while curr:
        nxt = curr.next
        curr.next = prev
        prev = curr
        curr = nxt
    return prev

# Detect cycle (Floyd's)
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
def find_middle(head):
    slow = fast = head
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next
    return slow

# Merge two sorted lists
def merge(l1, l2):
    dummy = ListNode(0)
    curr = dummy
    while l1 and l2:
        if l1.val <= l2.val:
            curr.next = l1; l1 = l1.next
        else:
            curr.next = l2; l2 = l2.next
        curr = curr.next
    curr.next = l1 or l2
    return dummy.next
```

24. Interval Problems

```
# Merge overlapping intervals
def merge_intervals(intervals):
    intervals.sort()
    merged = [intervals[0]]
    for start, end in intervals[1:]:
        if start <= merged[-1][1]:
            merged[-1][1] = max(merged[-1][1], end)
        else:
            merged.append([start, end])
    return merged

# Insert interval
def insert(intervals, new):
    result = []
    for i, (s, e) in enumerate(intervals):
        if new[1] < s:
            result.append(new)
            return result + intervals[i:]
        elif new[0] > e:
            result.append([s, e])
        else:
            new = [min(new[0], s), max(new[1], e)]
    result.append(new)
    return result
```

25. Union-Find (Disjoint Set)

```
class UnionFind:
    def __init__(self, n):
        self.parent = list(range(n))
        self.rank = [0] * n
        self.count = n          # number of components

    def find(self, x):
        if self.parent[x] != x:
            self.parent[x] = self.find(self.parent[x])  # path compression
        return self.parent[x]

    def union(self, x, y):
        px, py = self.find(x), self.find(y)
        if px == py: return False
        if self.rank[px] < self.rank[py]: px, py = py, px
        self.parent[py] = px
        if self.rank[px] == self.rank[py]: self.rank[px] += 1
        self.count -= 1
        return True
```

26. Essential Standard Library

```
import math
math.gcd(12, 8)        # 4
math.lcm(4, 6)         # 12     (Python 3.9+)
math.ceil(3.2)         # 4
math.floor(3.8)        # 3
math.log2(8)           # 3.0
math.inf               # positive infinity
-math.inf              # negative infinity

import itertools
itertools.permutations([1,2,3])        # all permutations
itertools.combinations([1,2,3], 2)     # C(3,2) pairs
itertools.product([0,1], repeat=3)     # cartesian: 000..111
itertools.accumulate([1,2,3,4])        # prefix sums: 1,3,6,10
```

```
itertools.chain([1,2], [3,4])          # flatten iterables

import string
string.ascii_lowercase   # 'abcdefghijklmnopqrstuvwxyz'
string.ascii_uppercase   # 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
string.digits            # '0123456789'

from functools import lru_cache
@lru_cache(maxsize=None)    # memoization decorator
def fib(n):
    if n <= 1: return n
    return fib(n-1) + fib(n-2)

import sys
sys.maxsize               # ~9.2 * 10^18
-sys.maxsize              # use as -infinity for ints
```

27. Big-O Complexity Quick Reference

Operation / Structure	Time	Notes
List: index / assign	O(1)	
List: append / pop (end)	O(1)	amortized
List: insert / pop(0) / del	O(n)	use deque for O(1) front
List: sort	O(n log n)	Timsort
List: in / index / count	O(n)	linear scan
Dict: get / set / del / in	O(1)	avg case, O(n) worst
Set: add / discard / in	O(1)	avg case
Heap: push / pop	O(log n)	heapq module
Heap: heapify	O(n)	
Binary Search	O(log n)	sorted input required
BFS / DFS (graph)	O(V + E)	V=vertices, E=edges
String slicing s[a:b]	O(b - a)	creates new string
String concatenation +=	O(n)	per operation — use join
Deque: appendleft / popleft	O(1)	
bisect_left / insort	O(log n) / O(n)	search / insert

28. CodeSignal Tips & Common Gotchas

```
# ■■ Input parsing (CodeSignal typically provides parsed input) ■■
# But if needed:
n = int(input())
nums = list(map(int, input().split()))

# ■■ Common gotchas ■■
# 1. Integer division truncates toward NEGATIVE infinity
#    -7 // 2 = -4   (not -3!)
#    Use int(-7 / 2) = -3 for truncation toward zero

# 2. Mutable default arguments
def bad(lst=[]):   # WRONG — shared across calls!
    lst.append(1)
def good(lst=None):# CORRECT
    if lst is None: lst = []

# 3. Shallow vs deep copy
import copy
a = [[1,2],[3,4]]
b = a[:]              # shallow copy — inner lists shared!
c = copy.deepcopy(a)  # deep copy — fully independent

# 4. Global in nested scope
count = 0
def f():
    nonlocal count     # for enclosing function scope
    # global count     # for module-level scope

# 5. String building — O(n) with join, O(n^2) with +=
```

```
chars = []
for c in s:
    chars.append(c.upper())
result = ''.join(chars)
```

29. Problem-Solving Pattern Cheat Sheet

If You See...	Think...	Key Structure
"find pair/sum"	Two Pointers / HashMap	dict, sorted arr
"contiguous subarray"	Sliding Window / Prefix Sum	dict {sum: count}
"sorted array search"	Binary Search	lo, hi, mid
"top K / Kth largest"	Heap / QuickSelect	heapq
"tree traversal"	BFS (level) / DFS (recursive)	deque / stack
"shortest path"	BFS (unweighted) / Dijkstra	deque / heap
"connected components"	DFS / BFS / Union-Find	visited set
"overlapping subproblems"	Dynamic Programming	dp[] / memo
"all combinations/perms"	Backtracking	path + recurse
"merge intervals"	Sort + Sweep	sort by start
"string matching"	HashMap / Sliding Window	Counter/dict
"grid/matrix search"	BFS/DFS + directions array	visited set
"parentheses/brackets"	Stack	stack = []
"find cycle"	Floyd's (slow/fast)	two pointers