Data Structures & Common Python Patterns (OA Cheatsheet)

Code lines	What it achieves	Notes (only if necessary)

d = {}
d[key] = val
val = d.get(key)
	Hash table insert + safe lookup	`get` avoids `KeyError`

val = d.get(key, default)	Lookup with explicit default	Very common for “return null/false if missing”
if key in d:	Membership test	Average O(1)
d.pop(key, None)	Safe delete	Returns removed value or `None`
for k, v in d.items():	Iterate key/value	Prefer `items()` when you need values
from collections import defaultdict dd = defaultdict(list) dd[k].append(x)	Dict of lists without guards	For grouping / adjacency lists
dd = defaultdict(set) dd[k].add(x)	Dict of sets	Inverted indexes, ownership sets

lst = d.setdefault(k, [])
lst.append(x)
	Default-init without defaultdict	Keeps plain dict semantics

s = set()
s.add(x)
x in s
	Set insert + membership	Average O(1)

s.discard(x)	Remove if present (no error)	Safer than `remove`

from collections import Counter
cnt = Counter(arr)
cnt[x] += 1
	Frequency counting	`Counter` behaves like dict

from collections import deque
q = deque()
q.append(x)
x = q.popleft()
	Queue (FIFO)	Don’t use `list.pop(0)`

stack = []
stack.append(x)
x = stack.pop()
	Stack (LIFO)	Standard

import heapq
h = []
heapq.heappush(h, x)
x = heapq.heappop(h)
	Min-heap priority queue	Heap is min-only

heapq.heappush(h, (prio, item)) prio, item = heapq.heappop(h)	Heap with priorities	Tuples compare lexicographically

heapq.heappush(h, (-prio, item))
p, item = heapq.heappop(h)
prio = -p
	Max-heap behavior	Negate priority

heapq.heapify(arr)	Build heap in-place	O(n) build

import bisect
i = bisect.bisect_left(a, x)
	Binary search position	`a` must be sorted

bisect.insort(a, x)	Insert while keeping list sorted	Insert is O(n)
a.sort()	In-place sort	Stable sort in Python
a.sort(key=lambda x: (-x.size, x.name))	Multi-key sort	Common “size desc, name asc” pattern
best = max(items, key=lambda it: (it.size, it.name))	Argmax with tie-break	Tuple tie-break is automatic
res = [x for x in arr if pred(x)]	Filter with list comprehension	Often clearer than `filter()`
res = [f(x) for x in arr]	Map/transform with list comprehension	Often clearer than `map()`
res = [f(x) for x in arr if pred(x)]	Filter + map in one pass	Keep it readable (don’t nest too much)
mp = {k: v for k, v in pairs}	Dict comprehension	Quick remap/build
st = {x for x in arr if pred(x)}	Set comprehension	Dedup + filter
any(pred(x) for x in arr)	Existence check	Short-circuits
all(pred(x) for x in arr)	Universal check	Short-circuits
total = sum(x.size for x in items)	Aggregate derived values	Generator avoids temp list

op = q[0]
args = q[1:]
	Clean query parsing	Reduces indexing errors

op, ts, key, field = q	Unpacking fixed-arity queries	Only if you’re sure of length
from itertools import groupby items.sort(key=lambda x: x.cat) for cat, grp in groupby(items, key=lambda x: x.cat):	Grouping by key	Must sort by same key first
from itertools import accumulate pref = list(accumulate(arr))	Prefix sums	Useful for range sums

import copy
snap = copy.deepcopy(state)
	Deep copy state	Needed for backups/snapshots

a2 = a[:]
d2 = dict(d)
	Shallow copies	Won’t copy nested objects

seen = set()
for x in arr:
if x in seen: ...
seen.add(x)
	Duplicate detection	O(n) average

i, j = 0, len(a)-1
while i < j:
s = a[i] + a[j]
...
	Two pointers on sorted array	Requires sorted input

left = 0
for right, x in enumerate(arr):
add(x)
while bad():
remove(arr[left]); left += 1
	Sliding window template	Define `add/remove/bad`