Module 20 of 20

Interview idioms

The round and the file template with fast I/O, the idiom sheet of shapes and their Python, the twelve pitfalls interviewers watch for, a hash map, dynamic array, LRU cache and heap by hand, clean solutions under pressure, and the Python theory drill.

Nineteen modules taught the language. This one teaches the round: how a Python interview or contest problem is actually conducted, what is actually judged, and the small set of habits that turn knowing Python into passing.

It opens with the rhythm — restate, example, brute force, shape, complexity, code, test — and the file template every judged solution starts from: all tokens read once, `main` plus a `solve`, the recursion limit when recursing, batched output, each line with its cost so it can be defended. The idiom sheet names the shapes problems take and the two-to-six-line Python for each: counting with `Counter`, sorting by tuple key, windows and two pointers, prefix sums, stacks, heaps, BFS with a deque, `bisect`, set arithmetic. The pitfalls lesson is the twelve one-line mistakes — mutable defaults, the shared-row grid, late-binding closures, `is` for values, mutating while iterating, `.sort()` returning `None`, floor division, float equality, the O(n²) string, the recursion limit, `or` returning operands, the broad `except` — as a checklist. Implement the built-in builds a chained hash map with resize, a doubling array, an `OrderedDict` LRU cache and a binary heap in the shape interviewers expect: invariant, operations, complexity. Writing clean solutions is structure, names, edges first and tests in the file. The theory drill is thirty questions with the answers an interviewer wants to hear.

The exercises are the problems interviewers set, written the way the module says to write them: statistics and a region count with the template, anagram groups and the longest window without repeats, a fixer for four planted bugs and fifteen expressions to predict before running, the hash map and the LRU cache by hand, balanced brackets and a leaderboard with pure helpers, the MRO by hand and a generator pipeline whose trace shows laziness. The final checkpoint sets a sliding-window rate limiter, an RPN evaluator with error handling, and an interval merger with `bisect` queries — and, with the other nineteen modules, completes the plan.

CONTENTS

1 The interview template — the round, the file, fast I/O and what is actually judged 14 min

2	The idiom sheet — the shapes interview problems take and the Python for each	16 min
3	Pitfalls that fail interviews — the twelve Python mistakes interviewers watch for	15 min
4	Implement the built-in — a hash map, a dynamic array, an LRU cache and a heap by hand	16 min
5	Writing clean solutions — structure, names, edges first and the code you can read aloud	13 min
6	The Python theory drill — thirty questions, thirty answers	25 min
✓	Final checkpoint — Interview idioms	30 min

Exercises in this module compile on CPython 3.11. Reference solutions and the answer key follow each lesson.

LESSON 1 · 14 MIN

The interview template — the round, the file, fast I/O and what is actually judged

Nineteen modules taught the language; this one teaches the round. A Python interview or contest problem is conducted in a fixed rhythm — clarify, example, brute force, complexity, better idea, code, test — and judged on things that are only partly about Python: whether you asked about the edge cases, whether you named the complexity before coding, whether the code you wrote can be read aloud. This lesson gives the rhythm, the file template that every judged solution starts from (fast input, `main`, the recursion limit, batched output), the cost of each line so you can defend it, and the list of what the person across the table is actually scoring.

The rhythm of a round

Restate the problem in one sentence and confirm it. Ask the questions that change the algorithm: sizes (n up to 10^5 or 10^9?), value ranges, duplicates, empty input, sorted or not, what to return on ties.

Work an example by hand — the one given, then a small one you make up with an edge in it.

Say the brute force and its complexity aloud, even if you will not write it: it shows you understand the problem and gives a baseline.

Find the better idea by naming the shape (lesson 2): "this is a sliding window", "this is a frequency count then a heap", "this is a graph and BFS".

State the complexity of the plan before typing. If n is 10^5 and the plan is $O(n^2)$, say so and keep thinking.

Code top-down: `main` reads and prints, a helper does the work, the helper's name is the idea.

Test by tracing your own example through the code, then the edges: empty, one element, all equal, the boundary.

Silence is the failure mode; narrating the steps above is most of the score.

The file template

```python
import sys
from collections import Counter, defaultdict, deque
from functools import cache
from heapq import heappush, heappop
from bisect import bisect_left, bisect_right

def solve(nums: list[int]) -> int:
    ...

def main() -> None:
    data = sys.stdin.buffer.read().split()          # every token, once
    n = int(data[0])
    nums = list(map(int, data[1:n + 1]))
    print(solve(nums))

if __name__ == "__main__":
    sys.setrecursionlimit(1_000_000)
    main()
```

Each line has a reason you should be able to give:

`sys.stdin.buffer.read().split()` reads the whole input as bytes and splits on whitespace: one system call, no decoding, `int(b"42")` works directly. For 10^5 lines it is ten times faster than `input()` in a loop. When lines matter (a grid, text with spaces), use `sys.stdin.read().splitlines()` instead, or `sys.stdin.readline` for a mixed format.

`main()` keeps the work out of module scope: locals are faster than globals (Module 19) and the function can be imported and tested.

`sys.setrecursionlimit(1_000_000)` allows deep recursion (a DFS on 10^5 nodes); the default 1000 raises `RecursionError` on the first tree deeper than that. On a small C stack a very deep recursion can still crash, so an iterative DFS with an explicit stack is the safer answer for the largest inputs.

The imports are the ones you reach for without thinking; unused imports cost nothing and save a scroll.

Output: build a list of strings and `print("\n".join(out))` or `sys.stdout.write` once. `print` per line is a system call per line.

Reading the formats

Format	Read with
`n` then `n` ints on one line	`data = read().split(); n = int(data[0]); nums = map(int, data[1:n+1])`
`n` lines of `a b`	tokens in pairs: `zip(it, it)` over an iterator of the tokens
a grid of characters	`lines = sys.stdin.read().split()` — rows have no spaces
lines with spaces	`sys.stdin.read().splitlines()`
until EOF	`for line in sys.stdin:`
a single line of words	`input().split()`

`it = iter(map(int, data)); n = next(it)` then `next(it)` per value is the cleanest way to walk a token stream with a mixed layout.

The costs, honestly

Fast I/O matters only when the input is large: for ten lines, `input()` is fine and clearer. The recursion limit matters only for recursive solutions on deep structures. `main()` matters always. The wildcard imports and one-letter names of contest code cost readability that an interviewer is scoring, so the template is a middle path: the fast lines, standard names, and a solve function whose signature says what it does.

What is judged

- Did you clarify sizes and edges before designing?
- Did you name the brute force and its complexity?
- Did you name the shape and the data structure, and the complexity of the plan?
- Is the code readable aloud: a `solve` with a good name, helpers, no globals, no clever one-liners?
- Did you test with your own example and the edges without being asked?
- Did you handle the edge that was in the question (empty, one element, negative, duplicates)?
- Can you say what you would change for 10× the input, or for production?

Correctness is necessary and not sufficient; a working solution with none of the above scores lower than a nearly working one with all of them.

Pitfalls

- Starting to type before restating the problem.
- `input()` in a loop over 10^5 lines.
- Recursion on 10^5-deep structures without raising the limit — or with it, on a machine with a small stack.
- Printing inside the loop that computes, then being unable to test the function.
- Not asking what to do on ties or empty input, then guessing wrong.
- Ending without testing an edge case.

Key takeaways

- The rhythm: restate, example, brute force, shape, complexity, code, test — narrated aloud.
- The template: read all tokens once, `main` plus a `solve`, the recursion limit when recursing, batched output.
- Every template line has a cost and a reason; use the fast lines when the input is large and say why.
- Mixed input layouts are walked with an iterator over the tokens.
- The score is clarification, complexity, readability and testing — not just a green check.

EXERCISE 1

Statistics with the template

Using the template — `sys.stdin.buffer.read().split()`, a `solve` function and `main` — read `n` followed by `n` integers (which may span several lines) and print `count <n>`, `sum <s>`, `min <m>`, `max <M>`, `mean <mean:.2f>` and `median <median:.1f>` (the median of an even count is the mean of the two middle values).

Input: `n`, then `n` integers. **Output:** six lines.

```
5
3 1 4 1 5
```

prints

```
count 5
sum 14
min 1
max 5
mean 2.80
median 3.0
```

INPUT 1

```
5
3 1 4 1 5
```

OUTPUT 1

```
count 5
sum 14
min 1
max 5
mean 2.80
median 3.0
```

REFERENCE SOLUTION

```python
import sys


def solve(nums: list[int]) -> list[str]:
    ordered = sorted(nums)
    n = len(ordered)
    mid = n // 2
    median = ordered[mid] if n % 2 else (ordered[mid - 1] + ordered[mid]) / 2
    return [
        f"count {n}",
        f"sum {sum(nums)}",
        f"min {ordered[0]}",
        f"max {ordered[-1]}",
        f"mean {sum(nums) / n:.2f}",
        f"median {median:.1f}",
    ]


def main() -> None:
    data = sys.stdin.buffer.read().split()
    n = int(data[0])
    nums = [int(tok) for tok in data[1:n + 1]]
    print("\n".join(solve(nums)))


if __name__ == "__main__":
    main()
```

EXERCISE 2

Count the regions

Read `r c` and then `r` rows of `.` and `#`. Count the 4-connected regions of `#` with a breadth-first search (a `deque` and a `seen` set, no recursion) and print `regions <count>` and `largest <size of the largest region>` (0 when there is none).

Input: the dimensions, then the grid. **Output:** two lines.

```
4 5
##..#
#...#
..#..
..##.
```

prints

```
regions 3
largest 3
```

INPUT 1

```
4 5
##..#
#...#
..#..
..##.
```

OUTPUT 1

```
regions 3
largest 3
```

REFERENCE SOLUTION

```python
import sys
from collections import deque


def region_sizes(grid: list[str]) -> list[int]:
    rows, cols = len(grid), len(grid[0]) if grid else 0
    seen = set()
    sizes = []
    for r0 in range(rows):
        for c0 in range(cols):
            if grid[r0][c0] != "#" or (r0, c0) in seen:
                continue
            queue = deque([(r0, c0)])
            seen.add((r0, c0))
            size = 0
            while queue:
                r, c = queue.popleft()
                size += 1
                for dr, dc in ((1, 0), (-1, 0), (0, 1), (0, -1)):
                    nr, nc = r + dr, c + dc
                    if 0 <= nr < rows and 0 <= nc < cols and grid[nr][nc] == "#" and
                        seen.add((nr, nc))
                        queue.append((nr, nc))
            sizes.append(size)
    return sizes


def main() -> None:
    tokens = sys.stdin.read().split()
    rows, cols = int(tokens[0]), int(tokens[1])
    grid = tokens[2:2 + rows]
    sizes = region_sizes(grid)
    print("regions", len(sizes))
    print("largest", max(sizes, default=0))


if __name__ == "__main__":
    main()
```

QUIZ

1. What is the first thing to do when given a problem in a round?

A. Start typing the brute force

B. Restate it and ask about sizes and edge cases

C. Ask for the answer

D. Write tests

2. Why `sys.stdin.buffer.read().split()` rather than `input()` in a loop for 10^5 lines?

A. It is shorter

B. One system call and no decoding — roughly ten times faster

C. `input()` is deprecated

D. It handles Unicode

3. What does `sys.setrecursionlimit(1_000_000)` allow, and what remains a risk?

A. Nothing

B. Deep recursion past the default 1000 frames; a very deep recursion can still exhaust the C stack

C. Faster recursion

D. Tail calls

4. Why keep the work in `main()` and a `solve()` rather than at module level?

A. Style only

B. Locals are faster than globals and the function can be imported and tested

C. It is required

D. Globals are not allowed

5. Which of these is scored in a round beyond correctness?

A. Typing speed

B. Naming the complexity before coding and testing the edges unasked

C. Using the fewest lines

D. Avoiding imports

ANSWER KEY · THE INTERVIEW TEMPLATE — THE ROUND, THE FILE, FAST I/O AND WHAT IS ACTUALLY JUDGED

1. B The sizes decide the algorithm; the edges decide the code; silence is the failure mode.

2. B For ten lines it does not matter and `input()` is clearer.

3. B An iterative DFS with an explicit stack is the safe answer for the largest inputs.

4. B A solve function whose name is the idea is also what the interviewer reads.

5. B Clarification, complexity, readability and testing are most of the score.

LESSON 2 · 16 MIN

The idiom sheet — the shapes interview problems take and the Python for each

Most interview problems are one of about twenty shapes wearing a story, and each shape has a two-to-six-line Python idiom that a fluent candidate types without thinking. Recognising the shape is the skill; the idiom is the reward. This lesson is the sheet: counting and grouping, sorting by key, two pointers and sliding windows, prefix sums, stacks, heaps, BFS with a deque, binary search with `bisect`, set arithmetic, string building, and the `itertools` and `math` helpers — each named, each with its idiom and its complexity, so that in the round you say "this is a frequency count, then a heap" and write it.

Counting and grouping

```
from collections import Counter, defaultdict
counts = Counter(words)                          # O(n)
counts.most_common(3)                            # top three (ties in insertion o
groups = defaultdict(list)
for word in words:
    groups["".join(sorted(word))].append(word)    # anagram groups
by_first = {}
for name in names:
    by_first.setdefault(name[0], []).append(name)
```

"How many of each", "group by", "find duplicates", "first unique" are all a `Counter` or a `defaultdict`. `Counter` supports `+`, `-`, `&` and `|` between counters, and `counts[x]` is 0 for a missing key.

Sorting by key

```
sorted(items, key=lambda p: (-p.score, p.name))     # descending score, then n
sorted(words, key=len)                               # stable: equal lengths k
max(items, key=lambda p: p.score)                    # first maximum
sorted(pairs)                                        # tuples compare lexicogr
```

Negate numbers for descending inside a tuple key; `reverse=True` reverses the whole order. Stability means two sorts compose: sort by the secondary key first, then by the primary.

Two pointers and sliding windows

```
# longest substring without a repeated character: O(n)
last = {}
start = best = 0
for i, ch in enumerate(s):
    if ch in last and last[ch] ⩾ start:
        start = last[ch] + 1
    last[ch] = i
    best = max(best, i - start + 1)

# pair with a target sum in a sorted list
lo, hi = 0, len(xs) - 1
while lo < hi:
    total = xs[lo] + xs[hi]
    if total == target: break
    if total < target: lo += 1
    else: hi -= 1
```

A window is a start index that only moves forward and a state (a count, a set, a sum) updated as elements enter and leave. "Longest/shortest subarray with property P", "at most k distinct" are windows. Two pointers walk a sorted array from both ends, or two arrays in step (merging).

Prefix sums and differences

```
prefix = [0]
for x in xs: prefix.append(prefix[-1] + x)
range_sum = prefix[j + 1] - prefix[i]              # O(1) per query
from itertools import accumulate
prefix = [0, *accumulate(xs)]                      # the same in one line
```

"Sum of a range", "count of something in a range", "subarray with sum k" (prefix sums plus a dict of seen prefixes) are prefix problems.

Stacks

```
stack = []
for ch in s:                                          # balanced brackets
    if ch in "([{": stack.append(ch)
    elif not stack or PAIRS[stack.pop()] ≠ ch: return False
return not stack

for i, h in enumerate(heights):                       # next greater element / r
    while stack and heights[stack[-1]] < h:
        result[stack.pop()] = h
    stack.append(i)
```

Brackets, "next greater", "largest rectangle", expression evaluation, undo, DFS without recursion — a list used with `append` and `pop`.

Heaps

```
import heapq
heapq.nlargest(k, xs)                                 # O(n log k)
heapq.nsmallest(k, words, key=lambda w: (counts[w], w))
heap = []; heapq.heappush(heap, (dist, node)); dist, node = heapq.heappop(heap
heapq.heappush(heap, (-value, item))                  # a max-heap by negation
heapq.merge(a, b, c)                                  # k sorted iterables, lazi
```

"Top k", "k-th largest", "merge k sorted", Dijkstra, "schedule by earliest" are heaps. `heapify(xs)` is O(n).

BFS, DFS and graphs

```
from collections import deque
graph = defaultdict(list)
for a, b in edges: graph[a].append(b); graph[b].append(a)

dist = {start: 0}
queue = deque([start])
while queue:                                        # BFS: shortest path in ea
    node = queue.popleft()
    for nxt in graph[node]:
        if nxt not in dist:
            dist[nxt] = dist[node] + 1
            queue.append(nxt)

for dr, dc in ((1, 0), (-1, 0), (0, 1), (0, -1)):   # grid neighbours
    r, c = row + dr, col + dc
    if 0 ⩽ r < rows and 0 ⩽ c < cols and grid[r][c] == "#" and (r, c) not in
```

"Shortest path in an unweighted graph or grid", "count islands", "is it connected" are BFS with a deque and a `seen` set; DFS is the same with a stack (or recursion with the limit raised).

Binary search

```
from bisect import bisect_left, bisect_right, insort
i = bisect_left(sorted_xs, x)                       # first index with xs[i]
count_less = bisect_left(sorted_xs, x); count_le = bisect_right(sorted_xs, x)
lo, hi = 0, n                                       # search on the answer: s
while lo < hi:
    mid = (lo + hi) // 2
    if ok(mid): hi = mid
    else: lo = mid + 1
```

"First element not below x", "how many are less than x", and "the smallest value for which a predicate holds" (capacity, speed, days — the answer is monotonic).

Sets, strings, itertools, math

```
a & b, a | b, a - b, a ^ b                              # intersection, union, d
"".join(reversed(s)); s[::-1]; s.split(); " ".join(words)
ord(ch) - ord("a"); chr(ord("a") + k)
s.translate(str.maketrans("", "", ".,!?"))             # strip punctuation in (
from itertools import combinations, permutations, product, groupby, accumulate
for key, run in groupby(sorted(xs)): ...               # runs of equal values
for a, b in pairwise(xs): ...                          # adjacent pairs
from math import gcd, lcm, isqrt, comb, inf
list(zip(*grid))                                       # transpose
divmod(seconds, 60); n.bit_count(); f"{x:.2f}"
```

The sheet, in one table

Words in the problem	Shape	Idiom
how many, duplicates, group	count	`Counter`, `defaultdict`
top k, k-th, merge sorted	heap	`nlargest`, `heappush`
longest/shortest subarray with	window	start index + state
pair sums, sorted arrays	two pointers	`lo` / `hi`
range sum, subarray sum k	prefix	`accumulate`, dict of prefixes
brackets, next greater	stack	list `append` / `pop`
shortest path, islands, reachable	BFS/DFS	`deque`, `seen` set
first ≥, count <, smallest that works	binary search	`bisect`, search on answer
by score then name	sort by key	tuple key
common, distinct, either	sets	`&`, `|`, `-`

Key takeaways

Name the shape first; the idiom follows. Counting, sorting by key, windows, prefixes, stacks, heaps, BFS, bisect and sets cover most rounds.

`Counter` / `defaultdict` for counting and grouping; tuple keys with negation for multi-key sorts.

Windows and two pointers make O(n²) scans O(n); prefix sums make range queries O(1).

`heapq` for top-k and scheduling, `deque` for BFS, `bisect` for sorted queries and "search on the answer".

`itertools` and `math` have the helper you were about to write.

EXERCISE 1

Group anagrams

Read one line of words. Group words that are anagrams of each other with a `defaultdict(list)` keyed by the word's sorted letters. Print each group on its own line — words in input order, groups in order of first appearance — then `groups <count>`.

Input: one line of words. **Output:** one line per group, then the count.

```
eat tea tan ate nat bat
```

prints

```
eat tea ate
tan nat
bat
groups 3
```

INPUT 1

```
eat tea tan ate nat bat
```

OUTPUT 1

```
eat tea ate
tan nat
bat
groups 3
```

REFERENCE SOLUTION

```python
from collections import defaultdict


def group_anagrams(words: list[str]) -> list[list[str]]:
    groups = defaultdict(list)
    for word in words:
        groups["".join(sorted(word))].append(word)
    return list(groups.values())


groups = group_anagrams(input().split())
for group in groups:
    print(" ".join(group))
print("groups", len(groups))
```

EXERCISE 2

Longest window without repeats

Read one word. Find the longest substring with no repeated character in O(n) with a sliding window — a `start` index and a dict of each character's last index. Print `length <n>` and `window <substring>` (the first such substring when several have the maximum length).

Input: one word. **Output:** two lines.

```
abcabcbb
```

prints

```
length 3
window abc
```

INPUT 1

```
abcabcbb
```

OUTPUT 1

```
length 3
window abc
```

REFERENCE SOLUTION

```python
def longest_unique(s: str) -> str:
    last = {}
    start = 0
    best_start, best_len = 0, 0
    for i, ch in enumerate(s):
        if ch in last and last[ch] >= start:
            start = last[ch] + 1
        last[ch] = i
        if i - start + 1 > best_len:
            best_start, best_len = start, i - start + 1
    return s[best_start:best_start + best_len]


window = longest_unique(input().strip())
print("length", len(window))
print("window", window)
```

QUIZ

1. "Find the k most frequent words" is which shape?

A. Two pointers

B. A frequency count, then a heap (`nsmallest` with a key) or a sort

C. Binary search

D. BFS

2. How do you sort by score descending and then name ascending in one call?

A. `sorted(xs, reverse=True)`

B. `sorted(xs, key=lambda p: (-p.score, p.name))`

C. Two `reverse` flags

D. `sorted(xs, key=score)` then `reverse()`

3. What is the idiom for shortest path in an unweighted grid?

A. Recursion

B. BFS with a `deque` and a `seen` set

C. Dijkstra with a heap

D. Sorting

4. What does `bisect_left(xs, x)` return on a sorted list?

A. Whether `x` is present

B. The first index whose element is not below `x` — also the count of elements less than `x`

C. The last index of `x`

D. The median

5. Which expression transposes a grid stored as a list of rows?

A. `grid[::-1]`

B. `list(zip(*grid))`

C. `reversed(grid)`

D. `grid.T`

ANSWER KEY · THE IDIOM SHEET — THE SHAPES INTERVIEW PROBLEMS TAKE AND THE PYTHON FOR EACH

1. B `Counter` then `heapq.nsmallest(k, counts, key=lambda w: (-counts[w], w))`.

2. B Negate the number inside the tuple key; `reverse=True` would flip the name order too.

3. B Each layer of the BFS is one more step; a heap is for weighted edges.

4. B `bisect_right` gives the count of elements ≤ `x`.

5. B Unpacking the rows into `zip` pairs up the columns.

LESSON 3 · 15 MIN

Pitfalls that fail interviews — the twelve Python mistakes interviewers watch for

Interviewers keep a short list of Python mistakes, because the same twelve appear in most rounds and each one says something about how well the candidate knows the language. None is exotic; all are one line. This lesson names them, shows each as it appears in practice, explains the mechanism (which is what the follow-up question will ask), and gives the fix. Read it as a checklist to run over your own code before you say "done": mutable defaults, the shared-row grid, late-binding closures, `is` for values, modifying a list while iterating, `.sort()` returning `None`, integer division and modulo with negatives, float equality, the O(n²) string, the recursion limit, `and` / `or` returning operands, and the broad `except`.

1. A mutable default argument

```
def add_item(item, items=[]):        # the list is created once, at def time
    items.append(item)
    return items
add_item(1); add_item(2)             # [1, 2] — shared between calls
```

Default values are evaluated once when the `def` runs and stored on the function. Fix: `items=None` then `items = [] if items is None else items`. The same applies to `{}` and to any object created in the default.

2. The shared-row grid

```
grid = [[0] * 3] * 2                 # two references to ONE row
grid[0][0] = 1                       # [[1, 0, 0], [1, 0, 0]]
grid = [[0] * 3 for _ in range(2)]   # two rows
```

`*` on a list repeats references, not copies. Fine for immutables (`[0] * 3`), wrong for a list of lists.

3. Late-binding closures

```
fs = [lambda: i for i in range(3)]
[f() for f in fs]                    # [2, 2, 2] — all see the final i
fs = [lambda i=i: i for i in range(3)]      # bind now: [0, 1, 2]
```

A closure captures the variable, not its value at creation. Fix: a default argument, `functools.partial`, or a factory function that takes `i` as a parameter (Module 16).

4. `is` for values

```
if x is 5: ...                       # works by the small-int cache; SyntaxWa
if s is "done": ...                  # depends on interning
if x is None: ...                    # correct: None is a singleton
```

`is` is identity. Use `==` for values and `is` only for `None`, `True`, `False` and sentinel objects.

5. Modifying a list while iterating over it

```
for x in xs:                         # skips elements after each removal
    if x % 2 == 0: xs.remove(x)
xs = [x for x in xs if x % 2]        # build the new list
for x in xs[:]: ...                  # or iterate a copy
```

Removing shifts the remaining elements under the iterator's index. Dicts raise `RuntimeError: dictionary changed size during iteration` for the same mistake; iterate `list(d)` or build a new dict.

6. `.sort()` returns `None`

```
xs = xs.sort()                       # xs is now None
xs.sort()                            # in place, returns None
ys = sorted(xs)                      # a new list
```

Also `list.append`, `list.reverse`, `dict.update`, `random.shuffle`: in-place methods return `None` by convention.

---

7. Integer division and modulo with negatives

```
-7 // 2                          # -4: floors towards negative infinity
-7 % 2                           # 1: the result takes the divisor's sign
int(-7 / 2)                      # -3: truncation, when that is what is me
```

Python floors; C and Java truncate. `divmod(a, b)` returns the consistent pair. For "round half up" use `(a + b // 2) // b` or `decimal`.

---

8. Float equality

```
0.1 + 0.2 == 0.3                 # False
math.isclose(0.1 + 0.2, 0.3)     # True
round(2.675, 2)                  # 2.67 — the float is slightly below 2.67
```

Compare with a tolerance, format with a fixed precision, or use `fractions.Fraction` / `decimal.Decimal` when exactness is the point (money).

---

9. The O(n²) string

```
out = ""
for row in rows: out += f"{row}\n"   # may copy the growing string each time
out = "\n".join(f"{row}" for row in rows)
```

CPython optimises some `+=` on strings in place, but not reliably; `join` is always O(n). Same for repeated `list.insert(0, x)` and `pop(0)` (use a deque).

---

10. The recursion limit

```python
def depth(node): return 1 + max(map(depth, node.children), default=0)   # Recu
sys.setrecursionlimit(1_000_000)     # allows it — and iterative BFS/DFS avoi
```

The default limit is 1000 frames; a DFS on a 10^5-node path exceeds it. Raise the limit, or convert to an explicit stack.

---

11. `and` / `or` return operands, and chained comparisons

```python
name = user_input or "anonymous"     # fine: "" → default
count = n and n * 2                  # returns 0 when n is 0 — and 0 may not b
0 < x < 10                           # chained: (0 < x) and (x < 10)
x == 1 or 2                          # always truthy: (x == 1) or 2
```

`or` returns the first truthy operand, `and` the first falsy one; `x in (1, 2)` is the fix for the last line.

---

12. The broad `except`

```python
try:
    value = int(text)
except:                              # catches KeyboardInterrupt, SystemExit,
    value = 0
except ValueError:                   # the one you mean
    value = 0
```

A bare `except` or `except Exception` around more than one statement hides the bug that would have explained the failure. Catch the specific exception around the specific statement.

The checklist, before "done"

Defaults immutable? Grid built with a comprehension? Closures bound? `==` for values? Not mutating what I iterate? `sorted` where I need the value? Division floors — intended? Floats compared with tolerance? Strings joined? Recursion depth bounded? `in` instead of `== a or b`? Exceptions specific? Twelve seconds, twelve points.

Key takeaways

Defaults and `[[…]] * n` share objects; closures capture variables late; build fresh objects and bind values explicitly.

`is` is identity — only for `None` and sentinels; `.sort()` and friends return `None`.

Never mutate the list or dict you are iterating; build a new one.

`//` and `%` floor; floats need `isclose` or fixed formatting; strings are built with `join`.

Raise the recursion limit or go iterative; `or` returns operands; catch specific exceptions around specific statements.

EXERCISE 1

Fix the idioms

The starter is a working program with four bugs from this lesson planted in `make_grid`, `record`, `make_multipliers` and `remove_evens`: a shared-row grid, a mutable default, late-binding closures and mutation while iterating. Fix all four without changing `main`. It reads `r c` and a line of integers, sets `grid[0][0] = 1` and prints `rows <sum of each row>`, calls `record` once per integer with a fresh history each time and prints `history <length of the last one>`, applies multipliers 1, 2, 3 to the first integer and prints `multipliers <results>`, and prints `odds <the odd integers>` (or `odds none`).

Input: `r c`, then the integers. **Output:** four lines.

```
2 3
2 2 3 4 4 5
```

prints

```
rows 1 0
history 1
multipliers 2 4 6
odds 3 5
```

INPUT 1

```
2 3
2 2 3 4 4 5
```

OUTPUT 1

```
rows 1 0
history 1
multipliers 2 4 6
odds 3 5
```

REFERENCE SOLUTION

```
def make_grid(rows, cols):
    return [[0] * cols for _ in range(rows)]


def record(value, history=None):
    if history is None:
        history = []
    history.append(value)
    return history


def make_multipliers(ks):
    return [lambda x, k=k: x * k for k in ks]


def remove_evens(xs):
    return [x for x in xs if x % 2]


def main():
    rows, cols = map(int, input().split())
    nums = [int(x) for x in input().split()]
    grid = make_grid(rows, cols)
    grid[0][0] = 1
    print("rows", " ".join(str(sum(row)) for row in grid))
    for value in nums:
        history = record(value)
    print("history", len(history))
    print("multipliers", " ".join(str(f(nums[0])) for f in make_multipliers([1, 2, 3
    odds = remove_evens(list(nums))
    print("odds", " ".join(map(str, odds)) if odds else "none")


main()
```

EXERCISE 2

Predict, then verify

Before running anything, write down what each expression below produces. Then write the program: read expressions one per line until EOF, evaluate each with `eval` in a namespace containing only `math`, and print `<expression> ⇒ <repr of the value>` or `<expression> raises <exception type name>`.

Input: one expression per line. **Output:** one line per expression.

```
-7 // 2
-7 % 2
0.1 + 0.2 == 0.3
[1, 2] * 2
"a" * 3
1 < 2 < 3
1 == 1 or 2
[] or "x"
0 and 1
sorted([3, 1, 2]) == [1, 2, 3]
[1, 2, 3].sort()
int("12") + 1
"5" + 5
1 / 0
round(2.5)
```

prints

```
-7 // 2 ⇒ -4
-7 % 2 ⇒ 1
0.1 + 0.2 == 0.3 ⇒ False
[1, 2] * 2 ⇒ [1, 2, 1, 2]
"a" * 3 ⇒ 'aaa'
1 < 2 < 3 ⇒ True
1 == 1 or 2 ⇒ True
[] or "x" ⇒ 'x'
0 and 1 ⇒ 0
sorted([3, 1, 2]) == [1, 2, 3] ⇒ True
[1, 2, 3].sort() ⇒ None
int("12") + 1 ⇒ 13
"5" + 5 raises TypeError
1 / 0 raises ZeroDivisionError
round(2.5) ⇒ 2
```

INPUT 1

```
-7 // 2
-7 % 2
0.1 + 0.2 == 0.3
[1, 2] * 2
"a" * 3
1 < 2 < 3
1 == 1 or 2
[] or "x"
0 and 1
sorted([3, 1, 2]) == [1, 2, 3]
[1, 2, 3].sort()
int("12") + 1
"5" + 5
1 / 0
round(2.5)
```

OUTPUT 1

```
-7 // 2 ⇒ -4
-7 % 2 ⇒ 1
0.1 + 0.2 == 0.3 ⇒ False
[1, 2] * 2 ⇒ [1, 2, 1, 2]
"a" * 3 ⇒ 'aaa'
1 < 2 < 3 ⇒ True
1 == 1 or 2 ⇒ True
[] or "x" ⇒ 'x'
0 and 1 ⇒ 0
sorted([3, 1, 2]) == [1, 2, 3] ⇒ True
[1, 2, 3].sort() ⇒ None
int("12") + 1 ⇒ 13
"5" + 5 raises TypeError
1 / 0 raises ZeroDivisionError
round(2.5) ⇒ 2
```

REFERENCE SOLUTION

```python
import math
import sys


def evaluate(expression: str) -> str:
    try:
        value = eval(expression, {"__builtins__": __builtins__, "math": math})
    except Exception as e:
        return f"{expression} raises {type(e).__name__}"
    return f"{expression} ⇒ {value!r}"


for line in sys.stdin:
    expression = line.strip()
    if expression:
        print(evaluate(expression))
```

QUIZ

1. What does `[[0] * 3] * 2` build?

A. Two independent rows

B. Two references to one row — writing one cell changes both rows

C. A 3×2 grid

D. A tuple

2. Why does `def f(x, acc=[])` keep growing across calls?

A. Lists are global

B. The default is evaluated once at `def` time and shared by every call that omits it

C. `append` is static

D. It does not

3. What is `-7 // 2` and `-7 % 2`?

A. -3 and -1

B. -4 and 1

C. -3 and 1

D. -4 and -1

4. What does `xs = xs.sort()` leave in `xs`?

A. The sorted list

B. `None` — in-place methods return `None`

C. A copy

D. An iterator

5. What does `x == 1 or 2` evaluate to when `x` is 5?

A. `False`

B. `2` — `or` returns its second operand

C. `True`

D. Raises

ANSWER KEY · PITFALLS THAT FAIL INTERVIEWS — THE TWELVE PYTHON MISTAKES INTERVIEWERS WATCH FOR

1. B `*` repeats references; use a comprehension per row.

2. B Use `None` and create the list inside.

3. B Floor division and a remainder with the divisor's sign.

4. B `sorted(xs)` returns the new list; `xs.sort()` sorts in place.

5. B Always truthy; `x in (1, 2)` is the intended test.

LESSON 4 · 16 MIN

Implement the built-in — a hash map, a dynamic array, an LRU cache and a heap by hand

"Implement a hash map" is the interview question that checks whether you know what `dict` costs and why; "implement an LRU cache" checks whether you can combine two structures to get O(1) for everything; "implement a heap" checks whether you understand the invariant behind `heapq`. None of these is used in production Python — the built-ins are faster and correct — but each is a compact demonstration of the ideas the earlier modules taught, and each has a standard shape an interviewer expects to see: the invariant stated, the operations written against it, the amortised analysis given. This lesson builds the four with those shapes.

A hash map with chaining

```python
class HashMap:
    def __init__(self, capacity=8):
        self._buckets = [[] for _ in range(capacity)]
        self._size = 0

    def _bucket(self, key):
        return self._buckets[hash(key) % len(self._buckets)]

    def get(self, key, default=None):
        for k, v in self._bucket(key):
            if k == key:
                return v
        return default

    def put(self, key, value):
        bucket = self._bucket(key)
        for i, (k, _) in enumerate(bucket):
            if k == key:
                bucket[i] = (key, value)
                return
        bucket.append((key, value))
        self._size += 1
        if self._size > 0.75 * len(self._buckets):
            self._resize(2 * len(self._buckets))

    def delete(self, key):
        bucket = self._bucket(key)
        for i, (k, _) in enumerate(bucket):
            if k == key:
                del bucket[i]
                self._size -= 1
                return True
        return False

    def _resize(self, capacity):
        pairs = [pair for bucket in self._buckets for pair in bucket]
        self._buckets = [[] for _ in range(capacity)]
        for key, value in pairs:
            self._bucket(key).append((key, value))

    def __len__(self):
        return self._size
```

The points to say aloud: the *invariant* is that a key lives in the bucket its hash selects; `get` / `put` / `delete` are O(1) *average* because buckets stay short while the *load factor* (size / buckets) is bounded — hence the resize at 0.75, which rehashes every key (O(n)) but happens so rarely that the cost is amortised O(1) per insertion; a bad hash (every key to one bucket) degrades to O(n). CPython's dict uses open addressing with a compact entry array instead of chains, keeps insertion order, and resizes at 2/3 — the follow-up questions. Keys must be hashable, and equal keys must hash equal, which is why `__eq__` and `__hash__` go together (Module 8).

A dynamic array

```python
class DynamicArray:
    def __init__(self):
        self._capacity = 4
        self._store = [None] * self._capacity
        self._size = 0

    def append(self, value):
        if self._size == self._capacity:
            self._grow()
        self._store[self._size] = value
        self._size += 1

    def _grow(self):
        self._capacity *= 2
        new = [None] * self._capacity
        for i in range(self._size):
            new[i] = self._store[i]
        self._store = new

    def __getitem__(self, i):
        if not 0 <= i < self._size:
            raise IndexError(i)
        return self._store[i]

    def pop(self):
        if not self._size:
            raise IndexError("pop from empty")
        self._size -= 1
        value, self._store[self._size] = self._store[self._size], None
        return value

    def __len__(self):
        return self._size
```

Doubling the capacity makes `append` amortised O(1): n appends copy at most 1 + 2 + 4 + … + n < 2n elements in total. Growing by a constant instead of a factor would make it O(n²). CPython's list over-allocates by about 12.5 % (a smaller factor, so more frequent but cheaper copies), which is the follow-up.

An LRU cache

```python
from collections import OrderedDict

class LRUCache:
    def __init__(self, capacity):
        self.capacity = capacity
        self._items = OrderedDict()

    def get(self, key):
        if key not in self._items:
            return None
        self._items.move_to_end(key)          # most recently used goes last
        return self._items[key]

    def put(self, key, value):
        if key in self._items:
            self._items.move_to_end(key)
        self._items[key] = value
        if len(self._items) > self.capacity:
            self._items.popitem(last=False)   # evict the least recently used
```

The requirement is O(1) `get` and `put` including eviction. A dict gives O(1) lookup but no order; a list gives order but O(n) moves; the combination is a dict whose values are nodes of a doubly linked list, so a node can be unlinked and re-linked at the tail in O(1) — which is exactly what `OrderedDict.move_to_end` and `popitem(last=False)` do. In a round, write the `OrderedDict` version first and say how the linked list would replace it; if asked for the linked list, a dummy head and tail node remove every edge case from unlink/append. `functools.lru_cache` is the decorator form.

A binary heap

```python
class MinHeap:
    def __init__(self):
        self._a = []

    def push(self, x):
        a = self._a
        a.append(x)
        i = len(a) - 1
        while i and a[(i - 1) // 2] > a[i]:              # sift up
            a[(i - 1) // 2], a[i] = a[i], a[(i - 1) // 2]
            i = (i - 1) // 2

    def pop(self):
        a = self._a
        top, last = a[0], a.pop()
        if a:
            a[0] = last
            i, n = 0, len(a)
            while True:                                  # sift down
                left, right, smallest = 2 * i + 1, 2 * i + 2, i
                if left < n and a[left] < a[smallest]: smallest = left
                if right < n and a[right] < a[smallest]: smallest = right
                if smallest == i: break
                a[i], a[smallest] = a[smallest], a[i]
                i = smallest
        return top

    def peek(self):
        return self._a[0]

    def __len__(self):
        return len(self._a)
```

The invariant: every parent is at most its children, stored in an array where the children of `i` are `2i + 1` and `2i + 2`. Push appends and sifts up; pop swaps the last element to the root and sifts down; both are O(log n) because the tree's height is log n. `heapify` builds one in O(n) by sifting down from the last parent. A max-heap flips the comparison or negates the keys.

What else gets asked

`enumerate` and `zip` as generators (`yield` with a counter; `zip` stops at the shortest); a `Counter` as a `defaultdict(int)`; `defaultdict` as a dict subclass with `__missing__` (Module 16); `functools.cache` as a dict in a closure; a `deque` as a ring buffer with head and tail indices; `itertools.groupby` as a generator over runs; a trie as nested dicts. For each, state the invariant, write the operations, give the complexity — the same shape.

Pitfalls

- A hash map with no resize (O(n) lookups as it fills) or one that resizes by a constant.
- Forgetting that `put` on an existing key must replace, not append a duplicate.
- An LRU with a list — O(n) moves — and calling it O(1).
- A heap that sifts only one direction or forgets the empty case in `pop`.
- Using `hash()` of strings to derive *output* — it is randomised per process (Module 7); bucket counts and lookups are fine, printed bucket contents are not.
- Skipping the complexity statement.

Key takeaways

- A hash map is buckets selected by `hash(key) % n` with a bounded load factor and a doubling resize: O(1) average, amortised.
- A dynamic array doubles its capacity for amortised O(1) `append`.
- An LRU cache is a dict plus a doubly linked list — `OrderedDict` with `move_to_end` and `popitem(last=False)` — for O(1) everything.
- A binary heap is an array with the parent-child invariant; push sifts up, pop sifts down, O(log n).
- The interview shape: invariant, operations, complexity, then the built-in that does it better.

EXERCISE 1

A hash map with chaining

Implement `HashMap` with separate chaining: buckets are lists of `(key, value)` pairs selected by `hash(key) % capacity`, the initial capacity is 4, and after a `put` that adds a new key the table doubles when `size > 0.75 * capacity`, rehashing every pair. Read commands until EOF: `put <key> <value>` (replace on an existing key), `get <key>` (print the value or `missing`), `del <key>`

(print `deleted` or `absent`), `len` (print the size) and `capacity` (print the bucket count). Keys and values are words.

Input: one command per line. **Output:** one line per printing command.

```
put a 1
put b 2
put a 3
get a
len
put c 4
put d 5
capacity
del b
get b
len
```

prints

```
3
2
8
deleted
missing
3
```

INPUT 1

```
put a 1
put b 2
put a 3
get a
len
put c 4
put d 5
capacity
del b
get b
len
```

OUTPUT 1

```
3
2
8
deleted
missing
3
```

REFERENCE SOLUTION

```python
import sys


class HashMap:
    def __init__(self, capacity=4):
        self._buckets = [[] for _ in range(capacity)]
        self._size = 0

    def _bucket(self, key):
        return self._buckets[hash(key) % len(self._buckets)]

    def get(self, key, default=None):
        for k, v in self._bucket(key):
            if k == key:
                return v
        return default

    def put(self, key, value):
        bucket = self._bucket(key)
        for i, (k, _) in enumerate(bucket):
            if k == key:
                bucket[i] = (key, value)
                return
        bucket.append((key, value))
        self._size += 1
        if self._size > 0.75 * len(self._buckets):
            self._resize(2 * len(self._buckets))

    def delete(self, key):
        bucket = self._bucket(key)
        for i, (k, _) in enumerate(bucket):
            if k == key:
                del bucket[i]
                self._size -= 1
                return True
        return False

    def _resize(self, capacity):
        pairs = [pair for bucket in self._buckets for pair in bucket]
        self._buckets = [[] for _ in range(capacity)]
        for key, value in pairs:
            self._bucket(key).append((key, value))

    @property
    def capacity(self):
        return len(self._buckets)

    def __len__(self):
        return self._size
```

```
table = HashMap()
for line in sys.stdin:
    parts = line.split()
    if not parts:
        continue
    command = parts[0]
    if command == "put":
        table.put(parts[1], parts[2])
    elif command == "get":
        print(table.get(parts[1], "missing"))
    elif command == "del":
        print("deleted" if table.delete(parts[1]) else "absent")
    elif command == "len":
        print(len(table))
    elif command == "capacity":
        print(table.capacity)
```

EXERCISE 2

An LRU cache

Implement `LRUCache(capacity)` with O(1) `get` and `put` using an `OrderedDict`: `get` returns the value and marks the key most recently used, or returns `-1`; `put` inserts or updates and marks it most recently used, evicting the least recently used key when the size exceeds the capacity. Read the capacity on the first line, then commands `get <key>` (print the result) and `put <key> <value>` until EOF; finally print `keys <keys from least to most recently used>`.

Input: the capacity, then commands. **Output:** one line per `get`, then the keys.

```
2
put 1 1
put 2 2
get 1
put 3 3
get 2
put 4 4
get 1
get 3
get 4
```

prints

```
1
-1
-1
3
4
keys 3 4
```

INPUT 1

```
2
put 1 1
put 2 2
get 1
put 3 3
get 2
put 4 4
get 1
get 3
get 4
```

OUTPUT 1

```
1
-1
-1
3
4
keys 3 4
```

REFERENCE SOLUTION

```
import sys
from collections import OrderedDict


class LRUCache:
    def __init__(self, capacity):
        self.capacity = capacity
        self._items = OrderedDict()

    def get(self, key):
        if key not in self._items:
            return -1
        self._items.move_to_end(key)
        return self._items[key]

    def put(self, key, value):
        if key in self._items:
            self._items.move_to_end(key)
        self._items[key] = value
        if len(self._items) > self.capacity:
            self._items.popitem(last=False)

    def keys(self):
        return list(self._items)


capacity = int(sys.stdin.readline())
cache = LRUCache(capacity)
for line in sys.stdin:
    parts = line.split()
    if not parts:
        continue
    if parts[0] == "get":
        print(cache.get(parts[1]))
    elif parts[0] == "put":
        cache.put(parts[1], parts[2])
print("keys", " ".join(cache.keys()))
```

QUIZ

1. Why does a hash map resize when its load factor passes a threshold?

A. To save memory

B. To keep buckets short so lookups stay O(1) average; the O(n) rehash is amortised over many inserts

C. To sort the keys

D. It need not

2. Why does a dynamic array double its capacity rather than add a constant?

A. Simplicity

B. Doubling makes `append` amortised O(1); a constant step makes n appends $O(n^2)$

C. Memory alignment

D. It does not matter

3. Which pair of structures gives an LRU cache O(1) for get, put and eviction?

A. A list and a set

B. A dict plus a doubly linked list — what `OrderedDict` provides

C. Two lists

D. A heap and a dict

4. What is the heap invariant in an array-backed binary min-heap?

A. The array is sorted

B. Every parent at index i is ≤ its children at 2i+1 and 2i+2

C. The root is the largest

D. Children are adjacent

5. Why must a program never print the bucket contents of a hash map keyed by strings?

A. Buckets are private

B. String hashing is randomised per process, so the bucket layout differs run to run

C. Lists cannot be printed

D. It is slow

ANSWER KEY · IMPLEMENT THE BUILT-IN — A HASH MAP, A DYNAMIC ARRAY, AN LRU CACHE AND A HEAP BY HAND

1. B Without resizing, chains grow and every operation degrades towards O(n).

2. B The total copying across n appends is under 2n elements.

3. B The dict finds the node; the list moves it to the tail and pops the head in O(1).

4. B Push sifts up, pop sifts down; both O(log n).

5. B Lookups, sizes and capacities are deterministic; the layout is not.

LESSON 5 · 13 MIN

Writing clean solutions — structure, names, edges first and the code you can read aloud

Two solutions to the same problem can both pass and score very differently, because the interviewer is reading the code as a sample of how you write for colleagues. Clean does not mean long: it means a `main` that reads and prints, a solve function whose name is the idea, helpers that are pure and small, names that say what they hold, edge cases handled first, and nothing clever that needs a comment to decode. This lesson gives the structure, the naming rules, the edge-first habit, how type hints and dataclasses serve as documentation, how to test inside the file, and a before-and-after that shows the difference.

The structure

```python
import sys
from dataclasses import dataclass

@dataclass(frozen=True)
class Player:
    name: str
    score: int

def parse(lines: list[str]) -> list[Player]:
    return [Player(name, int(score)) for name, score in (line.split() for line

def rank(players: list[Player]) -> list[tuple[int, Player]]:
    """Dense ranks: equal scores share a rank; the next distinct score takes t
    ordered = sorted(players, key=lambda p: (-p.score, p.name))
    ranks, current, previous = [], 0, None
    for player in ordered:
        if player.score != previous:
            current += 1
            previous = player.score
        ranks.append((current, player))
    return ranks

def main() -> None:
    players = parse(sys.stdin.read().splitlines())
    for position, player in rank(players):
        print(position, player.name, player.score)

if __name__ == "__main__":
    main()
```

Reading and printing live in `main`; `parse` turns text into typed values; `rank` is the idea and touches no I/O, so it can be called in a test with a literal list. The dataclass gives the record a name and its fields names — `p.score` instead of `p[1]`. A one-line docstring states the rule that is not obvious from the code (dense versus competition ranking).

Names

Variables hold what their names say: `players`, not `data`; `count_by_word`, not `d`; `start`, `end`, not `i`, `j` — except in the tightest loops where `i` is the index and everyone knows it. Functions are verbs or questions: `parse`, `rank`, `is_balanced`, `next_greater`. Booleans read as conditions: `is_empty`, `has_cycle`, `seen`. Constants are upper case at module level:

`PAIRS = {")": "(", "]": "[", "}": "{"}`. A name that needs a comment is the wrong name.

Edges first

```python
def is_balanced(s: str) -> bool:
    if not s:
        return True
    ...

def median(xs: list[float]) -> float:
    if not xs:
        raise ValueError("median of empty list")
    ...
```

Handle the empty input, the single element and the invalid input at the top with an early return or a raise, so the main body reads without guards. State which you chose — "empty is balanced; median of nothing raises" — because that is the question you asked in step one of the round.

Pure helpers, small functions

A function that takes values and returns values, with no prints and no globals, is testable by calling it and reusable inside the solve. Each helper does one thing and fits on a screen; the solve function reads as a sequence of named steps. `global` is a smell; a mutable module-level container that helpers modify is the same smell with a different spelling. Pass state in, return results out.

Type hints and dataclasses as documentation

`def rank(players: list[Player]) -> list[tuple[int, Player]]` tells the reader the shapes without a comment; `@dataclass(frozen=True)` says the record is a value; an `Enum` names the states. None of it is checked at run time, and all of it is read at review time. Hints on the public functions, not on every local.

Avoiding cleverness

```
# clever
result = [x for xs in (a if c else b for a, b, c in rows) for x in xs if x]
# clear
result = []
for a, b, c in rows:
    chosen = a if c else b
    result.extend(x for x in chosen if x)
```

A nested comprehension with a conditional expression inside it is shorter and slower to read; the loop names the intermediate. Prefer the version a colleague understands on first reading — the interviewer is that colleague. Likewise: no one-letter lambdas that span a line, no `reduce` where a loop is clear, no walrus in a comprehension that already has a condition.

Tests in the file

```
def _test() → None:
    assert is_balanced("") is True
    assert is_balanced("([])") is True
    assert is_balanced("(]") is False
    assert rank([Player("a", 5), Player("b", 5), Player("c", 1)])[2][0] == 2

if __name__ == "__main__":
    if len(sys.argv) > 1 and sys.argv[1] == "--test":
        _test()
    else:
        main()
```

Three `assert` lines with the edge cases, runnable with a flag, show the interviewer you test without being asked. In a judged submission they stay dormant behind the flag; in a real repository they become a `test_*.py` file (Module 18).

Before and after

```python
# before
d = {}
for l in sys.stdin:
    w = l.split()
    for x in w:
        if x in d: d[x] += 1
        else: d[x] = 1
for k in sorted(d, key=lambda k: (-d[k], k))[:3]: print(k, d[k])

# after
def top_words(text: str, k: int = 3) -> list[tuple[str, int]]:
    counts = Counter(text.split())
    return sorted(counts.items(), key=lambda item: (-item[1], item[0]))[:k]

def main() -> None:
    for word, count in top_words(sys.stdin.read()):
        print(word, count)
```

Same output; the second names the idea, uses the built-in, separates I/O, and can be tested with a string.

Pitfalls

All the logic in module scope, with prints inside the computation.

`d`, `l`, `x` for things that have names.

Edge cases handled by a guard deep inside the loop, or not at all.

A comprehension that needs a comment.

Globals mutated by helpers.

No test, no docstring on the one non-obvious rule.

Key takeaways

`main` reads and prints; `parse` types the input; `solve` is the idea; helpers are pure and small.

Names say what they hold; functions are verbs; booleans are conditions; constants are upper case.

Edges first with early returns or raises, and say which you chose.

Hints and dataclasses document shapes; avoid cleverness that costs a reader a second look.

Three asserts behind a flag show testing without being asked.

EXERCISE 1

Balanced brackets, cleanly

Write `check(s)` as a pure function returning `"balanced"`, `"unbalanced at <i>"` for the index of the first closing bracket that has no matching opener, or `"unclosed <n>"` when the scan ends with `n` openers still open. Use a stack and a `PAIRS` constant mapping each closer to its opener; keep I/O in `main`. Read lines until EOF and print the verdict for each.

Input: one bracket string per line. **Output:** one verdict per line.

```
([]{})
([)]
((
)
```

prints

```
balanced
unbalanced at 2
unclosed 2
unbalanced at 0
```

INPUT 1

```
([]{})
([)]
((
)
```

OUTPUT 1

```
balanced
unbalanced at 2
unclosed 2
unbalanced at 0
```

REFERENCE SOLUTION

```python
import sys

PAIRS = {")": "(", "]": "[", "}": "{"}


def check(s: str) -> str:
    stack = []
    for i, ch in enumerate(s):
        if ch in PAIRS.values():
            stack.append(ch)
        elif ch in PAIRS:
            if not stack or stack.pop() != PAIRS[ch]:
                return f"unbalanced at {i}"
    return "balanced" if not stack else f"unclosed {len(stack)}"


def main() -> None:
    for line in sys.stdin:
        text = line.strip()
        if text:
            print(check(text))


if __name__ == "__main__":
    main()
```

EXERCISE 2

A leaderboard

Read lines `<name> <score>` until EOF into a frozen dataclass `Player`. Write `rank(players)` returning `(rank, player)` pairs ordered by score descending then name ascending, with dense ranks — equal scores share a rank and the next distinct score takes the next integer. Print `<rank> <name> <score>` per line. Keep parsing, ranking and printing in separate functions.

Input: one player per line. **Output:** one line per player.

```
ann 50
bob 70
cy 50
dee 90
```

prints

```
1 dee 90
2 bob 70
3 ann 50
3 cy 50
```

INPUT 1

```
ann 50
bob 70
cy 50
dee 90
```

OUTPUT 1

```
1 dee 90
2 bob 70
3 ann 50
3 cy 50
```

REFERENCE SOLUTION

```python
import sys
from dataclasses import dataclass


@dataclass(frozen=True)
class Player:
    name: str
    score: int


def parse(lines: list[str]) -> list[Player]:
    return [Player(name, int(score)) for name, score in (line.split() for line in li


def rank(players: list[Player]) -> list[tuple[int, Player]]:
    """Dense ranks: equal scores share a rank; the next distinct score takes the nex
    ordered = sorted(players, key=lambda p: (-p.score, p.name))
    ranks, current, previous = [], 0, None
    for player in ordered:
        if player.score != previous:
            current += 1
            previous = player.score
        ranks.append((current, player))
    return ranks


def main() -> None:
    players = parse([line for line in sys.stdin.read().splitlines() if line.strip()]
    for position, player in rank(players):
        print(position, player.name, player.score)


if __name__ == "__main__":
    main()
```

QUIZ

1. Where should reading and printing live in a clean solution?

A. Inside the solve function

B. In `main`, with the solve function taking and returning values

C. At module level

D. Anywhere

2. Which name is right for a dict from word to count?

A. `d`

B. `count_by_word`

C. `data`

D. `tmp`

3. How should the empty-input case be handled?

A. Inside the main loop with a guard

B. First, at the top of the function, with an early return or a raise — and say which you chose

C. Ignored

D. With a try/except

4. What do type hints and a dataclass provide in a solution?

A. Run-time checks

B. Documentation of shapes that a reader gets without comments

C. Speed

D. Nothing

5. Which version should you write when a nested comprehension and a short loop do the same thing?

A. Always the comprehension

B. The one a colleague understands on first reading — usually the loop that names the intermediate

C. Always the loop

D. Whichever is shorter

ANSWER KEY · WRITING CLEAN SOLUTIONS — STRUCTURE, NAMES, EDGES FIRST AND THE CODE YOU CAN READ ALOUD

1. B A pure solve can be tested by calling it with a literal.

2. B Names say what they hold; a name that needs a comment is the wrong name.

3. B Edges first keeps the body free of guards and answers the clarifying question you asked.

4. B `p.score` instead of `p[1]`; the signature says what goes in and out.

5. B The interviewer is that colleague.

LESSON 6 · 25 MIN

The Python theory drill — thirty questions, thirty answers

Every Python interview has a theory section, and its questions have barely changed in a decade: what the GIL is, how a dict works, what a generator does, why a mutable default is wrong, the difference between `is` and `==`. This lesson is the drill: thirty questions, each with the answer an interviewer wants to hear — the definition first, the one consequence that matters, then a stop so the follow-up can come — and a pointer to the module that teaches it in depth. Read it twice a week before interviews. A Python answer is expected to name a mechanism: not "lists are slow to search" but "`in` on a list is a linear scan; a set hashes".

The language and its objects

What does "everything is an object" mean in practice? Every value — ints, functions, classes, modules — is a heap object with a type and a reference count; a variable is a name bound to one. Consequences: assignment copies nothing, functions are passed as values, and `type(x)` is itself an object. *(Modules 1, 19)*

Mutable versus immutable? An immutable object's value cannot change after creation (int, float, str, tuple, frozenset, bytes); a mutable one can (list, dict, set, bytearray, most instances). Immutables are hashable by default and safe to share; a "change" to one is a new object. *(2, 6)*

`is` versus `==`? `is` compares identity (the same object); `==` compares value through `__eq__`. Use `is` only for `None`, `True`, `False` and sentinels; `x is 5` works by the small-int cache and is a bug. *(2, 19)*

How are arguments passed? By assignment: the parameter is bound to the same object the caller passed ("pass by object reference"). Mutating the object is visible to the caller; rebinding the parameter is not. *(4)*

Why is a mutable default argument a bug? Defaults are evaluated once at `def` time and stored on the function, so every call that omits the argument shares the same list. Fix with `None` and create inside. *(4, 20)*

What does a closure capture? The variable, by cell, not its value at creation — late binding. Three lambdas made in a loop all see the loop's final value; bind with a default argument or a factory. *(16)*

`//` and `%` with negatives? Floor division rounds towards negative infinity and the remainder takes the divisor's sign: `-7 // 2 == -4`, `-7 % 2 == 1`. C and Java truncate instead. *(2)*

Why does `0.1 + 0.2 != 0.3`? Binary floating point cannot represent those decimals exactly; compare with `math.isclose`, format to a fixed precision, or use `decimal` / `fractions` for exact arithmetic. *(2)*

Data structures

How does a dict work? A hash table: `hash(key)` selects a slot in an open-addressed array; equal keys must hash equal; lookups, inserts and deletes are O(1) on average and O(n) in a pathological worst case. Since 3.7 insertion order is guaranteed; the table resizes as it fills. *(7, 20)*

What makes an object hashable, and why must lists not be keys? It defines `__hash__` consistent with `__eq__` and its hash never changes; a list's contents can change, so its hash would go stale and the key would be lost. Tuples of hashables and frozensets are the substitutes. *(7, 8)*

List versus tuple? Both are ordered sequences; a list is mutable and over-allocates for appends, a tuple is immutable, slightly smaller, hashable when its elements are, and signals "a fixed record". *(6)*

The cost of the common operations? List: index O(1), append amortised O(1), `in` and `insert(0)` O(n); dict and set: O(1) average; sort O(n log n); `deque` O(1) at both ends; `heapq` O(log n) push/pop. A linear operation inside a loop over the same data is the quadratic to look for. *(19)*

When do you use a set, a deque, a heap, `bisect`? Membership and de-duplication; a queue or sliding window; repeated minimum or top-k; queries on a sorted list or search on a monotonic answer. *(7, 13, 20)*

Shallow versus deep copy? A shallow copy (`list(xs)`, `xs[:]`, `copy.copy`) is a new container with the same element objects; a deep copy (`copy.deepcopy`) recursively copies the elements. `[[0] * n] * m` is m references to one row. *(6, 19)*

Functions, iteration and classes

What is a generator, and why use one? A function with `yield` returns an iterator that runs lazily, one value per `next`, keeping its frame between values; it processes streams of any

length in constant memory and composes into pipelines. A generator expression is the inline form. *(11)*

Iterable versus iterator? An iterable has `__iter__` returning an iterator; an iterator has `__next__` and raises `StopIteration` when done (and is its own iterable). A list can be iterated many times; an iterator is consumed once. *(11)*

What does a decorator do? `@deco` above `def f` rebinds `f = deco(f)`: a function that takes a function and returns a replacement, usually a wrapper with `@functools.wraps`. Used for logging, caching, registration, access control. *(16)*

`*args` and `**kwargs`? Collect extra positional arguments into a tuple and extra keyword arguments into a dict; in a call, `*` and `**` unpack them. A wrapper forwards both to be transparent. *(4)*

What is a context manager? An object with `__enter__` and `__exit__` used by `with`; `__exit__` runs on every exit including exceptions, so resources are released deterministically. `contextlib.contextmanager` writes one from a generator. *(10)*

How does attribute lookup and inheritance work? `obj.x` checks data descriptors on the type, then the instance dict, then the type and its bases in MRO order (C3 linearisation), then `__getattr__`. `super()` follows the MRO, which is why cooperative `__init__` works with multiple inheritance. *(8, 9, 16)*

`@staticmethod` versus `@classmethod`? A static method receives nothing implicit — a function in the class's namespace; a class method receives the class as `cls` and is the tool for alternative constructors that work in subclasses. *(8)*

What is duck typing, and what are protocols and ABCs for? Behaviour is decided by what an object can do, not its type; `typing.Protocol` names a set of methods for static checking, and an ABC (`collections.abc`) enforces them at run time and can supply mixin methods. *(9, 15)*

What does `__slots__` do? Replaces the per-instance `__dict__` with fixed attribute slots: less memory, faster access, no ad-hoc attributes. *(19)*

Dataclass versus named tuple versus plain class? A dataclass generates `__init__`, `__repr__`, `__eq__` (and ordering, hashing, frozenness on request) for a record with named fields; a named tuple is an immutable tuple with names; a plain class is for behaviour beyond a record. *(8)*

Errors, modules and the run time

`try/except/else/finally`? `except` catches; `else` runs only when nothing was raised; `finally` always runs. Catch specific exceptions around specific statements; `raise ... from e` chains a cause; exceptions are the control flow for failure, not return codes. *(10)*

What is the GIL? A mutex that lets one thread execute bytecode at a time in CPython, released during blocking I/O and by some C extensions. Threads help I/O-bound code, not CPU-bound code; processes give parallelism; asyncio gives single-threaded concurrency for many waits. *(17)*

What is a race condition, and how is it fixed? Two threads performing a read-modify-write on shared state between bytecodes — `count += 1` is three instructions — so an update is lost. A `Lock` makes the compound operation atomic; a `Queue` avoids the sharing. *(17)*

How does memory management work? Reference counting frees an object the moment its last reference goes; a generational cycle collector finds reference cycles the counts cannot free; `del` removes a name, not the object. *(19)*

What happens on `import`? The module is found on `sys.path`, executed once top to bottom, and cached in `sys.modules`; later imports return the cached object. `if __name__ == "__main__":` distinguishes running from importing. Circular imports fail when a name is used before the other module finished executing. *(12)*

How do you make Python code faster? In order: fix the algorithm, choose the data structure, move the loop into a built-in or a comprehension, batch I/O, then hoist lookups in the profiled hot loop; NumPy for numeric arrays, processes for CPU-bound parallelism. Measure with `cProfile` and `timeit` before and after. *(19)*

How to use the drill

Read a question, answer aloud in two sentences, then read the answer and note what you left out. The interviewer's follow-up is always "why" or "what happens if" — the mechanism in each answer is what survives that. When an answer references a module number, that module has the code that proves it.

Key takeaways

Objects, references and mutability explain most language questions; name the mechanism.

Dicts hash, lists scan, deques and heaps and bisect each answer one shape of question.

Generators are lazy frames; decorators rebind; context managers guarantee `__exit__`; the MRO orders lookup.

Exceptions are for failure, the GIL serialises bytecode, reference counting frees immediately and the collector handles cycles.

Speed comes from the algorithm and the data structure first, measured before and after.

EXERCISE 1

The MRO by hand

Define `A`, `B(A)`, `C(A)` and `D(B, C)`, each with a method `who()` that returns its own letter followed by `>` and `super().who()` — except `A`, which returns `"A"`. Read a class name and print `mro <the class names in __mro__>` and `chain <the result of who() on an instance>`.

Input: one class name. **Output:** two lines.

```
D
```

prints

```
mro D B C A object
chain D>B>C>A
```

INPUT 1

```
D
```

OUTPUT 1

```
mro D B C A object
chain D>B>C>A
```

REFERENCE SOLUTION

```
class A:
    def who(self):
        return "A"


class B(A):
    def who(self):
        return "B>" + super().who()


class C(A):
    def who(self):
        return "C>" + super().who()


class D(B, C):
    def who(self):
        return "D>" + super().who()


CLASSES = {"A": A, "B": B, "C": C, "D": D}
name = input().strip()
cls = CLASSES[name]
print("mro", " ".join(k.__name__ for k in cls.__mro__))
print("chain", cls().who())
```

EXERCISE 2

Laziness observed

Read `k` on the first line and integers on the second. Build a generator pipeline over a shared `log` list: `read_all` yields each integer after appending `read <x>`; `only_even` appends `keep <x>` and yields evens, or appends `drop <x>`; `squared` appends `square <x>` and yields `x * x`. Take the first `k` results with `itertools.islice`. Print each log line, then `result <values or none>` and `consumed <number of read lines> of <count of integers>` — showing that only the integers needed were pulled.

Input: `k`, then the integers. **Output:** the log, then two lines.

```
2
1 2 3 4 5 6
```

prints

```
read 1
drop 1
read 2
keep 2
square 2
read 3
drop 3
read 4
keep 4
square 4
result 4 16
consumed 4 of 6
```

INPUT 1

```
2
1 2 3 4 5 6
```

OUTPUT 1

```
read 1
drop 1
read 2
keep 2
square 2
read 3
drop 3
read 4
keep 4
square 4
result 4 16
consumed 4 of 6
```

REFERENCE SOLUTION

```
from itertools import islice

k = int(input())
nums = [int(x) for x in input().split()]
log = []


def read_all(values):
    for x in values:
        log.append(f"read {x}")
        yield x


def only_even(it):
    for x in it:
        if x % 2 == 0:
            log.append(f"keep {x}")
            yield x
        else:
            log.append(f"drop {x}")


def squared(it):
    for x in it:
        log.append(f"square {x}")
        yield x * x


result = list(islice(squared(only_even(read_all(nums))), k))
for line in log:
    print(line)
print("result", " ".join(map(str, result)) if result else "none")
print("consumed", sum(line.startswith("read ") for line in log), "of", len(nums))
```

QUIZ

1. How are arguments passed in Python?

A. By value

B. By assignment: the parameter is bound to the caller's object — mutation is visible, rebinding is not

C. By reference

D. By copy for lists

2. What is a generator?

A. A list built lazily

B. A function with `yield` whose call returns an iterator that keeps its frame between values

C. A thread

D. A decorator

3. What does `super()` follow?

A. The class's first base

B. The instance's MRO, so cooperative methods run once each in a diamond

C. The parent's parent

D. Nothing in multiple inheritance

4. What is the GIL, in one sentence?

A. A garbage collector

B. A mutex letting one thread execute bytecode at a time, released during I/O — threads for waiting, processes for CPU

C. A thread pool

D. A lock on dicts

5. What must be true of a dict key?

A. It is a string

B. It is hashable: `__hash__` consistent with `__eq__` and never changing — so lists cannot be keys

C. It is immutable

D. It is small

ANSWER KEY · THE PYTHON THEORY DRILL — THIRTY QUESTIONS, THIRTY ANSWERS

1. B "Pass by object reference" is the phrase interviewers accept.

2. B Constant memory for streams of any length; consumed once.

3. B In `D(B, C)`, `super()` inside `B.who` is `C`.

4. B Name the mechanism and its consequence.

5. B Tuples of hashables and frozensets are the substitutes for lists and sets.

CHECKPOINT · 30 MIN

Final checkpoint — Interview idioms

The last checkpoint of the Python plan. It covers the round and the file template, the idiom sheet, the twelve pitfalls, the hash map, dynamic array, LRU cache and heap by hand, clean-solution habits and the theory drill — and, being the last one, it reaches back across the whole track: collections, generators, classes, exceptions, the standard library and the cost model.

How it works. Fifteen questions and three programs; 70% on the questions and every program accepted clears the module. With every other module complete, that also completes the plan.

Before you start, make sure you can answer these from memory:

- Why `sys.stdin.buffer.read().split()` beats `input()` in a loop, and when it does not matter.
- Which shape "longest subarray with at most k distinct" is, and the idiom for it.
- Why `[[0] * n] * m` is wrong, and why a mutable default argument is shared.
- What makes `get`, `put` and eviction all O(1) in an LRU cache.
- Why a hash map resizes at a load factor, and what that does to the cost of `put`.
- What `-7 // 2` and `-7 % 2` are, and why.
- The one-sentence answers for the GIL, a generator and `is` versus `==`.

The three programs are the classics interviewers set: a sliding-window rate limiter over a stream of timestamps with a deque, an RPN evaluator with a stack, an operator table and per-line error handling, and an interval merger with sorted keys, a coverage total and `bisect` queries.

EXERCISE 1

A sliding-window rate limiter

Read `limit window` on the first line, then non-decreasing integer timestamps one per line until EOF. A request at time `t` is allowed when fewer than `limit` requests were allowed in the interval `(t - window, t]`; keep the allowed timestamps in a `deque`, dropping those at or before `t - window` from the front. Print `t=<t> allow` or `t=<t> deny` per request, then `allowed <a> denied <d>`.

Input: the limit and window, then timestamps. **Output:** one line per request, then the totals.

```
2 10
1
2
3
11
12
13
```

prints

```
t=1 allow
t=2 allow
t=3 deny
t=11 allow
t=12 allow
t=13 deny
allowed 4 denied 2
```

INPUT 1

```
2 10
1
2
3
11
12
13
```

OUTPUT 1

```
t=1 allow
t=2 allow
t=3 deny
t=11 allow
t=12 allow
t=13 deny
allowed 4 denied 2
```

REFERENCE SOLUTION

```python
import sys
from collections import deque


class RateLimiter:
    def __init__(self, limit, window):
        self.limit = limit
        self.window = window
        self._allowed = deque()

    def allow(self, t):
        while self._allowed and self._allowed[0] <= t - self.window:
            self._allowed.popleft()
        if len(self._allowed) < self.limit:
            self._allowed.append(t)
            return True
        return False


def main():
    limit, window = map(int, sys.stdin.readline().split())
    limiter = RateLimiter(limit, window)
    allowed = denied = 0
    for line in sys.stdin:
        if not line.strip():
            continue
        t = int(line)
        if limiter.allow(t):
            allowed += 1
            print(f"t={t} allow")
        else:
            denied += 1
            print(f"t={t} deny")
    print(f"allowed {allowed} denied {denied}")


main()
```

EXERCISE 2

An RPN evaluator

Read one reverse-Polish expression per line until EOF: integer tokens and the operators `+`, `-`, `*`, `/` (division truncates towards zero, as `int(a / b)`). Evaluate each with a stack and a dict from operator to function, and print `= <value>`, or `error: stack underflow` when an operator lacks two operands, `error: division by zero`, `error: bad token <token>` for anything else, or `error:`

`leftover operands` when more than one value remains. Handle each error with a specific exception, not a bare `except`.

Input: one expression per line. **Output:** one line per expression.

```
2 1 + 3 *
4 13 5 / +
10 6 9 3 + -11 * / * 17 + 5 +
1 +
1 0 /
1 2
1 x +
```

prints

```
= 9
= 6
= 22
error: stack underflow
error: division by zero
error: leftover operands
error: bad token x
```

INPUT 1

```
2 1 + 3 *
4 13 5 / +
10 6 9 3 + -11 * / * 17 + 5 +
1 +
1 0 /
1 2
1 x +
```

OUTPUT 1

```
= 9
= 6
= 22
error: stack underflow
error: division by zero
error: leftover operands
error: bad token x
```

REFERENCE SOLUTION

```python
import sys

OPS = {
    "+": lambda a, b: a + b,
    "-": lambda a, b: a - b,
    "*": lambda a, b: a * b,
    "/": lambda a, b: int(a / b),
}


class RpnError(Exception):
    pass


def evaluate(tokens: list[str]) -> int:
    stack = []
    for token in tokens:
        if token in OPS:
            if len(stack) < 2:
                raise RpnError("stack underflow")
            b, a = stack.pop(), stack.pop()
            try:
                stack.append(OPS[token](a, b))
            except ZeroDivisionError:
                raise RpnError("division by zero") from None
        else:
            try:
                stack.append(int(token))
            except ValueError:
                raise RpnError(f"bad token {token}") from None
    if len(stack) != 1:
        raise RpnError("leftover operands" if stack else "stack underflow")
    return stack[0]


for line in sys.stdin:
    tokens = line.split()
    if not tokens:
        continue
    try:
        print("=", evaluate(tokens))
    except RpnError as e:
        print("error:", e)
```

EXERCISE 3

Merge intervals with bisect queries

Read `n`, then `n` lines `a b` (inclusive integer intervals, in any order), then `q` and `q` query points one per line. Merge overlapping or touching intervals after sorting by start, and print `merged [a,b] [c,d] ...`, `covered <sum of (end - start) over the merged intervals>`, then for each query `t=<x> covered` or `t=<x> free`, found with `bisect_right` on the merged starts.

Input: the intervals, then the queries. **Output:** two summary lines, then one line per query.

```
3
1 3
2 6
8 10
3
5
7
10
```

prints

```
merged [1,6] [8,10]
covered 7
t=5 covered
t=7 free
t=10 covered
```

INPUT 1

```
3
1 3
2 6
8 10
3
5
7
10
```

OUTPUT 1

```
merged [1,6] [8,10]
covered 7
t=5 covered
t=7 free
t=10 covered
```

REFERENCE SOLUTION

```python
import sys
from bisect import bisect_right


def merge(intervals: list[tuple[int, int]]) -> list[tuple[int, int]]:
    merged = []
    for start, end in sorted(intervals):
        if merged and start <= merged[-1][1]:
            merged[-1] = (merged[-1][0], max(merged[-1][1], end))
        else:
            merged.append((start, end))
    return merged


def is_covered(merged: list[tuple[int, int]], starts: list[int], t: int) -> bool:
    i = bisect_right(starts, t) - 1
    return i >= 0 and merged[i][0] <= t <= merged[i][1]


def main() -> None:
    it = iter(sys.stdin.read().split())
    n = int(next(it))
    intervals = [(int(next(it)), int(next(it))) for _ in range(n)]
    q = int(next(it))
    queries = [int(next(it)) for _ in range(q)]
    merged = merge(intervals)
    starts = [a for a, _ in merged]
    print("merged", " ".join(f"[{a},{b}]" for a, b in merged))
    print("covered", sum(b - a for a, b in merged))
    for t in queries:
        print(f"t={t} {'covered' if is_covered(merged, starts, t) else 'free'}")


if __name__ == "__main__":
    main()
```

CHECKPOINT QUESTIONS

1. Which line of the template reads all input in one system call?

A. `input()`

B. `sys.stdin.buffer.read().split()`

C. `sys.stdin.readline()`

D. `open(0).read()` in a loop

2. "Longest subarray with at most k distinct values" is which shape?

A. Prefix sums

B. A sliding window with a count dict

C. A heap

D. Binary search

3. What is the O(1) idiom for "how many elements are less than x" on a sorted list?

A. `xs.index(x)`

B. `bisect_left(xs, x)` — O(log n)

C. `xs.count(x)`

D. A linear scan

4. Why is `for x in xs: xs.remove(x)` wrong?

A. `remove` is slow

B. Removal shifts elements under the iterator, so items are skipped

C. It raises

D. Lists are immutable during iteration

5. What does a closure created in a loop see when called later?

A. The loop variable's value at creation

B. The variable's final value — late binding — unless bound with a default argument

C. An error

D. `None`

6. What makes `put` on a chained hash map amortised O(1) despite the O(n) resize?

A. Resizes never happen

B. Doubling makes resizes rare enough that the total work spreads to a constant per insert

C. Buckets are sorted

D. Keys are ints

7. In an `OrderedDict` LRU cache, what does `popitem(last=False)` do?

A. Removes the newest

B. Removes the oldest — the least recently used

C. Clears the cache

D. Returns the size

8. Which is the clean shape for a judged solution?

A. Everything at module level

B. `main` for I/O, `parse` for typing the input, a pure `solve` and small helpers

C. One long function

D. A class for everything

9. What is `-7 % 2`?

A. -1

B. 1

C. 0

D. -7

10. Which comparison is right for floats that should be equal?

A. `==`

B. `math.isclose(a, b)`

C. `is`

D. `round(a) == round(b)`

11. What does `type(e).__name__` give inside `except Exception as e`?

A. The message

B. The exception class name, such as `ValueError`

C. The traceback

D. The module

12. In `D(B, C)` with both bases deriving from `A`, what is `D.__mro__`?

A. D, A, B, C

B. D, B, C, A, object

C. D, B, A, C, A

D. D, C, B, A, object

13. Why does a generator pipeline with `islice(…, 2)` read only as many inputs as needed?

A. It reads all and discards

B. Each `next` pulls one item through the stages; nothing runs ahead of demand

C. It uses threads

D. It caches

14. A rate limiter keeps allowed timestamps in a deque. Why a deque and not a list?

A. Deques are sorted

B. Expiring the oldest is `popleft()` in O(1); a list's `pop(0)` is O(n)

C. Lists cannot hold ints

D. Deques are thread-safe

15. How do you answer "is t inside one of the merged intervals" quickly?

A. Scan every interval

B. `bisect_right` on the starts to find the last interval starting at or before t, then check its end

C. A set of every integer

D. Recursion

ANSWER KEY · FINAL CHECKPOINT — INTERVIEW IDIOMS

1. B Bytes tokens, no decoding, `int()` works directly.

2. B Advance the start while the window has more than k distinct values.

3. B Logarithmic, not constant — but the point is that it is not linear.

4. B Build a new list with a comprehension.

5. B `lambda x, k=k: ...` binds now.

6. B Same argument as the dynamic array.

7. B `move_to_end` keeps the most recently used at the end.

8. B Testable, readable aloud, and the idea has a name.

9. B The remainder takes the divisor's sign in Python.

10. B Binary floats cannot represent most decimals exactly.

11. B Useful for reporting; catching the specific class is still the rule.

12. B C3: each class before its bases, bases in the declared order, `A` once.

13. B Laziness is per-item, end to end.

14. B Each timestamp is pushed and popped once: O(n) for the stream.

15. B O(log n) per query after one O(n log n) merge.