Module 6 of 20

Lists, tuples and sequences

Lists and their method costs, aliasing and copying, comprehensions in every form, sorting with keys, stability and bisect, tuples with unpacking and namedtuple, grids with neighbours and transposes, and the sequence tools with deque and the sequence protocol.

The list is where Python's name-and-object model first becomes visible — two names, one list, one surprise — and where the cost of an operation first matters: `append` is constant, `insert(0, x)` is linear, `x in xs` scans. This module teaches the list and tuple thoroughly, the comprehension syntax that builds collections from expressions, and the sorting machinery — keys, stability, `bisect` — that every ordering question reduces to. It then applies all of it to the grid, the data structure behind most interview problems, and closes with the built-ins that work on every sequence and the two methods that make a class one.

Lists covers construction, the method table with costs, shallow and deep copies, the multiplication trap and the rule against modifying while iterating. Comprehensions gives the list, set, dict and generator forms, nesting, scope and the readability limit. Sorting covers `sort` versus `sorted`, key recipes, stability and multi-pass sorts, `bisect`. Tuples and unpacking fixes the comma, immutability and its caveat, unpacking with stars, and `namedtuple`. Grids gives templates for reading, building, walking, neighbours with a bounds check, transposing and printing. Sequence tools collects `enumerate`, `zip`, `reversed`, `any` / `all`, `range` as a sequence, `deque`, `array` and the sequence protocol.

The exercises are whole programs: a list editor with every method and its failure, shallow versus deep copies made observable, comprehension drills, a word index built from set and dict comprehensions, rank queries with `bisect`, a two-pass stable sort, starred unpacking of variable records, named points with `_replace`, a Game of Life step, transposes and rotations, sliding-window maxima on a bounded `deque`, and a class that implements the sequence protocol. The checkpoint adds an inventory edited and reported in aligned columns, an island counter over a grid with an explicit stack, and a live leaderboard kept sorted with `insort`.

CONTENTS

1 Lists — the mutable sequence 14 min

2	Comprehensions — building collections from expressions	13 min
3	Sorting — sort, sorted, keys, stability and bisect	13 min
4	Tuples, unpacking and named tuples	13 min
5	Grids and nested lists	14 min
6	The sequence tools — enumerate, zip, reversed, any, all, deque and the protocol	13 min
✓	Checkpoint — Lists, tuples and sequences	25 min

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

LESSON 1 · 14 MIN

Lists — the mutable sequence

The list is Python's workhorse collection: an ordered, mutable sequence of any objects, growable at the end in constant time, indexable and sliceable like a string, and — because it is mutable — the first place the name-versus-object model of Module 2 produces a visible surprise. This lesson covers construction, the methods grouped by what they do, the cost of each, the aliasing and copying rules, and the `[[0] * n] * m` trap that every learner meets exactly once.

Building lists

```
empty = []
nums = [3, 1, 4, 1, 5]
mixed = [1, "two", 3.0, [4]]          # any objects, any mix
from_iter = list("abc")               # ['a', 'b', 'c'] — any iterable
zeros = [0] * 5                       # [0, 0, 0, 0, 0]
squares = [x * x for x in range(5)]   # comprehension (next lesson)
```

`list(iterable)` copies any iterable into a new list; `[x] * n` repeats a *reference* `n` times, which is fine for immutable `x` and the trap for mutable `x` (below).

Reading

`xs[i]`, negative indexes, and slicing work exactly as for strings (Module 5): `xs[-1]` is the last element, `xs[1:3]` a new list of two, `xs[::-1]` a reversed copy. `len(xs)`, `x in xs` (a linear scan), `xs.index(x)` (position or `ValueError`), `xs.count(x)`, `min` / `max` / `sum` / `sorted` all apply. Indexing past the end is `IndexError`; slicing clips.

Changing a list

Method	Effect	Cost
`xs.append(x)`	add at the end	O(1) amortised
`xs.extend(iter)` / `xs += iter`	add every element of an iterable	O(k)
`xs.insert(i, x)`	insert before index `i`	O(n) — shifts the tail
`xs.pop()`	remove and return the last	O(1)
`xs.pop(i)`	remove and return index `i`	O(n)
`xs.remove(x)`	remove the first equal element (`ValueError` if none)	O(n)
`del xs[i]`, `del xs[a:b]`	remove by index or slice	O(n)
`xs.clear()`	empty it	O(n)
`xs[i] = v`, `xs[a:b] = [...]`	replace an element or a slice (the slice may change length)	O(1) / O(n)
`xs.sort()`, `xs.reverse()`	in place	O(n log n) / O(n)

Every method in this table returns `None` — `append`, `sort`, `reverse`, `extend` all modify in place — so `xs = xs.append(4)` throws the list away (Module 4). The copying counterparts are `sorted(xs)` and `xs[::-1]` (or `reversed(xs)` for an iterator).

The costs matter in loops. `pop(0)` and `insert(0, x)` shift every element and make a queue built on a list quadratic; `collections.deque` is the constant-time answer (Module 6 lesson 6). `x in xs` scans the whole list; a `set` answers in constant time (Module 7). `append` is cheap because the list over-allocates and grows geometrically.

Aliasing and copying

```
a = [1, 2, 3]
b = a               # alias: same list
b.append(4)         # a is [1, 2, 3, 4]

c = a.copy()        # or list(a), or a[:] — a new list with the same elements
c.append(5)         # a unchanged
```

A copy made by `copy()`, `list()` or `[:]` is **shallow**: the new list holds the *same element objects*. For a list of numbers or strings that is a full copy, because those cannot change. For a list of lists it is not:

```
grid = [[1, 2], [3, 4]]
g2 = grid.copy()
g2[0].append(9)     # grid[0] is [1, 2, 9] too — the inner lists are shared
g2[0] = [0]         # grid[0] unchanged — rebinding an element of the copy is

import copy
g3 = copy.deepcopy(grid)   # recursive copy; nothing shared
```

Passing a list to a function passes the object, so the function can mutate it; if the caller must keep the original, the function should copy or the caller should pass a copy — say which in the docstring.

The multiplication trap

```
row = [0] * 3
grid = [row] * 3           # three references to ONE row
grid[0][0] = 1
print(grid)                # [[1, 0, 0], [1, 0, 0], [1, 0, 0]]

grid = [[0] * 3 for _ in range(3)]   # three separate rows
grid[0][0] = 1
print(grid)                # [[1, 0, 0], [0, 0, 0], [0, 0, 0]]
```

`[x] * n` copies the reference, not the object. For an inner list the comprehension creates a fresh one per row — the only correct way to build a grid of mutable rows (Module 6 lesson 5).

Equality and ordering

`==` compares element by element (and nested lists recursively); `is` compares identity. Lists compare lexicographically with `<`, like strings and tuples, which is what makes `sorted` on a list of lists work. `xs == []` and `not xs` both test emptiness; the second is idiomatic.

Iterating and modifying

Do not append to or remove from a list while a `for` is iterating over it (Module 3). Build a new list with a comprehension, iterate over a copy (`for x in xs[:]`), or iterate by index *backwards* when deleting in place — `for i in range(len(xs) - 1, -1, -1)` — so that deletions do not shift elements you have yet to visit. Assigning `xs[i] = …` during iteration is fine.

Pitfalls

- `xs = xs.sort()` (or `append`, `reverse`).
- `[[0] * n] * m`.
- `b = a` when a copy was meant; `a.copy()` when a deep copy was meant.
- `pop(0)` / `insert(0, …)` in a loop — use a `deque`.
- `x in xs` inside a loop over another list — quadratic; use a set.
- `xs.remove(x)` inside `for x in xs`.

Key takeaways

- Lists are mutable, ordered, heterogeneous; `list(iterable)` copies, `[x] * n` repeats references.
- `append` / `pop()` are O(1); `insert` / `pop(i)` / `remove` / `in` are O(n); in-place methods return `None`.
- `b = a` aliases; `a.copy()` / `list(a)` / `a[:]` copy shallowly; `copy.deepcopy` copies nested structure.
- `[[0] * n for _ in range(m)]` for a grid — never `[[0] * n] * m`.
- Never change a list's length while iterating it; build a new list or iterate a copy.

EXERCISE 1

List editor

Maintain a list of integers from commands read until the end of input: `push v` appends; `pop` removes the last (`empty` if there is none); `insert i v` inserts before index `i`; `remove v` removes the first equal value (`not found` if absent); `del i` deletes by index (`out of range` if invalid); `show` prints the list. Every error is reported without crashing, and the list is printed as space-separated values or `(empty)`.

Input: commands. **Output:** one line per `show` and per error.

```
push 1
push 2
insert 0 9
remove 2
remove 7
show
```

prints

```
not found
9 1
```

INPUT 1

```
push 1
push 2
insert 0 9
remove 2
remove 7
show
```

OUTPUT 1

```
not found
9 1
```

REFERENCE SOLUTION

```python
import sys

xs = []
for line in sys.stdin:
    cmd, *args = line.split()
    if cmd == "push":
        xs.append(int(args[0]))
    elif cmd == "pop":
        if xs:
            xs.pop()
        else:
            print("empty")
    elif cmd == "insert":
        xs.insert(int(args[0]), int(args[1]))
    elif cmd == "remove":
        value = int(args[0])
        if value in xs:
            xs.remove(value)
        else:
            print("not found")
    elif cmd == "del":
        i = int(args[0])
        if -len(xs) ⩽ i < len(xs):
            del xs[i]
        else:
            print("out of range")
    elif cmd == "show":
        print(" ".join(map(str, xs)) if xs else "(empty)")
```

EXERCISE 2

Shallow versus deep

Read a list literal (parse it with `ast.literal_eval`), make a **shallow** copy with `list()` and a **deep** copy with `copy.deepcopy`, then mutate the original: if its first element is a list, append `99` to that inner list; otherwise append `99` to the outer list. Print all three lists so the sharing is visible.

Input: one list literal. **Output:** `original: <list>`, `shallow: <list>`, `deep: <list>` (printed with `print`'s own list format).

```
[[1, 2], [3]]
```

prints

```
original: [[1, 2, 99], [3]]
shallow: [[1, 2, 99], [3]]
deep: [[1, 2], [3]]
```

INPUT 1

```
[[1, 2], [3]]
```

OUTPUT 1

```
original: [[1, 2, 99], [3]]
shallow: [[1, 2, 99], [3]]
deep: [[1, 2], [3]]
```

REFERENCE SOLUTION

```python
import ast
import copy

original = ast.literal_eval(input())
shallow = list(original)
deep = copy.deepcopy(original)
if original and isinstance(original[0], list):
    original[0].append(99)
else:
    original.append(99)
print("original:", original)
print("shallow:", shallow)
print("deep:", deep)
```

QUIZ

1. Which operation is O(n) on a list of length n?

A. `xs.append(x)`

B. `xs.pop()`

C. `xs.insert(0, x)`

D. `xs[i]`

2. What does this print?

```
grid = [[0] * 2] * 2
grid[0][0] = 1
print(grid)
```

A. `[[1, 0], [0, 0]]`

B. `[[1, 0], [1, 0]]`

C. `[[1, 1], [1, 1]]`

D. `[[0, 0], [0, 0]]`

3. After `a = [[1], [2]]; b = a.copy(); b[0].append(9); b[1] = [0]`, what is `a`?

A. `[[1], [2]]`

B. `[[1, 9], [2]]`

C. `[[1, 9], [0]]`

D. `[[1], [0]]`

4. Which line does what its author intended?

A. `xs = xs.append(4)`

B. `xs = xs.sort()`

C. `xs = sorted(xs)`

D. `xs = xs.reverse()`

5. What is the safe way to delete every element that satisfies `bad(x)` from `xs`?

A. `for x in xs: if bad(x): xs.remove(x)`

B. `xs = [x for x in xs if not bad(x)]`

C. `for i in range(len(xs)): if bad(xs[i]): del xs[i]`

D. `xs.remove(bad)`

ANSWER KEY · LISTS — THE MUTABLE SEQUENCE

1. C Inserting at the front shifts every element. Appending, popping the end and indexing are constant time; `collections.deque` makes front insertion O(1).

2. B `* 2` repeats the reference to one inner list, so both rows are the same object. `[[0] * 2 for _ in range(2)]` creates two rows.

3. B The shallow copy shares the inner lists, so appending through `b[0]` is visible in `a`; rebinding `b[1]` only changes the copy's slot.

4. C `sorted` returns a new list. The in-place methods return `None`, so the other assignments discard the list.

5. B Building a new list never skips or shifts. Removing during forward iteration skips neighbours, and deleting by ascending index raises `IndexError` once the list shrinks.

LESSON 2 · 13 MIN

Comprehensions — building collections from expressions

A list comprehension is a loop, a filter and a transform folded into one expression that *builds a list*: `[f(x) for x in xs if p(x)]`. It replaces the four-line pattern of creating an empty list, looping, testing and appending, and it says what the result *is* rather than how it is assembled. Python has the same shape for sets and dictionaries and a lazy version for generators. This lesson teaches the syntax, the nested forms, the readability limit past which a loop is better, the scoping rule, and where a comprehension is the wrong tool — when the loop body has side effects, or when the result is thrown away.

The basic form

```
squares = []                            # the loop
for x in range(6):
    if x % 2 == 0:
        squares.append(x * x)

squares = [x * x for x in range(6) if x % 2 == 0]     # the comprehension: [0,
```

Read it as: *the list of* `x * x` *for each* `x` *in* `range(6)` *such that* `x` *is even*. The expression comes first, the `for` supplies the elements, the optional `if` filters. The expression can be anything — a call, a tuple, a conditional expression:

```
labels = ["even" if x % 2 == 0 else "odd" for x in xs]   # a conditional expre
pairs = [(x, x * x) for x in xs]
lengths = [len(w) for w in words]
cleaned = [w.strip().lower() for w in words if w.strip()]
```

Note the two positions of `if`: `if` *after* the `for` filters (produces fewer elements); a conditional expression *before* the `for` chooses a value (produces one per element).

Nested loops

Several `for` clauses nest in reading order, left to right — the first `for` is the outer loop:

```
pairs = [(a, b) for a in range(3) for b in range(2)]
# [(0, 0), (0, 1), (1, 0), (1, 1), (2, 0), (2, 1)]

flat = [x for row in grid for x in row]          # flatten one level: outer lo
triangles = [(a, b) for a in range(1, 5) for b in range(a, 5)]   # the inner r
```

A comprehension *inside* a comprehension builds nested output:

```
grid = [[0] * cols for _ in range(rows)]          # a fresh row per iteration
table = [[r * c for c in range(1, 4)] for r in range(1, 4)]
transposed = [[row[i] for row in matrix] for i in range(len(matrix[0]))]
```

Two nested levels is the readable limit. Three `for` clauses, or a comprehension inside a comprehension inside a comprehension, is a loop written as a puzzle — expand it.

Set, dict and generator forms

```
unique_lengths = {len(w) for w in words}                    # a set: braces, one
index = {w: i for i, w in enumerate(words)}                 # a dict: key: value
squares_by_x = {x: x * x for x in range(4)}
total = sum(x * x for x in xs)                              # a generator: no li
first_neg = next((x for x in xs if x < 0), None)
```

The dict comprehension is the idiom for inverting a mapping (`{v: k for k, v in d.items()}`) and for building lookups. The generator expression (Module 11) is the same syntax in parentheses and produces values lazily — the right form when the consumer (`sum`, `max`, `any`, `"".join`) only needs to see each value once, because no intermediate list is allocated. When a generator is the sole argument to a call, its parentheses double as the call's: `sum(x for x in xs)`.

Comprehension versus map/filter versus loop

```
[int(t) for t in tokens]          # comprehension
list(map(int, tokens))            # map — fine when the function already exists
[x for x in xs if x > 0]          # comprehension
list(filter(lambda x: x > 0, xs))   # filter with a lambda — the comprehension
```

Reach for `map` when the function exists (`int`, `str.strip`, a `def`); reach for the comprehension whenever a lambda would be needed or a filter and a transform combine. Write a plain `for` loop when the body does something *other* than produce a value — printing, updating several variables, breaking early — and never write a comprehension whose result you discard just to run its side effects (`[print(x) for x in xs]` is a loop wearing a costume).

Scope

A comprehension has its own scope: the loop variable does not leak.

```
x = "outer"
squares = [x * x for x in range(3)]
print(x)          # 'outer' — untouched (in Python 2 it would have been 2)
```

The comprehension can read enclosing names (`[x * factor for x in xs]`), and the walrus operator binds in the *enclosing* scope: `[y for x in xs if (y := f(x)) > 0]` leaves `y` defined afterwards — occasionally useful for "compute once, filter and keep".

Performance

A comprehension is somewhat faster than the equivalent `append` loop because the interpreter uses a specialised instruction for the append, and it allocates the list once when it can predict the size. That is a constant-factor gain, not an algorithmic one; a comprehension containing `x in other_list` is still quadratic. The generator form uses O(1) memory where the list form uses O(n) — the difference that matters at scale.

Pitfalls

A comprehension for side effects (`[print(x) …]`).

Three or more nested clauses; a conditional expression with an `if` filter and a nested comprehension on one line.

Putting the filter `if` in front of the `for` (syntax error) or the conditional expression after it (also wrong).

`[[0] * n] * m` instead of a comprehension for the rows.

A list comprehension where a generator would do — `sum([x for x in xs])` builds a list for nothing.

Expecting the loop variable to be visible after the comprehension.

Key takeaways

`[expr for x in iterable if cond]` — expression, loop, filter; a conditional expression in the `expr` position chooses a value per element.

Several `for` clauses nest left to right; nested comprehensions build nested lists — a fresh inner list each time.

`{…}` builds a set or a dict (`k: v`); `(…)` is a lazy generator, ideal as the argument to `sum` , `max` , `any` , `join` .

Prefer `map` when the function exists, the comprehension when a lambda would be needed, a loop when there are side effects or early exit.

Comprehensions have their own scope; keep them to two levels.

EXERCISE 1

Comprehension drills

From one line of integers produce four results, each with a single comprehension: the squares of the even values; a label per value (`pos` , `neg` or `zero`) using a conditional expression; the number of pairs `(a, b)` with `a < b` taken by index order from the list (a nested `for` with a filter, counted with `len`); and the sum of the absolute values as a generator expression inside `sum` .

Input: one line of integers. **Output:** `squares: <values>` , `labels: <words>` , `pairs: <n>` , `abs-sum: <n>` (an empty list prints `squares:` with nothing after it).

```
1 -2 3 0
```

prints

```
squares: 4 0
labels: pos neg pos zero
pairs: 3
abs-sum: 6
```

INPUT 1

```
1 -2 3 0
```

OUTPUT 1

```
squares: 4 0
labels: pos neg pos zero
pairs: 3
abs-sum: 6
```

REFERENCE SOLUTION

```
xs = list(map(int, input().split()))
squares = [x * x for x in xs if x % 2 == 0]
labels = ["pos" if x > 0 else "neg" if x < 0 else "zero" for x in xs]
pairs = len([(a, b) for i, a in enumerate(xs) for b in xs[i + 1:] if a < b])
abs_sum = sum(abs(x) for x in xs)
print("squares:", " ".join(map(str, squares)))
print("labels:", " ".join(labels))
print(f"pairs: {pairs}")
print(f"abs-sum: {abs_sum}")
```

EXERCISE 2

Word index

Read one line of words. Print the distinct words in sorted order (a set comprehension over the lower-cased words, then `sorted`); a mapping from each distinct word to its length in first-seen order (a dict comprehension over `dict.fromkeys(words)`, which keeps order); and the total number of characters (a generator expression in `sum`).

Input: one line of words. **Output:** `unique: <sorted words>`, `lengths: <word=len pairs>`, `total: <n>`.

```
Kiwi apple kiwi Fig
```

prints

```
unique: apple fig kiwi
lengths: kiwi=4 apple=5 fig=3
total: 16
```

INPUT 1

```
Kiwi apple kiwi Fig
```

OUTPUT 1

```
unique: apple fig kiwi
lengths: kiwi=4 apple=5 fig=3
total: 16
```

REFERENCE SOLUTION

```
words = [w.lower() for w in input().split()]
unique = {w for w in words}
lengths = {w: len(w) for w in dict.fromkeys(words)}
total = sum(len(w) for w in words)
print("unique:", " ".join(sorted(unique)))
print("lengths:", " ".join(f"{w}={n}" for w, n in lengths.items()))
print(f"total: {total}")
```

QUIZ

1. What is `[x if x > 0 else 0 for x in [-1, 2]]`?

A. `[2]`

B. `[0, 2]`

C. `SyntaxError`

D. `[-1, 2]`

2. What is `[(a, b) for a in range(2) for b in "xy"]`?

A. `[(0, 'x'), (0, 'y'), (1, 'x'), (1, 'y')]`

B. `[(0, 'x'), (1, 'y')]`

C. `[(0, 'x'), (1, 'x'), (0, 'y'), (1, 'y')]`

D. `SyntaxError`

3. What does `x` hold afterwards?

```
x = 10
ys = [x for x in range(3)]
```

A. `2`

B. `10`

C. `[0, 1, 2]`

D. `NameError`

4. Which is the better spelling of `sum([x * x for x in xs])`?

A. `sum(x * x for x in xs)`

B. `sum(map(x * x, xs))`

C. `reduce(sum, xs)`

D. It is already optimal

5. Why is `[print(x) for x in xs]` discouraged?

A. `print` cannot be called in a comprehension

B. It builds a list of `None` values only to run side effects — a `for` loop says what is meant

C. It prints in reverse order

D. It is slower than `map(print, xs)`

ANSWER KEY · COMPREHENSIONS — BUILDING COLLECTIONS FROM EXPRESSIONS

1. B A conditional expression before the `for` produces one value per element. A filtering `if` after the `for` would produce `[2]`.

2. A Multiple `for` clauses nest left to right: the first is the outer loop.

3. B A comprehension has its own scope; its loop variable does not leak into the enclosing scope.

4. A The generator form feeds `sum` one value at a time without building an intermediate list. `map` needs a function, and `reduce(sum, …)` is wrong.

5. B Comprehensions build values; a loop with side effects should be a loop. The list of `None`s is allocated and thrown away.

LESSON 3 · 13 MIN

Sorting — sort, sorted, keys, stability and bisect

Python sorts with one algorithm — Timsort, a stable merge sort tuned for runs that already exist in the data — behind two calls: `list.sort()` in place and `sorted()` returning a new list. Every ordering question reduces to choosing a *key function*, which is why the previous module's key idiom matters here, and stability is what makes multi-pass and multi-key sorting predictable. This lesson gives the two calls, the key recipes (tuples, negation, case-insensitivity, `itemgetter`), the stability guarantee and what it buys, `bisect` for keeping a sorted list sorted, and the reasons `cmp_to_key` is rarely needed.

---

sort versus sorted

```
xs = [3, 1, 2]
ys = sorted(xs)      # [1, 2, 3]; xs unchanged; works on any iterable, always
xs.sort()            # xs is now [1, 2, 3]; returns None
sorted("bca")        # ['a', 'b', 'c'] — from a string, a list comes back
sorted({3: "a", 1: "b"})   # [1, 3] — the keys
```

Use `sorted` when the original must survive or the input is not a list; use `sort` when the list is yours and the copy would be waste. Both take `key=` and `reverse=`. Elements must be mutually comparable: `sorted([1, "a"])` is a `TypeError`.

Keys

The key function is called once per element and its result is what gets compared:

```
words = ["banana", "Apple", "cherry"]
sorted(words, key=str.lower)                    # case-insensitive
sorted(words, key=len)                          # by length; equal lengths kee
sorted(words, key=lambda w: (len(w), w))        # by length, then alphabetical
sorted(nums, key=abs)                           # by magnitude
sorted(people, key=lambda p: (-p.age, p.name))  # age descending, name ascendi
sorted(rows, key=itemgetter(2, 0))              # by column 2 then column 0
sorted(items, key=attrgetter("price"))          # by attribute
```

The tuple key is the multi-key sort: tuples compare element by element, so `(len(w), w)` orders by length first and breaks ties alphabetically. Negation reverses a numeric component only; for a string component that must go the other way, use two stable sorts (below) or `reverse=True` on the whole key.

`reverse=True` reverses the *result order* while keeping stability: equal keys still appear in their original relative order, not reversed.

Stability

A stable sort keeps elements with equal keys in their input order. Timsort is stable, and that guarantee turns into two techniques. First, a sort by a secondary key followed by a sort by the primary key produces a multi-key ordering:

```
rows.sort(key=itemgetter(1))          # secondary key first
rows.sort(key=itemgetter(0))          # primary key last; ties keep the second
```

That is how you sort by one field descending and another ascending when the tuple trick does not apply (`rows.sort(key=name)` then `rows.sort(key=score, reverse=True)`). Second, `sorted(xs, key=…)` on already-grouped data keeps each group's internal order — useful when the input order is meaningful (arrival order, file order).

Sorting objects without a key

Tuples, lists and strings compare lexicographically, so `sorted(pairs)` sorts by first element then second without any key — but it also compares the second elements when the first are equal, which is a `TypeError` if those are incomparable (`None`, mixed types). Give a key when the tie-break should not touch the other fields. Your own classes sort once they define `__lt__` (Module 8); dataclasses with `order=True` get it for free.

---

min, max and the top k

`min(xs, key=…)` and `max(xs, key=…)` take the same key and return the *first* extreme element on a tie. For the top `k` of `n` elements, `sorted(xs, reverse=True)[:k]` is O(n log n); `heapq.nlargest(k, xs, key=…)` is O(n log k) and is what you want when `k` is small (Module 7 lesson 6).

bisect: keeping a list sorted

`bisect` finds where a value belongs in a sorted list by binary search, in O(log n):

```
import bisect
xs = [1, 3, 3, 7]
bisect.bisect_left(xs, 3)     # 1 — first position where 3 could go (before ea
bisect.bisect_right(xs, 3)    # 3 — after equal elements
bisect.insort(xs, 5)          # xs is [1, 3, 3, 5, 7] — insert keeping order (
bisect.bisect_left(xs, 4)     # 3 — where 4 would go: the count of elements <
```

`bisect_left` answers "how many elements are less than x" and "is x present" (`i < len(xs) and xs[i] == x`); `bisect_right - bisect_left` counts occurrences. Since 3.10 the functions take `key=`. Grade boundaries, percentile lookups and "first element ≥ x" are all one `bisect` call on a sorted list; a linear scan for the same is O(n) per query.

cmp_to_key

Older code and other languages sort with a *comparison* function returning negative, zero or positive. Python dropped `cmp=` because keys are faster (one call per element, not per comparison) and clearer. When an ordering genuinely cannot be expressed as a key — comparing strings by a custom rule that depends on both operands — `functools.cmp_to_key(compare)` wraps a comparison function into a key. It is rare; if you reach for it, first check whether a tuple key does the job.

Pitfalls

- `xs = xs.sort()`.
- `sorted` on mixed types, or a tuple key whose tie-break compares incomparable values.
- `key=lambda w: -w` on strings; use `reverse=True` or two sorts.
- Expecting `reverse=True` to reverse the order of equal elements.
- Linear search in a sorted list where `bisect` is O(log n).
- `sorted(...)[:k]` for a small `k` on a huge list — `heapq.nlargest`.

Key takeaways

`sorted(iterable)` returns a new list; `list.sort()` sorts in place and returns `None`; both take `key` and `reverse`.

A key returning a tuple sorts by several criteria; negate a numeric component to reverse it; `itemgetter` / `attrgetter` build keys.

Timsort is stable: equal keys keep input order, so successive sorts (secondary then primary) compose.

`min` / `max` with a key give the first extreme; `heapq.nlargest` for a small top-k.

`bisect_left` / `bisect_right` / `insort` search and insert in a sorted list in O(log n).

EXERCISE 1

Rank queries with bisect

Read a line of scores, then `q` queries. The rank of a score `x` is one plus the number of scores strictly greater than `x`. Sort the scores once (ascending) and answer each query in O(log n) with `bisect.bisect_right`.

Input: a line of integers, then `q`, then `q` integers. **Output:** `rank <r>` per query.

```
50 80 80 90
3
80
100
0
```

prints

```
rank 2
rank 1
rank 5
```

INPUT 1

```
50 80 80 90
3
80
100
0
```

OUTPUT 1

```
rank 2
rank 1
rank 5
```

REFERENCE SOLUTION

```python
import bisect

scores = sorted(map(int, input().split()))
q = int(input())
for _ in range(q):
    x = int(input())
    greater = len(scores) - bisect.bisect_right(scores, x)
    print(f"rank {greater + 1}")
```

EXERCISE 2

Two stable passes

Read `n` employee records `name dept salary` and print them ordered by department ascending, then salary **descending**, then name ascending — using **two** `sort` calls, secondary keys first, relying on stability (no tuple with a negated string is possible here, so this is the technique).

Input: `n`, then `n` lines. **Output:** the records, one per line, as `dept name salary`.

```
4
ada eng 120
bob ops 90
cy eng 120
dee eng 100
```

prints

```
eng ada 120
eng cy 120
eng dee 100
ops bob 90
```

INPUT 1

```
4
ada eng 120
bob ops 90
cy eng 120
dee eng 100
```

OUTPUT 1

```
eng ada 120
eng cy 120
eng dee 100
ops bob 90
```

REFERENCE SOLUTION

```python
n = int(input())
rows = []
for _ in range(n):
    name, dept, salary = input().split()
    rows.append((name, dept, int(salary)))
rows.sort(key=lambda r: r[0])
rows.sort(key=lambda r: r[2], reverse=True)
rows.sort(key=lambda r: r[1])
for name, dept, salary in rows:
    print(dept, name, salary)
```

QUIZ

1. What is `sorted(["b", "A", "c"], key=str.lower)`?

A. `['A', 'b', 'c']`

B. `['b', 'c', 'A']`

C. `['a', 'b', 'c']`

D. `['A', 'c', 'b']`

2. Which key sorts pairs by second element descending, then first ascending?

A. `key=lambda p: (-p[1], p[0])`

B. `key=lambda p: (p[1], p[0])` with `reverse=True`

C. `key=lambda p: p[1] - p[0]`

D. `key=lambda p: (p[0], -p[1])`

3. Timsort is stable. What does that guarantee?

A. It never uses extra memory

B. Elements with equal keys keep their relative input order

C. It runs in O(n) on any input

D. It sorts strings before numbers

4. For sorted `xs = [1, 3, 3, 7]`, what is `bisect.bisect_left(xs, 3)` and `bisect.bisect_right(xs, 3)`?

A. `1` and `3`

B. `1` and `2`

C. `2` and `3`

D. `3` and `1`

5. What does `sorted([3, 1, 2], reverse=True)[:1]` cost, and what is cheaper for the largest element?

A. O(n log n); `max(xs)` is O(n)

B. O(n); nothing is cheaper

C. O(1); it is optimal

D. O(n²); `heapq` is O(1)

6. What happens with `sorted([1, "2", 3])`?

A. `[1, 3, '2']`

B. `TypeError` — int and str are not comparable

C. `['2', 1, 3]`

D. `[1, 2, 3]`

ANSWER KEY · SORTING — SORT, SORTED, KEYS, STABILITY AND BISECT

1. A The key lowercases for comparison only; the original strings are returned in that order.

2. A Negating the numeric second element reverses just that component; `reverse=True` would reverse both.

3. B Stability is what lets a sort by a secondary key followed by a sort by the primary key produce a multi-key order.

4. A `bisect_left` gives the first position where 3 fits (before equal elements), `bisect_right` the position after them; the difference is the count of 3s.

5. A Sorting the whole list to take one element is wasteful; `max` scans once. For a small top-k, `heapq.nlargest` is O(n log k).

6. B Ordering comparisons between unrelated types raise. Convert first, or give a key that maps everything to one type.

LESSON 4 · 13 MIN

Tuples, unpacking and named tuples

A tuple is an immutable sequence, and that one property makes it the right type for three jobs: a fixed-shape record (`(name, age, city)`), a value returned from a function that has several results, and a key in a dictionary or a member of a set. Python's unpacking syntax — `a, b = pair`, `first, *rest = xs`, the parallel assignment `a, b = b, a` — is what makes tuples pleasant rather than clumsy. This lesson fixes the literal syntax (the comma, not the parentheses), the immutability rule and its one exception, unpacking in all its forms, and `namedtuple` for records whose fields deserve names.

Literals — it is the comma

```
point = (3, 4)
point = 3, 4          # the same tuple: the comma makes it, the parentheses gr
single = (5,)         # a one-element tuple needs the trailing comma
not_a_tuple = (5)     # just the int 5 in parentheses
empty = ()
from_list = tuple([1, 2, 3])
```

Parentheses are required only where the comma would be ambiguous — inside a function call, a subscript or a larger expression — and by convention for readability. `type((1))` is `int`; `type((1,))` is `tuple`. Everything a string or list can do read-only, a tuple can do: index, slice, `len`, `in`, iterate, `count`, `index`, compare lexicographically, concatenate with `+`, repeat with `*`.

Immutable, with a caveat

```
t = (1, 2, 3)
t[0] = 9              # TypeError: 'tuple' object does not support item assigr
t = t + (4,)          # a new tuple; the name rebinds

t = ([1, 2], 3)
t[0].append(9)        # fine — the tuple still holds the same list object, whi
t[0] = [0]            # TypeError — the *slot* cannot be rebound
```

The tuple's *structure* is fixed: which objects it references, in what order. Whether those objects change is up to them. A tuple of numbers and strings is therefore fully immutable and hashable; a tuple containing a list is neither hashable (`{([1], 2)}` raises `TypeError: unhashable type: 'list'`) nor truly frozen.

Why tuples

Records. A row from a file, a coordinate, an RGB triple: fixed length, position has meaning. Lists are for collections of *the same kind of thing* with a variable length.

Multiple return values. `return lo, hi` builds a tuple; the caller unpacks it.

Dictionary keys and set members. `seen = {(r, c)}`, `distances[(a, b)]` — a list cannot be a key (Module 7).

Safety. A function receiving a tuple cannot accidentally append to it.

Speed and size. Slightly smaller and faster to build than a list; a constant tuple is built once at compile time.

Unpacking

```
a, b = 1, 2                          # parallel assignment
a, b = b, a                          # swap: the right side is built before any r
x, y, z = "xyz"                      # any iterable of the right length
(name, age), city = ("Ada", 36), "London"     # nested
first, *middle, last = [1, 2, 3, 4, 5]        # first=1, middle=[2, 3, 4], las
head, *tail = [1]                             # head=1, tail=[]
*init, last = xs
```

The counts must match, or `ValueError: too many values to unpack (expected 2)` / `not enough values`. A starred target absorbs the surplus as a list (possibly empty) and may appear once, anywhere. Unpacking works in every binding position — `for k, v in d.items()`, `for i, (a, b) in enumerate(pairs)`, `lambda p: p[0]` versus `def f(p): x, y = p`, and in function calls as `f(*args)`.

The swap works because the right-hand tuple is fully evaluated first; `xs[i], xs[j] = xs[j], xs[i]` swaps list elements in one line for the same reason.

Ignoring values

`_` is the conventional name for a value you do not need: `for _ in range(3)`, `name, _, city = row`, `*_, last = xs`. It is an ordinary name and nothing stops you reading it; the convention is for the reader.

Named tuples

A tuple whose fields have names, with the tuple's size and immutability and a readable `repr`:

```
from collections import namedtuple

Point = namedtuple("Point", ["x", "y"])
p = Point(3, 4)
p.x, p[0]                  # 3, 3 — by name or by position
x, y = p                   # still unpacks
p._replace(x=0)            # a new Point(0, 4)
p._asdict()                # {'x': 3, 'y': 4}
Point._fields              # ('x', 'y')
print(p)                   # Point(x=3, y=4)
```

`typing.NamedTuple` is the class-syntax version with type hints:

```
from typing import NamedTuple

class Point(NamedTuple):
    x: int
    y: int = 0

    def norm(self) -> float:
        return (self.x ** 2 + self.y ** 2) ** 0.5
```

Named tuples are the lightest way to give a record a shape; when you need mutability, defaults with factories or methods that change state, Module 8's `dataclass` is the next step up.

Pitfalls

- `(5)` when `(5,)` was meant.

Assuming a tuple with a list inside is hashable or frozen.

Unpacking a row with the wrong number of fields — the `ValueError` is the input validation.

Using a list for a fixed record (`row[2]` with no meaning) where a named tuple would document the fields.

Writing `return (a, b)` and `x = f()[0]` when unpacking would be clearer.

Two starred targets in one unpacking (a syntax error).

Key takeaways

The comma makes a tuple; `(5,)` is a one-tuple, `(5)` is an int.

Tuples fix their structure, not their contents; a tuple of immutables is hashable and a valid dict key.

Use tuples for fixed-shape records, multiple returns and keys; lists for variable collections of like items.

`a, b = b, a`, `first, *rest = xs`, nested targets, and unpacking in `for` and calls; counts must match.

`namedtuple` / `NamedTuple` add field names, `_replace` and `_asdict` to a tuple; `dataclass` is the mutable step up.

EXERCISE 1

Records with a star

Each line holds a name, then any number of integer scores (possibly none), then a city. Unpack every line with one starred assignment — `name, *scores, city = tokens` — and print the name, the city and the mean score with two decimals, or `no scores`.

Input: lines. **Output:** `<name> <city> <mean>` or `<name> <city> no scores` per line.

```
ada 90 80 london
bob paris
```

prints

```
ada london 85.00
bob paris no scores
```

INPUT 1

```
ada 90 80 london
bob paris
```

OUTPUT 1

```
ada london 85.00
bob paris no scores
```

REFERENCE SOLUTION

```python
import sys

for line in sys.stdin:
    tokens = line.split()
    name, *scores, city = tokens
    if scores:
        mean = sum(map(int, scores)) / len(scores)
        print(f"{name} {city} {mean:.2f}")
    else:
        print(f"{name} {city} no scores")
```

EXERCISE 2

Named points

Define `Point = namedtuple("Point", "x y")`. Read `n` points and print each one's `repr` with its distance from the origin (two decimals); then the farthest point (`max` with a key; first on ties); then that point reflected through the origin using `_replace`.

Input: `n`, then `n` lines `x y`. **Output:** `n` lines `Point(x=..., y=...) <dist>`, then `farthest: <repr>`, then `reflected: <repr>`.

```
2
3 4
-1 0
```

prints

```
Point(x=3, y=4) 5.00
Point(x=-1, y=0) 1.00
farthest: Point(x=3, y=4)
reflected: Point(x=-3, y=-4)
```

INPUT 1

```
2
3 4
-1 0
```

OUTPUT 1

```
Point(x=3, y=4) 5.00
Point(x=-1, y=0) 1.00
farthest: Point(x=3, y=4)
reflected: Point(x=-3, y=-4)
```

REFERENCE SOLUTION

```python
import math
from collections import namedtuple

Point = namedtuple("Point", "x y")


def dist(p):
    return math.hypot(p.x, p.y)


n = int(input())
points = []
for _ in range(n):
    x, y = map(int, input().split())
    points.append(Point(x, y))
for p in points:
    print(f"{p!r} {dist(p):.2f}")
far = max(points, key=dist)
print(f"farthest: {far!r}")
print(f"reflected: {far._replace(x=-far.x, y=-far.y)!r}")
```

QUIZ

1. What is `type((1))` and `type((1,))`?

A. `tuple` and `tuple`

B. `int` and `tuple`

C. `int` and `int`

D. `tuple` and `int`

2. What happens?

```
t = ([1], 2)
t[0].append(3)
t[0] = [9]
```

A. Both lines succeed

B. The append succeeds; the assignment raises `TypeError`

C. Both raise `TypeError`

D. The append raises; the assignment succeeds

3. What does `first, *rest = [1]` bind?

A. `first = 1, rest = []`

B. `first = 1, rest = None`

C. `ValueError`

D. `first = [1], rest = []`

4. Why does `a, b = b, a` swap correctly?

A. Python has a special swap instruction

B. The right-hand tuple is fully built before any left-hand name is bound

C. Because tuples are immutable

D. It does not — it needs a temporary

5. Which is true of `namedtuple`?

A. Its instances are mutable like dataclasses

B. It is a tuple subclass with named fields, so unpacking and indexing still work and `_replace` returns a new instance

C. Fields can only be accessed by name

D. It cannot be used as a dictionary key

ANSWER KEY · TUPLES, UNPACKING AND NAMED TUPLES

1. B Parentheses alone only group; the comma makes the tuple.

2. B The tuple's slots cannot be rebound, but the list inside is an ordinary mutable object that can be changed through the tuple.

3. A The starred target absorbs the surplus, which may be nothing — an empty list.

4. B Assignment evaluates the whole right side first, then binds left to right, so the old values are captured before either name changes.

5. B A named tuple is still a tuple: immutable, hashable (if its fields are), indexable and unpackable, with attribute access added.

LESSON 5 · 14 MIN

Grids and nested lists

A two-dimensional grid — a game board, a matrix, a maze, a spreadsheet — is a list of rows, each row a list of cells, indexed `grid[r][c]` with the row first. Almost every interview problem over grids uses the same six operations: read it from input, build an empty one of a given size, walk every cell, visit a cell's neighbours without falling off the edge, transpose or rotate it, and print it back. This lesson gives each as a template, states the row-major convention and the `[[0] * C] * R` trap once more where it does the damage, and shows the flattening trick for problems that are one-dimensional in disguise.

Reading a grid

```
R, C = map(int, input().split())
grid = [list(map(int, input().split())) for _ in range(R)]      # numbers with
maze = [list(input().rstrip("\n")) for _ in range(R)]           # characters,
maze = [input().rstrip("\n") for _ in range(R)]                 # rows as str
```

Rows as strings are fine for reading (`maze[r][c]` works) and immutable; convert each row to a list when cells must be changed. Trust the declared `R`; validate `len(row) == C` if the input might be ragged.

Building an empty grid

```
grid = [[0] * C for _ in range(R)]        # R independent rows
grid = [[0] * C] * R                      # WRONG: R references to one row
```

The second form is Module 6 lesson 1's trap in its natural habitat: writing `grid[0][0] = 1` sets column 0 of *every* row. The comprehension runs `[0] * C` once per row and is the only correct spelling. For a grid of strings or `None`, the same shape: `[[None] * C for _ in range(R)]`.

Walking every cell

```
for r in range(R):
    for c in range(C):
        cell = grid[r][c]

for r, row in enumerate(grid):            # when the row itself is useful
    for c, cell in enumerate(row):
        ...

for row in grid:                          # order does not matter, index not r
    for cell in row:
        total += cell
```

`sum(sum(row) for row in grid)` sums a numeric grid; `sum(row.count("#") for row in grid)` counts a character; `max(max(row) for row in grid)` finds the largest. Column sums come from the transpose (below) or `sum(grid[r][c] for r in range(R))`.

Neighbours and bounds

```
DIRS4 = [(-1, 0), (1, 0), (0, -1), (0, 1)]                    # up, down, le
DIRS8 = [(dr, dc) for dr in (-1, 0, 1) for dc in (-1, 0, 1) if (dr, dc) ≠ (0,

def neighbours(r, c):
    for dr, dc in DIRS4:
        nr, nc = r + dr, c + dc
        if 0 ⩽ nr < R and 0 ⩽ nc < C:
            yield nr, nc
```

The bounds check `0 ⩽ nr < R and 0 ⩽ nc < C` is the line that prevents both `IndexError` and the subtler bug of a negative index silently wrapping to the last row (`grid[-1]` is valid Python). A direction list makes the four-way and eight-way cases one loop rather than four or eight `if`s, and it is the shape flood fill, BFS on a grid and counting islands all use. Returning neighbours as a generator (`yield`, Module 11) keeps the caller's loop simple: `for nr, nc in neighbours(r, c):`.

Transposing and rotating

```
transposed = [list(col) for col in zip(*grid)]          # columns become rows
rotated_cw = [list(col) for col in zip(*grid[::-1])]    # rotate 90° clockwise
rotated_ccw = [list(col) for col in zip(*grid)][::-1]
flipped_h = [row[::-1] for row in grid]                 # mirror left-right
flipped_v = grid[::-1]                                  # mirror top-bottom (
```

`zip(*grid)` is the transpose: the star spreads the rows as separate arguments, and `zip` pairs up their first elements, then their second, and so on. Column sums are `[sum(col) for col in zip(*grid)]`. For a ragged grid `zip` truncates to the shortest row.

Printing

```
for row in grid:
    print(" ".join(map(str, row)))        # numbers separated by spaces

for row in maze:
    print("".join(row))                   # characters, no separator

print("\n".join(" ".join(map(str, row)) for row in grid))    # one print
```

`print(row)` shows the list's `repr` with brackets and is almost never the expected format. Aligned columns use a format spec per cell: `" ".join(f"{v:3}" for v in row)`.

Flattening and the index trick

A grid is also one list of `R * C` cells: cell `(r, c)` is index `r * C + c`, and index `i` is `divmod(i, C)`. Flattening (`[cell for row in grid for cell in row]`) lets you sort every cell, find the k-th smallest, or use `bisect`; the index arithmetic lets you store a grid in a flat list or a `bytearray` when memory matters. Row-major order — all of row 0, then all of row 1 — is also the order the grid is read from input and printed to output, and the order in which walking with `r` outer and `c` inner visits cells.

Copying a grid

`grid.copy()` copies the outer list only; the rows are shared. `[row[:] for row in grid]` copies every row (a full copy for a grid of immutables); `copy.deepcopy(grid)` for anything

nested deeper. A function that mutates a grid it was given should say so; a simulation that needs "the grid at the previous step" needs a real copy, not an alias.

Pitfalls

`[[0] * C] * R`.

`grid[c][r]` — row first, always.

A missing bounds check, and negative indexes wrapping instead of failing.

`print(row)`.

Modifying the grid while walking it and reading cells already changed this step — use a copy or a second grid.

Comparing a row string with a row list (`"..." ≠ [".", ".", "."]`).

Key takeaways

A grid is a list of row lists, `grid[r][c]`, row-major; read rows with a comprehension over `range(R)`.

Build with `[[0] * C for _ in range(R)]`; copy rows with `[row[:] for row in grid]`.

Walk with nested `range` or `enumerate`; neighbours through a direction list and the bounds check `0 ⩽ nr < R and 0 ⩽ nc < C`.

`zip(*grid)` transposes; `zip(*grid[::-1])` rotates; `row[::-1]` mirrors.

Print with `" ".join(map(str, row))`; flatten with a double comprehension; `(r, c)` ↔ `r * C + c`.

EXERCISE 1

One step of Life

Read an `R × C` grid of `#` (live) and `.` (dead) cells and print the next generation of Conway's Game of Life: a live cell with 2 or 3 live neighbours (of its 8) stays alive, a dead cell with exactly 3 becomes alive, every other cell is dead. Use a direction list, a bounds check, and a **new** grid — never write into the grid you are reading.

Input: `R C`, then `R` rows. **Output:** `R` rows.

```
3 3
.#.
.#.
.#.
```

prints

```
...
###
...
```

INPUT 1

```
3 3
.#.
.#.
.#.
```

OUTPUT 1

```
...
###
...
```

REFERENCE SOLUTION

```
R, C = map(int, input().split())
grid = [input().rstrip("\n") for _ in range(R)]
DIRS = [(dr, dc) for dr in (-1, 0, 1) for dc in (-1, 0, 1) if (dr, dc) != (0, 0)]


def live_neighbours(r, c):
    count = 0
    for dr, dc in DIRS:
        nr, nc = r + dr, c + dc
        if 0 <= nr < R and 0 <= nc < C and grid[nr][nc] == "#":
            count += 1
    return count


nxt = [["."] * C for _ in range(R)]
for r in range(R):
    for c in range(C):
        n = live_neighbours(r, c)
        if grid[r][c] == "#" and n in (2, 3):
            nxt[r][c] = "#"
        elif grid[r][c] == "." and n == 3:
            nxt[r][c] = "#"
for row in nxt:
    print("".join(row))
```

EXERCISE 2

Transpose and rotate

Read an `R × C` grid of integers and print its transpose, then the grid rotated 90° clockwise, then the column sums — each built with `zip(*...)`. Separate the three blocks with a line `--`.

Input: `R C`, then `R` rows. **Output:** the transpose (`C` rows), `--`, the rotation (`C` rows), `--`, one line of `C` column sums.

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

prints

```
1 4
2 5
3 6
--
4 1
5 2
6 3
--
5 7 9
```

INPUT 1

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

OUTPUT 1

```
1 4
2 5
3 6
--
4 1
5 2
6 3
--
5 7 9
```

REFERENCE SOLUTION

```
R, C = map(int, input().split())
grid = [list(map(int, input().split())) for _ in range(R)]


def show(rows):
    for row in rows:
        print(" ".join(map(str, row)))


show(zip(*grid))
print("--")
show(zip(*grid[::-1]))
print("--")
print(" ".join(str(sum(col)) for col in zip(*grid)))
```

QUIZ

1. What is the right way to build a 3×4 grid of zeros?

A. `[[0] * 4] * 3`

B. `[[0] * 4 for _ in range(3)]`

C. `[0] * 12`

D. `[[0, 0, 0]] * 4`

2. What does `list(zip(*[[1, 2], [3, 4]]))` produce?

A. `[(1, 3), (2, 4)]`

B. `[(1, 2), (3, 4)]`

C. `[[1, 3], [2, 4]]`

D. `[(1, 2, 3, 4)]`

3. Why must a neighbour check test `0 ⩽ nr` and not just `nr < R`?

A. Because `grid[-1]` raises `IndexError`

B. Because a negative index silently wraps to the last row instead of failing

C. It need not — negative rows are impossible

D. Because `R` may be zero

4. Which prints a row of integers as `1 2 3`?

A. `print(row)`

B. `print(" ".join(map(str, row)))`

C. `print(" ".join(row))`

D. `print(*row, sep="")`

5. In a row-major flat list of an `R × C` grid, which index holds cell `(r, c)`?

A. `r * C + c`

B. `c * R + r`

C. `r + c`

D. `r * R + c`

ANSWER KEY · GRIDS AND NESTED LISTS

1. B The comprehension creates a fresh row each time; `* 3` would repeat one shared row.

2. A The star passes each row as an argument; `zip` pairs first elements, then second — the transpose, as tuples.

3. B `grid[-1][c]` is valid Python and returns the last row, so a missing lower-bound check produces wrong answers rather than an error.

4. B `join` needs strings, hence `map(str, …)`; `print(row)` shows brackets; `sep=""` would give `123`.

5. A Each row contributes `C` cells, so `r` full rows precede the cell, then `c` more.

LESSON 6 · 13 MIN

The sequence tools — enumerate, zip, reversed, any, all, deque and the protocol

The built-ins that work on *every* sequence are a small set worth knowing cold: `enumerate`, `zip`, `reversed`, `sorted`, `min` / `max` / `sum`, `any` / `all`, `len`, `in`, slicing. Most have appeared already; this lesson collects them, adds `range` as a first-class sequence, introduces `collections.deque` for the two-ended operations a list does badly, mentions `array` for compact numeric storage, and closes with the *sequence protocol* — the two methods a class implements to be treated as a sequence by all of the above.

The tools

Tool	What it gives	Note
`len(s)`	element count	O(1)
`x in s`	membership	O(n) for list/tuple/str; O(1) for set/dict
`s[i]`, `s[a:b:c]`	element, slice	slicing copies
`enumerate(s, start=0)`	`(index, element)` pairs	lazy
`zip(a, b, strict=False)`	tuples of parallel elements	stops at the shortest; `strict=True` raises on mismatch
`reversed(s)`	elements back to front	an iterator, not a list; `s[::-1]` for a list
`sorted(s, key=, reverse=)`	a new sorted list	works on any iterable
`min(s)`, `max(s)`, `sum(s)`	extremes and total	`default=` for empty; `key=` on min/max; `sum(s, start)`
`any(s)`, `all(s)`	is any / every element truthy	short-circuit; `all([])` is `True`
`list(s)`, `tuple(s)`, `set(s)`	conversions	from any iterable

```
any(x < 0 for x in xs)                    # "has a negative"
all(a ⩽ b for a, b in zip(xs, xs[1:]))    # "is sorted"
max(range(10), key=lambda i: -abs(i - 4)) # argmax-style: the i closest to 4
sum(len(w) for w in words)                # total length
list(zip(*pairs))                         # unzip: [(a1, a2, …), (b1, b2, …)]
```

`zip(xs, xs[1:])` pairs each element with its successor — the "consecutive pairs" idiom, also available as `itertools.pairwise(xs)` (3.10).

---

range as a sequence

`range` is not just for loops. It is an immutable sequence: `len(range(10))` is 10, `range(10)[3]` is 3, `range(0, 100, 5)[::-1]` is another range, `50 in range(0, 100, 5)` is answered

arithmetically, and `list(range(5))` materialises it. A range holds three integers however large its extent — `range(10 ** 12)` costs nothing until iterated.

---

deque: a double-ended queue

A list is fast at its right end and slow at its left (`insert(0, x)` and `pop(0)` shift every element). `collections.deque` is fast at both ends:

```
from collections import deque

q = deque([1, 2, 3])
q.append(4)          # right end, O(1)
q.appendleft(0)      # left end, O(1)
q.pop()              # 4
q.popleft()          # 0
q.rotate(1)          # [3, 1, 2] — right rotation; negative rotates left
q[0], q[-1]          # indexing works; the middle is O(n)
window = deque(maxlen=3)      # a bounded deque drops the oldest when full
```

Two uses dominate. A **queue** for breadth-first search: `append` to enqueue, `popleft` to dequeue, `while q:` to drain. A **sliding window** with `maxlen`: append each new value and the deque holds the last k automatically. A deque is not a list — no slicing, and indexing the middle is linear — so convert with `list(q)` when you need those.

---

array: compact homogeneous storage

`array.array("i", [1, 2, 3])` stores C ints in a contiguous buffer, one machine word each, instead of a list of Python objects (about eight times smaller for ints). It supports the sequence operations and is what to reach for when millions of numbers must be held and memory matters; for numeric *computation* NumPy is the tool (Module 19). `bytearray` is the mutable byte sequence and `bytes` the immutable one (Module 14).

The sequence protocol

The tools above work on strings, lists, tuples, ranges and deques because those types implement a protocol, and a class of yours can implement it too:

```python
class Countdown:
    def __init__(self, n):
        self.n = n

    def __len__(self):
        return self.n

    def __getitem__(self, i):
        if i < 0:
            i += self.n
        if not 0 <= i < self.n:
            raise IndexError(i)
        return self.n - i

c = Countdown(3)
len(c)              # 3
c[0], c[-1]         # 3, 1
list(c)             # [3, 2, 1] — iteration falls back on __getitem__ from 0 u
2 in c              # True — membership falls back on iteration
list(reversed(c))   # [1, 2, 3] — reversed uses __len__ and __getitem__
```

`__len__` and `__getitem__` (raising `IndexError` at the end) are enough for `len`, indexing, iteration, `in`, `reversed`, `enumerate`, `zip` and `sorted` to work. `__contains__`, `__iter__` and `__reversed__` can be added for efficiency or a different behaviour; `collections.abc.Sequence` (Module 9) fills in `index` and `count` from the two basic methods. Slicing support means handling a `slice` object in `__getitem__` (`if isinstance(i, slice)`). Module 8 covers the dunder methods in full.

Pitfalls

- `reversed(xs)` used twice — it is an iterator, exhausted after one pass; `xs[::-1]` is a list.
- `max([])` — `ValueError`; pass `default=`.
- `zip` silently truncating when the inputs should match — `strict=True`.
- `list.pop(0)` in a BFS — a `deque`.
- Slicing a `deque` — convert to a list first.
- A `__getitem__` that never raises `IndexError`, making `list(obj)` an infinite loop.

Key takeaways

- `enumerate`, `zip`, `reversed`, `sorted`, `min` / `max` / `sum`, `any` / `all` work on every sequence and most iterables; the generator argument form is the idiom.
- `range` is a lazy immutable sequence — index it, slice it, test membership arithmetically.
- `deque` gives O(1) `append` / `pop` at both ends; use it for queues and `maxlen` windows, not for slicing or middle access.
- `array` stores homogeneous numbers compactly; NumPy is the computation tool.
- Implement `__len__` and `__getitem__` (raising `IndexError`) and the whole toolset works on your class.

EXERCISE 1

Sliding window maxima

Read a window size `k` and a line of integers. Slide a `collections.deque(maxlen=k)` over the values; for every position where the window is full, print its maximum and its mean (two decimals). If there are fewer than `k` values print `no full window`.

Input: `k`, then a line of integers. **Output:** `max <m> mean <a>` per full window, or `no full window`.

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

prints

```
max 3 mean 2.00
max 5 mean 3.33
max 5 mean 3.67
```

INPUT 1

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

OUTPUT 1

```
max 3 mean 2.00
max 5 mean 3.33
max 5 mean 3.67
```

REFERENCE SOLUTION

```python
from collections import deque

k = int(input())
xs = list(map(int, input().split()))
window = deque(maxlen=k)
full = 0
for x in xs:
    window.append(x)
    if len(window) == k:
        full += 1
        print(f"max {max(window)} mean {sum(window) / k:.2f}")
if not full:
    print("no full window")
```

EXERCISE 2

A sequence of your own

Implement `Squares(n)`, the sequence of the first `n` squares `0, 1, 4, …`, with only `__len__` and `__getitem__` (supporting negative indexes and raising `IndexError` past the ends). Then let the built-ins do the rest: print its length, its first and last elements, the whole sequence via `list`, whether a query value is `in` it, and it reversed via `reversed`.

Input: `n q`. **Output:** `len <n>`, `first <v> last <v>`, `all <values>`, `<q> in: <True/False>`, `reversed <values>`.

```
4 9
```

prints

```
len 4
first 0 last 9
all 0 1 4 9
9 in: True
reversed 9 4 1 0
```

INPUT 1

```
4 9
```

OUTPUT 1

```
len 4
first 0 last 9
all 0 1 4 9
9 in: True
reversed 9 4 1 0
```

REFERENCE SOLUTION

```
class Squares:
    def __init__(self, n):
        self.n = n

    def __len__(self):
        return self.n

    def __getitem__(self, i):
        if i < 0:
            i += self.n
        if not 0 <= i < self.n:
            raise IndexError(i)
        return i * i


n, q = map(int, input().split())
s = Squares(n)
print(f"len {len(s)}")
print(f"first {s[0]} last {s[-1]}")
print("all", " ".join(map(str, list(s))))
print(f"{q} in: {q in s}")
print("reversed", " ".join(map(str, reversed(s))))
```

QUIZ

1. What does `all([])` return?

A. `False`

B. `True`

C. `None`

D. `ValueError`

2. Which structure gives O(1) removal from the front?

A. `list`

B. `collections.deque`

C. `tuple`

D. `array.array`

3. What does `list(zip(xs, xs[1:]))` produce for `xs = [1, 2, 3]`?

A. `[(1, 2), (2, 3)]`

B. `[(1, 2), (2, 3), (3, None)]`

C. `[(1, 1), (2, 2), (3, 3)]`

D. `[(2, 1), (3, 2)]`

4. What happens after `r = reversed(xs); list(r); list(r)`?

A. Both lists are the reversed `xs`

B. The second list is empty — `reversed` returns a one-shot iterator

C. `TypeError`

D. The second list is `xs` in original order

5. A class defines only `__len__` and `__getitem__` (raising `IndexError` past the end). Which of these work on its instances?

A. Only `len` and indexing

B. `len`, indexing, `for`, `in`, `list()`, `reversed()`

C. Nothing until `__iter__` is defined

D. Only `for` loops

ANSWER KEY · THE SEQUENCE TOOLS — ENUMERATE, ZIP, REVERSED, ANY, ALL, DEQUE AND THE PROTOCOL

1. B Vacuous truth: no element is falsy. `any([])` is `False`. Check emptiness separately when it matters.

2. B `deque.popleft()` is constant time; `list.pop(0)` shifts every remaining element.

3. A Each element is paired with its successor; `zip` stops at the shorter slice. `itertools.pairwise` does the same.

4. B `reversed` gives an iterator that is exhausted after one pass; `xs[::-1]` gives a reusable list.

5. B Iteration falls back on indexing from 0 until `IndexError`; membership falls back on iteration; `reversed` uses both methods.

CHECKPOINT · 25 MIN

Checkpoint — Lists, tuples and sequences

This checkpoint covers the whole module: lists and their method costs, aliasing and shallow versus deep copies, comprehensions in list, set, dict and generator form, sorting with keys and stability and `bisect`, tuples and unpacking with `namedtuple`, grids with neighbours, transposes and the multiplication trap, and the sequence tools with `deque` and the sequence protocol.

How it works. Twelve questions and three programs. You need 70% on the questions and every program accepted to clear the module. You can retake it as often as you like; your best score counts.

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

- Which list operations are O(1) and which are O(n), and what replaces `pop(0)`?
- After `b = a.copy()` for a list of lists, what is shared?
- What is the difference between `[x for x in xs if c]` and `[x if c else y for x in xs]`?
- Why does `rows.sort(key=name); rows.sort(key=score, reverse=True)` give score-then-name order?
- What does `first, *rest = [1]` bind?
- Why is `[[0] * C] * R` wrong, and what is right?
- Which two methods make a class usable with `len`, indexing, iteration and `in`?

The three programs are an inventory that is edited, sorted by several keys and reported in aligned columns, a grid simulation that counts live neighbours with a direction list and a bounds check, and a sliding-window statistics tool built on a bounded `deque`.

EXERCISE 1

Inventory report

Keep an inventory from commands read until `report`: `add name qty price` adds an item (or replaces an existing one), `remove name` deletes it (`unknown item` if absent), `restock name qty` adds to its quantity (`unknown item` if absent). On `report` print the items ordered by quantity descending then name ascending, as a table: name left-aligned in 10, quantity right-aligned in 5,

price right-aligned in 8 with two decimals, value (`qty * price`) right-aligned in 10 with two decimals; then `TOTAL` with the total value in the last column.

Input: commands. **Output:** the error lines as they occur, then the table.

```
add bolt 100 0.05
add nut 250 0.02
restock bolt 50
remove screw
report
```

prints

```
unknown item
nut         250    0.02      5.00
bolt        150    0.05      7.50
TOTAL                       12.50
```

INPUT 1

```
add bolt 100 0.05
add nut 250 0.02
restock bolt 50
remove screw
report
```

OUTPUT 1

```
unknown item
nut         250    0.02      5.00
bolt        150    0.05      7.50
TOTAL                       12.50
```

REFERENCE SOLUTION

```python
import sys

items = {}  # name → [qty, price]
for line in sys.stdin:
    cmd, *args = line.split()
    if cmd == "report":
        break
    if cmd == "add":
        items[args[0]] = [int(args[1]), float(args[2])]
    elif cmd == "remove":
        if args[0] in items:
            del items[args[0]]
        else:
            print("unknown item")
    elif cmd == "restock":
        if args[0] in items:
            items[args[0]][0] += int(args[1])
        else:
            print("unknown item")

rows = sorted(((name, qty, price) for name, (qty, price) in items.items()), key=lamb
total = 0.0
for name, qty, price in rows:
    value = qty * price
    total += value
    print(f"{name:<10}{qty:>5}{price:>8.2f}{value:>10.2f}")
print(f"{'TOTAL':<10}{'':>5}{'':>8}{total:>10.2f}")
```

EXERCISE 2

Islands

Count the islands in an `R × C` grid of `#` (land) and `.` (water), where land cells are connected horizontally or vertically. Use an explicit stack (a list with `append` / `pop`) to flood each island from its first unvisited cell, with a direction list and a bounds check, and a `visited` grid built with a comprehension. Print the number of islands and the size of the largest.

Input: `R C`, then `R` rows. **Output:** `islands <n>` then `largest <size>` (0 when there is no land).

```
3 4
##..
#..#
..##
```

prints

```
islands 2
largest 3
```

INPUT 1

```
3 4
##..
#..#
..##
```

OUTPUT 1

```
islands 2
largest 3
```

REFERENCE SOLUTION

```python
R, C = map(int, input().split())
grid = [input().rstrip("\n") for _ in range(R)]
visited = [[False] * C for _ in range(R)]
DIRS = [(-1, 0), (1, 0), (0, -1), (0, 1)]


def flood(r, c):
    stack = [(r, c)]
    visited[r][c] = True
    size = 0
    while stack:
        cr, cc = stack.pop()
        size += 1
        for dr, dc in DIRS:
            nr, nc = cr + dr, cc + dc
            if 0 <= nr < R and 0 <= nc < C and grid[nr][nc] == "#" and not visited[r
                visited[nr][nc] = True
                stack.append((nr, nc))
    return size


islands = 0
largest = 0
for r in range(R):
    for c in range(C):
        if grid[r][c] == "#" and not visited[r][c]:
            islands += 1
            largest = max(largest, flood(r, c))
print(f"islands {islands}")
print(f"largest {largest}")
```

EXERCISE 3

Live leaderboard

Scores arrive one per line as `name score`. Keep a list of `(-score, name)` tuples sorted at all times with `bisect.insort`, and after every arrival print the current top three names, comma-separated (fewer if there are fewer entries). A name may appear again with a new score: remove its old entry first.

Input: lines `name score`. **Output:** one line per input line.

```
ada 50
bob 70
cy 60
ada 80
```

prints

```
ada
bob,ada
bob,cy,ada
ada,bob,cy
```

INPUT 1

```
ada 50
bob 70
cy 60
ada 80
```

OUTPUT 1

```
ada
bob,ada
bob,cy,ada
ada,bob,cy
```

REFERENCE SOLUTION

```python
import bisect
import sys

board = []       # sorted list of (-score, name)
current = {}     # name -> score
for line in sys.stdin:
    name, score = line.split()
    score = int(score)
    if name in current:
        board.remove((-current[name], name))
    current[name] = score
    bisect.insort(board, (-score, name))
    print(",".join(name for _, name in board[:3]))
```

CHECKPOINT QUESTIONS

1. What does `xs[::-1]` return for a list, and does it modify `xs`?

A. A reversed copy; no

B. A reversed view; yes

C. It reverses in place and returns `None`

D. An iterator

2. What is `[x for x in range(6) if x % 2 for _ in range(2)]`?

A. `[1, 1, 3, 3, 5, 5]`

B. `[1, 3, 5]`

C. `SyntaxError`

D. `[0, 0, 2, 2, 4, 4]`

3. Which builds the dictionary `{'a': 1, 'b': 2}` from `keys = ['a', 'b']` and `vals = [1, 2]`?

A. `dict(zip(keys, vals))`

B. `{keys: vals}`

C. `dict(keys, vals)`

D. `zip(keys, vals)`

4. What is `sorted([(2, 'b'), (1, 'z'), (2, 'a')])`?

A. `[(1, 'z'), (2, 'a'), (2, 'b')]`

B. `[(1, 'z'), (2, 'b'), (2, 'a')]`

C. `[(2, 'a'), (2, 'b'), (1, 'z')]`

D. `TypeError`

5. What does `bisect.insort(xs, 4)` do to a sorted `xs = [1, 3, 5]`?

A. Returns `2` and leaves `xs` alone

B. Inserts 4 keeping order: `xs` becomes `[1, 3, 4, 5]`

C. Appends 4: `[1, 3, 5, 4]`

D. Raises because 4 is not present

6. What does `a, (b, c) = 1, (2, 3)` bind?

A. `a=1, b=2, c=3`

B. `a=1, b=(2, 3), c=None`

C. `ValueError`

D. `SyntaxError`

7. Why is `row = [0] * 3; grid = [row, row, row]` a bug?

A. It is not — the rows are copied

B. All three rows are the same list; changing one cell changes every row

C. Lists cannot contain lists

D. `* 3` is not allowed on lists

8. What is `list(zip(*[(1, 'a'), (2, 'b')]))`?

A. `[(1, 2), ('a', 'b')]`

B. `[(1, 'a'), (2, 'b')]`

C. `[1, 2, 'a', 'b']`

D. `[(1, 'a', 2, 'b')]`

9. Which deque method drops the oldest element automatically?

A. `popleft()` must be called by hand

B. None — but `deque(maxlen=k)` discards the opposite end on `append` once full

C. `rotate()`

D. `clear()`

10. What does `max(["apple", "fig", "banana"], key=len)` return?

A. `'banana'`

B. `'apple'`

C. `6`

D. `'fig'`

11. What does `t = (1, 2); t += (3,)` do?

A. Mutates the tuple in place

B. Creates a new tuple `(1, 2, 3)` and rebinds `t`

C. `TypeError`

D. Creates `(1, 2, (3,))`

12. A class defines `__getitem__` that never raises `IndexError`. What happens on `list(obj)`?

A. An empty list

B. An infinite loop — iteration by indexing stops only at `IndexError`

C. `TypeError`

D. A list of one element

ANSWER KEY · CHECKPOINT — LISTS, TUPLES AND SEQUENCES

1. A Slicing always creates a new list. `xs.reverse()` is the in-place version and returns `None`.

2. A The filter applies to the outer loop, and the inner `for` repeats each surviving value twice. Clauses read left to right.

3. A `zip` pairs the elements and `dict` consumes the pairs; `{k: v for k, v in zip(keys, vals)}` is the comprehension form.

4. A Tuples compare element by element, so ties on the number are broken by the string.

5. B `insort` finds the position by binary search and inserts there. `bisect_left` would only return the position.

6. A Unpacking targets nest; the inner tuple target unpacks the inner tuple value.

7. B The same object appears three times. `[[0] * 3 for _ in range(3)]` builds separate rows.

8. A Transposing a list of pairs 'unzips' it into the firsts and the seconds.

9. B A bounded deque keeps at most `maxlen` items; appending to a full one evicts from the other end.

10. A The key compares lengths; the element with the greatest length is returned, not the length itself.

11. B Tuples are immutable; `+=` falls back to `t = t + (3,)`, a fresh object.

12. B The fallback iteration protocol calls `obj[0]`, `obj[1]`, … until `IndexError`; without it, it never stops.