Module 7 of 20

Dictionaries and sets

Dictionaries with get, setdefault, views and merging, counting and grouping with Counter and defaultdict, sets and their algebra, the hashing contract that decides what can be a key, nested data and JSON, and the complexity table with heapq.

The dictionary is the structure Python is built from — namespaces, attributes and keyword arguments are all dicts — and the one a program reaches for whenever it needs to find something by name in constant time. This module teaches it thoroughly, then the two `collections` types that name the commonest dict jobs (`Counter` for counting, `defaultdict` for grouping), the set as the membership and algebra structure, and the hashing mechanism underneath both, which explains why a list cannot be a key, why string hashes are randomised, and what a class must define to be a key.

Dictionaries covers construction, `d[k]` versus `get` versus `in` versus `setdefault`, views, merging with `|`, and the rule against resizing during iteration. Counting and grouping gives `Counter` and `defaultdict` with the plain-dict spellings and the reason `itertools.groupby` is not a grouping tool. Sets covers the operations, `frozenset`, and the ordering caveat that decides how a set is printed. Hashing and keys explains the slot mechanism, the equality contract and `__eq__` / `__hash__`. Nested data and JSON covers `loads` / `dumps`, safe navigation, recursive walks and deterministic output with `sort_keys`. Choosing a collection gives the complexity table, the decision rules and `heapq`.

The exercises are whole programs: a phone book, a configuration merge that reports overrides, word frequencies with an explicit tie-break, grouping by department, set algebra over two lists, deduplication with and without order, points as tuple keys, a hashable value class, a recursive JSON walk, a canonical JSON printer, a priority queue on `heapq`, and top-k selection. The checkpoint adds a word-frequency report, a two-snapshot diff built on set algebra, and a scheduler that pops the highest-priority job from a heap of tuples.

CONTENTS

1	Dictionaries — the mapping at the centre of Python	14 min
2	Counting and grouping — Counter, defaultdict and the accumulation idioms	13 min
3	Sets — membership, deduplication and set algebra	13 min

4	Hashing and keys — what makes an object usable in a dict or set	13 min
5	Nested data and JSON	14 min
6	Choosing a collection — the complexity table and heapq	13 min
✓	Checkpoint — Dictionaries and sets	25 min

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

LESSON 1 · 14 MIN

Dictionaries — the mapping at the centre of Python

The dictionary is the data structure Python itself is built on — module namespaces, object attributes, keyword arguments are all dicts — and it is the one you will reach for most often: a mapping from keys to values with constant-time lookup, insertion and deletion, keeping the order in which keys were added. This lesson covers construction, the access methods and which one to use when a key may be missing, the views, iteration, updating and merging, dict comprehensions, and the rule against changing a dict's size while iterating it.

Building

```
empty = {}
ages = {"ada": 36, "grace": 45}
from_pairs = dict([("a", 1), ("b", 2)])
from_zip = dict(zip(names, scores))
from_kwargs = dict(x=1, y=2)                    # keys must be identifiers
defaults = dict.fromkeys(["a", "b"], 0)         # {'a': 0, 'b': 0} — one shared
squares = {n: n * n for n in range(4)}          # comprehension
```

Keys must be *hashable* — immutable, in practice: strings, numbers, tuples of hashables, frozensets (lesson 4). A list as a key is `TypeError: unhashable type: 'list'`. Values can be anything. Duplicate keys in a literal keep the last value.

Reading

```
ages["ada"]              # 36
ages["bob"]              # KeyError: 'bob'
ages.get("bob")          # None — no exception
ages.get("bob", 0)       # 0 — a default
"bob" in ages            # False — the membership test is on keys, O(1)
len(ages)                # 2
```

`d[k]` when the key *must* exist — its `KeyError` is the right failure for a bug. `d.get(k, default)` when absence is normal. `k in d` when only presence matters. The idiom `if k in`

`d: v = d[k]` does two lookups where `get` does one; the idiom `d.get(k) or default` is the truthiness trap from Module 2 (a stored `0` or `""` is replaced).

Writing

```
ages["bob"] = 30            # insert or overwrite
ages["ada"] += 1            # read, add, write back — KeyError if absent
del ages["bob"]             # KeyError if absent
ages.pop("bob")             # remove and return; KeyError if absent
ages.pop("bob", None)       # remove and return, or the default
ages.popitem()              # remove and return the *last* inserted (key, valu
ages.setdefault("cy", 0)    # insert 0 only if absent; return the value either
ages.update({"dee": 1}, ed=2)   # merge in pairs from a mapping / iterable / k
ages.clear()
```

`setdefault` is the one-line "insert if missing, then use": `d.setdefault(key, []).append(x)` groups values under a key without an `if`. The next lesson replaces that with `defaultdict`, which is clearer when every key gets the same kind of default.

Merging

```
merged = {**a, **b}         # 3.5: a new dict; b's values win on shared keys
merged = a | b              # 3.9: the same, as an operator
a |= b                      # update a in place
a.update(b)                 # the same as |=
```

Merges are left-to-right: later mappings overwrite earlier ones.

Views and iteration

```
for k in d:                    # keys, in insertion order
for k in d.keys():             # the same, explicit
for v in d.values():
for k, v in d.items():         # pairs — the usual loop

list(d)                        # the keys as a list
sorted(d)                      # keys sorted
sorted(d.items(), key=lambda kv: kv[1], reverse=True)   # pairs by value, desc
max(d, key=d.get)              # the key with the largest value
```

`keys()`, `values()` and `items()` return *views*: live windows onto the dict that reflect later changes, support `len` and `in`, and — for keys and items — behave like sets (`a.keys() & b.keys()` is the common keys). They are not lists: index them with `list(d.values())[0]`, or `next(iter(d))` for the first key. Since 3.7 insertion order is guaranteed by the language, so a dict doubles as an ordered record of arrival; `reversed(d)` iterates the keys backwards (3.8).

Changing size during iteration

```
for k in d:
    if bad(k):
        del d[k]               # RuntimeError: dictionary changed size during it
```

Iterate over a copy of the keys (`for k in list(d):`), or build a new dict with a comprehension (`{k: v for k, v in d.items() if not bad(k)}`). Changing a *value* during iteration (`d[k] = …` for an existing `k`) is fine.

Nested dictionaries

```
users = {"ada": {"age": 36, "langs": ["python"]}}
users["ada"]["langs"].append("c")
users.setdefault("bob", {})["age"] = 30
users.get("cy", {}).get("age")          # None — safe navigation with defaults
```

Two-level access is common enough that the pattern `d.get(k1, {}).get(k2)` is worth knowing, and `defaultdict(dict)` (next lesson) removes the `setdefault`. JSON documents arrive as nested dicts and lists (lesson 5).

Dicts as records and as tables

A dict with fixed, known keys (`{"name": …, "age": …}`) is a *record* — a lighter `dataclass` (Module 8), fine for JSON and quick scripts. A dict with keys that vary with the data (`counts[word]`) is a *table* — the lookup structure. Both are dicts, but the first is usually better as a class once it has behaviour, and the second is where `get` , `setdefault` and the counting idioms live.

Pitfalls

`d[k]` where `k` may be absent; `get` or `in` .

`d.get(k) or default` when the stored value can be falsy.

A list as a key.

`dict.fromkeys(keys, [])` — one shared list for every key.

Deleting from a dict while iterating it.

Assuming `d.keys()` is a list (`d.keys()[0]` is a `TypeError`).

Key takeaways

Dicts map hashable keys to any values with O(1) lookup and insertion, in insertion order.

`d[k]` raises `KeyError` , `d.get(k, default)` does not, `k in d` tests presence; `setdefault` inserts a default and returns it.

`pop` , `del` , `popitem` , `update` , `|` and `⊨` change or merge; later mappings win.

`keys()` / `values()` / `items()` are live views; iterate `items()` for pairs; sort with a key on `items()` .

Never change a dict's size while iterating it; iterate `list(d)` or build a new dict.

EXERCISE 1

Phone book

Keep a phone book from commands read until the end of input: `add name number` stores or replaces; `find name` prints the number or `not found` ; `remove name` prints `removed` or `not found` ; `list` prints every entry as `name: number` in name order. Use `get` for lookups, `pop` with a default for removal, and never raise `KeyError` .

Input: commands. **Output:** one line per `find`, `remove` and per entry of `list`.

```
add ada 100
add bob 200
find ada
find cy
remove bob
remove bob
list
```

prints

```
100
not found
removed
not found
ada: 100
```

INPUT 1

```
add ada 100
add bob 200
find ada
find cy
remove bob
remove bob
list
```

OUTPUT 1

```
100
not found
removed
not found
ada: 100
```

REFERENCE SOLUTION

```python
import sys

book = {}
for line in sys.stdin:
    cmd, *args = line.split()
    if cmd == "add":
        book[args[0]] = args[1]
    elif cmd == "find":
        print(book.get(args[0], "not found"))
    elif cmd == "remove":
        print("removed" if book.pop(args[0], None) is not None else "not found")
    elif cmd == "list":
        for name in sorted(book):
            print(f"{name}: {book[name]}")
```

EXERCISE 2

Merge with overrides

The first line holds default settings as `key=value` pairs, the second the user's settings in the same form (either may be empty). Merge them with the `|` operator so the user's values win, print the merged settings in the order the operator produces (defaults' order, then new user keys), and then the keys whose value the user **changed**, sorted.

Input: two lines. **Output:** `key=value` per merged entry, then `overridden: <keys>` or `overridden: none`.

```
theme=light size=12 lang=en
size=14 lang=en font=mono
```

prints

```
theme=light
size=14
lang=en
font=mono
overridden: size
```

INPUT 1

```
theme=light size=12 lang=en
size=14 lang=en font=mono
```

OUTPUT 1

```
theme=light
size=14
lang=en
font=mono
overridden: size
```

REFERENCE SOLUTION

```python
def parse(line):
    return dict(tok.split("=", 1) for tok in line.split())


defaults = parse(input())
user = parse(input())
merged = defaults | user
for key, value in merged.items():
    print(f"{key}={value}")
overridden = sorted(k for k in defaults.keys() & user.keys() if defaults[k] != user[
print("overridden:", " ".join(overridden) if overridden else "none")
```

QUIZ

1. What does `d = {}; d["a"] += 1` do?

A. Sets `d["a"]` to 1

B. Raises `KeyError`

C. Sets `d["a"]` to `None`

D. Raises `TypeError`

2. What is `d.setdefault("k", []).append(1)` equivalent to?

A. Replace `d["k"]` with `[1]`

B. If `"k"` is absent store `[]` under it; then append 1 to whatever list is stored

C. Append 1 only if `"k"` exists

D. Raise `KeyError` when `"k"` is absent

3. What does `{**a, **b}` do when `a` and `b` share a key?

A. Raises `KeyError`

B. Keeps `a`'s value

C. Keeps `b`'s value — later mappings win

D. Stores both in a list

4. What happens here?

```
for k in d:
    if k.startswith("_"):
        del d[k]
```

A. The keys are removed

B. `RuntimeError: dictionary changed size during iteration`

C. Only the first matching key is removed

D. `KeyError`

5. What does `d.keys()` return?

A. A list of keys

B. A live view that supports `len`, `in` and set operations, but not indexing

C. A tuple of keys

D. An iterator that can be consumed once

ANSWER KEY · DICTIONARIES — THE MAPPING AT THE CENTRE OF PYTHON

1. B Augmented assignment reads the key first, and it is missing. Use `d["a"] = d.get("a", 0) + 1`, a `defaultdict(int)` or a `Counter`.

2. B `setdefault` inserts the default only when the key is missing and returns the stored value either way, so the append always lands on the dict's list.

3. C Merges are left to right and later entries overwrite; `a | b` behaves the same way.

4. B Resizing a dict while iterating it is detected and refused. Iterate over `list(d)` or build a filtered copy with a comprehension.

5. B Views reflect later changes to the dict and behave like sets for keys and items; `list(d.keys())` materialises them when indexing is needed.

LESSON 2 · 13 MIN

Counting and grouping — Counter, defaultdict and the accumulation idioms

Half of all dictionary code does one of two things: counts how often each key occurs, or collects the items that share a key into a list. Both have a plain-dict spelling, a `setdefault` spelling, and a `collections` type that says exactly what is meant — `Counter` for counting, `defaultdict` for grouping. This lesson gives all three for each job, the `Counter` API (`most_common`, arithmetic, `elements`), `defaultdict` with different factories including nested ones, inverting a mapping, and the reason `itertools.groupby` is *not* the grouping tool it sounds like.

Counting

```
counts = {}
for word in words:
    if word in counts:                # plain dict, verbose
        counts[word] += 1
    else:
        counts[word] = 1

counts[word] = counts.get(word, 0) + 1          # plain dict, one line

from collections import Counter
counts = Counter(words)                         # the type that means "count"
```

`Counter` is a dict subclass whose missing keys read as `0` (never `KeyError`) and which is built from any iterable or mapping:

```
c = Counter("mississippi")
c["s"]                  # 4
c["z"]                  # 0 — no KeyError
c.most_common(2)        # [('i', 4), ('s', 4)] — ties keep first-seen order
c.total()               # 11 (3.10)
c.update("miss")        # add counts
c.subtract("ss")        # subtract counts (can go negative)
list(c.elements())      # each key repeated by its count
Counter(a) + Counter(b) # add counts; & is min, | is max; - drops non-positive
sorted(c.items())       # deterministic order for output
```

`most_common()` with no argument returns everything sorted by count descending, with ties in insertion order — which is the deterministic ordering to print when the tie-break does not matter; when it does, sort explicitly: `sorted(c.items(), key=lambda kv: (-kv[1], kv[0]))` for count descending then key ascending.

Grouping

```
groups = {}
for name, dept in staff:
    groups.setdefault(dept, []).append(name)     # plain dict

from collections import defaultdict
groups = defaultdict(list)
for name, dept in staff:
    groups[dept].append(name)                    # the type that means "group"
```

`defaultdict(factory)` calls `factory()` to create the value for a missing key *on first access* and stores it; afterwards it is an ordinary dict. The factory is any zero-argument callable: `list`, `set`, `int` (counting), `dict`, `lambda: [0, 0]`, or a class. Nested groupings compose: `defaultdict(lambda: defaultdict(int))` counts pairs as `table[a][b] += 1`.

Two behaviours to know. Reading a missing key *creates* it (`groups["x"]` inserts an empty list), so use `in` to test presence without side effects. And `defaultdict` prints as `defaultdict(<class 'list'>, {...})`; convert with `dict(groups)` for a clean display, and sort the keys for output.

Inverting and indexing

```
inverse = {v: k for k, v in d.items()}              # one-to-one only: later ke

by_value = defaultdict(list)                        # many-to-one: group keys b
for k, v in d.items():
    by_value[v].append(k)

index = defaultdict(list)                           # a word index: word → lin
for n, line in enumerate(lines, start=1):
    for word in set(line.split()):
        index[word].append(n)
```

The comprehension inverts a mapping whose values are unique; when several keys share a value, the grouping form keeps them all.

Accumulating other things

The same shape works for any per-key accumulation:

```
totals = defaultdict(float)
for name, amount in transactions:
    totals[name] += amount

best = {}
for name, score in results:
    best[name] = max(best.get(name, float("-inf")), score)

first_seen = {}
for i, key in enumerate(keys):
    first_seen.setdefault(key, i)               # keeps the first, ignores lat
```

Counting with `defaultdict(int)` and with `Counter` are equivalent; `Counter` adds the API and reads as intent.

Counting pairs and combinations

A key can be a tuple, so co-occurrence counts need no nesting: `Counter((a, b) for a, b in pairs)` tallies each ordered pair, `Counter(frozenset((a, b)) for a, b in pairs)` each unordered one, and `Counter(zip(words, words[1:]))` counts adjacent word pairs — bigrams — in one line. `most_common` then answers "which pair is commonest" directly.

itertools.groupby is not this

`itertools.groupby(iterable, key)` groups *consecutive* elements with equal keys — it is a run-length tool, and on unsorted input it produces one group per run, not per key. It groups whole data only after a `sorted(..., key=key)`, and even then yields iterators that must be consumed immediately. For "group all items by key", `defaultdict(list)` is the tool; `groupby` is for runs, or for already-sorted data streamed in one pass (Module 11).

Printing counts and groups

Output must be deterministic, and a dict's order is insertion order — fine when the input order is meaningful, but usually you want sorted keys or descending counts:

```
for key in sorted(groups):
    print(key, ", ".join(groups[key]))

for key, n in sorted(counts.items(), key=lambda kv: (-kv[1], kv[0])):
    print(f"{key}: {n}")
```

Never print a `set` or iterate one into output (lesson 3); a `Counter`'s `most_common` is fine because its order is defined.

Pitfalls

- `counts[k] += 1` on a plain dict for a new key — `KeyError`; use `get`, `defaultdict(int)` or `Counter`.
- Testing `if key in groups` after a `groups[key]` read — the read already created it.
- `{v: k …}` inverting a many-to-one mapping and losing keys.
- `groupby` on unsorted data.
- Printing a `defaultdict` or a `Counter` directly and getting the class name in the output.
- Relying on `most_common` tie order when the spec says ties break alphabetically.

Key takeaways

Count with `Counter(iterable)`; missing keys read as 0; `most_common`, `update`, `+` / `-` / `&` / `|`, `total`, `elements`.

Group with `defaultdict(list)`; any zero-argument factory works, including nested defaultdicts.

`get(k, 0) + 1` and `setdefault(k, []).append(x)` are the plain-dict spellings.

Invert with a comprehension when values are unique, with grouping when they are not.

`itertools.groupby` groups consecutive runs only; sort first or use `defaultdict`.

EXERCISE 1

Word frequencies

Read text until the end of input and count words (case-folded, with leading and trailing punctuation stripped; empty results ignored) with a `Counter`. Read `k` from the first line. Print the `k` most frequent words as `word count`, ordering ties **alphabetically** (so sort explicitly rather than relying on `most_common`), then the number of words that occur exactly once.

Input: `k`, then lines of text. **Output:** up to `k` lines, then `once: <n>`.

```
2
the cat and the hat. The end, and... fin
```

prints

```
the 3
and 2
once: 4
```

INPUT 1

```
2
the cat and the hat. The end, and... fin
```

OUTPUT 1

```
the 3
and 2
once: 4
```

REFERENCE SOLUTION

```python
import string
import sys
from collections import Counter

k = int(input())
counts = Counter()
for line in sys.stdin:
    words = (w.casefold().strip(string.punctuation) for w in line.split())
    counts.update(w for w in words if w)
ranked = sorted(counts.items(), key=lambda kv: (-kv[1], kv[0]))
for word, n in ranked[:k]:
    print(word, n)
print(f"once: {sum(1 for n in counts.values() if n == 1)}")
```

EXERCISE 2

Group by department

Read lines `name dept` until the end of input and group the names by department with a `defaultdict(list)`. Print each department in alphabetical order with its names sorted and comma-separated, then the largest department (ties broken alphabetically).

Input: lines. **Output:** `<dept>: <names>` per department, then `largest: <dept>` (or `largest: none`).

```
ada eng
bob ops
cy eng
```

prints

```
eng: ada,cy
ops: bob
largest: eng
```

INPUT 1

```
ada eng
bob ops
cy eng
```

OUTPUT 1

```
eng: ada,cy
ops: bob
largest: eng
```

REFERENCE SOLUTION

```python
import sys
from collections import defaultdict

groups = defaultdict(list)
for line in sys.stdin:
    name, dept = line.split()
    groups[dept].append(name)
for dept in sorted(groups):
    print(f"{dept}: {','.join(sorted(groups[dept]))}")
if groups:
    print("largest:", min(groups, key=lambda d: (-len(groups[d]), d)))
else:
    print("largest: none")
```

QUIZ

1. What is `Counter("aab")["z"]`?

A. `KeyError`

B. `0`

C. `None`

D. `-1`

2. What does `groups = defaultdict(list); print("x" in groups); groups["x"]; print("x" in groups)` print?

A. `False` then `False`

B. `False` then `True`

C. `True` then `True`

D. `KeyError`

3. How does `Counter.most_common()` order ties?

A. Alphabetically

B. In the order the keys were first inserted

C. Randomly

D. By key hash

4. What does `itertools.groupby("aabab")` produce?

A. Two groups: all the `a`s and all the `b`s

B. Four groups of consecutive runs: `aa`, `b`, `a`, `b`

C. A dict

D. `TypeError`

5. Which builds a nested counter usable as `t[a][b] += 1` without `KeyError`?

A. `defaultdict(int)`

B. `defaultdict(lambda: defaultdict(int))`

C. `defaultdict(dict)`

D. `Counter(Counter)`

ANSWER KEY · COUNTING AND GROUPING — COUNTER, DEFAULTDICT AND THE ACCUMULATION IDIOMS

1. B A `Counter` returns 0 for a missing key rather than raising, which is what makes `c[x] += 1` work unconditionally.

2. B Reading a missing key through `[]` calls the factory and *stores* the result, so the key now exists. `in` never inserts.

3. B The sort is stable over insertion order. For an alphabetical tie-break, sort `items()` by `(-count, key)` yourself.

4. B `groupby` groups consecutive equal keys only. For whole-data grouping, sort first or use `defaultdict(list)`.

5. B The outer factory must return an inner `defaultdict(int)`; `defaultdict(dict)` would give a plain dict whose missing keys raise.

LESSON 3 · 13 MIN

Sets — membership, deduplication and set algebra

A set is an unordered collection of distinct hashable objects with constant-time membership. That single property — `x in s` is O(1) regardless of size — turns quadratic "is this element in that list" loops into linear ones, and the set operations (union, intersection, difference) express "in both", "in either", "in one but not the other" without loops at all. This lesson covers construction, the mutating and non-mutating operations, `frozenset`, the ordering caveat that decides how a set is printed, and the recipes: deduplicate (with and without order), find duplicates, compare two collections.

Building

```
s = {1, 2, 3}
empty = set()               # {} is an empty *dict*
from_list = set([1, 2, 2, 3])          # {1, 2, 3} — duplicates collapse
from_str = set("hello")                # {'h', 'e', 'l', 'o'}
evens = {x for x in range(10) if x % 2 == 0}   # set comprehension
```

Elements must be hashable: numbers, strings, tuples of hashables, frozensets — not lists, dicts or sets. `{[1, 2]}` is `TypeError: unhashable type: 'list'`; a tuple `(1, 2)` works, so pairs, coordinates and visited-cell records are tuples in a set.

Membership and size

```
3 in s              # True — O(1) average
len(s)              # 3
```

The classic upgrade is a loop that tests `x in some_list` for many `x`: converting the list to a set once makes every test constant time. `seen = set()` with `if x in seen: … seen.add(x)` is the visited-set idiom of every graph search.

Changing a set

```
s.add(4)            # insert (no effect if present)
s.remove(4)         # KeyError if absent
s.discard(4)        # no error if absent
s.pop()             # remove and return an arbitrary element
s.clear()
s.update([5, 6])    # add many
```

`discard` is the one to use when absence is normal; `remove` when absence is a bug.

Set algebra

Operation	Operator	Method	Meaning
union	`a | b`	`a.union(b)`	in either
intersection	`a & b`	`a.intersection(b)`	in both
difference	`a - b`	`a.difference(b)`	in `a` but not `b`
symmetric difference	`a ^ b`	`a.symmetric_difference(b)`	in exactly one
subset	`a ⩽ b`, `a < b`	`a.issubset(b)`	every element of `a` is in `b` (strict with `<`)
superset	`a ⩾ b`, `a > b`	`a.issuperset(b)`	
disjoint	—	`a.isdisjoint(b)`	no common element

```
a = {1, 2, 3}
b = {3, 4}
a | b          # {1, 2, 3, 4}
a & b          # {3}
a - b          # {1, 2}
a ^ b          # {1, 2, 4}
a ⊨ b          # update a in place; &=, -=, ^= likewise
```

The operators require both sides to be sets; the methods accept any iterable (`a.union([4, 5])`). The in-place forms (`|=`, `intersection_update`, …) modify the left set. `a - b` is the "what is missing" question, `a & b` the "what do they share" question, `a ^ b` the "what changed" question between two snapshots.

---

frozenset

An immutable set — hashable, so usable as a dict key or a member of another set:

```
fs = frozenset({1, 2})
groups = {frozenset({"a", "b"}): "pair"}      # a key that is a set of names
seen_states = {frozenset(state) for state in states}
```

It supports every non-mutating operation. Use it whenever a set must go into another set or serve as a key; a plain set there is a `TypeError`.

Order: the caveat that matters for output

A set has no order, and the order it iterates in depends on the elements' hashes, which for strings are **randomised per process** (`PYTHONHASHSEED`). `print({"b", "a"})` may show `{'a', 'b'}` on one run and `{'b', 'a'}` on the next. Two rules follow:

> **Never print a set, and never build output by iterating one.** Sort it: `sorted(s)` gives a list in a defined order, and `" ".join(sorted(s))` is the line to print.
>
> **Never depend on which element `s.pop()` or `next(iter(s))` returns**, or on the order of `list(s)`.

Small integers happen to iterate in ascending order in CPython, which is an implementation accident, not a guarantee; sort those too.

Recipes

```
unique_count = len(set(xs))                          # how many distinct
unique_ordered = list(dict.fromkeys(xs))             # deduplicate, keeping f
duplicates = {x for x in xs if xs.count(x) > 1}      # O(n²) — fine for small
seen, dups = set(), set()                            # O(n)
for x in xs:
    if x in seen:
        dups.add(x)
    seen.add(x)
missing = set(range(1, n + 1)) - set(xs)             # which of 1..n are abse
common = set(a) & set(b)                             # shared elements of two
same_elements = set(a) == set(b)                     # ignoring order and repe
```

`dict.fromkeys(xs)` is the ordered deduplication: dict keys are unique *and* keep insertion order, so the keys of that dict are the distinct elements in first-seen order — the one thing a set cannot give you.

Pitfalls

`{}` for an empty set.

A list or a set as an element; use a tuple or `frozenset`.

Printing a set, or joining its elements without sorting.

`remove` on a possibly-absent element; `discard`.

`a | [1, 2]` — the operator needs two sets; use `a.union([1, 2])`.

Expecting `set(xs)` to preserve order; `dict.fromkeys(xs)`.

Key takeaways

A set holds distinct hashable elements with O(1) membership; build with `{…}`, `set(iterable)` or a comprehension; `set()` is the empty one.

`add` / `discard` / `remove` / `update`; `|`, `&`, `-`, `^` and their in-place forms; `⩽` / `⩾` for subsets; methods accept any iterable.

`frozenset` is the immutable, hashable set for keys and nested sets.

Iteration order is arbitrary and randomised for strings — sort before printing, never depend on it.

Deduplicate with `set` (unordered) or `dict.fromkeys` (ordered); find missing and common elements with `-` and `&`.

EXERCISE 1

Set algebra

Read two lines of tokens `A` and `B` and print, each sorted and space-separated (or `(none)`): the union, the intersection, `A - B`, `B - A` and the symmetric difference; then `A ⩽ B: True/False` and `disjoint: True/False`.

Input: two lines. **Output:** seven lines with the labels shown.

```
a b c
b c d
```

prints

```
union: a b c d
intersection: b c
A-B: a
B-A: d
symmetric: a d
A ⩽ B: False
disjoint: False
```

INPUT 1

```
a b c
b c d
```

OUTPUT 1

```
union: a b c d
intersection: b c
A-B: a
B-A: d
symmetric: a d
A ⩽ B: False
disjoint: False
```

REFERENCE SOLUTION

```
A = set(input().split())
B = set(input().split())


def show(label, items):
    print(f"{label}: {' '.join(sorted(items)) if items else '(none)'}")


show("union", A | B)
show("intersection", A & B)
show("A-B", A - B)
show("B-A", B - A)
show("symmetric", A ^ B)
print(f"A ⩽ B: {A ⩽ B}")
print(f"disjoint: {A.isdisjoint(B)}")
```

EXERCISE 2

Deduplicate three ways

Read a line of positive integers. Print the number of distinct values; the distinct values in **first-seen order** (via `dict.fromkeys`); the values that occur more than once, sorted; and the values from 1 to the maximum that are missing, sorted (via set difference).

Input: one line of positive integers (at least one). **Output:** `distinct: <n>`, `ordered: <values>`, `dups: <values>` (or `(none)`), `missing: <values>` (or `(none)`).

```
3 1 3 5 1
```

prints

```
distinct: 3
ordered: 3 1 5
dups: 1 3
missing: 2 4
```

INPUT 1

```
3 1 3 5 1
```

OUTPUT 1

```
distinct: 3
ordered: 3 1 5
dups: 1 3
missing: 2 4
```

REFERENCE SOLUTION

```
xs = list(map(int, input().split()))


def show(label, items):
    print(f"{label}: {' '.join(map(str, items)) if items else '(none)'}")


print(f"distinct: {len(set(xs))}")
show("ordered", list(dict.fromkeys(xs)))
seen, dups = set(), set()
for x in xs:
    if x in seen:
        dups.add(x)
    seen.add(x)
show("dups", sorted(dups))
show("missing", sorted(set(range(1, max(xs) + 1)) - set(xs)))
```

QUIZ

1. What is `{}`?

A. An empty set

B. An empty dict

C. An empty list

D. A syntax error

2. Why does `{[1, 2], 3}` fail?

A. Sets cannot mix types

B. A list is unhashable and cannot be a set element

C. Sets need at least three elements

D. Lists must be converted with `list()` first

3. What is wrong with `print(set(words))` in a judged program?

A. Nothing

B. Set iteration order is arbitrary and randomised for strings, so the output differs between runs — print `sorted(...)`

C. It prints the class name

D. It raises `TypeError`

4. Which removes `x` from a set without raising when `x` is absent?

A. `s.remove(x)`

B. `s.discard(x)`

C. `s.pop(x)`

D. `del s[x]`

5. Which expression deduplicates `xs` while keeping first-seen order?

A. `list(set(xs))`

B. `list(dict.fromkeys(xs))`

C. `sorted(set(xs))`

D. `set(xs)`

ANSWER KEY · SETS — MEMBERSHIP, DEDUPLICATION AND SET ALGEBRA

1. B Braces alone make a dict; the empty set is `set()`.

2. B Set elements must be hashable; use a tuple `(1, 2)` or a `frozenset`.

3. B String hashes are salted per process; the printed order is not reproducible. Sort first.

4. B `remove` raises `KeyError`, `pop` takes no argument and `del` does not apply to sets.

5. B Dict keys are unique and ordered by insertion. `set` loses order; `sorted` imposes a different order.

LESSON 4 · 13 MIN

Hashing and keys — what makes an object usable in a dict or set

Dictionaries and sets are fast because they do not search; they *hash*. A key's hash — an integer computed from its value — picks a slot in a table, and equality is checked only against the one or two keys that landed there. That design imposes one rule on keys, explains why lists cannot be keys and tuples can, decides how your own classes behave as keys, and is the reason string hashes are randomised. This lesson explains the mechanism at the level an interviewer expects, states the hash/equality contract, shows what the built-in types do, and what to define in a class of your own.

The mechanism

```
d["ada"]:  hash("ada") → 3 971 …  → slot 7 of the table → is the key there
```

A dict is an array of slots. Inserting a key computes `hash(key)`, reduces it to an index, and stores the key and value there (probing to the next slot on a collision). Looking up repeats the computation and compares with `==` only against keys in the probed slots. On average that is O(1) whatever the size; the table is resized when it gets too full, so amortised insertion stays O(1). Since 3.6 CPython keeps a separate dense array of entries in insertion order, which is why iteration order is insertion order at no cost.

The worst case — every key hashing to the same slot — degrades to O(n) per operation, which is why hash functions are chosen to spread keys, and why a key type with a constant `__hash__` is a performance bug.

The contract

For hashing to work, two objects that are equal must hash equally:

> If `a == b` then `hash(a) == hash(b)`.

The converse is not required (unequal objects may collide). And a key's hash must **not change** while it is in a table — the table stored it under the old hash and would never find it again.

That is why keys must be immutable in the respects that affect equality, and why a class that is mutable should not be hashable at all.

What the built-ins do

Type	Hashable?	Why
`int`, `float`, `bool`, `str`, `bytes`, `None`	yes	immutable; `hash(1) == hash(1.0) == hash(True)` because they compare equal
`tuple`	yes, if every element is	its hash combines the elements' hashes
`frozenset`	yes	immutable
`list`, `dict`, `set`, `bytearray`	**no**	mutable — the hash would change
user-defined class instances	yes, by default	hashed by *identity* (`id`), and equal only to themselves

`hash(1) == hash(1.0)` and `1 == 1.0` together mean `{1: "a", 1.0: "b"}` has one key. `hash("a")` is stable within one process and different between processes: CPython salts string hashing with a random seed at start-up to defeat collision attacks on web servers, so `hash("a")` is not a value to print, store or compare across runs. Set iteration order for strings follows from this (lesson 3).

Your own classes as keys

By default an instance hashes by identity and `==` is identity, so two `Point(1, 2)` objects are different keys. Defining `__eq__` to compare by value **removes** the default `__hash__` — Python sets it to `None`, because value equality with identity hashing would break the contract — and instances become unhashable until you define `__hash__` too:

```python
class Point:
    def __init__(self, x, y):
        self.x, self.y = x, y

    def __eq__(self, other):
        if not isinstance(other, Point):
            return NotImplemented
        return (self.x, self.y) == (other.x, other.y)

    def __hash__(self):
        return hash((self.x, self.y))       # combine the fields that define e
```

The pattern: hash the tuple of exactly the attributes that `__eq__` compares. A `@dataclass(frozen=True)` (Module 8) writes both methods for you and makes the fields read-only, which is the safest way to get a value-type key; `@dataclass(eq=True)` without `frozen` gives `__eq__` and an *unhashable* class, by the same rule.

Keys in practice

Tuples as compound keys. `visited[(r, c)]`, `distances[(a, b)]`, `counts[(word, year)]` — cheap, ordered, hashable. The parentheses are optional in a subscript: `d[r, c]` is `d[(r, c)]`.

Normalised strings. `d[name.casefold()]` so that "Ada" and "ada" share an entry; normalise once at the boundary.

frozenset for unordered groups. A key that means "these three people, in any order".

Never floats that are computed. `d[0.1 + 0.2]` and `d[0.3]` are different keys (Module 2); round or use integer cents.

Never a mutable object, even one you promise not to change.

---

hash() and the dunders

`hash(x)` calls `x.__hash__()`. `hash` of a tuple of hashables combines them deterministically for a given process (ints and tuples of ints are stable across runs; anything containing a string is not). `__eq__` returning `NotImplemented` for a foreign type lets Python try the reflected comparison and finally fall back to identity, which is the correct behaviour for `point == 5`.

Pitfalls

A list, set or dict as a key or set element.

Defining `__eq__` without `__hash__` and then using the instances as keys — `TypeError: unhashable type`.

Hashing attributes that `__eq__` does not compare (or the reverse) — breaks the contract.

Mutating an object after using it as a key.

Printing or persisting `hash("text")`.

`{1: "a", True: "b"}` having one key.

Key takeaways

Dicts and sets hash a key to a slot and compare with `==` only there: O(1) average, O(n) only under pathological collisions.

Contract: equal objects hash equal; a key's hash must not change while stored — hence immutable keys.

Immutable built-ins and tuples/frozensets of them are hashable; lists, dicts, sets are not; instances hash by identity unless `__eq__` / `__hash__` say otherwise.

Define `__hash__` as `hash(tuple_of_the_compared_fields)` alongside `__eq__`, or use `@dataclass(frozen=True)`.

String hashes are randomised per process — never print, store or order by them.

EXERCISE 1

Points as keys

Read `n` points `x y` (some repeated) and count how often each occurs using a dictionary keyed by the tuple `(x, y)` — written in subscript form as `counts[x, y]`. Print each distinct point with its count, ordered by `x` then `y`, then how many distinct points lie on the diagonal `x == y`.

Input: `n`, then `n` lines. **Output:** `(x, y): count` per distinct point, then `diagonal: <n>`.

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

prints

```
(0, 0): 1
(1, 2): 2
(3, 3): 1
diagonal: 2
```

INPUT 1

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

OUTPUT 1

```
(0, 0): 1
(1, 2): 2
(3, 3): 1
diagonal: 2
```

REFERENCE SOLUTION

```python
n = int(input())
counts = {}
for _ in range(n):
    x, y = map(int, input().split())
    counts[x, y] = counts.get((x, y), 0) + 1
for (x, y), c in sorted(counts.items()):
    print(f"({x}, {y}): {c}")
print(f"diagonal: {sum(1 for x, y in counts if x == y)}")
```

EXERCISE 2

A hashable version

Implement `Version` holding `major` and `minor` parsed from text like `1.2`. Define `__eq__` (by value, returning `NotImplemented` for other types) **and** `__hash__` (over the same fields) so that instances work as set members and dict keys. Read `n` versions; print the number of distinct versions, whether the first two are equal, and the distinct versions sorted (define `__lt__` too) as `major.minor`.

Input: `n`, then `n` version strings (`n ⩾ 2`). **Output:** `distinct <k>`, `first two equal <True/False>`, `sorted <versions>`.

```
4
1.2
1.10
1.2
0.9
```

prints

```
distinct 3
first two equal False
sorted 0.9 1.2 1.10
```

INPUT 1

```
4
1.2
1.10
1.2
0.9
```

OUTPUT 1

```
distinct 3
first two equal False
sorted 0.9 1.2 1.10
```

REFERENCE SOLUTION

```python
class Version:
    def __init__(self, text):
        major, minor = text.split(".")
        self.major, self.minor = int(major), int(minor)

    def _key(self):
        return (self.major, self.minor)

    def __eq__(self, other):
        if not isinstance(other, Version):
            return NotImplemented
        return self._key() == other._key()

    def __hash__(self):
        return hash(self._key())

    def __lt__(self, other):
        return self._key() < other._key()

    def __str__(self):
        return f"{self.major}.{self.minor}"


n = int(input())
versions = [Version(input().strip()) for _ in range(n)]
print(f"distinct {len(set(versions))}")
print(f"first two equal {versions[0] == versions[1]}")
print("sorted", " ".join(str(v) for v in sorted(set(versions))))
```

QUIZ

1. A class defines `__eq__` but not `__hash__`. What happens to `{obj}`?

A. Works, hashing by identity

B. `TypeError: unhashable type` — defining `__eq__` sets `__hash__` to `None`

C. Works, hashing by `__eq__`

D. Works only for frozen instances

2. Which rule must `__hash__` and `__eq__` obey together?

A. Unequal objects must have different hashes

B. Equal objects must have equal hashes

C. Hashes must be positive

D. `__hash__` must return the object's `id`

3. How many keys does `{1: "a", 1.0: "b", True: "c"}` have?

A. 3

B. 1

C. 2

D. It raises `TypeError`

4. Why is `hash("abc")` different between two runs of the same program?

A. Strings are mutable

B. CPython salts string hashing with a random seed per process, to defeat collision attacks

C. It is a bug

D. It is not — it is always the same

5. Which is a valid dictionary key?

A. `[1, 2]`

B. `{1, 2}`

C. `(1, [2])`

D. `(1, frozenset({2}))`

ANSWER KEY · HASHING AND KEYS — WHAT MAKES AN OBJECT USABLE IN A DICT OR SET

1. B Value equality with identity hashing would break the contract, so Python removes the default hash. Define `__hash__` over the compared fields.

2. B Lookups find candidates by hash and confirm by equality, so equal keys must land in the same slot. Collisions between unequal keys are allowed.

3. B `1 == 1.0 == True` and they hash equally, so each assignment overwrites the same key; the value is `"c"`.

4. B Hash randomisation is on by default; it is why set order for strings varies and why hashes must never be printed or persisted.

5. D A tuple is hashable only if all its elements are; lists and sets are not, frozensets are.

LESSON 5 · 14 MIN

Nested data and JSON

Real data is nested: a list of records, each a dict, some values themselves lists of dicts. JSON is the text form of exactly that shape — objects, arrays, strings, numbers, booleans and null — and Python's `json` module maps it onto dicts, lists, `str`, `int` / `float`, `bool` and `None` with no loss. This lesson covers walking nested structures safely, `json.loads` / `dumps` and their options, the round trip and what it does not preserve, deterministic output with `sort_keys`, `pprint`, and the recursive walk that handles a document of unknown depth.

The mapping

JSON	Python
object `{"k": v}`	`dict` (keys are always strings)
array `[…]`	`list`
string	`str`
number	`int` if it has no fraction or exponent, else `float`
`true` / `false`	`True` / `False`
`null`	`None`

```
import json
doc = json.loads('{"name": "ada", "langs": ["python", "c"], "age": 36, "active
doc["langs"][0]                    # 'python'
doc["boss"] is None                # True
json.dumps(doc)                    # '{"name": "ada", "langs": ["python", "c"]
```

`loads` / `dumps` work on strings; `load` / `dump` on file objects (`json.load(sys.stdin)` reads a whole document from standard input). Tuples serialise as arrays and come back as lists; sets, dates, `Decimal` and class instances have no JSON form and raise `TypeError` unless you supply `default=` (Module 14).

dumps options

```
json.dumps(doc, indent=2)                  # pretty-printed, nested by two spa
json.dumps(doc, sort_keys=True)            # keys in sorted order — determinis
json.dumps(doc, separators=(",", ":"))     # compact: no spaces
json.dumps("café", ensure_ascii=False)     # '"café"' rather than '"caf\\u00e9
```

`sort_keys=True` is the option to remember for judged output and for comparing documents: two equal dicts built in different orders serialise identically. `indent` with `sort_keys` is the standard "show me this structure" combination; `pprint.pprint(doc)` does the same for any Python object (sorting dict keys by default) without JSON's restrictions.

Walking with defaults

```
doc.get("address", {}).get("city")                 # None instead of KeyError
langs = doc.get("langs") or []                     # missing or null → empty
first = doc["langs"][0] if doc.get("langs") else None
```

Chained `get` with an empty default of the right type is the safe navigation idiom. When a missing field is a *bug* rather than an option, index directly and let `KeyError` say so. `match` (Module 3) is the cleanest way to validate a document's shape when several shapes are possible:

```
match event:
    case {"type": "click", "pos": [int(x), int(y)]}: ...
    case {"type": "key", "key": str(k)}: ...
    case _: raise ValueError("unknown event")
```

Recursive walks

A document of unknown depth is walked with a recursive function that dispatches on the type of the current node:

```python
def walk(node, path=""):
    if isinstance(node, dict):
        for key, value in node.items():
            yield from walk(value, f"{path}.{key}" if path else key)
    elif isinstance(node, list):
        for i, value in enumerate(node):
            yield from walk(value, f"{path}[{i}]")
    else:
        yield path, node

for path, leaf in walk(doc):
    print(path, "=", leaf)
```

```
name = ada
langs[0] = python
langs[1] = c
age = 36
active = True
boss = None
```

The same skeleton counts leaves, finds every value under a given key, computes depth, or rewrites values in place (return a new structure rather than mutating while walking). `yield from` (Module 11) makes the recursion produce a flat stream.

Building nested structures

```python
report = {"total": 0, "items": []}
for name, qty in rows:
    report["items"].append({"name": name, "qty": qty})
    report["total"] += qty

by_dept = defaultdict(list)                    # then
doc = {"departments": [{"name": d, "staff": sorted(v)} for d, v in sorted(by_
```

Build dicts and lists as the data dictates, then serialise once. Sorting keys and lists before `dumps` is what makes the output stable when the source order was not.

The round trip and its limits

`json.loads(json.dumps(x)) == x` holds for dicts with string keys, lists, strings, ints, floats (within float precision), booleans and `None`. It does **not** hold for: tuples (become lists), non-string keys (`{1: "a"}` becomes `{"1": "a"}`), `float("nan")` / `inf` (emitted as `NaN` / `Infinity`, which is not valid JSON, unless `allow_nan=False` raises), and any custom object. When keys are integers, convert on the way back (`{int(k): v for k, v in loaded.items()}`).

Parse errors raise `json.JSONDecodeError` (a `ValueError`) with the line and column: trailing commas, single quotes and comments are all invalid JSON, however common they are in JavaScript.

Pitfalls

- `json.loads` on a file object or `json.load` on a string (swap them).
- Expecting tuples, sets or int keys to survive the round trip.
- Printing a nested structure without `sort_keys` and comparing against expected output built in another order.
- `doc["a"]["b"]` on a document that may lack `"a"`; chain `get`s or catch `KeyError`.
- Mutating a list while walking it recursively.
- Single quotes or trailing commas in hand-written JSON.

Key takeaways

- JSON objects, arrays, strings, numbers, booleans and null map to dict, list, str, int/float, bool and None; keys are always strings.
- `loads` / `dumps` for strings, `load` / `dump` for files; `indent`, `sort_keys=True` and `ensure_ascii=False` control the text.
- Navigate optional fields with chained `get` and typed defaults; validate shapes with `match`.
- Walk unknown depth with a recursive function dispatching on `dict` / `list` / leaf.
- The round trip loses tuples and non-string keys; sort keys and lists for deterministic output.

EXERCISE 1

Walk the document

Read one JSON document from standard input with `json.load(sys.stdin)` and print every leaf value with its path, using a recursive walk: object keys join with `.`, array indexes appear as `[i]`. Print booleans and `null` the Python way (`True`, `None`). Finish with the number of leaves.

Input: a JSON document. **Output:** `<path> = <value>` per leaf in document order, then `leaves: <n>`.

```
{"name": "ada", "langs": ["python", "c"], "active": true, "boss": null}
```

prints

```
name = ada
langs[0] = python
langs[1] = c
active = True
boss = None
leaves: 5
```

INPUT 1

```
{"name": "ada", "langs": ["python",
"c"], "active": true, "boss": null}
```

OUTPUT 1

```
name = ada
langs[0] = python
langs[1] = c
active = True
boss = None
leaves: 5
```

REFERENCE SOLUTION

```python
import json
import sys


def walk(node, path=""):
    if isinstance(node, dict):
        for key, value in node.items():
            yield from walk(value, f"{path}.{key}" if path else key)
    elif isinstance(node, list):
        for i, value in enumerate(node):
            yield from walk(value, f"{path}[{i}]")
    else:
        yield path, node


doc = json.load(sys.stdin)
count = 0
for path, leaf in walk(doc):
    print(f"{path} = {leaf}")
    count += 1
print(f"leaves: {count}")
```

EXERCISE 2

Canonical JSON

Read a JSON document and print it in canonical form — `json.dumps` with sorted keys and compact separators — then the number of top-level keys (`0` if the document is not an object) and the maximum nesting depth, where a scalar has depth 0 and each enclosing object or array adds 1.

Input: a JSON document. **Output:** the canonical text, then `keys: <n>`, then `depth: <d>`.

```
{"b": [1, {"z": null, "a": 2}], "a": "x"}
```

prints

```
{"a":"x","b":[1,{"a":2,"z":null}]}
keys: 2
depth: 3
```

INPUT 1

```
{"b": [1, {"z": null, "a": 2}], "a":
"x"}
```

OUTPUT 1

```
{"a":"x","b":[1,{"a":2,"z":null}]}
keys: 2
depth: 3
```

REFERENCE SOLUTION

```
import json
import sys


def depth(node):
    if isinstance(node, dict):
        return 1 + max((depth(v) for v in node.values()), default=0)
    if isinstance(node, list):
        return 1 + max((depth(v) for v in node), default=0)
    return 0


doc = json.load(sys.stdin)
print(json.dumps(doc, sort_keys=True, separators=(",", ":")))
print(f"keys: {len(doc) if isinstance(doc, dict) else 0}")
print(f"depth: {depth(doc)}")
```

QUIZ

1. What does `json.loads('{"a": [1, null, true]}')` return?

A. `{'a': [1, None, True]}`

B. `{'a': [1, 'null', 'true']}`

C. `{'a': (1, None, True)}`

D. `JSONDecodeError`

2. What does `json.loads(json.dumps({1: (2, 3)}))` return?

A. `{1: (2, 3)}`

B. `{'1': [2, 3]}`

C. `{1: [2, 3]}`

D. `TypeError`

3. Which `dumps` option makes equal dicts built in different orders produce identical text?

A. `indent=2`

B. `sort_keys=True`

C. `ensure_ascii=False`

D. `separators=(",", ":")`

4. What is the safe way to read `doc["a"]["b"]` when either key may be missing?

A. `doc["a"]["b"] or None`

B. `doc.get("a", {}).get("b")`

C. `doc.get("a").get("b")`

D. `doc["a", "b"]`

5. Which of these is invalid JSON?

A. `{"a": 1}`

B. `[1, 2, 3]`

C. `{'a': 1}`

D. `"text"`

ANSWER KEY · NESTED DATA AND JSON

1. A JSON arrays become lists, `null` becomes `None`, `true` becomes `True`.

2. B JSON keys are always strings, and arrays come back as lists — two things the round trip does not preserve.

3. B Sorting the keys removes insertion order from the output; the other options only change formatting.

4. B The first `get` must supply an empty dict as the default so the second `get` has something to call; `doc.get("a").get("b")` fails with `AttributeError` on `None`.

5. C JSON strings and keys use double quotes only; single quotes, trailing commas and comments are all rejected by `json.loads`.

LESSON 6 · 13 MIN

Choosing a collection — the complexity table and heapq

Every collection question in an interview comes down to a table: which operations does the problem need, and which structure does each of them in constant, logarithmic or linear time. Python ships six structures that cover almost everything — `list`, `tuple`, `dict`, `set`, `deque`, and the heap functions in `heapq` — plus `bisect` over a sorted list. This lesson gives the table, the decision rules that follow from it, the `heapq` API that has not appeared yet, and the memory picture that decides between a list of tuples and a dict of lists when both would work.

The table

Operation	`list`	`tuple`	`dict`	`set`	`deque`	heap (`heapq` on a list)	sorted list + `bisect`
index `x[i]`	O(1)	O(1)	O(1) by key	—	O(n) middle, O(1) ends	—	O(1)
`x in c`	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(log n)
append / add	O(1)	—	O(1)	O(1)	O(1) both ends	O(log n) push	O(n) insort
pop end	O(1)	—	O(1) `popitem`	O(1) arbitrary	O(1) both ends	O(log n) pop min	O(1)
insert / delete middle	O(n)	—	O(1) by key	O(1)	O(n)	—	O(n)
min / max	O(n)	O(n)	O(n)	O(n)	O(n)	O(1) peek min	O(1) ends
ordered iteration	insertion	insertion	insertion	none	insertion	none	sorted
`len`	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)	O(1)

"Amortised" applies to `list.append` and dict/set insertion (occasional resizes); "average" to hashing (pathological collisions aside).

The decision rules

Need to look things up by a key or test membership many times? `dict` or `set`. A list scanned with `in` inside a loop is the most common quadratic bug in Python.

Need order of arrival and access by position? `list`; `tuple` if it never changes.

Need a queue (first in, first out) or a sliding window? `deque` — `append` and `popleft`. A list's `pop(0)` is O(n).

Need a stack (last in, first out)? `list` — `append` and `pop`.

Need the smallest (or largest) element repeatedly while elements keep arriving? A heap — `heapq.heappush` / `heappop` in O(log n). Sorting after every insert is O(n log n) each time.

Need sorted order with binary search, and inserts are rare? A sorted list with `bisect`.

Need to count or group? `Counter`, `defaultdict` (lesson 2).

Need order by key with fast insert and search? Python has no built-in balanced tree; a sorted list with `bisect` (O(n) insert) covers small cases, `sortedcontainers` (third-party) the large ones.

---

heapq

`heapq` turns a plain list into a binary min-heap: the smallest element is always `h[0]`, push and pop are O(log n), and the list itself is the storage.

```
import heapq

h = []
heapq.heappush(h, 5)
heapq.heappush(h, 1)
heapq.heappush(h, 3)
h[0]                        # 1 — peek the minimum without removing
heapq.heappop(h)            # 1
heapq.heapify(xs)           # turn an existing list into a heap in O(n)
heapq.heappushpop(h, x)     # push then pop, cheaper than the two calls
heapq.nsmallest(3, xs)      # the three smallest — O(n log k)
heapq.nlargest(3, xs, key=len)
```

There is no max-heap; push the negation (`-x`) or a tuple `(-priority, item)`. Tuples give priority queues: `heappush(h, (dist, node))` orders by distance first, and ties break on `node` — which must itself be comparable, or you insert a counter as the second element: `(priority, count, item)`. Dijkstra, k-way merge, "top k" and event simulations are the heap's problems. Never iterate a heap expecting sorted order — only `h[0]` is guaranteed; pop repeatedly to get sorted output.

Memory, briefly

A list of `n` small ints is about `8n` bytes of pointers plus the int objects; a dict is roughly three times a list of the same length because of its hash table; a set is similar to a dict; a tuple is slightly smaller than a list; `array` and NumPy store raw numbers at 4–8 bytes each with no per-element object. The consequence for design: a list of tuples `[(name, score), …]` is compact and fine for iteration; a dict `{name: score}` costs more but answers lookups — choose by the operations, and convert (`dict(pairs)`) when the access pattern changes.

Combining structures

Real solutions pair them: a `deque` for the BFS frontier *and* a `set` for visited; a `dict` from key to index *and* a `list` for order; a heap of `(priority, item)` *and* a `dict` of current priorities to detect stale entries; `Counter` for counts *and* `heapq.nlargest` for the top ones. The question is always "which operations, how often" — and the table answers it.

Pitfalls

- `x in list` inside a loop.
- `list.pop(0)` for a queue.
- Re-sorting a list to get the minimum after each insert.
- Pushing incomparable items into a heap as tuples without a tie-breaker.
- Expecting `heapq` to give a max-heap or sorted iteration.
- Choosing a dict for data that is only ever iterated in order.

Key takeaways

- Lookup and membership: dict/set O(1). Position and order: list/tuple. Two-ended queue: deque. Repeated minimum: heap. Sorted with rare inserts: bisect.
- `list.pop(0)` , `in` on a list, and re-sorting per insert are the three quadratic habits to unlearn.
- `heapq` is a min-heap over a list: `heappush` , `heappop` , `h[0]` , `heapify` , `nsmallest` / `nlargest` ; negate or use tuples for max and priorities, with a counter to break ties.
- A dict costs about three times a list; choose by operations, convert when the access pattern changes.

Real solutions combine structures: frontier + visited, heap + dict, counter + top-k.

EXERCISE 1

Priority queue

Implement a priority queue on `heapq` from commands: `push p task` adds a task with integer priority `p` (smaller is more urgent); `pop` removes and prints the most urgent task — on equal priorities the one pushed **first** — or `empty`; `peek` prints it without removing (or `empty`). Store `(priority, sequence, task)` tuples so ties never compare the task text.

Input: commands. **Output:** one line per `pop` and `peek`.

```
push 2 write
push 1 read
push 2 test
pop
pop
peek
pop
pop
```

prints

```
read
write
test
test
empty
```

INPUT 1

```
push 2 write
push 1 read
push 2 test
pop
pop
peek
pop
pop
```

OUTPUT 1

```
read
write
test
test
empty
```

REFERENCE SOLUTION

```
import heapq
import sys

heap = []
seq = 0
for line in sys.stdin:
    cmd, *args = line.split()
    if cmd == "push":
        heapq.heappush(heap, (int(args[0]), seq, " ".join(args[1:])))
        seq += 1
    elif cmd == "pop":
        print(heapq.heappop(heap)[2] if heap else "empty")
    elif cmd == "peek":
        print(heap[0][2] if heap else "empty")
```

EXERCISE 2

Top and bottom k

Read `k` and a line of words. Print the `k` longest words with `heapq.nlargest` and the `k` shortest with `heapq.nsmallest`, both using `key=len` — and, because the heap functions keep first-seen order on ties, print exactly what they return. Then print the `k` largest numbers from a third line of integers, descending.

Input: `k`, a line of words, a line of integers. **Output:** `longest: <words>`, `shortest: <words>`, `largest: <numbers>`.

```
2
fig banana kiwi apple
5 1 9 3
```

prints

```
longest: banana apple
shortest: fig kiwi
largest: 9 5
```

INPUT 1

```
2
fig banana kiwi apple
5 1 9 3
```

OUTPUT 1

```
longest: banana apple
shortest: fig kiwi
largest: 9 5
```

REFERENCE SOLUTION

```python
import heapq

k = int(input())
words = input().split()
nums = list(map(int, input().split()))
print("longest:", " ".join(heapq.nlargest(k, words, key=len)))
print("shortest:", " ".join(heapq.nsmallest(k, words, key=len)))
print("largest:", " ".join(map(str, heapq.nlargest(k, nums))))
```

QUIZ

1. Which structure answers `x in c` in O(1)?

A. `list`

B. `tuple`

C. `set`

D. `deque`

2. What is `h[0]` after `heapq.heapify(h)` on `[5, 1, 3]`?

A. `5`

B. `1`

C. `3`

D. Undefined

3. How do you get a max-heap from `heapq`?

A. `heapq.heapify(h, reverse=True)`

B. Push negated values, or tuples with a negated priority

C. `heapq.maxheap(h)`

D. Sort the list descending first

4. Why is a counter inserted as the second element of heap tuples `(priority, count, task)`?

A. To make pops faster

B. So equal priorities never fall through to comparing `task`, which may be incomparable, and earlier entries win

C. Because heaps require three-tuples

D. To store the heap size

5. Which is the O(n) mistake?

A. `deque.popleft()`

B. `list.pop()`

C. `list.pop(0)`

D. `dict.pop(key)`

ANSWER KEY · CHOOSING A COLLECTION — THE COMPLEXITY TABLE AND HEAPQ

1. C Sets and dicts hash; the sequence types scan.

2. B A heap keeps its minimum at index 0. The rest of the list is *not* sorted — only the heap property holds.

3. B `heapq` is min-only; negation flips the order. Tuples `(-priority, count, item)` build a max-priority queue with stable ties.

4. B Tuples compare element by element; the unique rising counter settles ties deterministically before the third element is ever compared.

5. C Removing the first element of a list shifts every other element; the other operations are constant time.

CHECKPOINT · 25 MIN

Checkpoint — Dictionaries and sets

This checkpoint covers the whole module: dictionaries with `get`, `setdefault`, views and merging, counting and grouping with `Counter` and `defaultdict`, sets with their algebra and the ordering caveat, hashing and the equality contract that decides what can be a key, nested data and JSON with deterministic output, and the complexity table with `heapq`.

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:

- When do you use `d[k]`, `d.get(k)`, `k in d` and `d.setdefault(k, v)`?
- What does `Counter(words).most_common(3)` return, and how are ties ordered?
- What does reading a missing key from a `defaultdict` do?
- Why must a set never be printed directly, and what do you print instead?
- What happens to `__hash__` when a class defines `__eq__`, and how do you fix it?
- Which `json.dumps` option makes two equal dicts serialise identically?
- Which structure gives repeated minimum extraction in O(log n), and how do you get a max-heap from it?

The three programs are a word-frequency report with a deterministic tie-break, a two-snapshot comparison built on set algebra, and a task scheduler that pops the highest-priority job from a heap of tuples.

EXERCISE 1

Word-frequency report

Read text until the end of input. Words are whitespace tokens, case-folded, with leading and trailing punctuation stripped; empty results are ignored. Print the five most frequent words as `<word> <count>` with ties broken alphabetically, then `distinct: <n>` and `hapax: <n>` (words occurring once).

Input: any text. **Output:** up to five lines, then two summary lines.

```
It was the best of times, it was the worst of times.
```

prints

```
it 2
of 2
the 2
times 2
was 2
distinct: 7
hapax: 2
```

INPUT 1

```
It was the best of times, it was the
worst of times.
```

OUTPUT 1

```
it 2
of 2
the 2
times 2
was 2
distinct: 7
hapax: 2
```

REFERENCE SOLUTION

```python
import string
import sys
from collections import Counter

counts = Counter()
for line in sys.stdin:
    for tok in line.split():
        word = tok.casefold().strip(string.punctuation)
        if word:
            counts[word] += 1
for word, n in sorted(counts.items(), key=lambda kv: (-kv[1], kv[0]))[:5]:
    print(word, n)
print(f"distinct: {len(counts)}")
print(f"hapax: {sum(1 for n in counts.values() if n == 1)}")
```

EXERCISE 2

Snapshot diff

Two snapshots of a key–value store arrive as `key=value` lines, separated by a line `--`. Using set operations on the dictionaries' keys, print the keys only in the second snapshot, the keys only in the

first, and the keys in both whose values differ — each sorted and space-separated, or `(none)`.

Input: lines, `--`, lines. **Output:** `added: …`, `removed: …`, `changed: …`.

```
a=1
b=2
c=3
--
b=2
c=4
d=5
```

prints

```
added: d
removed: a
changed: c
```

INPUT 1

```
a=1
b=2
c=3
--
b=2
c=4
d=5
```

OUTPUT 1

```
added: d
removed: a
changed: c
```

REFERENCE SOLUTION

```python
import sys

before, after = {}, {}
current = before
for line in sys.stdin:
    line = line.strip()
    if line == "--":
        current = after
        continue
    if line:
        key, _, value = line.partition("=")
        current[key] = value


def show(label, keys):
    print(f"{label}: {' '.join(sorted(keys)) if keys else '(none)'}")


show("added", after.keys() - before.keys())
show("removed", before.keys() - after.keys())
show("changed", {k for k in before.keys() & after.keys() if before[k] != after[k]})
```

EXERCISE 3

Scheduler

Jobs arrive as `add priority name` (lower number runs first; equal priorities run in arrival order) and `run` executes the next job, printing `run <name>` — or `idle` when nothing is pending. Use a heap of `(priority, arrival, name)` tuples. After the input ends print `pending <n>`.

Input: commands. **Output:** one line per `run`, then the pending count.

```
add 2 backup
add 1 deploy
run
add 1 alert
run
run
run
```

prints

```
run deploy
run alert
run backup
idle
pending 0
```

INPUT 1

```
add 2 backup
add 1 deploy
run
add 1 alert
run
run
run
```

OUTPUT 1

```
run deploy
run alert
run backup
idle
pending 0
```

REFERENCE SOLUTION

```python
import heapq
import sys

heap = []
arrival = 0
for line in sys.stdin:
    cmd, *args = line.split()
    if cmd == "add":
        heapq.heappush(heap, (int(args[0]), arrival, args[1]))
        arrival += 1
    elif cmd == "run":
        if heap:
            _, _, name = heapq.heappop(heap)
            print(f"run {name}")
        else:
            print("idle")
print(f"pending {len(heap)}")
```

CHECKPOINT QUESTIONS

1. What does `d.get("k", 0)` return when `d["k"]` is `None`?

A. `0`

B. `None`

C. `KeyError`

D. `False`

2. What is `Counter("abca").most_common(1)`?

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

B. `('a', 2)`

C. `['a']`

D. `{'a': 2}`

3. After `g = defaultdict(list); g["a"].append(1)`, what is `dict(g)`?

A. `{}`

B. `{'a': [1]}`

C. `{'a': 1}`

D. `defaultdict(...)`

4. What is `{1, 2, 3} - {2} | {4}`?

A. `{1, 3, 4}`

B. `{1, 3}`

C. `{4}`

D. `TypeError`

5. Which object can be a set element?

A. `[1]`

B. `{1}`

C. `{1: 2}`

D. `(1, 2)`

6. Two `Point` instances compare equal by `__eq__` but have no `__hash__`. What does `len({p1, p2})` do?

A. Returns `1`

B. Returns `2`

C. Raises `TypeError: unhashable type`

D. Returns `0`

7. What does `json.dumps({"b": 1, "a": 2}, sort_keys=True)` produce?

A. `'{"b": 1, "a": 2}'`

B. `'{"a": 2, "b": 1}'`

C. `'{a: 2, b: 1}'`

D. `"{'a': 2, 'b': 1}"`

8. What does `json.loads('[1, 2, 3]')` return?

A. `(1, 2, 3)`

B. `[1, 2, 3]`

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

D. `{1, 2, 3}`

9. Which operation on a `heapq` heap is O(log n)?

A. Reading `h[0]`

B. `heapq.heappush`

C. `heapq.heapify`

D. `len(h)`

10. What is the right structure for 'is this element one I have seen before?' inside a loop over a million items?

A. A list with `in`

B. A set with `in`

C. A sorted list with `bisect`

D. A tuple

11. What does `d.pop("k")` do when `"k"` is absent?

A. Returns `None`

B. Raises `KeyError`

C. Returns `False`

D. Inserts `None`

12. Why sort a set before joining it into output?

A. `join` requires sorted input

B. Set iteration order is unspecified and, for strings, randomised per process

C. Sorting removes duplicates

D. It is faster

ANSWER KEY · CHECKPOINT — DICTIONARIES AND SETS

1. B The default applies only when the key is *absent*. A stored `None` is a present value and is returned as is.

2. A `most_common` returns a list of `(element, count)` pairs, even for `n=1`.

3. B The first access created the list and stored it; `dict()` gives a plain copy for display.

4. A `-` binds tighter than `|`, so the difference `{1, 3}` is computed first and then unioned with `{4}`.

5. D Only hashable objects: a tuple of hashables qualifies; lists, sets and dicts do not.

6. C Defining `__eq__` disables the inherited hash; instances cannot enter a set until `__hash__` is defined.

7. B Keys are sorted, and JSON uses double-quoted keys with `: ` and `, ` separators by default.

8. B JSON arrays map to Python lists.

9. B Push and pop are logarithmic; peeking the minimum and `len` are constant; `heapify` of a whole list is O(n).

10. B Set membership is O(1); a list scan would make the loop quadratic; `bisect` needs the collection sorted and inserts are O(n).

11. B Without a default argument `pop` raises; `d.pop("k", None)` returns the default instead.

12. B Only a sorted list gives a reproducible order; the judge compares text exactly.