Python `collections` — 100 Interview Questions with Solutions (Now with Intros)

This light■theme guide opens each section with a crisp introduction covering purpose, when to use, complexities and pitfalls, followed by interview■style Q&As; with runnable snippets.

Type	Core Idea	Complexities (common ops)	Typical Uses
Counter	Multiset (elem→count)	build O(n) • lookup O(1)	Frequency, top■K, histograms
defaultdict	Dict with default factory	get/set O(1)	Grouping, auto init, nested maps
deque	Double■ended queue	append/pop ends O(1) • index O(n)	Queues, sliding windows, BFS
namedtuple	Immutable tuple subclass	index O(1)	Lightweight records, API tuples
OrderedDict	Dict preserving order	insertion O(1) • move/pop O(1)	LRU caches, ordered configs
ChainMap	Stacked dict views	lookup O(k) maps	Layered configs/scopes
UserDict/List/String	Subclassable wrappers	overrides vary	Custom validation/behavior

A. Counter — Introduction

Purpose: fast frequency counting of hashable items with helpers like **most_common**, **elements**, arithmetic, and multiset ops (&, |). Use when you need top■K, duplicates, or histogram■style summaries.
Complexities: build O(n), lookup O(1), most_common(k) ~ O(n log k).
Pitfalls: negative/zero counts persist until cleaned (e.g., c += Counter()); not order■stable beyond Counter rules.

B. defaultdict — Introduction

Purpose: dictionary that supplies a default value for missing keys via a **factory** (e.g., list, int, set). Great for grouping, counting, and nested structures without KeyErrors.
Complexities: get/set average O(1).
Pitfalls: Accessing a missing key *creates* it. Convert to plain dict for JSON/serialization.

C. deque — Introduction

Purpose: a fast double■ended queue supporting O(1) appends/pops from either end; ideal for queues, stacks, and sliding windows.
Complexities: append/pop ends O(1), random access O(n).
Pitfalls: extendleft reverses input order; beware maxlen truncation.

D. namedtuple — Introduction

Purpose: tuple subclass with named fields; immutable, lightweight, hashable, and indexable. Great for simple records without class boilerplate.
Complexities: field access O(1), iteration O(n).
Pitfalls: immutability (use _replace), no methods unless added via subclass/wrapper.

E. OrderedDict — Introduction

Purpose: dict variant that preserves insertion order and supports order operations like move_to_end/popitem; still useful for LRU caches and stable iteration.
Complexities: insert/get O(1), move/pop O(1).
Pitfalls: Normal dicts preserve insertion order since 3.7; use OrderedDict when you need explicit order ops.

F. ChainMap — Introduction

Purpose: present multiple dicts as a single mapping without copying; lookups search through the chain (left■to■right). Great for layered configuration: CLI > env > defaults.
Complexities: lookup O(k) across maps; write goes to first map.

G. UserDict / UserList / UserString — Introduction

Purpose: easy subclassable wrappers around built■ins for adding validation, logging, or custom behavior without touching CPython internals.
Complexities: similar to underlying types; you control overrides.
Pitfalls: Remember to call super() in overrides; ensure invariants in all mutators.

H. Advanced Mixed — Introduction

Purpose: combine multiple `collections` types to solve real interview■style tasks (sliding windows, caches, layered configs, word stats).

A. Q1. Create a Counter from a list

Solution: Use Counter(iterable) to map items to frequencies.

```
from collections import Counter
nums=[1,2,2,3,3,3]
print(Counter(nums))  # Counter({3:3,2:2,1:1})
```

A. Q2. Count characters in a string

Solution: Counter over the string counts each character.

```
from collections import Counter
print(Counter("banana"))  # Counter({'a':3,'n':2,'b':1})
```

A. Q3. Find top 3 frequent elements

Solution: Use most_common(k) to get (item,count) pairs.

```
from collections import Counter
c=Counter([1,1,1,2,2,3])
print(c.most_common(3))  # [(1,3),(2,2),(3,1)]
```

A. Q4. Convert Counter to dict

Solution: dict(counter) returns a plain dict.

```
from collections import Counter
print(dict(Counter('aab')))  # {'a':2,'b':1}
```

A. Q5. Add/Subtract Counters

Solution: Use + to add counts and - to subtract (no negatives kept).

```
from collections import Counter
a=Counter('apple'); b=Counter('pear')
print(a+b); print(a-b)
```

A. Q6. Intersection / Union of Counters

Solution: & takes min counts, | takes max counts.

```
from collections import Counter
a=Counter('banana'); b=Counter('bandana')
print(a&b)  # common
print(a|b)  # merged
```

A. Q7. Check anagrams quickly

Solution: Two strings are anagrams if their Counters match.

```
from collections import Counter
def is_anagram(a,b): return Counter(a)==Counter(b)
print(is_anagram('listen','silent'))
```

A. Q8. Find duplicates in a list

Solution: Filter keys with count>1.

```
from collections import Counter
nums=[1,2,2,3,3,3]
dupes=[k for k,v in Counter(nums).items() if v>1]
print(dupes)
```

A. Q9. Update Counter dynamically

Solution: Use update(iterable or mapping) to increment counts.

```
from collections import Counter
c=Counter(); c.update('abc'); c.update('ab')
print(c)  # Counter({'a':2,'b':2,'c':1})
```

A. Q10. Reconstruct elements from counts

Solution: Use elements() to expand items per frequency.

```
from collections import Counter
c=Counter({'a':2,'b':1})
print(''.join(sorted(c.elements())))  # 'aab'
```

A. Q11. Clean zero/negative entries

Solution: Adding an empty Counter removes zero/negative counts.

```
from collections import Counter
c=Counter(a=3,b=0,c=-1); c+=Counter(); print(c)  # Counter({'a':3})
```

A. Q12. Normalize counts to probabilities

Solution: Divide by total to get simple PMF.

```
from collections import Counter
c=Counter('mississippi'); total=sum(c.values())
pmf={k:v/total for k,v in c.items()}; print(pmf)
```

A. Q13. Frequency histogram buckets

Solution: Map values to bins and count.

```
from collections import Counter
nums=[1,5,7,12,14,20]
bins=lambda x:(x//5)*5
print(Counter(bins(n) for n in nums))
```

A. Q14. Stable top-K with ties

Solution: most_common returns items in descending count; tie-order by first appearance.

```
from collections import Counter
print(Counter('ababc').most_common(2))
```

A. Q15. Count pairs/bigrams

Solution: Slide window of size 2 and count tuples.

```
from collections import Counter
s='banana'
bigrams=[(s[i],s[i+1]) for i in range(len(s)-1)]
print(Counter(bigrams))
```

B. Q16. Create defaultdict(list)

Solution: Auto-initializes missing keys with empty list.

```
from collections import defaultdict
g=defaultdict(list); g['a'].append(1); g['a'].append(2)
print(g)
```

B. Q17. Group words by first letter

Solution: Use first char as key and append words.

```
from collections import defaultdict
words=['apple','banana','avocado','berry']
groups=defaultdict(list)
for w in words: groups[w[0]].append(w)
print(dict(groups))
```

B. Q18. Counting with defaultdict(int)

Solution: Increment counts without KeyError.

```
from collections import defaultdict
cnt=defaultdict(int)
for ch in 'mississippi': cnt[ch]+=1
print(dict(cnt))
```

B. Q19. Nested defaultdict for tree

Solution: Use lambda to create nested dicts on demand.

```
from collections import defaultdict
tree=lambda: defaultdict(tree)
root=tree(); root['cfg']['db']['host']='localhost'
print(root['cfg']['db']['host'])
```

B. Q20. Build adjacency list from edges

Solution: Append neighbors to a list per node.

```
from collections import defaultdict
edges=[('A','B'),('A','C'),('B','C')]
adj=defaultdict(list)
for u,v in edges: adj[u].append(v)
print(dict(adj))
```

B. Q21. Group anagrams

Solution: Key by sorted word; append original.

```
from collections import defaultdict
words=['eat','tea','tan','ate','nat','bat']
grp=defaultdict(list)
for w in words: grp[''.join(sorted(w))].append(w)
print(list(grp.values()))
```

B. Q22. Avoid KeyError with set factory

Solution: Use set to deduplicate during grouping.

```
from collections import defaultdict
m=defaultdict(set)
m['a'].add(1); m['a'].add(1)
print(m)  # {'a':{1}}
```

B. Q23. Join-like grouping by key

Solution: Accumulate values across rows by id.

```
from collections import defaultdict
rows=[('u1','click'),('u1','buy'),('u2','view')]
by_user=defaultdict(list)
for u,ev in rows: by_user[u].append(ev)
print(by_user)
```

B. Q24. Default factory with lambda

Solution: Use lambda for zero/default structure.

```
from collections import defaultdict
zero=defaultdict(lambda:0)
zero['x']+=5; print(zero)
```

B. Q25. Compare with normal dict

Solution: defaultdict avoids try/except or setdefault boilerplate.

```
d={}; key='x'
d[key]=d.get(key,0)+1
# vs:
from collections import defaultdict
dd=defaultdict(int); dd[key]+=1
```

B. Q26. Build invert index

Solution: Map value -> list of keys containing it.

```
from collections import defaultdict
pairs={'a':[1,2],'b':[2,3]}
inv=defaultdict(list)
for k,vals in pairs.items():
    for v in vals: inv[v].append(k)
print(inv)
```

B. Q27. Cumulative sums per group

Solution: Group by key and sum numeric values.

```
from collections import defaultdict
rows=[('A',10),('B',5),('A',7)]
tot=defaultdict(int)
for k,v in rows: tot[k]+=v
print(dict(tot))
```

B. Q28. Two-level grouping

Solution: Group by (country->city->names).

```
from collections import defaultdict
level=lambda: defaultdict(list)
d=defaultdict(level)
d['US']['NY'].append('Alice'); d['US']['SF'].append('Bob')
print(d['US']['NY'])
```

B. Q29. Safe missing key read

Solution: Reading a missing key creates it; beware if you need strict keys.

```
from collections import defaultdict
```

```
dd=defaultdict(list); _=dd['new']; print('new' in dd)  # True
```

B. Q30. Serialize to plain dict

Solution: Convert recursively to dict for JSON.

```
from collections import defaultdict
def to_dict(d):
    if isinstance(d, defaultdict):
        d={k:to_dict(v) for k,v in d.items()}
    return d
```

C. Q31. Append/pop both ends

Solution: deque supports O(1) append/pop at ends.

```
from collections import deque
dq=deque(); dq.append(1); dq.appendleft(0)
dq.pop(); dq.popleft()
```

C. Q32. Rotate k steps

Solution: Positive rotates right, negative left.

```
from collections import deque
dq=deque([1,2,3,4]); dq.rotate(1); print(dq)  # deque([4,1,2,3])
```

C. Q33. Bounded deque with maxlen

Solution: Auto-drops from opposite end.

```
from collections import deque
dq=deque(maxlen=3); [dq.append(i) for i in range(5)]
print(dq)  # deque([2,3,4], maxlen=3)
```

C. Q34. Reverse efficiently

Solution: Use reverse() or slicing-like list(dq)[::-1].

```
from collections import deque
dq=deque([1,2,3]); dq.reverse(); print(dq)
```

C. Q35. Queue (FIFO) example

Solution: append + popleft implements queue.

```
from collections import deque
q=deque(); q.append('task'); print(q.popleft())
```

C. Q36. Stack (LIFO) example

Solution: append + pop implements stack.

```
from collections import deque
s=deque(); s.append(1); s.append(2); print(s.pop())
```

C. Q37. Sliding window sum

Solution: Maintain window of size k with maxlen.

```
from collections import deque
def window_sums(arr,k):
    dq=deque(maxlen=k); out=[]
    cur=0
```

```
    for x in arr:
        if len(dq)==k: cur-=dq[0]
        dq.append(x); cur+=x
        if len(dq)==k: out.append(cur)
    return out
print(window_sums([1,2,3,4,5],3))
```

C. Q38. BFS traversal

Solution: Use deque for O(1) queue ops.

```
from collections import deque
g={0:[1,2],1:[2],2:[3],3:[]}
def bfs(s):
    vis=set([s]); q=deque([s]); order=[]
    while q:
        u=q.popleft(); order.append(u)
        for v in g[u]:
            if v not in vis: vis.add(v); q.append(v)
    return order
print(bfs(0))
```

C. Q39. Palindrome check

Solution: Compare ends popping inward.

```
from collections import deque
def is_pal(s):
    dq=deque(ch.lower() for ch in s if ch.isalnum())
    while len(dq)>1:
        if dq.popleft()!=dq.pop(): return False
    return True
```

C. Q40. Producer-consumer sketch

Solution: deque can model simple buffers (threading omitted).

```
from collections import deque
buf=deque(maxlen=5)
for i in range(7): buf.append(i)
print(buf)
```

C. Q41. Rotate to bring target front

Solution: Find index then rotate.

```
from collections import deque
dq=deque('abcdef'); idx=dq.index('d'); dq.rotate(-idx); print(dq[0])
```

C. Q42. Remove first occurrence

Solution: Use remove(value).

```
from collections import deque
dq=deque([1,2,3,2]); dq.remove(2); print(dq)
```

C. Q43. Merging deques

Solution: Extend or extendleft (note order reversal for extendleft).

```
from collections import deque
a=deque([1,2]); b=deque([3,4])
```

```
a.extend(b); print(a)
```

C. Q44. Streaming average

Solution: Maintain rolling average with window + sum.

```
from collections import deque
class RollingAvg:
    def __init__(self,k): self.k=k; self.d=deque(); self.s=0
    def add(self,x):
        self.d.append(x); self.s+=x
        if len(self.d)>self.k: self.s-=self.d.popleft()
        return self.s/len(self.d)
ra=RollingAvg(3); print(ra.add(1), ra.add(5), ra.add(3), ra.add(7))
```

C. Q45. Time-windowed log buffer

Solution: Use timestamps and pop left when too old.

```
from collections import deque
import time
def keep_last_n_seconds(events, n=60):
    now=time.time()
    while events and now-events[0][0]>n: events.popleft()
```

D. Q46. Create Point namedtuple

Solution: Fields are accessible by name and index.

```
from collections import namedtuple
Point=namedtuple('Point','x y')
p=Point(2,3); print(p.x,p[1])
```

D. Q47. Convert to dict

Solution: Use _asdict() to get a mapping for serialization.

```
from collections import namedtuple
Emp=namedtuple('Emp','id name')
e=Emp(1,'Ana'); print(e._asdict())
```

D. Q48. _replace to copy with changes

Solution: Returns a new instance with fields replaced.

```
from collections import namedtuple
P=namedtuple('P','x y'); p=P(1,2); print(p._replace(y=5))
```

D. Q49. Defaults for fields

Solution: Set defaults via defaults= on Python 3.7+.

```
from collections import namedtuple
Car=namedtuple('Car','make model year', defaults=[2025])
print(Car('Tesla','3'))  # year=2025
```

D. Q50. Serialize to JSON

Solution: Convert with _asdict for json.dumps.

```
from collections import namedtuple
import json
User=namedtuple('User','id name')
```

```
u=User(3,'Bob'); print(json.dumps(u._asdict()))
```

D. Q51. Use as lightweight records

Solution: Great for quick immutable schemas.

```
from collections import namedtuple
Order=namedtuple('Order','id qty price')
o=Order(10,2,5.0); print(o.qty*o.price)
```

D. Q52. Sorting namedtuples

Solution: Sort by field using key=lambda.

```
from collections import namedtuple
P=namedtuple('P','x y')
pts=[P(1,9),P(0,3),P(5,1)]
print(sorted(pts, key=lambda p:p.y))
```

D. Q53. Unpack in loops

Solution: Tuple semantics allow unpacking.

```
from collections import namedtuple
Rec=namedtuple('Rec','k v')
for k,v in [Rec('a',1), Rec('b',2)]: print(k,v)
```

D. Q54. Compare to dataclass

Solution: namedtuple is lightweight/immutable; dataclass is mutable and richer.

```
# Conceptual comparison
```

D. Q55. Nested namedtuple

Solution: Compose records for structure.

```
from collections import namedtuple
Coord=namedtuple('Coord','lat lon')
Place=namedtuple('Place','name loc')
p=Place('Home', Coord(1.0,2.0)); print(p.loc.lat)
```

D. Q56. Field names as list

Solution: Use _fields to inspect schema.

```
from collections import namedtuple
A=namedtuple('A','x y'); print(A._fields)
```

D. Q57. Memory efficiency

Solution: Smaller overhead than dict-based records.

```
# Conceptual note
```

D. Q58. Validation pattern

Solution: Wrap constructor to validate before making a record.

```
from collections import namedtuple
User=namedtuple('User','name age')
def mk(name,age):
    if age<0: raise ValueError
    return User(name,age)
```

D. Q59. Use with CSV

Solution: Map rows to namedtuples for readable access.

```
from collections import namedtuple
Row=namedtuple('Row','id name')
rows=[Row(1,'A'), Row(2,'B')]
print(rows[0].name)
```

D. Q60. Immutability benefits

Solution: Safe to share, hashable by default.

```
from collections import namedtuple
P=namedtuple('P','x y'); p=P(1,2); s=set([p])
```

E. Q61. Preserve insertion order

Solution: OrderedDict preserves order; since 3.7, regular dicts do too for insertion order.

```
from collections import OrderedDict
od=OrderedDict(); od['a']=1; od['b']=2; print(list(od.keys()))
```

E. Q62. Move key to end/start

Solution: Use move_to_end(key, last=True/False).

```
from collections import OrderedDict
od=OrderedDict([('a',1),('b',2),('c',3)])
od.move_to_end('b'); od.move_to_end('c', last=False); print(list(od))
```

E. Q63. Pop last item (LIFO)

Solution: Use popitem(last=True).

```
from collections import OrderedDict
od=OrderedDict([('a',1),('b',2)]); print(od.popitem())
```

E. Q64. Stable iteration for diffs

Solution: Order is predictable for diffing configs.

```
from collections import OrderedDict
# Conceptual example
```

E. Q65. Sort by value into OrderedDict

Solution: Create new OrderedDict from sorted items.

```
from collections import OrderedDict
d={'a':3,'b':1,'c':2}
od=OrderedDict(sorted(d.items(), key=lambda kv: kv[1]))
print(od)
```

E. Q66. Simple LRU cache

Solution: Reinsert on access; evict oldest when size exceeded.

```
from collections import OrderedDict
class LRU:
    def __init__(self, cap): self.cap=cap; self.od=OrderedDict()
    def get(self,k):
        if k not in self.od: return None
        self.od.move_to_end(k); return self.od[k]
```

```
    def put(self,k,v):
        if k in self.od: self.od.move_to_end(k)
        self.od[k]=v
        if len(self.od)>self.cap: self.od.popitem(last=False)
```

E. Q67. Equality with dicts

Solution: OrderedDict equality ignores order when compared to regular dict.

```
from collections import OrderedDict
od=OrderedDict([('a',1),('b',2)])
print(od=={'b':2,'a':1})
```

E. Q68. Reverse iterate

Solution: Use reversed(od).

```
from collections import OrderedDict
od=OrderedDict([('a',1),('b',2),('c',3)])
print(list(reversed(od)))
```

E. Q69. Track modification order

Solution: Move keys on update to reflect recent use.

```
from collections import OrderedDict
od=OrderedDict(); od['x']=1; od['x']=2; od.move_to_end('x'); print(list(od))
```

E. Q70. Freeze order snapshot

Solution: Convert to tuple of items for hashable snapshot.

```
from collections import OrderedDict
od=OrderedDict(a=1,b=2); snap=tuple(od.items())
```

E. Q71. Config writer preserves order

Solution: Write lines in insertion order.

```
from collections import OrderedDict
cfg=OrderedDict([('host','localhost'),('port',5432)])
```

E. Q72. Diff two OrderedDicts

Solution: Compare item sequences to detect reorders.

```
from collections import OrderedDict
a=OrderedDict([('a',1),('b',2)]); b=OrderedDict([('b',2),('a',1)])
# same values, different order
```

E. Q73. Cache of fixed size

Solution: Use popitem(last=False) to drop oldest.

```
from collections import OrderedDict
od=OrderedDict();
for i in range(5):
    od[i]=i*i
    if len(od)>3: od.popitem(last=False)
print(od)
```

E. Q74. JSON-friendly order

Solution: Ordered mapping serializes in order with json.dumps on 3.7+.

```
import json
from collections import OrderedDict
od=OrderedDict([('a',1),('b',2)]); print(json.dumps(od))
```

E. Q75. Reordering by custom rule

Solution: Rebuild OrderedDict from custom-sorted keys.

```
from collections import OrderedDict
d={'x':1,'y':2,'z':3}
keys=sorted(d, key=lambda k: (-len(k), k))
od=OrderedDict((k,d[k]) for k in keys)
```

F. Q76. Combine multiple dicts as layers

Solution: Searches in the first map, then next, etc.

```
from collections import ChainMap
defaults={'color':'blue'}; env={'color':'red','debug':True}
cm=ChainMap(env, defaults); print(cm['color'], cm['debug'])
```

F. Q77. Write-through to first map

Solution: Assignment affects only the first mapping.

```
from collections import ChainMap
a={'x':1}; b={'x':2}
cm=ChainMap(a,b); cm['x']=10; print(a['x'])
```

F. Q78. New child layer

Solution: Use new_child to push a new mapping.

```
from collections import ChainMap
cm=ChainMap({'a':1}).new_child({'b':2})
print(list(cm.maps))
```

F. Q79. Reordering layers

Solution: parents property allows changing order.

```
from collections import ChainMap
a={'x':1}; b={'x':2}
cm=ChainMap(a,b); cm2=cm.parents
print(cm2['x'])  # 2
```

F. Q80. Flatten ChainMap

Solution: Merge into a single dict (right-most precedence).

```
from collections import ChainMap
cm=ChainMap({'a':1},{'a':2,'b':3})
flat=dict(cm)  # first map takes precedence
```

F. Q81. Layered config example

Solution: Command-line > env > defaults.

```
from collections import ChainMap
cli={'timeout':5}; env={'timeout':10}; defaults={'timeout':30}
cfg=ChainMap(cli, env, defaults); print(cfg['timeout'])
```

F. Q82. Delete affects first map

Solution: Del removes key from first map if present.

```
from collections import ChainMap
a={'x':1}; b={'x':2}; cm=ChainMap(a,b)
del cm['x']; print('x' in a, 'x' in b)
```

F. Q83. Missing key lookup

Solution: Raises KeyError if missing from all maps.

```
from collections import ChainMap
cm=ChainMap({'a':1},{'b':2})
# cm['c'] -> KeyError
```

F. Q84. Iterate keys/values

Solution: Iteration reflects combined keys.

```
from collections import ChainMap
cm=ChainMap({'a':1},{'b':2,'a':9})
print(list(cm.keys()), list(cm.values()))
```

F. Q85. ChainMap vs dict union

Solution: Union copies; ChainMap is a view (no copy).

```
from collections import ChainMap
a={'x':1}; b={'y':2}
cm=ChainMap(a,b); d=a|b  # Python 3.9+
```

G. Q86. UserDict with validation

Solution: Subclass to enforce rules on setitem.

```
from collections import UserDict
class Positive(UserDict):
    def __setitem__(self,k,v):
        if v<0: raise ValueError('neg'); super().__setitem__(k,v)
p=Positive(); p['a']=1
```

G. Q87. Case-insensitive dict

Solution: Normalize keys on set/get.

```
from collections import UserDict
class CIDict(UserDict):
    def __setitem__(self,k,v): super().__setitem__(k.lower(),v)
    def __getitem__(self,k): return super().__getitem__(k.lower())
d=CIDict(); d['Host']='x'; print(d['HOST'])
```

G. Q88. Access counter dict

Solution: Track reads via __getitem__.

```
from collections import UserDict
class ReadCount(UserDict):
    def __init__(self): super().__init__(); self.reads=0
    def __getitem__(self,k): self.reads+=1; return super().__getitem__(k)
rc=ReadCount(); rc['a']=1; _=rc['a']; print(rc.reads)
```

G. Q89. Append-only list

Solution: Override remove/pop to forbid.

```
from collections import UserList
class AppendOnly(UserList):
    def pop(self,*a,**k): raise RuntimeError('no pop')
al=AppendOnly([1]); al.append(2)
```

G. Q90. Even-only list

Solution: Validate values on append/extend.

```
from collections import UserList
class EvenList(UserList):
    def append(self,x):
        if x%2: raise ValueError; super().append(x)
el=EvenList(); el.append(2)
```

G. Q91. Bounded list length

Solution: Enforce max size in append.

```
from collections import UserList
class Bounded(UserList):
    def __init__(self,maxlen): super().__init__(); self.maxlen=maxlen
    def append(self,x):
        if len(self)>=self.maxlen: raise OverflowError
        super().append(x)
b=Bounded(2); b.append(1); b.append(2)
```

G. Q92. UserString transform

Solution: Auto uppercase on init.

```
from collections import UserString
class Upper(UserString):
    def __init__(self,s): super().__init__(str(s).upper())
u=Upper('hi'); print(u)
```

G. Q93. Audit trail dict

Solution: Record history of updates.

```
from collections import UserDict
class Audit(UserDict):
    def __init__(self): super().__init__(); self.log=[]
    def __setitem__(self,k,v): self.log.append((k,v)); super().__setitem__(k,v)
a=Audit(); a['x']=1; print(a.log)
```

G. Q94. Read-only view

Solution: Block writes by overriding methods.

```
from collections import UserDict
class ReadOnly(UserDict):
    def __setitem__(self,k,v): raise TypeError('ro')
ro=ReadOnly({'a':1})
```

G. Q95. Template string wrapper

Solution: Provide render().

```
from collections import UserString
class T(UserString):
    def render(self, **kw): return self.format_map(kw)
```

```
t=T('Hello {name}'); print(t.render(name='Ada'))
```

G. Q96. Versioned dict

Solution: Increment version on change.

```
from collections import UserDict
class Versioned(UserDict):
    def __init__(self): super().__init__(); self.ver=0
    def __setitem__(self,k,v): self.ver+=1; super().__setitem__(k,v)
v=Versioned(); v['a']=1; print(v.ver)
```

G. Q97. Key whitelist dict

Solution: Only allow predefined keys.

```
from collections import UserDict
class White(UserDict):
    def __init__(self, allowed): super().__init__(); self.allowed=set(allowed)
    def __setitem__(self,k,v):
        if k not in self.allowed: raise KeyError; super().__setitem__(k,v)
w=White({'id','name'}); w['id']=1
```

G. Q98. Lowercasing UserString ops

Solution: Override __add__/join behaviors if needed.

```
# Conceptual customization point
```

G. Q99. Comparable UserList

Solution: Define rich comparisons based on sum.

```
from collections import UserList
class Summed(UserList):
    def __lt__(self,other): return sum(self)<sum(other)
a=Summed([1,2]); b=Summed([2,2]); print(a<b)
```

H. Q100. Word frequency analyzer

Solution: Use Counter + defaultdict for positions.

```
from collections import Counter, defaultdict
text='to be or not to be'
freq=Counter(text.split()); pos=defaultdict(list)
for i,w in enumerate(text.split()): pos[w].append(i)
print(freq); print(dict(pos))
```

H. Q101. LRU Cache with OrderedDict

Solution: Move key to end on access; pop oldest when full.

```
from collections import OrderedDict
class LRU:
    def __init__(self,cap): self.cap=cap; self.od=OrderedDict()
    def get(self,k):
        if k not in self.od: return None
        self.od.move_to_end(k); return self.od[k]
    def put(self,k,v):
        if k in self.od: self.od.move_to_end(k)
        self.od[k]=v
        if len(self.od)>self.cap: self.od.popitem(last=False)
```

H. Q102. Flatten nested lists using deque

Solution: Iteratively pop/extend to avoid recursion.

```
from collections import deque
def flatten(xs):
    out=[]; dq=deque([xs])
    while dq:
        cur=dq.popleft()
        if isinstance(cur,list): dq.extendleft(reversed(cur))
        else: out.append(cur)
    return out
print(flatten([1,[2,[3,4]],5]))
```

H. Q103. BFS shortest path

Solution: Track parents while visiting.

```
from collections import deque
g={0:[1,2],1:[2],2:[3],3:[]}
def shortest(s,t):
    q=deque([s]); par={s:None}
    while q:
        u=q.popleft()
        if u==t: break
        for v in g[u]:
            if v not in par: par[v]=u; q.append(v)
    if t not in par: return None
    path=[]; cur=t
    while cur is not None: path.append(cur); cur=par[cur]
    return list(reversed(path))
print(shortest(0,3))
```

H. Q104. Layered configs via ChainMap

Solution: Override defaults by env/cli.

```
from collections import ChainMap
defaults={'timeout':30,'debug':False}
env={'debug':True}
cli={'timeout':5}
cfg=ChainMap(cli, env, defaults); print(cfg['timeout'], cfg['debug'])
```

H. Q105. Summary by record field

Solution: namedtuple list → Counter by field.

```
from collections import namedtuple, Counter
Sale=namedtuple('Sale','sku qty')
rows=[Sale('A',2),Sale('B',1),Sale('A',5)]
print(Counter(r.sku for r in rows))
```

H. Q106. Sliding window max with deque

Solution: Store indices; pop smaller from right, out-of-window from left.

```
from collections import deque
def sw_max(a,k):
    dq=deque(); out=[]
    for i,x in enumerate(a):
        while dq and a[dq[-1]]<=x: dq.pop()
```

```
        dq.append(i)
        if dq[0]==i-k: dq.popleft()
        if i>=k-1: out.append(a[dq[0]])
    return out
print(sw_max([1,3,-1,-3,5,3,6,7],3))
```

H. Q107. Word co-occurrence matrix (toy)

Solution: Count co-occurrences within a window.

```
from collections import defaultdict
words='a b c a b a'.split()
win=2; co=defaultdict(int)
for i,w in enumerate(words):
    for j in range(i+1, min(i+win+1, len(words))):
        co[tuple(sorted((w, words[j])))] += 1
print(dict(co))
```

H. Q108. Time-bucketed cache

Solution: OrderedDict keys by time; purge old.

```
from collections import OrderedDict
# Skeleton for TTL-based cache using OrderedDict order
```

H. Q109. Immutable config via UserDict

Solution: Expose mapping but block writes.

```
from collections import UserDict
class Frozen(UserDict):
    def __setitem__(self,k,v): raise TypeError('frozen')
cfg=Frozen({'a':1})
```