04

Data Structures

list, tuple, dict, set — and comprehensions

Chapter 4 · the four containers almost all AI code is built from

04 The journey — four containers, one superpower

Four built-in collections carry almost all the AI code you will read. We take them one at a time, lean on your JavaScript instincts, and flag exactly where they mislead you.

Master these and you can trace a request through an SDK, dig into a parsed LLM response, and spot the bugs reviewers miss.

04

list — the everyday sequence

Python's growable, ordered, changeable line-up of items

A list is like a JavaScript array: an ordered run of items you can grow and change (the word is 'mutable'). It can hold a mix of any types.

```
models = ['gpt-4o', 'claude-opus', 'llama-3']
mixed  = [1, 'two', 3.0, ['nested']]   # any mix is fine
len(models)   # => 3    (len is a function, not a .length)
models[0]     # => 'gpt-4o'
models[-1]    # => 'llama-3'   negative index counts from the END
```

Negative indexing is the treat JS lacks: models[-1] is the last item, models[-2] the one before. No arr[arr.length - 1] dance.

04

Slicing — a[start:stop:step]

The list feature JS developers stumble on

```
nums = [0, 1, 2, 3, 4, 5]
nums[1:4]    # => [1, 2, 3]   start:stop  (stop is EXCLUDED)
nums[:3]     # => [0, 1, 2]   from the beginning
nums[3:]     # => [3, 4, 5]   to the end
nums[::2]    # => [0, 2, 4]   every 2nd item  (the step)
nums[::-1]   # => [5, 4, 3, 2, 1, 0]   reverse — memorise this
```

A slice returns a NEW list. start is included, stop is excluded — same as JS arr.slice, but Python adds the step and negative indices. Out-of-range never errors: nums[10:99] is just [].

a[::-1] reverses a sequence. You will see it constantly — reversing token lists, undoing a sort.

04

Changing a list in place

Add, insert, remove — and one naming trap

```
xs = [3, 1, 2]
xs.append(4)        # add ONE item           -> [3, 1, 2, 4]
xs.extend([5, 6])   # add MANY items         -> [3, 1, 2, 4, 5, 6]
xs.insert(0, 99)    # insert at a position
last = xs.pop()     # remove AND return the last item
xs.remove(99)       # remove first matching value (errors if absent)
```

append vs extend — the classic mix-up

xs.append([5, 6]) adds the LIST itself as one nested item.
xs.extend([5, 6]) adds its elements. Different results, easy to confuse.

04

Sorting — sorted() vs .sort()

Two forms, and the distinction is a bug magnet

```
data = [3, 1, 2]
new = sorted(data)      # returns a NEW sorted list; data untouched
data.sort()             # sorts IN PLACE, returns None
result = data.sort()    # BUG magnet: result is None!

# sort by a key — e.g. tokens by descending logprob:
toks.sort(key=lambda d: d['lp'], reverse=True)
```

Reading and judging — x = mylist.sort() then using x is a bug: x is None. Good code uses sorted(...) when it needs the result back, and .sort() only for its side effect. JS sort mutates AND returns the array; Python's .sort() returns None.

04

tuple — the immutable sequence

Looks like a list, but cannot be changed once made

A tuple uses ( ) and CANNOT be changed after you create it (the word is 'immutable').

That is the point: it signals a fixed record, and — unlike a list — it can be a dict key or a set member.

```
point = (0.12, -0.34)
point[0]          # => 0.12
# point[0] = 9    # ERROR: can't change a tuple

a, b = 1, 2       # pack, then unpack
a, b = b, a       # swap — no temp variable

one = (5)         # just the int 5
one = (5,)        # a 1-tuple — the COMMA makes
it
```

Why so many functions 'return two things': they return a tuple and you unpack it. dict.items() gives (key, value) pairs; enumerate() gives (index, item) pairs.

04

dict — the workhorse

Every JSON payload, config and LLM response is one

A dict (hash map) pairs keys with values. Keys are usually text; values are anything. Since Python 3.7 it keeps items in the order you added them.

```
config = {'model': 'claude-opus', 'temperature': 0.7}
config['model']          # => 'claude-opus'
config['stream'] = True # add a new key
'model' in config        # => True   (in checks KEYS, not values)

config['seed']           # KeyError -> crashes!
config.get('seed')       # => None              safe
config.get('seed', 42)   # => 42                safe with a fallback
```

d[k] vs .get(k) is the safety split that bites hardest when parsing model output: d['missing'] throws, d.get('missing', fallback) is safe.

04

Walking a parsed LLM response

A dict of lists of dicts — dig in step by step

```
response = {
    'choices': [
        {'message': {'role': 'assistant', 'content': 'Hello there!'}}
    ],
    'usage': {'completion_tokens': 3},
}
# dig in:  dict -> list[0] -> dict -> dict -> str
text = response['choices'][0]['message']['content']   # 'Hello there!'

# defensive version for real code, where a field may be missing:
choices = response.get('choices', [])
if choices:
    text = choices[0].get('message', {}).get('content', '')
```

Reading and judging — the number-one LLM-parsing bug is raw resp['choices'][0]['message']['content'] on a response that came back empty or shaped differently (a tool call, not text). It throws in production. Raw [ ] indexing on model output is a red flag; .get with sensible defaults shows someone thought about failure.

04 set — unique items, fast membership

Two jobs: remove duplicates, and answer 'is it in here?' fast

```
tags = ['nlp', 'rag', 'nlp', 'agents', 'rag']
unique = set(tags)    # => {'agents', 'nlp', 'rag'}   duplicates gone
'gpt-4o' in seen      # => True   O(1) average, vs O(n) for a list

a = {'cat', 'dog'};   b = {'dog', 'bird'}
a | b   # union          => {'cat', 'dog', 'bird'}
a & b   # intersection   => {'dog'}
a - b   # difference     => {'cat'}
```

Reading and judging — if code keeps a LIST of seen IDs and does 'if x in seen_list:' inside a loop, that is O(n) per check, quietly O(n squared) on big data. A set makes it O(1). Choosing a list where a set (membership) or dict (lookup) belongs is a common performance red flag.

Gotcha: {} on its own is an EMPTY DICT, not a set. Use set() for an empty set.

04

Comprehensions — the Pythonic superpower

Build a whole collection in one expression

A comprehension builds a collection in one line. The shape is: [expression for item in iterable if condition]. AI codebases are saturated with them — you must read them fluently.

```
nums = [1, 2, 3, 4, 5, 6]
squares = [n * n for n in nums]              # => [1, 4, 9, 16, 25, 36]
evens   = [n for n in nums if n % 2 == 0]  # => [2, 4, 6]

# JS:  nums.filter(n => n % 2 === 0).map(n => n * 10)
result = [n * 10 for n in nums if n % 2 == 0]   # => [20, 40, 60]
```

One comprehension replaces a chained .filter().map(). Read it as: build this, for each item, keeping only those that pass the if.

04

dict, set, generator & nested

The same idea, four more shapes

```
# dict & set comprehensions use { }
lengths = {w: len(w) for w in ['rag', 'agent']}   # {'rag': 3, 'agent': 5}
firsts  = {w[0] for w in ['rag', 'react']}        # {'r'}   (a set)

# generator ( ) is LAZY — one item at a time, no list built
total = sum(len(c) for c in contents)

# nested: flatten a list of lists (read for-clauses left to right)
flat = [x for batch in batches for x in batch]    # => [1, 2, 3, 4, 5]
```

Reading and judging — comprehensions are excellent for ONE map or filter. But a comprehension with multiple for's, multiple if's and a ternary is write-once, read-never. If you can't parse it in one glance, an explicit for-loop is the better code — deep nesting is a legitimate review comment, not nitpicking.

04

Unpacking & the star operator

Pulling 'the rest' out, and spreading it back in

```
xs = [1, 2, 3, 4, 5]
first, *rest = xs        # first=1,  rest=[2, 3, 4, 5]
first, *mid, last = xs   # first=1,  mid=[2, 3, 4],  last=5

def call_model(model, **extra):   # ** gathers extra keyword args
    ...
opts = {'temperature': 0.2, 'seed': 7}
call_model('claude-opus', **opts)   # ** spreads the dict back out
```

The * captures 'the rest' into a list. In a function's signature *args collects extra positional args into a tuple, **kwargs extra keyword args into a dict. The same stars SPREAD a sequence or dict when you call.

JS to Python: one '...' splits into two. * for sequences/positional, ** for dicts/keyword.
[a, ...rest] -> a, *rest
fn(...args) -> fn(*args)
{...obj} -> **obj

04

Copy semantics — the trap that bites everyone

Assignment never copies — it just adds another label

```
a = [1, 2, 3]
b = a              # b and a point at the SAME list — no copy!
b.append(4)
print(a)           # => [1, 2, 3, 4]   surprise — a changed
too
```

```
def add_msg(msg, history=[]):    # DON'T — the default [] is
    history.append(msg)          #   made ONCE and shared!
    return history
add_msg('a')    # => ['a']
add_msg('b')    # => ['a', 'b']   leaked across calls!

def add_msg_ok(msg, history=None):        # DO this instead
    history = [] if history is None else history
```

Reading and judging

def f(x, items=[]) is an instant red flag — the shared default builds up state across calls and causes ghost bugs. Fix: =None plus a guard inside.

.copy(), list(a) or a[:] make a SHALLOW copy — a new outer container, but nested objects are still shared.

copy.deepcopy(x) makes a fully independent copy. Good code copies before mutating a caller's data.

04 The JS –> Python cheat sheet

```
arr.length              ->   len(xs)

push  /  concat(b)      ->   append  /  a + b

arr.slice(1, 4)         ->   xs[1:4]   (plus step, negatives)

[...arr].reverse()      ->   xs[::-1]

.map()  /  .filter()    ->   [f(x) for x in xs]  /  ... if c

{ ...a, ...b }          ->   { **a, **b }

obj?.a?.b               ->   d.get('a', {}).get('b')
```

```
The four at a glance

list    [1, 2]     ordered · changeable · dupes ok

tuple   (1, 2)     ordered · FIXED · can be a key

dict    {'k': 1}   keyed lookup · ordered (3.7+)

set     {1, 2}     unordered · UNIQUE · fast 'in'
```

The teaching — pick the container that fits: list for order, tuple for fixed records, dict for lookup, set for membership. The bugs reviewers miss live in mutation, aliasing and raw [ ] indexing on model output — guard those and you can read, and judge, any AI codebase.