Python Programming

DAY 2: DATA STRUCTURES & FUNCTIONS

Complete Lecture Notes • Code Examples • Projects • Interview Questions

Topics Covered Today

Dictionaries • Sets • List Comprehensions • Complex Data
Functions • Scope • Lambda • *args/**kwargs • Recursion • Map/Filter

Sections 10 Detailed Sections	**Code Examples** 60+ Working Programs	**Interview Qs** 35+ Q&A Pairs

SECTION 1: Dictionaries — The Key-Value Store

1.1 What is a Dictionary?

Dictionary □□□ Python □□ □□□□□□□□ □□□□□□□□□□ □□□ □□□□ □□□□□□□□ data structure□ □□□ key-value pair □ data store □□□□ Real world □□ phone book □□ □□□ — □□□ (key) □□□□□ number (value) □□□□□□□ Dictionary □ □□□□□□□□ key unique □□□ □□□, □□□□□□□ value duplicate □□□ □□□□□□

Python 3.7+ □□□□ dictionary insertion order maintain □□□ — □□□□□□□ □□ order □ □□□□ item add □□□, □□ order □□ □□□□□□

Property	Explanation
`Ordered`	Python 3.7+ □ insertion order preserved
`Mutable`	□□□□□□□ □□□□ add, update, delete □□□ □□□□
`Keys unique`	□□□ key □□□□□□□ □□□□□□ □□□□ □□ — last value □□□□
`Keys immutable`	Key □□□□□□□ str, int, tuple □□□□□□□□ □□□ □□□□, list □□□
`Fast lookup`	Key □□□□□ value □□□□□ O(1) — hash table □□ □□□□□□

1.2 Creating Dictionaries

```
# ── Method 1: Curly braces (most common) ──
student = {
    "name": "Alice",
    "age": 22,
    "grade": "A",
    "is_active": True
}

# ── Method 2: dict() constructor ──
person = dict(name="Bob", age=25, city="Dhaka")

# ── Method 3: Empty dictionary ──
empty1 = {}
empty2 = dict()

# ── Method 4: From two lists using zip ──
keys   = ["a", "b", "c"]
values = [1,   2,   3  ]
d = dict(zip(keys, values))
print(d)        # {'a': 1, 'b': 2, 'c': 3}

# ── Method 5: dict.fromkeys() ──
# Create dict with same default value for all keys
subjects = ["Math", "English", "Science"]
```

```
marks = dict.fromkeys(subjects, 0)
print(marks)    # {'Math': 0, 'English': 0, 'Science': 0}

# ─── Keys can be any immutable type ───
mixed_keys = {
    "string_key": "hello",
    42:            "integer key",
    (1, 2):        "tuple key",
    True:          "bool key",
}
```

1.3 Accessing Dictionary Values

```
student = {"name": "Alice", "age": 22, "grade": "A"}

# ─── Method 1: Square bracket [] — raises KeyError if missing ───
print(student["name"])      # Alice
print(student["age"])       # 22
# print(student["phone"])   # ✕ KeyError!

# ─── Method 2: .get() — safe, returns None or default ───
print(student.get("grade"))         # A
print(student.get("phone"))         # None  (no error!)
print(student.get("phone", "N/A"))  # N/A   (custom default)

# ─── ALWAYS prefer .get() when key might not exist ───

# ─── Checking if key exists ───
if "name" in student:
    print("Name found:", student["name"])

if "phone" not in student:
    print("Phone not in record")

# ─── Accessing nested dict ───
person = {
    "name": "Bob",
    "address": {
        "city": "Dhaka",
        "zip": "1207"
    }
}
print(person["address"]["city"])            # Dhaka
print(person.get("address", {}).get("zip")) # 1207
```

1.4 Modifying Dictionaries

```
student = {"name": "Alice", "age": 22}

# ─── Adding new key-value ───
student["email"] = "alice@email.com"
```

```
student["gpa"]   = 3.8
print(student)

# ─── Updating existing value ───
student["age"] = 23          # overwrite
print(student["age"])        # 23

# ─── .update() — merge another dict ───
extra_info = {"city": "Dhaka", "year": 3}
student.update(extra_info)
print(student)

# ─── .update() also works with keyword args ───
student.update(phone="01711-000000", gpa=3.9)

# ─── Deleting entries ───
del student["gpa"]                      # delete specific key
popped = student.pop("email")           # remove & return value
popped2 = student.pop("phone", "NA")  # safe pop with default
last = student.popitem()                # remove & return LAST item

# ─── Clearing all entries ───
temp = {"a": 1, "b": 2}
temp.clear()
print(temp)   # {}
```

1.5 Dictionary Methods — Complete Reference

Method	□□ □□□
`d.keys()`	□□ keys return □□□ — dict_keys(['name', 'age'])
`d.values()`	□□ values return □□□ — dict_values(['Alice', 22])
`d.items()`	□□ (key,value) tuples return □□□
`d.get(k,def)`	Key □□ value □□ default — KeyError □□□
`d.update(d2)`	d2 □□ □□ items □□ d □□ merge □□□
`d.pop(k,def)`	Key remove □□□ value return □□□
`d.popitem()`	Last inserted item remove □□□ (k,v) return
`d.setdefault(k,v)`	Key □□ □□□□□ default value set □□□
`d.copy()`	Shallow copy □□□□ □□□
`d.clear()`	□□ items delete □□□
`k in d`	Key □□□ □□□□ check — O(1) operation
`len(d)`	Dictionary □□ □□□□ items □□□

```
info = {"name": "Alice", "age": 22, "city": "Dhaka"}
```

```
# ─── Iterating over dictionary ───
# Iterate keys (default)
for key in info:
    print(key, "->", info[key])

# Iterate keys explicitly
for key in info.keys():
    print(key)

# Iterate values
for val in info.values():
    print(val)

# Iterate key-value pairs (MOST COMMON)
for key, value in info.items():
    print(f"{key}: {value}")

# ─── setdefault ───
d = {"visits": 5}
d.setdefault("visits", 0)    # already exists — no change
d.setdefault("likes", 0)     # doesn't exist — sets to 0
print(d)    # {'visits': 5, 'likes': 0}

# ─── Counting with dict ───
text = "banana"
freq = {}
for char in text:
    freq[char] = freq.get(char, 0) + 1
print(freq)   # {'b': 1, 'a': 3, 'n': 2}
```

1.6 Nested Dictionaries

Dictionary □□ value □□□□□□□ □□□□□□□ dictionary □□□□ □□□□□ □□□□□ nested dictionary □□□□ Real-world □ JSON data, database records, config files □□□□□□□ organized □□□□□□

```
# ─── Nested dictionary example ───
school = {
    "class_10A": {
        "teacher": "Mr. Rahman",
        "students": ["Alice", "Bob", "Charlie"],
        "room": 101
    },
    "class_10B": {
        "teacher": "Ms. Fatima",
        "students": ["Dave", "Eve"],
        "room": 102
    }
}

# Accessing nested values
print(school["class_10A"]["teacher"])           # Mr. Rahman
print(school["class_10B"]["students"][0])       # Dave
print(len(school["class_10A"]["students"]))     # 3
```

```
# Modifying nested
school["class_10A"]["students"].append("Frank")
school["class_10A"]["room"] = 105

# Iterating nested dict
for class_name, info in school.items():
    print(f"\nClass: {class_name}")
    print(f"  Teacher : {info['teacher']}")
    print(f"  Students: {', '.join(info['students'])}")
    print(f"  Room    : {info['room']}")

# ── Real-world: Student database ──
students_db = {
    "S001": {"name": "Alice", "marks": [85, 92, 78], "grade": "A"},
    "S002": {"name": "Bob",   "marks": [70, 65, 80], "grade": "B"},
    "S003": {"name": "Carol", "marks": [95, 98, 92], "grade": "A+"},
}

for sid, data in students_db.items():
    avg = sum(data["marks"]) / len(data["marks"])
    print(f"{sid} | {data['name']:<10} | Avg: {avg:.1f} | Grade: {data['grade']}")
```

□ Interview Q: Dictionary □ key □□□□□□□ list □□□ use □□□ □□□□□ □□?

✅Answer: Dictionary □ key must be hashable (immutable)□ list mutable — □□□□ list □□ modify □□□□□ □□□□□, □□□ □□ hash change □□□□□ □□□□□, □□ dictionary □□ internal hash table □□ corrupt □□□□□ tuple immutable □□□ tuple key □□□ □□□□□: d[(1,2)] = 'ok'□ □□□□□□□ d[[1,2]] = 'ok' □□□□□ TypeError: unhashable type: 'list' □□□□□□

□ Interview Q: dict.get() □□□ dict[key] □□ □□□□□□ □□□□□□□□□□ □□? □□□ □□□□□□ □□□□□□□□ □□□□□?

✅Answer: dict[key] key □□ □□□□□□ KeyError raise □□□ — program crash □□□□□ dict.get(key, default) key □□ □□□□□□ None □□ custom default return □□□ — safe□ Rule: □□□ □□□□□ 100% □□□□□□□□ □□ key □□□, □□□□□□ [] □□□□□□□□ □□□□□ □□□ key absent □□□□□□ possible □□□, □□□□□□□ .get() □□□□□□□□ □□□□□ Production code □ .get() prefer □□□ □□□□□

□ Interview Q: Python dictionary □□□□□□□ internally □□□ □□□□?

✅Answer: Dictionary hash table □□□□□□□□ □□□□□ key □□ hash() function □ □□□□□ □□□□□ integer □□□□□□□□ □□□□□□ □□ hash □□□□□□ memory □□ specific slot □□□□□□ □□□□□ □□□□ lookup O(1) — size □□□ □□□□□ □□□□□ key □□ same hash □□□ 'hash collision' □□□ — Python open addressing □□□□□□ □□□ handle □□□□□ □□ □□□□□□□ key must be hashable (immutable)□

SECTION 2: Sets — Unique Collection

2.1 What is a Set?

Set □□□ unordered collection of UNIQUE elements□ □□□□ duplicate value □□□□ □□ — automatically remove □□□ □□□□□□ Mathematics □□ set theory □□□□ □□ concept □□□□□□□ Set □ union, intersection, difference □□ □□□ mathematical operations □□□ □□□□□□

Property	Explanation
Unordered	Items □□ □□□□ fixed order □□□ — indexing □□□ □□□□ □□
Unique	Duplicate automatically remove □□□
Mutable	Items add/remove □□□ □□□□, □□□□□□ items □□□□□□□ immutable □□□ □□□
No indexing	set[0] □□□ □□□ □□ — for loop □□□□□ iterate □□□□ □□□
Fast lookup	x in my_set — O(1), list □□ O(n) □□ □□□□□ □□□□ □□□□□

2.2 Creating Sets

```
# ── Method 1: Curly braces ──
fruits = {"apple", "banana", "cherry"}
print(fruits)   # unordered output — order may vary

# ── Method 2: set() constructor ──
numbers = set([1, 2, 3, 2, 1, 4])   # duplicates removed!
print(numbers)   # {1, 2, 3, 4}

# ── Creating from string ──
chars = set("banana")
print(chars)    # {'b', 'a', 'n'} — unique characters only

# ── IMPORTANT: empty set — must use set(), NOT {} ──
empty_set  = set()      # ✓ correct
empty_dict = {}         # ✗ this is a dict, not a set!
print(type(empty_set))  # <class 'set'>
print(type(empty_dict)) # <class 'dict'>

# ── frozenset — immutable set ──
frozen = frozenset([1, 2, 3])
# frozen.add(4)    # ✗ AttributeError — cannot modify
print(frozen)      # frozenset({1, 2, 3})
```

2.3 Set Operations

```
a = {1, 2, 3, 4, 5}
b = {4, 5, 6, 7, 8}

# ─── Union: □□ elements (duplicates exclude) ───
print(a | b)            # {1, 2, 3, 4, 5, 6, 7, 8}
print(a.union(b))       # same result

# ─── Intersection: □□□□□ common elements ───
print(a & b)            # {4, 5}
print(a.intersection(b))

# ─── Difference: a □□ □□□ □□□□□□□ b □□ □□□ ───
print(a - b)            # {1, 2, 3}
print(a.difference(b))

# ─── Symmetric Difference: either □ □□□ □□□□□□□ □□□□□□□□ □□□ ───
print(a ^ b)                     # {1, 2, 3, 6, 7, 8}
print(a.symmetric_difference(b))

# ─── Subset / Superset ───
x = {1, 2}
y = {1, 2, 3, 4}
print(x.issubset(y))    # True  — x □□ □□ elements y □□ □□□
print(y.issuperset(x))  # True  — y □□ x □□ □□ elements □□□
print(x <= y)           # True  (subset operator)
print(y >= x)           # True  (superset operator)
print(x.isdisjoint({5, 6}))  # True — □□□□□ common element □□□
```

2.4 Set Methods — Add, Remove, Update

```
s = {1, 2, 3}

# ─── Adding elements ───
s.add(4)           # single element add
print(s)           # {1, 2, 3, 4}
s.add(2)           # already exists — no error, no duplicate
print(s)           # {1, 2, 3, 4}

s.update([5, 6, 7])       # add multiple elements
s.update({8}, [9, 10])    # add from multiple iterables

# ─── Removing elements ───
s.remove(10)       # removes 10; raises KeyError if not found
s.discard(99)      # removes 99; NO error if not found ← safer!
popped = s.pop()   # removes & returns RANDOM element
s.clear()          # removes ALL elements

# ─── Checking membership — O(1) ───
fruits = {"apple", "banana", "cherry"}
print("apple" in fruits)   # True
print("grape" in fruits)   # False

# ─── Real-world use: finding duplicates in a list ───
```

```
data = [1, 2, 3, 2, 4, 3, 5, 1]
unique = list(set(data))          # remove duplicates
duplicates = [x for x in data if data.count(x) > 1]
print("Unique:", unique)
print("Has duplicates:", len(data) != len(set(data)))
```

2.5 When to Use Which Data Structure?

Structure	□□□ □□□□□□□□ □□□□
`list`	Order matters + duplicates allowed + index □□□□□□ e.g. shopping cart items, steps in order
`tuple`	Order matters + immutable data□ e.g. coordinates (x,y), RGB (255,0,0), database row
`dict`	Key □□□□□ value □□□□□□ □□□□ e.g. phone book, word frequency, config settings
`set`	Unique elements □□□□□ + fast membership check□ e.g. unique visitors, tag list, deduplication

```
# ─── Practical comparison ───

# Scenario 1: Check if username already taken
# ✕ Bad — O(n) for each check
taken_usernames_list = ["alice", "bob", "charlie", ...]
if "alice" in taken_usernames_list:  # slow for large lists
    pass

# ✅ Good — O(1) always
taken_usernames_set = {"alice", "bob", "charlie"}
if "alice" in taken_usernames_set:   # fast!
    pass

# Scenario 2: Unique words in a document
words = "the cat sat on the mat the cat".split()
unique_words = set(words)
print(f"Total: {len(words)}, Unique: {len(unique_words)}")

# Scenario 3: Common friends (intersection)
alice_friends = {"Bob", "Charlie", "Dave", "Eve"}
bob_friends   = {"Alice", "Charlie", "Frank", "Eve"}
common = alice_friends & bob_friends
print("Common friends:", common)    # {'Charlie', 'Eve'}
```

□ Interview Q: Set □ □□□ indexing □□□ □□□□ □□?

✅Answer: Set internally hash table □□□□□□□ □□□□ Elements □□□□ specific position □ □□□□ □□□ □□ — hash value □□□□□□□□ random slot □ □□□□□ □□□ 'first element', 'second element' □□□ □□□□ □□□□ Iteration □□□ □□□□ □□□□□□ order guaranteed □□□ □□□ ordered unique collection □□□□□ □□□, list(dict.fromkeys(lst)) □□□□ Python 3.7+ □ insertion-order dict □□□□□□□ □□□□

□ Interview Q: set.remove() □□□ set.discard() □□ □□□□□□□□□ □□?

✅Answer: remove(x) — x □□ □□□□□□ KeyError raise □□□□ discard(x) — x □□ □□□□□□ □□□□□□ □□□ □□, □□□□□ error □□□□□ Rule: □□□ □□□□□□□□ □□ element □□□ □□□□ remove(), □□□ □□□ □□□□□ □□□□□ □□ □□□ discard() □□□□□□□□ □□□□□ list □□ remove() vs set □□ discard() □□ □□□□□□ Production code □ discard() safer□

□ Interview Q: List □ duplicate remove □□□□□ □□□□□□□□□ □□□□□ □□□□□□ □□?

✅Answer: Fast □□□□□□□ order lost: list(set(original_list))□ Order preserved (Python 3.7+): list(dict.fromkeys(original_list))□ dict.fromkeys() □□□□□ dict □□□□□ □□□ □□□□□□□ list □□ items keys □□□ — keys unique □□□□□□□□ duplicates automatically remove □□□, □□□ insertion order preserved □□□□□ Benchmark □ dict.fromkeys() set approach □□ □□□□□□ □□□□□□□ □□□ □□□□□□□ order preserve □□□□

SECTION 3: List Comprehensions

3.1 What is a List Comprehension?

List comprehension □□□ Python □□ □□□□ elegant □□□ concise way to create lists□ □□□ □□□□ single line □ for loop □□□ optional condition combine □□□ □□□□ □□□□ list □□□□ □□□□ Pythonic code □□□□□ □□□□□□ □□□□□□□□□□□□ skill□

□ **Syntax**

result = [expression for item in iterable]
result = [expression for item in iterable if condition]
result = [expression for item in iterable if condition else other_expr]

□□□ □□□□ □□□ □□□□ □□□□: 'expression □□□□□□□□ item □□ □□□□ □□□□□□ item iterable □ □□□ □□□ condition □□□□ □□□'

3.2 Basic List Comprehensions

```
# ─── Traditional loop vs comprehension ───

# Traditional: squares of 1-10
squares = []
for i in range(1, 11):
    squares.append(i ** 2)
print(squares)   # [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

# Comprehension: same result, one line!
squares = [i ** 2 for i in range(1, 11)]
print(squares)   # [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

# ─── More examples ───
# Double all values
nums = [1, 2, 3, 4, 5]
doubled = [n * 2 for n in nums]          # [2, 4, 6, 8, 10]

# Convert Celsius to Fahrenheit
celsius    = [0, 20, 37, 100]
fahrenheit = [(c * 9/5) + 32 for c in celsius]
print(fahrenheit)   # [32.0, 68.0, 98.6, 212.0]

# Uppercase all strings
words = ["hello", "world", "python"]
upper = [w.upper() for w in words]
print(upper)   # ['HELLO', 'WORLD', 'PYTHON']

# Length of each word
lengths = [len(w) for w in words]
print(lengths)   # [5, 5, 6]
```

```
# Strip whitespace from list of strings
raw = ["  Alice  ", "  Bob ", " Carol"]
cleaned = [name.strip() for name in raw]
print(cleaned)   # ['Alice', 'Bob', 'Carol']
```

3.3 Conditional List Comprehensions

```
# ─── Filter: if condition ───
nums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# Only even numbers
evens = [n for n in nums if n % 2 == 0]
print(evens)   # [2, 4, 6, 8, 10]

# Only positive numbers from mixed list
mixed = [-3, -1, 0, 2, 5, -7, 8]
positives = [n for n in mixed if n > 0]
print(positives)   # [2, 5, 8]

# Words longer than 4 characters
words = ["cat", "elephant", "dog", "python", "ant"]
long_words = [w for w in words if len(w) > 4]
print(long_words)   # ['elephant', 'python']

# Filter out None values
data = [1, None, 2, None, 3, 4, None]
clean = [x for x in data if x is not None]
print(clean)   # [1, 2, 3, 4]

# ─── Transform with condition: if-else ───
# Note: if-else goes BEFORE the for, filter if goes AFTER

# Label each number as "even" or "odd"
labels = ["even" if n % 2 == 0 else "odd" for n in range(1, 8)]
print(labels)   # ['odd', 'even', 'odd', 'even', 'odd', 'even', 'odd']

# Grade labeling
scores = [85, 42, 91, 67, 55, 78]
grades = ["Pass" if s >= 60 else "Fail" for s in scores]
print(grades)   # ['Pass', 'Fail', 'Pass', 'Pass', 'Fail', 'Pass']

# Clamp values between 0 and 100
raw_scores = [-5, 45, 105, 78, -2, 100]
clamped = [max(0, min(100, s)) for s in raw_scores]
print(clamped)   # [0, 45, 100, 78, 0, 100]
```

3.4 Nested List Comprehensions

```
# ─── Flattening a 2D list ───
```

```python
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

# Traditional nested loop
flat = []
for row in matrix:
    for val in row:
        flat.append(val)

# Comprehension version
flat = [val for row in matrix for val in row]
print(flat)   # [1, 2, 3, 4, 5, 6, 7, 8, 9]

# ─── Creating a matrix ───
# 3x3 zero matrix
zeros = [[0 for _ in range(3)] for _ in range(3)]
print(zeros)   # [[0, 0, 0], [0, 0, 0], [0, 0, 0]]

# Multiplication table (3x3)
table = [[i * j for j in range(1, 4)] for i in range(1, 4)]
for row in table:
    print(row)
# [1, 2, 3]
# [2, 4, 6]
# [3, 6, 9]

# ─── Nested with condition ───
# Only pairs where both numbers are even
pairs = [(x, y) for x in range(1, 5) for y in range(1, 5)
         if x % 2 == 0 and y % 2 == 0]
print(pairs)   # [(2, 2), (2, 4), (4, 2), (4, 4)]
```

3.5 Dictionary Comprehensions

```python
# ─── Syntax: {key_expr: val_expr for item in iterable} ───

# Square of each number as key-value
squares = {n: n**2 for n in range(1, 6)}
print(squares)   # {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}

# Word length dictionary
words = ["Python", "is", "awesome"]
word_len = {word: len(word) for word in words}
print(word_len)   # {'Python': 6, 'is': 2, 'awesome': 7}

# Flip keys and values (invert a dict)
original = {"a": 1, "b": 2, "c": 3}
inverted = {v: k for k, v in original.items()}
print(inverted)   # {1: 'a', 2: 'b', 3: 'c'}

# Dict comprehension with condition — filter a dict
scores = {"Alice": 85, "Bob": 42, "Carol": 91, "Dave": 58}
passed = {name: score for name, score in scores.items() if score >= 60}
print(passed)   # {'Alice': 85, 'Carol': 91}
```

```
# Uppercase keys
data = {"name": "alice", "city": "dhaka"}
upper_keys = {k.upper(): v for k, v in data.items()}
print(upper_keys)    # {'NAME': 'alice', 'CITY': 'dhaka'}

# ─── Set comprehension ───
# {expr for item in iterable}
unique_lengths = {len(w) for w in ["cat", "dog", "elephant", "ant"]}
print(unique_lengths)    # {3, 8} — unique lengths only
```

□ Interview Q: List comprehension □□ □□□□□□□ for loop □□ □□□□□□ □□□□□?

✅Answer: List comprehension □□□□□□□□ 30–50% □□□□□□ □□□□□ □□□ C level □ optimized□ □□□□□□□ □□□□□□□ □□□□□□□□ □□□ □□□□□ □□□□□ Simple, one-liner transformation □□ □□□□□ comprehension □□□□□□ Complex logic (multiple conditions, side effects, multiple steps) □□□□□□ regular for loop □□□□□ readable□ Rule of thumb: □□□ comprehension □□□□□□ □□□□□□□□ □□□, for loop □□□□□□ Readability > cleverness□

□ Interview Q: [expression if cond else other for x in lst] □□□ [expression for x in lst if cond] □□ □□□□□□□□□□ □□?

✅Answer: if-else BEFORE for: □□ elements process □□□, condition true □□□ expression, false □□□ other□ □□□ transform□ [n*2 if n>0 else 0 for n in lst]. if AFTER for: condition false □□□ element skip □□□ — filter□ [n for n in lst if n>0]. □□□ □□□ □□□□□□ □□□□□□: if-else □□□ = transform/replace, if □□□ = filter/exclude□

SECTION 4: Working with Complex Data

4.1 Nested Data Structures

Real-world data □□□□□□□ □□□□□□□ complex □□□ nested □□□□ JSON API response, database records, configuration files — □□ □□□□□□ nested structures □□□□□□□□ □□□□ Python □ list, dict, tuple □□□□□□□ combination □ nest □□□ □□□□□□

```
# ─── Common nested patterns ───

# 1. List of lists (2D grid/matrix)
chessboard = [
    ["R", "N", "B", "Q", "K", "B", "N", "R"],
    ["P", "P", "P", "P", "P", "P", "P", "P"],
    [".", ".", ".", ".", ".", ".", ".", "."],
]
print(chessboard[0][3])   # Q (row 0, column 3)

# 2. List of tuples (database rows)
employees = [
    ("E001", "Alice",   "Engineering", 75000),
    ("E002", "Bob",     "Marketing",   65000),
    ("E003", "Charlie", "Engineering", 80000),
]
for emp_id, name, dept, salary in employees:
    print(f"{emp_id}: {name} — {dept} — Tk {salary:,}")

# 3. Dict of lists
schedule = {
    "Monday":    ["Math", "English", "Science"],
    "Tuesday":   ["History", "Math", "Art"],
    "Wednesday": ["Science", "PE", "English"],
}
for day, subjects in schedule.items():
    print(f"{day}: {', '.join(subjects)}")

# 4. Dict of dicts (most common in real apps)
inventory = {
    "laptop": {"price": 45000, "stock": 10, "brand": "Dell"},
    "phone":  {"price": 25000, "stock": 50, "brand": "Samsung"},
    "tablet": {"price": 30000, "stock": 0,  "brand": "Apple"},
}
for item, details in inventory.items():
    status = "In Stock" if details["stock"] > 0 else "Out of Stock"
    print(f"{item}: Tk {details['price']:,} [{status}]")
```

4.2 Lists of Dictionaries — Most Common Pattern

API □□□□□ data, database query result, CSV file — □□ □□□□□ □□□□□□□□ 'list of dicts' format □ □□□□□ □□□□□□□□ dict □□□□ record/row represent □□□□

```python
students = [
    {"name": "Alice",   "age": 20, "marks": 88, "city": "Dhaka"},
    {"name": "Bob",     "age": 22, "marks": 75, "city": "Chittagong"},
    {"name": "Charlie", "age": 21, "marks": 92, "city": "Dhaka"},
    {"name": "Dave",    "age": 19, "marks": 65, "city": "Sylhet"},
    {"name": "Eve",     "age": 22, "marks": 55, "city": "Dhaka"},
]

# ─── Accessing data ───
print(students[0]["name"])      # Alice
print(students[2]["marks"])     # 92

# ─── Iterating ───
for student in students:
    print(f"{student['name']}: {student['marks']}")

# ─── Filtering — students from Dhaka ───
dhaka_students = [s for s in students if s["city"] == "Dhaka"]
print("Dhaka:", [s["name"] for s in dhaka_students])

# ─── Sorting by marks (descending) ───
sorted_students = sorted(students, key=lambda s: s["marks"], reverse=True)
for rank, s in enumerate(sorted_students, 1):
    print(f"Rank {rank}: {s['name']} — {s['marks']}")

# ─── Statistics ───
all_marks = [s["marks"] for s in students]
print(f"Highest : {max(all_marks)}")
print(f"Lowest  : {min(all_marks)}")
print(f"Average : {sum(all_marks)/len(all_marks):.1f}")

# ─── Adding a new field ───
for student in students:
    student["grade"] = "Pass" if student["marks"] >= 60 else "Fail"

# ─── Searching ───
target = "Charlie"
result = next((s for s in students if s["name"] == target), None)
if result:
    print(f"Found: {result}")
```

4.3 Dictionaries of Lists

```python
# ─── Grouping data by category ───
# Example: Group students by city

students = [
    {"name": "Alice",   "city": "Dhaka"},
    {"name": "Bob",     "city": "Chittagong"},
    {"name": "Charlie", "city": "Dhaka"},
    {"name": "Dave",    "city": "Sylhet"},
    {"name": "Eve",     "city": "Dhaka"},
]
```

```
# Group students by city
city_groups = {}
for student in students:
    city = student["city"]
    if city not in city_groups:
        city_groups[city] = []        # initialize empty list
    city_groups[city].append(student["name"])

print(city_groups)
# {'Dhaka': ['Alice', 'Charlie', 'Eve'], 'Chittagong': ['Bob'], 'Sylhet':
['Dave']}

# Cleaner with setdefault()
city_groups2 = {}
for student in students:
    city_groups2.setdefault(student["city"], []).append(student["name"])

# Or with defaultdict (from collections module)
from collections import defaultdict
city_groups3 = defaultdict(list)
for student in students:
    city_groups3[student["city"]].append(student["name"])

# ─── Accessing grouped data ───
for city, names in city_groups.items():
    print(f"{city} ({len(names)} students): {', '.join(names)}")
```

□ Interview Q: List of dicts □ specific key □□□□□□ □□□□□□□ sort □□□□□?

✅Answer: sorted(lst, key=lambda x: x['key_name']) □□□□□□□□ □□□□□ Descending □□ □□□□□ reverse=True add □□□□□ Multiple keys: key=lambda x: (x['dept'], x['salary'])□ In-place sort □□□□□□ lst.sort(key=lambda x: x['marks'])□ Python □□ sort stable — same value □□ elements □□ relative order change □□□ □□□

SECTION 5: Functions — Reusable Blocks of Code

5.1 What is a Function and Why Use It?

Function □□□ named block of code □□□□□ □□□□□ specific task perform □□□ □□□ □□□ □□□□□□ □□□ call □□□ □□□□□□ Function □□□□□□ □□□ code □□□□□□□ □□□□□□ □□□□ — □□□□□□ □□□ code duplication□ Functions □□ □□□□□□□□ DRY principle follow □□□ □□□: Don't Repeat Yourself□

Benefit	Explanation
Reusability	□□□□□□ □□□□□, □□□□□□ □□□□□ call □□□ — code duplication □□□□□□
Modularity	□□□ problem □□ □□□ □□□ pieces □ □□□ □□□ □□□□□□□ □□□
Readability	main() code clean □□□□□ — details function □ □□□□□□□
Testability	□□□ functions individually test □□□ □□□
Abstraction	User □□ '□□ □□□' □□□□□□ □□□ □□, □□□□□ '□□ □□□' call □□□□□□ □□□

5.2 Defining and Calling Functions

```python
# ─── Basic function syntax ───
# def keyword → function_name → () → colon → indented body

def greet():
    """This is a docstring — explains what the function does."""
    print("Hello, World!")

# Call the function
greet()         # Hello, World!
greet()         # Hello, World! (reusable!)
greet()         # Hello, World!

# ─── Function with parameters ───
def greet_person(name):
    print(f"Hello, {name}!")

greet_person("Alice")   # Hello, Alice!
greet_person("Bob")     # Hello, Bob!

# ─── Function with multiple parameters ───
def add(a, b):
    result = a + b
    print(f"{a} + {b} = {result}")

add(3, 5)    # 3 + 5 = 8
add(10, 20)  # 10 + 20 = 30
```

5.3 Return Values

return statement function □□□□ □□□□ value □□□ □□□ □□□□□ return □□□□□ function None return □□□□ return □□ □□ function □□ execution □□□□ □□□□ □□□□□

```python
# ─── Single return value ───
def square(n):
    return n ** 2

result = square(5)
print(result)          # 25
print(square(7) + 1)   # 50 — directly use in expression

# ─── Multiple return values (as tuple) ───
def min_max(numbers):
    return min(numbers), max(numbers)   # returns a tuple

low, high = min_max([3, 1, 7, 2, 9, 4])
print(f"Min: {low}, Max: {high}")   # Min: 1, Max: 9

# Can also unpack manually
result = min_max([3, 1, 7])
print(result)        # (1, 7) — tuple
print(result[0])     # 1

# ─── Early return ───
def is_even(n):
    if n % 2 == 0:
        return True    # exits here if even
    return False       # only reaches here if odd
    # NOTE: this is same as: return n % 2 == 0

# ─── Return in the middle of loops ───
def find_first_negative(numbers):
    for n in numbers:
        if n < 0:
            return n     # exit immediately when found
    return None          # not found

print(find_first_negative([1, 2, -3, 4, -5]))  # -3
print(find_first_negative([1, 2, 3, 4]))       # None
```

5.4 Parameters and Arguments — Types

```python
# ─── 1. Positional arguments (most common) ───
def describe(name, age, city):
    print(f"{name}, {age} years old, from {city}")

describe("Alice", 22, "Dhaka")     # positional — order matters!

# ─── 2. Keyword arguments ───
describe(age=22, city="Dhaka", name="Alice")  # order doesn't matter
```

```
# ─── 3. Default parameters ───
def power(base, exponent=2):    # exponent defaults to 2
    return base ** exponent

print(power(3))       # 9  (3^2 — default exponent)
print(power(3, 3))    # 27 (3^3 — override default)
print(power(2, 10))   # 1024

# ─── IMPORTANT: Default must come AFTER non-default ───
# def wrong(a=1, b):   # ✕ SyntaxError!
# def correct(a, b=1): # ✓ OK

# ─── 4. Mix of positional and keyword ───
def create_user(name, age, role="user", active=True):
    return {"name": name, "age": age, "role": role, "active": active}

u1 = create_user("Alice", 22)                        # both defaults used
u2 = create_user("Bob", 30, role="admin")            # override role
u3 = create_user("Carol", 25, "moderator", False) # positional override
```

5.5 Docstrings — Documenting Functions

```
# ─── Single-line docstring ───
def add(a, b):
    """Return the sum of a and b."""
    return a + b

# ─── Multi-line docstring (Google style) ───
def calculate_bmi(weight_kg, height_m):
    """
    Calculate Body Mass Index (BMI).

    Args:
        weight_kg (float): Weight in kilograms
        height_m  (float): Height in meters

    Returns:
        float: BMI value rounded to 2 decimal places

    Examples:
        >>> calculate_bmi(70, 1.75)
        22.86
    """
    bmi = weight_kg / (height_m ** 2)
    return round(bmi, 2)

# Access docstring
print(calculate_bmi.__doc__)
help(calculate_bmi)    # shows formatted docstring

result = calculate_bmi(70, 1.75)
print(f"BMI: {result}")
```

□ Interview Q: Function □ return statement □□ □□□□ □□ return □□□?

✅Answer: Python □ □□ function implicitly None return □□□ □□□ □□□□ return statement □□ □□□□ □□ return □□□□ □□□ □□□□ (without value)□ □□□ result = my_func() □□□□ result □□ value □□□ None□ □□□ common mistake — print() □□□□□ None return □□□□ result = print('hello') □□□□ 'hello' print □□□ □□□□□□ result □□□ None□

□ Interview Q: Default mutable argument □□ □□ □□□□□□ □□□ □□□□?

✅Answer: Python □ default argument □□□□□ evaluate □□□ — function call □□□□□□ □□□□ □□□, define □□□□□□ □□□□□ □□□ def add_item(item, lst=[]) □□□□ □□ calls □□□ list share □□□! Bug: add_item(1) → [1], add_item(2) → [1, 2]! □□□□ pattern: def add_item(item, lst=None): if lst is None: lst = []. □□□ Python □ classic gotcha□

SECTION 6: Advanced Function Concepts

6.1 Scope — Local vs Global Variables

Scope □□□□ □□□ □□□□□□□ variable □□ accessible□ Python □ LEGB Rule follow □□□ variable □□□□□ □□□: Local → Enclosing → Global → Built-in□

Scope Level	Explanation
`Local`	Function □□ □□□□□ define □□□ variable — □□□□ □□□ function □ accessible
`Enclosing`	Nested function □□ outer function □□ variable (nonlocal)
`Global`	Module level □ define □□□ variable — □□□□□□□ readable
`Built-in`	Python □□ built-in names: len, print, range, etc.

```
# ─── Local scope ───
def my_func():
    x = 10        # local variable
    print(x)      # 10 — OK inside function

my_func()
# print(x)      # ✗ NameError — x not accessible outside

# ─── Global scope ───
counter = 0       # global variable

def show_counter():
    print(counter)    # ✓ can READ global variable

def increment():
    global counter    # declare intent to MODIFY global
    counter += 1

show_counter()    # 0
increment()
increment()
show_counter()    # 2

# ─── Local shadows global ───
name = "Global Alice"

def greet():
    name = "Local Bob"     # creates new local variable
    print(name)            # Local Bob — local takes priority

greet()
print(name)    # Global Alice — global unchanged

# ─── nonlocal — for nested functions ───
```

```
def outer():
    count = 0

    def inner():
        nonlocal count   # access outer function's variable
        count += 1
        print(f"Inner count: {count}")

    inner()    # 1
    inner()    # 2
    print(f"Outer count: {count}")   # 2

outer()
```

□ Best Practice: Avoid Global Variables

Global variables make code hard to debug and test.
Functions should ideally work only with their parameters and local variables.
If you need to share state, use return values or pass data as arguments.
global keyword: use sparingly and only when truly necessary.

6.2 Lambda Functions — Anonymous Functions

Lambda □□□ small, anonymous (nameless) function □□ single expression evaluate □□□□ Complex logic □□ □□□□ □□□ — simple one-liner operations □□ □□□□□ sorted(), map(), filter() □□ □□□□ □□□□□□□□ □□□□ □□□□□□□□ □□□□

```
# ─── Syntax: lambda arguments: expression ───
# lambda □□□□□□□ □□□□□□ expression — no statements, no multiple lines

# Regular function vs lambda
def square(x):
    return x ** 2

square_lambda = lambda x: x ** 2

print(square(5))         # 25
print(square_lambda(5))  # 25

# ─── Lambda with multiple arguments ───
add   = lambda a, b: a + b
mul   = lambda a, b, c: a * b * c
clamp = lambda val, lo, hi: max(lo, min(hi, val))

print(add(3, 5))          # 8
print(mul(2, 3, 4))       # 24
print(clamp(150, 0, 100)) # 100

# ─── Lambda with condition ───
is_even = lambda n: n % 2 == 0
```

```
grade   = lambda s: "Pass" if s >= 60 else "Fail"

print(is_even(4))    # True
print(grade(75))     # Pass

# ─── Lambda in sorted() ───
students = [
    {"name": "Alice", "marks": 88},
    {"name": "Bob",   "marks": 75},
    {"name": "Carol", "marks": 92},
]

# Sort by marks
by_marks = sorted(students, key=lambda s: s["marks"])
by_name  = sorted(students, key=lambda s: s["name"])

# Sort by multiple criteria: first by dept, then salary
employees = [("Alice","Eng",70000), ("Bob","HR",60000), ("Carol","Eng",80000)]
sorted_emp = sorted(employees, key=lambda e: (e[1], -e[2]))  # dept asc, salary
desc
print(sorted_emp)
```

6.3 *args and **kwargs

□□□□ □□□□ □□□□ □□□□ □□ function □ □□□□ arguments □□□□□ *args □□□ **kwargs □□□□□ variable number of arguments handle □□□ □□□□□

```
# ─── *args — variable positional arguments ───
# args □□□ □□□□ TUPLE of extra positional arguments

def my_sum(*args):
    print(type(args))   # <class 'tuple'>
    print(args)         # (1, 2, 3, 4, 5)
    return sum(args)

print(my_sum(1, 2))         # 3
print(my_sum(1, 2, 3, 4))   # 10
print(my_sum())              # 0

# ─── Mix: regular + *args ───
def greet_all(greeting, *names):
    for name in names:
        print(f"{greeting}, {name}!")

greet_all("Hello", "Alice", "Bob", "Carol")
# Hello, Alice!
# Hello, Bob!
# Hello, Carol!

# ─── **kwargs — variable keyword arguments ───
# kwargs □□□ □□□□ DICT of extra keyword arguments

def describe_person(**kwargs):
    print(type(kwargs))    # <class 'dict'>
    for key, value in kwargs.items():
        print(f"  {key}: {value}")

describe_person(name="Alice", age=22, city="Dhaka", hobby="coding")

# ─── Mix: regular + *args + **kwargs ───
def full_func(required, *args, **kwargs):
    print("Required:", required)
    print("Args:", args)
    print("Kwargs:", kwargs)

full_func("must", 1, 2, 3, color="red", size=10)

# ─── Unpacking with * and ** ───
nums   = [1, 2, 3]
config = {"sep": ", ", "end": "!\n"}

print(*nums)          # 1 2 3 (unpack list as positional args)
print(*nums, **config) # 1, 2, 3! (unpack dict as keyword args)
```

6.4 Recursion

Recursion □□□ □□□ □□□□ function □□□□□ □□□□□□ call □□□□ □□□□□□□□ recursive function □□ □□□□ □□□ □□□□ □□□□□□: Base Case (□□□ □□□□□) □□□ Recursive Case (□□□□□□ call □□□□ □□□)□

□ Recursion □□ □□□□ □□□□□□□□□□ □□□

1. Base Case — □□□ recursion □□ stopping condition□ □□□ □□ □□□□□ infinite recursion □□□□
2. Recursive Case — problem □□ smaller version □ □□□□ □□□□□□ call □□□□ □□□□□□□ recursive call □□ base case □□ □□□□ □□□□□□ □□□□ □□□□

```
# ─── Example 1: Factorial ───
# factorial(5) = 5 * 4 * 3 * 2 * 1 = 120
# factorial(n) = n * factorial(n-1)
# factorial(0) = 1  ← base case

def factorial(n):
    # Base case
    if n == 0 or n == 1:
        return 1
    # Recursive case
    return n * factorial(n - 1)

print(factorial(5))   # 120
print(factorial(10))  # 3628800

# Trace of factorial(4):
# factorial(4) = 4 * factorial(3)
#               = 4 * (3 * factorial(2))
#               = 4 * (3 * (2 * factorial(1)))
#               = 4 * (3 * (2 * 1))
#               = 4 * (3 * 2)
#               = 4 * 6
#               = 24

# ─── Example 2: Fibonacci ───
# fib(0)=0, fib(1)=1, fib(n) = fib(n-1) + fib(n-2)

def fibonacci(n):
    if n <= 0:
        return 0
    if n == 1:
        return 1
    return fibonacci(n - 1) + fibonacci(n - 2)

for i in range(10):
    print(fibonacci(i), end=" ")   # 0 1 1 2 3 5 8 13 21 34

# ─── Example 3: Sum of list ───
def list_sum(numbers):
    if len(numbers) == 0:    # base case: empty list
        return 0
    return numbers[0] + list_sum(numbers[1:])   # head + tail sum

print(list_sum([1, 2, 3, 4, 5]))   # 15

# ─── Example 4: Power ───
def power(base, exp):
    if exp == 0:
        return 1
    return base * power(base, exp - 1)

print(power(2, 10))   # 1024
```

6.5 Higher-Order Functions — map(), filter(), zip()

Higher-order function □□□ function □□□□ □□□□ function □□ argument □□□□□□□ □□□□ □□ return □□□□ Python □□ built-in map() □□□ filter() □□ category □□ □□□□□□

```
# ─── map(function, iterable) ───
# □□□□□□□□ element □ function apply □□□ — map object return □□□

nums = [1, 2, 3, 4, 5]

# Traditional loop
squares = []
for n in nums:
    squares.append(n ** 2)

# map with lambda
squares = list(map(lambda n: n ** 2, nums))
```

```
print(squares)    # [1, 4, 9, 16, 25]

# map with named function
def double(x):
    return x * 2
doubled = list(map(double, nums))
print(doubled)    # [2, 4, 6, 8, 10]

# map with multiple iterables
a = [1, 2, 3]
b = [10, 20, 30]
sums = list(map(lambda x, y: x + y, a, b))
print(sums)    # [11, 22, 33]

# ─── filter(function, iterable) ───
# function True return □□□□ element □□□□, False □□□ □□□□ □□□□

nums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

evens  = list(filter(lambda n: n % 2 == 0, nums))
print(evens)    # [2, 4, 6, 8, 10]

positives = list(filter(lambda n: n > 0, [-3, -1, 0, 2, 5, -7, 8]))
print(positives)  # [2, 5, 8]

# ─── map + filter combined ───
# Square of only even numbers
result = list(map(lambda n: n**2, filter(lambda n: n % 2 == 0, range(1, 11))))
print(result)   # [4, 16, 36, 64, 100]

# Same thing with list comprehension (more readable)
result2 = [n**2 for n in range(1, 11) if n % 2 == 0]
print(result2)  # [4, 16, 36, 64, 100]
```

□ Interview Q: *args □□□ **kwargs □□ □□□□□ □□□□□□□□ □□?

✅Answer: *args positional arguments □□ □□□□ tuple □ collect □□□□ **kwargs keyword arguments □□ □□□□ dict □ collect □□□□ Function signature □ order □□□: def func(positional, *args, keyword_only, **kwargs)□ Call □: func(1, 2, 3, key='val')□ *args □□ name 'args' □□□ **kwargs □□ name 'kwargs' convention, □□□ mandatory □□□ — *numbers, **options □ valid□

□ Interview Q: Recursion vs Iteration — □□□ □□□□□ □□□□□□□ □□□□?

✅Answer: Recursion: problem naturally recursive structure □ □□□□□ (tree traversal, divide & conquer, backtracking)□ Code □□□ □ elegant □□□□□□ stack overhead □□□ — Python default recursion limit 1000□ Iteration: performance critical □□□, large dataset □□□, tail recursion optimize □□□ □□ □□□□□ Python □ Fibonacci □□ recursion exponential time O(2^n) — iteration □□ memoization □□□□□□□ □□□□

□ Interview Q: map() vs list comprehension — □□□□□ □□□□□□□ □□□□?

✅Answer: List comprehension □□□□□□□□ □□□□□□□□ □□□□□□□□□ □□□□ □□□ □□□□ readable □□□ Pythonic□ [n**2 for n in nums] □□□□ list(map(lambda n: n**2, nums))□ map()

□□□□□□□ □□□ □□□: already named function □□□ (map(str, nums)), multiple iterables □□□ (map(func, list1, list2)), □□ lazy evaluation □□□□□ (map object □□□□□□ □□□□ □□□□□□)□

SECTION 7: Key Concepts Deep Dive

7.1 Dictionary vs List — Performance Comparison

```
import time

# ─── Searching: list O(n) vs dict O(1) ───
# Imagine 1 million user records

# List approach — slow
user_list = [{"id": i, "name": f"User{i}"} for i in range(100000)]
# Finding user with id=99999 requires scanning all 100000 items!

# Dict approach — fast
user_dict = {i: f"User{i}" for i in range(100000)}
# Finding user with id=99999 is instant!

# ─── Memory: dict uses more memory than list ───
import sys
my_list = list(range(1000))
my_dict = {i: i for i in range(1000)}
print(f"List memory: {sys.getsizeof(my_list)} bytes")
print(f"Dict memory: {sys.getsizeof(my_dict)} bytes")

# ─── Choosing right structure ───
# Need fast lookup by key?    → dict
# Need ordered sequence?      → list
# Need uniqueness?            → set
# Need immutable sequence?    → tuple
```

7.2 Shallow Copy vs Deep Copy

```
import copy

# ─── The problem with assignment ───
original = [1, 2, [3, 4]]
alias = original          # NOT a copy — same object!
alias[0] = 99
print(original)   # [99, 2, [3, 4]] — BOTH changed!

# ─── Shallow copy — copy outer container only ───
original = [1, 2, [3, 4]]
shallow = original.copy()     # or: list(original) or original[:]
shallow[0] = 99               # changes only shallow
print(original)               # [1, 2, [3, 4]] — outer OK
shallow[2][0] = 99            # BUT nested list is still shared!
print(original)               # [1, 2, [99, 4]] — nested changed!

# ─── Deep copy — completely independent ───
original = [1, 2, [3, 4]]
```

```
deep = copy.deepcopy(original)
deep[2][0] = 99
print(original)   # [1, 2, [3, 4]] — unchanged! completely independent

# ─── Dict copies ───
d = {"a": [1, 2], "b": 3}
shallow_d = d.copy()         # shallow
deep_d    = copy.deepcopy(d) # deep
```

7.3 Comprehension vs Generator — Memory

```
# ─── List comprehension — creates full list in memory ───
squares_list = [n**2 for n in range(1000000)]    # 8MB+ in memory!
print(type(squares_list))   # <class 'list'>

# ─── Generator expression — lazy, one at a time ───
squares_gen = (n**2 for n in range(1000000))     # tiny memory!
print(type(squares_gen))    # <class 'generator'>

# Generator produces values ON DEMAND
import sys
print(sys.getsizeof(squares_list))  # ~8MB
print(sys.getsizeof(squares_gen))   # ~128 bytes!

# Use generator when:
# 1. Large data — don't need all at once
# 2. Streaming/pipeline — one item at a time
# 3. Infinite sequences

# Generator function with yield
def count_up(start, end):
    current = start
    while current <= end:
        yield current      # pause and return value
        current += 1

for n in count_up(1, 5):
    print(n, end=" ")  # 1 2 3 4 5
```

7.4 Common Functional Patterns

```
from functools import reduce

# ─── reduce — fold a list into single value ───
nums = [1, 2, 3, 4, 5]
total    = reduce(lambda acc, x: acc + x, nums)   # 15 (sum)
product  = reduce(lambda acc, x: acc * x, nums)   # 120
maximum  = reduce(lambda a, b: a if a > b else b, nums)  # 5

# ─── any() and all() ───
nums = [2, 4, 6, 8, 10]
```

```
print(all(n % 2 == 0 for n in nums))  # True — all even
print(any(n > 7 for n in nums))       # True — at least one > 7

scores = [85, 90, 78, 92]
print(all(s >= 60 for s in scores))   # True — all pass
print(any(s >= 90 for s in scores))   # True — at least one A

# ─── enumerate() deep dive ───
fruits = ["apple", "banana", "cherry"]
for i, fruit in enumerate(fruits, start=1):
    print(f"{i}. {fruit}")

# ─── zip() deep dive ───
names  = ["Alice", "Bob", "Carol"]
scores = [88, 75, 92]
grades = ["A",  "B",  "A"]

for name, score, grade in zip(names, scores, grades):
    print(f"{name}: {score} ({grade})")

# zip_longest — continue even when lengths differ
from itertools import zip_longest
for a, b in zip_longest([1,2,3], [10,20], fillvalue=0):
    print(a, b)  # 1 10 / 2 20 / 3 0
```

□ Interview Q: Python □ mutable default argument □□ □□ □□□□□□□? Code □□□□□□ □□□□□□□

✅Answer: def append_to(item, lst=[]): lst.append(item); return lst. □□□ append_to(1) → [1], append_to(2) → [1, 2] □□□□□ same list reuse □□□□□□! Fix: def append_to(item, lst=None): if lst is None: lst = []. lst.append(item); return lst. □□□ Python □□ □□□□□□□□ common gotcha □□□ interview □□ □□□□□□□□□ □□□□□□□□□ □□□ □□□□□

□ Interview Q: Python □ first-class function □□□□ □□?

✅Answer: Python □ functions □□□ first-class objects — variable □ assign □□□ □□□□□, function □□ argument □□□□□□□ pass □□□ □□□□□, function □□□□□ return □□□ □□□□□, list/dict □ store □□□ □□□□□□ □□ □□□□□□□ map(func, lst), sorted(lst, key=func) □□□ □□□□ Higher-order functions □□□ decorators □□ concept □□ □□□ built□

SECTION 8: Mini Projects — Day 2 Concepts

□□ projects □□□□ Day 2 □□ concepts □□□□□ □□□□: dictionaries, sets, list comprehensions, complex data□ □□□□□□□□ project □ functions □□□□□□□□ □□□ □□□□□□□ □□□□ Day 2 □□ functions □□□□ □□□□□□□□ Code □□□ □□□□ □□□□□□□□

Project 1: Word Frequency Analyzer

Concept Used	□□□□□□□ □□□□□□□□ □□□□□□□
`Dictionary`	□□□□□□□□ word □□ key, □□□ count □□ value □□□□□□□ store
`Dict comprehension`	□□□□□□□ □□□□□□ N words □□□□ comprehension □□□□□□
`Set`	Unique words count □□□□ set □□□□□□□□
`sorted() + lambda`	Frequency □□□□□□□□□ words sort □□□
`String methods`	.split(), .lower(), .strip() □□□□□ text clean □□□

```
# ══════════════════════════════════════════════
# PROJECT 1: WORD FREQUENCY ANALYZER
# Uses: dict, set, list comprehension, lambda, sorted
# ══════════════════════════════════════════════

print("=" * 50)
print("   □ WORD FREQUENCY ANALYZER")
print("=" * 50)

text = input("Enter a sentence or paragraph:\n> ")

# ─── Clean and tokenize ───
words_raw = text.lower().split()

# Remove punctuation from each word using comprehension
import string
words = [w.strip(string.punctuation) for w in words_raw if
w.strip(string.punctuation)]

if not words:
    print("No valid words found!")
else:
    # ─── Count frequency using dict ───
    freq = {}
    for word in words:
        freq[word] = freq.get(word, 0) + 1

    # ─── Statistics ───
    total_words    = len(words)
    unique_words   = len(set(words))
    most_common    = max(freq, key=lambda k: freq[k])
    print(f"\n□ Statistics:")
    print(f"  Total words  : {total_words}")
    print(f"  Unique words : {unique_words}")
    print(f"  Most common  : '{most_common}' ({freq[most_common]} times)")
    print(f"  Avg frequency: {total_words / unique_words:.1f}")

    # ─── Sort by frequency (descending) ───
    sorted_freq = sorted(freq.items(), key=lambda x: x[1], reverse=True)

    # ─── Top 10 words ───
    top_n = 10
    print(f"\n□ Top {min(top_n, len(sorted_freq))} Most Frequent Words:")
    print("-" * 35)
    print(f"  {'WORD':<20} {'COUNT':>6}  {'BAR'}")
    print("-" * 35)

    for word, count in sorted_freq[:top_n]:
        bar = "█" * count
        print(f"  {word:<20} {count:>6}  {bar}")

    # ─── Words appearing only once ───
    hapax = [w for w, c in freq.items() if c == 1]
    print(f"\nWords appearing only once ({len(hapax)}): {', '.join(hapax[:8])}")
```

Project 2: Student Database with Dict & Comprehensions

```
# ══════════════════════════════════════════════════
# PROJECT 2: STUDENT DATABASE SYSTEM
# Uses: dict of dicts, list comprehension,
#       dict comprehension, lambda, sorted
# ══════════════════════════════════════════════════

# ─── Sample database ───
students = {
    "S001": {"name": "Alice",   "marks": {"Math": 88, "English": 92, "Science":
85}},
    "S002": {"name": "Bob",     "marks": {"Math": 72, "English": 68, "Science":
75}},
    "S003": {"name": "Charlie", "marks": {"Math": 95, "English": 89, "Science":
97}},
    "S004": {"name": "Dave",    "marks": {"Math": 55, "English": 60, "Science":
48}},
    "S005": {"name": "Eve",     "marks": {"Math": 78, "English": 82, "Science":
80}},
}

# ─── Calculate average for each student using dict comprehension ───
averages = {
    sid: sum(data["marks"].values()) / len(data["marks"])
    for sid, data in students.items()
}

# ─── Find grade using comprehension ───
def get_grade(avg):
    if avg >= 90: return "A+"
    elif avg >= 80: return "A"
    elif avg >= 70: return "B"
    elif avg >= 60: return "C"
    else: return "F"

grades = {sid: get_grade(avg) for sid, avg in averages.items()}

# ─── Print report card ───
print("=" * 60)
print(f"  {'ID':<6} {'NAME':<12} {'MATH':>6} {'ENG':>6} {'SCI':>6} {'AVG':>7}
{'GRADE':>6}")
print("=" * 60)

# Sort by average (descending)
sorted_students = sorted(students.items(), key=lambda x: averages[x[0]],
reverse=True)

for rank, (sid, data) in enumerate(sorted_students, 1):
    m = data["marks"]
    avg = averages[sid]
    grade = grades[sid]
    print(f"  {sid:<6} {data['name']:<12} {m['Math']:>6} {m['English']:>6}
{m['Science']:>6} {avg:>7.1f} {grade:>6}")

# ─── Class statistics ───
all_avgs = list(averages.values())
print("=" * 60)
print(f"  Class Average: {sum(all_avgs)/len(all_avgs):.1f}")
print(f"  Highest: {max(all_avgs):.1f} | Lowest: {min(all_avgs):.1f}")

# ─── Filter using comprehension ───
passed = {sid: data["name"] for sid, data in students.items() if averages[sid] >=
60}
failed = {sid: data["name"] for sid, data in students.items() if averages[sid] <
60}

print(f"\n✓ Passed ({len(passed)}): {', '.join(passed.values())}")
print(f"✕ Failed ({len(failed)}): {', '.join(failed.values()) or 'None'}")

# ─── Subject-wise top scorer using dict comprehension ───
subjects = ["Math", "English", "Science"]
top_scorers = {
    sub: max(students.items(), key=lambda x: x[1]["marks"][sub])[1]["name"]
    for sub in subjects
}
print("\n□ Top Scorers by Subject:")
for sub, name in top_scorers.items():
    print(f"  {sub}: {name}")
```

Project 3: Set-Based Tag System

```
# ══════════════════════════════════════════════════
# PROJECT 3: ARTICLE TAG SYSTEM
# Uses: sets, set operations, dict, comprehensions
# Real-world use: blog tags, product categories
# ═══════════════════════════════════════════════════

# ─── Article database ───
articles = {
    "A001": {
        "title": "Python Basics",
        "tags": {"python", "programming", "beginner", "tutorial"}
    },
    "A002": {
        "title": "Web Development with Django",
        "tags": {"python", "django", "web", "backend"}
    },
    "A003": {
        "title": "Machine Learning Intro",
        "tags": {"python", "ml", "data-science", "beginner"}
    },
    "A004": {
        "title": "JavaScript Fundamentals",
        "tags": {"javascript", "programming", "beginner", "frontend"}
    },
    "A005": {
        "title": "React Tutorial",
        "tags": {"javascript", "react", "frontend", "web"}
    },
}

print("=" * 55)
print("   □  ARTICLE TAG SYSTEM")
print("=" * 55)

# ─── All unique tags across all articles ───
all_tags = set()
for article in articles.values():
    all_tags.update(article["tags"])
print(f"\nAll unique tags ({len(all_tags)}): {', '.join(sorted(all_tags))}")

# ─── Find articles by tag ───
search_tag = "beginner"
matching = {aid: data for aid, data in articles.items() if search_tag in
data["tags"]}
print(f"\nArticles tagged '{search_tag}':")
for aid, data in matching.items():
    print(f"  [{aid}] {data['title']}")

# ─── Articles with BOTH python AND beginner tags ───
target_tags = {"python", "beginner"}
both_tagged = [
    data["title"] for data in articles.values()
    if target_tags.issubset(data["tags"])
]
print(f"\nArticles with both {target_tags}:")
for title in both_tagged:
    print(f"  - {title}")

# ─── Tag frequency (how many articles each tag appears in) ───
tag_freq = {}
for article in articles.values():
    for tag in article["tags"]:
        tag_freq[tag] = tag_freq.get(tag, 0) + 1

sorted_tags = sorted(tag_freq.items(), key=lambda x: x[1], reverse=True)
print("\n□ Tag Popularity:")
for tag, count in sorted_tags[:5]:
    bar = "▌" * count
    print(f"  {tag:<15} {bar} ({count})")

# ─── Related articles (shared tags) ───
target_id  = "A001"
target_set = articles[target_id]["tags"]

similarity = {}
for aid, data in articles.items():
    if aid != target_id:
        common = target_set & data["tags"]
        if common:
            similarity[aid] = len(common)

print(f"\n□ Articles related to '{articles[target_id]['title']}':")
for aid, score in sorted(similarity.items(), key=lambda x: x[1], reverse=True):
    print(f"  [{aid}] {articles[aid]['title']} — {score} common tag(s)")
```

SECTION 9: Function Projects

□□ projects □ functions □□□□□□□□ □□□ □□□□□□□□ □□□□□□□□ project □ multiple functions □□□□ □□□□□□□ □□□ □□□□□□□ complex program □□ □□□ □□□ functions □ □□□ □□□□ code □□□ readable □□□ maintainable □□□□

Project 4: Recursive Data Processor

```
# ══════════════════════════════════════════════════
# PROJECT 4: RECURSIVE FUNCTIONS SHOWCASE
# Uses: recursion, functions, *args
# ══════════════════════════════════════════════════

def factorial(n):
    """Calculate n! recursively."""
    if n <= 1:
        return 1
    return n * factorial(n - 1)

def fibonacci_series(n):
    """Generate fibonacci series up to n terms."""
    if n <= 0:
        return []
    if n == 1:
        return [0]
    series = [0, 1]
    for _ in range(2, n):
        series.append(series[-1] + series[-2])
    return series

def power(base, exp):
    """Calculate base^exp recursively."""
    if exp == 0:
        return 1
    if exp < 0:
        return 1 / power(base, -exp)
    return base * power(base, exp - 1)

def flatten(nested_list):
    """Flatten an arbitrarily nested list recursively."""
    result = []
    for item in nested_list:
        if isinstance(item, list):
            result.extend(flatten(item))    # recursive call
        else:
            result.append(item)
    return result

def gcd(a, b):
    """Find Greatest Common Divisor using Euclidean algorithm."""
    if b == 0:
        return a
    return gcd(b, a % b)

# ─── Testing all functions ───
print("=" * 45)
print("   □ RECURSIVE FUNCTIONS DEMO")
print("=" * 45)

# Factorial
for n in [0, 1, 5, 10, 12]:
    print(f"  {n}! = {factorial(n):,}")

# Fibonacci
print(f"\nFibonacci (10 terms): {fibonacci_series(10)}")

# Power
print(f"\n2^10 = {power(2, 10)}")
print(f"3^0  = {power(3, 0)}")

# Flatten
nested = [1, [2, 3], [4, [5, 6]], [[7], 8, 9]]
print(f"\nFlatten: {nested}")
print(f"Result : {flatten(nested)}")

# GCD
pairs = [(48, 18), (100, 75), (17, 13)]
for a, b in pairs:
    print(f"  GCD({a}, {b}) = {gcd(a, b)}")
```

Project 5: Function Toolkit with *args and **kwargs

```
# ══════════════════════════════════════════════════
# PROJECT 5: FLEXIBLE FUNCTION TOOLKIT
# Uses: *args, **kwargs, lambda, map, filter
# ══════════════════════════════════════════════════

def stats(*numbers):
    """Calculate statistics for any number of values."""
    if not numbers:
        return None
    n = len(numbers)
    total = sum(numbers)
    avg = total / n
    sorted_nums = sorted(numbers)
    median = sorted_nums[n // 2] if n % 2 else (sorted_nums[n//2-1] +
sorted_nums[n//2]) / 2
    return {
        "count":  n,
        "sum":    total,
        "avg":    round(avg, 2),
        "min":    min(numbers),
        "max":    max(numbers),
        "median": median
    }
```

```
def format_table(data, **options):
    """
    Format a list of dicts as a text table.

    Args:
        data     : list of dicts
        **options: title, separator, col_width
    """
    title     = options.get("title",     "TABLE")
    separator = options.get("separator", "=")
    col_width = options.get("col_width", 15)

    if not data:
        return "Empty table"

    headers = list(data[0].keys())
    width   = col_width * len(headers) + len(headers) - 1

    lines = []
    lines.append(separator * width)
    lines.append(f" {title.upper()}")
    lines.append(separator * width)
    lines.append("  ".join(f"{h.upper():<{col_width}}" for h in headers))
    lines.append("-" * width)
    for row in data:
        lines.append("  ".join(f"{str(row.get(h,'')):<{col_width}}" for h in
headers))
    lines.append(separator * width)
    return "\n".join(lines)

def apply_transformations(data, *transforms):
    """Apply a chain of transformation functions to data."""
    result = data
    for transform in transforms:
        result = transform(result)
    return result

# ─── Testing ───
print(stats(10, 20, 30, 40, 50))
print(stats(7, 3, 1, 9, 5, 2, 8, 4, 6))

students = [
    {"name": "Alice", "grade": "A", "marks": 92},
    {"name": "Bob",   "grade": "B", "marks": 78},
    {"name": "Carol", "grade": "A", "marks": 88},
]
print(format_table(students, title="Student Results", col_width=12))
print(format_table(students, title="Results", separator="*", col_width=14))

# Chain transformations
result = apply_transformations(
    [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
    lambda lst: [x for x in lst if x % 2 == 0],  # keep evens
    lambda lst: [x ** 2 for x in lst],            # square them
    lambda lst: [x for x in lst if x > 20],       # keep > 20
)
print("\nTransformation result:", result)  # [36, 64, 100]
```

SECTION 10: Interview Questions — Complete Day 2

Day 2 □□ □□ topics □□ □□□ □□□□□□□□ □□□□ □□□□□□□□ □□□ interview questions □□□ detailed answers□

Dictionaries

□ Interview Q: Python dictionary □□ ordered? Python 2 □ □□ □□□?

✅Answer: Python 3.7+ □□□□□ dictionary insertion order guarantee □□□ — official language specification □□ Python 3.6 □ CPython implementation □ □□□ □□□□□□□ guarantee □□□ □□□ Python 2 □□ dictionary completely unordered □□□□ □□□ order □□□□□□□□□ □□□, Python 3.7+ □ regular dict □□□□□□□□ □□□□ Older code □ collections.OrderedDict □□□□□□ □□□□□□

□ Interview Q: dict.items() □□ return □□□? □□□ □□ □□□□□ list?

✅Answer: dict.items() □□□□□ view object return □□□ — dict_items type □□□ □□□ list □□□, □□□□□□□ iterable□ View □□□□ □□□ original dict □□ □□□□□ live window — dict change □□□ view automatically update □□□□ List □ convert □□□□: list(d.items())□ Tuple □□□□□□□ unpack □□□ □□□□: for k, v in d.items()□

□ Interview Q: Dictionary merge □□□□ □□ □□□□□□ □□ □□?

✅Answer: 1. d1.update(d2) — d1 □□ in-place modify □□□□ 2. merged = {**d1, **d2} — Python 3.5+, new dict create □□□□ 3. merged = d1 | d2 — Python 3.9+, cleanest syntax□ 4. merged = dict(d1, **d2)□ Conflict □□□ right side (d2) □□□□□

Sets

□ Interview Q: frozenset □□ □□□ □□□ □□□□□□□□ □□□□?

✅Answer: frozenset □□□ immutable set□ □□□□□□ create □□□□ add/remove □□□ □□□□ □□□ □□□□: immutable □□□□□□□□ hashable — dictionary key □□□□□□□ □□ □□□□ set □□ element □□□□□□□ □□□□□□□□ □□□ □□□□□□ Use case: set of sets, immutable tag collections, caching purposes□ frozenset({1,2,3}) | frozenset({3,4,5}) — operations □□□ □□□, □□□□ modification □□□□

□ Interview Q: Set □□ intersection_update() □□□ intersection() □□ □□□□□□□□□?

✅Answer: intersection() □□□□ set return □□□, original unchanged□ intersection_update() original set □□ in-place modify □□□, None return □□□□ Same pattern: difference()/difference_update(), union()/update(), symmetric_difference()/symmetric_difference_update()□ update() = |=, intersection_update() = &=, difference_update() = -=□

List Comprehensions

□ Interview Q: List comprehension □□□ generator expression □□ □□□□□□□□□?

✅Answer: [x for x in range(10)] — list comprehension, square brackets, creates full list immediately in memory□ (x for x in range(10)) — generator expression, round brackets, lazy evaluation — values on demand□ Generator memory efficient □□□□□□□ □□□□□□□ iterate □□□ □□□□□□ list comprehension □□□□□□□ □□□□□□□□□□ iterate □□□ □□□□□□ Large data □□ □□□□□ generator use □□□□

□ Interview Q: Nested list comprehension □□□□□□ □□□□□□ □□?

✅Answer: [expr for outer in outer_list for inner in inner_list] — outer loop □□□, inner loop □□□□□ Traditional loop □□ □□□ □□□□□: □□□□□□□ for □□□, □□□□□□□ for □□□□□ Matrix flatten: [item for row in matrix for item in row]□ Condition □□□ □□□ □□□□□: [item for row in matrix for item in row if item > 0]□

Functions

□ Interview Q: Python □ function □□ object? □□□□□□□ □□□□□

✅Answer: □□□□□□□ □□□□□□□: def add(a,b): return a+b; f = add; print(f(2,3)) — variable □ assign □□□ □□□□□□ functions = [add, subtract]; functions[0](1,2) — list □ store □□□ □□□□□□ def apply(func, x, y): return func(x, y); apply(add, 2, 3) — argument □□□□□□□ pass □□□ □□□□□□ add.__name__, add.__doc__ — attributes □□□□□

□ Interview Q: Closure □□? □□□□□ □□□□□□□ □□□□□

✅Answer: Closure □□□ inner function □□□□□ outer function □□ variable □□ remember □□□□ □□□□□ outer function □□□ □□□□□ □□□□□□□□ □□□□□□ def make_multiplier(n): def multiply(x): return x * n; return multiply. times3 = make_multiplier(3); print(times3(5)) # 15. □□□□□□ multiply function □□ n variable □□ 'close over' □□□ □□□□□□ Counter, memoization, decorators □□ pattern □□□□□□□□ □□□□□

□ Interview Q: Python □ □□□□□ deep □□□□□□□□ recursion □□□□□□□ □□□□□?

✅Answer: Python □□ default recursion limit □□□ 1000□ □□ □□□□□ □□□□□ RecursionError: maximum recursion depth exceeded□ sys.setrecursionlimit(2000) □□□□□□ □□□□□□□□ □□□□□ □□□□□□□ dangerous□ Python tail recursion optimize □□□ □□ (C, Scheme □□□)□ Deep recursion □□ □□□□□□□□□□ iteration □□ explicit stack □□□□□□□□ □□□□□

Lambda, *args, **kwargs

□ Interview Q: Lambda function □□ limitations □□ □□?

✅Answer: 1. Single expression only — multiple statements □□□□□ 2. No assignments — x = 5 □□□ □□□□□ □□□ 3. No if-elif-else blocks — □□□□□ ternary expression□ 4. No try-except□ 5. No docstring□ 6. No type annotations□ Lambda □□ 'throwaway' function □□□ □□□□ — quick, short, one-off use □□ □□□□□□ Complex logic □□ □□□□□ □□□□□□□ named function □□□□□□□□ □□□□□

□ Interview Q: *args □□□ **kwargs □□ order □□□ □□□□□□□□□□□□□□?

✅Answer: Function signature □ □□□□□ order: def func(positional, *args, keyword_only, **kwargs)□ Rule: positional → *args → keyword-only → **kwargs□ *args □□ □□□ □□ □□□ □□□□ keyword-only (default □□□□□□ call □ □□□ □□□□□□ □□□□□ □□□)□ Calling □: *list □□□, **dict □□□□ func(1, 2, *[3,4], key='val', **{'a':1})□

Quick Reference Cheat Sheet

Syntax / Tool	Purpose / Usage
`dict.get(k,'x')`	KeyError □□□□□ safe access
`d | d2`	Dict merge (Python 3.9+)
`{**d1, **d2}`	Dict merge (Python 3.5+)
`x in my_set`	O(1) membership check
`{x for x in lst}`	Set comprehension
`{k:v for...}`	Dict comprehension
`(x for x in lst)`	Generator expression
`*args`	Variable positional args → tuple
`**kwargs`	Variable keyword args → dict
`lambda x: x*2`	Anonymous function
`map(f, lst)`	Apply f to each element
`filter(f, lst)`	Keep elements where f is True
`sorted(lst,key=f)`	Sort using key function
`functools.reduce`	Fold list to single value
`copy.deepcopy(obj)`	Completely independent copy