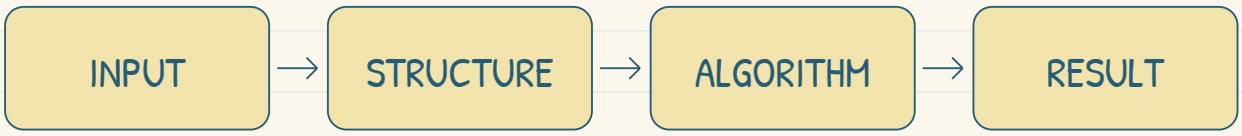
THE PRACTICAL NOTEBOOK

Python with DSA

Think clearly. Choose structures. Solve problems.

64 pages / 20 guided lessons / 40 code files



Understand it. Try it. Explain it.

Learn how to reason about data structures and algorithms using small Python programs. You will trace operations, compare costs, choose suitable structures and build a priority-based task planner. Each lesson connects a technique to a concrete problem instead of asking you to memorise code.

PRIVATUS COSMAS / PRIVER

depriver.tech · First edition / October 2026

Your 20-lesson route

Work in order, or use this clickable map to revisit a topic. Each lesson has a learn page, a practice page and an explained solution.

01	Measure growth	4
02	Lists and copying	7
03	Strings and two pointers	10
04	Hash maps and counting	13
05	Sets and membership	16
06	Stacks and matching	19
07	Queues and scheduling	22
08	Binary search	25
09	Sorting with a purpose	28
10	Recursion and base cases	31
11	Linked structures	34
12	Trees and traversal	37
13	Graphs and breadth-first search	40
14	Depth-first search	43
15	Heaps and priorities	46
16	Greedy interval choices	49
17	Dynamic programming	52
18	Backtracking	55
19	Sliding windows and prefixes	58
20	Project: a task planner	61

Setup and study method · 3 References and next steps · 64

Make this book yours

BEFORE YOU START

You should be able to write a loop, define a function and use a list. If those are new, complete Python: the practical notebook first. No external packages are required for this book.

YOUR WORKSPACE

Create a folder named python-dsa. Use Python 3.11 or newer and a text editor. Each example and solution is independent. Run a file with python lesson_01_example.py. Assertions check expected results and remain silent when they pass. Use a disposable folder for your work.

TERMINAL / FOLLOW THE COMMANDS FOR THIS BOOK

```
python --version
python lesson_01_example.py
python lesson_01_solution.py
```

A SIMPLE STUDY RHYTHM

1. Read: predict what the example will do before running it.
2. Practise: write your answer using the acceptance cases.
3. Compare: read the solution, explain the difference, then extend it.

The companion ZIP contains lesson_01_example.py through lesson_20_example.py and matching solution files. Run them individually from the extracted folder. Most successful checks are silent; a traceback means a check failed.

READABLE NOTES, PRECISE CODE

Handwritten headings keep the notebook feel; body text and code use clear fonts. Copy runnable code from the ZIP to preserve indentation. The PDF is searchable and its contents links are clickable.

Measure growth

THE GOAL

Explain how work changes as input grows.

Complexity describes growth, not stopwatch time. A scan that visits n values performs work proportional to n: O(n). Comparing every pair can require O(n squared). Constant factors still matter in real programs, but growth helps predict which design becomes expensive. Count the operation that dominates, and state what n represents.

lesson_01_example.py

```
def scan_count(values):
    count = 0
    for value in values:
        count += 1
    return count

assert scan_count(range(5)) == 5
assert scan_count(range(10)) == 10
print("Doubling input doubles visits")
```

The function counts visits rather than elapsed time. A fast machine does not change the fact that every value is visited. Empty input produces zero visits. Extra storage is constant because only the counter is retained.

WATCH FOR THIS

Do not label every two-loop program quadratic: two separate scans are O(n), while nested full scans are O(n squared).

Your turn: Measure growth

THE CHALLENGE

Count unordered pairs in a list without comparing an item with itself or counting a pair twice. Return the number of comparisons, not the pairs themselves.

ACCEPTANCE CHECKS

[ ] 0 items -> 0 pairs

[ ] 1 item -> 0 pairs

[ ] 4 items -> 6 pairs

[ ] 8 items -> 28 pairs

A NUDGE, IF YOU NEED ONE

For each index i, start the second index at i + 1.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

MEASURE GROWTH

lesson_01_solution.py

```
def pair_count(values):
    count = 0
    for i in range(len(values)):
        for j in range(i + 1, len(values)):
            count += 1
    return count

assert pair_count([]) == 0
assert pair_count([1]) == 0
assert pair_count(range(4)) == 6
assert pair_count(range(8)) == 28
```

WHY IT WORKS

The result is n(n-1)/2. The loops are quadratic even though they skip half the square. Input values do not affect the count; input length does.

TAKE IT FURTHER

Compare the measured counts for 10, 100 and 1000 items with the formula. Explain why storing every pair would also consume quadratic space.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Lists and copying

THE GOAL

Recognise mutation, aliasing and list-operation costs.

Python lists store an ordered sequence of object references. Index access is constant time; searching for a value requires a scan. Appending is amortised constant time, while inserting at the front shifts existing references. A second name assigned to a list points to the same list unless you explicitly copy it.

lesson_02_example.py

```
values = [4, 7, 9]
alias = values
copy = values.copy()
alias.append(12)
assert values == [4, 7, 9, 12]
assert copy == [4, 7, 9]
print(values, copy)
```

The alias shares the original object. The shallow copy has a separate outer list; nested lists would still be shared. Predict mutations before running the code.

WATCH FOR THIS

A slice is not a deep copy of nested objects. Explain whether your function promises to copy containers or the objects inside them.

Your turn: Lists and copying

THE CHALLENGE

Write rotate_left(values, steps) that returns a rotated new list without modifying the input. Allow steps larger than the length and handle an empty list.

ACCEPTANCE CHECKS

[ ] [1,2,3], 1 -> [2,3,1]

[ ] [1,2,3], 4 -> [2,3,1]

[ ] [], 5 -> []

[ ] Original list remains unchanged

A NUDGE, IF YOU NEED ONE

Reduce steps modulo length, then join two slices. Check empty input before using modulo.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

LISTS AND COPYING

lesson_02_solution.py

```
def rotate_left(values, steps):
    if not values:
        return []
    k = steps % len(values)
    return values[k:] + values[:k]

source = [1, 2, 3]
assert rotate_left(source, 4) == [2, 3, 1]
assert rotate_left([], 5) == []
assert source == [1, 2, 3]
```

WHY IT WORKS

The slices and concatenation visit or copy up to n references, so the solution takes O(n) time and O(n) extra space. It prioritises a clear non-mutating contract.

TAKE IT FURTHER

Extend the function to accept negative steps and document their meaning. Then compare it with repeated pop(0) operations.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Strings and two pointers

THE GOAL

Scan from both ends without making extra substrings.

Strings are immutable sequences. Two-pointer techniques track positions rather than repeatedly copying parts of the text. A palindrome reads the same in both directions. First decide whether case, punctuation and spaces count; an algorithm is only correct relative to a defined rule.

lesson_03_example.py

```
def palindrome(text):
    left, right = 0, len(text) - 1
    while left < right:
        if text[left] != text[right]:
            return False
        left += 1
        right -= 1
    return True

assert palindrome("level")
assert not palindrome("python")
assert palindrome("")
```

Each iteration compares the remaining outer characters. If they match, the unchecked region shrinks. The scan uses O(n) time and O(1) extra storage.

WATCH FOR THIS

Do not claim constant extra space when your implementation builds a cleaned string or a reversed copy.

Your turn: Strings and two pointers

THE CHALLENGE

Create a phrase palindrome checker that ignores case and non-alphanumeric characters. State that the cleaned string is the object being tested.

ACCEPTANCE CHECKS

[ ] "Never odd or even" -> True

[ ] "Python notes" -> False

[ ] "!!!" -> True under the empty-string rule

A NUDGE, IF YOU NEED ONE

Build a cleaned lowercase string, then compare it with its reverse.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

STRINGS AND TWO POINTERS

lesson_03_solution.py

```
def phrase_palindrome(text):
    clean = "".join(c.casefold() for c in text
                    if c.isalnum())
    return clean == clean[::-1]

assert phrase_palindrome("Never odd or even")
assert not phrase_palindrome("Python notes")
assert phrase_palindrome("!!!")
```

WHY IT WORKS

This version spends O(n) extra memory for clarity. Unicode case folding can change character counts, so the rule is explicitly based on the normalised string.

TAKE IT FURTHER

Write a two-pointer version for ASCII letters and digits that skips ignored characters without constructing a cleaned copy.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Hash maps and counting

THE GOAL

Replace repeated scans with keyed lookups.

A dictionary maps keys to values. Typical lookup and update are average O(1), making it useful for counts and indexes. Keys must be hashable. Equality and hashing determine whether two keys represent the same entry. A dictionary is not automatically a sorted structure.

lesson_04_example.py

```
def frequencies(words):
    counts = {}
    for word in words:
        counts[word] = counts.get(word, 0) + 1
    return counts

assert frequencies(["api", "web", "api"]) == {
    "api": 2, "web": 1
}
assert frequencies([]) == {}
```

Each word causes one lookup and one assignment. The map stores one entry per distinct word, so space is O(k) for k distinct values.

WATCH FOR THIS

Be explicit about normalisation: A and a remain different in this contract.

Your turn: Hash maps and counting

THE CHALLENGE

Find the first character that appears exactly once in a string. Return None if there is no such character. Preserve original character order.

ACCEPTANCE CHECKS

[ ] "swiss" -> "w"

[ ] "aabb" -> None

[ ] "" -> None

A NUDGE, IF YOU NEED ONE

Count in one pass and inspect the original string in a second pass.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

HASH MAPS AND COUNTING

lesson_04_solution.py

```
def first_unique(text):
    counts = {}
    for char in text:
        counts[char] = counts.get(char, 0) + 1
    for char in text:
        if counts[char] == 1:
            return char
    return None

assert first_unique("swiss") == "w"
assert first_unique("aabb") is None
assert first_unique("") is None
```

WHY IT WORKS

Two linear passes are still O(n). Returning during counting would be wrong because a later character could repeat an earlier candidate.

TAKE IT FURTHER

Use collections.Counter to express the counting step. Compare readability, not just line count.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Sets and membership

THE GOAL

Track what has already been seen.

A set stores distinct hashable values. It helps with membership, duplicates, intersections and differences. Use it when position and multiplicity do not matter. When order matters, combine a set for lookup with a list for ordered output.

lesson_05_example.py

```
def unique_in_order(values):
    seen, result = set(), []
    for value in values:
        if value not in seen:
            seen.add(value)
            result.append(value)
    return result

assert unique_in_order([3, 1, 3, 2]) == [3, 1, 2]
```

The result retains first appearances. Average work is O(n); additional space is O(k). Converting directly to a set would remove duplicates but not preserve this output contract.

WATCH FOR THIS

A sorted two-pointer solution uses different assumptions and may mutate the input if you sort in place.

Your turn: Sets and membership

THE CHALLENGE

Return whether two different positions contain numbers that sum to a target. Do not reuse one position twice.

ACCEPTANCE CHECKS

[ ] [2,7,11], 9 -> True

[ ] [4], 8 -> False

[ ] [4,4], 8 -> True

[ ] [], 0 -> False

A NUDGE, IF YOU NEED ONE

Look for target minus current value before adding the current value to seen.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

SETS AND MEMBERSHIP

lesson_05_solution.py

```
def has_pair(values, target):
    seen = set()
    for value in values:
        if target - value in seen:
            return True
        seen.add(value)
    return False

assert has_pair([2, 7, 11], 9)
assert not has_pair([4], 8)
assert has_pair([4, 4], 8)
assert not has_pair([], 0)
```

WHY IT WORKS

Checking before adding prevents using the same position as both members. Repeated equal values are allowed when they occupy different positions.

TAKE IT FURTHER

Return the two indexes instead of a boolean by storing value-to-index entries in a dictionary.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Stacks and matching

THE GOAL

Use last-in, first-out order to track unfinished work.

A stack removes the most recently added item first. A Python list supports append and pop at the right end. Stacks model undo history, nested expressions and iterative depth-first traversal. Always decide what an empty pop should mean before writing the caller.

lesson_06_example.py

```
history = []
history.append("typed title")
history.append("added paragraph")
assert history.pop() == "added paragraph"
assert history.pop() == "typed title"
assert not history
```

The newest action is undone before the older action. This is a data-ordering rule, not a special Python type.

WATCH FOR THIS

Counting total openings and closings is insufficient: ")(" has equal counts but the wrong order.

Your turn: Stacks and matching

THE CHALLENGE

Check whether a string containing only parentheses has balanced opening and closing symbols. Reject an early closing parenthesis.

ACCEPTANCE CHECKS

[ ] "(())()" -> True

[ ] ")(" -> False

[ ] "(()" -> False

[ ] "" -> True

A NUDGE, IF YOU NEED ONE

Push on an opening symbol; on a closing symbol, require a non-empty stack before popping.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

STACKS AND MATCHING

lesson_06_solution.py

```
def balanced(text):
    stack = []
    for char in text:
        if char == "(":
            stack.append(char)
        elif char == ")":
            if not stack:
                return False
            stack.pop()
        else:
            raise ValueError("Only parentheses allowed")
    return not stack

assert balanced("(())()")
assert not balanced(")(")
assert not balanced("(()")
assert balanced("")
```

WHY IT WORKS

A final empty stack means every opening symbol found a closing symbol. Worst-case space is O(n), for example a string containing only openings.

TAKE IT FURTHER

Extend the checker to parentheses, square brackets and braces using a closing-to-opening lookup.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Queues and scheduling

THE GOAL

Use first-in, first-out order for fair processing.

A queue processes arrivals in order. collections.deque supports efficient appends and pops at both ends. A list with pop(0) must shift remaining elements. Queues appear in breadth-first search, job processing and simulations; production queues also need persistence and failure handling.

lesson_07_example.py

```
from collections import deque

jobs = deque(["scan", "resize", "report"])
processed = []
while jobs:
    processed.append(jobs.popleft())
assert processed == ["scan", "resize", "report"]
```

The oldest pending job runs first. The queue empties after all jobs are removed. This example assumes each job succeeds immediately.

WATCH FOR THIS

This simulation does not implement real process scheduling, concurrency or durability.

Your turn: Queues and scheduling

THE CHALLENGE

Simulate a round-robin scheduler. Each job is a name and a positive number of time units. Process one unit per turn and requeue unfinished jobs. Return completion order.

ACCEPTANCE CHECKS

[ ] [("A",2),("B",1)] -> ["B","A"]

[ ] [] -> []

[ ] A job needing zero units must be rejected

A NUDGE, IF YOU NEED ONE

Pop a job, subtract one, and either append it again or record completion.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

QUEUES AND SCHEDULING

lesson_07_solution.py

```
from collections import deque

def finish_order(jobs):
    if any(units <= 0 for _, units in jobs):
        raise ValueError("Positive work required")
    queue, done = deque(jobs), []
    while queue:
        name, units = queue.popleft()
        if units == 1:
            done.append(name)
        else:
            queue.append((name, units - 1))
    return done

assert finish_order([("A", 2), ("B", 1)]) == ["B", "A"]
assert finish_order([]) == []
```

WHY IT WORKS

Runtime depends on total work units, not only the number of jobs. A long job does not block shorter jobs from taking their next turn.

TAKE IT FURTHER

Add a timeline of which job ran on each turn and calculate each job's waiting time.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Binary search

THE GOAL

Halve a sorted search interval while preserving an invariant.

Binary search requires sorted input. Maintain a half-open interval [low, high): the desired position is never outside it. Each comparison removes about half the candidates. The search is O(log n), but sorting an unsorted collection first has its own cost.

lesson_08_example.py

```
def lower_bound(values, target):
    low, high = 0, len(values)
    while low < high:
        mid = (low + high) // 2
        if values[mid] < target:
            low = mid + 1
        else:
            high = mid
    return low

assert lower_bound([1, 3, 3, 7], 3) == 1
assert lower_bound([], 4) == 0
```

The result is the first position where target could be inserted while preserving order. It may equal len(values), so do not index without checking.

WATCH FOR THIS

Calling bisect on unsorted input can return plausible but incorrect positions without raising an error.

Your turn: Binary search

THE CHALLENGE

Count target occurrences in a sorted list using left and right insertion positions. Avoid scanning through a long run of duplicates.

ACCEPTANCE CHECKS

[ ] [1,3,3,3,7], 3 -> 3

[ ] [1,2], 5 -> 0

[ ] [], 1 -> 0

A NUDGE, IF YOU NEED ONE

Use bisect_left and bisect_right, then subtract the positions.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

BINARY SEARCH

lesson_08_solution.py

```
from bisect import bisect_left, bisect_right

def count_target(values, target):
    return bisect_right(values, target) - bisect_left(
        values, target)

assert count_target([1, 3, 3, 3, 7], 3) == 3
assert count_target([1, 2], 5) == 0
assert count_target([], 1) == 0
```

WHY IT WORKS

The left boundary sits before equal values and the right boundary after them. Two logarithmic searches remain O(log n).

TAKE IT FURTHER

Write upper_bound yourself and trace low, mid and high for a target larger than every value.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Sorting with a purpose

THE GOAL

Choose keys and preserve equal-key order.

Sorting groups comparable values into an order. Python's sorted returns a new list; list.sort changes the existing list. Key functions express the comparison meaning. Python sorting is stable: equal-key items retain their original relative order.

lesson_09_example.py

```
rows = [("Asha", 80), ("Juma", 90), ("Neema", 80)]
ranked = sorted(rows, key=lambda row: -row[1])
assert ranked == [("Juma", 90), ("Asha", 80),
                  ("Neema", 80)]
assert rows[0] == ("Asha", 80)
```

Negating the score orders it descending. The two students with score 80 remain in arrival order. Define tie-breaking intentionally when a different result is required.

WATCH FOR THIS

Using >= instead of > can reorder equal-key records and lose stability in an adapted record sort.

Your turn: Sorting with a purpose

THE CHALLENGE

Implement insertion sort on a copy. Use it to understand shifting and the sorted-prefix invariant, not as a replacement for Python's general-purpose sort.

ACCEPTANCE CHECKS

[ ] [3,1,2] -> [1,2,3]

[ ] [2,2,1] -> [1,2,2]

[ ] [] -> []

[ ] Input stays unchanged

A NUDGE, IF YOU NEED ONE

At index i, save its value and shift larger prefix elements right until the insertion position is free.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

SORTING WITH A PURPOSE

lesson_09_solution.py

```
def insertion_sort(values):
    result = list(values)
    for i in range(1, len(result)):
        item, j = result[i], i - 1
        while j >= 0 and result[j] > item:
            result[j + 1] = result[j]
            j -= 1
        result[j + 1] = item
    return result

source = [3, 1, 2]
assert insertion_sort(source) == [1, 2, 3]
assert source == [3, 1, 2]
assert insertion_sort([]) == []
```

WHY IT WORKS

After each outer iteration the prefix through i is sorted. Worst-case shifting is O(n squared), while already sorted input needs only one comparison per insertion.

TAKE IT FURTHER

Count shifts for sorted, reversed and randomly ordered inputs of the same size.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Recursion and base cases

THE GOAL

Solve a smaller version until a stopping rule applies.

A recursive function calls itself with a smaller problem. Correctness needs a base case and progress toward it. Each active call uses stack space. Python does not eliminate tail calls, so a deeply recursive solution can hit the recursion limit even when the idea is mathematically correct.

lesson_10_example.py

```
def factorial(n):
    if n < 0:
        raise ValueError("n must be non-negative")
    if n == 0:
        return 1
    return n * factorial(n - 1)

assert factorial(0) == 1
assert factorial(5) == 120
```

factorial(3) waits for factorial(2), which waits for factorial(1), then factorial(0). Results return in reverse call order.

WATCH FOR THIS

A base case that is unreachable from the recursive step causes unbounded recursion.

Your turn: Recursion and base cases

THE CHALLENGE

Sum all integers in a nested structure containing only integers and lists. Empty lists contribute zero. Use small nesting depths for this exercise.

ACCEPTANCE CHECKS

[ ] [1,[2,3],[]] -> 6

[ ] [] -> 0

[ ] [[-2],5] -> 3

A NUDGE, IF YOU NEED ONE

Treat an integer as a finished subproblem; otherwise recurse into the list's elements.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

RECURSION AND BASE CASES

lesson_10_solution.py

```
def nested_sum(value):
    if isinstance(value, int):
        return value
    return sum(nested_sum(item) for item in value)

assert nested_sum([1, [2, 3], []]) == 6
assert nested_sum([]) == 0
assert nested_sum([[-2], 5]) == 3
```

WHY IT WORKS

Every integer is visited once. Call-stack usage follows maximum nesting depth. The stated contract excludes strings and self-referential lists.

TAKE IT FURTHER

Rewrite nested_sum with an explicit stack and explain the tradeoff for deeply nested input.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Linked structures

THE GOAL

Follow references when elements are not stored by position.

A singly linked node stores a value and a reference to the next node. Accessing the kth value requires following links; insertion after a known node can be constant time. Python lists are usually more convenient, but linked structures teach reference manipulation and traversal invariants.

lesson_11_example.py

```
class Node:
    def __init__(self, value, next_node=None):
        self.value = value
        self.next = next_node

head = Node(1, Node(2, Node(3)))
values, node = [], head
while node is not None:
    values.append(node.value)
    node = node.next
assert values == [1, 2, 3]
```

head identifies the first node. The loop ends at None, which represents no next node. An accidental cycle prevents this traversal from terminating.

WATCH FOR THIS

If you overwrite current.next before saving it, the remaining suffix can become unreachable.

Your turn: Linked structures

THE CHALLENGE

Reverse an acyclic singly linked list in place. Return the new head. Preserve a reference to the next node before changing the current link.

ACCEPTANCE CHECKS

[ ] 1 -> 2 -> 3 becomes 3 -> 2 -> 1

[ ] None -> None

[ ] One node remains itself

A NUDGE, IF YOU NEED ONE

Keep previous, current and next-node references. Change one link per iteration.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

LINKED STRUCTURES

lesson_11_solution.py

```
class Node:
    def __init__(self, value, next_node=None):
        self.value, self.next = value, next_node

def reverse(head):
    previous, current = None, head
    while current is not None:
        following = current.next
        current.next = previous
        previous, current = current, following
    return previous

head = reverse(Node(1, Node(2, Node(3))))
assert [head.value, head.next.value,
        head.next.next.value] == [3, 2, 1]
assert reverse(None) is None
```

WHY IT WORKS

The reversed prefix grows while the unreversed suffix shrinks. Work is O(n) and extra pointer storage is O(1).

TAKE IT FURTHER

Use slow and fast pointers to detect a cycle before reversing an unknown input structure.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Trees and traversal

THE GOAL

Represent a hierarchy and visit every node deliberately.

A tree connects a root to child subtrees without cycles. A binary tree has at most two children per node. Inorder visits left subtree, node, then right subtree. It produces sorted values only when the tree also satisfies a binary-search-tree ordering rule.

lesson_12_example.py

```
tree = (4, (2, None, None), (7, None, None))

def inorder(node):
    if node is None:
        return []
    value, left, right = node
    return inorder(left) + [value] + inorder(right)

assert inorder(tree) == [2, 4, 7]
```

Tuples make the example small. This concatenating implementation is clear but can copy many intermediate lists on a skewed tree. Production traversal often appends into one output list or yields values.

WATCH FOR THIS

Height conventions differ across texts. State whether a leaf has height zero or one before comparing results.

Your turn: Trees and traversal

THE CHALLENGE

Count nodes and compute tree height. Define empty height as zero and a leaf's height as one.

ACCEPTANCE CHECKS

[ ] Empty -> count 0, height 0

[ ] A leaf -> count 1, height 1

[ ] Root with two leaves -> count 3, height 2

A NUDGE, IF YOU NEED ONE

Combine left and right results: counts add, heights use the maximum.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

TREES AND TRAVERSAL

lesson_12_solution.py

```
def stats(node):
    if node is None:
        return 0, 0
    _, left, right = node
    lc, lh = stats(left)
    rc, rh = stats(right)
    return 1 + lc + rc, 1 + max(lh, rh)

tree = (4, (2, None, None), (7, None, None))
assert stats(None) == (0, 0)
assert stats((1, None, None)) == (1, 1)
assert stats(tree) == (3, 2)
```

WHY IT WORKS

Each node is processed once: O(n) time. The call stack grows with height h, giving O(h) extra space.

TAKE IT FURTHER

Write an iterative traversal that gathers nodes level by level and compare its maximum queue size.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Graphs and breadth-first search

THE GOAL

Explore neighbours in layers to find unweighted distances.

A graph consists of vertices and edges. An adjacency list maps each vertex to its neighbours. Breadth-first search uses a queue to visit all distance-one neighbours before distance-two neighbours. Mark a vertex as seen when enqueueing it so cycles do not repeatedly add the same vertex.

lesson_13_example.py

```
from collections import deque

def distances(graph, start):
    result, queue = {start: 0}, deque([start])
    while queue:
        node = queue.popleft()
        for neighbour in graph.get(node, []):
            if neighbour not in result:
                result[neighbour] = result[node] + 1
                queue.append(neighbour)
    return result

g = {"A": ["B"], "B": ["A", "C"], "C": []}
assert distances(g, "A") == {"A": 0, "B": 1, "C": 2}
```

The first discovery of a vertex gives its shortest edge count from start in an unweighted graph. Unreachable vertices do not appear in the result.

WATCH FOR THIS

BFS does not solve general weighted shortest paths merely by placing weights beside neighbours.

Your turn: Graphs and breadth-first search

THE CHALLENGE

Return the set of vertices reachable from a start vertex. Include the start even if its adjacency list is absent.

ACCEPTANCE CHECKS

[ ] A-B-C chain from A -> {A,B,C}

[ ] Isolated Z -> {Z}

[ ] A cycle must terminate

A NUDGE, IF YOU NEED ONE

The keys of a BFS distance dictionary already describe reachability.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

GRAPHS AND BREADTH-FIRST SEARCH

lesson_13_solution.py

```
from collections import deque

def reachable(graph, start):
    seen, queue = {start}, deque([start])
    while queue:
        for nxt in graph.get(queue.popleft(), []):
            if nxt not in seen:
                seen.add(nxt)
                queue.append(nxt)
    return seen

assert reachable({"A": ["B"], "B": ["A"]}, "A") == {
    "A", "B"}
assert reachable({}, "Z") == {"Z"}
```

WHY IT WORKS

With adjacency lists, BFS runs in O(V+E) over the reachable portion. Storing every possible pair in a matrix changes the traversal cost.

TAKE IT FURTHER

Store each vertex's parent and reconstruct one shortest path from start to a chosen destination.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Depth-first search

THE GOAL

Explore one branch before returning to alternatives.

Depth-first search uses recursion or an explicit stack. It supports reachability, component exploration and many dependency algorithms. Visit order depends on neighbour order; the visited set is usually the reliable result rather than a single mandatory traversal sequence.

lesson_14_example.py

```
def dfs(graph, start):
    seen, stack = set(), [start]
    while stack:
        node = stack.pop()
        if node in seen:
            continue
        seen.add(node)
        stack.extend(graph.get(node, []))
    return seen

assert dfs({1: [2], 2: [1, 3]}, 1) == {1, 2, 3}
```

The visited check prevents cycles from causing endless processing. Stack entries may contain duplicates before they are popped; only unseen vertices are expanded.

WATCH FOR THIS

Connected components in directed graphs have different definitions; this algorithm assumes an undirected graph.

Your turn: Depth-first search

THE CHALLENGE

Count connected components in an undirected graph. Assume every vertex is a dictionary key and every undirected edge is listed in both directions.

ACCEPTANCE CHECKS

[ ] {1:[2],2:[1],3:[]} -> 2

[ ] {} -> 0

[ ] Three isolated vertices -> 3

A NUDGE, IF YOU NEED ONE

Start a new DFS only when a vertex has not already been reached by an earlier search.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

DEPTH-FIRST SEARCH

lesson_14_solution.py

```
def components(graph):
    seen, count = set(), 0
    for start in graph:
        if start in seen:
            continue
        count += 1
        stack = [start]
        while stack:
            node = stack.pop()
            if node not in seen:
                seen.add(node)
                stack.extend(graph[node])
    return count

assert components({1: [2], 2: [1], 3: []}) == 2
assert components({}) == 0
```

WHY IT WORKS

Although DFS is nested inside a loop, each vertex is expanded once. Under the graph contract, total work remains O(V+E).

TAKE IT FURTHER

Return the members of each component, then test that every input vertex appears exactly once.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Heaps and priorities

THE GOAL

Retrieve the smallest pending item efficiently.

A min-heap guarantees that its smallest element is at index zero, without fully sorting the list. heapq pushes and pops in O(log n). Build a heap with heapify in O(n). Use a sequence number to break equal-priority ties without comparing arbitrary task objects.

lesson_15_example.py

```
import heapq

jobs = []
heapq.heappush(jobs, (2, 0, "report"))
heapq.heappush(jobs, (1, 1, "urgent"))
heapq.heappush(jobs, (2, 2, "backup"))
assert heapq.heappop(jobs)[2] == "urgent"
assert heapq.heappop(jobs)[2] == "report"
```

Tuples compare left to right: priority first, then arrival order. Lower numbers represent higher priority in this example.

WATCH FOR THIS

A heap list is not a sorted list. Only the root is guaranteed to be globally smallest in a min-heap.

Your turn: Heaps and priorities

THE CHALLENGE

Return the k largest numbers from an iterable in descending order. Handle k = 0 and k larger than the collection. Reject negative k.

ACCEPTANCE CHECKS

[ ] [4,1,7,2], 2 -> [7,4]

[ ] [3], 5 -> [3]

[ ] [3], 0 -> []

A NUDGE, IF YOU NEED ONE

Use heapq.nlargest for a readable standard-library solution.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

HEAPS AND PRIORITIES

lesson_15_solution.py

```
import heapq

def top_k(values, k):
    if k < 0:
        raise ValueError("k must not be negative")
    return heapq.nlargest(k, values)

assert top_k([4, 1, 7, 2], 2) == [7, 4]
assert top_k([3], 5) == [3]
assert top_k([3], 0) == []
```

WHY IT WORKS

For a small k, maintaining a limited heap avoids sorting all values. Library implementations may choose different fast paths for special sizes.

TAKE IT FURTHER

Implement a fixed-size min-heap of k elements and compare its result with sorted(values, reverse=True)[:k].

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Greedy interval choices

THE GOAL

Make a local choice only when its correctness can be justified.

A greedy algorithm commits to a local choice and does not revisit it. For selecting the maximum number of non-overlapping intervals, choosing the next interval with the earliest finishing time leaves as much room as possible for later intervals. This reasoning does not apply to every scheduling objective.

lesson_16_example.py

```
def select_intervals(intervals):
    selected, end = [], float("-inf")
    for start, finish in sorted(intervals,
                                key=lambda pair: pair[1]):
        if start >= end:
            selected.append((start, finish))
            end = finish
    return selected

assert select_intervals([(0, 3), (1, 2), (2, 4)]) == [
    (1, 2), (2, 4)]
```

Intervals are treated as half-open, so finishing at time 2 allows another interval to start at 2. Sorting dominates the O(n log n) cost.

WATCH FOR THIS

Passing several examples does not prove a greedy strategy is always optimal.

Your turn: Greedy interval choices

THE CHALLENGE

Demonstrate why always taking the largest coin can fail. For coins 1, 3 and 4, compare greedy change for amount 6 with a better combination.

ACCEPTANCE CHECKS

[ ] Greedy for 6 -> [4,1,1]

[ ] Better combination -> [3,3]

[ ] Both combinations sum to 6

A NUDGE, IF YOU NEED ONE

Implement the greedy routine, then validate the two candidate lists separately.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

GREEDY INTERVAL CHOICES

lesson_16_solution.py

```
def greedy_change(amount, coins):
    chosen = []
    for coin in sorted(coins, reverse=True):
        while amount >= coin:
            chosen.append(coin)
            amount -= coin
    return chosen, amount

chosen, rest = greedy_change(6, [1, 3, 4])
assert chosen == [4, 1, 1] and rest == 0
better = [3, 3]
assert sum(better) == 6
assert len(better) < len(chosen)
```

WHY IT WORKS

A counterexample disproves a universal claim. The routine is not a minimum-coin solver for arbitrary denominations; dynamic programming addresses that broader problem.

TAKE IT FURTHER

Write the exchange argument for interval scheduling, then identify why it does not justify the coin strategy.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Dynamic programming

THE GOAL

Reuse overlapping subproblem results.

Dynamic programming stores answers to smaller states and combines them. Define the state, recurrence, base cases and evaluation order before coding. A table is useful only when its entries have a precise meaning. For climbing stairs with steps of one or two, ways(n) combines ways(n-1) and ways(n-2).

lesson_17_example.py

```
def stair_ways(n):
    if n < 0:
        raise ValueError("n must be non-negative")
    previous, current = 1, 1
    for _ in range(n):
        previous, current = current, previous + current
    return previous

assert stair_ways(0) == 1
assert stair_ways(4) == 5
```

One way to climb zero stairs means choosing no steps. Only the last two states are needed, so extra storage is constant.

WATCH FOR THIS

Using an uncomputed state or the wrong base value can make every later entry wrong.

Your turn: Dynamic programming

THE CHALLENGE

Find the minimum number of coins needed for a non-negative amount. Coins are positive integers and may be reused. Return None if the amount is unreachable.

ACCEPTANCE CHECKS

[ ] Coins [1,3,4], amount 6 -> 2

[ ] Coins [2], amount 3 -> None

[ ] Amount 0 -> 0

A NUDGE, IF YOU NEED ONE

Let dp[x] mean the minimum number of coins for total x. Try every coin that fits.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

DYNAMIC PROGRAMMING

lesson_17_solution.py

```
def min_coins(coins, amount):
    if amount < 0 or any(c <= 0 for c in coins):
        raise ValueError("Invalid amount or coin")
    dp = [0] + [float("inf")] * amount
    for total in range(1, amount + 1):
        for coin in coins:
            if coin <= total:
                dp[total] = min(dp[total], dp[total-coin] + 1)
    return None if dp[amount] == float("inf") else dp[amount]

assert min_coins([1, 3, 4], 6) == 2
assert min_coins([2], 3) is None
assert min_coins([2], 0) == 0
```

WHY IT WORKS

Time is O(amount times number of denominations); space is O(amount). This is pseudo-polynomial in the numeric amount, not polynomial in its digit count.

TAKE IT FURTHER

Store the last chosen coin for each reachable total and reconstruct an optimal combination.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Backtracking

THE GOAL

Explore choices, then undo them cleanly.

Backtracking builds a partial candidate, explores a choice and restores state before trying another choice. It is useful when a solution space is naturally a tree. Pruning eliminates branches that cannot satisfy the rules, but worst-case work can still be exponential.

lesson_18_example.py

```
def subsets(values):
    result = [[]]
    for value in values:
        result += [part + [value] for part in result]
    return result

assert subsets([1, 2]) == [[], [1], [2], [1, 2]]
assert subsets([]) == [[]]
```

Each value doubles the number of subsets. Output size itself is exponential, so no implementation can list every subset in linear time.

WATCH FOR THIS

If you append the same mutable working list to every result, later edits can change previously saved answers. Copy it when saving.

Your turn: Backtracking

THE CHALLENGE

Generate all length-n binary strings with no adjacent 1s. Treat n = 0 as one empty string and reject negative n.

ACCEPTANCE CHECKS

[ ] n=2 -> 00, 01, 10

[ ] n=0 -> one empty string

[ ] Every result has length n

A NUDGE, IF YOU NEED ONE

Always try 0. Try 1 only if the prefix is empty or ends in 0.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

BACKTRACKING

lesson_18_solution.py

```
def binary_strings(n):
    if n < 0:
        raise ValueError("Negative length")
    result = []
    def visit(prefix):
        if len(prefix) == n:
            result.append(prefix)
            return
        visit(prefix + "0")
        if not prefix or prefix[-1] == "0":
            visit(prefix + "1")
    visit("")
    return result

assert binary_strings(2) == ["00", "01", "10"]
assert binary_strings(0) == [""]
```

WHY IT WORKS

The invalid 11 branch is never explored. Immutable prefixes avoid accidental sharing, but each concatenation allocates another string.

TAKE IT FURTHER

Add a rule requiring exactly k ones and prune when too few positions remain to reach k.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Sliding windows and prefixes

THE GOAL

Reuse work between neighbouring ranges.

A fixed-width sliding window updates a range total by removing the outgoing value and adding the incoming value. A prefix-sum array stores cumulative totals so later range queries become subtraction. Choose the method according to whether ranges are processed sequentially or queried arbitrarily.

lesson_19_example.py

```
def prefix_sums(values):
    prefix = [0]
    for value in values:
        prefix.append(prefix[-1] + value)
    return prefix

p = prefix_sums([2, 5, 1, 4])
assert p[3] - p[1] == 6
assert p[4] - p[0] == 12
```

The sum of the half-open range [left, right) is prefix[right] minus prefix[left]. Building takes O(n); each valid range query then takes O(1).

WATCH FOR THIS

Setting best to zero gives an incorrect result when every valid window sum is negative.

Your turn: Sliding windows and prefixes

THE CHALLENGE

Find the maximum sum of exactly k consecutive values. Require 1 <= k <= len(values). Support negative numbers.

ACCEPTANCE CHECKS

[ ] [2,5,1,4], k=2 -> 7

[ ] [-5,-2,-7], k=2 -> -7

[ ] Invalid k -> ValueError

A NUDGE, IF YOU NEED ONE

Initialise both the current sum and the best sum from the first full window.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

SLIDING WINDOWS AND PREFIXES

lesson_19_solution.py

```
def max_window(values, k):
    if not 1 <= k <= len(values):
        raise ValueError("Invalid window size")
    current = best = sum(values[:k])
    for i in range(k, len(values)):
        current += values[i] - values[i-k]
        best = max(best, current)
    return best

assert max_window([2, 5, 1, 4], 2) == 7
assert max_window([-5, -2, -7], 2) == -7
```

WHY IT WORKS

Each subsequent range reuses the preceding total. The scan is O(n), and the working totals use constant extra storage after the initial slice.

TAKE IT FURTHER

Return both the maximum sum and its start index; define how ties are handled.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Project: a task planner

THE GOAL

Combine a dictionary with a priority queue and explain the tradeoffs.

A small task planner needs two views of the same work: lookup by identifier and retrieval by priority. Store current records in a dictionary and pending priorities in a heap. For this first version, tasks are added once and consumed once. Reprioritisation needs stale-entry handling, which is a separate design decision.

lesson_20_example.py

```
import heapq

class Planner:
    def __init__(self):
        self.tasks, self.queue = {}, []
    def add(self, task_id, title, priority):
        if task_id in self.tasks:
            raise ValueError("Duplicate task id")
        self.tasks[task_id] = title
        heapq.heappush(self.queue, (priority, task_id))
    def take(self):
        if not self.queue:
            return None
        _, task_id = heapq.heappop(self.queue)
        return task_id, self.tasks.pop(task_id)

p = Planner()
p.add(1, "Write tests", 2)
p.add(2, "Fix error", 1)
assert p.take() == (2, "Fix error")
```

Identifiers are integers, so equal-priority heap entries can compare them. This contract orders ties by ID, not arrival time. The two structures must remain consistent.

WATCH FOR THIS

Lazy deletion introduces stale entries; forgetting that fact creates subtle bugs when IDs are reused.

Your turn: Project: a task planner

THE CHALLENGE

Extend the planner with cancellation. Avoid searching through the heap to remove an entry; discard cancelled entries when they reach the front.

ACCEPTANCE CHECKS

[ ] Cancel pending task -> it is never returned

[ ] Cancel missing ID -> False

[ ] All cancelled -> take returns None

[ ] Tie priorities -> smaller ID first

A NUDGE, IF YOU NEED ONE

Remove the current record from the dictionary. Pop until a heap ID still exists in the dictionary.

TRACE BEFORE YOU CODE

INPUT / STATE	PREDICTED RESULT	ACTUAL / WHY

EXPLAIN YOUR APPROACH

Write the steps in plain language. Include one edge case and why your approach handles it.

Try the task before turning the page. Your first attempt is part of the lesson.

One clear solution

PROJECT: A TASK PLANNER

lesson_20_solution.py

```
import heapq

class Planner:
    def __init__(self):
        self.tasks, self.queue = {}, []
    def add(self, ident, title, priority):
        if ident in self.tasks:
            raise ValueError("Duplicate id")
        self.tasks[ident] = title
        heapq.heappush(self.queue, (priority, ident))
    def cancel(self, ident):
        return self.tasks.pop(ident, None) is not None
    def take(self):
        while self.queue:
            _, ident = heapq.heappop(self.queue)
            if ident in self.tasks:
                return ident, self.tasks.pop(ident)
        return None

p = Planner()
p.add(1, "A", 1)
p.add(2, "B", 2)
assert p.cancel(1)
assert p.take() == (2, "B")
assert p.take() is None
```

WHY IT WORKS

For this version, never reuse a cancelled ID before its stale heap entry is consumed. A robust version stores a unique generation number in both structures. Cancellation is average O(1), but stale entries still occupy memory until popped.

TAKE IT FURTHER

Add generation counters, a rebuild threshold for stale entries, and a test that cancels then re-adds the same ID at another priority.

CHECK YOUR UNDERSTANDING

Can you explain the important decision in this solution? Change one input and predict the result. Add an assertion for that case, then run the file again.

Sources & next steps

Use these official references when you need the full API contract or a deeper explanation. Documentation can evolve; check the version used by your own environment.

Python data structures

https://docs.python.org/3/tutorial/datastructures.html

Collections

https://docs.python.org/3/library/collections.html

Binary search helpers

https://docs.python.org/3/library/bisect.html

Heap queues

https://docs.python.org/3/library/heapq.html

Sorting guide

https://docs.python.org/3/howto/sorting.html

WHAT THE CHECKS ESTABLISH

All 40 companion scripts were executed independently during preparation. Their assertions check the small fixtures shown here. They do not establish production readiness, security under attack, browser accessibility or performance at scale.

YOUR NEXT PROJECT

Choose a small problem you can explain. Write its inputs, outputs and success checks. Build a minimal version, test one failure path, and ask another learner to follow your README. Record what you would improve next.

Keep your notes. Keep experimenting.