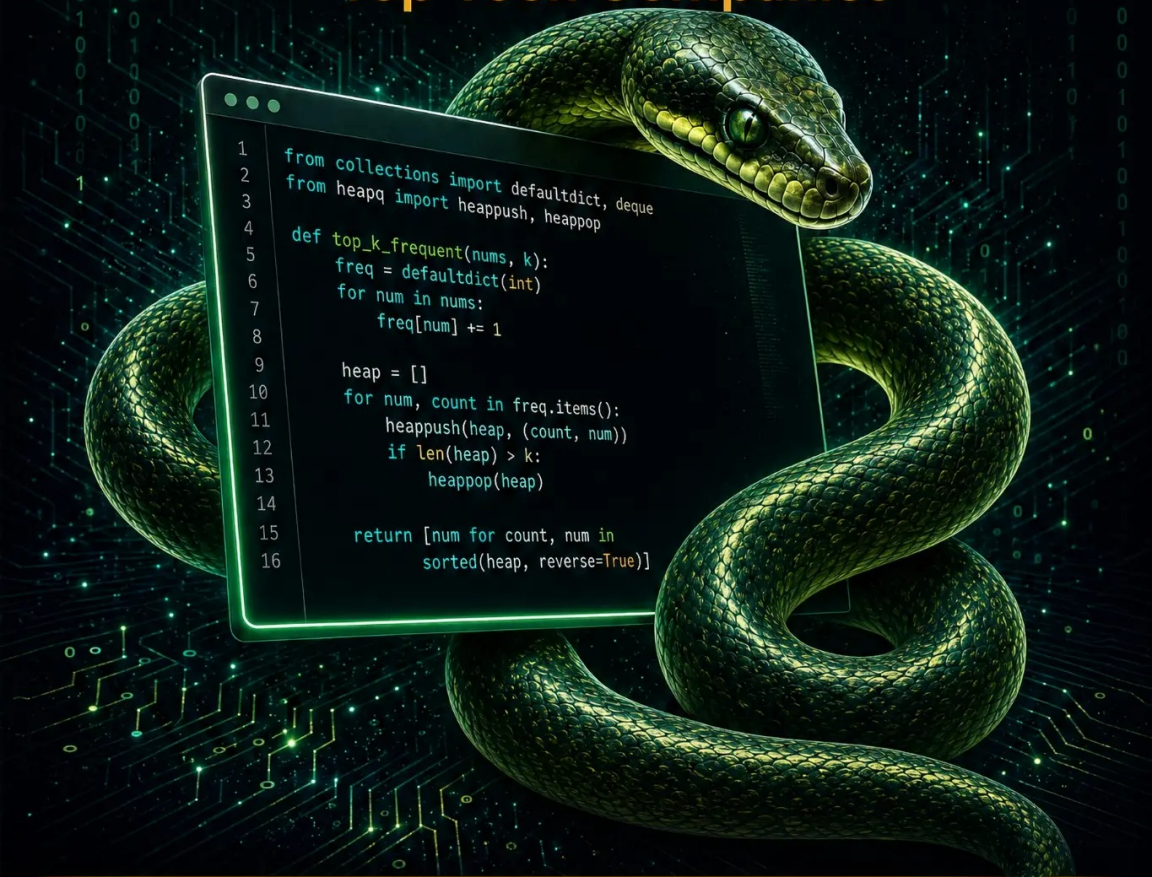
VLAD AZARKHIN

Python for Coding Interviews

Ace Coding Rounds at Top Tech Companies



First Edition 2026

Python for Coding Interviews

Master Python's Secret Weapons to Ace Coding Rounds at Top Tech Companies

Vlad Azarkhin

First Edition · v1.1 · 2026

Python for Coding Interviews

First published 2026.

ISBN: 979-8-180462-87-9 (Paperback)

ISBN: 979-8-180467-46-1 (Hardcover)

About Me, This Project & Disclaimers

This is a project, not a book. I want to be upfront about that. I am a software engineer — not a technical author — and I approached this the way I approach any engineering problem: identify the need, explore the landscape, build an execution plan, execute, and iterate until it is good enough to ship.

The trigger was May 20, 2026 — a round of layoffs at Meta. Watching engineers I respect suddenly find themselves back on the job market made me want to do something useful. I had spent years on both sides of the interview table, and one thing had always been clear to me: Python is by far the most convenient language for coding interviews. The syntax is concise, the standard library is rich, and the signal-to-noise ratio in your solution is high. But not every engineer uses Python day-to-day. I had been looking for a resource that could bring a strong engineer up to speed on Python specifically for interviews — not a general Python tutorial, not a LeetCode grind guide, but something targeted and fast. I could not find it. So I decided to build it.

On AI collaboration. I collaborated heavily with AI throughout this project — for drafting, structuring, code generation, and review. I want to be transparent about that. What I also want to be clear about: I reviewed every single piece of AI output. Every code example was

read, tested mentally, and cross-checked. Every explanation was evaluated for accuracy and clarity. The AI was my most productive collaborator; the editorial judgment was mine.

The review team. Over several weeks, four AI reviewer personas became my closest collaborators on this project. *Margaret Holloway*, a senior technical editor with 15 years in programming book publishing, kept the prose honest and the structure sound. *Daniel Park*, a junior developer coming from Java with no Python background, was the primary signal for whether the Java-to-Python translations actually landed. *Priya Nair*, a mid-level developer with some Python experience, caught the gaps between "I know the syntax" and "I understand why." And *Marcus Chen*, a senior Python engineer who interviews candidates regularly, was the technical authority — he found the edge cases, questioned the conventions, and confirmed that the code was production-quality. Getting all four of them to say "I would buy this" was the bar I set for myself.

I genuinely enjoyed making this. That is not something I say lightly. The combination of a real problem, a tight scope, and a feedback loop that actually worked made this one of the more satisfying projects I have built. I hope it saves you some time.

A note on quality. I spent weeks reviewing and polishing this book chapter by chapter, paragraph by paragraph — reading every sentence, testing every code example, and refining every explanation until it met the bar I set for myself. That said, no book is perfect, and errors may still exist. If you find one, I genuinely want to know. Please submit errata or reach out directly at https://python-book.dotvlad.com — there are dedicated Submit Errata and Contact pages on the site.

About the Author

Vlad Azarkhin is a Principal/Staff-level software engineer and infrastructure architect with 25+ years of experience. His work spans distributed systems, infrastructure platforms, performance, and AI-native engineering — turning ambiguous, high-impact problems into architecture that teams can operate and evolve with confidence.

Table of Contents

Preface

Why Python Wins Interviews

You Already Know How to Code

You've shipped production systems. You've reasoned through algorithms. You know what a hash map is, you understand Big-O, and you've debugged things that would make a junior developer cry. The only thing standing between you and acing that technical interview is **not knowing Python's idioms**.

This book isn't about teaching you to program. It's about handing you a set of power tools you didn't know existed — tools that will let you express solutions more concisely, more readably, and with fewer lines of boilerplate. If you're coming from Java, C++, Go, JavaScript, or any other language, you already have the hardest part covered: you know how to think algorithmically. This book handles the rest.

> **What this book is not:** This is not a data structures and algorithms (DSA) textbook. It does not teach you how to design a binary search tree, derive dynamic programming recurrences, or prove algorithm correctness. Those topics are covered in many excellent books and courses. The sole focus of this book is Python — specifically, the idioms, built-ins, and standard library tools that let you *implement* your algorithmic thinking faster, more cleanly, and with fewer mistakes during a live interview.

The Numbers Don't Lie

Python has been the **#1 language used in technical interviews** at top-tier technology companies for years running — a trend consistently reflected in Stack Overflow's annual Developer Survey and interview platform data. When candidates are given a free choice of language, the overwhelming majority pick Python. The reason is simple: fewer lines means fewer bugs, and fewer bugs means more time to think about the actual problem.

Language	Approx. Lines for Binary Search	Built-in Sorted?	Built-in Heap?
Java	~12–15	Yes (verbose)	PriorityQueue
C++	~12–15	std::sort	priority_queue
Python	~8	**sorted()**	**heapq**

The Python Interview Mindset

Here's the shift in thinking you need to make: **stop writing code and start expressing intent**. In Python, the distance between "what you want" and "what you type" is remarkably small.

Want the top 3 most frequent words? In Java, that's 15 lines. In Python:

```python
from collections import Counter
words = ["apple", "banana", "apple",
         "cherry", "banana", "apple"]
top3 = Counter(words).most_common(3)
# [('apple', 3), ('banana', 2), ('cherry', 1)]
print(top3)
```

That's not a trick. That's just Python. And the interviewer watching you write that in 4 seconds instead of 4 minutes will notice.

The engineers who struggle with Python in interviews aren't bad programmers — they're *experienced* programmers who keep reaching for patterns from their primary language. The biggest unlock is **trusting Python's built-ins**. When you find yourself about to write a for-loop to count elements, stop. There's a better way. This book will wire those instincts in.

How This Book Is Structured

The book opens with **Chapter 0: Python Crash Course for Programmers** — a compact recap of Python syntax for engineers who already know how to code. It is optional, but recommended as a quick orientation to Python's conventions before diving into the interview-specific material.

Each subsequent chapter corresponds to a core Python capability. Every chapter opens with the *why* — why this feature matters in interviews — before diving into the *how*. Code samples throughout the book are production-quality and interview-ready. The conclusion includes a structured practice

plan with 30 specific LeetCode problems mapped to each chapter's techniques — roughly one problem per day over four to five weeks.

The final chapters cover the algorithmic patterns that appear most frequently in technical interviews — two pointers, BFS/DFS, dynamic programming, and backtracking — all implemented with idiomatic Python so you can see how the language features from earlier chapters come together.

> 💡 **A Note on Complexity**
> Key code samples in this book include their time and space complexity. This isn't academic — interviewers will ask. Knowing the Big-O of every Python built-in you use is as important as knowing the algorithm itself.

CH 00 - Python Crash Course for Programmers

Optional: Everything You Need to Read and Write Python from Day One

No Boilerplate. No Ceremony.

The first thing that strikes engineers coming from Java or C++ is how little Python requires to get started. There is no `public static void main`, no type declarations, no semicolons, and no curly braces. Indentation defines code blocks.

In Java, printing *Hello, World!* requires a class declaration, a `main` method signature, and a `System.out.println` call — roughly 5 lines of ceremony. In Python:

```
print("Hello, World!")
```

That is the entire program. No class. No `main`. No import. This is what "no boilerplate" means in practice.

The snippets below are not a complete program — they are standalone examples illustrating individual Python features. In Python, any of these lines can run at the top level of a file with no surrounding ceremony:

```
# Variables — no type declarations, no semicolons
x = 10
name = "Alice"
result = x * 2     # no int, no String, no ;
# Multiple assignment in one line
a, b, c = 1, 2, 3
```

Coming from Java/C++: Python variables are dynamically typed — the type is attached to the value, not the variable. You never declare `int x`; you just write `x = 5`. The interpreter infers the type at runtime.

Control Flow

Conditionals and loops look familiar. Two Python-specific rules apply everywhere: every block header ends with a colon, and indentation (4 spaces by convention) defines the block — there are no curly braces. A wrong indent level is a syntax error, not a style issue.

```
# Indentation IS the syntax — these two are different programs:
if True:
    print("inside the if")   # indented = part of the block
print("outside the if")      # not indented = runs always
```

Conditionals use `if`, `elif` (not `else if`), and `else`. There is no `switch` statement — `elif` chains handle all multi-branch logic:

```
# if / elif / else
x = 42
if x > 100:
    print("large")
elif x > 10:
    print("medium")   # this branch executes
else:
    print("small")
```

Loops in Python iterate directly over sequences — no index bookkeeping required. `range(n)` generates integers 0 to n-1 and is the idiomatic way to loop a fixed number of times.

```
# for loop over a range
for i in range(5):
    print(i)           # 0, 1, 2, 3, 4

# for loop directly over a list — no index needed
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print(fruit)
```

The `while` loop works as expected:

```
# Standard while — loop while condition is true
n = 10
while n > 0:
    n -= 2
print(n)               # 0
```

Python has no `do-while` — the idiomatic replacement is `while True:` with an explicit `break`:

```
# while True with break — Python's do-while equivalent
result = 0
while True:
    result += 1
    if result >= 5:
        break
print(result)          # 5
```

Python also has a **ternary expression** — a one-line conditional that returns one of two values. The syntax is intentionally readable, but the order is the opposite of C, Java, and JavaScript:

```
# Python ternary: value_if_true IF condition ELSE fallback
x = 10
result = "positive" if x > 0 else "non-positive"

# Compare with C / Java / JavaScript:
# result = x > 0 ? "positive" : "non-positive"

# Read it as: "give me X, if condition, otherwise Y"
abs_val = x if x >= 0 else -x
```

The condition sits in the *middle*, not at the start. This is the most common source of confusion for programmers coming from other languages. You will see this pattern constantly in comprehensions — for example, `["even" if n % 2 == 0 else "odd" for n in range(5)]` — so it is worth internalising now.

> 💡 **No i++ in Python**
> Python has no increment operator. Use `i += 1` instead. This trips up nearly every C++ and Java developer on their first Python session.

Exception Handling

Python uses `try/except` for exception handling. The syntax is cleaner than Java's `try/catch`, and the most common interview use case is safely converting strings to numbers or handling missing keys:

```
# Catch a single exception type
try:
    n = int(user_input)   # ValueError if not a number
except ValueError:
    n = 0                 # fallback
```

```
# Catch multiple exception types in one clause
try:
    val = d[key]          # KeyError if key missing
    result = 10 / val     # ZeroDivisionError if val is 0
except (KeyError, ZeroDivisionError):
    result = -1
```

```
# else runs only if no exception was raised
# finally always runs, even if an exception occurred
try:
    x = int(s)
except ValueError:
    x = 0
else:
    print(f"Parsed: {x}")
finally:
    print("done")
```

The most common exceptions you will encounter in interview code are `ValueError` (bad type conversion), `KeyError` (missing dict key), and `IndexError` (out-of-bounds list access). In practice, most interview solutions avoid exceptions entirely

by checking preconditions — exceptions are rarely the focus of an interview problem, but knowing the syntax is expected.

> 💡 **Prefer .get() over try/except for dicts**
> Instead of `try: v = d[k] except KeyError: v = 0`, write `v = d.get(k, 0)`. It is one line, clearly expresses intent, and avoids exception overhead.

Functions

Python functions are defined with `def`. There are no access modifiers, and type annotations are entirely optional. Python supports default arguments and multiple return values natively — multiple return values are actually a tuple under the hood, unpacked automatically by the caller.

```
def greet(name, greeting="Hello"):
    return f"{greeting}, {name}!"
print(greet("Alice"))           # Hello, Alice!
print(greet("Bob", "Hi"))       # Hi, Bob!
```

Python supports keyword arguments — you can pass arguments by name in any order. This is used heavily in standard library calls like `sorted(key=..., reverse=True)`:

```
def describe(name, age, city="Unknown"):
    return f"{name}, {age}, from {city}"
# keyword args can be passed in any order
print(describe(age=30, name="Alice"))   # Alice, 30, from
Unknown
print(describe("Bob", 25, city="NYC"))  # Bob, 25, NYC
```

Python supports optional **return type annotations** using the `->` syntax. These are not enforced at runtime — they are purely informational, serving as documentation for the reader and for static analysis tools. In interviews, you are never required to use them, but they can make your intent clearer:

```
# Without annotation — perfectly valid
def add(a, b):
    return a + b
# With annotation — same function, more explicit
def add(a: int, b: int) -> int:
    return a + b
# -> None for functions that return nothing
def log_message(msg: str) -> None:
    print(f"[LOG] {msg}")
# Multiple return values annotated as a tuple
def min_max(nums: list) -> tuple:
    return min(nums), max(nums)
```

> 💡 **Annotations in interviews**
> You are never required to write return type annotations. Most interviewers will not expect them. However, using `-> int` or `-> None` on a function you define can demonstrate Python fluency and make your code easier to follow during a live session.

Lambdas

A **lambda** is a one-line anonymous function. The syntax is `lambda arguments: expression` — it is exactly equivalent to a named `def` function that returns the expression, just written inline. Lambdas cannot contain statements or multiple lines; they are for simple, throwaway logic only.

```
# These two are identical:
def square(x):
    return x * x

square = lambda x: x * x

# Lambda with two arguments
add = lambda a, b: a + b
print(add(3, 4))   # 7
```

In coding interviews, you will almost never define a lambda and assign it to a variable. Instead, you will pass one directly as the `key=` argument to `sorted()`, `min()`, or `max()`. This is the pattern that appears in nearly every sorting or greedy problem:

```
# Sort a list of intervals by their start value
intervals = [[3, 6], [1, 4], [2, 5]]
# [[1,4], [2,5], [3,6]]
sorted(intervals, key=lambda x: x[0])

# Sort strings by length, then alphabetically
words = ["banana", "fig", "apple", "kiwi"]
sorted(words, key=lambda w: (len(w), w))

# Find the point closest to the origin
points = [(3, 4), (1, 1), (2, 3)]
min(points, key=lambda p: p[0]**2 + p[1]**2)   # (1, 1)
```

> 💡 **Lambda vs def**
> Use a lambda when the logic is a single expression and you are passing it directly as an argument. Use `def` when the logic is more than one line, needs a name for clarity, or will be reused. In interviews, `key=lambda ...` is idiomatic and expected — interviewers recognise it immediately.

Nested Functions

Python allows you to define a function *inside* another function. These **nested functions** (also called inner functions) are the natural complement to lambdas: use a lambda for a single-expression key, and a nested `def` when the logic needs more than one line, a conditional, or access to pre-computed data.

The key feature that makes nested functions powerful is the **closure**: the inner function automatically has read access to all variables in the outer function's scope, without needing to pass them as arguments.

```
def outer(x):
    def inner(y):
        # inner reads x from outer scope (closure)
        return x + y
    return inner

add5 = outer(5)
print(add5(3))   # 8
```

In interviews, you will use this pattern most often when the sort key depends on data computed before the sort — for example, a frequency map. A lambda cannot reference a local variable cleanly in a one-liner, but a nested function can:

```
from collections import Counter

def sort_by_frequency(nums):
    freq = Counter(nums)

    # Nested function has access to freq via closure
    def sort_key(x):
        return (-freq[x], x)   # desc freq, then asc value

    return sorted(nums, key=sort_key)

# [4, 4, 4, 1, 1, 2, 2]
sort_by_frequency([4, 1, 1, 2, 4, 4, 2])
```

Another common use is helper functions for recursion. Defining the recursive helper inside the main function keeps the outer interface clean — the caller only sees one function, not the internal helper:

```
def max_depth(root):
    def dfs(node):
        if not node:
            return 0
        return 1 + max(dfs(node.left), dfs(node.right))
    return dfs(root)
```

Closures give read access to outer variables automatically. But if you need to **write** to an outer variable from inside a nested function, you must declare it with `nonlocal`. Without it, Python treats the assignment as creating a new local variable, causing an `UnboundLocalError`:

```python
def count_nodes(root):
    count = 0
    def dfs(node):
        nonlocal count  # required to modify outer variable
        if not node: return
        count += 1      # without nonlocal: UnboundLocalError!
        dfs(node.left)
        dfs(node.right)
    dfs(root)
    return count
```

⚠️ nonlocal is an interview trap
Forgetting `nonlocal` when a nested function modifies an outer counter is one of the common bugs in Python. The error — `UnboundLocalError: local variable 'count' referenced before assignment` — is confusing if you don't know why it happens. Rule: if a nested function *assigns* to an outer variable (not just reads it), add `nonlocal variable_name` at the top of the inner function.

💡 When to use each
Lambda: single expression, passed directly inline — `key=lambda x: x[0]`.
Nested def: multi-line logic, needs a conditional, or reads from the outer scope (closure). Both are idiomatic Python — choose based on complexity.

Core Data Structures

Python's four built-in data structures — **list**, **tuple**, **dict**, and **set** — cover the vast majority of what you will need in interviews. This section is a quick syntax reference so you can

read code fluently from Chapter 1 onward. Each structure is covered in depth in its own dedicated chapter: lists in Chapter 3, strings in Chapter 4, hash maps and sets in Chapter 6.

A **list** is an ordered, mutable sequence. Indexing is O(1); `append` and `pop()` from the end are O(1) amortised; inserting or removing at an arbitrary position is O(n) because elements must shift.

```
nums = [3, 1, 4, 1, 5, 9]
print(nums[0])          # 3  — first element
print(nums[-1])         # 9  — last element
print(nums[1:4])        # [1, 4, 1]
print(nums[::-1])       # [9, 5, 1, 4, 1, 3]
nums.append(7)          # O(1) amortized
nums.pop()              # O(1) — removes last
```

The slice syntax is `[start:stop:step]`. Omitting `start` and `stop` selects the whole list; a `step` of `-1` walks from the end to the beginning, which is the idiomatic way to reverse any sequence in Python.

A **tuple** is an immutable sequence — once created, its elements cannot change. Tuples are commonly used for coordinates, heap entries (which sort by first element), and anywhere you want to signal that a value should not be modified.

```
point = (3, 7)
x, y = point           # unpacking works just like a list
coords = [(0, 0), (1, 2), (3, 4)]  # list of tuples is common
heap_entry = (priority, value)  # tuples sort by first element
```

The `dict` is Python's hash map — O(1) average for get, set, and delete.

```
freq = {}
freq["apple"] = 3
count = freq.get("cherry", 0)   # 0 — no KeyError
```

A **set** stores unique elements with O(1) average membership testing, add, and discard. Always prefer `x in my_set` over `x in my_list` when you only need to check existence — a list scan is O(n).

```
seen = set()
seen.add(5)
seen.add(5)            # duplicates ignored
if 5 in seen: print("found")  # O(1)
seen.discard(5)        # remove if present, no error if missing
```

> 💡 **Set vs List for Membership Testing**
> When tracking visited nodes in BFS/DFS, always use a `set`, not a `list`. `x in my_list` is O(n); `x in my_set` is O(1). This single change can turn a time-limit-exceeded submission into an accepted one.

Iterables — Everything You Can Loop Over

In C++ and Java, looping typically means working with arrays or explicit iterator objects. Python generalises this into a single concept: the **iterable**. Any object that can be stepped through with a `for` loop is an iterable. This includes lists,

tuples, strings, dicts, sets, ranges, files, and generator expressions — all without any extra ceremony.

```
# All of these work identically in a for loop:
# string, tuple, dict (keys), set, range — all iterable
for ch in "hello":           print(ch)   # h e l l o
for x  in (1, 2, 3):        print(x)    # 1 2 3
for k  in {"a": 1, "b": 2}: print(k)    # a b
for v  in {10, 20, 30}:     print(v)    # unordered
for i  in range(5):          print(i)    # 0 1 2 3 4
```

The key practical consequence is that Python's built-in functions — `sorted()`, `sum()`, `min()`, `max()`, `list()`, `len()` — accept *any* iterable, not just lists. This is why `sorted("hello")` returns `['e', 'h', 'l', 'l', 'o']` and `sum(range(100))` works without building a list first.

```
# sorted() works on any iterable
sorted("hello")          # ['e', 'h', 'l', 'l', 'o']
sorted({3, 1, 2})        # [1, 2, 3]  — set → sorted list
sorted({"b": 2, "a": 1}) # ['a', 'b'] — dict keys, sorted

# sum/min/max on ranges — no list needed
sum(range(101))          # 5050
min(x*x for x in range(10))  # 0  — generator expression
```

Coming from C++/Java: C++ has range-based `for` loops over containers, and Java has the enhanced `for` loop, but neither has the unified iterable protocol Python does. In Python, if something can be looped over, it can be passed to `sorted()`, `sum()`, `list()`, and most other built-ins directly — no conversion needed.

Generator expressions are a special kind of iterable. They look like list comprehensions (covered in the next section) but use `()` instead of `[]`, and they produce values one at a time without building the full sequence in memory. You will see them most often as arguments passed directly to `sum()`, `min()`, or `max()`:

```
# List comprehension — builds full list in memory
squares_list = [x*x for x in range(1000000)]

# Generator expression — computes on demand, no list built
total = sum(x*x for x in range(1000000))

# Common pattern: find min/max with a condition
min_even = min(x for x in nums if x % 2 == 0)
```

For coding interviews, the main takeaway is simple: when you pass a generator expression directly to `sum()`, `min()`, or `max()`, you do not need the extra `[]`. It is both more readable and more memory-efficient.

Classes and Node Definitions

In interviews, you will mostly define simple node classes for linked lists and trees. Python classes are far less ceremonious than Java or C++. The constructor is always named `__init__`, and `self` is explicit in every method signature — Python has no implicit `this`.

```
# Linked list node — memorize this
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

# Binary tree node — memorize this too
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right
```

Building and traversing a linked list is straightforward — just follow the `next` pointers until you hit `None`:

```
head = ListNode(1)
head.next = ListNode(2)
head.next.next = ListNode(3)

curr = head
while curr:
    print(curr.val)     # 1, 2, 3
    curr = curr.next
```

None is Python's equivalent of `null`. It is the default return value of any function that does not explicitly return something. Check for it with `is None`, not `== None`. One critical gotcha: `list.sort()` returns `None` — it sorts in place. Use `sorted(lst)` if you need a new sorted list.

```
# The None gotcha — trips up almost every newcomer
nums = [3, 1, 2]
bad = nums.sort()       # bad is None!
good = sorted(nums)     # good is [1, 2, 3]
```

> 💡 **You Are Ready**
> That covers everything you need to read Python code fluently and write basic solutions. The rest of this book focuses on the Python features that make interview solutions elegant and fast — the idioms, the standard library power tools, and the algorithmic patterns. Let's go.

NOW IT'S YOUR TURN

TRANSLATE · ~2 MIN

Rewrite this C/Java ternary in Python: `x > 0 ? "pos" : "neg"`

PREDICT · ~1 MIN

What error does this raise, and why?

```
def count(root):
    n = 0
    def go(node):
        n += 1
    go(root)
```

APPLY · ~5 MIN

Given a linked-list head built from `ListNode`, return its length by walking `.next` (no recursion).

Stretch: re-derive the `TreeNode` / `ListNode` classes from memory. Solutions in the Appendix.

CH 01 - Sorting

Python's Sort Is a Secret Weapon

Why Sorting Matters More Than You Think

Sorting appears in a significant fraction of medium-difficulty interview problems — not always as the main task, but as a preprocessing step that makes the real solution possible. Knowing how to sort quickly and correctly in Python is one of the highest-leverage skills you can develop.

Python's sort is implemented as **Timsort**, a hybrid merge sort / insertion sort algorithm that runs in O(n log n) worst case and O(n) best case on already-sorted data. It is stable, meaning equal elements maintain their original relative order — a property that matters when you sort by one key and need to preserve a secondary ordering.

Ascending and Descending Sort

Python gives you two ways to sort: `.sort()` modifies the list in place and returns `None`, while `sorted()` leaves the original untouched and returns a new list. The choice between them is deliberate — use `.sort()` when you no longer need the original order, and `sorted()` when you need both versions or when the input is not a list (tuples, strings, generators, sets).

A common mistake from Java or C++ is writing `result = nums.sort()` and then using `result` — it will be `None`. Python's in-place sort intentionally returns nothing to signal that no new object was created:

```
# In-place sort — modifies the original list
nums = [3, 1, 4, 1, 5, 9]
nums.sort()
print(nums)  # [1, 1, 3, 4, 5, 9]
# sorted() — returns a new list, original unchanged
nums = [3, 1, 4, 1, 5, 9]
sorted_nums = sorted(nums)
print(nums)        # [3, 1, 4, 1, 5, 9] — unchanged
print(sorted_nums) # [1, 1, 3, 4, 5, 9]
```

Descending order and sorting strings are equally simple:

```
# Descending
nums.sort(reverse=True)
print(nums)  # [9, 5, 4, 3, 1, 1]

# Sort strings — lexicographic by default
words = ["banana", "apple", "cherry"]
words.sort()
print(words)  # ['apple', 'banana', 'cherry']
```

Complexity: O(n log n) time · O(n) auxiliary space for both `sorted()` and `.sort()` — Timsort allocates a merge buffer up to n/2 elements even for in-place sort. The difference is that `sorted()` also allocates the output list.

> 💡 **Interview Tip**
> Use `.sort()` when you don't need the original list preserved. Use `sorted()` when you need both the original and the sorted version, or when sorting any non-list iterable (tuples, strings, sets, generators — anything you can loop over; see Chapter 0).

Custom Sort with key=

Coming from Java or C++, sorting with a custom comparator involves boilerplate. Python's `key=` eliminates it. Here are the two most common patterns:

Sort by element length

```
// Java
Collections.sort(list, (a, b) -> a.length() - b.length());
// C++
sort(v.begin(), v.end(), [](auto& a, auto& b){ return a.size() <
b.size(); });
# Python
lst.sort(key=len)
```

Sort by a field / second element

```
// Java
list.stream().sorted(Comparator.comparing(x ->
x[1])).collect(...);

// C++
sort(v.begin(), v.end(), [](auto& a, auto& b){ return a[1] <
b[1]; });

# Python
lst.sort(key=lambda x: x[1])
```

The `key=` parameter is where Python's sort becomes genuinely powerful. It accepts any callable that takes one element and returns a comparison value. The function is called *once per element* (not once per comparison), making it efficient.

```
# Sort by string length
words = ["banana", "fig", "apple", "kiwi"]
words.sort(key=len)
print(words)  # ['fig', 'kiwi', 'apple', 'banana']

# Sort by second element (index 1)
pairs = [(1, 3), (2, 1), (4, 2)]
pairs.sort(key=lambda x: x[1])
print(pairs)  # [(2, 1), (4, 2), (1, 3)]
```

Sorting Strings and Custom Objects

String sorting in Python is lexicographic by default, which is usually what you want. But for case-insensitive sorting or anagram grouping, you need the key:

```
# Case-insensitive sort
words = ["Banana", "apple", "Cherry"]
words.sort(key=str.lower)
print(words)  # ['apple', 'Banana', 'Cherry']

# Sort characters of a string
s = "dcba"
sorted_str = "".join(sorted(s))   # "abcd"

# Anagram check using sorted — O(n log n)
def is_anagram(s, t):
    return sorted(s) == sorted(t)
```

Python's sort is **stable** — elements with equal keys preserve their original relative order. This matters when you sort by one criterion and want to maintain a secondary ordering from a previous sort:

```
# Stability: sort words by frequency (most common first)
from collections import Counter
words = ["apple", "banana", "apple",
         "cherry", "banana", "apple"]
freq = Counter(words)
words.sort(key=lambda w: -freq[w])
print(words)
# ['apple'*3, 'banana'*2, 'cherry'*1]
```

Multi-Key Sort and the Negation Trick

The `key=` function can return a **tuple**. Python compares tuples element by element — the first element is the primary sort criterion, the second is the tiebreak, and so on. This gives you a free multi-level sort in a single pass, with no extra code.

```
# Primary: sort by length. Tiebreak: alphabetically
words = ["banana", "fig", "apple", "kiwi"]
words.sort(key=lambda w: (len(w), w))
print(words)  # ['fig', 'kiwi', 'apple', 'banana']

# Primary: sort by first element. Tiebreak: second element
pairs = [(1, 3), (2, 1), (1, 1), (2, 4)]
pairs.sort(key=lambda x: (x[0], x[1]))
print(pairs)  # [(1, 1), (1, 3), (2, 1), (2, 4)]
```

The negation trick solves the mixed-direction problem. Python's sort is always ascending, and `reverse=True` flips *all* criteria at once. When you need one criterion ascending and

another descending, negate the numeric field you want reversed:

Note: this example uses `Counter` *from the* `collections` *module — a dict subclass that counts element frequencies in one line. It is covered in depth in Chapter 6; for now, treat* `Counter(nums)` *as "build a frequency map of nums".*

```
# Sort by frequency DESC, then by value ASC (tiebreak)
from collections import Counter
nums = [4, 1, 1, 2, 4, 4, 2]
# Counter(nums) → {4: 3, 1: 2, 2: 2}  (element → count)
freq = Counter(nums)

# -freq[x] most-negative for highest freq → sorts first
nums.sort(key=lambda x: (-freq[x], x))
print(nums)  # [4, 4, 4, 1, 1, 2, 2]
```

To see why it works: negate a number and its relative order flips. The element with frequency 3 gets key `-3`; the elements with frequency 2 get key `-2`. Since `-3 < -2`, the most frequent element sorts first — exactly what we want. The second tuple element `x` is left positive, so tiebreaks sort ascending as normal.

```
# Another pattern: sort intervals by start ascending,
# then by end DESCENDING (merge intervals variant)
intervals = [[1, 4], [1, 6], [2, 3], [1, 2]]
intervals.sort(key=lambda x: (x[0], -x[1]))
print(intervals)  # [[1, 6], [1, 4], [1, 2], [2, 3]]
```

When key= Isn't Enough: cmp_to_key

The negation trick only works on numbers. When the comparison between two elements depends on *both* values together — not just a property of each one individually — you need a **comparator function**. A comparator takes two elements `a` and `b` and returns a negative number if `a` should come first, a positive number if `b` should come first, and zero if they are equal.

Python's `sorted()` doesn't accept a comparator directly. Wrap it with `functools.cmp_to_key`, which converts the comparator into a `key=` function:

```
import functools

# Classic: arrange numbers to form the largest number
# e.g. [3, 30, 34, 5, 9] → "9534330"
nums = [3, 30, 34, 5, 9]

def compare(a, b):
    # Which concatenation is larger: ab or ba?
    if str(a) + str(b) > str(b) + str(a):
        return -1   # a comes first
    elif str(a) + str(b) < str(b) + str(a):
        return 1    # b comes first
    else:
        return 0

nums.sort(key=functools.cmp_to_key(compare))
print("".join(map(str, nums)))  # "9534330"
```

This problem cannot be solved with a simple `key=` function because the right order of `3` and `30` depends on comparing `"330"` vs `"303"` — you need to look at both elements together.

> 💡 **The (a > b) − (a < b) idiom**
> This one-liner returns −1, 0, or 1 for standard ascending order — the Python equivalent of Java's `a.compareTo(b)`. Use it as the default branch inside a `cmp_to_key` comparator when you only want to override specific cases: `return (a > b) - (a < b)`.

Try this: LeetCode #56 Merge Intervals · #179 Largest Number · #75 Sort Colors · #1636 Sort Array by Increasing Frequency

NOW IT'S YOUR TURN

TRANSLATE · ~2 MIN

One line of Python for: *sort* `words` *by length descending.* (Java: `list.sort((a,b) -> b.length()-a.length())`.)

PREDICT · ~1 MIN

What does `sorted(["10", "9", "2"])` return? Why isn't it `["2","9","10"]`?

APPLY · ~5 MIN

Sort a list of `(name, age)` tuples by **age descending, then name ascending** — single `key=`.

Stretch: LeetCode #179 Largest Number (uses `cmp_to_key`). Solutions in the Appendix.