Python Lists & Strings — Detailed Concepts, Examples & Competitive Coding Techniques

This guide teaches lists and strings thoroughly: how they work, why certain operations behave the way they do, practical patterns used in competitive programming, step-by-step worked examples, and clear explanations aimed at building intuition (not memorization).

1. Lists — Concept & Memory Model

A Python list is an ordered, mutable collection of items. Each list stores references to objects. Lists are dynamic arrays under the hood (CPython implementation): they allocate a contiguous block of memory and expand with amortized cost. Because they store references, a list can hold heterogeneous types.

Key properties: ordered (indexable), mutable (elements can change), dynamic size (append/pop), and allow duplicates.

1.1 Indexing & Slicing (Step-by-step)

Indexing uses integer offsets starting at 0. Negative indices count from the end (-1 is last element). Slicing extracts a sub-list using start:stop:step and creates a new list (a shallow copy of references).

```
# Indexing examples
lst = ['a', 'b', 'c', 'd', 'e']
lst[0]      # 'a'    (first element)
lst[-1]     # 'e'    (last element)
lst[1:4]    # ['b', 'c', 'd']  (stop is exclusive)
lst[:3]     # ['a', 'b', 'c']
lst[::2]    # ['a', 'c', 'e']  (every second element)
# Important: slicing returns a NEW list (shallow copy)
sub = lst[1:4]
sub[0] = 'B'   # changes sub but not lst
```

Slicing complexity: O(k) where k is number of items in the slice because it copies references. Indexing is O(1).

1.2 Common Mutating Operations

append, extend, insert, pop, remove, clear — each has different costs and behavior.

```
# Append vs Extend vs Insert
a = [1, 2, 3]
a.append(4)        # [1,2,3,4]        O(1) amortized
a.extend([5,6])    # [1,2,3,4,5,6]    O(k) where k=len(arg)
a.insert(1, 99)    # [1,99,2,3,4,5,6] O(n) because elements shifted
a.pop()            # removes last element, O(1)
a.pop(0)           # removes first element, O(n) (shifts left)
a.remove(99)       # search then remove - O(n)
```

Tip: use deque from collections for frequent pops/inserts at both ends (O(1)).

1.3 Iteration Patterns & enumerate

Iterate directly to get elements; use enumerate when you need indices. Don't modify list length while iterating unless you know exactly what you're doing.

```
# Iteration patterns
```

```
fruits = ['apple', 'banana', 'cherry']
for item in fruits:
    print(item)
for i, item in enumerate(fruits):
    print(i, item)  # i is index, item is element
# If you must remove while iterating, iterate on a copy:
for item in fruits[:]:
    if item.startswith('b'):
        fruits.remove(item)
```

1.4 List Comprehensions — Readability + Power

Comprehensions build lists concisely. They are readable and fast because they are optimized in C. You can add conditionals and nested loops.

```
# Basic list comprehension
squares = [x*x for x in range(6)]          # [0,1,4,9,16,25]

# With condition
evens = [x for x in range(20) if x % 2 == 0]

# Nested comprehension (flatten)
matrix = [[1,2],[3,4],[5,6]]
flat = [v for row in matrix for v in row]  # [1,2,3,4,5,6]

# Comprehension vs generator expression
gen = (x*x for x in range(1000000))  # lazy, uses less memory
```

When to use comprehensions: for building transformed lists. Use generator expressions when you want lazy evaluation or memory savings.

1.5 Shallow vs Deep Copy (Common trap)

Copying a list copies references (shallow copy). If the list contains mutable objects (like other lists), you'll still share those inner objects unless you deepcopy.

```
import copy
a = [[1,2], [3,4]]
b = a.copy()            # shallow copy
b[0][0] = 99
# a becomes [[99,2],[3,4]] because inner lists shared

c = copy.deepcopy(a)    # deep copy: completely independent
```

1.6 Sorting — stable sort, key, reverse

Python's Timsort is stable and runs in O(n log n) worst-case. Use sort() to sort in-place and sorted() to return a new sorted list. The key parameter lets you provide a function to sort by.

```
data = [('alice', 25), ('bob', 20), ('carol', 25)]
# Sort by age (second element)
sorted_by_age = sorted(data, key=lambda x: x[1])  # [('bob',20),('alice',25),('carol',25)]

# Custom key: primary sort by age, secondary by name
sorted(data, key=lambda x: (x[1], x[0]))
```

Because sort is stable, when sorting by multiple criteria, you can sort by lesser priority first and then by higher priority—Timsort stability preserves previous order.

1.7 Lists in Competitive Coding — Patterns & Recipes

Lists are used to store sequences, intermediate results, and to implement algorithms. Competitive programming patterns include: two pointers (on lists/strings), sliding window (subarray/subsequence), prefix sums, and frequency arrays.

We will show worked examples below.

Worked Example: Rotate a list by k steps (right rotation)

Problem: Given a list and integer k, rotate the list to the right by k steps. Example: [1,2,3,4,5], k=2 -> [4,5,1,2,3].

Method 1 (using extra list): simple and clear. Method 2 (in-place using reversal): O(1) extra space.

```
# Method 1: extra list (clear and correct)
def rotate_extra(nums, k):
    n = len(nums)
    k %= n
    return nums[-k:] + nums[:-k]

# Method 2: in-place using reversal
def rotate_inplace(nums, k):
    n = len(nums)
    k %= n
    nums.reverse()                 # step1
    nums[:k] = reversed(nums[:k])  # step2 reverse first k elements
    nums[k:] = reversed(nums[k:])  # step3 reverse remaining
    return nums

# Example
arr = [1,2,3,4,5]
print(rotate_extra(arr, 2))     # [4,5,1,2,3]
print(rotate_inplace([1,2,3,4,5], 2))  # [4,5,1,2,3]
```

Complexities: rotate_extra uses O(n) time and O(n) extra space. rotate_inplace uses O(n) time and O(1) extra space (ignoring slicing temporarily creating iterators).

Worked Example: Two-sum (return indices of two numbers that add to target)

Pattern: frequency map or dict for O(n) solution. Idea: as you iterate, store seen element's index; for current x check if target-x was seen.

```
def two_sum(nums, target):
    seen = {}  # value -> index
    for i, x in enumerate(nums):
        need = target - x
        if need in seen:
            return [seen[need], i]
        seen[x] = i
    return None

# Example
print(two_sum([2,7,11,15], 9))  # [0,1]
```

Complexity: O(n) time, O(n) space. Important: handle duplicates carefully by storing indices.

2. Strings — Concept & Mutability

A Python string is an immutable sequence of characters (unicode text in Python 3). Because it is immutable, operations that look like mutation create new string objects. This has performance implications for repeated concatenation (use join or StringIO).

2.1 Indexing, Slicing & Common Methods

Indexing & slicing on strings behave like lists but return strings (slices are new strings). Common useful methods: len, lower, upper, strip, split, join, replace, find, index, startswith, endswith.

```
s = "  Hello, World!  "
s.strip()         # "Hello, World!"
s.lower()         # "  hello, world!  "
s.split(",")      # ['  Hello', ' World!  ']
",".join(['a','b','c'])  # 'a,b,c'
s.replace('World', 'Python')  # "  Hello, Python!  "
s.find('o')        # index of first 'o' or -1
```

2.2 Performance Tip: Building Strings

Avoid repeated concatenation in loops (O(n^2)). Prefer join on a list of pieces or use io.StringIO for very large incremental builds.

```
# Bad: repeated concatenation
res = ""
for part in parts:
    res += part   # O(n^2) over many parts

# Good:
res = "".join(parts)   # O(n) total

# Or use StringIO for streaming-like building
from io import StringIO
buf = StringIO()
for p in parts:
    buf.write(p)
res = buf.getvalue()
```

2.3 Two-pointer String Techniques and Palindromes

Two-pointer technique is common for string tasks: set left and right pointers and move inward based on condition. Useful for palindrome checks, reverse-in-place ideas (if mutable), and comparisons.

```
def is_palindrome(s):
    # Consider only alphanumeric and ignore case
    left, right = 0, len(s)-1
    while left < right:
        while left < right and not s[left].isalnum():
            left += 1
        while left < right and not s[right].isalnum():
            right -= 1
        if s[left].lower() != s[right].lower():
            return False
        left += 1
        right -= 1
    return True
```

```
print(is_palindrome("A man, a plan, a canal: Panama"))  # True
```

Time: O(n), Space: O(1). Two-pointer avoids building additional strings.

2.4 Sliding Window — Longest substring without repeating characters

Sliding window maintains a window [i,j) which contains no repeating characters; expand j while valid; when repeat occurs, move i to shrink window. Use a dict mapping char->last index to jump i forward.

```
def longest_unique_substring(s):
    last = {}  # char -> last seen index
    start = 0
    max_len = 0
    for i, ch in enumerate(s):
        if ch in last and last[ch] >= start:
            start = last[ch] + 1
        last[ch] = i
        max_len = max(max_len, i - start + 1)
    return max_len

print(longest_unique_substring("abcabcbb"))  # 3 ('abc')
```

Time: O(n), Space: O(min(n, alphabet_size)).

2.5 Anagram Check & Character Frequency (Hashing)

To check if two strings are anagrams, either sort and compare (O(n log n)) or count frequency with dict/Counter (O(n)).

```
from collections import Counter

def are_anagrams(a, b):
    return Counter(a) == Counter(b)

# Or manual dict count
def are_anagrams_dict(a, b):
    if len(a) != len(b):
        return False
    count = {}
    for ch in a:
        count[ch] = count.get(ch, 0) + 1
    for ch in b:
        if ch not in count:
            return False
        count[ch] -= 1
        if count[ch] < 0:
            return False
    return True
```

Choose sorting method when alphabet small and code simplicity matters; choose counting when you care about O(n) time.

2.6 Substring Search — Rabin-Karp (rolling hash) intuition

Brute-force substring search is O(n*m). Rabin-Karp computes a rolling hash for substrings of length m, allowing O(1) update from one window to next; worst-case collisions can degrade but average performance can be good. Here's a simple implementation (modular arithmetic).

```
def rabin_karp(text, pattern):
```

```
    if pattern == "": return 0
    n, m = len(text), len(pattern)
    base = 256
    mod = 10**9+7
    pat_hash = 0
    win_hash = 0
    power = 1  # base^(m-1) % mod
    for i in range(m-1):
        power = (power * base) % mod
    for i in range(m):
        pat_hash = (pat_hash*base + ord(pattern[i])) % mod
        win_hash = (win_hash*base + ord(text[i])) % mod if i < n else win_hash
    for i in range(n - m + 1):
        if pat_hash == win_hash:
            if text[i:i+m] == pattern:  # verify to avoid false positive
                return i
        if i < n - m:
            win_hash = (win_hash - ord(text[i]) * power) % mod
            win_hash = (win_hash * base + ord(text[i+m])) % mod
            win_hash = (win_hash + mod) % mod  # keep positive
    return -1

print(rabin_karp("hello world", "world"))  # 6
```

Rabin-Karp average time O(n+m), worst-case O(n*m) if many collisions (rare with good mod).

2.7 Common Pitfalls with Strings

1) Using '+' in loops (quadratic). 2) Forgetting immutability: s[i] = 'x' is invalid. 3) Confusing bytes and str when reading binary files (encode/decode). 4) Off-by-one errors in slicing and indices.

3. Practice Problems & Step-by-step Solutions (selected)

We present detailed walkthroughs for high-value problems used in interviews/contests. Follow the plain-English plan, pseudocode, then implement and analyze complexity.

Problem: Given a string, find the first non-repeating character's index (or -1 if none). Example: 'leetcode' -> 0, 'loveleetcode' -> 2.

Plan: Count frequency of each char, then scan in order to find first with freq 1.

```
def first_unique_char(s):
    # Step 1: count
    freq = {}
    for ch in s:
        freq[ch] = freq.get(ch, 0) + 1
    # Step 2: find first with freq 1
    for i, ch in enumerate(s):
        if freq[ch] == 1:
            return i
    return -1

print(first_unique_char("loveleetcode"))  # 2
```

Complexity: O(n) time, O(1) space if alphabet fixed (e.g., lowercase letters), otherwise O(k) where k distinct chars.

Problem: Given 'the sky is blue' -> 'blue is sky the'. Plan: split into words, reverse list, join with spaces. For in-place algorithms on char arrays use two reversals.

```
def reverse_words(s):
    words = s.split()
    # split() trims multiple spaces; use reversed order
    return ' '.join(reversed(words))

print(reverse_words("  the sky  is blue  "))  # 'blue is sky the'
```

Problem (classic): find longest palindromic substring. One simple method expands around centers (O(n^2) worst). Manacher's algorithm achieves O(n) but is more advanced. We'll show center expansion.

```
def longest_palindrome(s):
    def expand(l, r):
        while l >= 0 and r < len(s) and s[l] == s[r]:
            l -= 1; r += 1
        return s[l+1:r]
    res = ""
    for i in range(len(s)):
        # odd length
        t = expand(i, i)
        if len(t) > len(res):
            res = t
        # even length
        t = expand(i, i+1)
        if len(t) > len(res):
            res = t
    return res

print(longest_palindrome("babad"))  # 'bab' or 'aba'
```

Center expand complexity: O(n^2) time, O(1) extra space.

Problem: group anagrams together from list of strings. Approach: use a hashable key for each string (sorted tuple or frequency tuple).

```
from collections import defaultdict
```

```
def group_anagrams(strs):
    groups = defaultdict(list)
    for s in strs:
        key = tuple(sorted(s))  # or frequency tuple for O(n) per string
        groups[key].append(s)
    return list(groups.values())

print(group_anagrams(['eat','tea','tan','ate','nat','bat']))
# [['eat','tea','ate'], ['tan','nat'], ['bat']]
```

Sorting each string costs O(m log m) where m is string length; using frequency tuple costs O(m) with fixed alphabet.

4. Competitive Coding Tips — Practical

1) Read constraints carefully: if n ≤ 10^5, avoid O(n^2) solutions. 2) Choose the right data structure (dict for lookup, set for membership, deque for efficient pops from ends). 3) Think in steps: what to compute (counts/prefixes), how to update (sliding window), how to maintain invariants. 4) Write small helper functions (e.g., is_valid(window)), test with edge cases (empty, single element, all equal).

5. Exercises (Do these without looking at answers)

1) Implement your own split(s, sep=' ') that handles multiple consecutive separators like str.split(). 2) Given a list of numbers, return the length of the longest subarray with sum <= k (hint: prefix sums + deque for monotonic queue). 3) Given a string, return the minimum number of deletions to make it a palindrome (hint: relate to LCS).

Appendix: Quick Reference & Complexity Cheatsheet

Indexing: O(1). Slicing: O(k) to copy k elements. append: O(1) amortized. pop(): O(1) at end, O(n) at arbitrary index. insert/pop(0): O(n). Sorting: O(n log n). Joining strings: O(n).

Common libraries: collections (Counter, deque, defaultdict), itertools (combinations, permutations), io (StringIO), re (regex).

Closing Notes — How to use this guide

1) Read a concept and reproduce the short code by hand. 2) Trace the code on paper for small inputs. 3) Modify the example (add edge cases). 4) Solve the exercises and then compare. Consistency and active practice build intuition.