50 Days of Python Coding Interview Questions

Day 1: Count Unique Values in a List

Problem Statement:
Write a function to count how many unique values are in a list of integers.

Input: `nums = [1, 2, 2, 3, 4, 4, 4]`
Output: `4` (unique values: 1, 2, 3, 4)

Difficulty Level: Easy
Key Concepts Tested: Sets, Lists, Basic data cleaning

Python Solution:

```
def count_unique(nums):
    return len(set(nums))
```

Explanation: - Convert list to a `set()` to remove duplicates - Return the length of that set

> **Optimization Tip:** Set lookup is O(1), total time complexity is O(n)

Day 2: Most Frequent Element

Problem Statement:
Find the most frequent element in a list. If there's a tie, return any one.

Input: `data = [1, 2, 2, 3, 3, 3, 2]`
Output: `2` (appears 3 times)

Difficulty Level: Easy
Key Concepts Tested: Dictionaries, Counting, Data summarization

Python Solution:

```
from collections import Counter

def most_frequent(data):
    freq = Counter(data)
    return freq.most_common(1)[0][0]
```

Explanation: - Use `Counter` to count frequencies - Return the element with the highest count

Optimization Tip: Counter is optimized and readable—great for interviews

---

Day 3: Remove Outliers from Data

Problem Statement:
Given a list of numbers, remove values outside 1.5×IQR range.

Input: `nums = [10, 12, 14, 15, 18, 100]`
Output: `[10, 12, 14, 15, 18]`

Difficulty Level: Medium
Key Concepts Tested: Statistics, Data preprocessing

Python Solution:

```
import numpy as np

def remove_outliers(nums):
    q1 = np.percentile(nums, 25)
    q3 = np.percentile(nums, 75)
    iqr = q3 - q1
    lower = q1 - 1.5 * iqr
    upper = q3 + 1.5 * iqr
    return [x for x in nums if lower <= x <= upper]
```

Explanation: - Compute Q1 and Q3 percentiles using NumPy - Define the IQR (Interquartile Range) - Filter out values outside the valid range

Optimization Tip: Uses NumPy for fast vectorized operations

Day 4: Find Duplicates in a Dataset

Problem Statement:
Given a list of entries, return all items that appear more than once.

Input: `['apple', 'banana', 'apple', 'orange', 'banana']`
Output: `['apple', 'banana']`

Difficulty Level: Easy
Key Concepts Tested: Sets, Dictionaries, Deduplication

Python Solution:

```
from collections import Counter

def find_duplicates(data):
    count = Counter(data)
    return [item for item, freq in count.items() if freq > 1]
```

Explanation: - Count item frequency - Filter items where count > 1

> **Optimization Tip:** Scales well for large datasets—linear time complexity

---

Day 5: Flatten a Nested List

Problem Statement:
Given a list that may contain nested lists, return a flat list.

Input: `[1, [2, [3, 4]], 5]`
Output: `[1, 2, 3, 4, 5]`

Difficulty Level: Medium
Key Concepts Tested: Recursion, Lists, Data wrangling

Python Solution:

```
def flatten(lst):
    result = []
    for i in lst:
        if isinstance(i, list):
            result.extend(flatten(i))
        else:
            result.append(i)
    return result
```

Explanation: - Check if item is a list - If so, recursively flatten it - If not, append directly

Optimization Tip: Recursive but can be rewritten iteratively for large inputs

---

Day 6: Convert String Dates to Datetime Objects

Problem Statement:
Convert a list of date strings (YYYY-MM-DD) into Python datetime objects.

Input: `['2023-05-01', '2023-06-10']`
Output: `[datetime.date(2023, 5, 1), datetime.date(2023, 6, 10)]`

Difficulty Level: Easy
Key Concepts Tested: datetime, String parsing, Data formatting

Python Solution:

```
from datetime import datetime

def convert_dates(dates):
    return [datetime.strptime(date, "%Y-%m-%d").date() for date in dates]
```

Explanation: - Use `datetime.strptime()` to parse strings - Convert to `date()` object if time isn't needed

Optimization Tip: List comprehension improves readability and performance

---

Day 7: Group Data by Key

Problem Statement:
Group a list of dictionaries by a given key.

Input:

```
data = [{'city': 'NY', 'val': 1}, {'city': 'LA', 'val': 2}, {'city': 'NY', 'val': 3}]
```

Output: `{'NY': [1, 3], 'LA': [2]}`

Difficulty Level: Medium
Key Concepts Tested: Grouping, Dictionaries, Looping structures

Python Solution:

```
from collections import defaultdict

def group_by_key(data, key, value):
    grouped = defaultdict(list)
    for item in data:
        grouped[item[key]].append(item[value])
    return dict(grouped)
```

Explanation: - Use `defaultdict(list)` for easier grouping - Loop through each dictionary and group values

> **Optimization Tip:** Avoids conditionals by using defaultdict

---

Day 8: Calculate Moving Average

Problem Statement:
Given a list of numbers and a window size k, return a list of moving averages.

Input: `nums = [1, 2, 3, 4, 5]`, `k = 3`
Output: `[2.0, 3.0, 4.0]`

Difficulty Level: Medium
Key Concepts Tested: Sliding Window, Lists, Averages

Python Solution:

```
def moving_average(nums, k):
    return [sum(nums[i:i+k]) / k for i in range(len(nums) - k + 1)]
```

Explanation: - Slide a window of size k across the list - Calculate and store the average of each window

> **Optimization Tip:** Can use cumulative sums for O(1) per window average

---

Day 9: Detect Missing Values

Problem Statement:
Given a list of values, return the indices where the value is None or NaN.

Input: `[1, None, 3, float('nan'), 5]`
Output: `[1, 3]`

Difficulty Level: Easy
Key Concepts Tested: Null handling, NaN, List traversal

Python Solution:

```
import math

def find_missing(data):
    return [i for i, x in enumerate(data)
            if x is None or (isinstance(x, float) and math.isnan(x))]
```

Explanation: - Check for `None` directly - Use `math.isnan()` for NaN detection

> **Optimization Tip:** Handles both Python and NumPy-style missing values robustly

---

Day 10: Count Word Frequencies in Text

Problem Statement:
Given a block of text, return the frequency of each word.

Input: `"data is power and data drives insight"`
Output: `{'data': 2, 'is': 1, 'power': 1, 'and': 1, 'drives': 1, 'insight': 1}`

Difficulty Level: Easy
Key Concepts Tested: Strings, Text cleaning, Dictionaries

Python Solution:

```
from collections import Counter

def word_frequencies(text):
    words = text.lower().split()
    return dict(Counter(words))
```

Explanation: - Lowercase all text for uniformity - Split by whitespace and count occurrences

> **Optimization Tip:** For production use, consider regex-based tokenization

---

Day 11: Reverse a String Without Built-in Function

Problem Statement:
Reverse a string manually (without using `[::-1]` or `reversed`).

Input: `"analytics"`
Output: `"scitylana"`

Difficulty Level: Easy
Key Concepts Tested: Strings, Looping

Python Solution:

```
def reverse_string(s):
    result = ''
    for char in s:
        result = char + result
    return result
```

Explanation: - Loop through each character - Prepend each character to build reverse

Optimization Tip: Strings are immutable, so `''.join(reversed())` is more efficient in real use

---

Day 12: Check for Palindrome

Problem Statement:
Check if a given string is a palindrome (ignoring case and spaces).

Input: `"Madam"`
Output: `True`

Difficulty Level: Easy
Key Concepts Tested: Strings, Data cleaning

Python Solution:

```
def is_palindrome(s):
    cleaned = s.replace(" ", "").lower()
    return cleaned == cleaned[::-1]
```

Explanation: - Remove spaces and convert to lowercase - Compare string with its reverse

Optimization Tip: For large strings, use two-pointer approach for O(n) space

---

Day 13: Extract Domain from Email

Problem Statement:
Extract domain name from an email address.

Input: `"user@openai.com"`
Output: `"openai.com"`

Difficulty Level: Easy
Key Concepts Tested: Strings, Splitting

Python Solution:

```
def extract_domain(email):
    return email.split('@')[1]
```

Explanation: - Use `split('@')` to separate user from domain

Optimization Tip: Add validation checks in real-world usage

Day 14: Merge Two Sorted Lists

Problem Statement:
Merge two already sorted lists into one sorted list.

Input: `[1,3,5]` and `[2,4,6]`
Output: `[1,2,3,4,5,6]`

Difficulty Level: Medium
Key Concepts Tested: Lists, Pointers, Sorting

Python Solution:

```
def merge_sorted(a, b):
    i = j = 0
    result = []
    while i < len(a) and j < len(b):
        if a[i] < b[j]:
            result.append(a[i])
            i += 1
        else:
            result.append(b[j])
            j += 1

    return result + a[i:] + b[j:]
```

Explanation: - Use two pointers to compare and merge - Append remaining elements after loop

Optimization Tip: Linear time: O(n + m)

Day 15: Check If Two Strings Are Anagrams

Problem Statement:
Check if two strings are anagrams of each other.

Input: `"listen"`, `"silent"`
Output: `True`

Difficulty Level: Easy
Key Concepts Tested: Sorting, Strings

Python Solution:

```
def is_anagram(a, b):
    return sorted(a) == sorted(b)
```

Explanation: - Sort both strings and compare

Optimization Tip: Use Counter for better performance with large strings

---

Day 16: Find the Intersection of Two Lists

Problem Statement:
Return elements common to both lists.

Input: `[1,2,3]` and `[2,3,4]`
Output: `[2,3]`

Difficulty Level: Easy
Key Concepts Tested: Sets, Lists

Python Solution:

```
def intersect(a, b):
    return list(set(a) & set(b))
```

Explanation: - Convert to sets and use `&` operator

Optimization Tip: Time complexity: O(n + m)

Day 17: Find Maximum Subarray Sum (Kadane's Algorithm)

Problem Statement:
Find the contiguous subarray with the maximum sum.

Input: `[-2,1,-3,4,-1,2,1,-5,4]`
Output: `6` (from subarray `[4,-1,2,1]`)

Difficulty Level: Hard
Key Concepts Tested: Arrays, Dynamic Programming

Python Solution:

```
def max_subarray(nums):
    max_sum = current = nums[0]
    for num in nums[1:]:
        current = max(num, current + num)
        max_sum = max(max_sum, current)
    return max_sum
```

Explanation: - Track max subarray sum using dynamic approach - Compare current value vs sum continuation

Optimization Tip: O(n) time, O(1) space

Day 18: Validate a Password

Problem Statement:
Check if a password meets the following criteria: - At least 8 characters - Includes a digit - Includes an uppercase letter

Input: `"Data2023"`
Output: `True`

Difficulty Level: Medium
Key Concepts Tested: Strings, Regex, Validation

Python Solution:

```
import re

def is_valid(password):
    return (len(password) >= 8 and
            re.search(r'\d', password) and
            re.search(r'[A-Z]', password))
```

Explanation: - Use `re.search()` to validate conditions

Optimization Tip: Regex gives flexibility for rule changes

Day 19: Remove Duplicates from List of Dicts

Problem Statement:
Remove duplicate dictionaries based on a specific key.

Input: `[{'id':1}, {'id':2}, {'id':1}]`
Output: `[{'id':1}, {'id':2}]`

Difficulty Level: Medium
Key Concepts Tested: Sets, Dictionaries, Uniqueness

Python Solution:

```
def remove_dupes(data):
    seen = set()
    result = []
    for d in data:
        if d['id'] not in seen:
            seen.add(d['id'])
            result.append(d)
    return result
```

Explanation: - Use a set to track seen IDs

Optimization Tip: Works best with hashable keys like strings/ints

---

Day 20: Convert JSON String to Dictionary

Problem Statement:
Convert a JSON string to a Python dictionary.

Input: `'{"name": "Alice", "age": 30}'`
Output: `{'name': 'Alice', 'age': 30}`

Difficulty Level: Easy
Key Concepts Tested: JSON parsing, json module

Python Solution:

```
import json

def parse_json(json_str):
    return json.loads(json_str)
```

Explanation: - Use `json.loads()` to parse the string

Optimization Tip: Always wrap in try-except for production safety

---

Day 21: Find Most Frequent Element

Problem Statement:
Return the element that appears most frequently in a list.

Input: `[1, 2, 2, 3, 1, 2]`
Output: `2`

Difficulty Level: Medium
Key Concepts Tested: Dictionaries, Frequency counting

Python Solution Code:

```
from collections import Counter

def most_frequent(lst):
    return Counter(lst).most_common(1)[0][0]
```

Explanation: - Use `Counter` to count frequencies of all elements - `most_common(1)` returns the most frequent element as a tuple - Extract the element from the tuple

Optimization Tip:
✔ O(n) time complexity with Counter - efficient and readable

Call to Action:
Instagram: "Save this counting trick "
YouTube: "What's your most used Counter use case?"

Day 22: Convert String to Datetime

Problem Statement:
Convert a date string to a Python datetime object.

Input: `"2023-10-01"`
Output: `datetime(2023, 10, 1, 0, 0)`

Difficulty Level: Easy
Key Concepts Tested: Datetime parsing, String formatting

Python Solution Code:

```
from datetime import datetime

def to_date(date_string):
    return datetime.strptime(date_string, "%Y-%m-%d")
```

Explanation: - `strptime()` parses string according to specified format - `%Y-%m-%d` matches year-month-day format

Optimization Tip:
✔ Always validate format before parsing in production

Call to Action:
Instagram: "What's your date format? "
YouTube: "Comment your region's date format!"

Day 23: Get Top N Largest Values

Problem Statement:
Return the N largest numbers from a list.

Input: `[4, 1, 7, 3], N=2`
Output: `[7, 4]`

Difficulty Level: Medium
Key Concepts Tested: Sorting, Heaps, Data structures

Python Solution Code:

```
import heapq

def top_n_largest(nums, n):
    return heapq.nlargest(n, nums)
```

Explanation: - `heapq.nlargest()` efficiently finds N largest elements - More efficient than sorting the entire list

Optimization Tip:
✔ O(n log k) complexity - better than full sort for small k

Call to Action:
Instagram: "Save this for data crunching "
YouTube: "Drop your test list in comments!"

Day 24: Replace Null Values in Dictionary

Problem Statement:
Replace all None values in a dictionary with a default value.

Input: `{'a': None, 'b': 2}, default=0`
Output: `{'a': 0, 'b': 2}`

Difficulty Level: Easy
Key Concepts Tested: Dictionaries, Conditional logic

Python Solution Code:

```
def replace_nulls(dictionary, default=0):
    return {k: (v if v is not None else default) for k, v in dictionary.items()}
```

Explanation: - Dictionary comprehension with conditional expression - Preserves non-None values, replaces None with default

Optimization Tip:
✔ Fast and readable one-liner solution

Call to Action:
Instagram: "Save your default dict tip "
YouTube: "What's your go-to default value?"

Day 25: Flatten a Nested List

Problem Statement:
Convert a nested list structure into a single flat list.

Input: `[[1, 2], [3, 4], [5]]`
Output: `[1, 2, 3, 4, 5]`

Difficulty Level: Medium
Key Concepts Tested: List comprehension, Nested iteration

Python Solution Code:

```
def flatten_list(nested_list):
    return [item for sublist in nested_list for item in sublist]
```

Explanation: - Double for-loop in list comprehension - Iterates through each sublist, then each item

Optimization Tip:
✔ Works for 2D lists; use recursion for deeper nesting

Call to Action:
Instagram: "Flatten that data! "
YouTube: "Subscribe for Day 26!"

Day 26: Check if List is Sorted

Problem Statement:
Determine if a list is sorted in ascending order.

Input: `[1, 2, 3, 4]`
Output: `True`

Difficulty Level: Easy
Key Concepts Tested: List comparison, Sorting validation

Python Solution Code:

```
def is_sorted_ascending(lst):
    return lst == sorted(lst)
```

Explanation: - Compare original list with its sorted version - Returns True if they match

Optimization Tip:
✔ For large lists, use custom loop for O(n) without extra space

Call to Action:
Instagram: "Sorted or not? "
YouTube: "Try it with descending order too!"

Day 27: Count Missing Values in List

Problem Statement:
Count the number of None values in a list.

Input: `[1, None, 2, None, 3]`
Output: `2`

Difficulty Level: Easy
Key Concepts Tested: List traversal, None detection

Python Solution Code:

```
def count_missing_values(lst):
    return lst.count(None)
```

Explanation: - Use built-in `count()` method for simple counting - Efficiently counts occurrences of None

Optimization Tip:
✔ For complex conditions, use `sum(x is None for x in lst)`

Call to Action:
Instagram: "Check your data cleanliness "
YouTube: "Drop a if you clean data daily!"

Day 28: Rename Dictionary Keys

Problem Statement:
Rename dictionary keys using a mapping dictionary.

Input: `{'a': 1, 'b': 2}, mapping: {'a': 'alpha'}`
Output: `{'alpha': 1, 'b': 2}`

Difficulty Level: Medium
Key Concepts Tested: Dictionary manipulation, Key mapping

Python Solution Code:

```
def rename_dictionary_keys(dictionary, key_mapping):
    return {key_mapping.get(k, k): v for k, v in dictionary.items()}
```

Explanation: - Use `get()` method to replace key if mapping exists - Falls back to original key if no mapping found

Optimization Tip:
✔ Handles missing mappings gracefully with default behavior

Call to Action:
Instagram: "Customize your keys! "
YouTube: "What's your favorite dict trick?"

Day 29: Detect Duplicates in List

Problem Statement:
Check if a list contains any duplicate values.

Input: `[1, 2, 2, 3]`
Output: `True`

Difficulty Level: Easy
Key Concepts Tested: Set operations, Duplicate detection

Python Solution Code:

```
def has_duplicates(lst):
    return len(lst) != len(set(lst))
```

Explanation: - Convert to set to remove duplicates - Compare lengths to detect if duplicates existed

Optimization Tip:
✔ O(n) time complexity, very efficient

Call to Action:
Instagram: "Save this clean check "
YouTube: "Drop a if you hate duplicates!"

Day 30: Parse and Clean Column Names

Problem Statement:
Clean a list of column names by converting to lowercase and replacing spaces with underscores.

Input: `["First Name", "Last Name", "Age"]`
Output: `["first_name", "last_name", "age"]`

Difficulty Level: Medium
Key Concepts Tested: String manipulation, List comprehension

Python Solution Code:

```
def clean_column_names(columns):
    return [col.strip().lower().replace(" ", "_") for col in columns]
```

Explanation: - Chain string methods: strip whitespace, convert to lowercase, replace spaces - Apply transformation to all column names

Optimization Tip:
✔ Method chaining is efficient and readable

Call to Action:
Instagram: "Clean data is happy data 😊"
YouTube: "Share your cleanest dataset name!"

Day 31: Merge Two Sorted Lists

Problem Statement:
Merge two already sorted lists into one sorted list efficiently.

Input: `[1, 3, 5]` and `[2, 4, 6]`
Output: `[1, 2, 3, 4, 5, 6]`

Difficulty Level: Medium
Key Concepts Tested: Two-pointer technique, Merging algorithms

Python Solution Code:

```
def merge_sorted_lists(list1, list2):
    result = []
    i = j = 0

    # Compare elements from both lists
    while i < len(list1) and j < len(list2):
        if list1[i] <= list2[j]:
            result.append(list1[i])
            i += 1
        else:
            result.append(list2[j])
            j += 1

    # Add remaining elements
    result.extend(list1[i:])
    result.extend(list2[j:])

    return result
```

Explanation: - Use two pointers to compare elements from both lists - Add smaller element to result and advance corresponding pointer - Append remaining elements after one list is exhausted

Optimization Tip:
✔ O(n + m) linear time complexity - optimal for this problem

Call to Action:
Instagram: "Save this merge pattern! "
YouTube: "Subscribe for Day 32 tomorrow!"

Day 32: Count Word Frequencies in Text

Problem Statement:
Count the frequency of each word in a given text string.

Input: `"data science is powerful and data drives insights"`
Output: `{'data': 2, 'science': 1, 'is': 1, 'powerful': 1, 'and': 1, 'drives': 1, 'insights': 1}`

Difficulty Level: Easy
Key Concepts Tested: String processing, Dictionary operations

Python Solution Code:

```
def count_word_frequencies(text):
    words = text.lower().split()
    frequency = {}

    for word in words:
        frequency[word] = frequency.get(word, 0) + 1

    return frequency
```

Explanation: - Convert to lowercase for case-insensitive counting - Split text into words by whitespace - Use `get()` method to handle new words gracefully

Optimization Tip:
✔ Use `collections.Counter` for more concise code

Call to Action:
Instagram: "Try this with your favorite quote! "
YouTube: "Comment your word count results!"

Day 33: Find Top K Frequent Elements

Problem Statement:
Given a list of elements, return the k most frequently occurring elements.

Input: `[1, 1, 1, 2, 2, 3], k=2`
Output: `[1, 2]`

Difficulty Level: Medium
Key Concepts Tested: Frequency counting, Heap operations

Python Solution Code:

```
from collections import Counter

def top_k_frequent_elements(nums, k):
    count = Counter(nums)
    return [item for item, freq in count.most_common(k)]
```

Explanation: - Count frequencies using Counter - Use `most_common(k)` to get top k frequent elements - Extract just the elements from the (element, frequency) tuples

Optimization Tip:
✔ O(n log k) complexity using heap - efficient for large datasets

Call to Action:
Instagram: "Find your top patterns! "
YouTube: "What are your top 3 most used Python functions?"

Day 34: Remove Duplicates Preserving Order

Problem Statement:
Remove duplicates from a list while preserving the original order of first occurrences.

Input: `[1, 2, 2, 3, 1, 4]`
Output: `[1, 2, 3, 4]`

Difficulty Level: Easy
Key Concepts Tested: Set operations, Order preservation

Python Solution Code:

```
def remove_duplicates_preserve_order(lst):
    seen = set()
    result = []

    for item in lst:
        if item not in seen:
            seen.add(item)
            result.append(item)

    return result
```

Explanation: - Use a set to track seen elements - Only add to result if not previously seen - Maintains original order of first occurrences

Optimization Tip:
✔ O(n) time complexity with O(n) space - optimal solution

Call to Action:
Instagram: "Keep it unique! "
YouTube: "Tag someone who needs cleaner data!"

Day 35: Flatten Deeply Nested List

Problem Statement:
Flatten a list that may contain multiple levels of nesting.

Input: `[1, [2, [3, 4]], 5, [6]]`
Output: `[1, 2, 3, 4, 5, 6]`

Difficulty Level: Medium
Key Concepts Tested: Recursion, Deep data structures

Python Solution Code:

```
def flatten_deep_list(lst):
    result = []

    for item in lst:
        if isinstance(item, list):
            result.extend(flatten_deep_list(item))
        else:
            result.append(item)

    return result
```

Explanation: - Recursively check if each item is a list - If it's a list, recursively flatten it and extend result - If not a list, append directly to result

Optimization Tip:
✔ Handles arbitrary nesting depth - very flexible solution

Call to Action:
Instagram: "Recursion magic! "
YouTube: "Save if you love recursive solutions!"

Day 36: Find All Anagrams in String

Problem Statement:
Find all starting indices where anagrams of pattern p occur in string s.

Input: `s="abab"`, `p="ab"`
Output: `[0, 2]`

Difficulty Level: Medium
Key Concepts Tested: Sliding window, Character frequency

Python Solution Code:

```
from collections import Counter

def find_anagrams_in_string(s, p):
    if len(p) > len(s):
        return []

    p_count = Counter(p)
    window_count = Counter()
    result = []

    for i in range(len(s)):
        # Add current character to window
        window_count[s[i]] += 1

        # Remove character that's outside window
        if i >= len(p):
            if window_count[s[i - len(p)]] == 1:
                del window_count[s[i - len(p)]]
            else:
                window_count[s[i - len(p)]] -= 1

        # Check if current window is an anagram
        if window_count == p_count:
            result.append(i - len(p) + 1)

    return result
```

Explanation: - Use sliding window technique with character counting - Maintain frequency count of current window - Compare window count with pattern count

Optimization Tip:
✔ O(n) time complexity with efficient sliding window

Call to Action:
Instagram: "Master the sliding window! "
YouTube: "Subscribe for more algorithm patterns!"

Day 37: Reverse Words in String

Problem Statement:
Reverse the order of words in a given string.

Input: `"data science is awesome"`
Output: `"awesome is science data"`

Difficulty Level: Easy
Key Concepts Tested: String manipulation, List operations

Python Solution Code:

```
def reverse_words_in_string(s):
    return ' '.join(s.strip().split()[::-1])
```

Explanation: - Strip whitespace from both ends - Split into words (handles multiple spaces) - Reverse the list of words and join back

Optimization Tip:
✔ Handles multiple spaces and edge cases elegantly

Call to Action:
Instagram: "Flip your words! "
YouTube: "Try it with your favorite sentence!"

Day 38: Check if String is Palindrome

Problem Statement:
Check if a string is a palindrome, ignoring case, spaces, and punctuation.

Input: `"A man, a plan, a canal: Panama"`
Output: `True`

Difficulty Level: Easy
Key Concepts Tested: String normalization, Palindrome detection

Python Solution Code:

```
def is_palindrome_string(s):
    # Keep only alphanumeric characters and convert to lowercase
    cleaned = ''.join(char.lower() for char in s if char.isalnum())
    return cleaned == cleaned[::-1]
```

Explanation: - Filter to keep only alphanumeric characters - Convert to lowercase for case-insensitive comparison - Compare string with its reverse

Optimization Tip:
✔ Two-pointer approach can save space for very large strings

Call to Action:
Instagram: "Test your favorite phrase! "
YouTube: "Did your name pass the palindrome test?"

Day 39: Convert Excel Column Title to Number

Problem Statement:
Convert Excel column titles (like 'A', 'B', 'AA') to their corresponding column numbers.

Input: `"AB"`
Output: `28`

Difficulty Level: Medium
Key Concepts Tested: Base conversion, Mathematical operations

Python Solution Code:

```
def excel_column_to_number(column_title):
    result = 0
    for char in column_title:
        result = result * 26 + (ord(char.upper()) - ord('A') + 1)

    return result
```

Explanation: - Treat as base-26 number system (A=1, B=2, etc.) - Convert each character to its numeric value - Build result using positional notation

Optimization Tip:
✔ O(n) where n is length of column title - optimal solution

Call to Action:
Instagram: "Excel wizardry! "
YouTube: "Save for your next spreadsheet project!"

Day 40: Validate Parentheses

Problem Statement:
Check if a string containing parentheses, brackets, and braces is properly balanced.

Input: `"([{}])"`
Output: `True`

Difficulty Level: Medium
Key Concepts Tested: Stack data structure, String validation

Python Solution Code:

```
def is_valid_parentheses(s):
    stack = []
    mapping = {')': '(', '}': '{', ']': '['}

    for char in s:
        if char in mapping.values():  # Opening bracket
            stack.append(char)
        elif char in mapping:  # Closing bracket
            if not stack or mapping[char] != stack.pop():
                return False

    return not stack  # True if stack is empty
```

Explanation: - Use stack to track opening brackets - For each closing bracket, check if it matches the most recent opening - String is valid if stack is empty at the end

Optimization Tip:
✔ O(n) time and space complexity - classic stack application

Call to Action:
Instagram: "Balance your brackets! ⚖"
YouTube: "Tag a friend who loves algorithms!"

Day 41: Group Anagrams

Problem Statement:
Group a list of strings into anagrams (words with the same letters rearranged).

Input: `["eat", "tea", "tan", "ate", "nat", "bat"]`
Output: `[["eat", "tea", "ate"], ["tan", "nat"], ["bat"]]`

Difficulty Level: Medium
Key Concepts Tested: Hashing, Strings, Lists, Dictionary Operations

Python Solution Code:

```
from collections import defaultdict

def group_anagrams(strs):
    anagrams = defaultdict(list)
    for word in strs:
        key = tuple(sorted(word))
        anagrams[key].append(word)
    return list(anagrams.values())
```

Explanation: - Sort characters in each word to create a unique key for anagrams - Use defaultdict to group words with the same sorted character pattern - Return grouped anagram lists

Optimization Tip:
✔ Time complexity: O(n × m log m) where n is number of words, m is average word length

Call to Action:
Instagram: "Save this for string manipulation! "
YouTube: "Try it and comment your results!"

Day 42: Longest Consecutive Sequence

Problem Statement:
Find the length of the longest consecutive elements sequence in an unsorted array.

Input: `[100, 4, 200, 1, 3, 2]`
Output: `4` (sequence: [1, 2, 3, 4])

Difficulty Level: Medium
Key Concepts Tested: Sets, Sequence Detection, Optimization

Python Solution Code:

```
def longest_consecutive(nums):
    num_set = set(nums)
    longest = 0

    for n in num_set:
        # Only start counting from the beginning of a sequence
        if n - 1 not in num_set:
            length = 1
            while n + length in num_set:
                length += 1
            longest = max(longest, length)

    return longest
```

Explanation: - Convert to set for O(1) lookups - Only start sequence counting from numbers that don't have a predecessor - Track the maximum sequence length found

Optimization Tip:
✔ O(n) time complexity despite nested loops - each number is visited at most twice

Call to Action:
Instagram: "Master sequence problems! "
YouTube: "Subscribe for more algorithm content!"

Day 43: Find Peak Element

Problem Statement:
Find an element in an array that is greater than its neighbors. Array may have multiple peaks.

Input: `[1, 2, 3, 1]`
Output: `2` (index of peak element 3)

Difficulty Level: Medium
Key Concepts Tested: Binary Search, Array Traversal

Python Solution Code:

```
def find_peak(nums):
    left, right = 0, len(nums) - 1

    while left < right:
        mid = (left + right) // 2
        if nums[mid] > nums[mid + 1]:
            right = mid
        else:
            left = mid + 1

    return left
```

Explanation: - Use binary search to efficiently find a peak - Compare middle element with its right neighbor - Move towards the side that's likely to contain a peak

Optimization Tip:
✔ O(log n) time complexity using binary search instead of O(n) linear scan

Call to Action:
Instagram: "Binary search mastery! "
YouTube: "Comment your peak finding strategy!"

Day 44: Find Duplicates in DataFrame Column

Problem Statement:
Identify and return all duplicate rows in a pandas DataFrame based on a specific column.

Input: DataFrame with duplicate values in 'email' column
Output: DataFrame containing all rows with duplicate emails

Difficulty Level: Easy
Key Concepts Tested: Pandas, Data Analysis, Duplicate Detection

Python Solution Code:

```
import pandas as pd

def find_duplicates(df, column):
    return df[df.duplicated([column], keep=False)]

# Alternative: Get only the duplicate values
def get_duplicate_values(df, column):
    return df[df[column].duplicated(keep=False)][column].unique()
```

Explanation: - Use `.duplicated()` with `keep=False` to mark all duplicates (not just subsequent ones) - Filter DataFrame to return only rows with duplicates - Alternative method shows just the duplicate values

Optimization Tip:
✔ Use `keep='first'` or `keep='last'` to keep only one instance of duplicates

Call to Action:
Instagram: "Clean your data like a pro! "
YouTube: "Share your data cleaning tips!"

Day 45: Drop Missing Data

Problem Statement:
Remove rows with missing values from a pandas DataFrame with flexible options.

Input: DataFrame with NaN values
Output: Cleaned DataFrame without missing data

Difficulty Level: Easy
Key Concepts Tested: Pandas, Data Cleaning, Missing Value Handling

Python Solution Code:

```
import pandas as pd

def drop_missing(df, strategy='any'):
    if strategy == 'any':
        return df.dropna()  # Drop if any column has NaN
    elif strategy == 'all':
        return df.dropna(how='all')  # Drop only if all columns are NaN
    elif strategy == 'threshold':
        return df.dropna(thresh=len(df.columns)//2)  # Keep if at least half columns hav
```

```
# Drop missing from specific columns only
def drop_missing_columns(df, columns):
    return df.dropna(subset=columns)
```

Explanation: - `dropna()` removes rows with any NaN values by default - `how='all'` only drops rows where all values are NaN - `thresh` parameter sets minimum number of non-null values required - `subset` parameter focuses on specific columns

Optimization Tip:
✔ Consider imputation strategies before dropping valuable data

Call to Action:
Instagram: "Clean data = better insights! "
YouTube: "What's your favorite data cleaning method?"

Day 46: Find Correlation Matrix

Problem Statement:
Compute and analyze the correlation matrix of numerical columns in a DataFrame.

Input: DataFrame with numerical columns
Output: Correlation matrix showing relationships between variables

Difficulty Level: Easy
Key Concepts Tested: Pandas, Statistical Analysis, Correlation

Python Solution Code:

```
import pandas as pd
import numpy as np

def correlation_matrix(df, method='pearson'):
    # Basic correlation matrix
    corr_matrix = df.corr(method=method)
    return corr_matrix

def analyze_correlations(df, threshold=0.7):
    corr_matrix = df.corr()

    # Find highly correlated pairs
    high_corr = []
    for i in range(len(corr_matrix.columns)):
        for j in range(i+1, len(corr_matrix.columns)):
            if abs(corr_matrix.iloc[i, j]) > threshold:
```

```python
                high_corr.append({
                    'var1': corr_matrix.columns[i],
                    'var2': corr_matrix.columns[j],
                    'correlation': corr_matrix.iloc[i, j]
                })

    return pd.DataFrame(high_corr)
```

Explanation: - `df.corr()` computes pairwise correlation of columns - Methods available: 'pearson', 'kendall', 'spearman' - Enhanced function identifies highly correlated variable pairs - Useful for feature selection and multicollinearity detection

Optimization Tip:
✔ Visualize with seaborn heatmap for better interpretation

Call to Action:
Instagram: "Discover hidden relationships in your data! "
YouTube: "Show us your correlation insights!"

Day 47: Apply Function to Column

Problem Statement:
Apply custom transformations to DataFrame columns using various methods.

Input: DataFrame with columns needing transformation
Output: DataFrame with transformed values

Difficulty Level: Easy
Key Concepts Tested: Pandas, Lambda Functions, Data Transformation

Python Solution Code:

```python
import pandas as pd

def apply_function(df, column, func=None):
    if func is None:
        func = lambda x: x * 2  # Default: double the values

    df_copy = df.copy()
    df_copy[column] = df_copy[column].apply(func)
    return df_copy

# Multiple transformation examples
```

```
def advanced_transformations(df):
    df_transformed = df.copy()

    # Apply different functions to different columns
    transformations = {
        'salary': lambda x: x * 1.1,  # 10% increase
        'age': lambda x: 2024 - x if x > 1900 else x,  # Convert birth year to age
        'name': lambda x: x.title(),  # Title case
        'email': lambda x: x.lower()  # Lowercase
    }

    for col, func in transformations.items():
        if col in df_transformed.columns:
            df_transformed[col] = df_transformed[col].apply(func)

    return df_transformed
```

Explanation: - `apply()` method applies a function to each element in a series - Lambda functions provide concise inline transformations - Dictionary-based approach allows multiple column transformations - Always work on copies to preserve original data

Optimization Tip:
✔ Use vectorized operations when possible for better performance than apply()

Call to Action:
Instagram: "Transform your data with style! "
YouTube: "What's your favorite pandas transformation?"

Day 48: Rename DataFrame Columns

Problem Statement:
Rename DataFrame columns using various strategies for better data management.

Input: DataFrame with unclear or inconsistent column names
Output: DataFrame with clean, standardized column names

Difficulty Level: Easy
Key Concepts Tested: Pandas, Data Preprocessing, Column Management

Python Solution Code:

```
import pandas as pd
import re
```

```python
def rename_columns(df, col_map):
    return df.rename(columns=col_map)

def clean_column_names(df):
    """Standardize column names: lowercase, underscores, no spaces"""
    df_clean = df.copy()

    # Clean column names
    new_columns = []
    for col in df_clean.columns:
        # Convert to lowercase, replace spaces/special chars with underscores
        clean_col = re.sub(r'[^a-zA-Z0-9]', '_', str(col).lower())
        # Remove multiple underscores
        clean_col = re.sub(r'_+', '_', clean_col)
        # Remove leading/trailing underscores
        clean_col = clean_col.strip('_')
        new_columns.append(clean_col)

    df_clean.columns = new_columns
    return df_clean

def smart_rename(df, patterns):
    """Rename based on patterns"""
    rename_dict = {}
    for old_pattern, new_name in patterns.items():
        for col in df.columns:
            if old_pattern.lower() in col.lower():
                rename_dict[col] = new_name

    return df.rename(columns=rename_dict)
```

Explanation: - `rename()` method accepts a dictionary mapping old names to new names - `clean_column_names()` standardizes names following Python conventions - `smart_rename()` uses pattern matching for bulk renaming - Regular expressions help handle complex naming patterns

Optimization Tip:
✔ Establish naming conventions early in your data pipeline

Call to Action:
Instagram: "Clean columns = clean code! "
YouTube: "Share your column naming conventions!"

Day 49: Combine Multiple DataFrames

Problem Statement:
Efficiently combine multiple DataFrames using various merging strategies.

Input: List of DataFrames to combine
Output: Single combined DataFrame

Difficulty Level: Medium
Key Concepts Tested: Pandas, Data Concatenation, Merging Strategies

Python Solution Code:

```python
import pandas as pd

def concat_dfs(dfs, method='vertical'):
    if method == 'vertical':
        return pd.concat(dfs, ignore_index=True)
    elif method == 'horizontal':
        return pd.concat(dfs, axis=1)

def smart_combine_dfs(dfs, on_column=None):
    """Intelligently combine DataFrames"""
    if not dfs:
        return pd.DataFrame()

    if len(dfs) == 1:
        return dfs[0]

    # If joining on a column, use merge
    if on_column:
        result = dfs[0]
        for df in dfs[1:]:
            result = pd.merge(result, df, on=on_column, how='outer')
        return result

    # Otherwise, concatenate vertically
    return pd.concat(dfs, ignore_index=True, sort=False)

def combine_with_source(dfs, source_names):
    """Add source identifier when combining"""
    combined_dfs = []

    for df, source in zip(dfs, source_names):
        df_copy = df.copy()
        df_copy['source'] = source
        combined_dfs.append(df_copy)
    return pd.concat(combined_dfs, ignore_index=True)
```

Explanation: - `pd.concat()` stacks DataFrames vertically (axis=0) or horizontally (axis=1) - `ignore_index=True` creates new sequential index - `pd.merge()` joins DataFrames on common columns - Source tracking helps maintain data lineage

Optimization Tip:
✔ Use `pd.merge()` for database-style joins, `pd.concat()` for simple stacking

Call to Action:
Instagram: "Master data combination! "
YouTube: "Tag your data engineering friends!"

Day 50: Group By and Aggregate

Problem Statement:
Perform advanced grouping and aggregation operations on DataFrame data.

Input: DataFrame with categorical and numerical columns
Output: Aggregated results grouped by categories

Difficulty Level: Medium
Key Concepts Tested: Pandas, GroupBy Operations, Data Aggregation

Python Solution Code:

```
import pandas as pd
import numpy as np

def group_and_aggregate(df, by_col, agg_col, agg_func='sum'):
    return df.groupby(by_col)[agg_col].agg(agg_func).reset_index()

def advanced_groupby(df, group_cols, agg_dict):
    """Perform multiple aggregations"""
    return df.groupby(group_cols).agg(agg_dict).reset_index()

def comprehensive_analysis(df, group_col, numeric_cols):
    """Complete statistical analysis by group"""

    # Multiple aggregations for each numeric column
    agg_functions = ['count', 'mean', 'median', 'std', 'min', 'max']

    results = {}
    for col in numeric_cols:
        if col in df.columns:
            results[col] = agg_functions

    summary = df.groupby(group_col).agg(results)

    # Flatten column names
    summary.columns = ['_'.join(col).strip() for col in summary.columns.values]
    summary = summary.reset_index()

    return summary

def custom_aggregations(df):
    """Examples of custom aggregation functions"""
    return df.groupby('category').agg({
        'sales': ['sum', 'mean', lambda x: x.max() - x.min()],  # Range
        'quantity': ['count', 'std'],
        'date': ['min', 'max']  # Date range
    })
```

Explanation: - `groupby()` creates groups based on column values - `agg()` applies aggregation functions to grouped data - Multiple aggregations can be applied simultaneously - Custom lambda functions enable specialized calculations - `reset_index()` converts grouped result back to regular DataFrame

Optimization Tip:
✔ Use vectorized aggregation functions for better performance on large datasets

---

Thank you for following along on this coding journey!