

TIP102 | Intermediate Technical Interview Prep

Intermediate Technical Interview Prep Spring 2025 (@ Section 3 | Tuesdays and Thursdays 6PM - 8PM PT)

Personal Member ID#: 117667

Session 1: Dictionaries

Session Overview

The focus of this session is advanced data handling in Python, focusing on functions, lists, strings, and dictionaries. Students will tackle practical programming challenges such as verifying subsequences, creating and manipulating dictionaries, and calculating values based on dynamic inputs.

The tasks are designed to enhance understanding of data structures, algorithmic thinking, and conditional logic, preparing students for more complex problem-solving scenarios involving data manipulation and retrieval.

You can find all resources from today including session slide decks, session recordings, and more on the resources tab



Part 1 : Instructor Led Session

We'll spend the first portion of the synchronous class time in large groups, where the instructor will lead class instruction for 30-45 minutes.



Part 2: Breakout Session

In breakout sessions, we will explore and collaboratively solve problem sets in small groups. Here, the **collaboration, conversation, and approach** are just as important as “solving the problem” - please engage warmly, clearly, and plentifully in the process!

In breakout rooms you will:

- Screen-share the problem/s, and verbally review them together
- Screen-share an interactive coding environment, and talk through the steps of a solution approach
 - ProTip: - An Integrated Development Environment (IDE) is a fancy name for a tool you could use for shared writing of code - like Replit.com, Collabed.it, CodePen.io, or other - your staff team will specify which tool to use for this class!

- Screen-share an implementation of your proposed solution
- Independently follow-along, or create an implementation, in your own IDE.

Your program leader/s will indicate which code sharing tool/s to use as a group, and will help break down or provide specific scaffolding with the main concepts above.

► Note on Expectations

Problem Solving Approach

To build a long-term organized approach to problem solving, we'll start with three main steps. We'll refer to them as **UPI: Understand, Plan, and Implement**.

We'll apply these three steps to most of the problems we'll see in the first half of the course.

We will learn to:

- **Understand** the problem,
- **Plan** a solution step-by-step, and
- **Implement** the solution

▼ Comment on UPI

While **each problem may call for slightly different approaches to these three steps**, the basics of the steps won't change, and it's important to engage them each time. We've built out some starting points to use in our breakout sessions, below!

Please read the following carefully (take 10 minutes as a team, if you like) and follow these basic steps, as a group, through each of the problems in your problem set.

Fun Fact: We sometimes call the main beats of problem solving, or the tasks that teams are being asked to take, our "DoNow's". If you hear a staff member using this phrase, (e.g., "Ok great! Your team is struggling with the Understand step – what might be a DoNow?") you now know what they might mean!

▼ UPI Example

Step	What is it?	Try It!
1. Understand	Here we strive to Understand what the interviewer is asking for. It's common to restate the problem aloud, ask questions, and consider related test cases.	<ul style="list-style-type: none"> • Nominate one person to share their screen so everyone is on the same page, and bring up the problem at hand. (NOTE: Please trade-off and change who is screen sharing, roughly each problem) • Have one person read the problem aloud. • Have a different person restate the problem in their own words. • Have members of the group ask 2-3 questions about the problem, perhaps about the example usage, or expected output • Example: "Will the list always contain only numbers?"
2. Plan	Then we Plan a solution, starting with appropriate visualizations and pseudocode.	<ul style="list-style-type: none"> • Restate - have one person share the general idea about what the function is trying to accomplish. • Next, break down the problem into subproblems as a group. Each member should participate. <ul style="list-style-type: none"> ◦ If you don't know where to start, try to describe how you would solve the problem <i>without a computer</i>. • As a group, translate each subproblem into pseudocode. <ul style="list-style-type: none"> ◦ How do I do what I described in English in Python? ◦ Then, do I need to change my approach to any steps to make it work in code?
3. Implement	Now we Implement our solution, by translating our Plan into Python.	<ul style="list-style-type: none"> • Translate the pseudocode into Python—this is a stage at which you can consider working individually.

Breakout Problems Session 1

- Standard Problem Set Version 1
- Standard Problem Set Version 2
- ▼ Advanced Problem Set Version 1

Problem 1: Counting Treasure

Captain Blackbeard has a treasure map with several clues that point to different locations on an island. Each clue is associated with a specific location and the number of treasures buried there. Given a dictionary `treasure_map` where keys are location names and values are integers representing the number of treasures buried at those locations, write a function `total_treasures()` that returns the total number of treasures buried on the island.

```
def total_treasure(treasure_map):  
    pass
```

Example Usage:

```
treasure_map1 = {  
    "Cove": 3,  
    "Beach": 7,  
    "Forest": 5  
}  
  
treasure_map2 = {  
    "Shipwreck": 10,  
    "Cave": 20,  
    "Lagoon": 15,  
    "Island Peak": 5  
}  
  
print(total_treasures(treasure_map1))  
print(total_treasures(treasure_map2))
```

Example Output:

```
15  
50
```

► 💡 **Hint: Dictionaries**

► 💡 **Hint: Accessing Values in a Dictionary**

► 💡 **Hint: Accessing Keys, Values, and Key-Value Pairs**

Problem 2: Pirate Message Check

Taken captive, Captain Anne Bonny has been smuggled a secret message from her crew. She will know she can trust the message if it contains all of the letters in the alphabet. Given a string `message` containing only lowercase English letters and whitespace, write a function `can_trust_message()` that returns `True` if the message contains every letter of the English alphabet at least once, and `False` otherwise.

```
def can_trust_message(message):  
    pass
```

Example Usage:

```
message1 = "sphinx of black quartz judge my vow"
message2 = "trust me"

print(can_trust_message(message1))
print(can_trust_message(message2))
```

Example Output:

```
True
False
```

► 💡 **Hint: Introduction to sets**

Problem 3: Find All Duplicate Treasure Chests in an Array

Captain Blackbeard has an integer array `chests` of length `n` where all the integers in `chests` are in the range `[1, n]` and each integer appears once or twice. Return an array of all the integers that appear twice, representing the treasure chests that have duplicates.

```
def find_duplicate_chests(chests):
    pass
```

Example Usage:

```
chests1 = [4, 3, 2, 7, 8, 2, 3, 1]
chests2 = [1, 1, 2]
chests3 = [1]

print(find_duplicate_chests(chests1))
print(find_duplicate_chests(chests2))
print(find_duplicate_chests(chests3))
```

Example Output:

```
[2, 3]
[1]
[]
```

► 💡 **Hint: Frequency Maps**

Problem 4: Booby Trap

Captain Feathersword has found another pirate's buried treasure, but they suspect it's booby-trapped. The treasure chest has a secret code written in pirate language, and Captain Feathersword believes the trap can be disarmed if the code can be balanced. A balanced code is one where the

frequency of every letter present in the code is equal. To disable the trap, Captain Feathersword *must* remove exactly one letter from the message. Help Captain Feathersword determine if it's possible to remove one letter to balance the pirate code.

Given a 0-indexed string `code` consisting of only lowercase English letters, write a function `is_balanced()` that returns `True` if it's possible to remove one letter so that the frequency of all remaining letters is equal, and `False` otherwise.

```
def is_balanced(code):  
    pass
```

Example Usage:

```
code1 = "arghh"  
code2 = "haha"  
  
print(is_balanced(code1))  
print(is_balanced(code2))
```

Example Output:

```
True  
Explanation: Select index 4 and delete it: word becomes "argh" and each character has  
  
False  
Explanation: They must delete a character, so either the frequency of "h" is 1 and the
```

Problem 5: Overflowing With Gold

Captain Feathersword and their crew has discovered a list of gold amounts at various hidden locations on an island. Each number on the map corresponds to the amount of gold at a specific location. Captain Feathersword already has plenty of loot, and their ship is nearly full. They want to find two distinct locations on the map such that the sum of the gold amounts at these two locations is exactly equal to the amount of space left on their ship.

Given an array of integers `gold_amounts` representing the amount of gold at each location and an integer `target`, return the *indices* of the two locations whose gold amounts add up to the target.

Assume that each input has exactly one solution, and you may not use the same location twice. You can return the answer in any order.

```
def find_treasure_indices(gold_amounts, target):  
    pass
```

Example Usage:

```

gold_amounts1 = [2, 7, 11, 15]
target1 = 9

gold_amounts2 = [3, 2, 4]
target2 = 6

gold_amounts3 = [3, 3]
target3 = 6

print(find_treasure_indices(gold_amounts1, target1))
print(find_treasure_indices(gold_amounts2, target2))
print(find_treasure_indices(gold_amounts3, target3))

```

Example Output:

```

[0, 1]
[1, 2]
[0, 1]

```

Problem 6: Organize the Pirate Crew

Captain Blackbeard needs to organize his pirate crew into different groups for a treasure hunt. Each pirate has a unique ID from 0 to $n - 1$.

You are given an integer array `group_sizes`, where `group_sizes[i]` is the size of the group that pirate `i` should be in. For example, if `group_sizes[1] = 3`, then pirate 1 must be in a group of size 3.

Return a list of groups such that each pirate `i` is in a group of size `group_sizes[i]`.

Each pirate should appear in exactly one group, and every pirate must be in a group. If there are multiple answers, return any of them. It is guaranteed that there will be at least one valid solution for the given input.

```

def organize_pirate_crew(group_sizes):
    pass

```

Example Usage:

```

group_sizes1 = [3, 3, 3, 3, 3, 1, 3]
group_sizes2 = [2, 1, 3, 3, 3, 2]

print(organize_pirate_crew(group_sizes1))
print(organize_pirate_crew(group_sizes2))

```

Example Output:

```

[[5], [0, 1, 2], [3, 4, 6]]
[[1], [0, 5], [2, 3, 4]]

```

Problem 7: Minimum Number of Steps to Match Treasure Maps

Captain Blackbeard has two treasure maps represented by two strings of the same length `map1` and `map2`. In one step, you can choose any character of `map2` and replace it with another character.

Return the minimum number of steps to make `map2` an anagram of `map1`.

An Anagram of a string is a string that contains the same characters with a different (or the same) ordering.

```
def min_steps_to_match_maps(map1, map2):  
    pass
```

Example Usage:

```
map1_1 = "bab"  
map2_1 = "aba"  
map1_2 = "treasure"  
map2_2 = "huntgold"  
map1_3 = "anagram"  
map2_3 = "mangaar"  
  
print(min_steps_to_match_maps(map1_1, map2_1))  
print(min_steps_to_match_maps(map1_2, map2_2))  
print(min_steps_to_match_maps(map1_3, map2_3))
```

Example Output:

```
1  
6  
0
```

Problem 8: Counting Pirates' Action Minutes

Captain Dread is keeping track of the crew's activities using a log. The logs are represented by a 2D integer array `logs` where each `logs[i] = [pirateID, time]` indicates that the pirate with `pirateID` performed an action at the minute `time`.

Multiple pirates can perform actions simultaneously, and a single pirate can perform multiple actions in the same minute.

The pirate action minutes (PAM) for a given pirate is defined as the number of unique minutes in which the pirate performed an action. A minute can only be counted once, even if multiple actions occur during it.

You are to calculate a 1-indexed array `answer` of size `k` such that, for each `j` ($1 \leq j \leq k$), `answer[j]` is the number of pirates whose PAM equals `j`.

Return the array `answer` as described above.

```
def counting_pirates_action_minutes(logs, k):  
    pass
```

Example Usage:

```
logs1 = [[0, 5], [1, 2], [0, 2], [0, 5], [1, 3]]  
k1 = 5  
logs2 = [[1, 1], [2, 2], [2, 3]]  
k2 = 4  
  
print(counting_pirates_action_minutes(logs1, k1))  
print(counting_pirates_action_minutes(logs2, k2))
```

Example Output:

```
[0, 2, 0, 0, 0]  
[1, 1, 0, 0]
```

[Close Section](#)

▼ Advanced Problem Set Version 2

Problem 1: The Library of Alexandria

In the ancient Library of Alexandria, a temporal rift has scattered several important scrolls across different rooms. You are given a dictionary `library_catalog` that maps room names to the number of scrolls that room should have and a second dictionary `actual_distribution` that maps room names to the number of scrolls found in that room after the temporal rift.

Write a function `analyze_library()` that determines if any room has more or fewer scrolls than it should. The function should return a dictionary where the keys are the room names and the values are the differences in the number of scrolls (actual number of scrolls - expected number of scrolls). You must loop over the dictionaries to compute the differences.

```
def analyze_library(library_catalog, actual_distribution):  
    pass
```

Example Usage:




```
library_catalog = {
    "Room A": 150,
    "Room B": 200,
    "Room C": 250,
    "Room D": 300
}

actual_distribution = {
    "Room A": 150,
    "Room B": 190,
    "Room C": 260,
    "Room D": 300
}

print(analyze_library(library_catalog, actual_distribution))
```

Example Output:

```
{'Room A': 0, 'Room B': -10, 'Room C': 10, 'Room D': 0}
```

- ▶  **Hint: Dictionaries**
- ▶  **Hint: Accessing Values in a Dictionary**
- ▶  **Hint: Accessing Keys, Values, and Key-Value Pairs**

Problem 2: Grecian Artifacts

You've spent your last few trips exploring different periods of Ancient Greece. During your travels, you discover several interesting artifacts. Some artifacts appear in multiple time periods, while others in just one.

You are given two lists of strings `artifacts1` and `artifacts2` representing the artifacts found in two different time periods. Write a function `find_common_artifacts()` that returns a list of artifacts common to both time periods.

```
def find_common_artifacts(artifacts1, artifacts2):
    pass
```

Example Usage:

```
artifacts1 = ["Statue of Zeus", "Golden Vase", "Bronze Shield"]
artifacts2 = ["Golden Vase", "Silver Sword", "Bronze Shield"]

print(find_common_artifacts(artifacts1, artifacts2))
```

Example Output:

```
["Golden Vase", "Bronze Shield"]
```

► 💡 **Hint: Introduction to sets**

Problem 3: Souvenir Declutter

As a time traveler, you've collected a mountain of souvenirs over the course of your travels. You're running out of room to store them all and need to declutter. Given a list of strings `souvenirs` and a integer `threshold`, declutter your souvenirs by writing a function `declutter()` that returns a list of souvenirs strictly below `threshold`.

```
def declutter(souvenirs, threshold):
    pass
```

Example Usage:

```
souvenirs1 = ["coin", "alien egg", "coin", "coin", "map", "map", "statue"]
threshold1 = 3

souvenirs2 = ["postcard", "postcard", "postcard", "sword"]
threshold = 2
```

Example Output:

```
["alien egg", "map", "map", "statue"]
["sword"]
```

► 💡 **Hint: Frequency Maps**

Problem 4: Time Portals

In your time travel adventures, you are given an array of digit strings `portals` and a digit string `destination`. Return the number of pairs of indices `(i, j)` (where `i != j`) such that the concatenation of `portals[i] + portals[j]` equals `destination`.

Note: For index values `i` and `j`, the pairs `(i, j)` and `(j, i)` are considered different - order matters.

```
def num_of_time_portals(portals, destination):  
    pass
```

Example Usage:

```
portals1 = ["777", "7", "77", "77"]  
destination1 = "7777"  
portals2 = ["123", "4", "12", "34"]  
destination2 = "1234"  
portals3 = ["1", "1", "1"]  
destination3 = "11"  
  
print(num_of_time_portals(portals1, destination1))  
print(num_of_time_portals(portals2, destination2))  
print(num_of_time_portals(portals3, destination3))
```

Example Output:

```
4  
2  
6
```

Problem 5: Detect Temporal Anomaly

As a time traveler, you have recorded the occurrences of specific events at different time points. You suspect that some events might be occurring too frequently within short time spans, indicating potential temporal anomalies. Given an array `time_points` where each element represents an event ID at a particular time point, and an integer `k`, determine if there are two distinct time points `i` and `j` such that `time_points[i] == time_points[j]` and the absolute difference between `i` and `j` is at most `k`.

Note: The indices must be unique, but not the values `i` and `j` themselves.

```
def detect_temporal_anomaly(time_points, k):  
    pass
```

Example Usage:

```

time_points1 = [1, 2, 3, 1]
k1 = 3

time_points2 = [1, 0, 1, 1]
k2 = 1

time_points3 = [1, 2, 3, 1, 2, 3]
k3 = 2

print(detect_temporal_anomaly(time_points1, k1))
print(detect_temporal_anomaly(time_points2, k2))
print(detect_temporal_anomaly(time_points3, k3))

```

Example Output:

```

True
True
False

```

Problem 6: Find Travelers with Zero or One Temporal Anomalies

In your time travel adventures, you are given an integer array `anomalies` where `anomalies[i] = [traveleri, anomalyi]` indicates that the traveler `traveleri` caused a temporal anomaly `anomalyi`.

Return a list `answer` of size 2 where:

- `answer[0]` is a list of all travelers that have not caused any anomalies.
- `answer[1]` is a list of all travelers that have caused exactly one anomaly.

The values in the two lists should be returned in increasing order.

Note: You should only consider the travelers that have experienced at least one anomaly.

```

def find_travelers(anomalies):
    pass

```

Example Usage:

```

anomalies1 = [[1,3],[2,3],[3,6],[5,6],[5,7],[4,5],[4,8],[4,9],[10,4],[10,9]]
anomalies2 = [[2,3],[1,3],[5,4],[6,4]]

print(find_travelers(anomalies1))
print(find_travelers(anomalies2))

```

Example Output:

```
[[1, 2, 10], [4, 5, 7, 8]]  
[[1, 2, 5, 6], []]
```

Problem 7: Lingual Frequencies

As a time traveling linguist, you are analyzing texts written in an ancient script. However, some words in the text are illegible and can't be deciphered. Write a function

`find_most_frequent_word()` that accepts a string `text` and a list of illegible words `illegibles` and returns the most frequent word in `text` that is not an illegible word.

```
def find_most_frequent_word(text, illegibles):  
    pass
```

Example Usage:

```
paragraph1 = "a."  
illegibles1 = []  
print(find_most_frequent_word(paragraph1, illegibles1))  
  
paragraph2 = "Bob hit a ball, the hit BALL flew far after it was hit."  
illegibles2 = ["hit"]  
print(find_most_frequent_word(paragraph2, illegibles2))
```

Example Output:

```
a  
  
ball  
Example 2 Explanation:  
"hit" occurs 3 times, but it is an unknown word.  
"ball" occurs twice (and no other word does), so it is the most frequent legible word.  
Note that words in the paragraph are not case sensitive,  
that punctuation is ignored (even if adjacent to words, such as "ball,"),  
and that "hit" isn't the answer even though it occurs more because it is illegible.
```

► 💡 **Hint: Cleaning up the String**

Problem 8: Time Portal Usage

In your time travel adventures, you have been collecting data on the usage of different time portals by various travelers. The data is represented by an array `usage_records`, where

`usage_records[i] = [traveler_name, portal_number, time_used]` indicates that the traveler `traveler_name` used the portal `portal_number` at the time `time_used`.

Return the adventure's "display table". The "display table" is a table whose row entries denote how many times each portal was used at each specific time. The first column is the portal number and the remaining columns correspond to each unique time in chronological order. The first row should be a header whose first column is "Portal", followed by the times in chronological order. Note that the traveler names are not part of the table. Additionally, the rows should be sorted in numerically increasing order.

```
def display_time_portal_usage(usage_records):  
    pass
```

Example Usage:

```
usage_records1 = [ ["David","3","10:00"],  
                   ["Corina","10","10:15"],  
                   ["David","3","10:30"],  
                   ["Carla","5","11:00"],  
                   ["Carla","5","10:00"],  
                   ["Rous","3","10:00"]]   
usage_records2 = [ ["James","12","11:00"],  
                   ["Ratesh","12","11:00"],  
                   ["Amadeus","12","11:00"],  
                   ["Adam","1","09:00"],  
                   ["Brianna","1","09:00"]]   
usage_records3 = [ ["Laura","2","08:00"],  
                   ["Jhon","2","08:15"],  
                   ["Melissa","2","08:30"]]   
  
print(display_time_portal_usage(usage_records1))  
print(display_time_portal_usage(usage_records2))  
print(display_time_portal_usage(usage_records3))
```

Example Output:

```
[['Portal', '10:00', '10:15', '10:30', '11:00'], ['3', '2', '0', '1', '0'], ['5', '1', '0', '0', '  
  '10', '0', '1', '0', '0']]   
[['Portal', '09:00', '11:00'], ['1', '2', '0'], ['12', '0', '3']]   
[['Portal', '08:00', '08:15', '08:30'], ['2', '1', '1', '1']]
```

►  **Hint:** `sorted()` Function

Close Section