PES UNIVERSITY
Python For Computational Problem Solving
Orange Problem Rubrics – 4 marks
Group-based critical thinking task

Criteria	Marks	Description
Correct Logic & Output	1	Function works correctly on edge cases and input variations
Use of Python Features	1	Meaningful use of control structures / list/set/dictionary/tuples/functions/strings/Files
Clarity in viva explanation	2	Each member demonstrates clear understanding during viva; reasoning and logic are well explained.

Bonus (optional): Pythonic code, modularity and extra edge case handling.

Final Plan for 60 Students – 15 Groups of 4

Total Time: 1 hour 30 minutes (90 minutes)

Phase	Duration	Notes
Group Setup + Instructions	15 mins	Reconfirm existing groups (capstone group) and assign questions. Explain rubrics and expectations briefly
Problem Solving and Implementation	40 mins	Teams collaboratively design, code, and test their solution
Viva & Evaluation	35 mins	Each group attends viva; members explain logic, approach, and code understanding. Evaluation based on rubrics.

Orange Problem Questions

Question 1

1. **Secure Contact Book – Vowel-Based Encryption**
   You are building a secure contact book application where only select contacts are stored in an encrypted format.
   You are given **two text files**:

   names.txt – contains names of people (one name per line)
   phones.txt – contains corresponding 9-digit phone numbers (one per line, in the same order as names)

   Example:
   Names.txt
   Arun
   Bala
   Uma
   Kiran
   Anu

   Phones.txt
   123456723
   894562189
   988776519
   113459842
   998877112

   Your task is to write a Python function that:

1. Combines the name and phone number into a dictionary.
2. Includes **only those names that start with a vowel (A, E, I, O, U)** – case-insensitive.
3. Encrypts the phone number by:
   - **Reversing it**, and
   - Replacing **all digits except the last two** with asterisks (*)
     (For example: "123456789" → "*******89" after reversing and masking)

   The function should return a dictionary of names and their encrypted phone numbers.

   Expected Output
   Only include names **starting with a vowel** (A, E, I, O, U) — Arun, Uma, Anu.
   Phone numbers are:
   - Reversed
   - Replaced all but **last 2 digits** with *

```
{
 'Arun': '*******21',
 'Uma': '*******89',
 'Anu': '*******99'
}
```

4. Write a function that sorts the encrypted contact dictionary **alphabetically by name** and saves it into a new file called **secure_contacts.txt** in the following format:

   Anu : *******99
   Arun : *******21
   Uma : *******89

5. Write a function that counts how many contacts start with each vowel (A, E, I, O, U).

   Return a dictionary in the form:
   {'A': 2, 'E': 0, 'I': 0, 'O': 0, 'U': 1}
   Consider both upper- and lower-case letters.

****

Question 2

Student Performance Tracker – Average Score Analysis

You are developing a system to track student performance and identify high-achieving students. You are provided with two text files:

1. students.txt: Contains information about students, with each line formatted as StudentID, StudentName.

   Example:

   - S001,Alice Smith
   - S002,Bob Johnson
   - S003,Charlie Brown
   - S004,Diana Prince
2. scores.txt: Contains individual subject scores for students, with each line formatted as StudentID, Subject,Score.

   Example:

   - S001,Math,92
   - S002,Science,78
   - S001,History,88
   - S003,Math,95
   - S002,History,85
   - S004,Science,90
   - S003,English,80
   - S001,Science,95

Your task is to write a Python function that:

1. Reads the students.txt file and creates a mapping (e.g., a dictionary) from StudentID to StudentName.
2. Reads the scores.txt file and, for each StudentID, calculates their **overall average score** across all subjects they have taken.
3. Identifies "High Achievers" – students whose overall average score is **85 or above**.
4. Returns a dictionary where:
   - Keys are the **names** of the "High Achievers".
   - Values are their calculated **average scores**, rounded to two decimal places.

Assumptions & Constraints:

- Assume all StudentIDs are unique.
- Scores are integers.

- A student might appear in students.txt but not scores.txt (meaning they have no scores yet – their average should be considered 0 in this case).
- Use only control structures (e.g., if, for, while), lists, tuples, sets, dictionaries, strings, and file operations.

Expected Output Example (based on the provided sample data):

```
{ 'Alice Smith': 91.67,                    # (92+88+95)/3
'Charlie Brown': 87.50,                 # (95+80)/2
'Diana Prince': 90.00 }
```

****

Question 3

Verifying Packet Routing Chain

In a communication network, each data packet must travel through a series of routers. Every router passes the packet to the **next** based on a predefined chain.

The routing path is given as a list of tuples. Each tuple represents a connection where:

- The **first value** is the current router ID
- The **second value** is the **next router ID**

You are given a list of such tuples.

Task

Write a Python function that:

1. Accepts a list of (from_router, to_router) tuples.
2. Verifies if the routing path is **continuous**, meaning:
    - For every two **consecutive tuples**, the to_router of the first tuple must match the from_router of the next.
3. Returns True if the chain is valid, otherwise returns False.

Expected Output:

Input: [(2, 3), (3, 4), (4, 5), (5, 1)]

Output: True

****

Input: [(1, 2), (2, 4), (3, 5)]

Output: False

Hint: Use indexing or zip() for comparing consecutive tuples.

4. **Loop Detection – Is the Chain Circular?**
   Write a function that checks whether the given routing path forms a **closed loop**.

- A chain is circular if the **to_router** of the last tuple equals the **from_router** of the first tuple.
- Return "Circular" if it forms a loop, otherwise "Not Circular".

Example:

Input: [(2, 3), (3, 4), (4, 2)]

Output: "Circular"

5. **Detect Missing or Duplicate Routers (Network Integrity Check)**
   Extend your function to verify if:

Any router ID is **missing** (i.e., breaks the continuity chain).
Any router ID **appears multiple times** as a starting router (possible faulty loop).
The function should return a message indicating:
"Valid Chain"
"Duplicate Routers Found"
"Missing Router in Sequence"
Input: [(1, 2), (2, 3), (3, 2), (2, 4)]
Output: "Duplicate Routers Found"

Input: [(1, 2), (4, 5)]
Output: "Missing Router in Sequence"
****

Question 4

Access Tracker – Analysing File Usage Logs

In a multi-user system, access logs are maintained to record which user accessed which file.

Each line in the log file follows this format:

username:accessed_file

For example:

alice:data.csv

bob:data.csv

charlie:report.docx

alice:report.docx

However, due to network noise or improper logging, some lines may be **blank** or **malformed** (i.e., missing the : separator).

Task

Write a Python function that:

1. Reads a log file line by line.
2. Ignores:
    - Blank lines
    - Malformed entries (lines without the : separator)
3. Builds and returns a **dictionary** where:
    - **Key** = File name
    - **Value** = A **set** of unique usernames who accessed that file

    Example: input.txt

    alice:data.csv

    bob:data.csv

    charlie:report.docx

    alice:report.docx

    badentrywithoutcolon

    david:

    :config.yaml

Expected Output:

```
{
 'data.csv': {'alice', 'bob'},
 'report.docx': {'alice', 'charlie'}
}
```

4. **Access Frequency Summary (File-Wise Count)**
   Extend your program to create another dictionary where:
   Key = File name
   Value = Count of *unique users* who accessed it
   Example Output:
   {'data.csv': 2, 'report.docx': 2}

5. **Reverse Mapping – Users and Their Accessed Files**
   Create a **reverse dictionary** where:
   Key = Username
   Value = A set of all files accessed by that user
   Example Output:

```
{
 'alice': {'data.csv', 'report.docx'},
 'bob': {'data.csv'},
 'charlie': {'report.docx'}
}
```

****

QUESTION 5

Pattern Monitoring in Security Logs

A cybersecurity tool monitors system logs to detect suspicious or previously unseen character patterns in commands entered by users.

Your task is to implement a function that analyzes a given command log string and **extracts all distinct 3-character sequences** that:

1. Contain **all unique characters** (i.e., no character repeats within the 3-character substring).
2. Have **not appeared before**, regardless of case (i.e., 'AbC' and 'abc' are considered the same pattern and should only be included once).

This helps identify potentially new or obfuscated access attempts that differ slightly from previously seen ones.

Input

- A single string log representing the command sequence.

Output

- A list of 3-character substrings that satisfy the above conditions, **in the order they appear**.

Example:

Input: "AbCxyZabc"

Output: ['AbC', 'bCx', 'Cxy', 'xyZ', 'yZa', 'Zab']

3. **Frequency Mapping of Valid Patterns**
   After extracting all valid 3-character patterns (as per Question 1),
   create a **dictionary** showing the **frequency** of each *lowercased* pattern's occurrence in the input string.
   This will help identify which sequences appear most frequently, even if repeated in different cases.
   Example:
   Input: "AbCxyZabcABc"
   Output:
   {'abc': 2, 'bcx': 1, 'cxy': 1, 'xyz': 1, 'yza': 1, 'zab': 1}

4. **Suspicious Pattern Detector (Digit or Symbol Involvement)**
   Enhance your program to **flag suspicious 3-character patterns** that include:
   Any **digit (0–9)** or
   Any **non-alphabetic character** (like #, $, @, _, etc.)
   Return a **set** of such suspicious patterns.

Example:

Input: "aB3c$Df@xy"

Output: {'B3c', 'c$D', 'f@x'}

****

QUESTION 6

Talent Spotter – Extracting Productive Passions

In a school's talent program, each student's profile is stored as a nested dictionary where:

- The key is the **student's name**.
- Each value is another dictionary containing:
    - "marks": A list of scores from two tests.
    - "hobbies": A tuple of that student's hobbies.

The school wants to **identify hobbies pursued by high-performing students** (those with an average score above 75), so that they can invite them to participate in clubs and competitions.

Task

Write a Python function that:

1. Accepts the nested dictionary of students.
2. Calculates the **average score** for each student.
3. For those with an average **strictly greater than 75**, extracts their hobbies.
4. Returns a **set** of all such hobbies (to ensure uniqueness).

A bonus part (return also the names of qualifying students)?

Example:

```
students = {
  'arun': {'marks': [78, 82], 'hobbies': ('chess', 'reading')},
  'bala': {'marks': [90, 95], 'hobbies': ('reading',)},
  'chitra': {'marks': [70, 65], 'hobbies': ('painting', 'reading')}
}
```

Expected Output:

{'chess', 'reading'}

5. **Return Also the Names of High Performers**
   Modify the function to **also return the names** of students who qualified (average > 75).
   Expected Output:
   ({'chess', 'reading'}, ['arun', 'bala'])

6. **Hobby Popularity Ranking Among High Performers**
   Extend your program to find **which hobby is most common** among all high-performing students.
   Return a **dictionary** showing the **count of each hobby** among them.
   Expected Output:
   {'reading': 2, 'chess': 1}

****

QUESTION 7

Resume Filter – Extracting Unique Skills from Candidates

A recruitment system receives hundreds of resumes, and each resume is processed to extract the candidate's skill set.

The extracted data is saved into a file called resumes.txt, where each line represents the **skills of one candidate**, written as **comma-separated values**. Skills may vary in:

- **Capitalization** (e.g., "Python" vs "python")
- **Extra whitespace** (e.g., " C++ ")

Your Task

Write a Python function that:

1. Reads the resumes.txt file.
2. For each line, splits the skills using commas.
3. Cleans the skills by:
    - Stripping leading/trailing whitespace
    - Converting to lowercase
4. Returns a **set** of all unique skills across all candidates.

Sample File:

resumes.txt

Python, C++, HTML

java, Python, SQL

c++, SQL , Data Analysis

JavaScript, html, CSS

Expected Output:

{'python', 'c++', 'html', 'java', 'sql', 'data analysis', 'javascript', 'css'}

Hint:

Use set() to store unique values.

Be careful with cleaning: ' SQL ' → 'sql'; 'HTML' and 'html' → same.

5. **Skill Popularity – Count Skill Occurrences**

Write a function that counts how many candidates mentioned each skill.
Return a **dictionary** where:

Key = skill name (in lowercase)

Value = number of candidates who listed that skill

```
{
 'python': 2,
 'c++': 2,
 'html': 2,
 'java': 1,
 'sql': 2,
 'data analysis': 1,
 'javascript': 1,
 'css': 1
}
```

****

QUESTION 8

Library Genre Reverse Mapper

A university library tracks student reading preferences in the following format:

- Each student is represented as a key in a dictionary.
- Each value is a list of **genres** that student enjoys reading.

To organize reading clubs based on genre interest, the library wants to reverse this mapping — from **students → genres** to **genres → set of students**.

Your Task

Write a Python function that:

1. Accepts a dictionary where:
    - Keys are student names (strings)
    - Values are lists of genres (strings)
2. Returns a new dictionary where:
    - **Each genre is a key**
    - **Each value is a set of students** who read that genre

Example Input:

```
readers = {
  'Arun': ['Fantasy', 'Sci-fi'],
  'Bala': ['Mystery'],
  'Chitra': ['Fantasy', 'Mystery']
}
```

Expected Output:

```
{
  'Fantasy': {'Arun', 'Chitra'},
  'Sci-fi': {'Arun'},
  'Mystery': {'Bala', 'Chitra'}
}
```

3. Extend the reverse_mapping function to also compute and return a second dictionary alongside the original reversed mapping. This second dictionary should map each genre to the average number of genres per student who enjoys that genre (rounded to 2 decimal places). Use the original readers dictionary to calculate the average for each group of students per genre.

Example input:

```
readers = {
   'Arun': ['Fantasy', 'Sci-fi'],           # Arun likes 2 genres
   'Bala': ['Mystery'],                     # Bala likes 1 genre
   'Chitra': ['Fantasy', 'Mystery']         # Chitra likes 2 genres
}

Expected Output: (A tuple of two dictionaries)
(
   {  # First dict: reversed mapping (same as Task 1)
      'Fantasy': {'Arun', 'Chitra'},
      'Sci-fi': {'Arun'},
      'Mystery': {'Bala', 'Chitra'}
   },
   {  # Second dict: genre to average genres per student
      'Fantasy': 2.00,   # (2 + 2) / 2 = 2.00
      'Sci-fi': 2.00,    # 2 / 1 = 2.00
      'Mystery': 1.50    # (1 + 2) / 2 = 1.50
   }
)
```

****

QUESTION 9

Time Slot Overlap Checker – Classroom Conflict Detector

In your university, multiple classes are scheduled throughout the day. Each class is assigned a **time slot** represented as a tuple (start_time, end_time), where the time is in 24-hour format (e.g., 9.0 for 9:00 AM, 13.5 for 1:30 PM, etc.).

The academic scheduler wants to ensure that **no two classes** overlap in time.

Your Task

Write a Python function that:

1. Accepts a list of time slot tuples.
2. Checks for **any overlap** between slots.
    - A time slot **(a, b)** and **(c, d)** overlap **if a < d and c < b**
3. Returns True **if there is any overlap**, otherwise returns False.

Example 1 (Overlapping Slots)

slots = [(9, 10), (10, 11), (10, 10.5), (11, 12)]

Output: True

#(10, 11) and (10, 10.5) overlap.

Example 2 (No Overlaps)

slots = [(8, 9), (9, 10), (10, 11), (11, 12)]

False

#No slots overlap

Hint:

Sort the list based on start time.

Compare each time slot with the next one to check if they overlap.

4. To optimize resource allocation, write a function min_classrooms_needed that takes a list of time slot tuples and returns the minimum number of classrooms required to schedule all classes without conflicts. This is determined by finding the maximum number of classes overlapping at any single point in time.

Approach Hint:

- Create a list of all start and end events, sorted by time.
- Use a sweep line algorithm: increment a counter for starts, decrement for ends, and track the maximum counter value.

Example 1 (Overlapping Slots):

slots = [(9, 10), (10, 11), (10, 10.5), (11, 12)]

Expected Output: 2 # At time 10, two classes overlap: (10, 11) and (10, 10.5).

Example 2 (No Overlaps):

slots = [(8, 9), (9, 10), (10, 11), (11, 12)]

Expected Output: 1 # Classes are sequential; only one classroom needed.

Example 3 (High Overlap):

slots = [(1, 4), (2, 5), (3, 6), (0, 7)]

Expected Output: 4 # All four overlap around time 3-4.

****

QUESTION 10

Tag Frequency Extractor – Blog Meta Analyzer

A blogging platform stores tags for each blog post in a dictionary, where:

- Keys are blog post titles (strings)
- Values are **lists of tags** (strings) used in that post

Your task is to **analyse how often each tag is used** across all posts.

Your Task

Write a Python function that:

1. Accepts a dictionary where keys are blog post titles and values are lists of tags.
2. Returns a **dictionary** where:
    - Keys are unique tags
    - Values are the **number of times** each tag was used across all posts.

    Example Input:

    posts = {

    'Post1': ['python', 'ai', 'ml'],

    'Post2': ['ai', 'datascience', 'python'],

    'Post3': ['ml', 'ai', 'cloud']

    }

    Expected Output:

    {'python': 2, 'ai': 3, 'ml': 2, 'datascience': 1, 'cloud': 1}

    Hint:

    Use a dictionary to store tag counts.

    Loop through each post and its tags.
3. Extend the analysis by writing a function tag_stats that returns a tuple of two dictionaries:
    - The tag frequency dictionary from Task 1.
    - A new dictionary where keys are tags and values are the percentage of posts that use that tag (rounded to 2 decimal places). The percentage is calculated as (number of posts with this tag / total posts) * 100.

    Example Input:
    posts = {
    'Post1': ['python', 'ai', 'ml'],

```
    'Post2': ['ai', 'datascience', 'python'],
    'Post3': ['ml', 'ai', 'cloud']
}
```

Expected Output:

```
(
  {  # First dict: frequencies (same as Task 1)
     'python': 2, 'ai': 3, 'ml': 2, 'datascience': 1, 'cloud': 1
  },
  {  # Second dict: percentages
     'python': 66.67, 'ai': 100.00, 'ml': 66.67,
     'datascience': 33.33, 'cloud': 33.33
  }
)
```

****