#### **Problem Statements**

#### There is a csv file called employee.csv.The file contains employee details with columns: Name, Age, Department, Salary, and Years\_of\_Experience.

1. Write a function department\_with\_highest\_experience() that calculates the average years of experience for each department and returns the department with the highest average experience. (Example Output: 'Engineering', 6.5 )

import csv

def department\_with\_highest\_experience(file\_path):

    # Dictionary to store experience totals and counts for each department

    department\_experience = {}

    # Read the CSV file

    with open(file\_path, mode='r') as file:

        reader = csv.DictReader(file)

        for row in reader:

            department = row['Department']

            years\_of\_experience = float(row['Years\_of\_Experience'])

            if department not in department\_experience:

                department\_experience[department] = [0, 0]  # [total\_experience, count]

            # Update total experience and count for the department

            department\_experience[department][0] += years\_of\_experience

            department\_experience[department][1] += 1

    # Calculate the department with the highest average experience

    highest\_avg\_experience = 0

    top\_department = None

    for department, (total\_experience, count) in department\_experience.items():

        avg\_experience = total\_experience / count

        if avg\_experience > highest\_avg\_experience:

            highest\_avg\_experience = avg\_experience

            top\_department = department

    return top\_department, highest\_avg\_experience

result = department\_with\_highest\_experience("employees.csv")

print(result)

Output : ('Engineering', 11.0)

1. Write a function salary\_by\_age\_group() that groups employees into age ranges (20-30, 31-40, etc.) and calculates the total salary for each age group.(Example Output: {'20-30': 148000, '31-40': 125000, '41-50': 0})

import csv

def salary\_by\_age\_group(file\_path):

    # Define age groups

    age\_groups = {

        '20-30': 0,

        '31-40': 0,

        '41-50': 0,

        '51-60': 0,

        '61+': 0

    }

    # Read the CSV file

    with open(file\_path, mode='r') as file:

        reader = csv.DictReader(file)

        for row in reader:

            age = int(row['Age'])

            salary = float(row['Salary'])

            # Determine the age group

            if 20 <= age <= 30:

                age\_groups['20-30'] += salary

            elif 31 <= age <= 40:

                age\_groups['31-40'] += salary

            elif 41 <= age <= 50:

                age\_groups['41-50'] += salary

            elif 51 <= age <= 60:

                age\_groups['51-60'] += salary

            elif age > 60:

                age\_groups['61+'] += salary

    return age\_groups

file\_path= "employees.csv"

result2= salary\_by\_age\_group(file\_path)

print(result2)

Output:

{'20-30': 208920.0, '31-40': 267960.0, '41-50': 181500.0, '51-60': 0, '61+': 0}

1. Write a function employees\_above\_average\_salary() that returns a list of employee names who are paid above the average salary for their respective department.

import csv

def employees\_above\_average\_salary(file\_path):

    # Dictionaries to store total salary and count of employees per department

    department\_salary = {}

    department\_count = {}

    # First pass: Calculate total salary and count of employees for each department

    with open(file\_path, mode='r') as file:

        reader = csv.DictReader(file)

        for row in reader:

            department = row['Department']

            salary = float(row['Salary'])

            if department not in department\_salary:

                department\_salary[department] = 0

                department\_count[department] = 0

            department\_salary[department] += salary

            department\_count[department] += 1

    # Calculate average salary for each department

    department\_avg\_salary = {

        department: department\_salary[department] / department\_count[department]

        for department in department\_salary

    }

    # Second pass: Find employees with salary above their department's average

    employees = []

    with open(file\_path, mode='r') as file:

        reader = csv.DictReader(file)

        for row in reader:

            department = row['Department']

            salary = float(row['Salary'])

            name = row['Name']

            if salary > department\_avg\_salary[department]:

                employees.append(name)

    return employees

result3 = employees\_above\_average\_salary(file\_path)

print(result3)

Output:

['Eva', 'Frank', 'Grace', 'Ian', 'Jack']

1. Write a function give\_raises that increases the salary of all employees with more than 5 years of experience by 10%. The updated data should be saved back into the employees.csv file.

import csv

def give\_raises(file\_path):

    updated\_rows = []

    # Read the CSV file and update salaries

    with open(file\_path, mode='r') as file:

        reader = csv.DictReader(file)

        fieldnames = reader.fieldnames  # Get the column headers

        for row in reader:

            years\_of\_experience = float(row['Years\_of\_Experience'])

            # Check if the employee has more than 5 years of experience

            if years\_of\_experience > 5:

                row['Salary'] = str(float(row['Salary']) \* 1.10)  # Increase salary by 10%

            updated\_rows.append(row)

    # Write the updated data back to the CSV file

    with open(file\_path, mode='w', newline='') as file:

        writer = csv.DictWriter(file, fieldnames=fieldnames)

        writer.writeheader()  # Write the headers

        writer.writerows(updated\_rows)  # Write the updated rows

    print("Salaries updated successfully.")

result4= give\_raises(file\_path)

print(result4)

Output:

Salaries updated successfully.

None

2) A startup is conducting a feedback program of a product, stored in a text file named feedback.txt, where each line represents one piece of feedback. The company wants to analyze the feedback to understand customer sentiment around certain aspects, such as "service," "price," and "quality." Each feedback line could express either positive or negative sentiment. Positive feedback often contains keywords such as "excellent," "great," "satisfied," and "happy," while negative feedback includes words like "poor," "bad," "disappointed," and "unsatisfied." Write a Python program to read feedback.txt, analyze the following.

* 1. Count keyword occurrences: Count how often each specified keyword appears in the feedback.
  2. Separate good and bad feedback: Save positive feedback lines in a file named good\_feedback.txt and negative feedback lines in bad\_feedback.txt.
  3. Summarize counts: Save the keyword counts and the overall counts of good and bad feedback lines in keyword\_counts.csv, without overwriting any existing data in this file.

Avoid overwriting- Ensure that no previous data is lost or overwritten in good\_feedback.txt and bad\_feedback.txt; if these files exist, append new data to them.

import csv

import os

# Keywords for analysis

positive\_keywords = ["excellent", "great", "satisfied", "happy"]

negative\_keywords = ["poor", "bad", "disappointed", "unsatisfied"]

# File names

input\_file = "feedback.txt"

good\_feedback\_file = "good\_feedback.txt"

bad\_feedback\_file = "bad\_feedback.txt"

summary\_file = "keyword\_counts.csv"

# Initialize counters

keyword\_counts = {word: 0 for word in positive\_keywords + negative\_keywords}

good\_feedback\_count = 0

bad\_feedback\_count = 0

# Read feedback and analyze

with open(input\_file, "r") as file:

    feedback\_lines = file.readlines()

good\_feedback = []

bad\_feedback = []

for line in feedback\_lines:

    line\_lower = line.lower()

    is\_positive = any(word in line\_lower for word in positive\_keywords)

    is\_negative = any(word in line\_lower for word in negative\_keywords)

    # Count keywords

    for word in positive\_keywords + negative\_keywords:

        keyword\_counts[word] += line\_lower.count(word)

    # Categorize feedback

    if is\_positive and not is\_negative:

        good\_feedback.append(line)

        good\_feedback\_count += 1

    elif is\_negative and not is\_positive:

        bad\_feedback.append(line)

        bad\_feedback\_count += 1

# Append good and bad feedback to respective files

with open(good\_feedback\_file, "a") as file:

    file.writelines(good\_feedback)

with open(bad\_feedback\_file, "a") as file:

    file.writelines(bad\_feedback)

# Append summary to CSV

file\_exists = os.path.exists(summary\_file)

with open(summary\_file, "a", newline="") as file:

    writer = csv.writer(file)

    if not file\_exists:

        writer.writerow(["Keyword", "Count"])  # Write header if file doesn't exist

    for keyword, count in keyword\_counts.items():

        writer.writerow([keyword, count])

    # Write overall counts

    writer.writerow(["Good Feedback Lines", good\_feedback\_count])

    writer.writerow(["Bad Feedback Lines", bad\_feedback\_count])

print("Analysis complete. Files updated successfully.")

3) A restaurant chain wants to calculate the final cost of a meal, taking into account various discounts, service charges, and taxes. Write a program that has 3 functions that call each other to get the final price. Each step involves multiple conditions based on default parameters:

* 1. apply\_discount: This function reduces the base price by a default discount rate (e.g., 10%). Additionally, if the customer is a member, it applies an additional loyalty discount (default 5%). If it’s a special promotion day, it applies a further discount (default 3%).
  2. add\_service\_charge: Calls apply\_discount, then adds a service charge based on the type of meal (dine-in or takeout) with default rates. It adds a flat surcharge for dine-in orders, whereas for takeout, it calculates a percentage of the discounted price.
  3. calculate\_final\_price: Calls add\_service\_charge and applies a tax. For high-value orders (default threshold $100), an additional luxury tax (default 2%) is applied.

In the main program, call calculate\_final\_price with parameters to test different scenarios. Make sure to use default parameters wherever required.

# Function to apply discounts

def apply\_discount(base\_price, discount\_rate=0.10, loyalty\_discount=0.05, promo\_discount=0.03, is\_member=False, is\_promo\_day=False):

    price\_after\_discount = base\_price \* (1 - discount\_rate)

    if is\_member:

        price\_after\_discount \*= (1 - loyalty\_discount)

    if is\_promo\_day:

        price\_after\_discount \*= (1 - promo\_discount)

    return price\_after\_discount

# Function to add service charges

def add\_service\_charge(discounted\_price, service\_type="dine-in", dine\_in\_charge=5, takeout\_rate=0.02):

    if service\_type == "dine-in":

        total\_price = discounted\_price + dine\_in\_charge

    elif service\_type == "takeout":

        total\_price = discounted\_price \* (1 + takeout\_rate)

    else:

        raise ValueError("Invalid service type. Use 'dine-in' or 'takeout'.")

    return total\_price

# Function to calculate the final price

def calculate\_final\_price(base\_price, service\_type="dine-in", tax\_rate=0.08, luxury\_tax\_rate=0.02, luxury\_threshold=100, is\_member=False, is\_promo\_day=False):

    # Step 1: Apply discounts

    discounted\_price = apply\_discount(base\_price, is\_member=is\_member, is\_promo\_day=is\_promo\_day)

    # Step 2: Add service charge

    price\_with\_service\_charge = add\_service\_charge(discounted\_price, service\_type=service\_type)

    # Step 3: Apply tax

    final\_price = price\_with\_service\_charge \* (1 + tax\_rate)

    if base\_price > luxury\_threshold:

        final\_price \*= (1 + luxury\_tax\_rate)

    return round(final\_price, 2)

# Main program to test different scenarios

print("Scenario 1: Regular customer, dine-in")

print(calculate\_final\_price(120, service\_type="dine-in"))

print("\nScenario 2: Member, takeout, promo day")

print(calculate\_final\_price(80, service\_type="takeout", is\_member=True, is\_promo\_day=True))

print("\nScenario 3: Non-member, dine-in, no promo")

print(calculate\_final\_price(150, service\_type="dine-in", is\_member=False, is\_promo\_day=False))

print("\nScenario 4: High-value order, member, takeout")

print(calculate\_final\_price(200, service\_type="takeout", is\_member=True, is\_promo\_day=False))

Output :

Scenario 1: Regular customer, dine-in

124.48

Scenario 2: Member, takeout, promo day

73.09

Scenario 3: Non-member, dine-in, no promo

154.22

Scenario 4: High-value order, member, takeout

192.14

1. You have a folder with 10 students output files and a main key output file. Each student output file and the key output file contain 10 lines of text. Your task is to compare the content of each student output file with the key file line by line and assign marks based on the correctness of each line. Each line is worth 2 marks, so the maximum score for a student is 20 marks. (files required are given.)

import os

def grade\_student\_files(key\_file, student\_folder, output\_file):

    # Open the key file and read its content

    with open(key\_file, 'r') as kf:

        key\_lines = kf.readlines()

    # Prepare to write the results

    results = []

    # Iterate through each student file in the folder

    for student\_file in os.listdir(student\_folder):

        if student\_file.endswith(".txt"):  # Ensure we process only text files

            student\_path = os.path.join(student\_folder, student\_file)

            with open(student\_path, 'r') as sf:

                student\_lines = sf.readlines()

            # Calculate the score for the student

            score = 0

            for i, (key\_line, student\_line) in enumerate(zip(key\_lines, student\_lines)):

                if key\_line.strip() == student\_line.strip():  # Compare line by line

                    score += 2

            # Append the result for the student

            student\_name = os.path.splitext(student\_file)[0]

            results.append((student\_name, score))

    # Write the results to the output file

    with open(output\_file, 'w') as of:

        of.write("Student,Score\n")

        for student\_name, score in results:

            of.write(f"{student\_name},{score}\n")

    print(f"Grading complete. Results saved in {output\_file}.")

# Specify the file paths and folder

key\_output\_file = "key\_output.txt"

student\_output\_folder = "student\_outputs"

results\_output\_file = "grading\_results.csv"

# Grade the student files

grade\_student\_files(key\_output\_file, student\_output\_folder, results\_output\_file)