TASK1:

# Few-shot examples:

# Example 1: convert\_currency(100, 'USD', 'EUR', {'USD': 1, 'EUR': 0.92}) -> 92.0

# Example 2: convert\_currency(50, 'EUR', 'USD', {'USD': 1, 'EUR': 0.92}) -> 54.35

# Example 3: convert\_currency(200, 'USD', 'JPY', {'USD': 1, 'JPY': 110}) -> 22000.0

def convert\_currency(amount, from\_currency, to\_currency, rates):

    """

    Converts an amount from one currency to another using exchange rates stored in a dictionary.

    rates: dictionary with currency codes as keys and their rates relative to a base currency.

    """

    if from\_currency not in rates or to\_currency not in rates:

        return "Currency not found in rates dictionary."

    # Convert amount to base currency, then to target currency

    base\_amount = amount / rates[from\_currency]

    converted = base\_amount \* rates[to\_currency]

    return round(converted, 2)

# Ask user for input

try:

    amount = float(input("Enter the amount to convert: "))

    from\_currency = input("Enter the currency to convert from (e.g., USD): ").upper()

    to\_currency = input("Enter the currency to convert to (e.g., EUR): ").upper()

    # Example rates dictionary

    rates = {'USD': 1, 'EUR': 0.92, 'JPY': 110}

    result = convert\_currency(amount, from\_currency, to\_currency, rates)

    print(f"{amount} {from\_currency} is {result} {to\_currency}")

except Exception as e:

    print("Error:", e)

task 2:

import re

def extract\_emails(text):

  """

  Extracts all email addresses from a block of text using regular expressions.

  """

  pattern = r'[\w\.-]+@[\w\.-]+\.[a-zA-Z]+'

  return re.findall(pattern, text)

# Ask user for input

user\_text = input("Enter text to extract email addresses from: ")

emails = extract\_emails(user\_text)

print("Extracted emails:", emails)

task 3:

*import* json

*from* typing *import* List, Dict, Optional

*import* random

class MovieRecommender:

    def \_\_init\_\_(*self*):

*# Sample movie database with genres*

*self*.movies = [

            {"title": "The Shawshank Redemption", "genres": ["Drama"], "rating": 9.3, "year": 1994},

            {"title": "The Godfather", "genres": ["Crime", "Drama"], "rating": 9.2, "year": 1972},

            {"title": "Pulp Fiction", "genres": ["Crime", "Drama"], "rating": 8.9, "year": 1994},

            {"title": "The Dark Knight", "genres": ["Action", "Crime", "Drama"], "rating": 9.0, "year": 2008},

            {"title": "Fight Club", "genres": ["Drama"], "rating": 8.8, "year": 1999},

            {"title": "Inception", "genres": ["Action", "Adventure", "Sci-Fi"], "rating": 8.8, "year": 2010},

            {"title": "The Matrix", "genres": ["Action", "Sci-Fi"], "rating": 8.7, "year": 1999},

            {"title": "Interstellar", "genres": ["Adventure", "Drama", "Sci-Fi"], "rating": 8.6, "year": 2014},

            {"title": "The Lion King", "genres": ["Animation", "Adventure", "Drama"], "rating": 8.5, "year": 1994},

            {"title": "Titanic", "genres": ["Drama", "Romance"], "rating": 7.9, "year": 1997},

            {"title": "La La Land", "genres": ["Comedy", "Drama", "Musical"], "rating": 8.0, "year": 2016},

            {"title": "Get Out", "genres": ["Horror", "Mystery", "Thriller"], "rating": 7.7, "year": 2017},

            {"title": "The Grand Budapest Hotel", "genres": ["Comedy", "Drama"], "rating": 8.1, "year": 2014},

            {"title": "Mad Max: Fury Road", "genres": ["Action", "Adventure", "Sci-Fi"], "rating": 8.1, "year": 2015},

            {"title": "The Martian", "genres": ["Adventure", "Drama", "Sci-Fi"], "rating": 8.0, "year": 2015}

        ]

*# Few-shot examples for genre matching*

*self*.few\_shot\_examples = {

            "Drama": [

                "User likes emotional storytelling and character development",

                "Recommend: The Shawshank Redemption, The Godfather, Fight Club"

            ],

            "Action": [

                "User enjoys fast-paced sequences and thrilling scenes",

                "Recommend: The Dark Knight, The Matrix, Mad Max: Fury Road"

            ],

            "Sci-Fi": [

                "User is interested in futuristic concepts and technology",

                "Recommend: Inception, The Matrix, Interstellar"

            ],

            "Comedy": [

                "User prefers light-hearted and humorous content",

                "Recommend: La La Land, The Grand Budapest Hotel"

            ],

            "Horror": [

                "User enjoys suspenseful and scary content",

                "Recommend: Get Out"

            ]

        }

    def get\_genre\_recommendations(*self*, *preferred\_genre*: str, *num\_recommendations*: int = 3) -> List[Dict]:

        """

        Recommend movies based on user's preferred genre using few-shot prompting approach.

        Args:

            preferred\_genre (str): The genre the user prefers

            num\_recommendations (int): Number of movies to recommend

        Returns:

            List[Dict]: List of recommended movies with their details

        """

*# Convert to title case for better matching*

*preferred\_genre* = *preferred\_genre*.title()

*# Get few-shot context for the genre*

        genre\_context = *self*.few\_shot\_examples.get(*preferred\_genre*, [])

*# Find movies that match the preferred genre*

        matching\_movies = []

*for* movie *in* *self*.movies:

*if* *preferred\_genre* in [genre.title() *for* genre *in* movie["genres"]]:

                matching\_movies.append(movie)

*# Sort by rating (highest first) and then by year (newest first)*

        matching\_movies.sort(*key*=lambda *x*: (*x*["rating"], *x*["year"]), *reverse*=True)

*# Apply few-shot learning logic*

*if* genre\_context:

*# Use few-shot examples to enhance recommendations*

            enhanced\_movies = *self*.\_apply\_few\_shot\_learning(matching\_movies, *preferred\_genre*, genre\_context)

            recommendations = enhanced\_movies[:*num\_recommendations*]

*else*:

*# Fallback to standard recommendations*

            recommendations = matching\_movies[:*num\_recommendations*]

*return* recommendations

    def \_apply\_few\_shot\_learning(*self*, *movies*: List[Dict], *genre*: str, *context*: List[str]) -> List[Dict]:

        """

        Apply few-shot learning to enhance movie recommendations.

        Args:

            movies (List[Dict]): List of movies matching the genre

            genre (str): The preferred genre

            context (List[str]): Few-shot examples for the genre

        Returns:

            List[Dict]: Enhanced list of movies

        """

        enhanced\_movies = *movies*.copy()

*# Apply genre-specific logic based on few-shot examples*

*if* *genre* == "Drama":

*# Drama movies: prioritize emotional depth and character development*

            enhanced\_movies.sort(*key*=lambda *x*: (*x*["rating"], len(*x*["genres"]) == 1, *x*["year"]), *reverse*=True)

*elif* *genre* == "Action":

*# Action movies: prioritize high energy and visual appeal*

            enhanced\_movies.sort(*key*=lambda *x*: (*x*["rating"], "Adventure" in *x*["genres"], *x*["year"]), *reverse*=True)

*elif* *genre* == "Sci-Fi":

*# Sci-Fi movies: prioritize innovative concepts and modern effects*

            enhanced\_movies.sort(*key*=lambda *x*: (*x*["rating"], *x*["year"], "Action" in *x*["genres"]), *reverse*=True)

*elif* *genre* == "Comedy":

*# Comedy movies: prioritize humor and entertainment value*

            enhanced\_movies.sort(*key*=lambda *x*: (*x*["rating"], "Musical" in *x*["genres"], *x*["year"]), *reverse*=True)

*elif* *genre* == "Horror":

*# Horror movies: prioritize suspense and psychological elements*

            enhanced\_movies.sort(*key*=lambda *x*: (*x*["rating"], "Mystery" in *x*["genres"], *x*["year"]), *reverse*=True)

*return* enhanced\_movies

    def get\_multi\_genre\_recommendations(*self*, *preferred\_genres*: List[str], *num\_recommendations*: int = 5) -> List[Dict]:

        """

        Recommend movies based on multiple preferred genres.

        Args:

            preferred\_genres (List[str]): List of genres the user prefers

            num\_recommendations (int): Number of movies to recommend

        Returns:

            List[Dict]: List of recommended movies

        """

*# Find movies that match any of the preferred genres*

        matching\_movies = []

*for* movie *in* *self*.movies:

            movie\_genres = [genre.title() *for* genre *in* movie["genres"]]

*if* any(genre.title() *in* movie\_genres *for* genre *in* *preferred\_genres*):

*# Calculate genre match score*

                match\_score = sum(1 *for* genre *in* *preferred\_genres* *if* genre.title() *in* movie\_genres)

                movie["genre\_match\_score"] = match\_score

                matching\_movies.append(movie)

*# Sort by genre match score, then by rating, then by year*

        matching\_movies.sort(*key*=lambda *x*: (*x*["genre\_match\_score"], *x*["rating"], *x*["year"]), *reverse*=True)

*# Remove the temporary score field*

*for* movie *in* matching\_movies:

            movie.pop("genre\_match\_score", None)

*return* matching\_movies[:*num\_recommendations*]

    def get\_personalized\_recommendations(*self*, *user\_preferences*: Dict[str, float], *num\_recommendations*: int = 5) -> List[Dict]:

        """

        Get personalized recommendations based on user genre preferences with weights.

        Args:

            user\_preferences (Dict[str, float]): Dictionary mapping genres to preference weights (0.0 to 1.0)

            num\_recommendations (int): Number of movies to recommend

        Returns:

            List[Dict]: List of recommended movies

        """

        scored\_movies = []

*for* movie *in* *self*.movies:

            total\_score = 0.0

*for* genre, weight *in* *user\_preferences*.items():

*if* genre.title() in [g.title() *for* g *in* movie["genres"]]:

                    total\_score += weight

*if* total\_score > 0:

*# Normalize score by number of genres and add rating bonus*

                normalized\_score = (total\_score / len(movie["genres"])) + (movie["rating"] / 10.0)

                movie\_copy = movie.copy()

                movie\_copy["personalized\_score"] = normalized\_score

                scored\_movies.append(movie\_copy)

*# Sort by personalized score*

        scored\_movies.sort(*key*=lambda *x*: *x*["personalized\_score"], *reverse*=True)

*# Remove the temporary score field*

*for* movie *in* scored\_movies:

            movie.pop("personalized\_score", None)

*return* scored\_movies[:*num\_recommendations*]

    def display\_recommendations(*self*, *recommendations*: List[Dict], *genre*: str = None):

        """

        Display movie recommendations in a formatted way.

        Args:

            recommendations (List[Dict]): List of recommended movies

            genre (str): The genre used for recommendations

        """

*if* not *recommendations*:

            print(f"No movies found for genre: {*genre*}")

*return*

        print(f"\n🎬 Movie Recommendations for {*genre* or 'your preferences'}:")

        print("=" \* 60)

*for* i, movie *in* enumerate(*recommendations*, 1):

            genres\_str = ", ".join(movie["genres"])

            print(f"{i}. {movie['title']} ({movie['year']})")

            print(f"   Genres: {genres\_str}")

            print(f"   Rating: ⭐ {movie['rating']}/10")

            print()

def main():

    """Main function to demonstrate the movie recommendation system."""

    recommender = MovieRecommender()

    print("🎭 Movie Recommendation System with Few-Shot Learning")

    print("=" \* 60)

*# Example 1: Single genre recommendation*

    print("\n1️⃣ Single Genre Recommendation:")

    drama\_movies = recommender.get\_genre\_recommendations("Drama", 3)

    recommender.display\_recommendations(drama\_movies, "Drama")

*# Example 2: Multi-genre recommendation*

    print("\n2️⃣ Multi-Genre Recommendation:")

    action\_sci\_fi = recommender.get\_multi\_genre\_recommendations(["Action", "Sci-Fi"], 4)

    recommender.display\_recommendations(action\_sci\_fi, "Action & Sci-Fi")

*# Example 3: Personalized recommendation*

    print("\n3️⃣ Personalized Recommendation:")

    user\_prefs = {"Drama": 0.8, "Comedy": 0.6, "Romance": 0.4}

    personalized = recommender.get\_personalized\_recommendations(user\_prefs, 4)

    recommender.display\_recommendations(personalized, "Personalized")

*# Interactive demo*

    print("\n🎯 Interactive Demo:")

    print("Enter a genre to get recommendations (or 'quit' to exit):")

*while* True:

        user\_input = input("\nEnter genre: ").strip()

*if* user\_input.lower() in ['quit', 'exit', 'q']:

            print("Thanks for using the Movie Recommendation System! 🎬")

*break*

*if* user\_input:

            recommendations = recommender.get\_genre\_recommendations(user\_input, 5)

            recommender.display\_recommendations(recommendations, user\_input)

*else*:

            print("Please enter a valid genre.")

*if* \_\_name\_\_ == "\_\_main\_\_":

    main()

task 4:

*import* csv

*import* os

*from* typing *import* List, Dict

*from* statistics *import* mean

class StudentGradeCalculator:

    def \_\_init\_\_(*self*):

        """Initialize the StudentGradeCalculator."""

*self*.students\_data = []

*self*.subjects = []

    def read\_csv\_file(*self*, *csv\_file\_path*: str) -> bool:

        """

        Read the CSV file and parse student data.

        Args:

            csv\_file\_path (str): Path to the CSV file containing student data

        Returns:

            bool: True if successful, False otherwise

        """

*try*:

*if* not os.path.exists(*csv\_file\_path*):

                print(f"❌ Error: File '{*csv\_file\_path*}' not found!")

*return* False

            print(f"🔍 Reading file: {*csv\_file\_path*}")

*with* open(*csv\_file\_path*, 'r', *newline*='', *encoding*='utf-8') *as* file:

                csv\_reader = csv.reader(file)

*# Read header row to get subjects*

                header = next(csv\_reader)

                print(f"📋 Header row: {header}")

*if* len(header) < 4:  *# Name + 3 subjects minimum*

                    print("❌ Error: CSV must have at least 4 columns (Name + 3 subjects)")

*return* False

*self*.subjects = header[1:]  *# Skip the 'Name' column*

                print(f"📚 Subjects found: {', '.join(*self*.subjects)}")

*# Read student data*

*for* row\_num, row *in* enumerate(csv\_reader, *start*=2):

                    print(f"📖 Processing row {row\_num}: {row}")

*if* len(row) != len(header):

                        print(f"⚠️  Warning: Row {row\_num} has {len(row)} columns, expected {len(header)}")

*continue*

*try*:

                        student\_name = row[0].strip()

                        marks = [float(mark.strip()) *for* mark *in* row[1:]]

                        print(f"   Student: {student\_name}, Marks: {marks}")

*# Validate marks (0-100 range)*

*if* any(mark < 0 *or* mark > 100 *for* mark *in* marks):

                            print(f"⚠️  Warning: Row {row\_num} has marks outside 0-100 range")

*continue*

                        total = sum(marks)

                        average = round(mean(marks), 2)

                        print(f"   Calculated - Total: {total}, Average: {average}")

                        student\_data = {

                            'name': student\_name,

                            'marks': marks,

                            'total': total,

                            'average': average

                        }

*self*.students\_data.append(student\_data)

*except* ValueError *as* e:

                        print(f"⚠️  Warning: Row {row\_num} has invalid marks: {e}")

*continue*

                print(f"✅ Successfully loaded {len(*self*.students\_data)} students")

*return* True

*except* Exception *as* e:

            print(f"❌ Error reading CSV file: {e}")

*return* False

    def display\_student\_results(*self*) -> None:

        """Display student results in a formatted table."""

*if* not *self*.students\_data:

            print("❌ No student data to display!")

*return*

*# Display header*

        print(f"\n📊 Student Results")

        print("=" \* 80)

*# Create header row*

        header = f"{'Name':<20} {'Total':<8} {'Average':<10}"

*for* subject *in* *self*.subjects:

            header += f" {subject:<8}"

        print(header)

        print("-" \* 80)

*# Display each student's data*

*for* student *in* *self*.students\_data:

            row = f"{student['name']:<20} {student['total']:<8} {student['average']:<10}"

*for* mark *in* student['marks']:

                row += f" {mark:<8}"

            print(row)

        print("-" \* 80)

    def display\_statistics(*self*) -> None:

        """Display overall statistics for the class."""

*if* not *self*.students\_data:

            print("❌ No statistics to display!")

*return*

*# Calculate statistics*

        all\_marks = []

*for* student *in* *self*.students\_data:

            all\_marks.extend(student['marks'])

        stats = {

            'total\_students': len(*self*.students\_data),

            'overall\_average': round(mean(all\_marks), 2),

            'highest\_total': max(student['total'] *for* student *in* *self*.students\_data),

            'lowest\_total': min(student['total'] *for* student *in* *self*.students\_data),

            'highest\_average': max(student['average'] *for* student *in* *self*.students\_data),

            'lowest\_average': min(student['average'] *for* student *in* *self*.students\_data)

        }

*# Subject-wise statistics*

        subject\_stats = {}

*for* i, subject *in* enumerate(*self*.subjects):

            subject\_marks = [student['marks'][i] *for* student *in* *self*.students\_data]

            subject\_stats[subject] = {

                'average': round(mean(subject\_marks), 2),

                'highest': max(subject\_marks),

                'lowest': min(subject\_marks)

            }

        print(f"\n📈 Class Statistics")

        print("=" \* 50)

        print(f"Total Students: {stats['total\_students']}")

        print(f"Overall Class Average: {stats['overall\_average']}%")

        print(f"Highest Total Marks: {stats['highest\_total']}")

        print(f"Lowest Total Marks: {stats['lowest\_total']}")

        print(f"Highest Average: {stats['highest\_average']}%")

        print(f"Lowest Average: {stats['lowest\_average']}%")

        print(f"\n📚 Subject-wise Statistics:")

        print("-" \* 50)

*for* subject, subject\_stat *in* subject\_stats.items():

            print(f"{subject}:")

            print(f"  Average: {subject\_stat['average']}%")

            print(f"  Highest: {subject\_stat['highest']}%")

            print(f"  Lowest: {subject\_stat['lowest']}%")

    def find\_top\_performers(*self*, *n*: int = 3) -> List[Dict]:

        """

        Find top n performers based on total marks.

        Args:

            n (int): Number of top performers to find

        Returns:

            List[Dict]: List of top performing students

        """

*if* not *self*.students\_data:

*return* []

        sorted\_students = sorted(*self*.students\_data, *key*=lambda *x*: *x*['total'], *reverse*=True)

*return* sorted\_students[:*n*]

    def display\_top\_performers(*self*, *n*: int = 3) -> None:

        """

        Display top n performers.

        Args:

            n (int): Number of top performers to display

        """

        top\_performers = *self*.find\_top\_performers(*n*)

*if* not top\_performers:

            print("❌ No top performers to display!")

*return*

        print(f"\n🏆 Top {*n*} Performers")

        print("=" \* 60)

*for* i, student *in* enumerate(top\_performers, 1):

            print(f"{i}. {student['name']}")

            print(f"   Total: {student['total']} | Average: {student['average']}%")

            print(f"   Marks: {', '.join(f'{mark}%' *for* mark *in* student['marks'])}")

            print()

    def export\_results(*self*, *output\_file*: str) -> bool:

        """

        Export results to a new CSV file.

        Args:

            output\_file (str): Name of the output CSV file

        Returns:

            bool: True if successful, False otherwise

        """

*try*:

*with* open(*output\_file*, 'w', *newline*='', *encoding*='utf-8') *as* file:

                csv\_writer = csv.writer(file)

*# Write header*

                header = ['Name', 'Total', 'Average'] + *self*.subjects

                csv\_writer.writerow(header)

*# Write student data*

*for* student *in* *self*.students\_data:

                    row = [student['name'], student['total'], student['average']] + student['marks']

                    csv\_writer.writerow(row)

            print(f"✅ Results exported to '{*output\_file*}'")

*return* True

*except* Exception *as* e:

            print(f"❌ Error exporting results: {e}")

*return* False

def create\_sample\_csv(*filename*: str = "sample\_students.csv") -> None:

    """

    Create a sample CSV file with student data for testing.

    Args:

        filename (str): Name of the CSV file to create

    """

    sample\_data = [

        ['Name', 'Mathematics', 'Science', 'English'],

        ['Alice Johnson', '85', '92', '78'],

        ['Bob Smith', '92', '88', '85'],

        ['Carol Davis', '78', '95', '90'],

        ['David Wilson', '88', '82', '88'],

        ['Emma Brown', '95', '89', '92'],

        ['Frank Miller', '82', '85', '79'],

        ['Grace Lee', '90', '91', '87'],

        ['Henry Taylor', '87', '84', '91'],

        ['Ivy Chen', '93', '88', '85'],

        ['Jack Anderson', '86', '90', '83']

    ]

*try*:

*with* open(*filename*, 'w', *newline*='', *encoding*='utf-8') *as* file:

            csv\_writer = csv.writer(file)

            csv\_writer.writerows(sample\_data)

        print(f"✅ Sample CSV file '{*filename*}' created successfully!")

        print("📝 Sample data includes 10 students with marks in Mathematics, Science, and English")

*except* Exception *as* e:

        print(f"❌ Error creating sample CSV: {e}")

def main():

    """Main function to demonstrate the Student Grade Calculator."""

    print("🎓 Student Grade Calculator")

    print("=" \* 50)

*# Ask user if they want to create a sample CSV*

    print("\nDo you want to create a sample CSV file for testing?")

    create\_sample = input("Enter 'y' for yes, any other key for no: ").strip().lower()

*if* create\_sample == 'y':

        sample\_filename = input("Enter sample CSV filename (default: sample\_students.csv): ").strip()

*if* not sample\_filename:

            sample\_filename = "sample\_students.csv"

        create\_sample\_csv(sample\_filename)

        print()

*# Get CSV file path from user*

*while* True:

        csv\_file\_path = input("\n📁 Enter the path to your CSV file: ").strip()

*if* not csv\_file\_path:

            print("❌ Please enter a valid file path!")

*continue*

*# Remove quotes if user included them*

        csv\_file\_path = csv\_file\_path.strip('"\'')

*if* os.path.exists(csv\_file\_path):

*break*

*else*:

            print(f"❌ File '{csv\_file\_path}' not found! Please check the path and try again.")

*# Initialize calculator*

    calculator = StudentGradeCalculator()

*# Read CSV file*

*if* not calculator.read\_csv\_file(csv\_file\_path):

        print("❌ Failed to read CSV file. Exiting...")

*return*

    print()

*# Display results*

    calculator.display\_student\_results()

*# Display statistics*

    calculator.display\_statistics()

*# Display top performers*

    calculator.display\_top\_performers(5)

*# Ask if user wants to export results*

    export\_choice = input("\n💾 Do you want to export results to a new CSV file? (y/n): ").strip().lower()

*if* export\_choice == 'y':

        output\_filename = input("Enter output filename (default: student\_results.csv): ").strip()

*if* not output\_filename:

            output\_filename = "student\_results.csv"

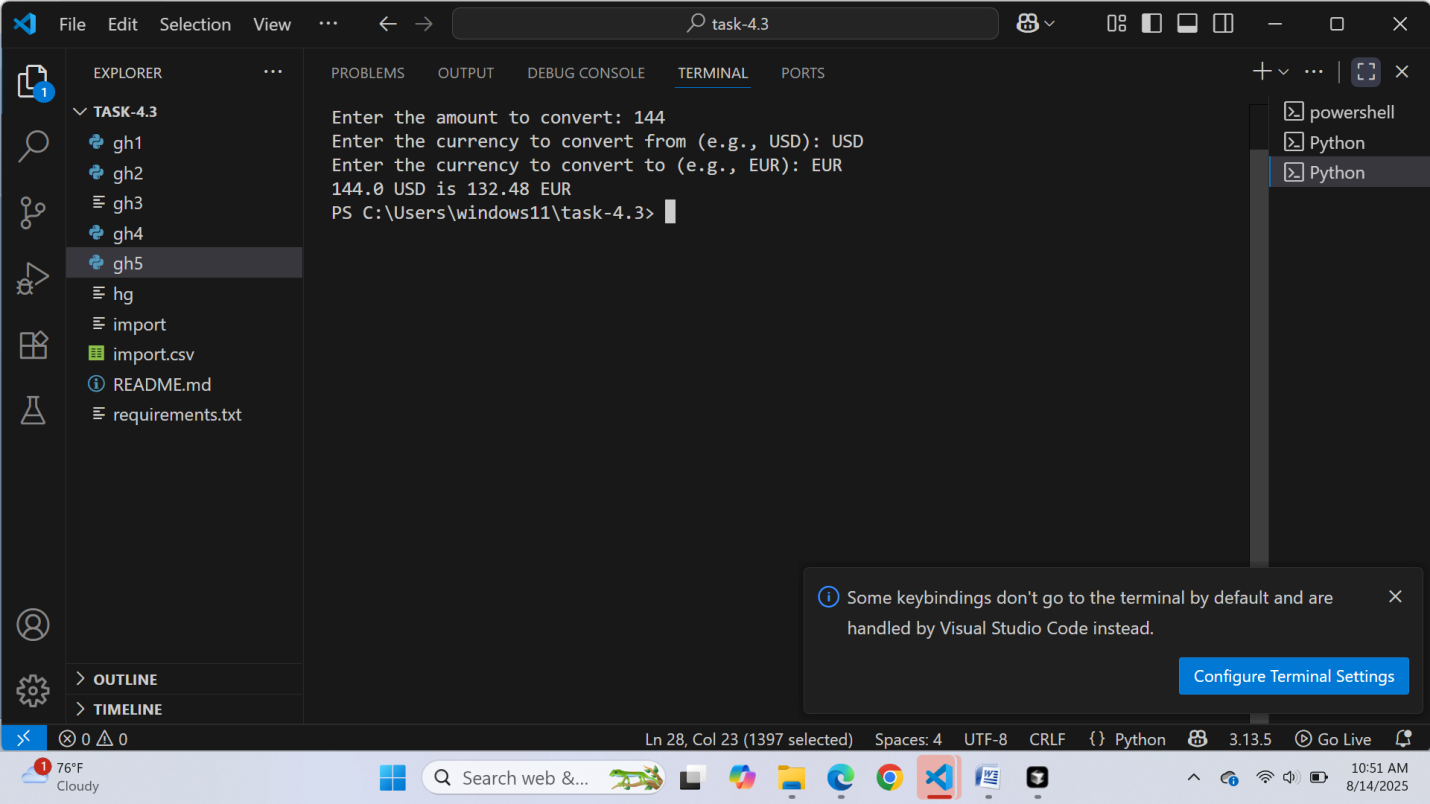
        calculator.export\_results(output\_filename)

    print("\n🎉 Analysis complete!")

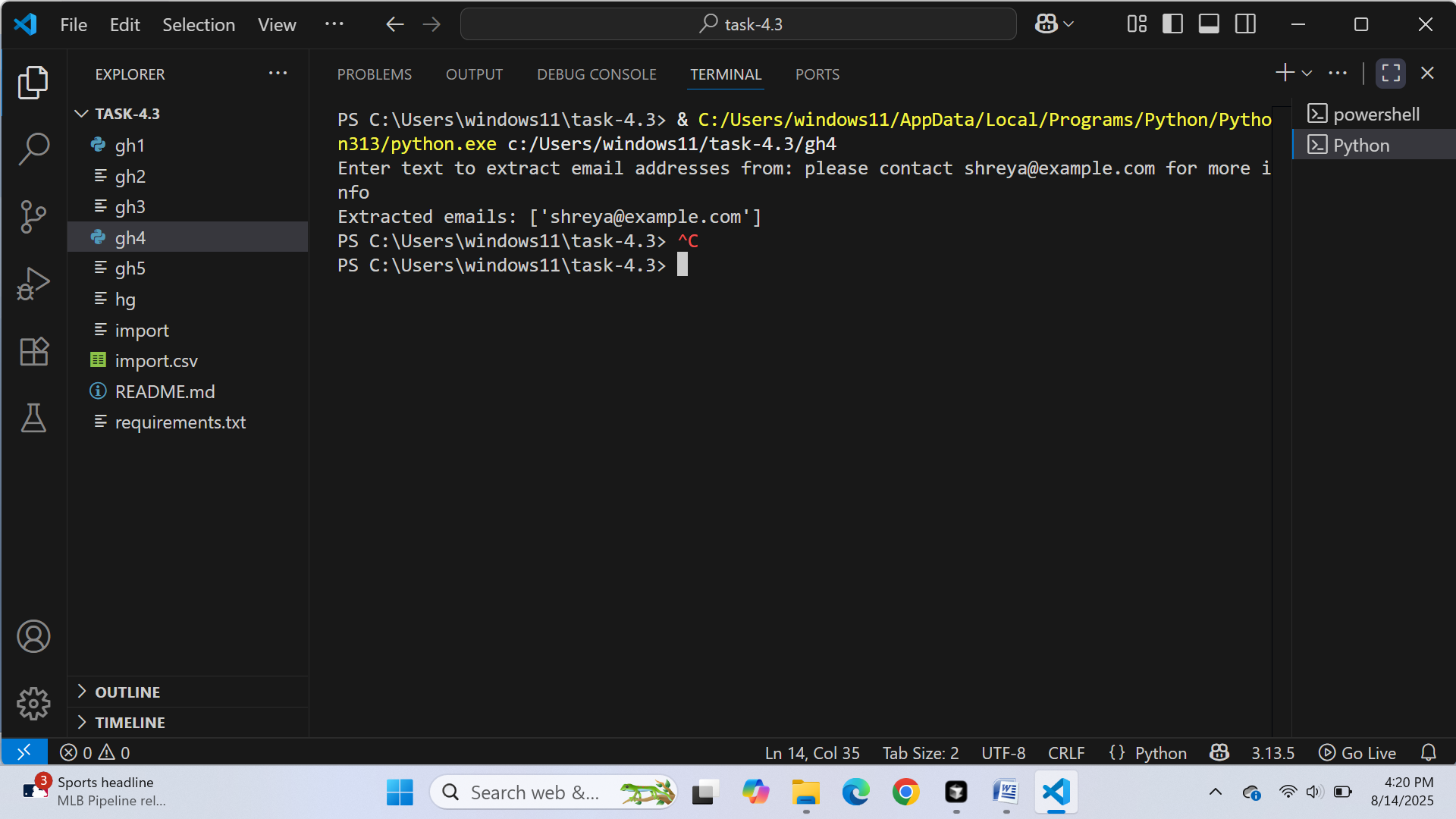
    print("💡 You can run the script again with different CSV files to analyze different data.")

*if* \_\_name\_\_ == "\_\_main\_\_":

    main()

task 1 output: 

task 2 ouput:



Task 3 output:



Task 4 output:

