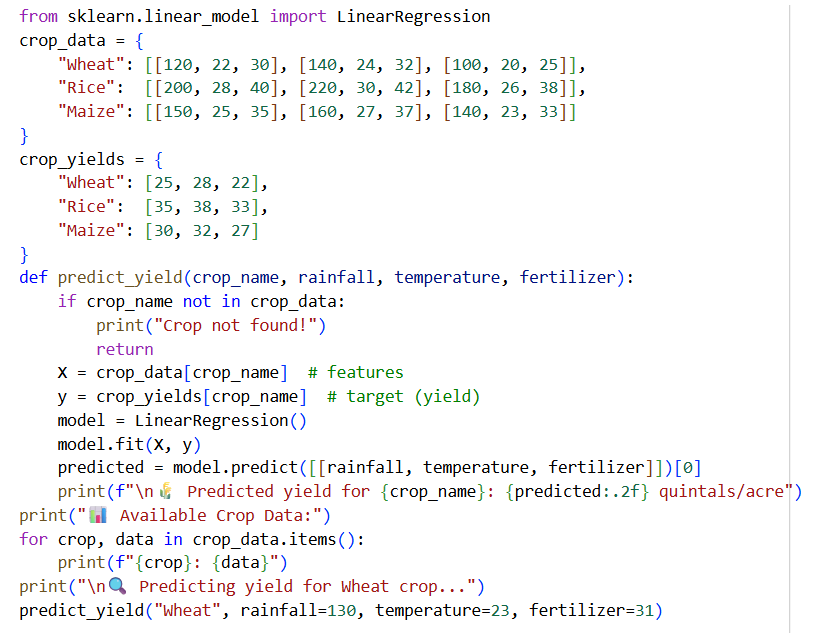
**AI ASSISSTED CODING EXAM:3**

**Set E4**

Q1:  
Scenario: In the Agriculture sector, a company faces a challenge related to data structures with ai.  
Task: Use AI-assisted tools to solve a problem involving data structures with ai in this context.  
Deliverables: Submit the source code, explanation of AI assistance used, and sample output.

Q2:  
Scenario: In the Education sector, a company faces a challenge related to backend api development.  
Task: Use AI-assisted tools to solve a problem involving backend api development in this context.  
Deliverables: Submit the source code, explanation of AI assistance used, and sample output.

Q1 code:



A white background with black text

AI-generated content may be incorrect.

**EXPLANATION:**

from sklearn.linear\_model import LinearRegression crop\_data = { "Wheat": [[120, 22, 30], [140, 24, 32], [100, 20, 25]], "Rice": [[200, 28, 40], [220, 30, 42], [180, 26, 38]], "Maize": [[150, 25, 35], [160, 27, 37], [140, 23, 33]] } crop\_yields = { "Wheat": [25, 28, 22], "Rice": [35, 38, 33], "Maize": [30, 32, 27] } def predict\_yield(crop\_name, rainfall, temperature, fertilizer): if crop\_name not in crop\_data: print("Crop not found!") return X = crop\_data[crop\_name] # features y = crop\_yields[crop\_name] # target (yield) model = LinearRegression() model.fit(X, y) predicted = model.predict([[rainfall, temperature, fertilizer]])[0] print(f"\n🌾 Predicted yield for {crop\_name}: {predicted:.2f} quintals/acre") print("📊 Available Crop Data:") for crop, data in crop\_data.items(): print(f"{crop}: {data}") print("\n🔍 Predicting yield for Wheat crop...") predict\_yield("Wheat", rainfall=130, temperature=23, fertilizer=31) explain the code

This code snippet uses the LinearRegression model from the sklearn.linear\_model library to predict crop yields based on rainfall, temperature, and fertilizer.

Here's a breakdown:

1. **from sklearn.linear\_model import LinearRegression**: This line imports the necessary class for creating a linear regression model.
2. **crop\_data and crop\_yields**: These dictionaries store sample data for different crops.
   * crop\_data holds the features (rainfall, temperature, fertilizer) for each crop as a list of lists.
   * crop\_yields holds the corresponding target variable (yield) for each crop as a list.
3. **predict\_yield(crop\_name, rainfall, temperature, fertilizer) function**: This function takes the crop name and environmental factors as input and predicts the yield.
   * It first checks if the crop\_name exists in the crop\_data dictionary.
   * If the crop is found, it extracts the feature data (X) and yield data (y) for that crop.
   * **model = LinearRegression()**: A Linear Regression model is initialized.
   * **model.fit(X, y)**: The model is trained using the provided crop data. It learns the relationship between the features and the yield.
   * **predicted = model.predict([[rainfall, temperature, fertilizer]])[0]**: The trained model is used to predict the yield for the given input values of rainfall, temperature, and fertilizer. The [[...]] is necessary because the predict method expects a 2D array-like input.
   * Finally, it prints the predicted yield formatted to two decimal places.
4. **print("📊 Available Crop Data:") and the loop**: This part of the code simply prints the sample crop data stored in the crop\_data dictionary.
5. **print("\n🔍 Predicting yield for Wheat crop...")**: This line indicates that a prediction is about to be made for Wheat.
6. **predict\_yield("Wheat", rainfall=130, temperature=23, fertilizer=31)**: This line calls the predict\_yield function with specific input values for Wheat to demonstrate its usage.

**Q2:**

A computer screen shot of a program

AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

**EXPLANATION:**

Here's a breakdown of the code:

1. **from flask import Flask, jsonify, request**: This line imports necessary components from the Flask library:
   * Flask: The main class for creating a Flask application.
   * jsonify: A helper function to convert Python dictionaries to JSON responses.
   * request: An object that contains incoming request data (like JSON payloads).
2. **app = Flask(\_\_name\_\_)**: This initializes the Flask application.
3. **students, courses, enrollments = {}, {}, {}**: These lines initialize three empty dictionaries. These dictionaries will store the application's data in memory:
   * students: Stores student information, with student IDs as keys.
   * courses: Stores course information, with course IDs as keys.
   * enrollments: Stores enrollment information, mapping course IDs to a list of student IDs enrolled in that course.
4. **@app.post('/students') def add\_student(): ...**: This defines an endpoint that handles HTTP POST requests to the /students URL.
   * It's used to add a new student.
   * A new student ID is generated based on the current number of students.
   * It reads the student data (name and email) from the incoming JSON request body using request.json.
   * The new student's data is stored in the students dictionary.
   * It returns the newly created student's data as a JSON response.
5. **@app.post('/courses') def add\_course(): ...**: This defines an endpoint that handles HTTP POST requests to the /courses URL.
   * It's used to add a new course.
   * A new course ID is generated.
   * It reads the course data (name and instructor) from the incoming JSON request body.
   * The new course's data is stored in the courses dictionary.
   * It returns the newly created course's data as a JSON response.
6. **@app.post('/enroll') def enroll(): ...**: This defines an endpoint that handles HTTP POST requests to the /enroll URL.
   * It's used to enroll a student in a course.
   * It reads the student\_id and course\_id from the incoming JSON request body.
   * It checks if both the student and course IDs are valid (exist in their respective dictionaries). If not, it returns an error.
   * If valid, it adds the student\_id to the list of students enrolled in the specified course\_id within the enrollments dictionary using setdefault.
   * It returns a success message as a JSON response.
7. **@app.get('/courses/<int:c\_id>/students') def get\_students(c\_id): ...**: This defines an endpoint that handles HTTP GET requests to the /courses/<c\_id>/students URL, where <c\_id> is a variable part of the URL representing the course ID (converted to an integer by Flask).
   * It's used to get all students enrolled in a specific course.
   * It retrieves the list of student IDs for the given c\_id from the enrollments dictionary.
   * It then retrieves the full student data for each student ID from the students dictionary.
   * It returns the list of student data as a JSON response.
8. **@app.get('/students/<int:s\_id>/courses') def get\_courses(s\_id): ...**: This defines an endpoint that handles HTTP GET requests to the /students/<s\_id>/courses URL, where <s\_id> is the student ID.
   * It's used to get all courses a specific student is enrolled in.
   * It iterates through the enrollments dictionary to find which courses (c) have the given s\_id in their list of enrolled students (s).
   * For each found course ID, it retrieves the full course data from the courses dictionary.
   * It returns the list of course data as a JSON response.
9. **if \_\_name\_\_ == '\_\_main\_\_': app.run(debug=True)**: This is the standard Python way to run the Flask development server when the script is executed directly. debug=True enables debug mode, which provides helpful error messages and automatically reloads the server on code changes.

This code demonstrates basic principles of building a simple RESTful API using Flask to manage relationships between different data entities (students, courses, enrollments).