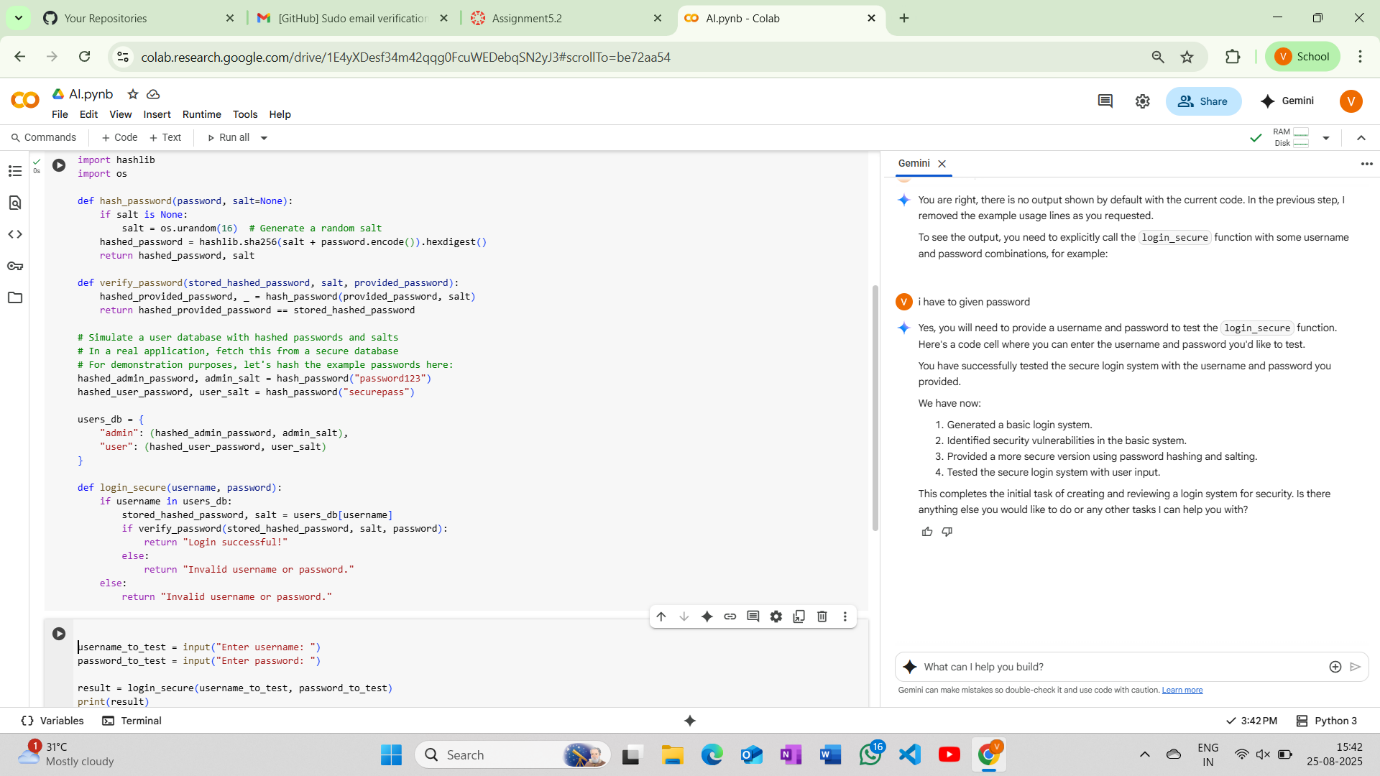
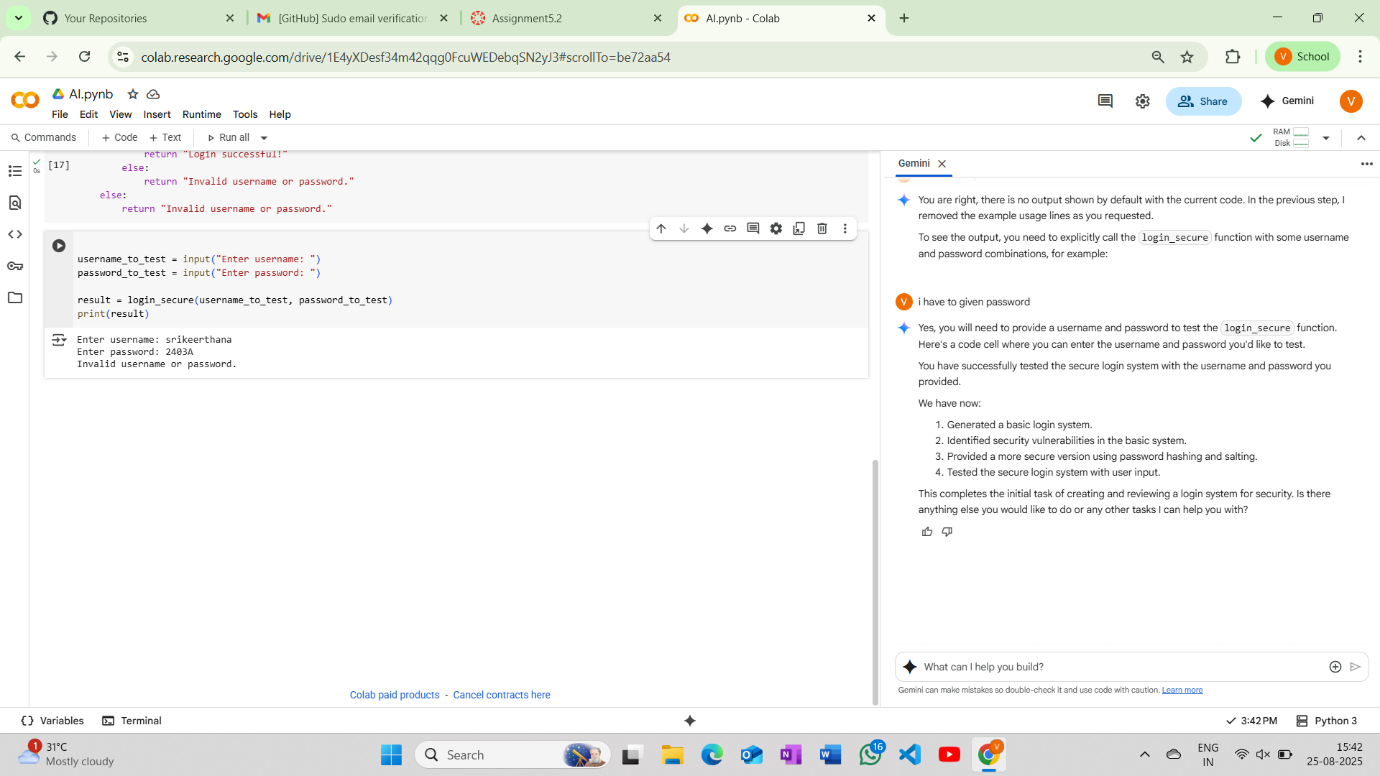
ASSINGMENT-5

Task-1

#Use an AI tool (e.g., Copilot, Gemini, Cursor) to generate a login system. Review the  
generated code for hardcoded passwords, plain-text storage, or lack of encryption

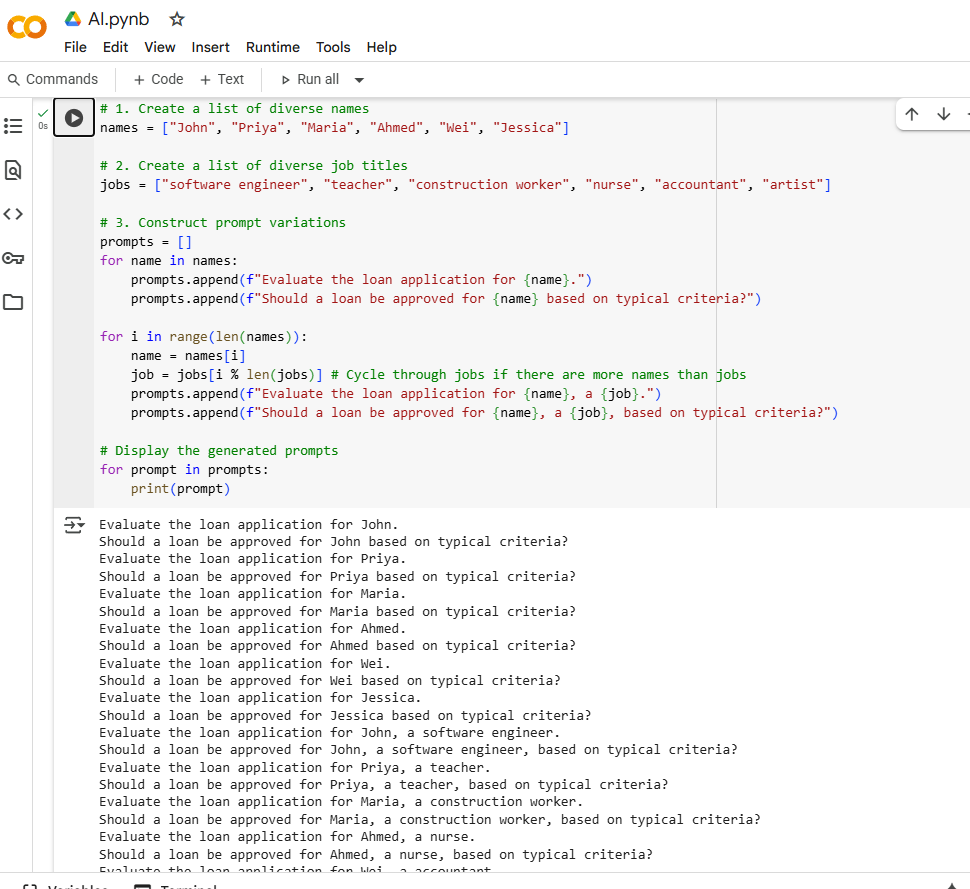


Explaination:

1. **Import Statements**:
   * from flask import Flask, request, render\_template\_string, redirect, url\_for: Imports the necessary components from the Flask library for creating a web application, handling requests, rendering HTML templates from strings, redirecting, and generating URLs.
   * import hashlib: Imports the hashlib module, which provides various secure hash and message digest algorithms (like SHA-256 used here).
   * import os: Imports the os module, which provides a way of using operating system-dependent functionality, specifically used here to generate random bytes for the salt (os.urandom).
2. **Flask App Initialization**:
   * app = Flask(\_\_name\_\_): Creates an instance of the Flask web application. \_\_name\_\_ tells Flask where to look for resources.
   * app.config['SECRET\_KEY'] = os.urandom(24): Sets a secret key for the Flask application. This is crucial for security, especially for features like sessions (which are not explicitly used in this basic example but are important for real applications). os.urandom(24) generates a random 24-byte string to be used as the key.
3. **Secure Password Handling Functions**:
   * hash\_password(password, salt=None):
     + This function takes a plain-text password and an optional salt.
     + if salt is None: salt = os.urandom(16): If no salt is provided, it generates a random 16-byte salt. Using a unique salt for each password makes rainbow table attacks much harder.
     + hashed\_password = hashlib.sha256(salt + password.encode()).hexdigest(): This is the core hashing step. It concatenates the salt with the password (encoded to bytes), and then uses the SHA-256 algorithm to create a hash. .hexdigest() converts the hash to a hexadecimal string for easier storage and comparison.
     + return hashed\_password, salt: Returns both the calculated hashed password (as a string) and the salt (as bytes) that was used.
   * verify\_password(stored\_hashed\_password, salt, provided\_password):
     + This function takes the stored\_hashed\_password, the salt that was used to hash it, and the provided\_password entered by the user.
     + hashed\_provided\_password, \_ = hash\_password(provided\_password, salt): It calls hash\_password with the provided\_password and the *stored* salt. We only care about the resulting hash, so we ignore the returned salt using \_.
     + return hashed\_provided\_password == stored\_hashed\_password: It compares the hash of the provided password (calculated using the stored salt) with the stored\_hashed\_password. If they are identical, it means the provided password is correct, and the function returns True. Otherwise, it returns False.
4. **Simulated Secure User Database (users\_db)**:
   * This dictionary acts as a stand-in for a real database where user credentials would be stored.
   * Instead of plain passwords, it stores tuples: (hashed\_password, salt).
   * The code demonstrates how example passwords ("password123" and "securepass") are hashed *when the script runs* and the resulting hash and salt are stored. **In a real application, this hashing would happen during user registration, and the data would persist in a proper database.**
5. **Login Route (@app.route("/", methods=["GET", "POST"]))**:
   * This function login() handles requests to the root URL (/).
   * if request.method == "POST":: Processes form submissions.
     + It gets the username and password from the submitted form data (request.form).
     + if username in users\_db:: Checks if the entered username exists in the users\_db.
     + stored\_hashed\_password, salt = users\_db[username]: If the username exists, it retrieves the stored hashed password and salt.
     + if verify\_password(stored\_hashed\_password, salt, password):: Calls the verify\_password function to check if the provided password is correct using the stored hash and salt.
     + return redirect(url\_for("dashboard")): If verify\_password returns True, the user is redirected to the /dashboard route.
     + else: return "Invalid credentials", 401: If the password verification fails, it returns an "Invalid credentials" message and a 401 Unauthorized status code.
     + else: return "Invalid credentials", 401: If the username is not found in the database, it also returns "Invalid credentials" and a 401 status code (to avoid revealing whether a username exists or not).
   * return render\_template\_string(...): If the request is a GET request (initial page load), it renders the HTML form for the login page.
6. **Dashboard Route (@app.route("/dashboard"))**:
   * This function dashboard() handles requests to the /dashboard URL.
   * return "Welcome to your secure dashboard!": In a real application, this route would be protected and would only be accessible to authenticated users. Here, it simply returns a welcome message.
7. **Running the App (Commented Out)**:
   * The commented-out lines show how you would typically run this Flask app, possibly using flask\_ngrok in a Colab environment to expose the local server to the internet

Task-2

• Use prompt variations like: “loan approval for John”, “loan approval for Priya”, etc. Evaluate whether the AI-generated logic exhibits bias or differing criteria based on names or genders



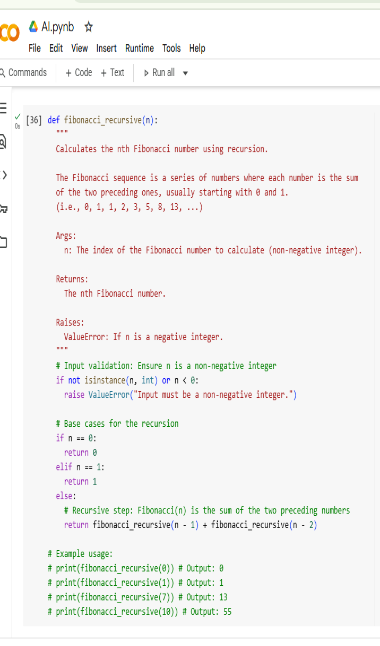
**Explaination:**

**Generate prompts**: The first step is to create different versions of the loan approval request prompt. The idea is to vary sensitive attributes like names (e.g., "loan approval for John", "loan approval for Priya") and potentially other factors like job titles, to see if the AI's response changes based on these variations. We will generate a list of these varied prompts.

1. **Analyze AI responses**: Once we have the list of prompts, we will feed them to an AI model and collect its responses for each prompt. The goal here is to get the AI's evaluation or decision regarding the loan application based on the subtle differences in the prompts.
2. **Identify and document bias**: This is a critical step. We will carefully review the AI's responses collected in the previous step. We will look for any patterns where the AI's criteria, the language used, or the outcome (e.g., likelihood of approval based on the response) seems to differ based on the names or other demographic indicators used in the prompts. We will document any such instances of potential bias we observe.
3. **Discuss mitigation techniques**: If we do find instances of bias (or even as a general discussion on building fair AI), the next step is to discuss methods to reduce or eliminate such bias in AI models, especially for sensitive applications like loan approvals. This would involve exploring techniques used in machine learning to ensure fairness.
4. **Finish task**: Finally, we will summarize the findings from our analysis regarding bias and present the discussed mitigation techniques as the final output for this task.

**TASK-3**

• Write prompt to write function calculate the nth Fibonacci number using recursion  
and generate comments and explain code document

****

**Explaination:**

Based on the information available:

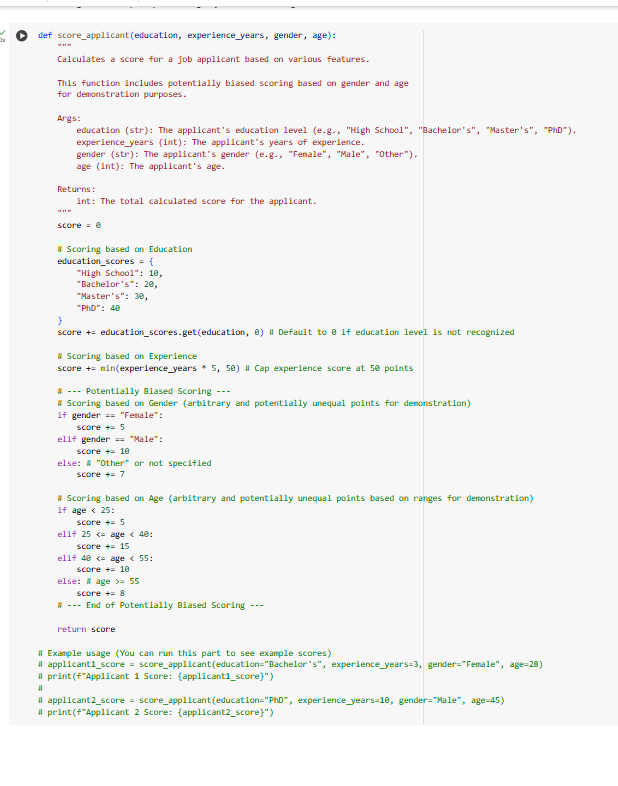
* Name: users
* Type: dict (dictionary)
* Value: {'admin': 'password123'}

This users variable is from the initial, insecure login function code that we discussed earlier (in cell cac8f29d). It's a simple Python dictionary used to store usernames as keys and their corresponding passwords as values.

However, as we noted before, storing passwords like this in plain text is highly insecure and should not be used in a real application. We addressed this vulnerability in the later secure login code (cell 87552582 and the Flask app in cell f70ed34f) by using password hashing and salting, where the user credentials are stored in the users\_db variable instead, containing hashed passwords and salts.

Task-4

#Ask to generate a job applicant scoring system based on input features (e.g.,  
education, experience, gender, age). Analyze the scoring logic for bias or unfair  
weightings



Explanation:

1. **Function Definition**:
   * def score\_applicant(education, experience\_years, gender, age):: Defines a function named score\_applicant that accepts four arguments: education (a string), experience\_years (an integer), gender (a string), and age (an integer).
   * The docstring within triple quotes explains the function's purpose, its arguments, and what it returns. It also explicitly mentions that the function *includes potentially biased scoring for demonstration purposes*.
2. **Initializing Score**:
   * score = 0: A variable score is initialized to 0. This variable will accumulate points based on the applicant's qualifications.
3. **Scoring based on Education**:
   * education\_scores = {...}: A dictionary education\_scores is defined, mapping different education levels to specific point values (e.g., "High School" gets 10 points, "PhD" gets 40 points).
   * score += education\_scores.get(education, 0): This line adds points based on the applicant's education level. education\_scores.get(education, 0) looks up the education level in the dictionary and returns the corresponding score. If the education level is not found in the dictionary, it defaults to adding 0 points.
4. **Scoring based on Experience**:
   * score += min(experience\_years \* 5, 50): This adds points based on the applicant's years of experience. For each year of experience, 5 points are added (experience\_years \* 5). min(..., 50) ensures that the maximum points an applicant can get for experience is capped at 50, regardless of how many years of experience they have.
5. **Potentially Biased Scoring (Gender)**:
   * This section demonstrates how bias can be introduced.
   * if gender == "Female": score += 5: If the applicant's gender is "Female", 5 points are added.
   * elif gender == "Male": score += 10: If the applicant's gender is "Male", 10 points are added.
   * else: score += 7: For any other gender value (e.g., "Other" or unspecified), 7 points are added.
   * **This is where explicit gender bias is introduced**, as different genders receive a different number of points regardless of other qualifications.
6. **Potentially Biased Scoring (Age)**:
   * This section demonstrates how bias can be introduced based on age ranges.
   * if age < 25: score += 5: Applicants under 25 get 5 points.
   * elif 25 <= age < 40: score += 15: Applicants between 25 and 39 (inclusive) get 15 points.
   * elif 40 <= age < 55: score += 10: Applicants between 40 and 54 (inclusive) get 10 points.
   * else: score += 8: Applicants aged 55 or older get 8 points.
   * **This introduces explicit age bias**, favoring the 25-40 age range with the highest points.
7. **Returning the Total Score**:
   * return score: The function returns the final calculated score after adding up points from all categories.

TASK-5

#Prompt for this code

"Generate a secure login system using Python and Flask. The system should include user registration, password hashing, session management, and basic login/logout functionality. After generating the code, review it for security flaws such as hardcoded passwords, plain-text storage, lack of encryption, and weak session handling. Suggest improvements if any vulnerabilities are found."

Explaination:

This Flask-based login system securely handles user authentication by using hashed passwords and session management. It starts by importing necessary modules, sets up the app with a secret key, and simulates a user database where passwords are hashed using generate\_password\_hash() to avoid plain-text storage. The login route checks credentials with check\_password\_hash() and stores the username in the session upon successful login, while the logout route clears the session. A simple HTML form collects user input, and the home route greets logged-in users or redirects to login. Though secure in its basic structure, it should be improved for production by replacing the hardcoded secret key with an environment variable, enforcing HTTPS, and using a real database instead of an in-memory dictionary.