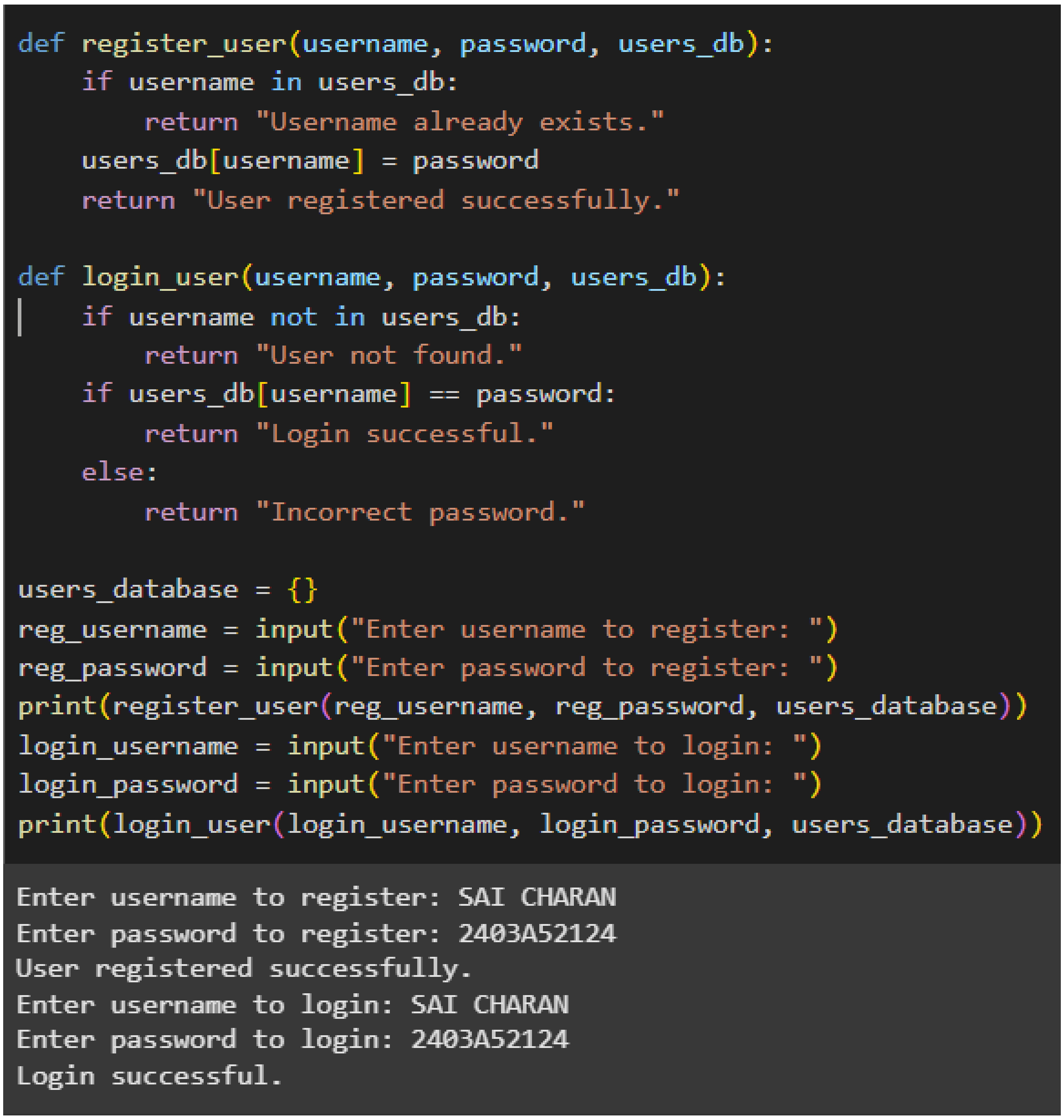
ASSISGNMENT 5

RAMIDI . SAI CHARAN

2403A52124

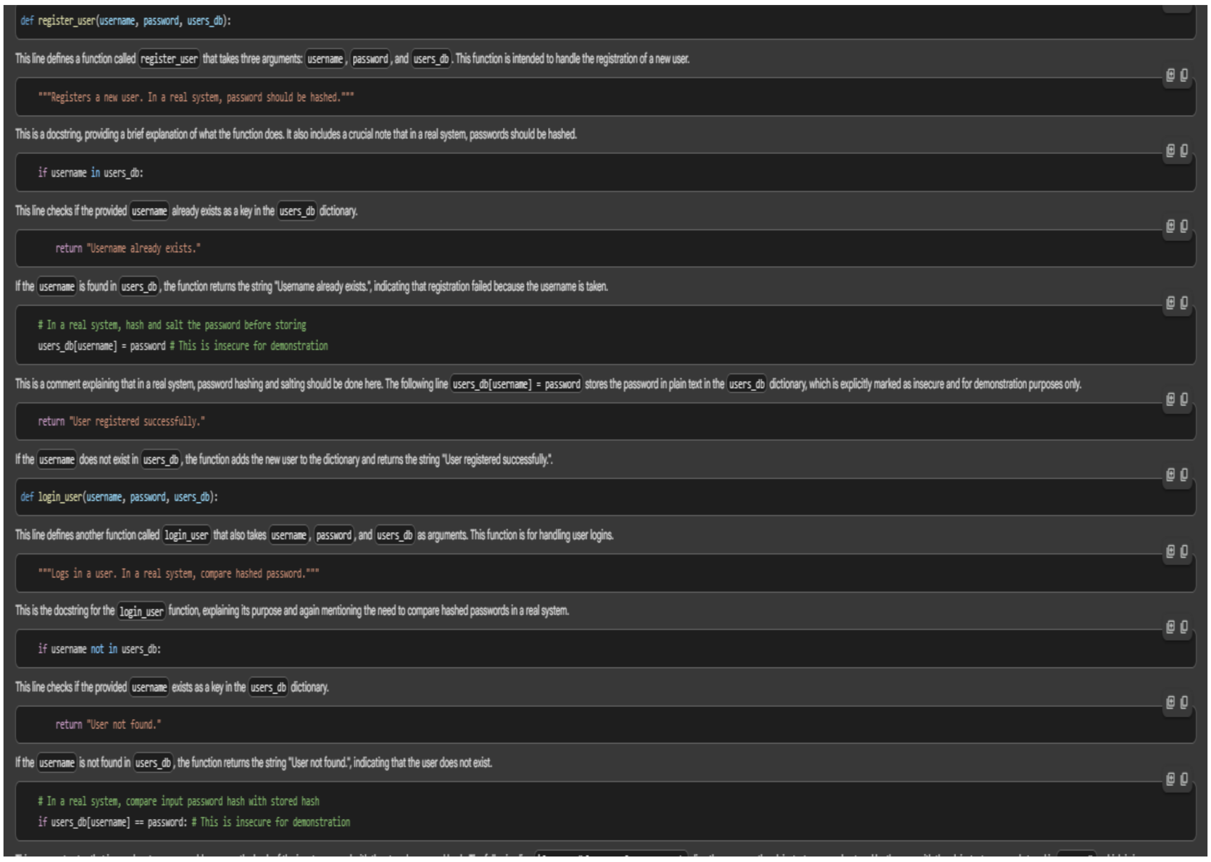
TASK 1:



```
def register_user(username, password, users_db):
    if username in users_db:
        return "Username already exists."
    users_db[username] = password
    return "User registered successfully."

def login_user(username, password, users_db):
    if username not in users_db:
        return "User not found."
    if users_db[username] == password:
        return "Login successful."
    else:
        return "Incorrect password."

users_database = {}
reg_username = input("Enter username to register: ")
reg_password = input("Enter password to register: ")
print(register_user(reg_username, reg_password, users_database))
login_username = input("Enter username to login: ")
login_password = input("Enter password to login: ")
print(login_user(login_username, login_password, users_database))
```

```
Enter username to register: SAI CHARAN
Enter password to register: 2403A52124
User registered successfully.
Enter username to login: SAI CHARAN
Enter password to login: 2403A52124
Login successful.
```

EXPLANATION :



```
def register_user(username, password, users_db):
```

This line defines a function called `register_user` that takes three arguments: `username`, `password`, and `users_db`. This function is intended to handle the registration of a new user.

```
    """Registers a new user. In a real system, password should be hashed."""
```

This is a docstring, providing a brief explanation of what the function does. It also includes a crucial note that in a real system, passwords should be hashed.

```
    if username in users_db:
```

This line checks if the provided `username` already exists as a key in the `users_db` dictionary.

```
        return "Username already exists."
```

If the `username` is found in `users_db`, the function returns the string "Username already exists.", indicating that registration failed because the username is taken.

```
    # In a real system, hash and salt the password before storing
    users_db[username] = password # This is insecure for demonstration
```

This is a comment explaining that in a real system, password hashing and salting should be done here. The following line `users_db[username] = password` stores the password in plain text in the `users_db` dictionary, which is explicitly marked as insecure and for demonstration purposes only.

```
    return "User registered successfully."
```

If the `username` does not exist in `users_db`, the function adds the new user to the dictionary and returns the string "User registered successfully.".

```
def login_user(username, password, users_db):
```

This line defines another function called `login_user` that also takes `username`, `password`, and `users_db` as arguments. This function is for handling user logins.

```
    """Logs in a user. In a real system, compare hashed password."""
```

This is the docstring for the `login_user` function, explaining its purpose and again mentioning the need to compare hashed passwords in a real system.

```
    if username not in users_db:
```

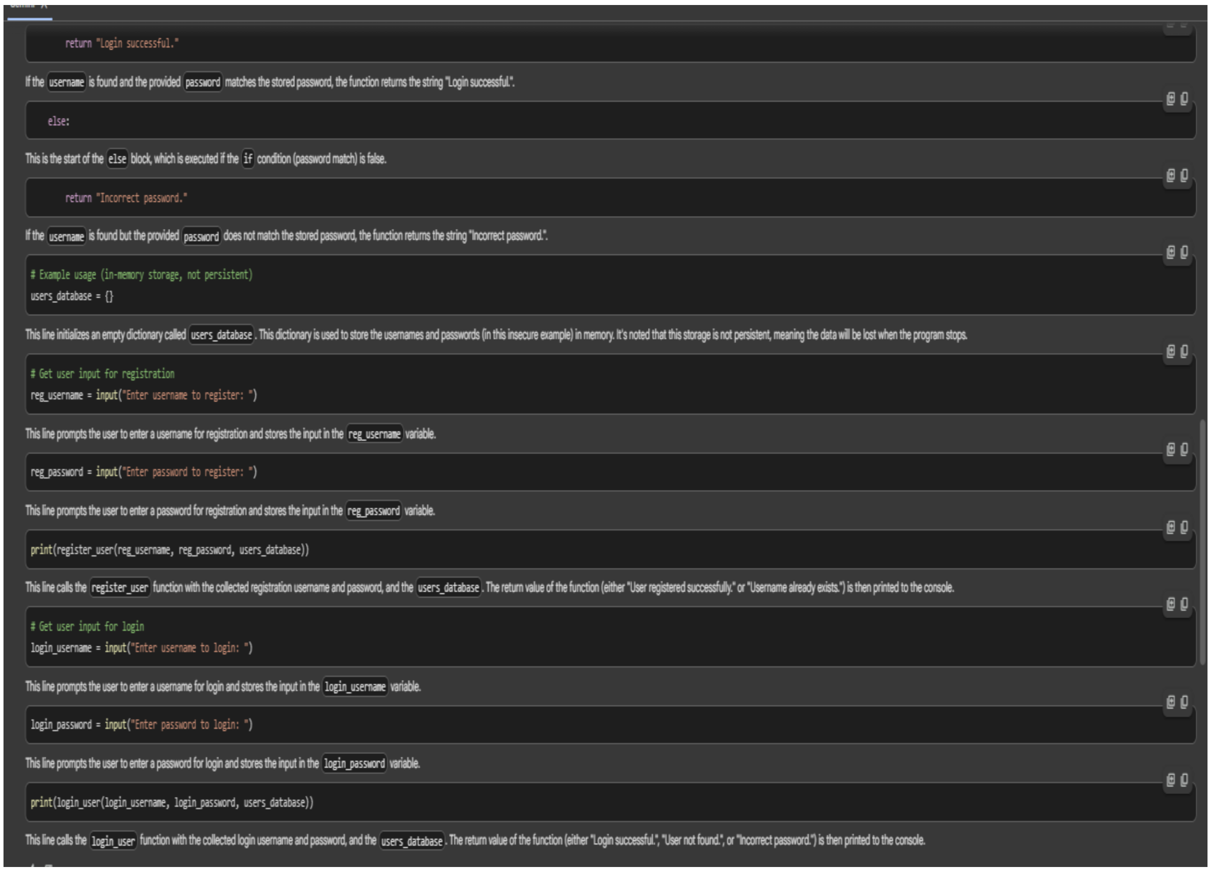
This line checks if the provided `username` exists as a key in the `users_db` dictionary.

```
        return "User not found."
```

If the `username` is not found in `users_db`, the function returns the string "User not found.", indicating that the user does not exist.

```
    # In a real system, compare input password hash with stored hash
    if users_db[username] == password: # This is insecure for demonstration
```

[illegible]



```
        return "Login successful."
```

If the `username` is found and the provided `password` matches the stored password, the function returns the string "Login successful.".

```
    else:
```

This is the start of the `else` block, which is executed if the `if` condition (password match) is false.

```
        return "Incorrect password."
```

If the `username` is found but the provided `password` does not match the stored password, the function returns the string "Incorrect password.".

```
# Example usage (in-memory storage, not persistent)
users_database = {}
```

This line initializes an empty dictionary called `users_database`. This dictionary is used to store the usernames and passwords (in this insecure example) in memory. It's noted that this storage is not persistent, meaning the data will be lost when the program stops.

```
# Get user input for registration
reg_username = input("Enter username to register: ")
```

This line prompts the user to enter a username for registration and stores the input in the `reg_username` variable.

```
reg_password = input("Enter password to register: ")
```

This line prompts the user to enter a password for registration and stores the input in the `reg_password` variable.

```
print(register_user(reg_username, reg_password, users_database))
```

This line calls the `register_user` function with the collected registration username and password, and the `users_database`. The return value of the function (either "User registered successfully." or "Username already exists.") is then printed to the console.

```
# Get user input for login
login_username = input("Enter username to login: ")
```

This line prompts the user to enter a username for login and stores the input in the `login_username` variable.

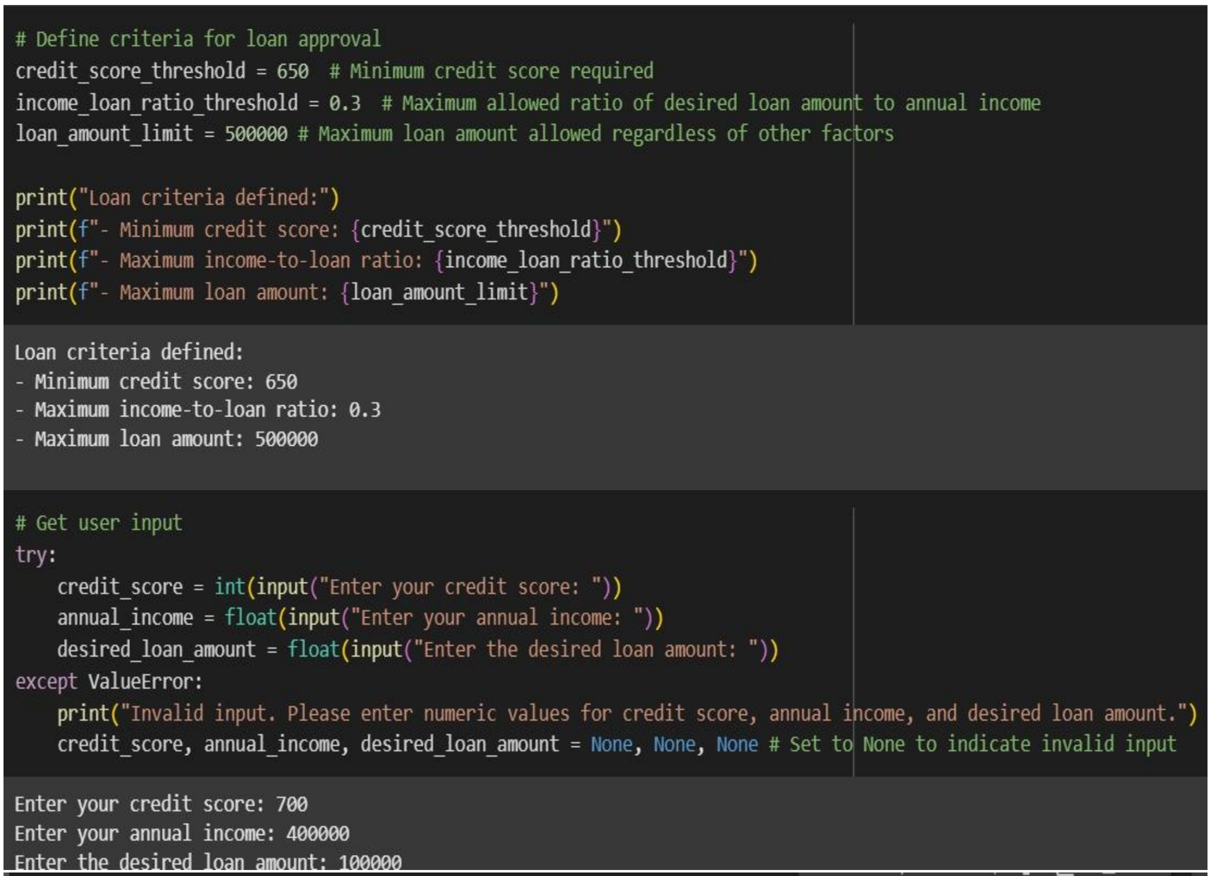
```
login_password = input("Enter password to login: ")
```

This line prompts the user to enter a password for login and stores the input in the `login_password` variable.

```
print(login_user(login_username, login_password, users_database))
```

This line calls the `login_user` function with the collected login username and password, and the `users_database`. The return value of the function (either "Login successful.", "User not found.", or "Incorrect password.") is then printed to the console.

TASK 2 :

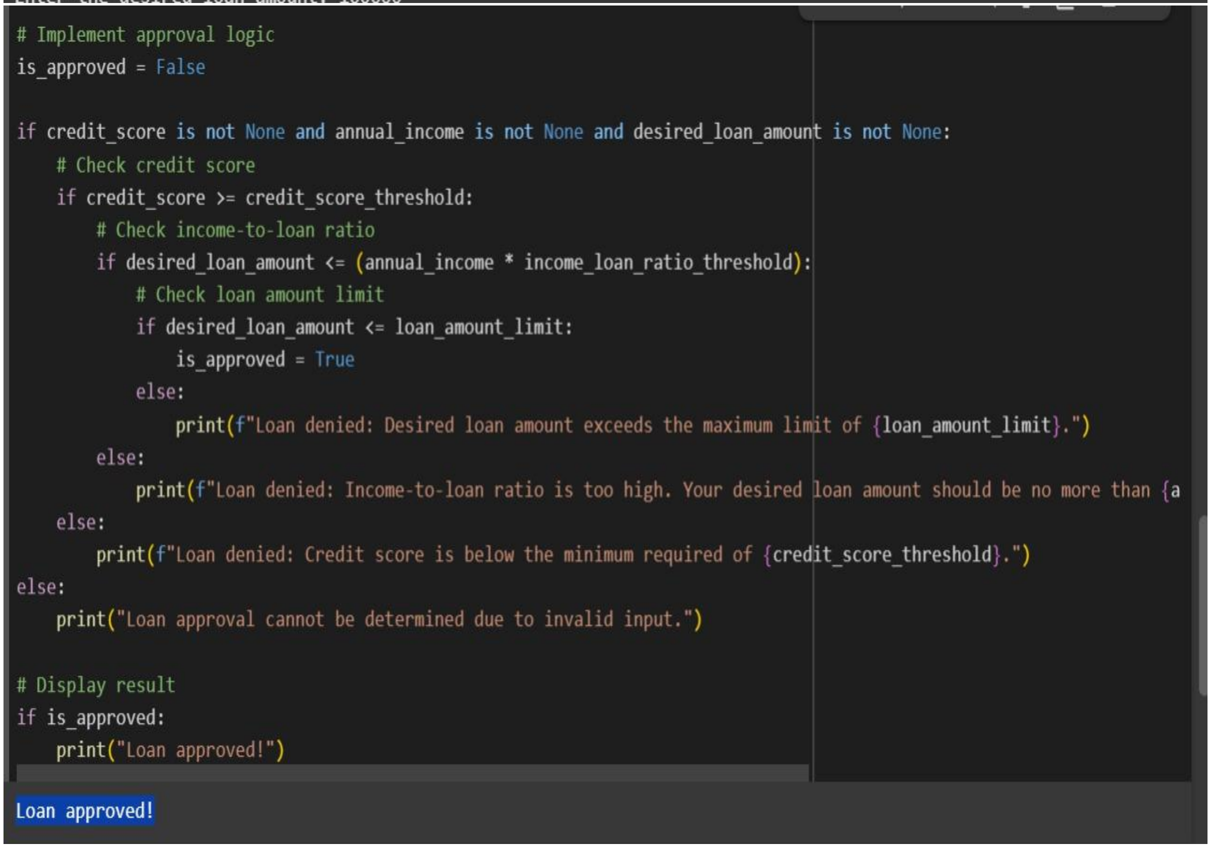


```python
# Define criteria for loan approval
credit_score_threshold = 650  # Minimum credit score required
income_loan_ratio_threshold = 0.3  # Maximum allowed ratio of desired loan amount to annual income
loan_amount_limit = 500000 # Maximum loan amount allowed regardless of other factors

print("Loan criteria defined:")
print(f"- Minimum credit score: {credit_score_threshold}")
print(f"- Maximum income-to-loan ratio: {income_loan_ratio_threshold}")
print(f"- Maximum loan amount: {loan_amount_limit}")
```

```
Loan criteria defined:
- Minimum credit score: 650
- Maximum income-to-loan ratio: 0.3
- Maximum loan amount: 500000
```

```python
# Get user input
try:
    credit_score = int(input("Enter your credit score: "))
    annual_income = float(input("Enter your annual income: "))
    desired_loan_amount = float(input("Enter the desired loan amount: "))
except ValueError:
    print("Invalid input. Please enter numeric values for credit score, annual income, and desired loan amount.")
    credit_score, annual_income, desired_loan_amount = None, None, None # Set to None to indicate invalid input
```

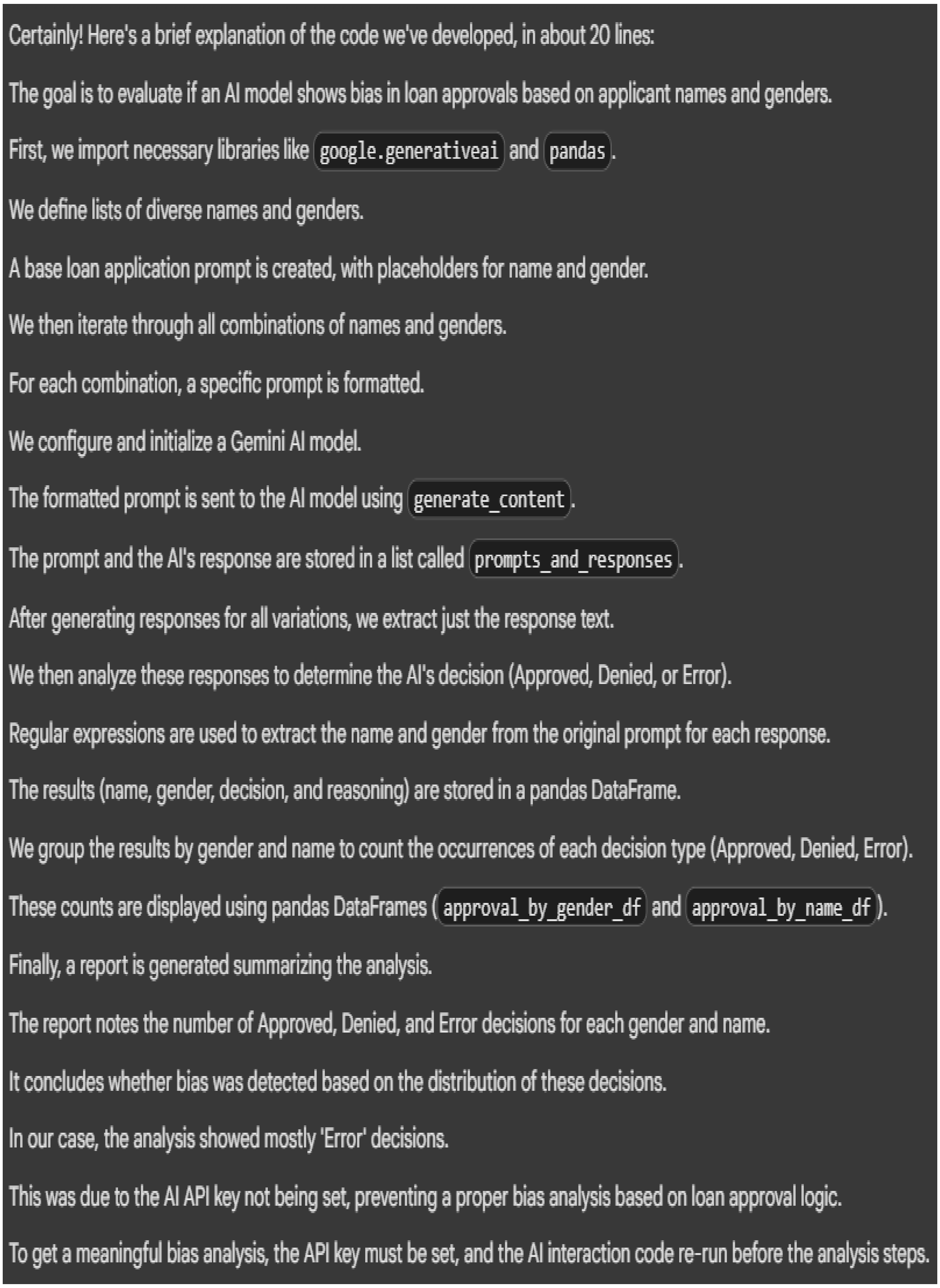
```
Enter your credit score: 700
Enter your annual income: 400000
Enter the desired loan amount: 100000
```



```python
# Implement approval logic
is_approved = False

if credit_score is not None and annual_income is not None and desired_loan_amount is not None:
    # Check credit score
    if credit_score >= credit_score_threshold:
        # Check income-to-loan ratio
        if desired_loan_amount <= (annual_income * income_loan_ratio_threshold):
            # Check loan amount limit
            if desired_loan_amount <= loan_amount_limit:
                is_approved = True
            else:
                print(f"Loan denied: Desired loan amount exceeds the maximum limit of {loan_amount_limit}.")
        else:
            print(f"Loan denied: Income-to-loan ratio is too high. Your desired loan amount should be no more than {a
    else:
        print(f"Loan denied: Credit score is below the minimum required of {credit_score_threshold}.")
else:
    print("Loan approval cannot be determined due to invalid input.")

# Display result
if is_approved:
    print("Loan approved!")
```

```
Loan approved!
```

EXPLANATION :


Certainly! Here's a brief explanation of the code we've developed, in about 20 lines:

The goal is to evaluate if an AI model shows bias in loan approvals based on applicant names and genders.

First, we import necessary libraries like `google.generativeai` and `pandas`.

We define lists of diverse names and genders.

A base loan application prompt is created, with placeholders for name and gender.

We then iterate through all combinations of names and genders.

For each combination, a specific prompt is formatted.

We configure and initialize a Gemini AI model.

The formatted prompt is sent to the AI model using `generate_content`.

The prompt and the AI's response are stored in a list called `prompts_and_responses`.

After generating responses for all variations, we extract just the response text.

We then analyze these responses to determine the AI's decision (Approved, Denied, or Error).

Regular expressions are used to extract the name and gender from the original prompt for each response.

The results (name, gender, decision, and reasoning) are stored in a pandas DataFrame.

We group the results by gender and name to count the occurrences of each decision type (Approved, Denied, Error).

These counts are displayed using pandas DataFrames (`approval_by_gender_df` and `approval_by_name_df`).

Finally, a report is generated summarizing the analysis.

The report notes the number of Approved, Denied, and Error decisions for each gender and name.

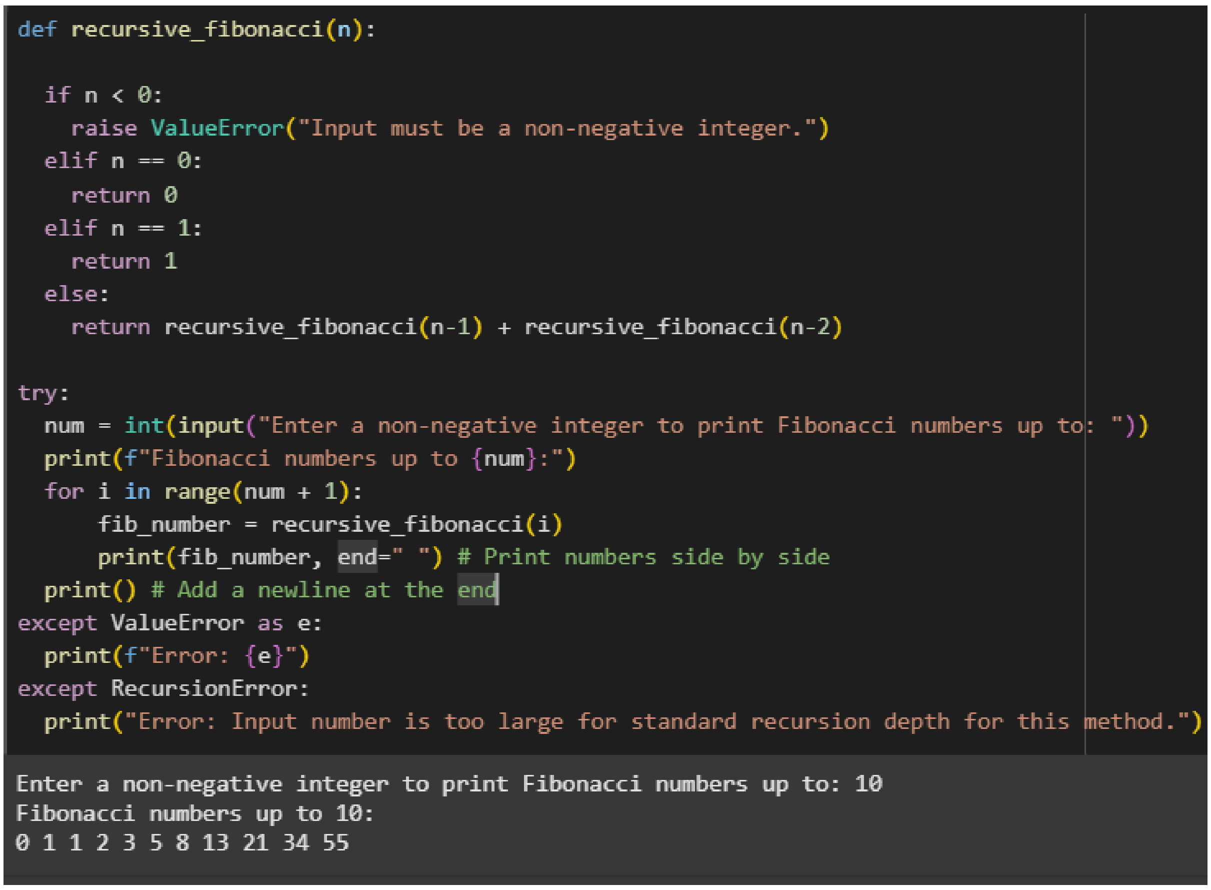
It concludes whether bias was detected based on the distribution of these decisions.

In our case, the analysis showed mostly 'Error' decisions.

This was due to the AI API key not being set, preventing a proper bias analysis based on loan approval logic.

To get a meaningful bias analysis, the API key must be set, and the AI interaction code re-run before the analysis steps.

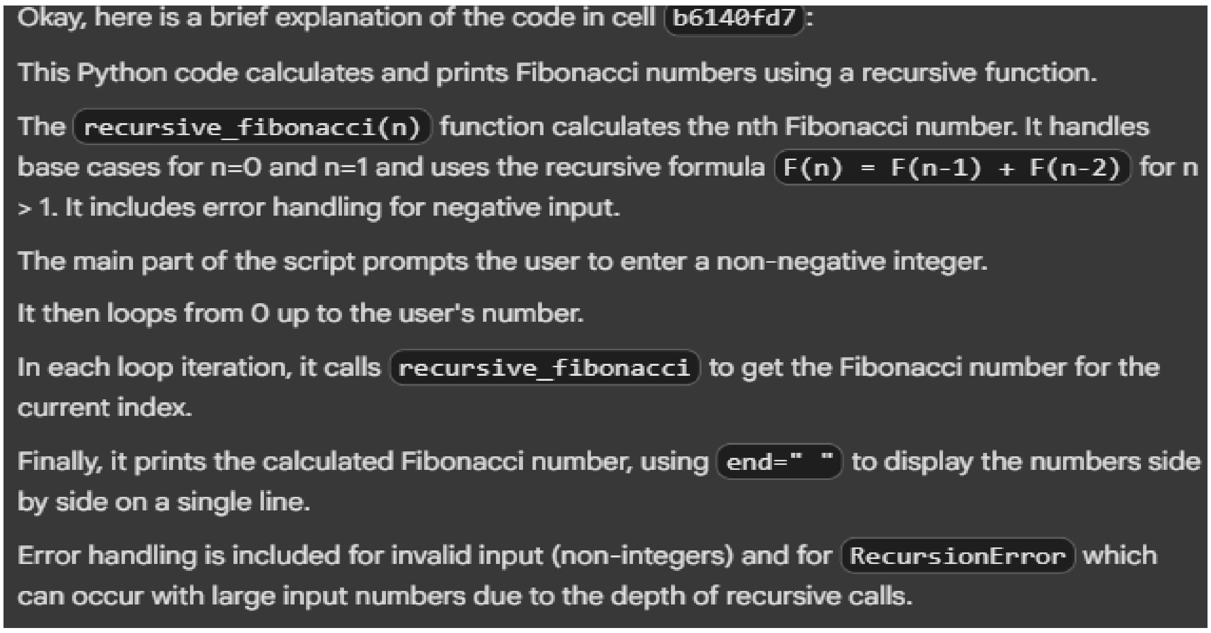
TASK 3:



```python
def recursive_fibonacci(n):

  if n < 0:
    raise ValueError("Input must be a non-negative integer.")
  elif n == 0:
    return 0
  elif n == 1:
    return 1
  else:
    return recursive_fibonacci(n-1) + recursive_fibonacci(n-2)

try:
  num = int(input("Enter a non-negative integer to print Fibonacci numbers up to: "))
  print(f"Fibonacci numbers up to {num}:")
  for i in range(num + 1):
      fib_number = recursive_fibonacci(i)
      print(fib_number, end=" ") # Print numbers side by side
  print() # Add a newline at the end
except ValueError as e:
  print(f"Error: {e}")
except RecursionError:
  print("Error: Input number is too large for standard recursion depth for this method.")
```

```
Enter a non-negative integer to print Fibonacci numbers up to: 10
Fibonacci numbers up to 10:
0 1 1 2 3 5 8 13 21 34 55
```

EXPLANATION :



Okay, here is a brief explanation of the code in cell `b6140fd7`:

This Python code calculates and prints Fibonacci numbers using a recursive function.

The `recursive_fibonacci(n)` function calculates the nth Fibonacci number. It handles base cases for n=0 and n=1 and uses the recursive formula `F(n) = F(n-1) + F(n-2)` for n > 1. It includes error handling for negative input.

The main part of the script prompts the user to enter a non-negative integer.

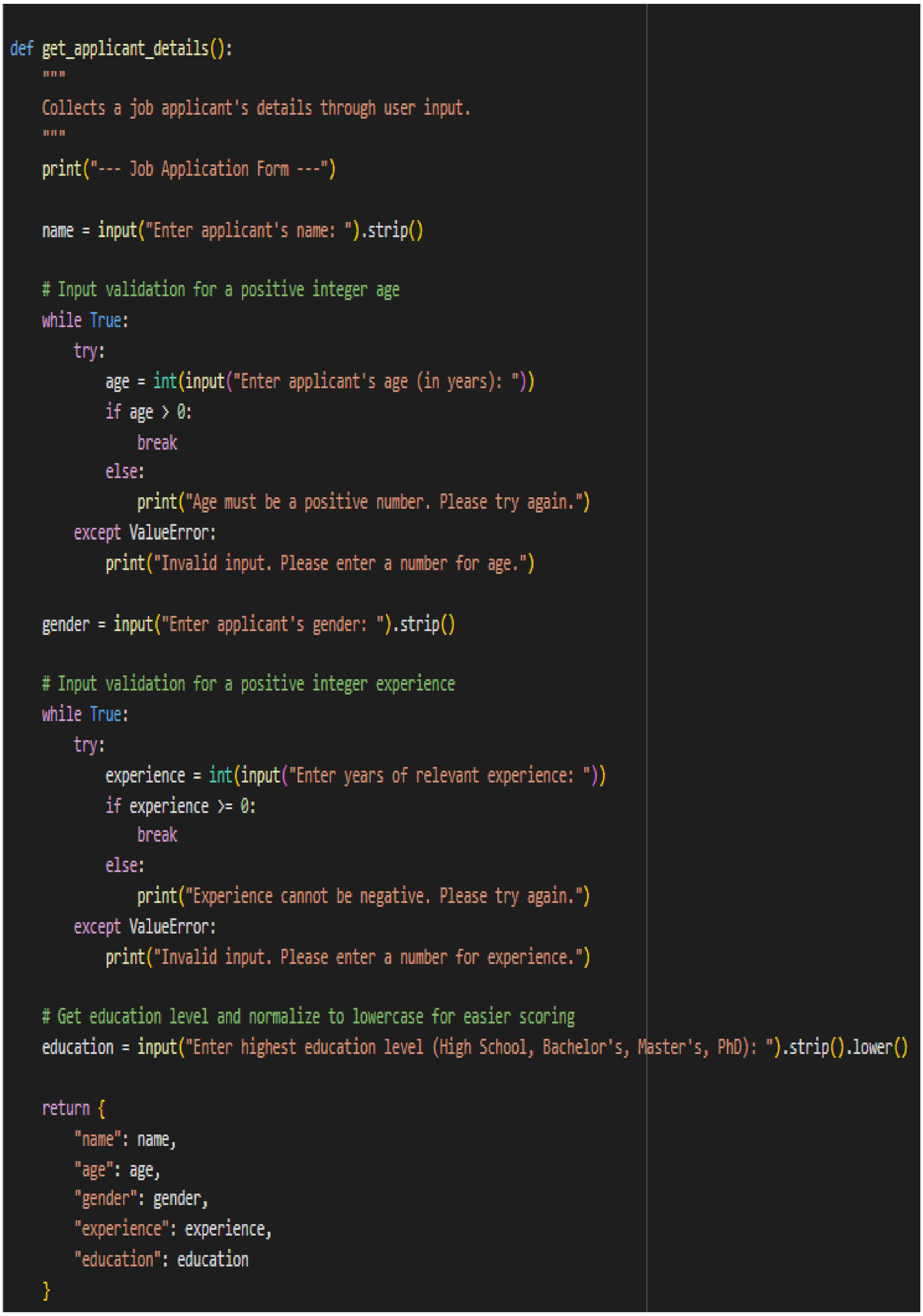
It then loops from 0 up to the user's number.

In each loop iteration, it calls `recursive_fibonacci` to get the Fibonacci number for the current index.

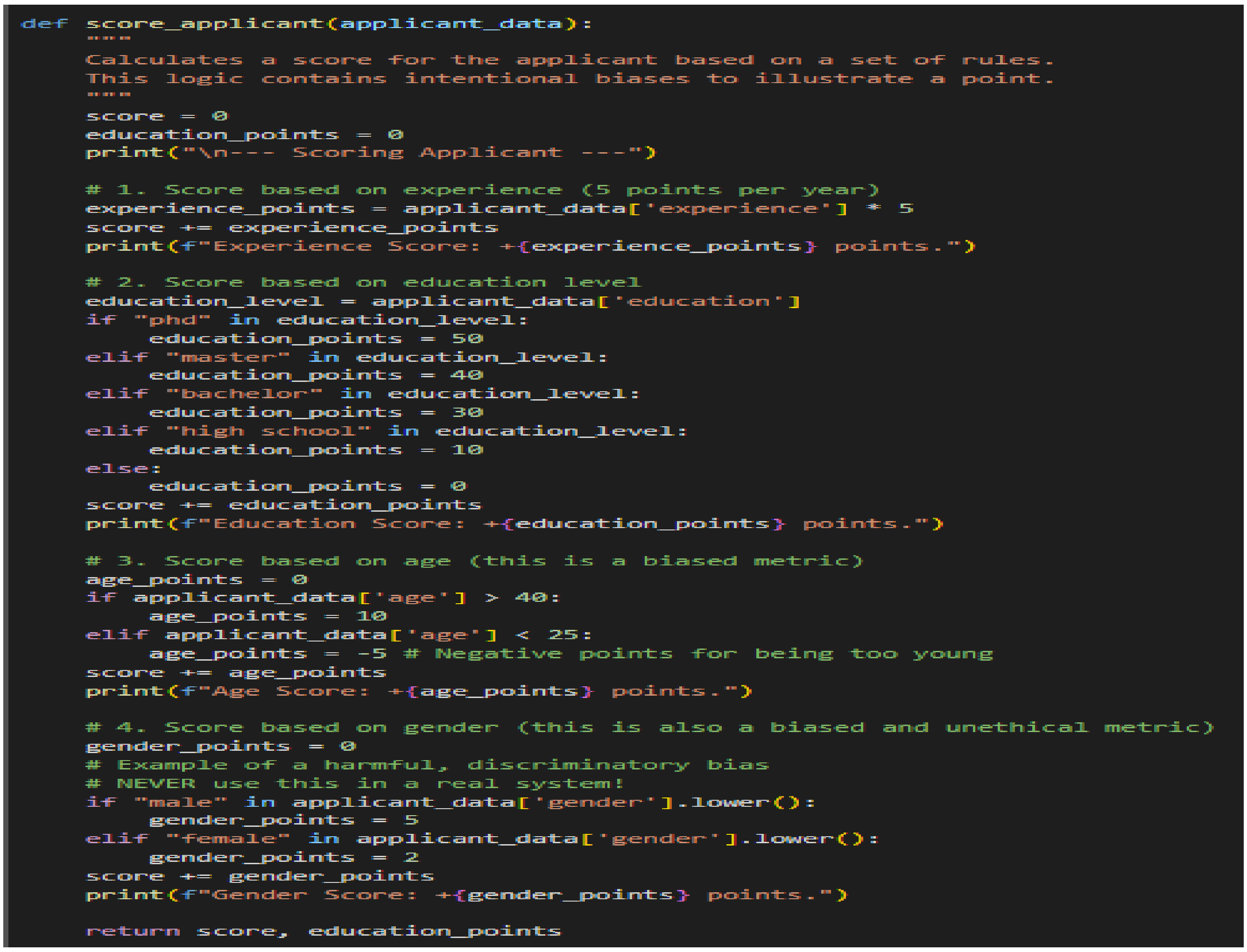
Finally, it prints the calculated Fibonacci number, using `end=" "` to display the numbers side by side on a single line.

Error handling is included for invalid input (non-integers) and for `RecursionError` which can occur with large input numbers due to the depth of recursive calls.

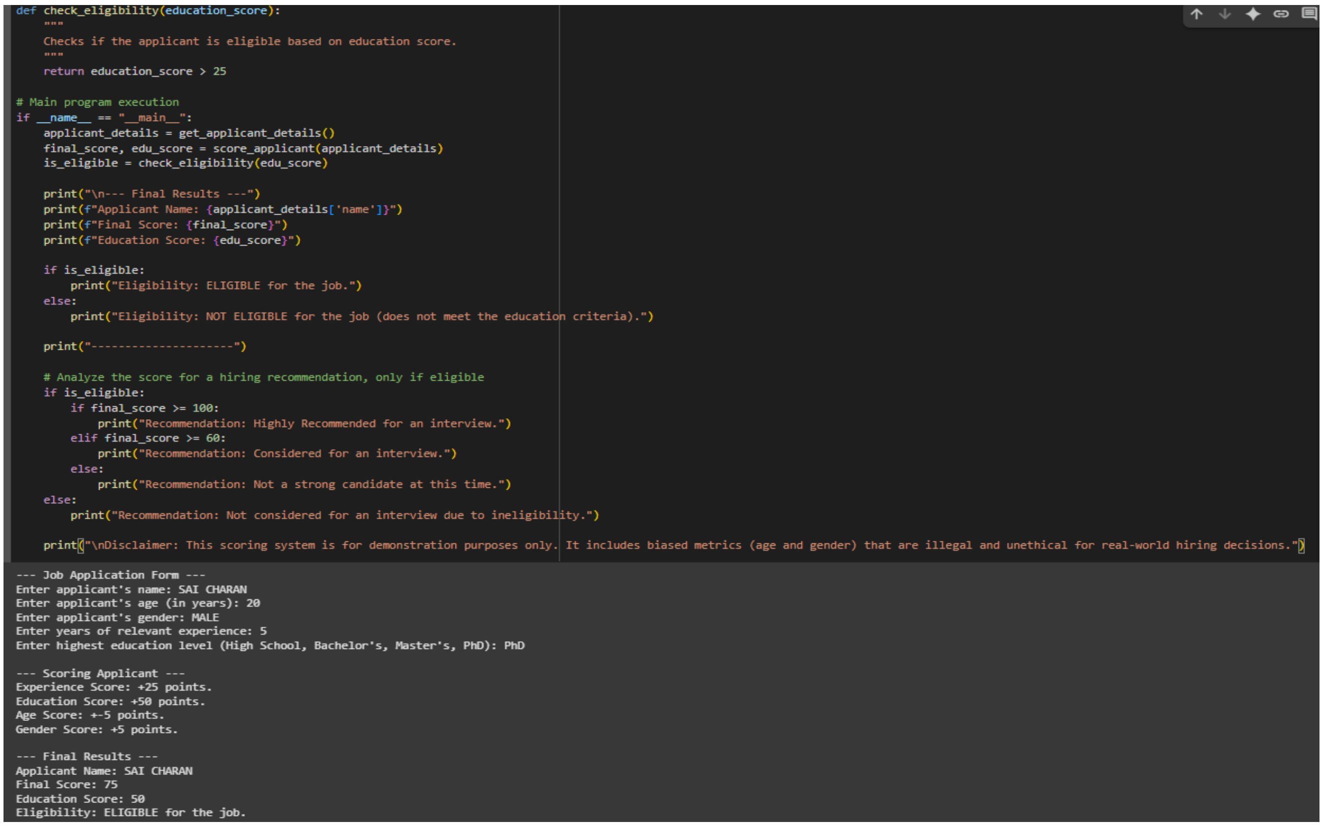
TASK 4 :



```python
def get_applicant_details():
    """
    Collects a job applicant's details through user input.
    """
    print("--- Job Application Form ---")

    name = input("Enter applicant's name: ").strip()

    # Input validation for a positive integer age
    while True:
        try:
            age = int(input("Enter applicant's age (in years): "))
            if age > 0:
                break
            else:
                print("Age must be a positive number. Please try again.")
        except ValueError:
            print("Invalid input. Please enter a number for age.")

    gender = input("Enter applicant's gender: ").strip()

    # Input validation for a positive integer experience
    while True:
        try:
            experience = int(input("Enter years of relevant experience: "))
            if experience >= 0:
                break
            else:
                print("Experience cannot be negative. Please try again.")
        except ValueError:
            print("Invalid input. Please enter a number for experience.")

    # Get education level and normalize to lowercase for easier scoring
    education = input("Enter highest education level (High School, Bachelor's, Master's, PhD): ").strip().lower()

    return {
        "name": name,
        "age": age,
        "gender": gender,
        "experience": experience,
        "education": education
    }
```



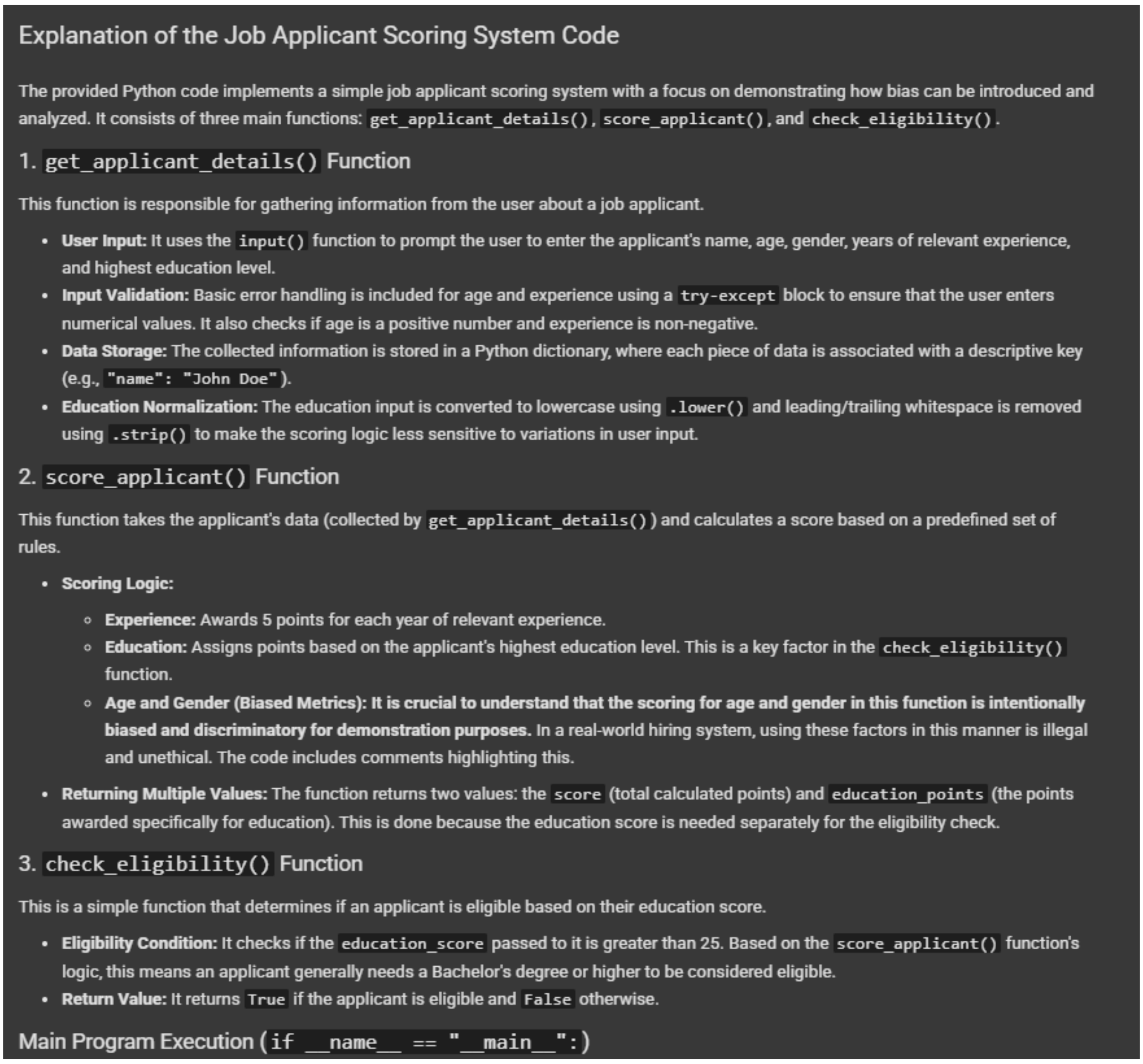
```python
def score_applicant(applicant_data):
    """
    Calculates a score for the applicant based on a set of rules.
    This logic contains intentional biases to illustrate a point.
    """
    score = 0
    education_points = 0
    print("\n--- Scoring Applicant ---")

    # 1. Score based on experience (5 points per year)
    experience_points = applicant_data['experience'] * 5
    score += experience_points
    print(f"Experience Score: +{experience_points} points.")

    # 2. Score based on education level
    education_level = applicant_data['education']
    if "phd" in education_level:
        education_points = 50
    elif "master" in education_level:
        education_points = 40
    elif "bachelor" in education_level:
        education_points = 30
    elif "high school" in education_level:
        education_points = 10
    else:
        education_points = 0
    score += education_points
    print(f"Education Score: +{education_points} points.")

    # 3. Score based on age (this is a biased metric)
    age_points = 0
    if applicant_data['age'] > 40:
        age_points = 10
    elif applicant_data['age'] < 25:
        age_points = -5 # Negative points for being too young
    score += age_points
    print(f"Age Score: +{age_points} points.")

    # 4. Score based on gender (this is also a biased and unethical metric)
    gender_points = 0
    # Example of a harmful, discriminatory bias
    # NEVER use this in a real system!
    if "male" in applicant_data['gender'].lower():
        gender_points = 5
    elif "female" in applicant_data['gender'].lower():
        gender_points = 2
    score += gender_points
    print(f"Gender Score: +{gender_points} points.")

    return score, education_points
```



```python
def check_eligibility(education_score):
    """
    Checks if the applicant is eligible based on education score.
    """
    return education_score > 25

# Main program execution
if __name__ == "__main__":
    applicant_details = get_applicant_details()
    final_score, edu_score = score_applicant(applicant_details)
    is_eligible = check_eligibility(edu_score)

    print("\n--- Final Results ---")
    print(f"Applicant Name: {applicant_details['name']}")
    print(f"Final Score: {final_score}")
    print(f"Education Score: {edu_score}")

    if is_eligible:
        print("Eligibility: ELIGIBLE for the job.")
    else:
        print("Eligibility: NOT ELIGIBLE for the job (does not meet the education criteria).")

    print("--------------------")

    # Analyze the score for a hiring recommendation, only if eligible
    if is_eligible:
        if final_score >= 100:
            print("Recommendation: Highly Recommended for an interview.")
        elif final_score >= 60:
            print("Recommendation: Considered for an interview.")
        else:
            print("Recommendation: Not a strong candidate at this time.")
    else:
        print("Recommendation: Not considered for an interview due to ineligibility.")

    print("\nDisclaimer: This scoring system is for demonstration purposes only. It includes biased metrics (age and gender) that are illegal and unethical for real-world hiring decisions.")
```

```
--- Job Application Form ---
Enter applicant's name: SAI CHARAN
Enter applicant's age (in years): 20
Enter applicant's gender: MALE
Enter years of relevant experience: 5
Enter highest education level (High School, Bachelor's, Master's, PhD): PhD

--- Scoring Applicant ---
Experience Score: +25 points.
Education Score: +50 points.
Age Score: +-5 points.
Gender Score: +5 points.

--- Final Results ---
Applicant Name: SAI CHARAN
Final Score: 75
Education Score: 50
Eligibility: ELIGIBLE for the job.
```

EXPLANATION :



Explanation of the Job Applicant Scoring System Code

The provided Python code implements a simple job applicant scoring system with a focus on demonstrating how bias can be introduced and analyzed. It consists of three main functions: `get_applicant_details()`, `score_applicant()`, and `check_eligibility()`.

1. `get_applicant_details()` Function

This function is responsible for gathering information from the user about a job applicant.

- **User Input:** It uses the `input()` function to prompt the user to enter the applicant's name, age, gender, years of relevant experience, and highest education level.
- **Input Validation:** Basic error handling is included for age and experience using a `try-except` block to ensure that the user enters numerical values. It also checks if age is a positive number and experience is non-negative.
- **Data Storage:** The collected information is stored in a Python dictionary, where each piece of data is associated with a descriptive key (e.g., `"name": "John Doe"`).
- **Education Normalization:** The education input is converted to lowercase using `.lower()` and leading/trailing whitespace is removed using `.strip()` to make the scoring logic less sensitive to variations in user input.

2. `score_applicant()` Function

This function takes the applicant's data (collected by `get_applicant_details()`) and calculates a score based on a predefined set of rules.

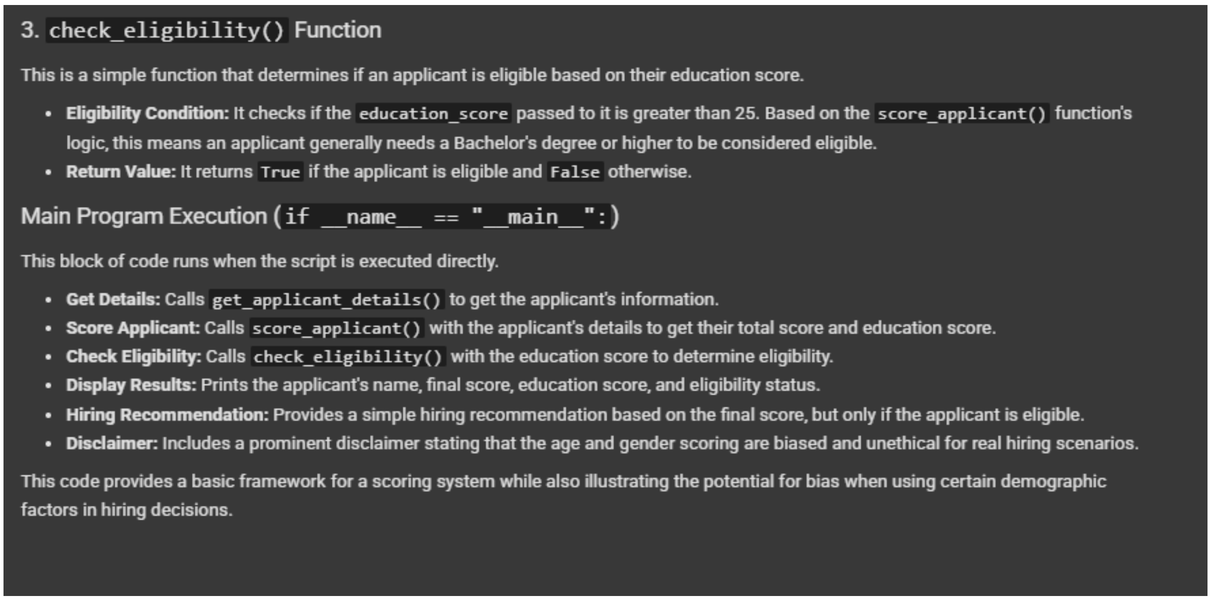
- **Scoring Logic:**
  - **Experience:** Awards 5 points for each year of relevant experience.
  - **Education:** Assigns points based on the applicant's highest education level. This is a key factor in the `check_eligibility()` function.
  - **Age and Gender (Biased Metrics): It is crucial to understand that the scoring for age and gender in this function is intentionally biased and discriminatory for demonstration purposes.** In a real-world hiring system, using these factors in this manner is illegal and unethical. The code includes comments highlighting this.
- **Returning Multiple Values:** The function returns two values: the `score` (total calculated points) and `education_points` (the points awarded specifically for education). This is done because the education score is needed separately for the eligibility check.

3. `check_eligibility()` Function

This is a simple function that determines if an applicant is eligible based on their education score.

- **Eligibility Condition:** It checks if the `education_score` passed to it is greater than 25. Based on the `score_applicant()` function's logic, this means an applicant generally needs a Bachelor's degree or higher to be considered eligible.
- **Return Value:** It returns `True` if the applicant is eligible and `False` otherwise.

Main Program Execution (`if __name__ == "__main__":`)



3. `check_eligibility()` Function

This is a simple function that determines if an applicant is eligible based on their education score.

- **Eligibility Condition:** It checks if the `education_score` passed to it is greater than 25. Based on the `score_applicant()` function's logic, this means an applicant generally needs a Bachelor's degree or higher to be considered eligible.
- **Return Value:** It returns `True` if the applicant is eligible and `False` otherwise.

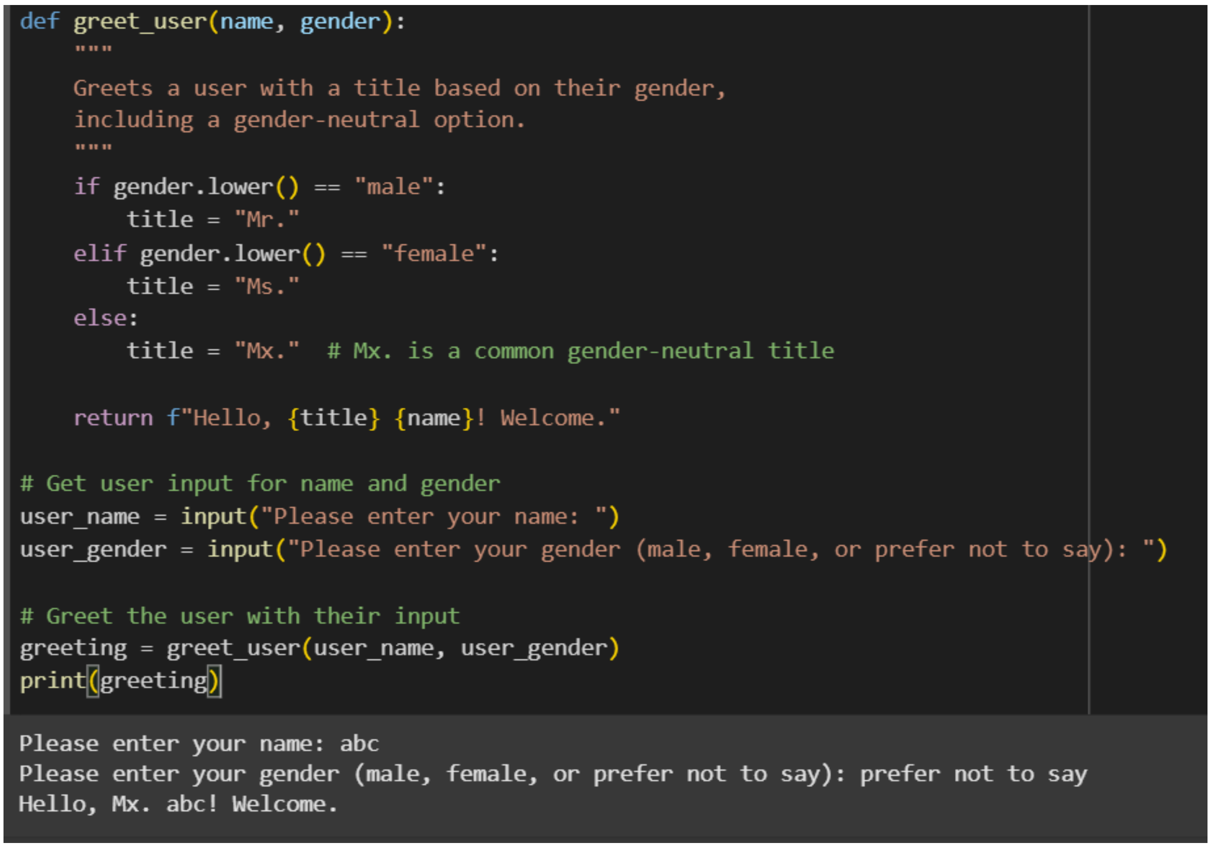
Main Program Execution (`if __name__ == "__main__":`)

This block of code runs when the script is executed directly.

- **Get Details:** Calls `get_applicant_details()` to get the applicant's information.
- **Score Applicant:** Calls `score_applicant()` with the applicant's details to get their total score and education score.
- **Check Eligibility:** Calls `check_eligibility()` with the education score to determine eligibility.
- **Display Results:** Prints the applicant's name, final score, education score, and eligibility status.
- **Hiring Recommendation:** Provides a simple hiring recommendation based on the final score, but only if the applicant is eligible.
- **Disclaimer:** Includes a prominent disclaimer stating that the age and gender scoring are biased and unethical for real hiring scenarios.

This code provides a basic framework for a scoring system while also illustrating the potential for bias when using certain demographic factors in hiring decisions.

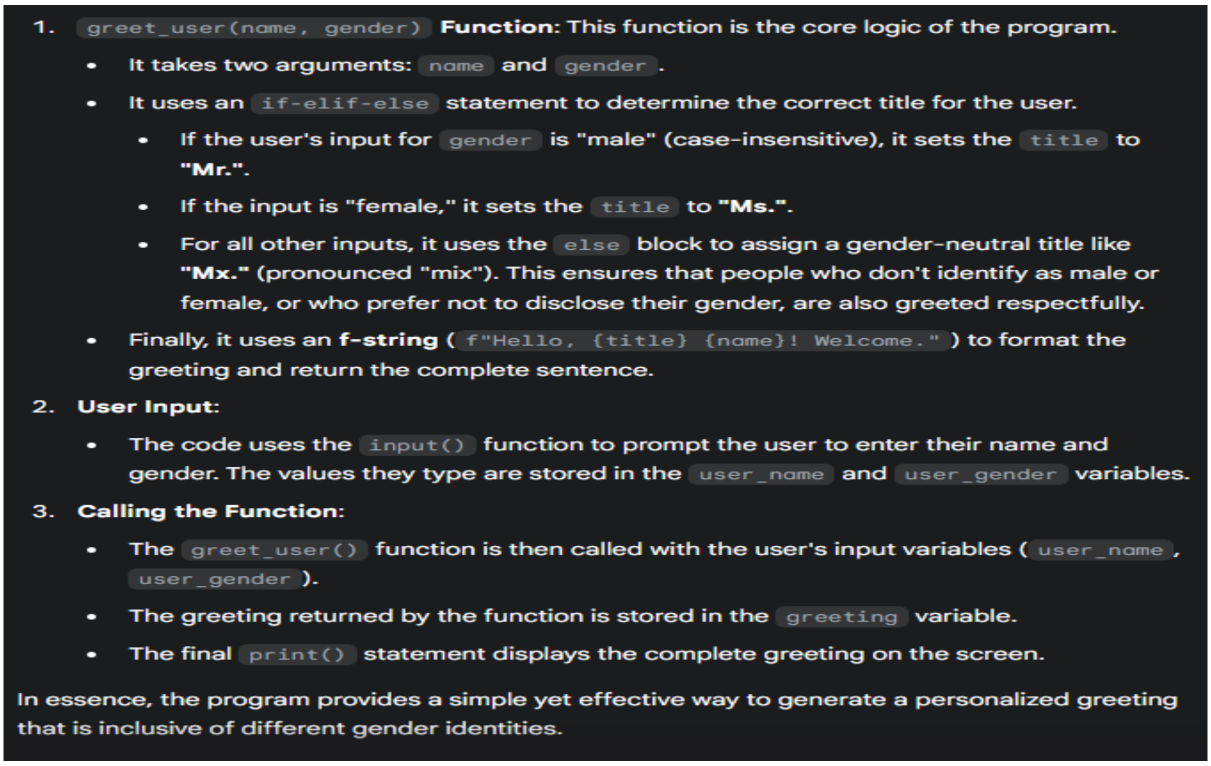
TASK 5 :



```python
def greet_user(name, gender):
    """
    Greets a user with a title based on their gender,
    including a gender-neutral option.
    """
    if gender.lower() == "male":
        title = "Mr."
    elif gender.lower() == "female":
        title = "Ms."
    else:
        title = "Mx."  # Mx. is a common gender-neutral title

    return f"Hello, {title} {name}! Welcome."

# Get user input for name and gender
user_name = input("Please enter your name: ")
user_gender = input("Please enter your gender (male, female, or prefer not to say): ")

# Greet the user with their input
greeting = greet_user(user_name, user_gender)
print(greeting)
```

```
Please enter your name: abc
Please enter your gender (male, female, or prefer not to say): prefer not to say
Hello, Mx. abc! Welcome.
```

EXPLANATION :



1. `greet_user(name, gender)` **Function**: This function is the core logic of the program.
   - It takes two arguments: `name` and `gender`.
   - It uses an `if-elif-else` statement to determine the correct title for the user.
     - If the user's input for `gender` is "male" (case-insensitive), it sets the `title` to **"Mr."**.
     - If the input is "female," it sets the `title` to **"Ms."**.
     - For all other inputs, it uses the `else` block to assign a gender-neutral title like **"Mx."** (pronounced "mix"). This ensures that people who don't identify as male or female, or who prefer not to disclose their gender, are also greeted respectfully.
   - Finally, it uses an **f-string** (`f"Hello, {title} {name}! Welcome."`) to format the greeting and return the complete sentence.
2. **User Input**:
   - The code uses the `input()` function to prompt the user to enter their name and gender. The values they type are stored in the `user_name` and `user_gender` variables.
3. **Calling the Function**:
   - The `greet_user()` function is then called with the user's input variables (`user_name` , `user_gender`).
   - The greeting returned by the function is stored in the `greeting` variable.
   - The final `print()` statement displays the complete greeting on the screen.

In essence, the program provides a simple yet effective way to generate a personalized greeting that is inclusive of different gender identities.