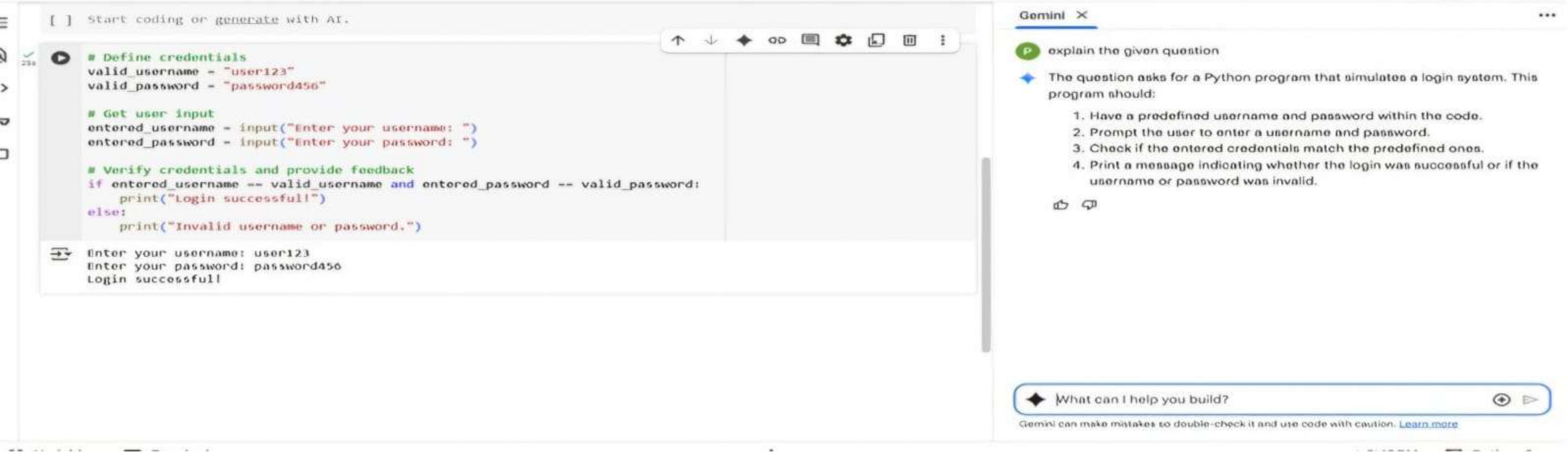
ASSIGNMENT 5

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.


[ ] Start coding or generate with AI.
Gemini
What can I help you build?
Gemini can make mistakes so double-check it and use code with caution. Learn more

```python
# Define credentials
valid_username = "user123"
valid_password = "password456"

# Get user input
entered_username = input("Enter your username: ")
entered_password = input("Enter your password: ")

# Verify credentials and provide feedback
if entered_username == valid_username and entered_password == valid_password:
    print("Login successful!")
else:
    print("Invalid username or password.")
```

```
Enter your username: user123
Enter your password: password456
Login successful!
```

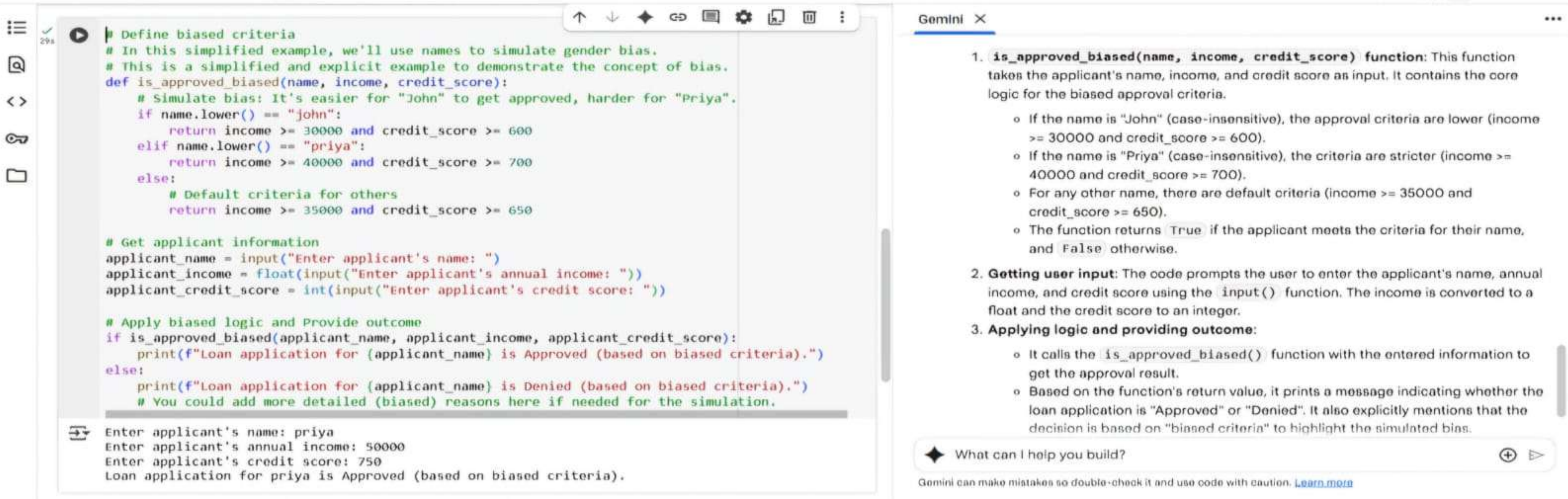
explain the given question

The question asks for a Python program that simulates a login system. This program should:

1. Have a predefined username and password within the code.
2. Prompt the user to enter a username and password.
3. Check if the entered credentials match the predefined ones.
4. Print a message indicating whether the login was successful or if the username or password was invalid.

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.



```
# Define biased criteria
# In this simplified example, we'll use names to simulate gender bias.
# This is a simplified and explicit example to demonstrate the concept of bias.
def is_approved_biased(name, income, credit_score):
    # Simulate bias: It's easier for "John" to get approved, harder for "Priya".
    if name.lower() == "john":
        return income >= 30000 and credit_score >= 600
    elif name.lower() == "priya":
        return income >= 40000 and credit_score >= 700
    else:
        # Default criteria for others
        return income >= 35000 and credit_score >= 650

# Get applicant information
applicant_name = input("Enter applicant's name: ")
applicant_income = float(input("Enter applicant's annual income: "))
applicant_credit_score = int(input("Enter applicant's credit score: "))

# Apply biased logic and Provide outcome
if is_approved_biased(applicant_name, applicant_income, applicant_credit_score):
    print(f"Loan application for {applicant_name} is Approved (based on biased criteria).")
else:
    print(f"Loan application for {applicant_name} is Denied (based on biased criteria).")
    # You could add more detailed (biased) reasons here if needed for the simulation.
```

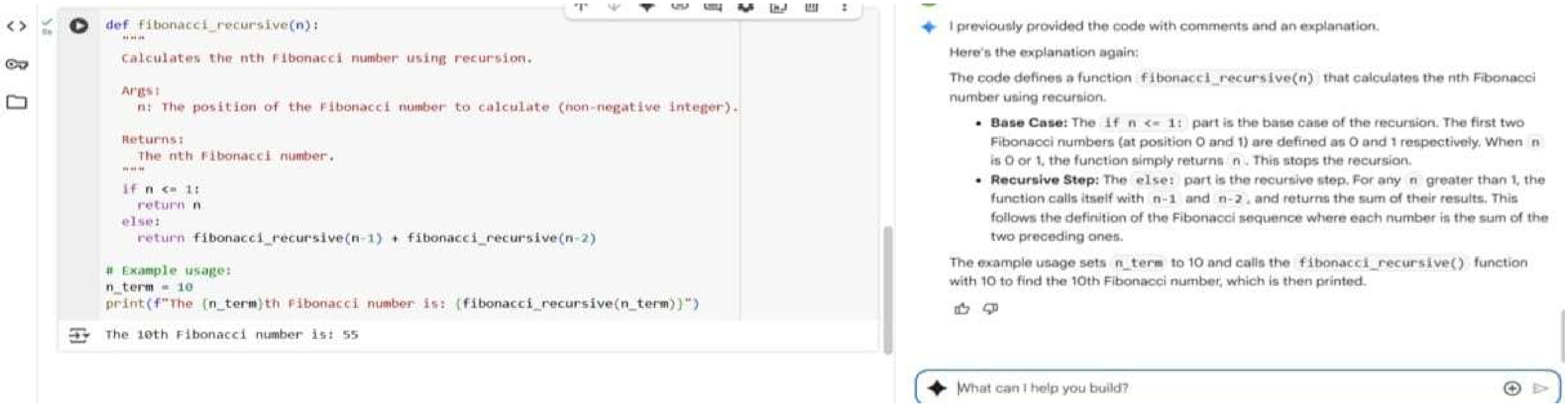
```
Enter applicant's name: priya
Enter applicant's annual income: 50000
Enter applicant's credit score: 750
Loan application for priya is Approved (based on biased criteria).
```

1. **`is_approved_biased(name, income, credit_score)` function**: This function takes the applicant's name, income, and credit score as input. It contains the core logic for the biased approval criteria.
    - If the name is "John" (case-insensitive), the approval criteria are lower (income >= 30000 and credit_score >= 600).
    - If the name is "Priya" (case-insensitive), the criteria are stricter (income >= 40000 and credit_score >= 700).
    - For any other name, there are default criteria (income >= 35000 and credit_score >= 650).
    - The function returns `True` if the applicant meets the criteria for their name, and `False` otherwise.
2. **Getting user input**: The code prompts the user to enter the applicant's name, annual income, and credit score using the `input()` function. The income is converted to a float and the credit score to an integer.
3. **Applying logic and providing outcome**:
    - It calls the `is_approved_biased()` function with the entered information to get the approval result.
    - Based on the function's return value, it prints a message indicating whether the loan application is "Approved" or "Denied". It also explicitly mentions that the decision is based on "biased criteria" to highlight the simulated bias.

Gemini can make mistakes so double-check it and use code with caution. Learn more

Task-3:

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



```
def fibonacci_recursive(n):
  """
  Calculates the nth Fibonacci number using recursion.

  Args:
    n: The position of the Fibonacci number to calculate (non-negative integer).

  Returns:
    The nth Fibonacci number.
  """
  if n <= 1:
    return n
  else:
    return fibonacci_recursive(n-1) + fibonacci_recursive(n-2)

# Example usage:
n_term = 10
print(f"The {n_term}th Fibonacci number is: {fibonacci_recursive(n_term)}")
```

```
The 10th Fibonacci number is: 55
```

I previously provided the code with comments and an explanation.

Here's the explanation again:

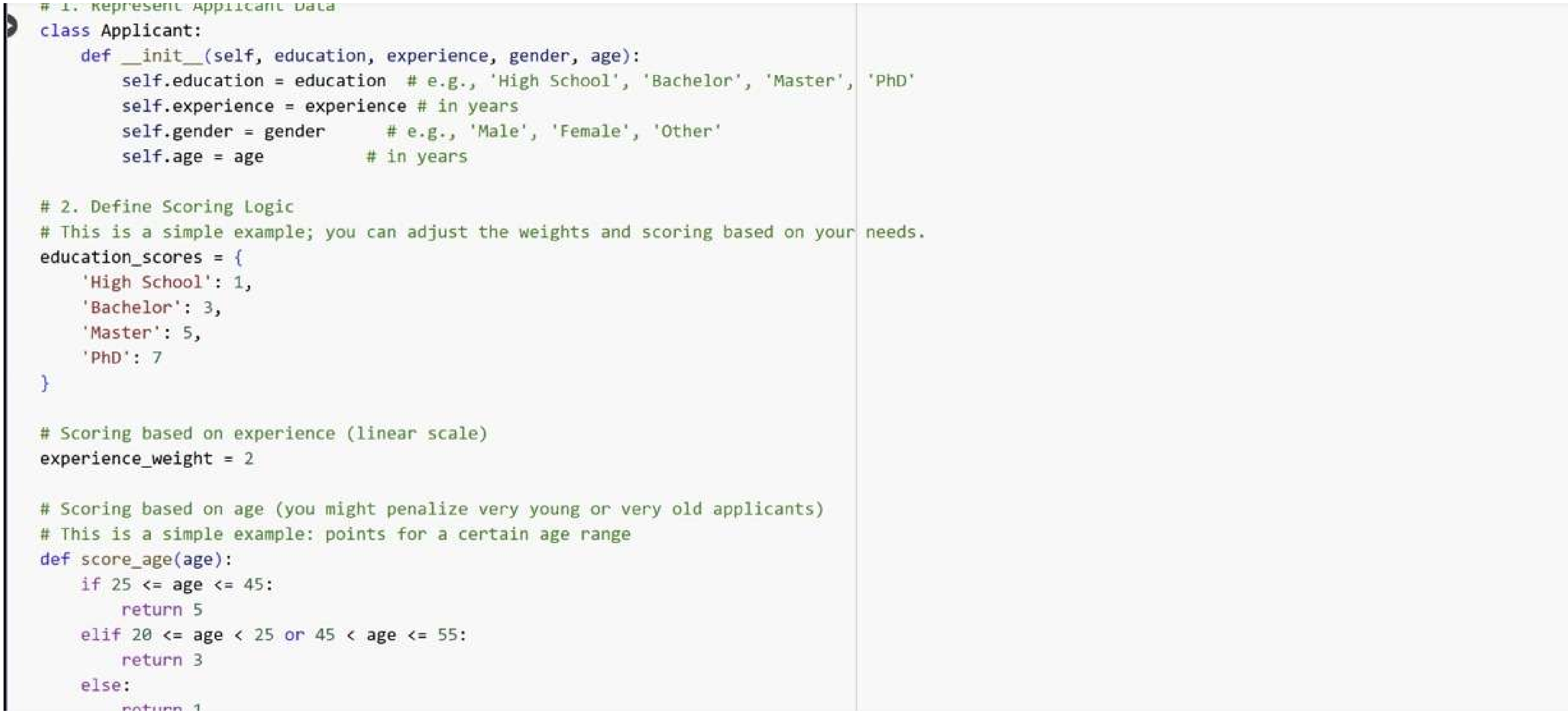
The code defines a function `fibonacci_recursive(n)` that calculates the nth Fibonacci number using recursion.

- **Base Case:** The `if n <= 1:` part is the base case of the recursion. The first two Fibonacci numbers (at position 0 and 1) are defined as 0 and 1 respectively. When `n` is 0 or 1, the function simply returns `n`. This stops the recursion.
- **Recursive Step:** The `else:` part is the recursive step. For any `n` greater than 1, the function calls itself with `n-1` and `n-2`, and returns the sum of their results. This follows the definition of the Fibonacci sequence where each number is the sum of the two preceding ones.

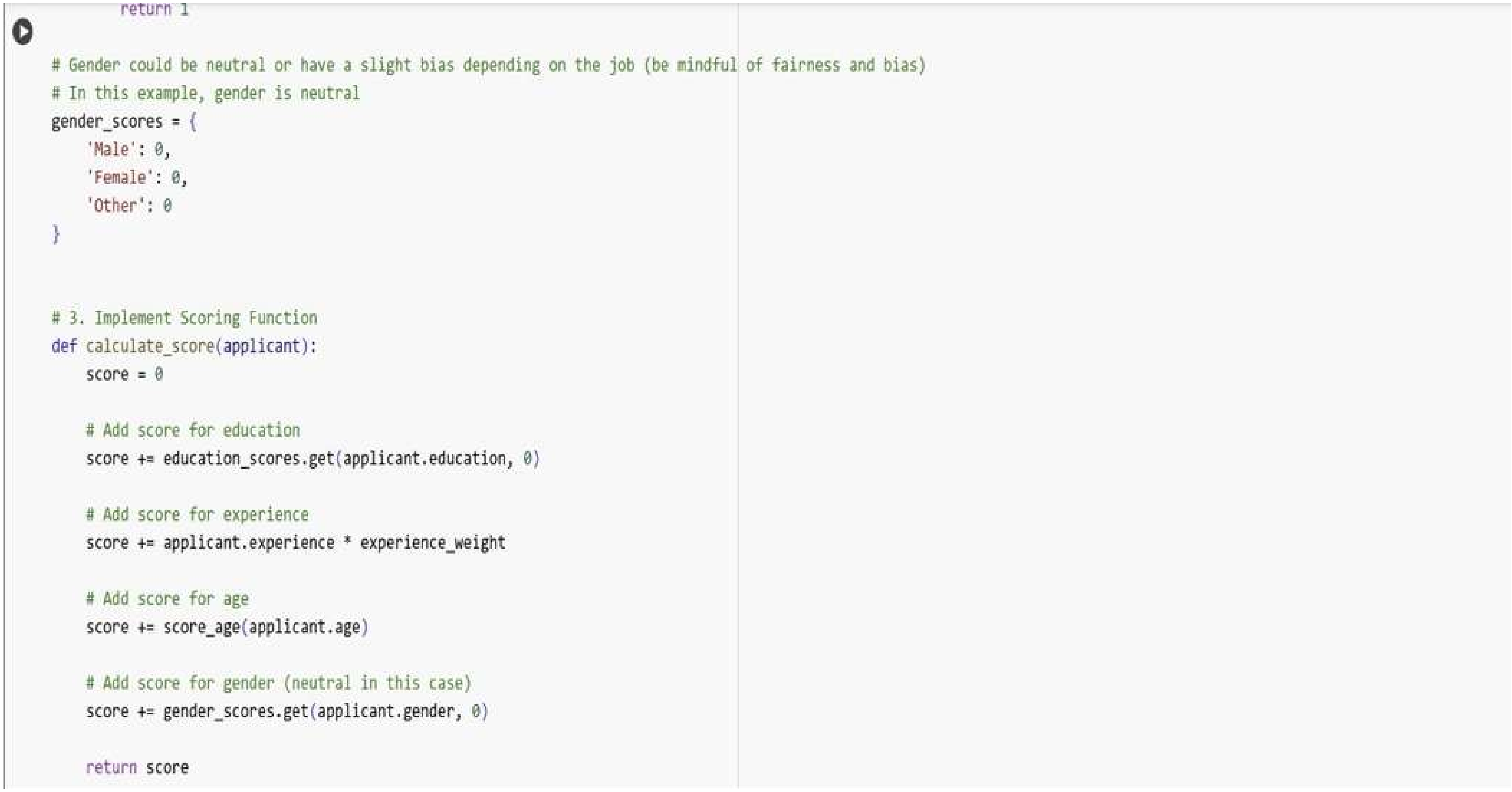
The example usage sets `n_term` to 10 and calls the `fibonacci_recursive()` function with 10 to find the 10th Fibonacci number, which is then printed.

Task-4:

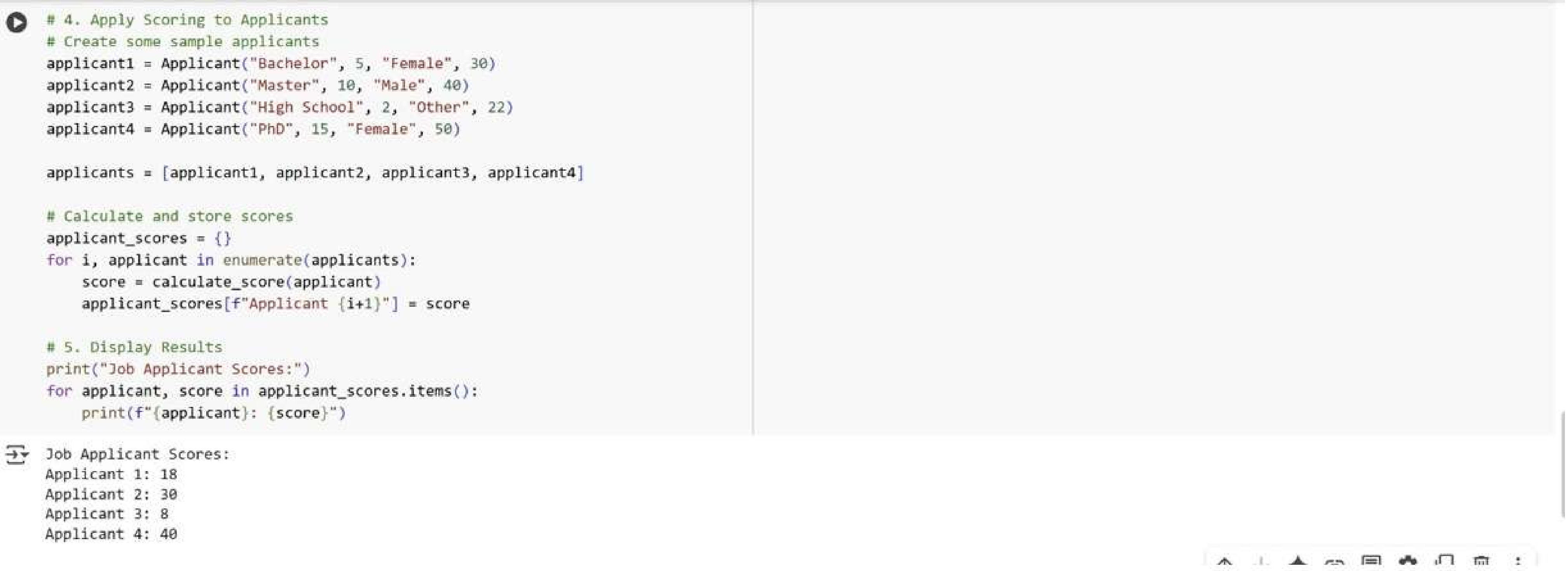
Ask to generate a job applicant scoring system based on input features (e.g., education, experience, gender, age). Analyze the scoring logic for bias or unfair weightings.



```
# 1. Represent Applicant Data
class Applicant:
    def __init__(self, education, experience, gender, age):
        self.education = education  # e.g., 'High School', 'Bachelor', 'Master', 'PhD'
        self.experience = experience # in years
        self.gender = gender      # e.g., 'Male', 'Female', 'Other'
        self.age = age          # in years

# 2. Define Scoring Logic
# This is a simple example; you can adjust the weights and scoring based on your needs.
education_scores = {
    'High School': 1,
    'Bachelor': 3,
    'Master': 5,
    'PhD': 7
}

# Scoring based on experience (linear scale)
experience_weight = 2

# Scoring based on age (you might penalize very young or very old applicants)
# This is a simple example: points for a certain age range
def score_age(age):
    if 25 <= age <= 45:
        return 5
    elif 20 <= age < 25 or 45 < age <= 55:
        return 3
    else:
        return 1
```



```
        return 1

# Gender could be neutral or have a slight bias depending on the job (be mindful of fairness and bias)
# In this example, gender is neutral
gender_scores = {
    'Male': 0,
    'Female': 0,
    'Other': 0
}


# 3. Implement Scoring Function
def calculate_score(applicant):
    score = 0

    # Add score for education
    score += education_scores.get(applicant.education, 0)

    # Add score for experience
    score += applicant.experience * experience_weight

    # Add score for age
    score += score_age(applicant.age)

    # Add score for gender (neutral in this case)
    score += gender_scores.get(applicant.gender, 0)

    return score
```

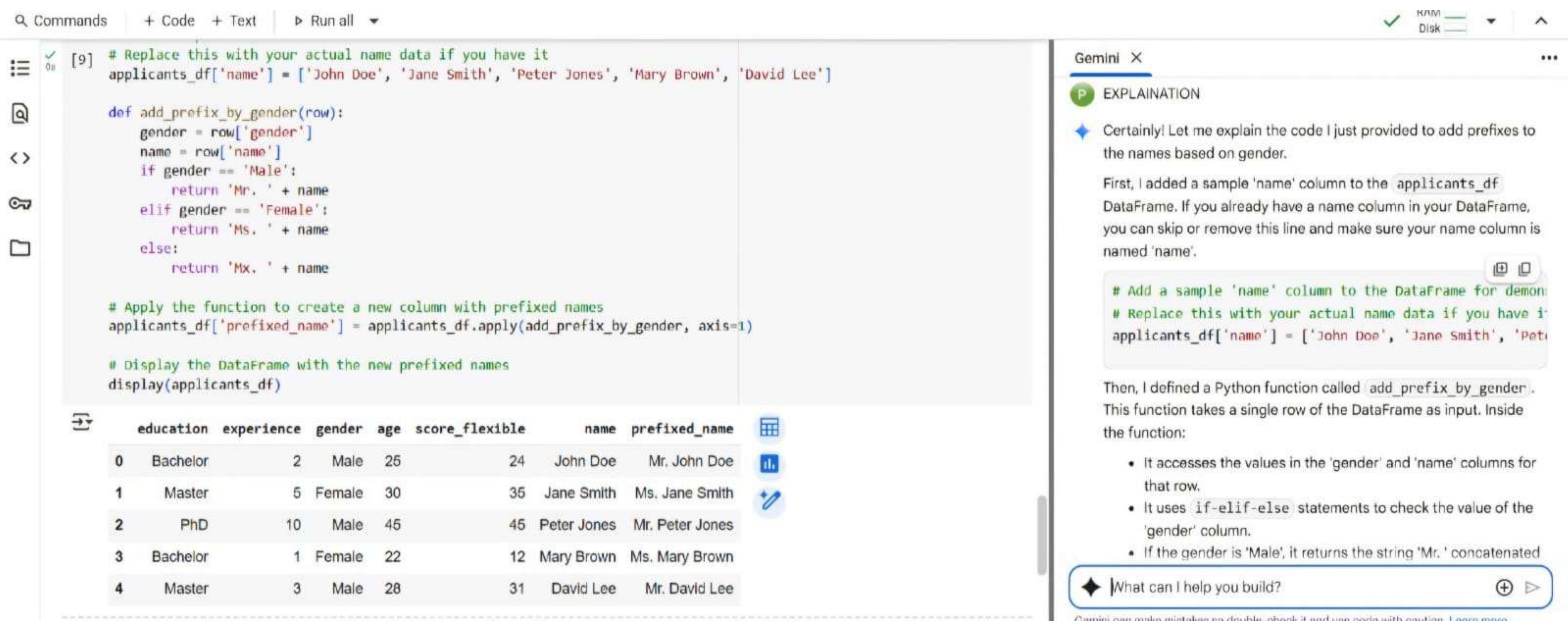


```
# 4. Apply Scoring to Applicants
# Create some sample applicants
applicant1 = Applicant("Bachelor", 5, "Female", 30)
applicant2 = Applicant("Master", 10, "Male", 40)
applicant3 = Applicant("High School", 2, "Other", 22)
applicant4 = Applicant("PhD", 15, "Female", 50)

applicants = [applicant1, applicant2, applicant3, applicant4]

# Calculate and store scores
applicant_scores = {}
for i, applicant in enumerate(applicants):
    score = calculate_score(applicant)
    applicant_scores[f"Applicant {i+1}"] = score

# 5. Display Results
print("Job Applicant Scores:")
for applicant, score in applicant_scores.items():
    print(f"{applicant}: {score}")
```

```
Job Applicant Scores:
Applicant 1: 18
Applicant 2: 30
Applicant 3: 8
Applicant 4: 40
```

Task-5:

write a program in python generate before the name for male is a MR for female is a MS for neutral is MX



```python
# Replace this with your actual name data if you have it
applicants_df['name'] = ['John Doe', 'Jane Smith', 'Peter Jones', 'Mary Brown', 'David Lee']

def add_prefix_by_gender(row):
    gender = row['gender']
    name = row['name']
    if gender == 'Male':
        return 'Mr. ' + name
    elif gender == 'Female':
        return 'Ms. ' + name
    else:
        return 'Mx. ' + name

# Apply the function to create a new column with prefixed names
applicants_df['prefixed_name'] = applicants_df.apply(add_prefix_by_gender, axis=1)

# Display the DataFrame with the new prefixed names
display(applicants_df)
```

	education	experience	gender	age	score_flexible	name	prefixed_name
0	Bachelor	2	Male	25	24	John Doe	Mr. John Doe
1	Master	5	Female	30	35	Jane Smith	Ms. Jane Smith
2	PhD	10	Male	45	45	Peter Jones	Mr. Peter Jones
3	Bachelor	1	Female	22	12	Mary Brown	Ms. Mary Brown
4	Master	3	Male	28	31	David Lee	Mr. David Lee

EXPLAINATION

Certainly! Let me explain the code I just provided to add prefixes to the names based on gender.

First, I added a sample 'name' column to the `applicants_df` DataFrame. If you already have a name column in your DataFrame, you can skip or remove this line and make sure your name column is named 'name'.

```python
# Add a sample 'name' column to the DataFrame for demon
# Replace this with your actual name data if you have i
applicants_df['name'] = ['John Doe', 'Jane Smith', 'Pet
```

Then, I defined a Python function called `add_prefix_by_gender`. This function takes a single row of the DataFrame as input. Inside the function:

- It accesses the values in the 'gender' and 'name' columns for that row.
- It uses `if-elif-else` statements to check the value of the 'gender' column.
- If the gender is 'Male', it returns the string 'Mr. ' concatenated