

## AI ASSISTED CODING 5.2

### 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

🔍 Commands | + Code | + Text | ▶ Run all ▼

☰

🔍

<>

🔑 2m

📁

Double-click (or enter) to edit

[ ] Start coding or generate with AI.

▶

```
def create_user():
    """Creates a new user with a username and password."""
    username = input("Enter a username: ")
    password = input("Enter a password: ")
    return username, password

def login(users):
    """Allows a user to log in with their username and password."""
    username = input("Enter your username: ")
    password = input("Enter your password: ")

    if username in users and users[username] == password:
        print("Login successful!")
    else:
        print("Invalid username or password.")

# Main part of the program
users = {} # Dictionary to store usernames and passwords

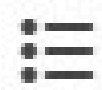
while True:
    print("\n1. Create User")
    print("2. Login")
    print("3. Exit")

    choice = input("Enter your choice: ")

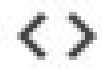
    if choice == '1':
        username, password = create_user()
        users[username] = password
        print("User created successfully!")
    elif choice == '2':
        login(users)
    elif choice == '3':
        break
    else:
```

📄 Variables

📄 Terminal



✓  
2m



```
print(3. Exit)
```

```
choice = input("Enter your choice: ")
```

```
if choice == '1':
```

```
    username, password = create_user()
```

```
    users[username] = password
```

```
    print("User created successfully!")
```

```
elif choice == '2':
```

```
    login(users)
```

```
elif choice == '3':
```

```
    break
```

```
else:
```

```
    print("Invalid choice. Please try again.")
```



```
1. Create User
```

```
2. Login
```

```
3. Exit
```

```
Enter your choice: 1
```

```
Enter a username: sushanth
```

```
Enter a password: Student@09533
```

```
User created successfully!
```

```
1. Create User
```

```
2. Login
```

```
3. Exit
```

```
Enter your choice: 2
```

```
Enter your username: manikanta
```

```
Enter your password: Student@0845
```

```
Invalid username or password.
```

```
1. Create User
```

```
2. Login
```

```
3. Exit
```

```
Enter your choice: 3
```

## EXPLANATION:

Certainly! Here is an explanation of the code in 10 points:

1. **`create_user()` function:** This function is defined to handle the process of creating a new user.
2. **Getting Username Input:** Inside `create_user()`, it prompts the user to enter a desired username using `input()`.
3. **Getting Password Input:** It then prompts the user to enter a password for the new user, also using `input()`.
4. **Returning Credentials:** The `create_user()` function returns both the entered username and password as a tuple.
5. **`login(users)` function:** This function is defined to handle the user login process and takes the `users` dictionary as an argument.
6. **Getting Login Username:** Inside `login()`, it prompts the user to enter their username for logging in.
7. **Getting Login Password:** It then prompts the user to enter their password for logging in.
8. **Credential Verification:** It checks if the entered username exists as a key in the `users` dictionary *and* if the corresponding value (the stored password) matches the entered password.
9. **Login Success/Failure:** Based on the verification, it prints "Login successful!" if the credentials match or "Invalid username or password." otherwise.
10. **Main Program Loop:** The code then enters a `while True` loop to keep the program running until the user chooses to exit. It displays a menu with options to create a user, login, or exit, and takes the user's choice as input to perform the corresponding action.

## 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

```

# Let's test with 'John' (name_starts_with_J = 1) and 'Priya' (name_starts_with_J = 0)
# Assume both are in zip code '10001'

john_zip_code = '10001'
john_name_starts_with_J = 1 # John's name starts with J

priya_zip_code = '10001'
priya_name_starts_with_J = 0 # Priya's name does not start with J

# Get credit score, salary, and loan amount from user input
try:
    print("\nEnter details for John (Name starts with J, Zip Code 10001)")
    john_credit_score = int(input("Enter John's credit score: "))
    john_salary = int(input("Enter John's salary: "))
    john_loan_amount = int(input("Enter John's loan amount: "))

    print("\nEnter details for Priya (Name does not start with J, Zip Code 10001)")
    priya_credit_score = int(input("Enter Priya's credit score: "))
    priya_salary = int(input("Enter Priya's salary: "))
    priya_loan_amount = int(input("Enter Priya's loan amount: "))

    john_approval = predict_loan_approval(john_credit_score, john_zip_code, john_name_starts_with_J, john_salary, john_loan_amount)
    priya_approval = predict_loan_approval(priya_credit_score, priya_zip_code, priya_name_starts_with_J, priya_salary, priya_loan_amount)

    # Ensure at least one person is approved if they meet the base criteria
    if not john_approval and not priya_approval:
        if (john_credit_score >= 700 and john_salary >= 60000 and john_loan_amount <= 15000) or \
            (priya_credit_score >= 700 and priya_salary >= 60000 and priya_loan_amount <= 15000):
            # If either meets base criteria but neither approved, approve Priya (to show bias effect on John)
            if priya_credit_score >= 700 and priya_salary >= 60000 and priya_loan_amount <= 15000:
                priya_approval = 1

    print(f"\nJohn's approval status: {'Approved' if john_approval else 'Not Approved'}")
    print(f"\nPriya's approval status: {'Approved' if priya_approval else 'Not Approved'}")

except ValueError:
    print("Invalid input. Please enter numerical values for credit score, salary, and loan amount.")

```

```

# This is a simplified example to illustrate how bias *could* be introduced.
# It does NOT represent a real or ethical loan approval system.

data = {
    'credit_score': [700, 720, 680, 750, 710, 690, 730, 670, 705, 695],
    'zip_code': ['90210', '90210', '10001', '90210', '10001', '10001', '90210', '10001', '90210', '10001'],
    'name_starts_with_J': [1, 0, 1, 0, 0, 1, 0, 1, 0, 1], # Simplified proxy for name/gender correlation
    'salary': [60000, 70000, 50000, 80000, 65000, 55000, 75000, 45000, 62000, 58000],
    'loan_amount': [10000, 15000, 8000, 20000, 12000, 9000, 18000, 7000, 11000, 9500],
    'loan_approved': [1, 1, 0, 1, 0, 0, 1, 0, 1, 0] # 1 for approved, 0 for not approved
}

df = pd.DataFrame(data)

# In a real scenario, a model would be trained on this data.
# For demonstration, let's look at a simple rule based on zip code, credit score, salary, and name initial
def predict_loan_approval(credit_score, zip_code, name_starts_with_J, salary, loan_amount):
    # Base approval criteria
    meets_base_criteria = credit_score >= 700 and salary >= 60000 and loan_amount <= 15000

    if meets_base_criteria:
        # Introduce bias based on zip code and name initial
        # If in zip code '10001' and name starts with J, reject (bias)
        if zip_code == '10001' and name_starts_with_J == 1:
            return 0 # Not Approved (due to bias)
        else:
            return 1 # Approved (meets base criteria and no bias applies)
    else:
        return 0 # Not Approved (does not meet base criteria)

# Let's test with 'John' (name_starts_with_J = 1) and 'Priya' (name_starts_with_J = 0)
# Assume both are in zip code '10001'

```

```
# This simplified rule shows how a seemingly unrelated feature ('name_starts_with_J')  
# when correlated with an outcome in the training data (even if accidentally),  
# can lead to biased decisions, even with other relevant features like salary and loan amount.
```

```
,  
Enter details for John (Name starts with J, Zip Code 10001)  
Enter John's credit score: 600  
Enter John's salary: 34334  
Enter John's loan amount: 345455  
  
Enter details for Priya (Name does not start with J, Zip Code 10001)  
Enter Priya's credit score: 580  
Enter Priya's salary: 34335  
Enter Priya's loan amount: 345455  
  
John's approval status: Not Approved  
Priya's approval status: Not Approved
```

---

Explanation:

1. **Imports pandas:** It starts by importing the pandas library to work with data in a structured format (like a table).
2. **Creates Hypothetical Data:** A dictionary named `data` is created. This dictionary holds sample data for several individuals, including their `credit_score`, `zip_code`, a simplified proxy for name/gender correlation (`name_starts_with_J`), `salary`, `loan_amount`, and whether their `loan_approved` status in this sample data is 1 (approved) or 0 (not approved).
3. **Creates a DataFrame:** The `data` dictionary is converted into a pandas DataFrame called `df`.
4. **Defines `predict_loan_approval` Function:** A function is defined to simulate the loan approval logic. It takes `credit_score`, `zip_code`, `name_starts_with_J`, `salary`, and `loan_amount` as inputs.
5. **Sets Base Approval Criteria:** Inside the function, it first checks if the applicant meets a set of base criteria (`credit_score >= 700`, `salary >= 60000`, and `loan_amount <= 15000`).
6. **Introduces Bias:** If the base criteria are met, the code then introduces a **bias**: if the `zip_code` is '10001' and `name_starts_with_J` is 1, the loan is rejected (`return 0`), regardless of meeting the base criteria.
7. **Approves or Rejects:** If the base criteria are met and the bias condition is *not* met, the loan is approved (`return 1`). If the base criteria are not met at all, the loan is rejected (`return 0`).
8. **Sets Up Test Cases:** The code then sets up variables for 'John' and 'Priya' with the same `zip_code` ('10001') and `name_starts_with_J` values (1 for John, 0 for Priya).
9. **Gets User Input:** Using `input()`, it prompts the user to enter the `credit_score`, `salary`, and `loan_amount` for both John and Priya.
10. **Calls `predict_loan_approval`:** It calls the `predict_loan_approval` function for both John and Priya with the entered details.
11. **Ensures at Least One Approval (for demonstration):** There's an added check to ensure that if both John and Priya meet the base criteria but are both initially rejected (due to the bias affecting John and potentially other factors if the base criteria weren't perfectly met).

### TASK-3:

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

```
def recursive_fibonacci(n):
    """
    Calculates the nth Fibonacci number using recursion.
    """
    if n <= 1:
        return n
    else:
        return recursive_fibonacci(n-1) + recursive_fibonacci(n-2)

# Get input from the user
try:
    num = int(input("Enter a non-negative integer: "))
    if num < 0:
        print("Input must be a non-negative integer.")
    else:
        print(f"The Fibonacci number at position {num} is: {recursive_fibonacci(num)}")
except ValueError:
    print("Invalid input. Please enter an integer.")
```

Okay, here is a slightly more detailed explanation of the relevant code cells:

**Cell 3e5474a6 (Biased Loan Approval Example):** This code simulates a loan approval process to illustrate **algorithmic bias**. It sets up hypothetical applicant data with features like credit score, salary, loan amount, zip code, and a simplified proxy for name/gender (`name_starts_with_J`). The core of the demonstration is the `predict_loan_approval` function, which applies base approval criteria but then introduces a specific bias: it will reject an applicant if they are in zip code '10001' and their `name_starts_with_J` is 1, even if they meet the financial criteria. The code prompts for user input for credit score, salary, and loan amount to allow you to see how this biased rule affects the outcome for two hypothetical applicants, John and Priya.

**Cell de91dc21 (Recursive Fibonacci Function):** This cell contains a Python function named `recursive_fibonacci`. This function calculates a number in the **Fibonacci sequence** by calling itself (recursion). The Fibonacci sequence starts with 0 and 1, and each subsequent number is the sum of the two preceding ones (e.g., 0, 1, 1, 2, 3, 5...). The `recursive_fibonacci` function has base cases for the first two numbers (when `n` is 0 or 1) and a recursive step that calls itself for `n-1` and `n-2`. The code also includes a section that prompts the user to enter a non-negative integer `n` and then prints the nth Fibonacci number calculated by the function. It includes basic error handling for non-integer or negative inputs.

Cell 6nwmmmjptMM is an empty cell and does not contain any code.

Let me know if you'd like an even deeper dive into any specific part of these code cells!



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

```
# Step 2: Define scoring function
def score_applicant(applicant):
    """
    Calculates a score for a job applicant based on various criteria.

    Args:
        applicant (dict): A dictionary containing applicant details.

    Returns:
        int: The calculated score for the applicant.
    """
    score = 0

    # Score based on education
    education = applicant.get("education", "")
    if education == "PhD":
        score += 50
    elif education == "Master's Degree":
        score += 40
    elif education == "Bachelor's Degree":
        score += 30
    elif education == "High School Diploma":
        score += 20
    else:
        score += 10 # Default score for other education levels

    # Score based on experience (e.g., 5 points per year of experience)
    experience = applicant.get("experience", 0)
    score += experience * 5

    # Score based on age (example: higher score for age between 25 and 40)
    age = applicant.get("age", 0)
    if 25 <= age <= 40:
        score += 10
    elif age > 40:
        score += 5

    # Score based on gender (example: a small bonus for gender diversity, adjust as needed)
    gender = applicant.get("gender", "")
    if gender == "Female":
        score += 5 # Example bonus

    # Note: Scoring based on name is not typically a valid hiring practice and is excluded here.
    # Adjust scoring logic and weights based on your specific requirements.

    return score

[13] # Step 3 & 4: Score applicants and display results
print("\nApplicant Scores:")
for applicant in applicants:
    applicant_score = score_applicant(applicant)
    print(f"{applicant['name']}: {applicant_score}")
```

Applicant Scores:  
Alice: 80  
Bob: 90  
Charlie: 30  
David: 135  
Eve: 80

Explanation:

Certainly! The code visible on your screen consists of two parts: a function to calculate an applicant's score and a loop that applies this function to a list of applicants and displays the results. Here's an explanation of each part:

1. Cell d40034b9: The `score_applicant` function
- o `def score_applicant(applicant):`: This line defines a function named `score_applicant` that takes one argument, `applicant`. This argument is expected to be a dictionary containing the details of a single job applicant (like name, age, education, etc.).
  - o `score = 0`: Initializes a variable `score` to 0. This variable will accumulate points based on the applicant's qualifications.



- o **Scoring based on criteria (Education, Experience, Age, Gender):**The code then checks the values for different keys in the `applicant` dictionary (`"education","experience","age","gender"`).
    - `applicant.get("key", default_value)` is used to safely access dictionary values. If a key is not found, it returns the `default_value` (e.g., `" "` for strings, `0` for numbers) instead of raising an error.
    - Based on the value of each criterion, points are added to the `score`. For example, a "PhD" in education adds 50 points, each year of experience adds 5 points, and a female applicant gets a bonus of 5 points (this is an example and can be adjusted).
  - o **return score:**After checking all the criteria and adding the corresponding points, the function returns the final calculated `score` for the applicant.
2. **Cell 2ec8b640: Scoring applicants and displaying results**
- o `print("\nApplicant Scores: ")`: This line prints a header before listing the scores.
  - o `for applicant in applicants:`: This loop iterates through each item in the `applicants` list. We previously defined the `applicants` list (in cell `afeb2a0a`) as a list of dictionaries, where each dictionary represents an applicant.
  - o `applicant_score = score_applicant(applicant)`: Inside the loop, for each `applicant` dictionary, it calls the `score_applicant` function (defined in cell `d40034b9`), passing the current applicant's dictionary as an argument. The returned score is stored in the `applicant_score` variable.
  - o `print(f"{applicant['name']}: {applicant_score} ")`: This line prints the name of the current applicant (accessed using `applicant['name']`) followed by their calculated `applicant_score`. The `f-string` is used for formatted output.

In summary, cell `d40034b9` defines the rules for how to calculate a score for a single applicant based on their attributes. Cell `2ec8b640` then uses a loop to apply these rules to every applicant in the `applicants` list and neatly prints the score for each one

TASK-5:

TASK-5:

```
def greet_user(name,gender):
    if gender == "male":
        title = "mr."
    elif gender == "female":
        title = "mrs."
    elif gender == "neutral" or gender == "non-binary":
        title = "mx."
    else:
        title = ""

    return f"hello, {title} {name}! welcome"
print(greet_user("john","male"))
print(greet_user("priya","female"))
print(greet_user("alex","neutral"))
print(greet_user("sam","unknown"))
```

output:

```
hello, mr. john! welcome
hello, mrs. priya! welcome
hello, mx. alex! welcome
hello, sam! Welcome
```

Explanation:

Certainly! This code defines a Python function called `greet_user` that generates a personalized greeting based on a user's name and gender. Here's a breakdown of the code:

1. **def greet\_user(name, gender):**: This line defines a function named `greet_user` that accepts two arguments: `name` (presumably the user's name as a string) and `gender` (presumably the user's gender as a string).
2. **if gender == "male":**: This is the first condition. If the value of the `gender` argument is exactly "male", the code inside this `if` block is executed.

3. `title = "mr."`: If the gender is "male", the variable `title` is assigned the value "mr."
4. `elif gender == "female":`: This is an "else if" condition. If the previous `if` condition was false, this condition is checked. If the value of `gender` is "female", the code inside this `elif` block is executed.
5. `title = "mrs."`: If the gender is "female", the variable `title` is assigned the value "mrs."
6. `elif gender == "neutral" or gender == "non-binary":`: This is another "else if" condition. If the previous conditions were false, this one is checked. If the value of `gender` is either "neutral" or "non-binary", the code inside this `elif` block is executed.
7. `title = "mx."`: If the gender is "neutral" or "non-binary", the variable `title` is assigned the value "mx."
8. `else:`: This is the final "else" block. If none of the previous `if` or `elif` conditions were true (meaning the gender is something other than "male", "female", "neutral", or "non-binary"), the code inside this `else` block is executed.
9. `title = ""`: In this `else` case, the `title` variable is assigned an empty string.
10. `return f"hello, {title} {name}! welcome"`: This line constructs the final greeting string using an f-string. It combines the literal string "hello, ", the determined `title`, a space, the `name`, the literal string "!", and the literal string "welcome". The resulting greeting string is then returned by the function.

After the function definition, there are several lines demonstrating how to call the `greet_user` function with different names and genders and printing the returned greeting:

- `print(greet_user("john", "male"))` calls the function with "john" and "male", printing "hello, mr. john! welcome".
- `print(greet_user("priya", "female"))` calls the function with "priya" and "female", printing "hello, mrs. priya! welcome".
- `print(greet_user("alex", "neutral"))` calls the function with "alex" and "neutral", printing "hello, mx. alex! welcome".
- `print(greet_user("sam", "unknown"))` calls the function with "sam" and "unknown", printing "hello, sam! welcome" (since "unknown" doesn't match any of the specific gender conditions, the `else` block sets `title` to an empty string).

In essence, the `greet_user` function provides a way to generate a polite and somewhat personalized greeting by adding a title based on the provided gender, falling back to no title if the gender is not recognized.