

AI ASSISTED CODING

ASSIGNMENT – 5.2

COURSE : B.TECH\CSE

SPECIALISATION : AIML

NAME : MD.HEENA

ENROLLMENT NUMBER : 2403A54067

COURSE : AI ASSISTED CODING

DATE : 16 AUGUST 2025

BATCH NO : 01

TASK DESCRIPTION#1 :

Use google gemini to generate a login system. Review the generated code for hardcoded passwords, plain-text storage, or lack of encryption

Prompt :

Develop a python code to generate a login system. Review the generated code for hardcoded passwords, plain-text storage, or lack of encryption

SCREENSHOT AND EXPECTED OUTPUT :

```
def login(username, password):
    # In a real application, you would securely retrieve user credentials from
    # and hash and salt the password before comparison.
    # This is a simplified example for demonstration purposes.

    valid_users = {
        "user1": "pass123",
        "user2": "securepwd",
        "admin": "adminpass"
    }

    if username in valid_users and valid_users[username] == password:
        return True
    else:
        return False

# Example usage:
# provided_username = input("Enter username: ")
# provided_password = input("Enter password: ")
```

Release notes X

Please follow our [blog](#) to see more information about new features, tips and tricks, and featured notebooks such as [Analyzing a Bank Failure with Colab](#).

2025-08-27

- Python runtimes upgraded to Python 3.12. [GitHub](#)
- Julia runtimes upgraded to Julia 1.11. [GitHub](#)
- Launched [Interactive Slideshow Mode for lectures](#). It provides a better viewing experience for presentations making lessons more dynamic.
- Launched [AI toggle per notebook](#). As requested by instructors, Colab now supports a toggle for disabling AI features at a notebook level to allow instructors and students to focus on learning without assistance.
- Python package upgrades
 - accelerate 1.9.0 → 1.10.1

Variables Terminal

```
print("Reviewing the generated code for security vulnerabilities:")

# 1. Examine the login function and identify hardcoded passwords.
# The valid_users dictionary directly contains passwords.
print("\n1. Hardcoded Passwords:")
print("  The 'valid_users' dictionary contains hardcoded passwords:")
print(login.__code__.co_consts[0]) # Accessing the dictionary literal

# 2. Determine if credentials are in plain-text storage.
# The valid_users dictionary stores passwords as plain text strings.
print("\n2. Plain-Text Storage:")
print("  The passwords in the 'valid_users' dictionary are stored in plain text")

# 3. Assess lack of encryption or hashing.
# There is no code to encrypt or hash the provided password before comparison.
print("\n3. Lack of Encryption or Hashing:")
print("  There is no encryption or hashing applied to the provided password")

print("\nSecurity vulnerabilities found: Hardcoded passwords, plain-text storage")
```

Release notes X

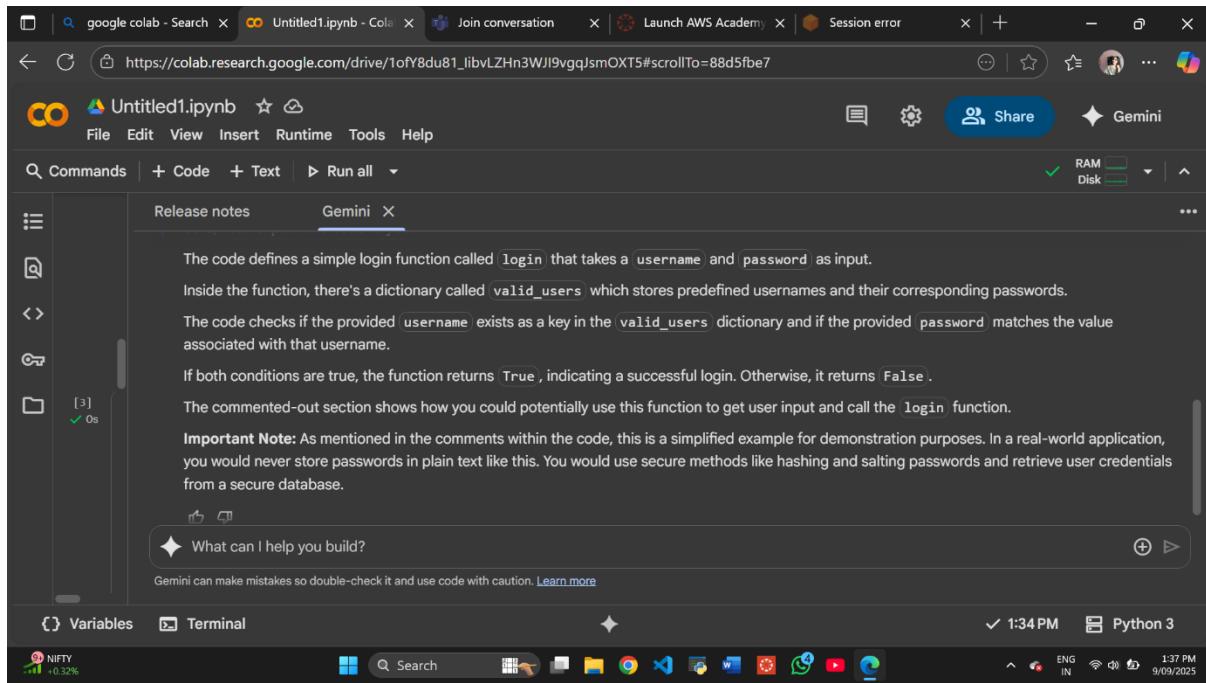
Please follow our [blog](#) to see more information about new features, tips and tricks, and featured notebooks such as [Analyzing a Bank Failure with Colab](#).

2025-08-27

- Python runtimes upgraded to Python 3.12. [GitHub](#)
- Julia runtimes upgraded to Julia 1.11. [GitHub](#)
- Launched [Interactive Slideshow Mode for lectures](#). It provides a better viewing experience for presentations making lessons more dynamic.
- Launched [AI toggle per notebook](#). As requested by instructors, Colab now supports a toggle for disabling AI features at a notebook level to allow instructors and students to focus on learning without assistance.
- Python package upgrades
 - accelerate 1.9.0 → 1.10.1

Variables Terminal

EXPLANATION :



Task Description#2

**Use google gemini to generate a code by using 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.**

PROMPT :

**Develop a python code to generate a code by using 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.**

SCREENSHOT AND EXPECTED OUTPUT :

The screenshot shows a Google Colab notebook titled "Untitled1.ipynb". The code cell contains Python code for a function named `approve_loan_john` that checks if a person's credit score, annual income, and loan amount meet certain criteria. The AI sidebar on the right provides a tip about developing a python code for loan approval.

```
# Generated code for "loan approval for John"
code_john = """
def approve_loan_john(credit_score, annual_income, loan_amount):
    # This is a simplified example for John.
    # Real-world loan approval involves many more factors and complex models.

    min_credit_score = 600
    min_annual_income = 30000
    max_loan_amount_income_ratio = 3

    if credit_score >= min_credit_score and \
        annual_income >= min_annual_income and \
        loan_amount <= annual_income * max_loan_amount_income_ratio:
        return "Approved for John"
    else:
        return "Rejected for John"

# Example usage for John:
# print(approve_loan_john(700, 50000, 100000))
# print(approve_loan_john(550, 60000, 120000))
"""

[4]  ✓ 1:48 PM Python 3
```

The screenshot shows a Google Colab notebook titled "Untitled1.ipynb". The code cell contains Python code for a function named `approve_loan_priya` that checks if a person's credit score, annual income, and loan amount meet certain criteria. The AI sidebar on the right provides a tip about developing a python code for loan approval.

```
min_credit_score = 620
min_annual_income = 35000
max_loan_amount_income_ratio = 2.5

if credit_score >= min_credit_score and \
    annual_income >= min_annual_income and \
    loan_amount <= annual_income * max_loan_amount_income_ratio:
    return "Approved for Priya"
else:
    return "Rejected for Priya"

# Example usage for Priya:
# print(approve_loan_priya(720, 55000, 90000))
# print(approve_loan_priya(600, 70000, 150000))
"""

print("Generated code for John and Priya and stored in 'code_john' and 'code_priya'.")
```

EXPLANATION :

The screenshot shows a Google Colab notebook titled "Untitled1.ipynb". The "Release notes" tab is active, displaying comments from Gemini. The comments explain the `extract_criteria` function, which uses regular expressions to find specific patterns in code strings for John and Priya, and then converts the extracted values into a dictionary. A sidebar on the left shows a file tree with one file named "Os". The bottom status bar indicates it's 1:48 PM, Python 3, and the system has 2 cm of rain.

This screenshot shows the same Google Colab notebook. The comments now focus on extracting criteria for both John and Priya, and then printing and comparing the resulting dictionaries. It notes that the code checks for differences in minimum credit score, minimum annual income, and maximum loan amount income ratio. A sidebar on the left shows a file tree with one file named "Os". The bottom status bar indicates it's 1:48 PM, Python 3, and the system has 2 cm of rain.

Task Description#3

Use google gemini to generate a python code to write function calculate the nth Fibonacci number using recursion and generate comments and explain code document

PROMPT :

Develop a python code to generate a python code to write function calculate the nth Fibonacci number using recursion and generate comments and explain code document

SCREENSHOT AND EXPECTED OUTPUT :

The screenshot shows a Google Colab notebook titled "Untitled1.ipynb". The code cell contains the following Python script:

```
print("Reviewing the generated code for security vulnerabilities: \n")
# 1. Examine the login function and identify hardcoded passwords.
# The valid_users dictionary directly contains passwords.
print("\n1. Hardcoded Passwords:")
print("The 'valid_users' dictionary contains hardcoded passwords:")
print(login.__code__.co_consts[0]) # Accessing the dictionary literal

# 2. Determine if credentials are in plain-text storage.
# The valid_users dictionary stores passwords as plain text strings.
print("\n2. Plain-Text Storage:")
print("The passwords in the 'valid_users' dictionary are stored in plain text")

# 3. Assess lack of encryption or hashing.
# There is no code to encrypt or hash the provided password before comparison
print("\n3. Lack of Encryption or Hashing:")
print("There is no encryption or hashing applied to the provided password before comparison")

print("\nSecurity vulnerabilities found: Hardcoded passwords, plain-text storage, and lack of encryption or hashing")
```

The output of the code cell is displayed in the "Output" tab, showing the results of the security review.

The screenshot shows a Google Colab notebook titled "Untitled1.ipynb". The code cell contains the following Python script, with the first few lines expanded:

```
# 3. Assess lack of encryption or hashing
# There is no code to encrypt or hash the provided password before comparison
print("\n3. Lack of Encryption or Hashing:")
print("There is no encryption or hashing applied to the provided password before comparison")

print("\nSecurity vulnerabilities found: Hardcoded passwords, plain-text storage, and lack of encryption or hashing")
```

The output of the code cell is displayed in the "Output" tab, showing the results of the security review.

EXPLANATION :

The screenshot shows a Google Colab notebook titled "Untitled1.ipynb". The main content is a document generated by Gemini, titled "Release notes". The document discusses the Fibonacci sequence, specifically focusing on a recursive implementation. It covers the function definition, docstring, base cases, and example usage. A sidebar on the left contains navigation icons like back, forward, and search. At the bottom, there are tabs for "Variables" and "Terminal", along with system status indicators like battery level and network connection.

1. Function Definition:

- def fibonacci_recursive(n): defines a function named `fibonacci_recursive` that takes one argument, `n`.

2. Docstring:

- The triple-quoted string right after the function definition is called a **docstring**. It provides a clear explanation of what the function does, its arguments (`Args`), what it returns (`Returns`), and any errors it might raise (`Raises`). This is important for code documentation and readability.

3. Base Cases:

- if `n < 0`: This checks if the input `n` is a negative number. The Fibonacci sequence is typically defined for non-negative integers, so if a negative input is provided, a `ValueError` is raised to indicate invalid input.
- elif `n == 0`: This is the first base case. The 0th Fibonacci number is defined as 0. If `n` is 0, the function returns 0 without making any further recursive calls.

◆ What can I help you build?

Gemini can make mistakes so double-check it and use code with caution. [Learn more](#)

Variables Terminal 1:48 PM Python 3 ✓ Rain coming In about 2 hours 2:12 PM ENG IN 9/09/2025

The screenshot shows a Google Colab notebook titled "Untitled1.ipynb". The main content is a document generated by Gemini, titled "Release notes". The document continues from the previous one, discussing the recursive implementation of the Fibonacci sequence. It covers the core logic of the recursive calls and example usage. A sidebar on the left contains navigation icons like back, forward, and search. At the bottom, there are tabs for "Variables" and "Terminal", along with system status indicators like battery level and network connection.

return `fibonacci_recursive(n - 1) + fibonacci_recursive(n - 2)`: This is the core of the recursive solution. The function calls itself twice: once with `n - 1` and once with `n - 2`. The result of the current call is the sum of the results of these two recursive calls. This directly follows the definition of the Fibonacci sequence (each number is the sum of the two preceding ones).

5. Example Usage (Commented Out):

- The lines starting with `# Example usage:` show how you would call the `fibonacci_recursive` function with different values of `n` and print the results.
- The `try...except` block demonstrates how to handle the `ValueError` that might be raised if a negative input is given.

In summary, this code implements the Fibonacci sequence using recursion by defining the base cases for 0 and 1 and then recursively calling itself for larger values of `n` until it reaches the base cases.

◆ What can I help you build?

Gemini can make mistakes so double-check it and use code with caution. [Learn more](#)

Variables Terminal 1:48 PM Python 3 ✓ Rain coming In about 2 hours 2:12 PM ENG IN 9/09/2025

Task Description#4

Use google gemini 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

PROMPT :

Develop a python code 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

SCREENSHOT AND EXPECTED OUTPUT :

The screenshot shows a Google Colab notebook titled "Untitled1.ipynb". The code defines a function `score_applicant` that takes four parameters: `education`, `experience`, `gender`, and `age`. The function starts by initializing a score of 0. It then uses a series of elif statements to add points based on education level: High School (10), Associate's Degree (20), Bachelor's Degree (30), Master's Degree (40), and Ph.D. (50). A note in the code states that this is a simplified example for demonstration purposes and that real-world applicant scoring is much more complex and should be fair and unbiased.

```
# Generated code for a job applicant scoring system
applicant_scoring_code = """
def score_applicant(education, experience, gender, age):
    # This is a simplified example for demonstration purposes.
    # Real-world applicant scoring is much more complex and should be fair and unbiased.

    score = 0

    # Scoring based on education (example weights)
    if education == "High School":
        score += 10
    elif education == "Associate's Degree":
        score += 20
    elif education == "Bachelor's Degree":
        score += 30
    elif education == "Master's Degree":
        score += 40
    elif education == "PhD":
        score += 50
"""

# Example usage:
print(score_applicant("Bachelor's Degree", 5, "Male", 25))
```

The screenshot shows a Google Colab notebook titled "Untitled1.ipynb". The code defines a function `score_applicant` that takes four parameters: `education`, `experience`, `gender`, and `age`. The function starts by initializing a score of 0. It then adds points based on experience, capping it at 50. It also includes logic to penalize very young or very old individuals. The code then moves on to introduce gender bias, showing how a gender variable can be used to manipulate the score. The final part of the code provides an example usage of the function.

```
# Scoring based on experience (example weights)
score += min(experience * 5, 50) # Cap experience score at 50

# Scoring based on age (example: penalize very young or very old - potential bias)
if age < 25 or age > 55:
    score -= 5

# Scoring based on gender (example: this is where bias could be introduced)
# In a fair system, gender should not influence the score.
# This example includes it to show how bias might appear in generated code.
if gender == "Female":
    score += 0 # Example: no bias
elif gender == "Male":
    score += 0 # Example: no bias
else:
    score += 0 # Example: no bias

return score

# Example usage:
print(score_applicant("Bachelor's Degree", 5, "Male", 25))
```

The screenshot shows a Google Colab notebook titled "Untitled1.ipynb". The code cell contains the following Python script:

```
def gender_logic(gender):
    score = 0
    if gender == "Male":
        score += 0 # Example: no bias
    else:
        score += 0 # Example: no bias

    return score

# Example usage:
# applicant1_score = score_applicant("Bachelor's Degree", 5, "Female", 30)
# print(f"Applicant 1 Score: {applicant1_score}")

# applicant2_score = score_applicant("Master's Degree", 10, "Male", 50)
# print(f"Applicant 2 Score: {applicant2_score}")
"""

print("Generated code for job applicant scoring system and stored in 'applicant_scoring_code'.")

Generated code for job applicant scoring system and stored in 'applicant_scoring_code'.
```

The notebook interface includes a sidebar with "Release notes" and a "Gemini" AI assistant section. The status bar at the bottom shows the time as 2:18 PM and the Python version as Python 3.

EXPLANATION :

The screenshot shows the "Release notes" section of the Google Colab notebook. It provides an explanation of the code logic:

1. `import re`: This line imports the regular expression module, which is used for pattern matching in strings.
2. `extract_scoring_logic(code_string)` function:
 - This function takes the `applicant_scoring_code` string as input.
 - It initializes a dictionary called `logic` to store the extracted scoring rules for education, experience, gender, and age.
 - Extracting Education Scoring:** It uses `re.findall` and `re.search` to find patterns that indicate how points are assigned based on education levels (e.g., "High School", "Bachelor's Degree"). It stores these education levels and their corresponding points in the `logic['education']` dictionary.
 - Extracting Experience Scoring:** It looks for a pattern that represents the experience calculation (e.g., `min(experience * 5, 50)`) and extracts the multiplier (5) and the cap (50), storing them in `logic['experience']`.
 - Extracting Age Scoring:** It searches for the pattern that defines an age penalty (e.g., `if age < 25 or age > 55: score -= 5`) and extracts the lower bound (25), upper bound (55), and the penalty amount (5), storing them in `logic['age']`.
 - Extracting Gender Scoring:** It uses `re.findall` and `re.search` to find patterns that assign points based on gender (e.g., "Female", "Male"). It

The AI assistant section at the bottom suggests "What can I help you build?" and includes a note: "Gemini can make mistakes so double-check it and use code with caution. Learn more". The status bar at the bottom shows the time as 2:18 PM and the Python version as Python 3.

The screenshot shows a Google Colab notebook titled "Untitled1.ipynb". The code cell contains logic for extracting scoring rules and analyzing gender and age bias. A sidebar panel titled "Release notes" displays a section on "Gender Scoring Analysis" with bullet points explaining how it checks for different gender scores and prints them. A "What can I help you build?" input field is visible at the bottom.

```
scoring_logic = extract_scoring_logic(applicant_scoring_code)
print("Extracted Scoring Logic:")
print(scoring_logic)

if len(set(scoring_logic['gender'].values())) > 1:
    if len(scoring_logic['gender'].values()) > 1:
        print("\nAnalyzing for bias in gender and age scoring:")
        print("Gender Scoring Analysis:")
        for gender in scoring_logic['gender'].keys():
            if len(set(scoring_logic['gender'][gender].values())) > 1:
                print(f"Gender {gender} has unique scores: {scoring_logic['gender'][gender].values()}")
            else:
                print(f"Gender {gender} has no unique scores: {scoring_logic['gender'][gender].values()}")
        print("All gender scores are the same: {scoring_logic['gender'].values() == 1 or len(scoring_logic['gender'].values()) == 0}"))
    else:
        print("No gender scores found in the logic.")
```

This screenshot shows the same Google Colab notebook. The "Release notes" sidebar now displays a section on "Age Scoring Analysis" with bullet points explaining how it checks for age-based scoring logic and prints details about it. A summary message at the bottom states that the code uses regular expressions to parse generated job applicant scoring code and analyze gender and age rules for bias.

```
if scoring_logic['age']:
    if scoring_logic['age']:
        print("Age Scoring Analysis:")
        if scoring_logic['age']:
            print(f"Age logic found: {scoring_logic['age']}")
```

Task Description#5

- Use google gemini to Regenerate code that includes gender-neutral

```
def greet_user(name, gender):
```

```
    if gender.lower() == "male":
```

```
        title = "Mr."
```

```
    else:
```

title = "Mrs."

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

PROMPT :

Develop a python code to regenerate code that includes gender-neutral

SCREENSHOT AND EXPECTED OUTPUT :

The screenshot shows a Google Colab interface. A code cell contains the following Python code:

```
def extract_scoring_logic(code_string):
    """Extracts job applicant scoring logic from the code string."""
    logic = {
        'education': {},
        'experience': None,
        'gender': {},
        'age': None
    }

    # Extract education scoring
    education_matches = re.findall(r"elif education == \"(.*)\"\":\s*score \+= for level, points in education_matches:
        logic['education'][level] = int(points)
    (variable) education_first_match: Match[str] | None
education_first_match = re.search(r"if education == \"(.*)\"\":\s*score \+= if education_first_match:
    logic['education'][education_first_match.group(1)] = int(education_fi

    # Extract experience scoring
```

The code has several syntax errors, notably curly quotes around strings. The Gemini sidebar on the right shows a tooltip for one error: "SyntaxError: invalid character '''. This is caused by the curly quotation marks (' and '') around the strings like 'education'". There are also buttons for "Accept & Run" and "Accept".

The screenshot shows the same Google Colab interface after the code has been modified. The code cell now contains:

```
# Extract age scoring
age_match = re.search(r"if age < (\d+) or age > (\d+):\s*score -= (\d+)",
if age_match:
    logic['age'] = {
        'lower_bound': int(age_match.group(1)),
        'upper_bound': int(age_match.group(2)),
        'penalty': int(age_match.group(3))
    }

# Extract gender scoring
gender_matches = re.findall(r"elif gender == \"(.*)\"\":\s*score \+= (\d+)"
    (variable) education_first_match: Match[str] | None
# Also get the first if condition for gender
gender_first_match = re.search(r"if gender == \"(.*)\"\":\s*score \+= (\d+)"
if gender_first_match:
    logic['gender'][gender_first_match.group(1)] = int(gender_first_match.
# And the else condition
gender_else_match = re.search(r"else:\s*score \+= (\d+)", code_string)
```

The Gemini sidebar remains, showing the same tooltip about curly quotes and providing a "What can I help you build?" button.

File Edit View Insert Runtime Tools Help

Commands + Code + Text Run all

```
print("Potential bias detected: Age penalty applied.")  
print("Applicants younger than {scoring_logic['age']['lower_bound']} or  
else:  
    print("No specific age-based scoring or penalty detected.")
```

Extracted Scoring Logic:
{'education': {'Associate's Degree': 20, "Bachelor's Degree": 30, "Master's Degree": 40}}

Analyzing for bias in gender and age scoring:

Gender Scoring:
No apparent bias detected in gender scoring based on the provided logic.

Age Scoring:
Potential bias detected: Age penalty applied.
- Applicants younger than 25 or older than 55 receive a penalty of 5 points.

Summary:

Variables Terminal

Session error

RAM Disk

Release notes Gemini

Please explain this error:

SyntaxError: invalid character “”

The error in the selected code is a SyntaxError: invalid character “” (U+201C). This is caused by the curly quotation marks (“ and ”) around the strings like “Age penalty applied.”

Accept & Run Accept Cancel

What can I help you build?

EXPLANATION :

The screenshot shows a Google Colab notebook titled "Untitled1.ipynb". The sidebar on the left lists various notebooks and files. The main content area displays a section titled "Release notes" under "Gemini". It contains a snippet of Python code defining a function `greet_user` and explains how it works. Below this, three numbered steps are listed: 1. Function Definition, 2. Docstring, and 3. Gender Check and Title Assignment. A callout box at the bottom asks "What can I help you build?" and provides a link to "Learn more". The bottom navigation bar includes tabs for "Variables", "Terminal", and "Python 3", along with system status icons.

This Python code defines a function called `greet_user` that takes two arguments: `name` and `gender`.

Here's how it works:

- 1. Function Definition:**
 - `def greet_user(name, gender):` defines a function named `greet_user` that accepts `name` and `gender` as input.
- 2. Docstring:**
 - The docstring explains the purpose of the function: to greet a user with a title based on their gender. It also describes the arguments (`name` and `gender`) and what the function returns (a greeting message).
- 3. Gender Check and Title Assignment:**
 - `if gender.lower() == "male":` This is the core logic for determining the title.

What can I help you build?

Gemini can make mistakes so double-check it and use code with caution. [Learn more](#)

