AI Assisted Coding

Name : PRAVALIKA MUTHOJU

H.no : 2303A52347

Batch : 34

Assignment Number : 5.5

Lab 5: Ethical Foundations – Responsible AI Coding Practices

Lab Objectives:

• To explore the ethical risks associated with AI-generated

Week3 - code.

• To recognize issues related to security, bias, transparency, and copyright.

• To reflect on the responsibilities of developers when using AI tools in software development.

• To promote awareness of best practices for responsible and ethical AI coding.

Lab Outcomes (LOs):

After completing this lab, students will be able to:

• Identify and avoid insecure coding patterns generated by AI tools.

• Detect and analyze potential bias or discriminatory logic in AI-generated outputs.

• Evaluate originality and licensing concerns in reused AI- generated code.

• Understand the importance of explainability and transparency in AI-assisted programming.

• Reflect on accountability and the human role in ethical AI coding practices.

Task Description #1 (Transparency in Algorithm Optimization)

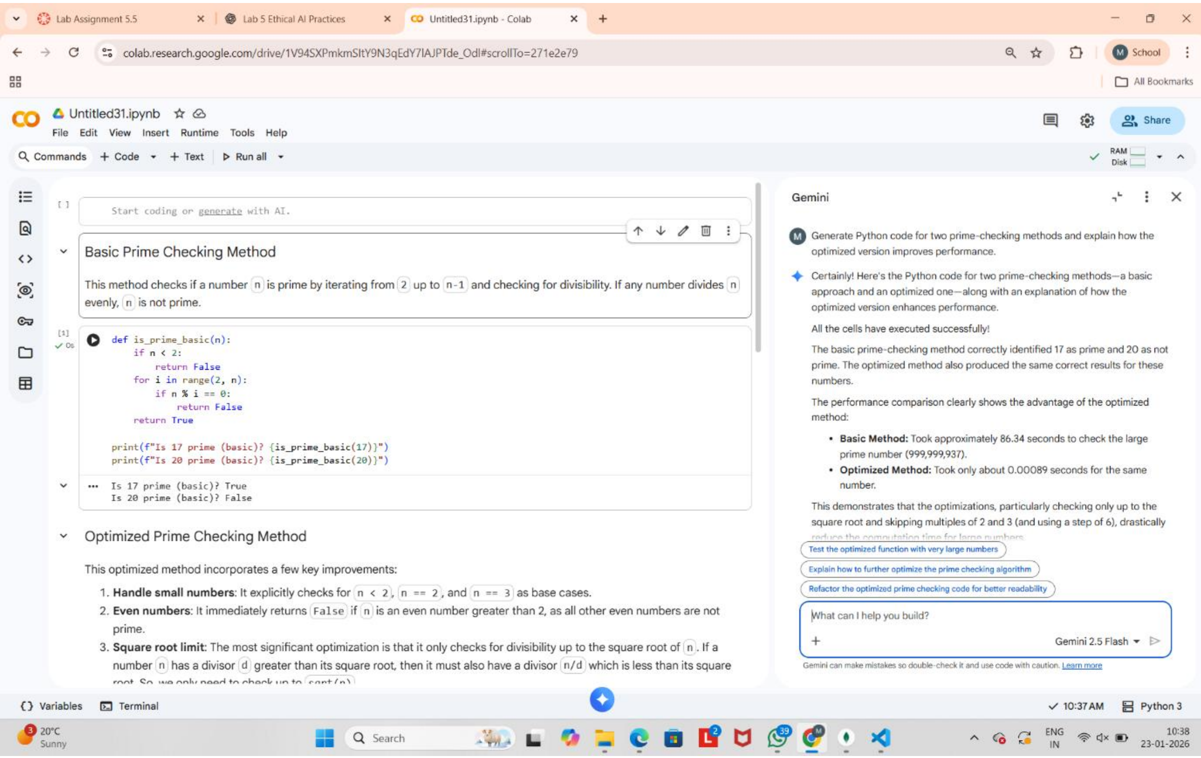
Task: Use AI to generate two solutions for checking prime numbers:

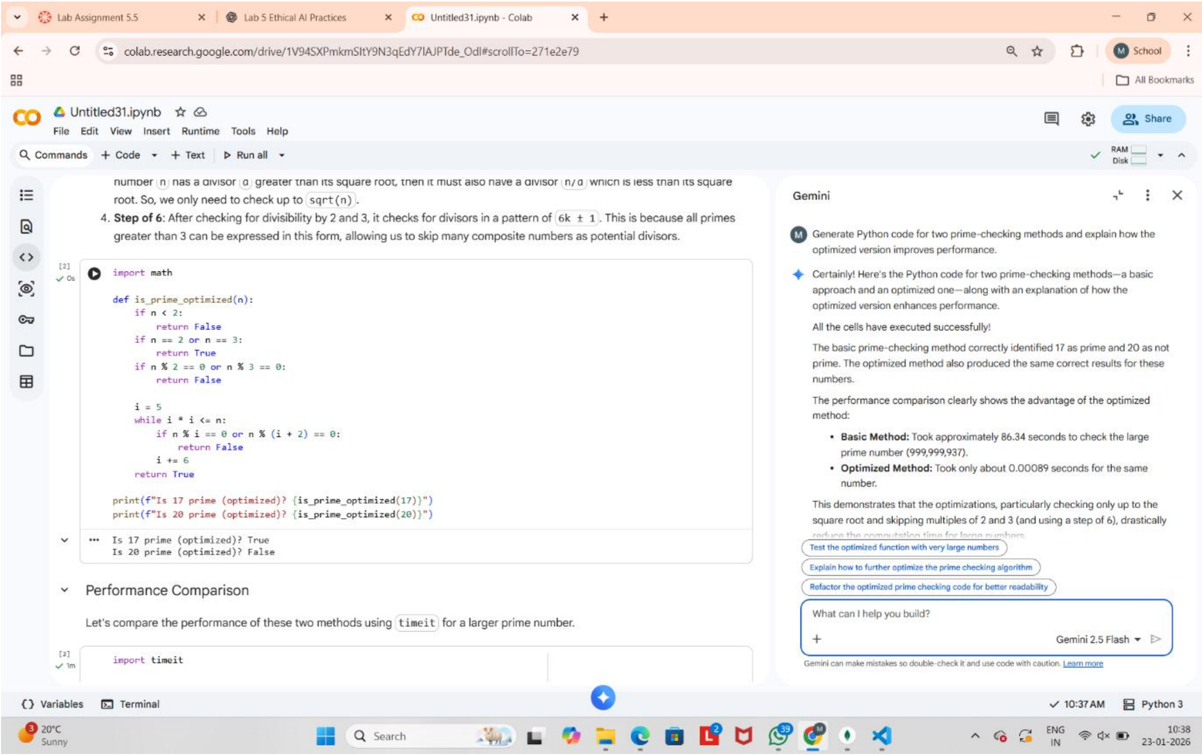
• Naive approach(basic)

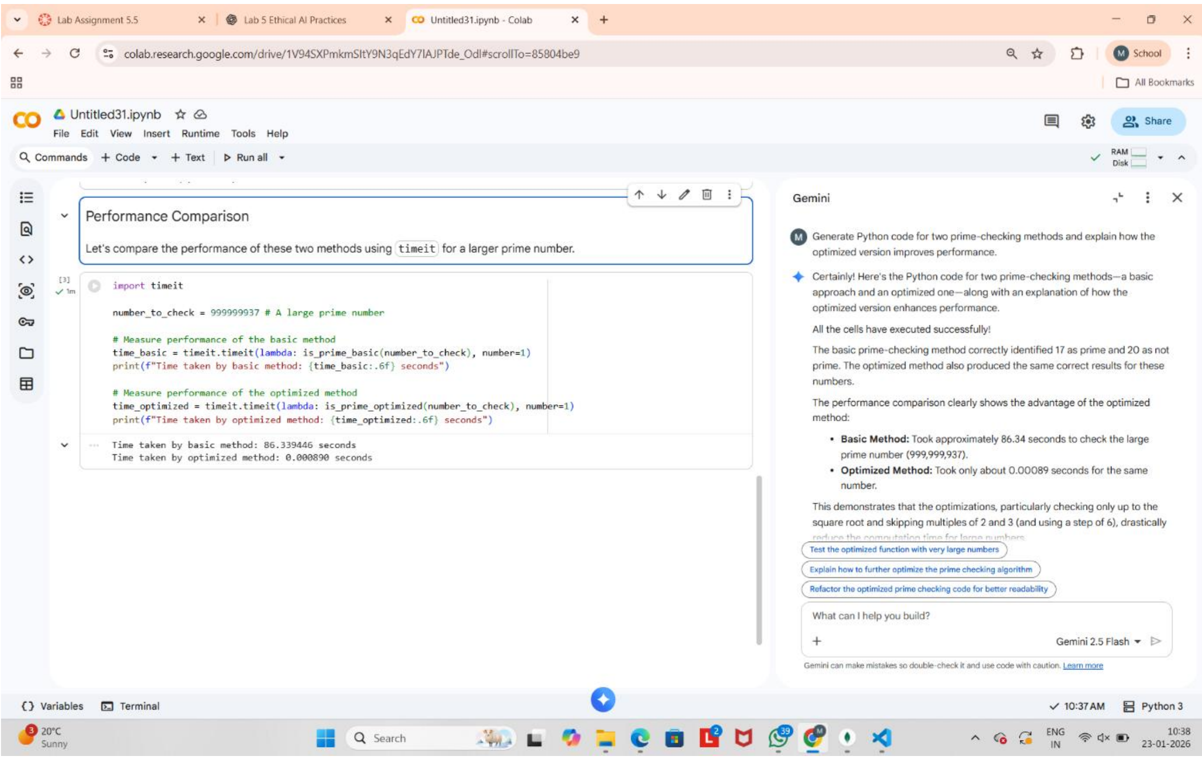
• Optimized approach

Prompt:

“Generate Python code for two prime-checking methods and explain how the optimized version improves performance.”

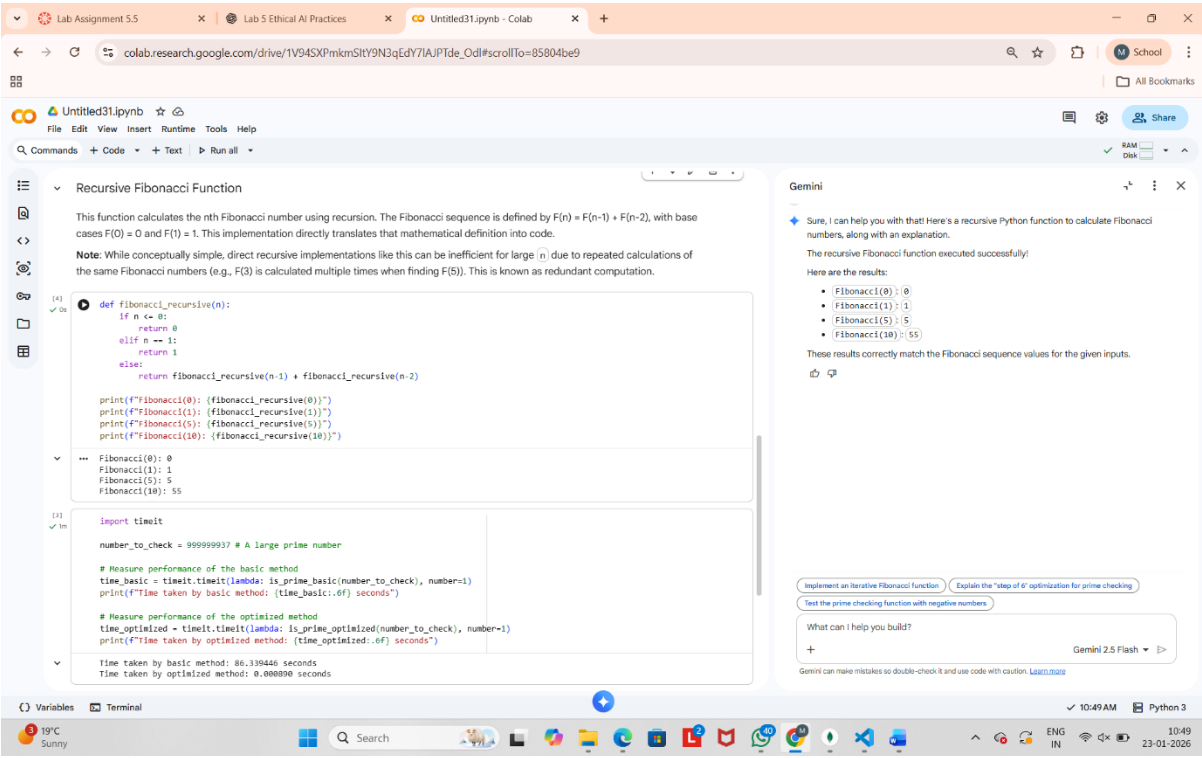



Lab Assignment 5.5
Lab 5 Ethical AI Practices
Untitled31.ipynb - Colab
colab.research.google.com/drive/1V94SXPmkmSltY9N3qEdY7IAJPTde_Odl#scrollTo=271e2e79
All Bookmarks
Untitled31.ipynb
File Edit View Insert Runtime Tools Help
Commands + Code + Text Run all
Share
number n has a divisor d greater than its square root, then it must also have a divisor n/d which is less than its square root. So, we only need to check up to sqrt(n).
4. Step of 6: After checking for divisibility by 2 and 3, it checks for divisors in a pattern of 6k ± 1. This is because all primes greater than 3 can be expressed in this form, allowing us to skip many composite numbers as potential divisors.
[2]
import math
def is_prime_optimized(n):
    if n < 2:
        return False
    if n == 2 or n == 3:
        return True
    if n % 2 == 0 or n % 3 == 0:
        return False
    i = 5
    while i * i <= n:
        if n % i == 0 or n % (i + 2) == 0:
            return False
        i += 6
    return True
print(f"Is 17 prime (optimized)? {is_prime_optimized(17)}")
print(f"Is 20 prime (optimized)? {is_prime_optimized(20)}")
Is 17 prime (optimized)? True
Is 20 prime (optimized)? False
Performance Comparison
Let's compare the performance of these two methods using timeit for a larger prime number.
[3]
import timeit
Gemini
Generate Python code for two prime-checking methods and explain how the optimized version improves performance.
Certainly! Here's the Python code for two prime-checking methods—a basic approach and an optimized one—along with an explanation of how the optimized version enhances performance.
All the cells have executed successfully!
The basic prime-checking method correctly identified 17 as prime and 20 as not prime. The optimized method also produced the same correct results for these numbers.
The performance comparison clearly shows the advantage of the optimized method:
Basic Method: Took approximately 86.34 seconds to check the large prime number (999,999,937).
Optimized Method: Took only about 0.00089 seconds for the same number.
This demonstrates that the optimizations, particularly checking only up to the square root and skipping multiples of 2 and 3 (and using a step of 6), drastically
Test the optimized function with very large numbers
Explain how to further optimize the prime checking algorithm
Refactor the optimized prime checking code for better readability
What can I help you build?
Gemini 2.5 Flash
Gemini can make mistakes so double-check it and use code with caution. Learn more
Variables
Terminal
10:37 AM
Python 3
20°C
Sunny
Search
ENG
IN
10:38
23-01-2026


Lab Assignment 5.5
Lab 5 Ethical AI Practices
Untitled31.ipynb - Colab
colab.research.google.com/drive/1V94SXPmkmSltY9N3qEdY7IAJPTde_Odl#scrollTo=85804be9
All Bookmarks
Untitled31.ipynb
File Edit View Insert Runtime Tools Help
Commands + Code + Text Run all
Share
Performance Comparison
Let's compare the performance of these two methods using timeit for a larger prime number.
[3]
import timeit
number_to_check = 999999937 # A large prime number
# Measure performance of the basic method
time_basic = timeit.timeit(lambda: is_prime_basic(number_to_check), number=1)
print(f"Time taken by basic method: {time_basic:.6f} seconds")
# Measure performance of the optimized method
time_optimized = timeit.timeit(lambda: is_prime_optimized(number_to_check), number=1)
print(f"Time taken by optimized method: {time_optimized:.6f} seconds")
Time taken by basic method: 86.339446 seconds
Time taken by optimized method: 0.000890 seconds
Gemini
Generate Python code for two prime-checking methods and explain how the optimized version improves performance.
Certainly! Here's the Python code for two prime-checking methods—a basic approach and an optimized one—along with an explanation of how the optimized version enhances performance.
All the cells have executed successfully!
The basic prime-checking method correctly identified 17 as prime and 20 as not prime. The optimized method also produced the same correct results for these numbers.
The performance comparison clearly shows the advantage of the optimized method:
Basic Method: Took approximately 86.34 seconds to check the large prime number (999,999,937).
Optimized Method: Took only about 0.00089 seconds for the same number.
This demonstrates that the optimizations, particularly checking only up to the square root and skipping multiples of 2 and 3 (and using a step of 6), drastically
Test the optimized function with very large numbers
Explain how to further optimize the prime checking algorithm
Refactor the optimized prime checking code for better readability
What can I help you build?
Gemini 2.5 Flash
Gemini can make mistakes so double-check it and use code with caution. Learn more
Variables
Terminal
10:37 AM
Python 3
20°C
Sunny
Search
ENG
IN
10:38
23-01-2026

Task Description #2 (Transparency in Recursive Algorithms)

Objective: Use AI to generate a recursive function to calculate Fibonacci numbers.

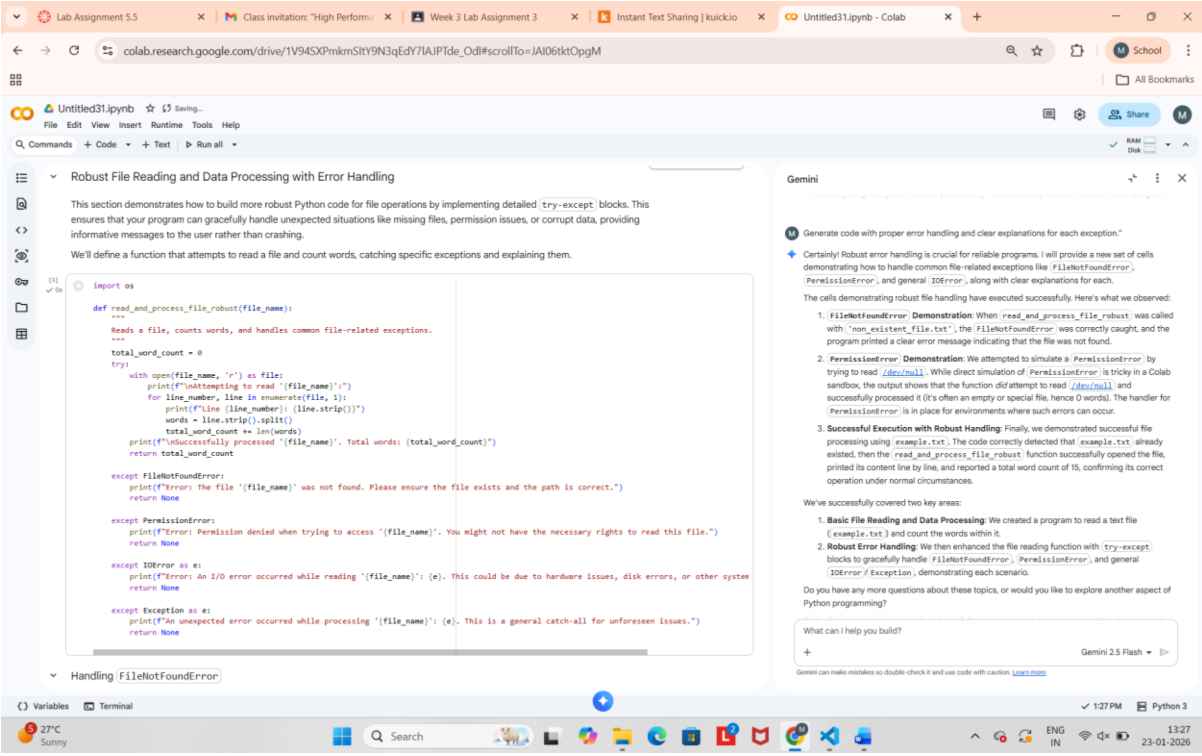


Task Description #3 (Transparency in Error Handling)

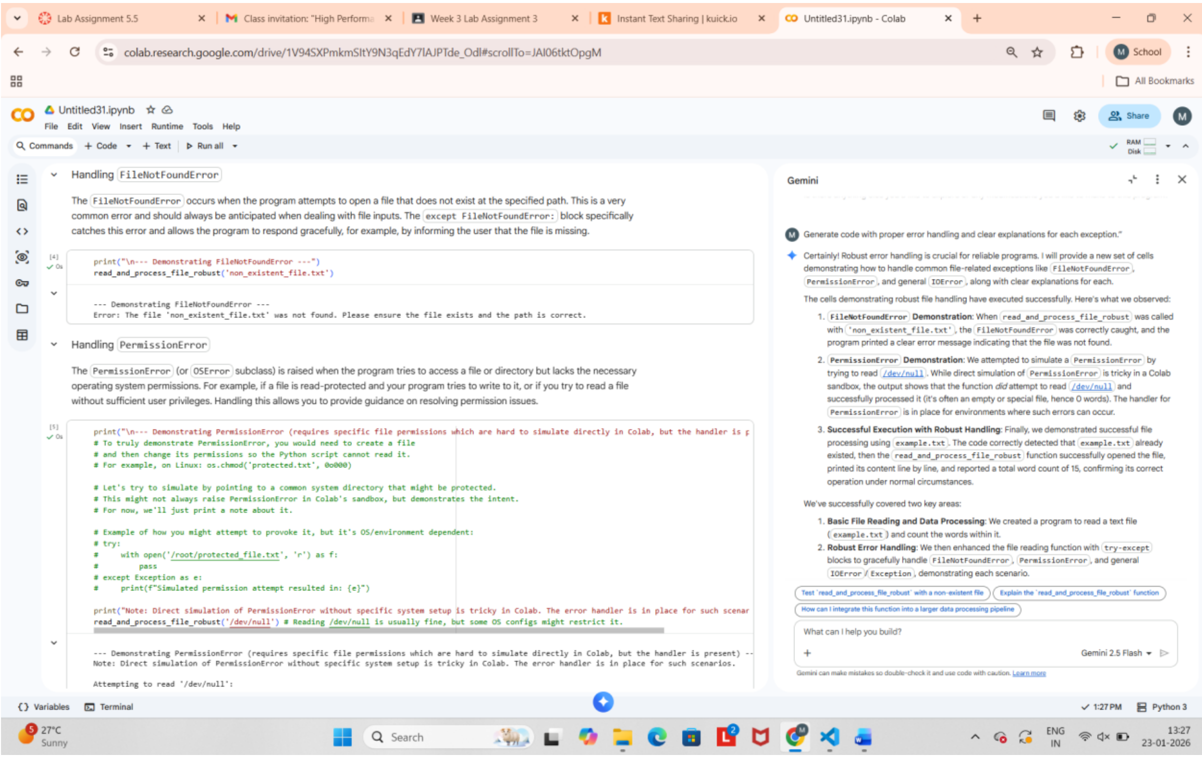
Task: Use AI to generate a Python program that reads a file and processes data.

Prompt:

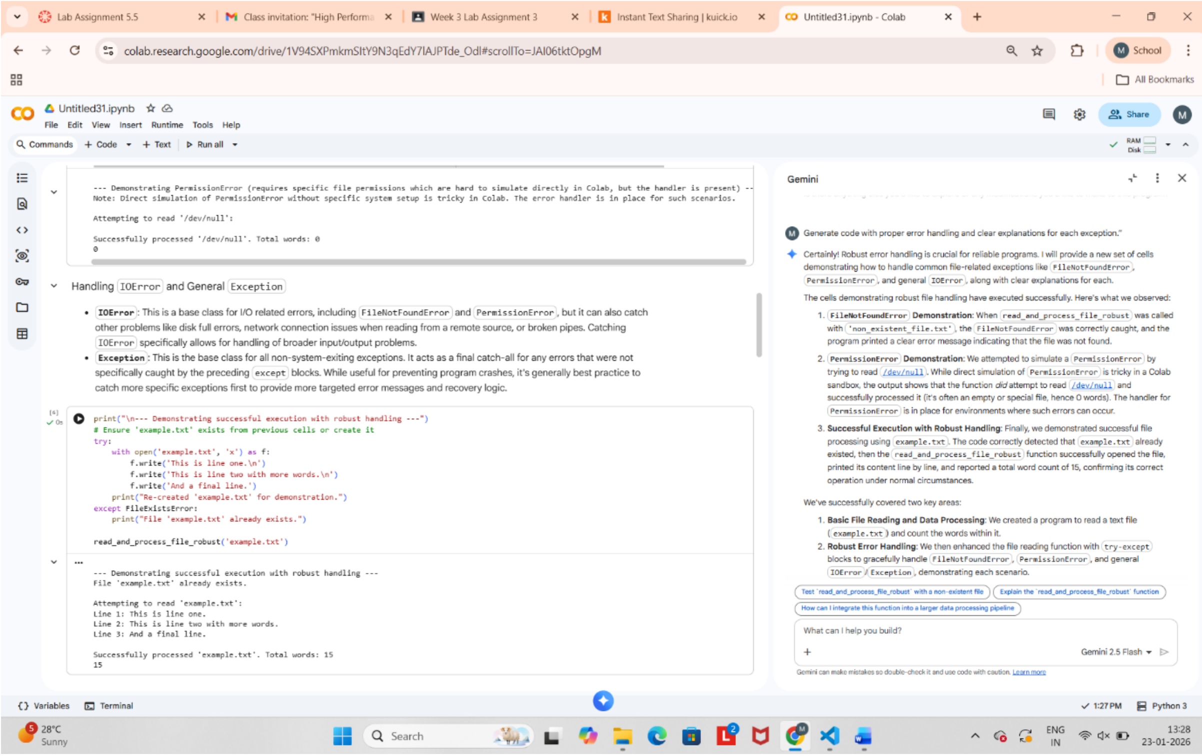
“Generate code with proper error handling and clear explanations for each exception.



Untitled31.ipynb
Robust File Reading and Data Processing with Error Handling
This section demonstrates how to build more robust Python code for file operations by implementing detailed try-except blocks. This ensures that your program can gracefully handle unexpected situations like missing files, permission issues, or corrupt data, providing informative messages to the user rather than crashing.
We'll define a function that attempts to read a file and count words, catching specific exceptions and explaining them.
import os
def read_and_process_file_robust(file_name):
"""
Reads a file, counts words, and handles common file-related exceptions.
"""
total_word_count = 0
try:
with open(file_name, 'r') as file:
print(f"\nAttempting to read '{file_name}':")
for line_number, line in enumerate(file, 1):
print(f"Line {line_number}: {line.strip()}")
words = line.strip().split()
total_word_count += len(words)
print(f"\nSuccessfully processed '{file_name}'. Total words: {total_word_count}")
return total_word_count
except FileNotFoundError:
print(f"Error: The file '{file_name}' was not found. Please ensure the file exists and the path is correct.")
return None
except PermissionError:
print(f"Error: Permission denied when trying to access '{file_name}'. You might not have the necessary rights to read this file.")
return None
except IOError as e:
print(f"Error: An I/O error occurred while reading '{file_name}': {e}. This could be due to hardware issues, disk errors, or other system
return None
except Exception as e:
print(f"An unexpected error occurred while processing '{file_name}': {e}. This is a general catch-all for unforeseen issues.")
return None
Handling FileNotFoundError
Gemini
Generate code with proper error handling and clear explanations for each exception."
Certainly! Robust error handling is crucial for reliable programs. I will provide a new set of cells demonstrating how to handle common file-related exceptions like FileNotFoundError, PermissionError, and general IOError, along with clear explanations for each.
The cells demonstrating robust file handling have executed successfully. Here's what we observed:
1. FileNotFoundError Demonstration: When read_and_process_file_robust was called with 'non_existent_file.txt', the FileNotFoundError was correctly caught, and the program printed a clear error message indicating that the file was not found.
2. PermissionError Demonstration: We attempted to simulate a PermissionError by trying to read /dev/null. While direct simulation of PermissionError is tricky in a Colab sandbox, the output shows that the function did attempt to read /dev/null and successfully processed it (it's often an empty or special file, hence 0 words). The handler for PermissionError is in place for environments where such errors can occur.
3. Successful Execution with Robust Handling: Finally, we demonstrated successful file processing using example.txt. The code correctly detected that example.txt already existed, then the read_and_process_file_robust function successfully opened the file, printed its content line by line, and reported a total word count of 15, confirming its correct operation under normal circumstances.
We've successfully covered two key areas:
1. Basic File Reading and Data Processing: We created a program to read a text file (example.txt) and count the words within it.
2. Robust Error Handling: We then enhanced the file reading function with try-except blocks to gracefully handle FileNotFoundError, PermissionError, and general IOError / Exception, demonstrating each scenario.
Do you have any more questions about these topics, or would you like to explore another aspect of Python programming?
What can I help you build?
Gemini 2.5 Flash
Gemini can make mistakes so double-check it and use code with caution. Learn more
Variables
Terminal
1:27 PM
Python 3
27°C Sunny
13:27
23-01-2026



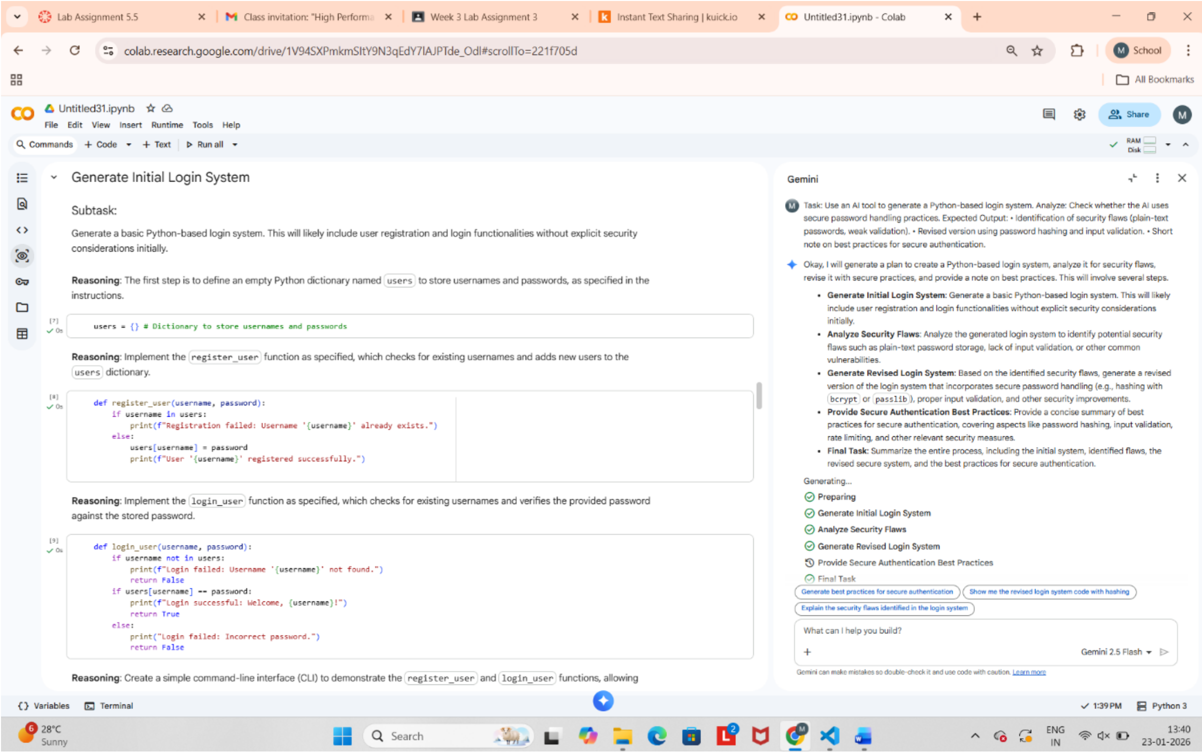
Untitled31.ipynb
Handling FileNotFoundError
The FileNotFoundError occurs when the program attempts to open a file that does not exist at the specified path. This is a very common error and should always be anticipated when dealing with file inputs. The except FileNotFoundError: block specifically catches this error and allows the program to respond gracefully, for example, by informing the user that the file is missing.
print("\n--- Demonstrating FileNotFoundError ---")
read_and_process_file_robust('non_existent_file.txt')
--- Demonstrating FileNotFoundError ---
Error: The file 'non_existent_file.txt' was not found. Please ensure the file exists and the path is correct.
Handling PermissionError
The PermissionError (or OSError subclass) is raised when the program tries to access a file or directory but lacks the necessary operating system permissions. For example, if a file is read-protected and your program tries to write to it, or if you try to read a file without sufficient user privileges. Handling this allows you to provide guidance on resolving permission issues.
print("\n--- Demonstrating PermissionError (requires specific file permissions which are hard to simulate directly in Colab, but the handler is p
To truly demonstrate PermissionError, you would need to create a file
and then change its permissions so the Python script cannot read it.
For example, on Linux: os.chmod('protected.txt', 0o000)
Let's try to simulate by pointing to a common system directory that might be protected.
This might not always raise PermissionError in Colab's sandbox, but demonstrates the intent.
For now, we'll just print a note about it.
Example of how you might attempt to provoke it, but it's OS/environment dependent:
try:
with open('/root/protected_file.txt', 'r') as f:
pass
except Exception as e:
print(f"Simulated permission attempt resulted in: {e}")
print("Note: Direct simulation of PermissionError without specific system setup is tricky in Colab. The error handler is in place for such scenar
read_and_process_file_robust('/dev/null') # Reading /dev/null is usually fine, but some OS configs might restrict it.
--- Demonstrating PermissionError (requires specific file permissions which are hard to simulate directly in Colab, but the handler is present) --
Note: Direct simulation of PermissionError without specific system setup is tricky in Colab. The error handler is in place for such scenarios.
Attempting to read '/dev/null':
Gemini
Test 'read_and_process_file_robust' with a non-existent file
Explain the 'read_and_process_file_robust' function
How can I integrate this function into a larger data processing pipeline
What can I help you build?
Gemini 2.5 Flash
1:27 PM
Python 3
13:27
23-01-2026



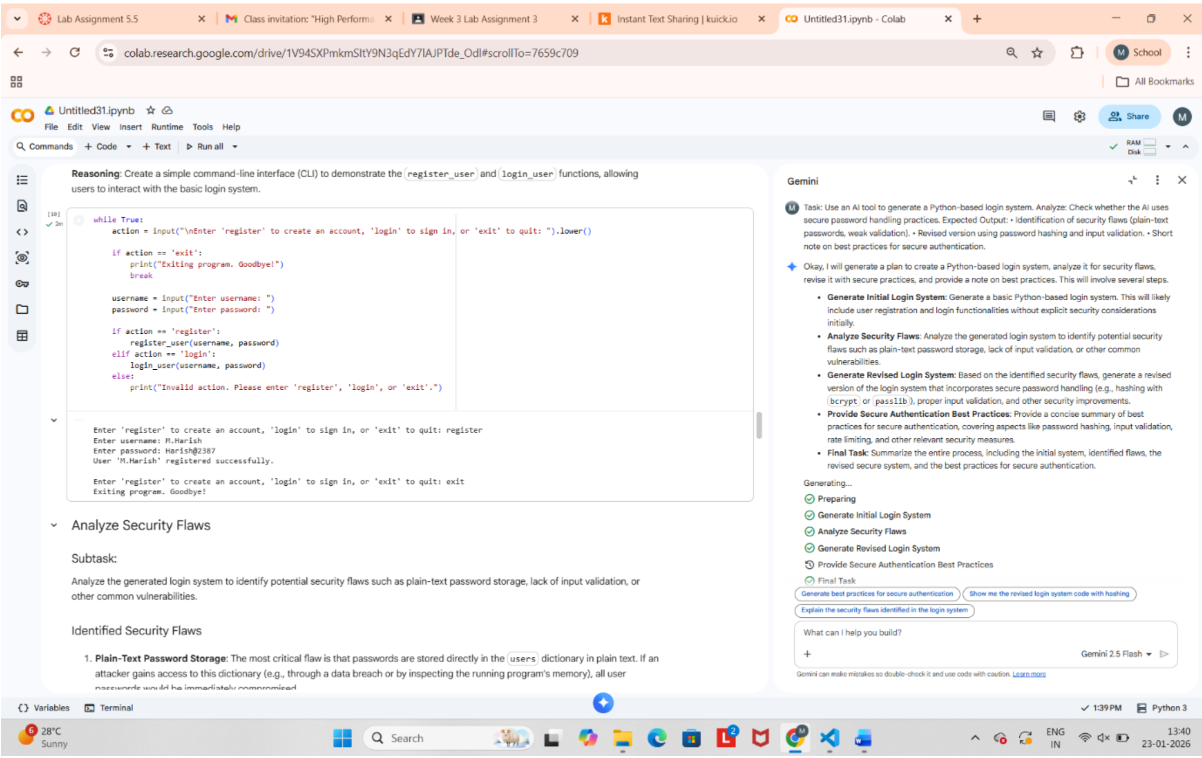
Task Description #4 (Security in User Authentication)

Task: Use an AI tool to generate a Python-based login system.

Analyze: Check whether the AI uses secure password handling practices.



colab.research.google.com/drive/1V94SXPmkmSItY9N3qEdY7lAJPTde_Odl#scrollTo=221f705d
Untitled31.ipynb
Generate Initial Login System
Subtask:
Generate a basic Python-based login system. This will likely include user registration and login functionalities without explicit security considerations initially.
Reasoning: The first step is to define an empty Python dictionary named users to store usernames and passwords, as specified in the instructions.
users = {} # Dictionary to store usernames and passwords
Reasoning: Implement the register_user function as specified, which checks for existing usernames and adds new users to the users dictionary.
def register_user(username, password):
    if username in users:
        print(f"Registration failed: Username '{username}' already exists.")
    else:
        users[username] = password
        print(f"User '{username}' registered successfully.")
Reasoning: Implement the login_user function as specified, which checks for existing usernames and verifies the provided password against the stored password.
def login_user(username, password):
    if username not in users:
        print(f"Login failed: Username '{username}' not found.")
        return False
    if users[username] == password:
        print(f"Login successful: Welcome, {username}!")
        return True
    else:
        print("Login failed: Incorrect password.")
        return False
Reasoning: Create a simple command-line interface (CLI) to demonstrate the register_user and login_user functions, allowing
Gemini
Task: Use an AI tool to generate a Python-based login system. Analyze: Check whether the AI uses secure password handling practices. Expected Output: • Identification of security flaws (plain-text passwords, weak validation). • Revised version using password hashing and input validation. • Short note on best practices for secure authentication.
Okay, I will generate a plan to create a Python-based login system, analyze it for security flaws, revise it with secure practices, and provide a note on best practices. This will involve several steps.
• Generate Initial Login System: Generate a basic Python-based login system. This will likely include user registration and login functionalities without explicit security considerations initially.
• Analyze Security Flaws: Analyze the generated login system to identify potential security flaws such as plain-text password storage, lack of input validation, or other common vulnerabilities.
• Generate Revised Login System: Based on the identified security flaws, generate a revised version of the login system that incorporates secure password handling (e.g., hashing with bcrypt or passlib), proper input validation, and other security improvements.
• Provide Secure Authentication Best Practices: Provide a concise summary of best practices for secure authentication, covering aspects like password hashing, input validation, rate limiting, and other relevant security measures.
• Final Task: Summarize the entire process, including the initial system, identified flaws, the revised secure system, and the best practices for secure authentication.
Generating...
Preparing
Generate Initial Login System
Analyze Security Flaws
Generate Revised Login System
Provide Secure Authentication Best Practices
Final Task
Generate best practices for secure authentication
Show me the revised login system code with hashing
Explain the security flaws identified in the login system
What can I help you build?
Gemini 2.5 Flash
Gemini can make mistakes so double-check it and use code with caution. Learn more
1:39 PM
Python 3
13:40
23-01-2026

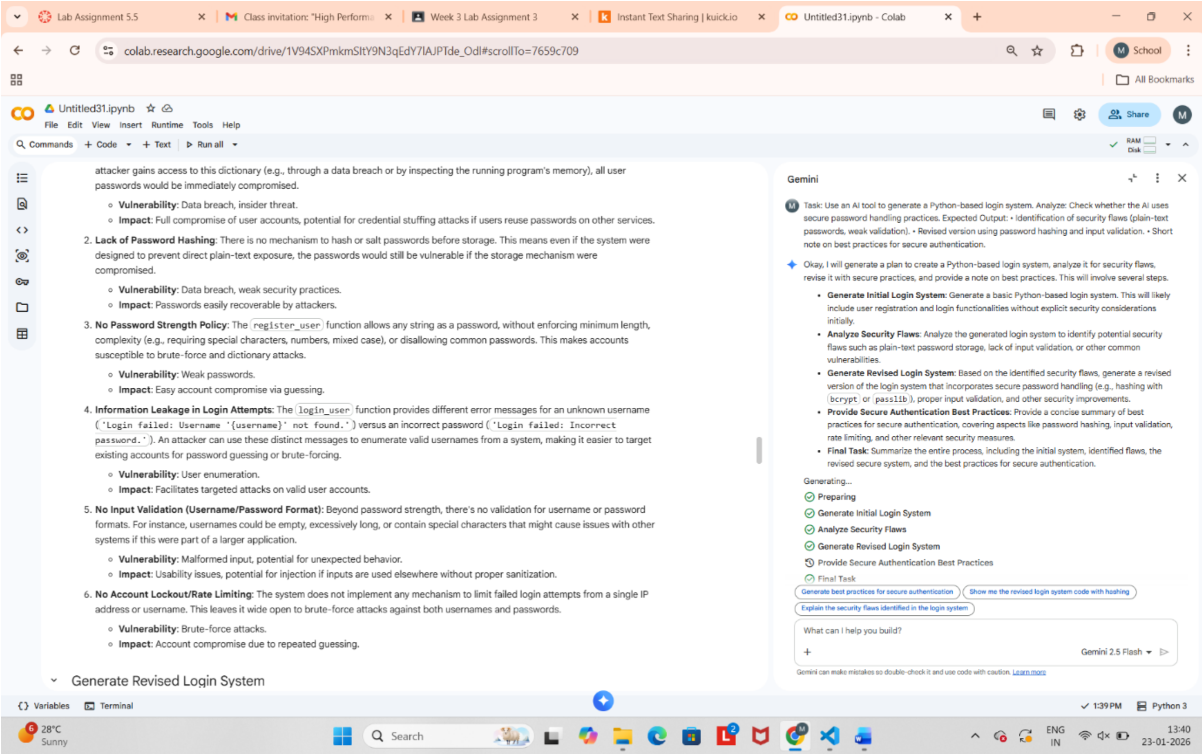


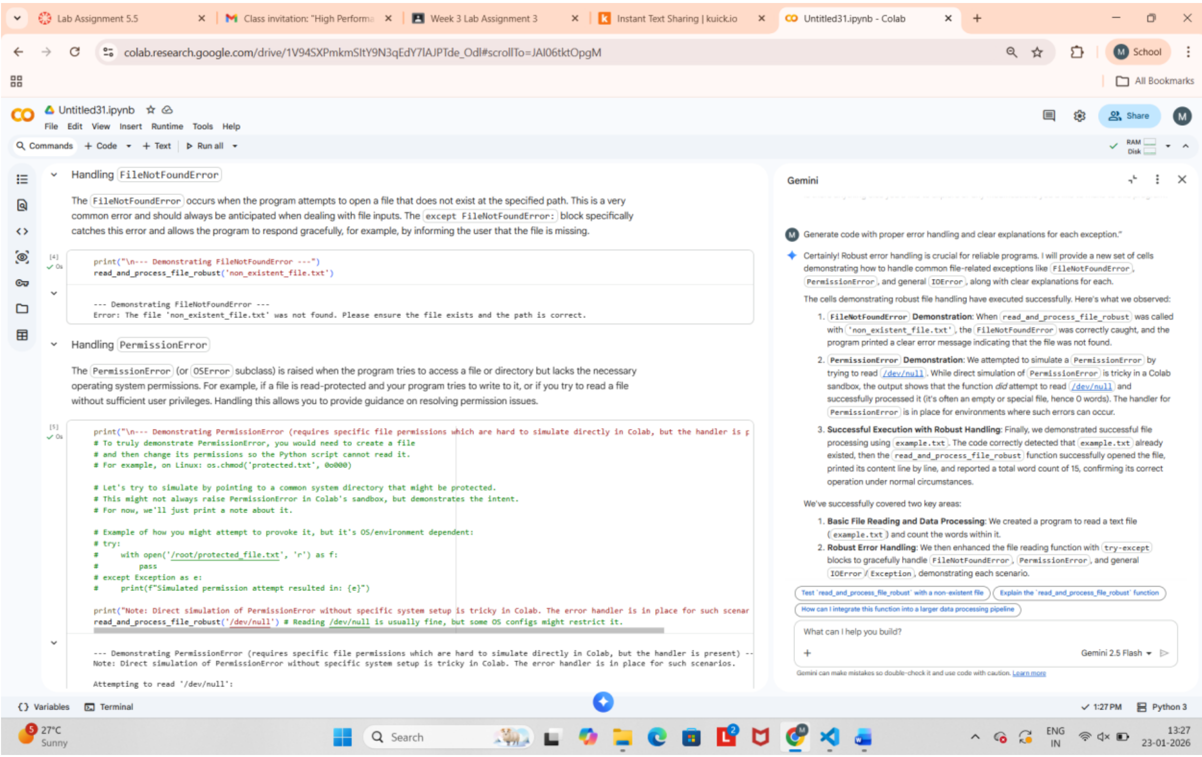
colab.research.google.com/drive/1V94SXPmkmSItY9N3qEdY7lAJPTde_Odl#scrollTo=7659c709
Untitled31.ipynb
Reasoning: Create a simple command-line interface (CLI) to demonstrate the register_user and login_user functions, allowing users to interact with the basic login system.
while True:
    action = input("\nEnter 'register' to create an account, 'login' to sign in, or 'exit' to quit: ").lower()

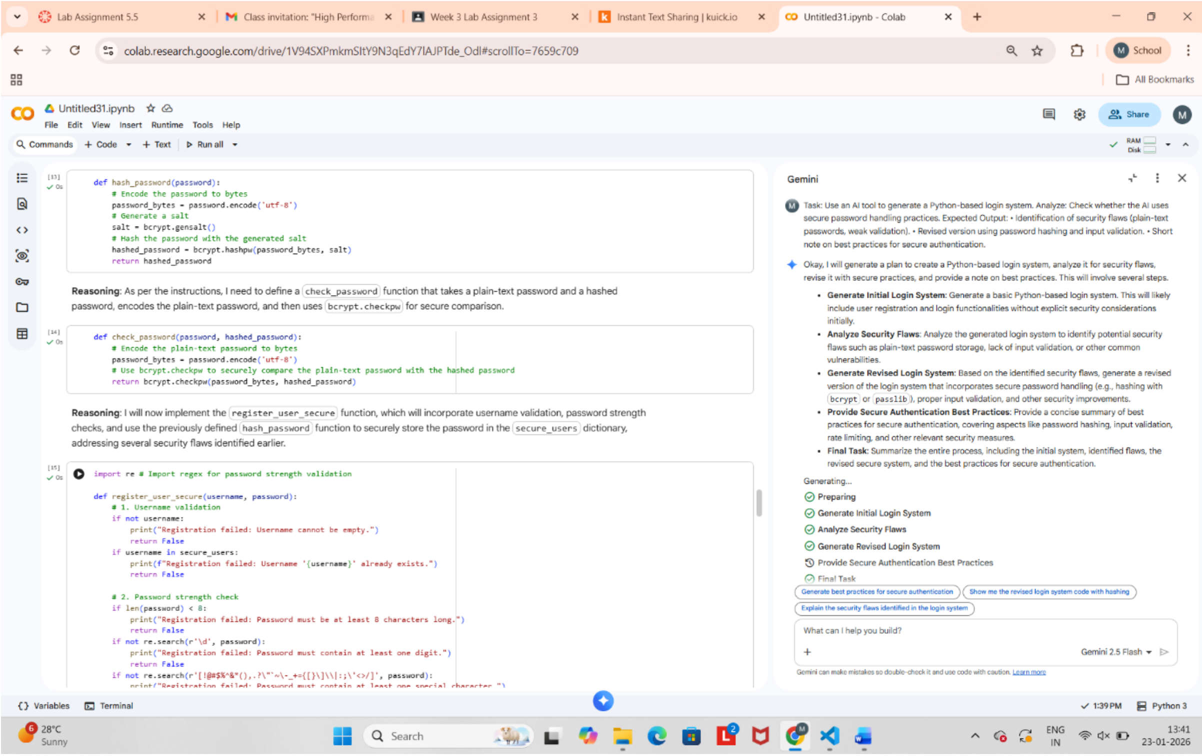
    if action == 'exit':
        print("Exiting program. Goodbye!")
        break

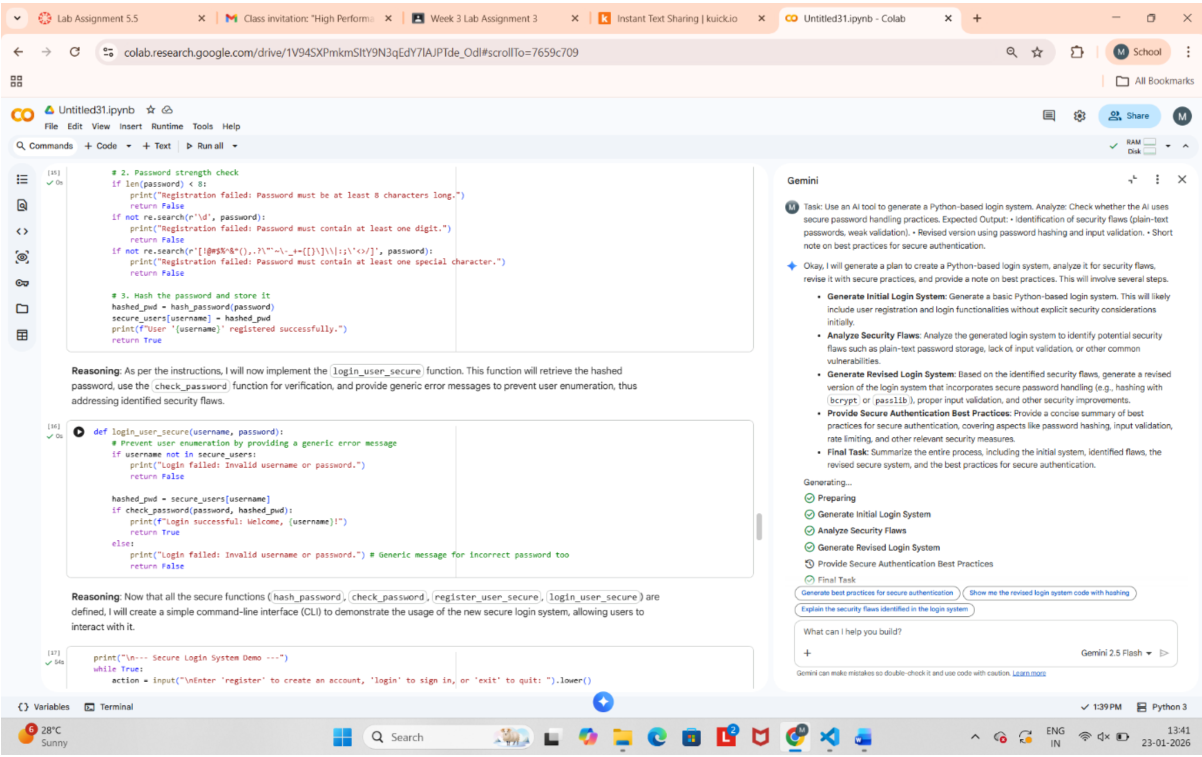
    username = input("Enter username: ")
    password = input("Enter password: ")

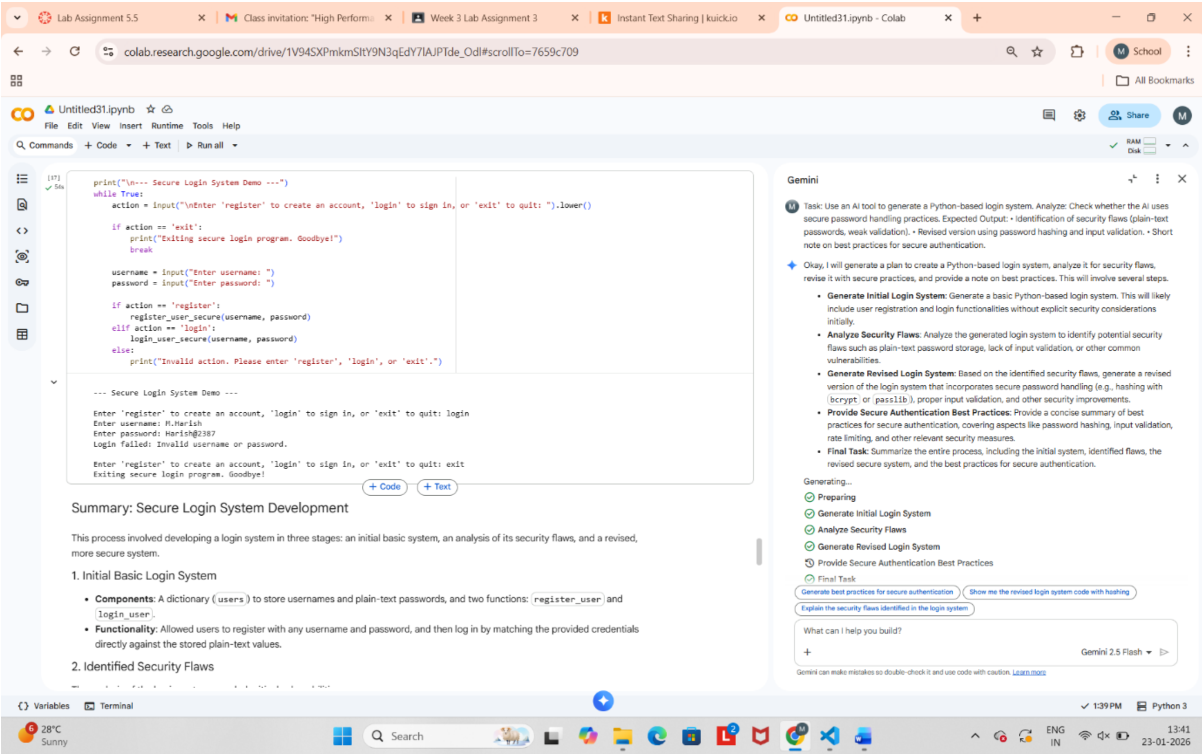
    if action == 'register':
        register_user(username, password)
    elif action == 'login':
        login_user(username, password)
    else:
        print("Invalid action. Please enter 'register', 'login', or 'exit'.")
Enter 'register' to create an account, 'login' to sign in, or 'exit' to quit: register
Enter username: M.Harish
Enter password: Harish@2387
User 'M.Harish' registered successfully.
Enter 'register' to create an account, 'login' to sign in, or 'exit' to quit: exit
Exiting program. Goodbye!
Analyze Security Flaws
Subtask:
Analyze the generated login system to identify potential security flaws such as plain-text password storage, lack of input validation, or other common vulnerabilities.
Identified Security Flaws
1. Plain-Text Password Storage: The most critical flaw is that passwords are stored directly in the users dictionary in plain text. If an attacker gains access to this dictionary (e.g., through a data breach or by inspecting the running program's memory), all user


Lab Assignment 5.5
Class invitation: "High Performa
Week 3 Lab Assignment 3
Instant Text Sharing | kuick.io
Untitled31.ipynb - Colab
colab.research.google.com/drive/1V94SXPmkmSItY9N3qEdY7IAJPTde_Odl#scrollTo=7659c709
School
All Bookmarks
Untitled31.ipynb
File Edit View Insert Runtime Tools Help
Commands + Code + Text Run all
attacker gains access to this dictionary (e.g., through a data breach or by inspecting the running program's memory), all user passwords would be immediately compromised.
Vulnerability: Data breach, insider threat.
Impact: Full compromise of user accounts, potential for credential stuffing attacks if users reuse passwords on other services.
2. Lack of Password Hashing: There is no mechanism to hash or salt passwords before storage. This means even if the system were designed to prevent direct plain-text exposure, the passwords would still be vulnerable if the storage mechanism were compromised.
Vulnerability: Data breach, weak security practices.
Impact: Passwords easily recoverable by attackers.
3. No Password Strength Policy: The register_user function allows any string as a password, without enforcing minimum length, complexity (e.g., requiring special characters, numbers, mixed case), or disallowing common passwords. This makes accounts susceptible to brute-force and dictionary attacks.
Vulnerability: Weak passwords.
Impact: Easy account compromise via guessing.
4. Information Leakage in Login Attempts: The login_user function provides different error messages for an unknown username ('Login failed: Username '{username}' not found.') versus an incorrect password ('Login failed: Incorrect password.'). An attacker can use these distinct messages to enumerate valid usernames from a system, making it easier to target existing accounts for password guessing or brute-forcing.
Vulnerability: User enumeration.
Impact: Facilitates targeted attacks on valid user accounts.
5. No Input Validation (Username/Password Format): Beyond password strength, there's no validation for username or password formats. For instance, usernames could be empty, excessively long, or contain special characters that might cause issues with other systems if this were part of a larger application.
Vulnerability: Malformed input, potential for unexpected behavior.
Impact: Usability issues, potential for injection if inputs are used elsewhere without proper sanitization.
6. No Account Lockout/Rate Limiting: The system does not implement any mechanism to limit failed login attempts from a single IP address or username. This leaves it wide open to brute-force attacks against both usernames and passwords.
Vulnerability: Brute-force attacks.
Impact: Account compromise due to repeated guessing.
Generate Revised Login System
Gemini
Task: Use an AI tool to generate a Python-based login system. Analyze: Check whether the AI uses secure password handling practices. Expected Output: • Identification of security flaws (plain-text passwords, weak validation). • Revised version using password hashing and input validation. • Short note on best practices for secure authentication.
Okay, I will generate a plan to create a Python-based login system, analyze it for security flaws, revise it with secure practices, and provide a note on best practices. This will involve several steps.
Generate Initial Login System: Generate a basic Python-based login system. This will likely include user registration and login functionalities without explicit security considerations initially.
Analyze Security Flaws: Analyze the generated login system to identify potential security flaws such as plain-text password storage, lack of input validation, or other common vulnerabilities.
Generate Revised Login System: Based on the identified security flaws, generate a revised version of the login system that incorporates secure password handling (e.g., hashing with bcrypt or passlib), proper input validation, and other security improvements.
Provide Secure Authentication Best Practices: Provide a concise summary of best practices for secure authentication, covering aspects like password hashing, input validation, rate limiting, and other relevant security measures.
Final Task: Summarize the entire process, including the initial system, identified flaws, the revised secure system, and the best practices for secure authentication.
Generating...
Preparing
Generate Initial Login System
Analyze Security Flaws
Generate Revised Login System
Provide Secure Authentication Best Practices
Final Task
Generate best practices for secure authentication
Show me the revised login system code with hashing
Explain the security flaws identified in the login system
What can I help you build?
Gemini 2.5 Flash
Gemini can make mistakes so double-check it and use code with caution. Learn more
Variables Terminal
1:39 PM Python 3
28°C Sunny
Search
ENG IN
13:40
23-01-2026

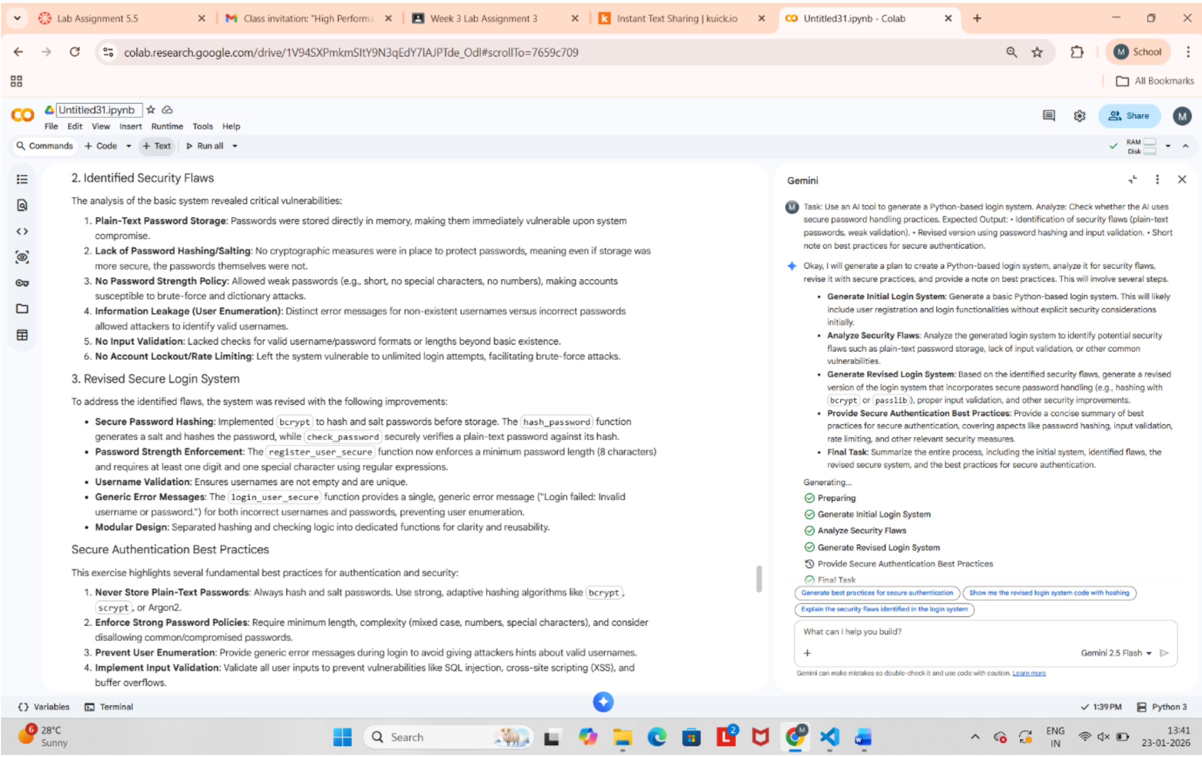

Lab Assignment 5.5
Class invitation: "High Performa
Week 3 Lab Assignment 3
Instant Text Sharing | kuick.io
Untitled31.ipynb - Colab
colab.research.google.com/drive/1V94SXPmkmSItY9N3qEdY7IAJPTde_Odl#scrollTo=JAI06tktOpgM
School
All Bookmarks
Untitled31.ipynb
File Edit View Insert Runtime Tools Help
Commands + Code + Text Run all
Handling FileNotFoundError
The FileNotFoundError occurs when the program attempts to open a file that does not exist at the specified path. This is a very common error and should always be anticipated when dealing with file inputs. The except FileNotFoundError: block specifically catches this error and allows the program to respond gracefully, for example, by informing the user that the file is missing.
[4]
print("\n--- Demonstrating FileNotFoundError ---")
read_and_process_file_robust('non_existent_file.txt')
--- Demonstrating FileNotFoundError ---
Error: The file 'non_existent_file.txt' was not found. Please ensure the file exists and the path is correct.
Handling PermissionError
The PermissionError (or OSError subclass) is raised when the program tries to access a file or directory but lacks the necessary operating system permissions. For example, if a file is read-protected and your program tries to write to it, or if you try to read a file without sufficient user privileges. Handling this allows you to provide guidance on resolving permission issues.
[5]
print("\n--- Demonstrating PermissionError (requires specific file permissions which are hard to simulate directly in Colab, but the handler is p
To truly demonstrate PermissionError, you would need to create a file
and then change its permissions so the Python script cannot read it.
For example, on Linux: os.chmod('protected.txt', 0o000)
Let's try to simulate by pointing to a common system directory that might be protected.
This might not always raise PermissionError in Colab's sandbox, but demonstrates the intent.
For now, we'll just print a note about it.
Example of how you might attempt to provoke it, but it's OS/environment dependent:
try:
with open('/root/protected_file.txt', 'r') as f:
pass
except Exception as e:
print(f"Simulated permission attempt resulted in: {e}")
print("Note: Direct simulation of PermissionError without specific system setup is tricky in Colab. The error handler is in place for such scenar
read_and_process_file_robust('/dev/null') # Reading /dev/null is usually fine, but some OS configs might restrict it.
--- Demonstrating PermissionError (requires specific file permissions which are hard to simulate directly in Colab, but the handler is present) --
Note: Direct simulation of PermissionError without specific system setup is tricky in Colab. The error handler is in place for such scenarios.
Attempting to read '/dev/null':
Gemini
Generate code with proper error handling and clear explanations for each exception."
Certainly! Robust error handling is crucial for reliable programs. I will provide a new set of cells demonstrating how to handle common file-related exceptions like FileNotFoundError, PermissionError, and general IOError, along with clear explanations for each.
The cells demonstrating robust file handling have executed successfully. Here's what we observed:
1. FileNotFoundError Demonstration: When read_and_process_file_robust was called with 'non_existent_file.txt', the FileNotFoundError was correctly caught, and the program printed a clear error message indicating that the file was not found.
2. PermissionError Demonstration: We attempted to simulate a PermissionError by trying to read /dev/null. While direct simulation of PermissionError is tricky in a Colab sandbox, the output shows that the function did attempt to read /dev/null and successfully processed it (it's often an empty or special file, hence 0 words). The handler for PermissionError is in place for environments where such errors can occur.
3. Successful Execution with Robust Handling: Finally, we demonstrated successful file processing using example.txt. The code correctly detected that example.txt already existed, then the read_and_process_file_robust function successfully opened the file, printed its content line by line, and reported a total word count of 15, confirming its correct operation under normal circumstances.
We've successfully covered two key areas:
1. Basic File Reading and Data Processing: We created a program to read a text file (example.txt) and count the words within it.
2. Robust Error Handling: We then enhanced the file reading function with try-except blocks to gracefully handle FileNotFoundError, PermissionError, and general IOError / Exception, demonstrating each scenario.
Test `read_and_process_file_robust` with a non-existent file
Explain the `read_and_process_file_robust` function
How can I integrate this function into a larger data processing pipeline
What can I help you build?
Gemini 2.5 Flash
Gemini can make mistakes so double-check it and use code with caution. Learn more
Variables Terminal
1:27 PM Python 3
27°C Sunny
Search
ENG IN
13:27
23-01-2026


Untitled31.ipynb
def hash_password(password):
# Encode the password to bytes
password_bytes = password.encode('utf-8')
# Generate a salt
salt = bcrypt.gensalt()
# Hash the password with the generated salt
hashed_password = bcrypt.hashpw(password_bytes, salt)
return hashed_password
Reasoning: As per the instructions, I need to define a check_password function that takes a plain-text password and a hashed password, encodes the plain-text password, and then uses bcrypt.checkpw for secure comparison.
def check_password(password, hashed_password):
# Encode the plain-text password to bytes
password_bytes = password.encode('utf-8')
# Use bcrypt.checkpw to securely compare the plain-text password with the hashed password
return bcrypt.checkpw(password_bytes, hashed_password)
Reasoning: I will now implement the register_user_secure function, which will incorporate username validation, password strength checks, and use the previously defined hash_password function to securely store the password in the secure_users dictionary, addressing several security flaws identified earlier.
import re # Import regex for password strength validation
def register_user_secure(username, password):
# 1. Username validation
if not username:
print("Registration failed: Username cannot be empty.")
return False
if username in secure_users:
print(f"Registration failed: Username '{username}' already exists.")
return False
# 2. Password strength check
if len(password) < 8:
print("Registration failed: Password must be at least 8 characters long.")
return False
if not re.search(r'\d', password):
print("Registration failed: Password must contain at least one digit.")
return False
if not re.search(r'[!@#$%^&*(),.?\"`~\-_+={[}\]\\|:;\'<>/]', password):

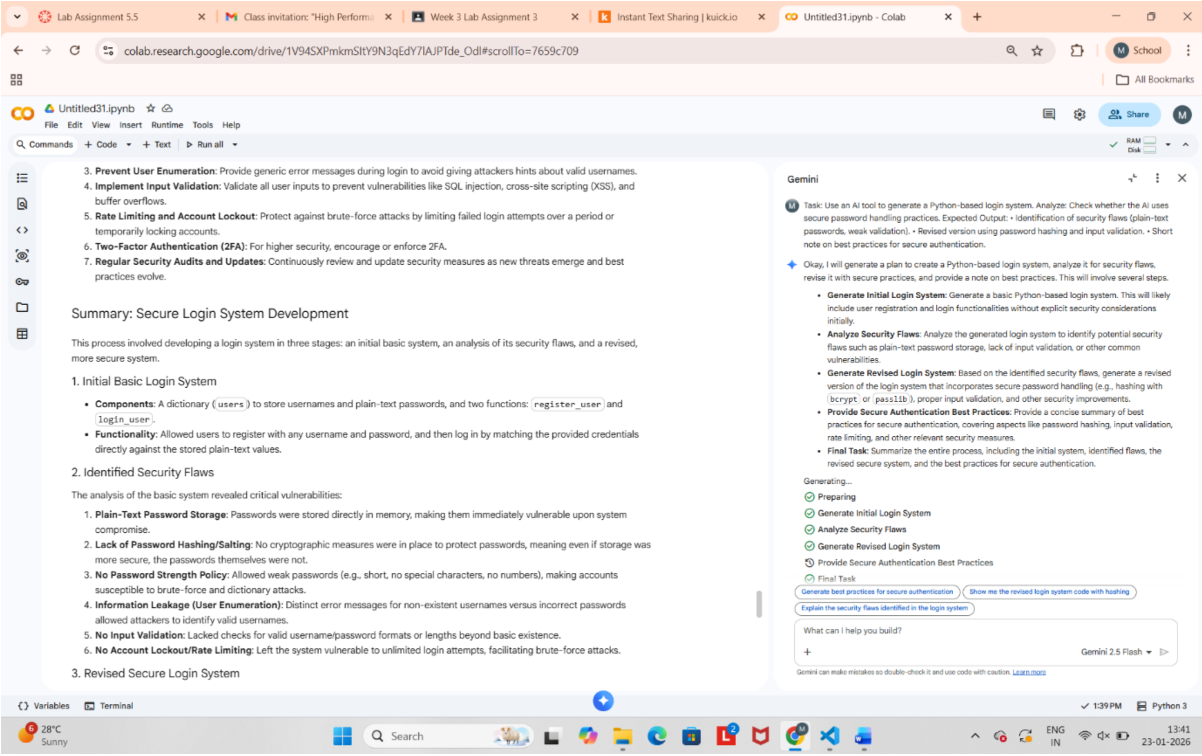

Untitled31.ipynb
# 2. Password strength check
if len(password) < 8:
print("Registration failed: Password must be at least 8 characters long.")
return False
if not re.search(r'\d', password):
print("Registration failed: Password must contain at least one digit.")
return False
if not re.search(r'[!@#$%^&*(),.?\"`~\-_+={[}\]\\|:;\'<>/]', password):
print("Registration failed: Password must contain at least one special character.")
return False
# 3. Hash the password and store it
hashed_pwd = hash_password(password)
secure_users[username] = hashed_pwd
print(f"User '{username}' registered successfully.")
return True
Reasoning: As per the instructions, I will now implement the login_user_secure function. This function will retrieve the hashed password, use the check_password function for verification, and provide generic error messages to prevent user enumeration, thus addressing identified security flaws.
def login_user_secure(username, password):
# Prevent user enumeration by providing a generic error message
if username not in secure_users:
print("Login failed: Invalid username or password.")
return False
hashed_pwd = secure_users[username]
if check_password(password, hashed_pwd):
print(f"Login successful: Welcome, {username}!")
return True
else:
print("Login failed: Invalid username or password.") # Generic message for incorrect password too
return False
Reasoning: Now that all the secure functions (hash_password, check_password, register_user_secure, login_user_secure) are defined, I will create a simple command-line interface (CLI) to demonstrate the usage of the new secure login system, allowing users to interact with it.
print("\n--- Secure Login System Demo ---")
while True:
action = input("\nEnter 'register' to create an account, 'login' to sign in, or 'exit' to quit: ").lower()

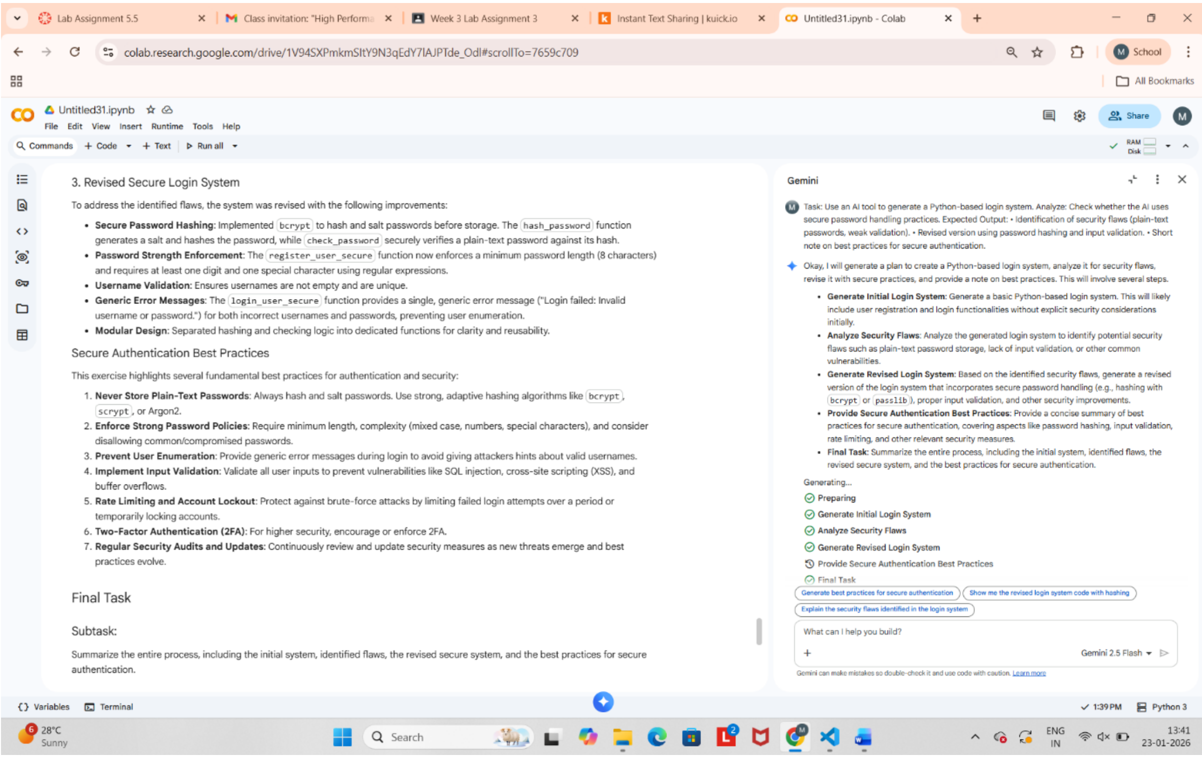


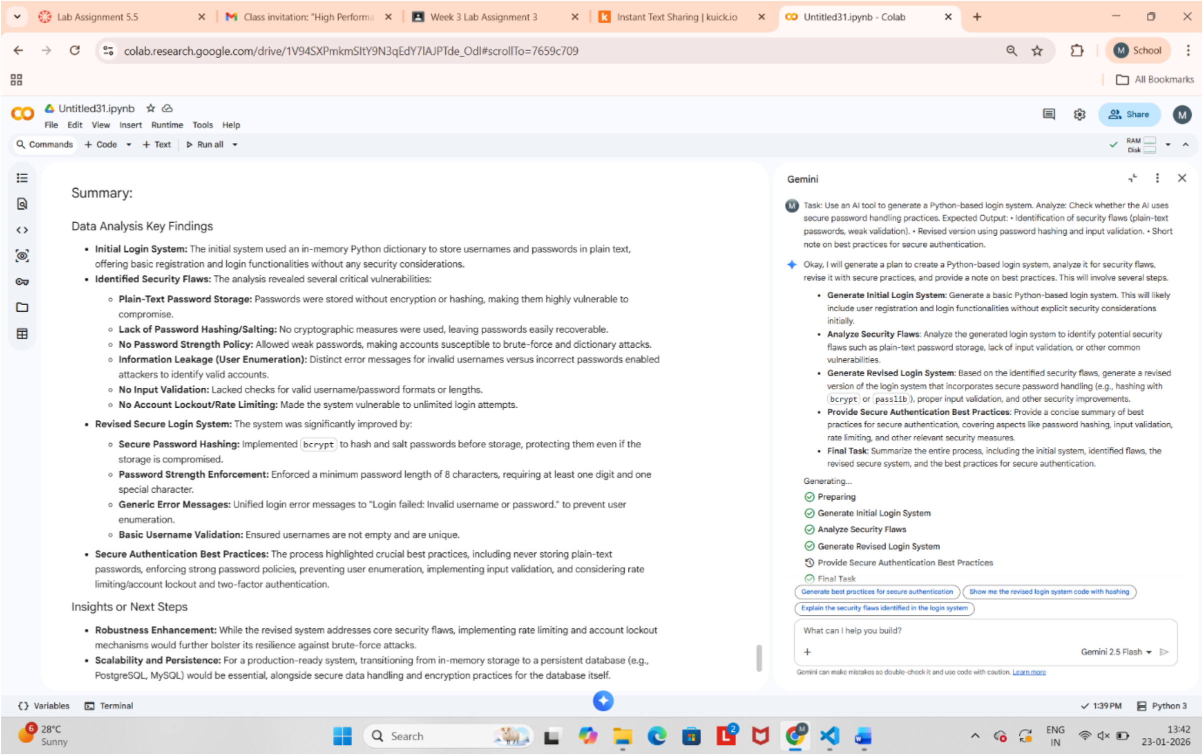
Untitled31.ipynb
print("\n--- Secure Login System Demo ---")
while True:
action = input("\nEnter 'register' to create an account, 'login' to sign in, or 'exit' to quit: ").lower()
if action == 'exit':
print("Exiting secure login program. Goodbye!")
break
username = input("Enter username: ")
password = input("Enter password: ")
if action == 'register':
register_user_secure(username, password)
elif action == 'login':
login_user_secure(username, password)
else:
print("Invalid action. Please enter 'register', 'login', or 'exit'.")
--- Secure Login System Demo ---
Enter 'register' to create an account, 'login' to sign in, or 'exit' to quit: login
Enter username: M.Harish
Enter password: Harish@2387
Login failed: Invalid username or password.
Enter 'register' to create an account, 'login' to sign in, or 'exit' to quit: exit
Exiting secure login program. Goodbye!
Summary: Secure Login System Development
This process involved developing a login system in three stages: an initial basic system, an analysis of its security flaws, and a revised, more secure system.
1. Initial Basic Login System
Components: A dictionary (users) to store usernames and plain-text passwords, and two functions: register_user and login_user.
Functionality: Allowed users to register with any username and password, and then log in by matching the provided credentials directly against the stored plain-text values.
2. Identified Security Flaws



Untitled31.ipynb
2. Identified Security Flaws
The analysis of the basic system revealed critical vulnerabilities:
1. Plain-Text Password Storage: Passwords were stored directly in memory, making them immediately vulnerable upon system compromise.
2. Lack of Password Hashing/Salting: No cryptographic measures were in place to protect passwords, meaning even if storage was more secure, the passwords themselves were not.
3. No Password Strength Policy: Allowed weak passwords (e.g., short, no special characters, no numbers), making accounts susceptible to brute-force and dictionary attacks.
4. Information Leakage (User Enumeration): Distinct error messages for non-existent usernames versus incorrect passwords allowed attackers to identify valid usernames.
5. No Input Validation: Lacked checks for valid username/password formats or lengths beyond basic existence.
6. No Account Lockout/Rate Limiting: Left the system vulnerable to unlimited login attempts, facilitating brute-force attacks.
3. Revised Secure Login System
To address the identified flaws, the system was revised with the following improvements:
Secure Password Hashing: Implemented bcrypt to hash and salt passwords before storage. The hash_password function generates a salt and hashes the password, while check_password securely verifies a plain-text password against its hash.
Password Strength Enforcement: The register_user_secure function now enforces a minimum password length (8 characters) and requires at least one digit and one special character using regular expressions.
Username Validation: Ensures usernames are not empty and are unique.
Generic Error Messages: The login_user_secure function provides a single, generic error message ("Login failed: Invalid username or password.") for both incorrect usernames and passwords, preventing user enumeration.
Modular Design: Separated hashing and checking logic into dedicated functions for clarity and reusability.
Secure Authentication Best Practices
This exercise highlights several fundamental best practices for authentication and security:
1. Never Store Plain-Text Passwords: Always hash and salt passwords. Use strong, adaptive hashing algorithms like bcrypt, scrypt, or Argon2.
2. Enforce Strong Password Policies: Require minimum length, complexity (mixed case, numbers, special characters), and consider disallowing common/compromised passwords.
3. Prevent User Enumeration: Provide generic error messages during login to avoid giving attackers hints about valid usernames.
4. Implement Input Validation: Validate all user inputs to prevent vulnerabilities like SQL injection, cross-site scripting (XSS), and buffer overflows.


Untitled31.ipynb
3. Prevent User Enumeration: Provide generic error messages during login to avoid giving attackers hints about valid usernames.
4. Implement Input Validation: Validate all user inputs to prevent vulnerabilities like SQL injection, cross-site scripting (XSS), and buffer overflows.
5. Rate Limiting and Account Lockout: Protect against brute-force attacks by limiting failed login attempts over a period or temporarily locking accounts.
6. Two-Factor Authentication (2FA): For higher security, encourage or enforce 2FA.
7. Regular Security Audits and Updates: Continuously review and update security measures as new threats emerge and best practices evolve.
Summary: Secure Login System Development
This process involved developing a login system in three stages: an initial basic system, an analysis of its security flaws, and a revised, more secure system.
1. Initial Basic Login System
Components: A dictionary (users) to store usernames and plain-text passwords, and two functions: register_user and login_user.
Functionality: Allowed users to register with any username and password, and then log in by matching the provided credentials directly against the stored plain-text values.
2. Identified Security Flaws
The analysis of the basic system revealed critical vulnerabilities:
1. Plain-Text Password Storage: Passwords were stored directly in memory, making them immediately vulnerable upon system compromise.
2. Lack of Password Hashing/Salting: No cryptographic measures were in place to protect passwords, meaning even if storage was more secure, the passwords themselves were not.
3. No Password Strength Policy: Allowed weak passwords (e.g., short, no special characters, no numbers), making accounts susceptible to brute-force and dictionary attacks.
4. Information Leakage (User Enumeration): Distinct error messages for non-existent usernames versus incorrect passwords allowed attackers to identify valid usernames.
5. No Input Validation: Lacked checks for valid username/password formats or lengths beyond basic existence.
6. No Account Lockout/Rate Limiting: Left the system vulnerable to unlimited login attempts, facilitating brute-force attacks.
3. Revised Secure Login System
Gemini
Task: Use an AI tool to generate a Python-based login system. Analyze: Check whether the AI uses secure password handling practices. Expected Output: • Identification of security flaws (plain-text passwords, weak validation). • Revised version using password hashing and input validation. • Short note on best practices for secure authentication.
Okay, I will generate a plan to create a Python-based login system, analyze it for security flaws, revise it with secure practices, and provide a note on best practices. This will involve several steps.
Generate Initial Login System: Generate a basic Python-based login system. This will likely include user registration and login functionalities without explicit security considerations initially.
Analyze Security Flaws: Analyze the generated login system to identify potential security flaws such as plain-text password storage, lack of input validation, or other common vulnerabilities.
Generate Revised Login System: Based on the identified security flaws, generate a revised version of the login system that incorporates secure password handling (e.g., hashing with bcrypt or passlib), proper input validation, and other security improvements.
Provide Secure Authentication Best Practices: Provide a concise summary of best practices for secure authentication, covering aspects like password hashing, input validation, rate limiting, and other relevant security measures.
Final Task: Summarize the entire process, including the initial system, identified flaws, the revised secure system, and the best practices for secure authentication.
Generating...
Preparing
Generate Initial Login System
Analyze Security Flaws
Generate Revised Login System
Provide Secure Authentication Best Practices
Final Task
Generate best practices for secure authentication
Show me the revised login system code with hashing
Explain the security flaws identified in the login system
What can I help you build?
Gemini 2.5 Flash
Gemini can make mistakes so double-check it and use code with caution. Learn more

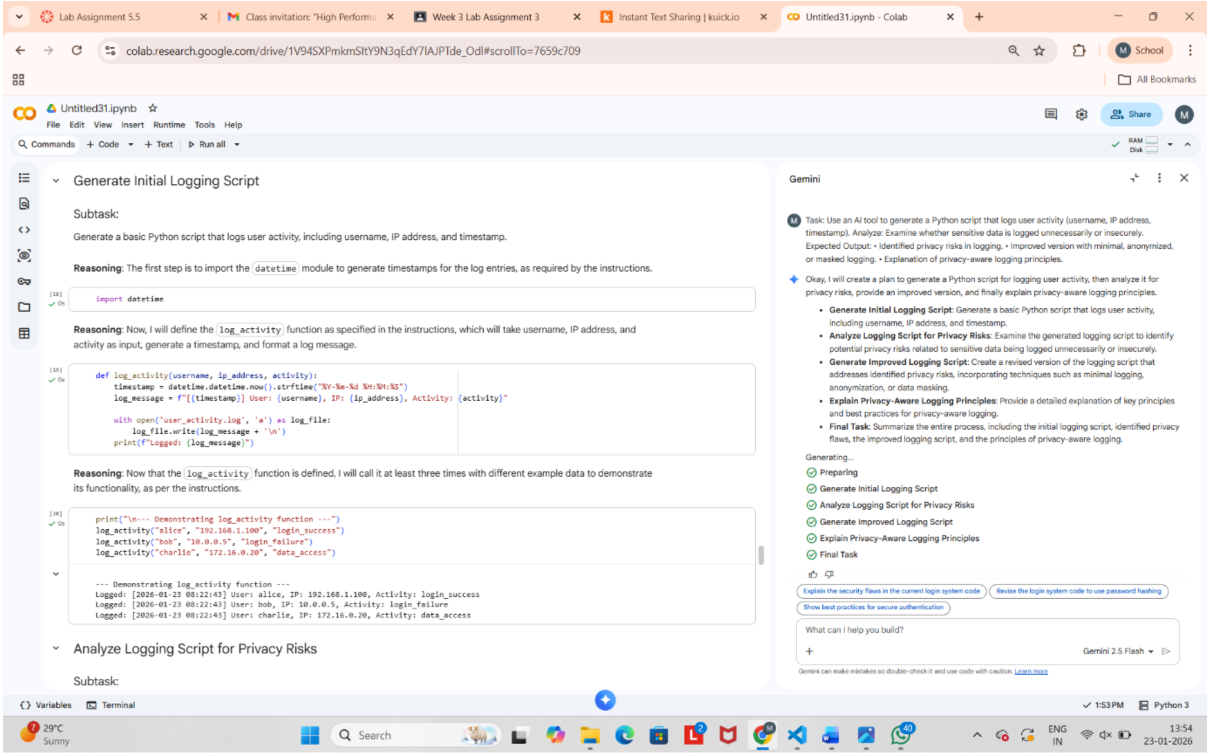

Untitled31.ipynb
3. Revised Secure Login System
To address the identified flaws, the system was revised with the following improvements:
Secure Password Hashing: Implemented bcrypt to hash and salt passwords before storage. The hash_password function generates a salt and hashes the password, while check_password securely verifies a plain-text password against its hash.
Password Strength Enforcement: The register_user_secure function now enforces a minimum password length (8 characters) and requires at least one digit and one special character using regular expressions.
Username Validation: Ensures usernames are not empty and are unique.
Generic Error Messages: The login_user_secure function provides a single, generic error message ("Login failed: Invalid username or password.") for both incorrect usernames and passwords, preventing user enumeration.
Modular Design: Separated hashing and checking logic into dedicated functions for clarity and reusability.
Secure Authentication Best Practices
This exercise highlights several fundamental best practices for authentication and security:
1. Never Store Plain-Text Passwords: Always hash and salt passwords. Use strong, adaptive hashing algorithms like bcrypt, scrypt, or Argon2.
2. Enforce Strong Password Policies: Require minimum length, complexity (mixed case, numbers, special characters), and consider disallowing common/compromised passwords.
3. Prevent User Enumeration: Provide generic error messages during login to avoid giving attackers hints about valid usernames.
4. Implement Input Validation: Validate all user inputs to prevent vulnerabilities like SQL injection, cross-site scripting (XSS), and buffer overflows.
5. Rate Limiting and Account Lockout: Protect against brute-force attacks by limiting failed login attempts over a period or temporarily locking accounts.
6. Two-Factor Authentication (2FA): For higher security, encourage or enforce 2FA.
7. Regular Security Audits and Updates: Continuously review and update security measures as new threats emerge and best practices evolve.
Final Task
Subtask:
Summarize the entire process, including the initial system, identified flaws, the revised secure system, and the best practices for secure authentication.
Gemini
Task: Use an AI tool to generate a Python-based login system. Analyze: Check whether the AI uses secure password handling practices. Expected Output: • Identification of security flaws (plain-text passwords, weak validation). • Revised version using password hashing and input validation. • Short note on best practices for secure authentication.
Okay, I will generate a plan to create a Python-based login system, analyze it for security flaws, revise it with secure practices, and provide a note on best practices. This will involve several steps.
Generate Initial Login System: Generate a basic Python-based login system. This will likely include user registration and login functionalities without explicit security considerations initially.
Analyze Security Flaws: Analyze the generated login system to identify potential security flaws such as plain-text password storage, lack of input validation, or other common vulnerabilities.
Generate Revised Login System: Based on the identified security flaws, generate a revised version of the login system that incorporates secure password handling (e.g., hashing with bcrypt or passlib), proper input validation, and other security improvements.
Provide Secure Authentication Best Practices: Provide a concise summary of best practices for secure authentication, covering aspects like password hashing, input validation, rate limiting, and other relevant security measures.
Final Task: Summarize the entire process, including the initial system, identified flaws, the revised secure system, and the best practices for secure authentication.
Generating...
Preparing
Generate Initial Login System
Analyze Security Flaws
Generate Revised Login System
Provide Secure Authentication Best Practices
Final Task
What can I help you build?
Gemini 2.5 Flash

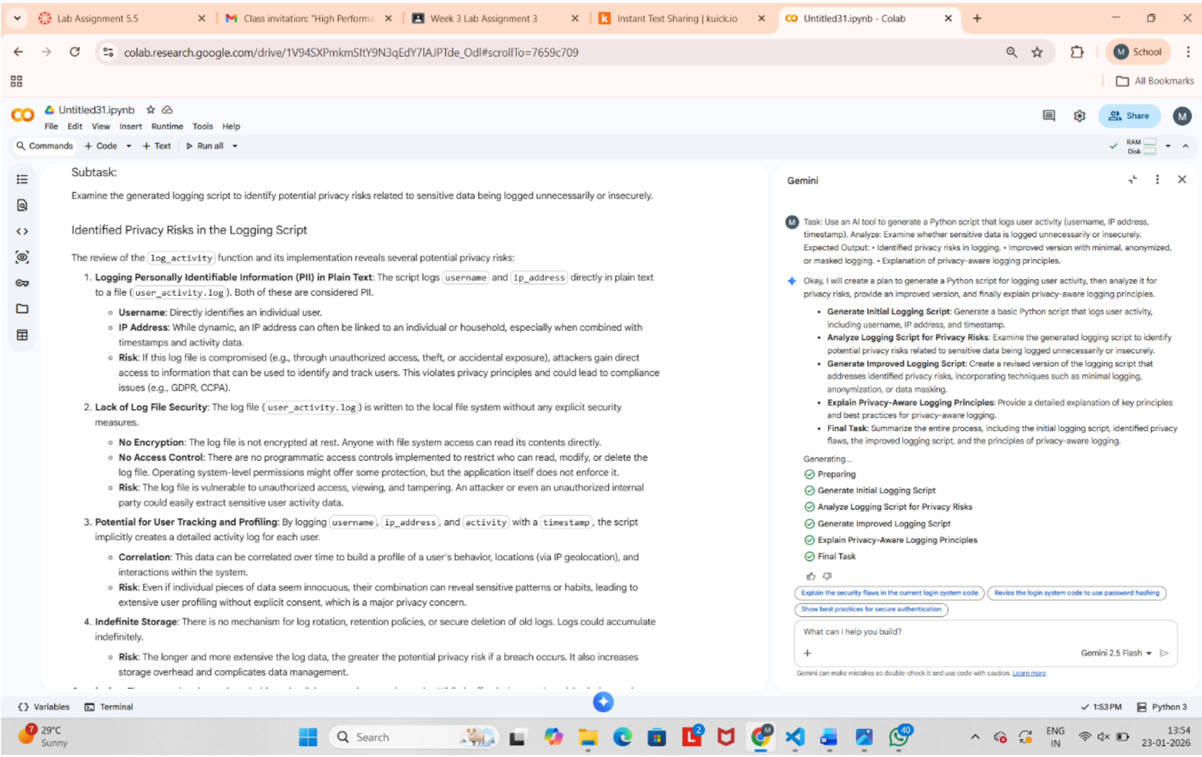


Task Description #5 (Privacy in Data Logging)

Task: Use an AI tool to generate a Python script that logs user activity (username, IP address, timestamp).

Analyze: Examine whether sensitive data is logged unnecessarily or insecurely.


Generate Initial Logging Script
Subtask:
Generate a basic Python script that logs user activity, including username, IP address, and timestamp.
Reasoning: The first step is to import the datetime module to generate timestamps for the log entries, as required by the instructions.
import datetime
Reasoning: Now, I will define the log_activity function as specified in the instructions, which will take username, IP address, and activity as input, generate a timestamp, and format a log message.
def log_activity(username, ip_address, activity):
timestamp = datetime.datetime.now().strftime("%Y-%m-%d %H:%M:%S")
log_message = f"[{timestamp}] User: {username}, IP: {ip_address}, Activity: {activity}"
with open('user_activity.log', 'a') as log_file:
log_file.write(log_message + '\n')
print(f"Logged: {log_message}")
Reasoning: Now that the log_activity function is defined, I will call it at least three times with different example data to demonstrate its functionality, as per the instructions.
print("\n--- Demonstrating log_activity function ---")
log_activity("alice", "192.168.1.100", "login_success")
log_activity("bob", "10.0.0.5", "login_failure")
log_activity("charlie", "172.16.0.20", "data_access")
--- Demonstrating log_activity function ---
Logged: [2026-01-23 08:22:43] User: alice, IP: 192.168.1.100, Activity: login_success
Logged: [2026-01-23 08:22:43] User: bob, IP: 10.0.0.5, Activity: login_failure
Logged: [2026-01-23 08:22:43] User: charlie, IP: 172.16.0.20, Activity: data_access
Analyze Logging Script for Privacy Risks
Subtask:


Subtask:
Examine the generated logging script to identify potential privacy risks related to sensitive data being logged unnecessarily or insecurely.
Identified Privacy Risks in the Logging Script
The review of the log_activity function and its implementation reveals several potential privacy risks:
1. Logging Personally Identifiable Information (PII) in Plain Text: The script logs username and ip_address directly in plain text to a file (user_activity.log). Both of these are considered PII.
Username: Directly identifies an individual user.
IP Address: While dynamic, an IP address can often be linked to an individual or household, especially when combined with timestamps and activity data.
Risk: If this log file is compromised (e.g., through unauthorized access, theft, or accidental exposure), attackers gain direct access to information that can be used to identify and track users. This violates privacy principles and could lead to compliance issues (e.g., GDPR, CCPA).
2. Lack of Log File Security: The log file (user_activity.log) is written to the local file system without any explicit security measures.
No Encryption: The log file is not encrypted at rest. Anyone with file system access can read its contents directly.
No Access Control: There are no programmatic access controls implemented to restrict who can read, modify, or delete the log file. Operating system-level permissions might offer some protection, but the application itself does not enforce it.
Risk: The log file is vulnerable to unauthorized access, viewing, and tampering. An attacker or even an unauthorized internal party could easily extract sensitive user activity data.
3. Potential for User Tracking and Profiling: By logging username, ip_address, and activity with a timestamp, the script implicitly creates a detailed activity log for each user.
Correlation: This data can be correlated over time to build a profile of a user's behavior, locations (via IP geolocation), and interactions within the system.
Risk: Even if individual pieces of data seem innocuous, their combination can reveal sensitive patterns or habits, leading to extensive user profiling without explicit consent, which is a major privacy concern.
4. Indefinite Storage: There is no mechanism for log rotation, retention policies, or secure deletion of old logs. Logs could accumulate indefinitely.
Risk: The longer and more extensive the log data, the greater the potential privacy risk if a breach occurs. It also increases storage overhead and complicates data management.

- **Risk**: The longer and more extensive the log data, the greater the potential privacy risk if a breach occurs. It also increases storage overhead and complicates data management.

Conclusion: The current logging script prioritizes simplicity over privacy and security. While it effectively records activity, it does so in a manner that poses significant risks to user privacy due to plain-text PII storage, lack of file security, and potential for detailed user tracking.

Generate Improved Logging Script

Subtask:

Create a revised version of the logging script that addresses identified privacy risks, incorporating techniques such as minimal logging, anonymization, or data masking.

Reasoning: The first step in generating an improved logging script is to import the `hashlib` module, as specified in the instructions, which will be used for hashing usernames to ensure privacy.

```
import hashlib
print("Imported hashlib for secure logging.")
```

```
Imported hashlib for secure logging.
```

Reasoning: Now, I will define the `log_activity_secure` function, incorporating the generation of a timestamp, hashing of the username, and masking of the IP address, as well as writing the secure log message to a file, as specified in the instructions.

```
import datetime # Already imported, but ensuring availability

def log_activity_secure(username, ip_address, activity):
    # a. Generate a timestamp
    timestamp = datetime.datetime.now().strftime("%Y-%m-%d %H:%M:%S")

    # b. Hash the username
    hashed_username = hashlib.sha256(username.encode('utf-8')).hexdigest()

    # c. Mask the IP address
    ip_parts = ip_address.split('.')
    if len(ip_parts) == 4:
        masked_ip = f"{ip_parts[0]}.{ip_parts[1]}.{ip_parts[2]}.XXX"
    else:
        masked_ip = "XXX.XXX.XXX.XXX" # Fallback for invalid IP formats
```

Gemini

Task: Use an AI tool to generate a Python script that logs user activity (username, IP address, timestamp). Analyze: Examine whether sensitive data is logged unnecessarily or insecurely. Expected Output: • Identified privacy risks in logging. • Improved version with minimal, anonymized, or masked logging. • Explanation of privacy-aware logging principles.

Okay, I will create a plan to generate a Python script for logging user activity, then analyze it for privacy risks, provide an improved version, and finally explain privacy-aware logging principles.

- **Generate Initial Logging Script**: Generate a basic Python script that logs user activity, including username, IP address, and timestamp.
- **Analyze Logging Script for Privacy Risks**: Examine the generated logging script to identify potential privacy risks related to sensitive data being logged unnecessarily or insecurely.
- **Generate Improved Logging Script**: Create a revised version of the logging script that addresses identified privacy risks, incorporating techniques such as minimal logging, anonymization, or data masking.
- **Explain Privacy-Aware Logging Principles**: Provide a detailed explanation of key principles and best practices for privacy-aware logging.
- **Final Task**: Summarize the entire process, including the initial logging script, identified privacy flaws, the improved logging script, and the principles of privacy-aware logging.

Generating...
- Preparing
- Generate Initial Logging Script
- Analyze Logging Script for Privacy Risks
- Generate Improved Logging Script
- Explain Privacy-Aware Logging Principles
- Final Task

Explain the security flaws in the current login system code | Revise the login system code to use password hashing | Show best practices for secure authentication

What can I help you build?

Gemini 2.5 Flash

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

```
    # d. Construct a log message
    secure_log_message = f"[{timestamp}] UserHash: {hashed_username}, MaskedIP: {masked_ip}, Activity: {activity}"

    # e. Append to a new secure log file
    with open('secure_user_activity.log', 'a') as log_file:
        log_file.write(secure_log_message + '\n')
    print(f"Logged securely: {secure_log_message}")
```

Reasoning: Now that the `log_activity_secure` function is defined, I will call it at least three times with different example data to demonstrate its functionality, as per the instructions.

```
print("\n--- Demonstrating secure log_activity_secure function ---")
log_activity_secure("alice_secure", "192.168.1.101", "secure_login_success")
log_activity_secure("bob_secure", "10.0.0.6", "secure_login_failure")
log_activity_secure("charlie_secure", "172.16.0.21", "secure_data_access")
```

```
--- Demonstrating secure log_activity_secure function ---
Logged securely: [2026-01-23 08:23:41] UserHash: 52a2fd1cfb6a7521eeba158e2c4cf9f122701341f47d784909557ca1af593846, MaskedIP: 192.168.1.XXX, Activi
Logged securely: [2026-01-23 08:23:41] UserHash: 4c5196ddb0f972fc8e5919ec0628d180007769805a752069c6fce1284389c6b7, MaskedIP: 10.0.0.XXX, Activity:
Logged securely: [2026-01-23 08:23:41] UserHash: ae8fc434dd88b07b261a4a31c3a0426f5b72083c5944493121f4d2a783e8fb9c, MaskedIP: 172.16.0.XXX, Activit
```

Explain Privacy-Aware Logging Principles

Subtask:

Provide a detailed explanation of key principles and best practices for privacy-aware logging.

Privacy-Aware Logging Principles and Best Practices

Privacy-aware logging is crucial for protecting sensitive user data and complying with data protection regulations (e.g., GDPR, CCPA). It involves a set of principles and practices designed to minimize privacy risks while still gathering necessary operational insights.

Here are the key principles and best practices:

1. **Data Minimization (Collect Less)**:
    - **Principle**: Only collect the absolute minimum amount of data required for the logging purpose. If a piece of data is not essential for debugging, security monitoring, or operational analysis, do not log it.

Gemini

Task: Use an AI tool to generate a Python script that logs user activity (username, IP address, timestamp). Analyze: Examine whether sensitive data is logged unnecessarily or insecurely. Expected Output: • Identified privacy risks in logging. • Improved version with minimal, anonymized, or masked logging. • Explanation of privacy-aware logging principles.

Okay, I will create a plan to generate a Python script for logging user activity, then analyze it for privacy risks, provide an improved version, and finally explain privacy-aware logging principles.

- **Generate Initial Logging Script**: Generate a basic Python script that logs user activity, including username, IP address, and timestamp.
- **Analyze Logging Script for Privacy Risks**: Examine the generated logging script to identify potential privacy risks related to sensitive data being logged unnecessarily or insecurely.
- **Generate Improved Logging Script**: Create a revised version of the logging script that addresses identified privacy risks, incorporating techniques such as minimal logging, anonymization, or data masking.
- **Explain Privacy-Aware Logging Principles**: Provide a detailed explanation of key principles and best practices for privacy-aware logging.
- **Final Task**: Summarize the entire process, including the initial logging script, identified privacy flaws, the improved logging script, and the principles of privacy-aware logging.

Generating...
- Preparing
- Generate Initial Logging Script
- Analyze Logging Script for Privacy Risks
- Generate Improved Logging Script
- Explain Privacy-Aware Logging Principles
- Final Task

Explain the security flaws in the current login system code | Revise the login system code to use password hashing | Show best practices for secure authentication

What can I help you build?

Gemini 2.5 Flash

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

Untitled31.ipynb

colab.research.google.com/drive/1V94SXPmkmSItY9N3qEdY7IAJPTde_Odl#scrollTo=7659c709

- **Best Practice**: Review log requirements regularly. Avoid logging Personally Identifiable Information (PII), sensitive data (e.g., health records, financial information), or unnecessary contextual details by default. Be explicit about what *is* logged, rather than what is *not*.

2. **Anonymization and Pseudonymization (Obscure Identities)**:
    - **Principle**: Transform identifying data into a format that cannot be directly linked to an individual, or can only be linked with additional, separate information.
    - **Best Practice**:
        - **Anonymization**: Irreversibly remove all direct and indirect identifiers (e.g., hash usernames, mask IP addresses, generalize location data). This data cannot be re-identified.
        - **Pseudonymization**: Replace direct identifiers with artificial identifiers (pseudonyms). This allows data analysis without directly identifying the data subject, but re-identification is possible if the key linking pseudonyms to real identities is available and used responsibly.
        - Use strong, one-way cryptographic hashes (e.g., SHA256 with a salt) for PII where re-identification is not needed.
3. **Access Control (Limit Who Sees What)**:
    - **Principle**: Restrict access to log data to only those individuals or systems that absolutely need it for their specific job functions.
    - **Best Practice**: Implement a robust Role-Based Access Control (RBAC) system. Differentiate access levels (e.g., some roles might only see anonymized logs, while incident response teams might have more granular access under strict conditions). Regularly audit and review access permissions.
4. **Secure Storage (Protect Logs at Rest and in Transit)**:
    - **Principle**: Ensure that log data is protected from unauthorized access, modification, or destruction throughout its lifecycle.
    - **Best Practice**:
        - **Encryption**: Encrypt log files at rest (on disk) and in transit (when being transferred between systems). Use industry-standard encryption protocols (e.g., TLS for transit, AES-256 for rest).
        - **Integrity**: Implement mechanisms (e.g., digital signatures, secure hashing) to ensure log data has not been tampered with.
        - **Segregation**: Store sensitive logs separately from less sensitive operational logs if possible. Isolate log storage infrastructure from production systems.
5. **Retention Policies (Don't Keep Forever)**:
    - **Principle**: Define and enforce clear policies for how long log data is stored, and securely delete it when it is no longer needed.
    - **Best Practice**: Establish retention periods based on legal requirements, compliance obligations, and business needs.

Gemini

Task: Use an AI tool to generate a Python script that logs user activity (username, IP address, timestamp). Analyze: Examine whether sensitive data is logged unnecessarily or insecurely. Expected Output: • Identified privacy risks in logging. • Improved version with minimal, anonymized, or masked logging. • Explanation of privacy-aware logging principles.

Okay, I will create a plan to generate a Python script for logging user activity, then analyze it for privacy risks, provide an improved version, and finally explain privacy-aware logging principles.

- **Generate Initial Logging Script**: Generate a basic Python script that logs user activity, including username, IP address, and timestamp.
- **Analyze Logging Script for Privacy Risks**: Examine the generated logging script to identify potential privacy risks related to sensitive data being logged unnecessarily or insecurely.
- **Generate Improved Logging Script**: Create a revised version of the logging script that addresses identified privacy risks, incorporating techniques such as minimal logging, anonymization, or data masking.
- **Explain Privacy-Aware Logging Principles**: Provide a detailed explanation of key principles and best practices for privacy-aware logging.
- **Final Task**: Summarize the entire process, including the initial logging script, identified privacy flaws, the improved logging script, and the principles of privacy-aware logging.

Generating...
- Preparing
- Generate Initial Logging Script
- Analyze Logging Script for Privacy Risks
- Generate Improved Logging Script
- Explain Privacy-Aware Logging Principles
- Final Task

Explain the security flaws in the current login system code | Revise the login system code to use password hashing | Show best practices for secure authentication

What can I help you build?

Gemini 2.5 Flash

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

Untitled31.ipynb

colab.research.google.com/drive/1V94SXPmkmSItY9N3qEdY7IAJPTde_Odl#scrollTo=7659c709

- **Best Practice**: Establish retention periods based on legal requirements, compliance obligations, and business needs.
    Implement automated log rotation and deletion processes. Ensure that deletion is secure and irreversible (e.g., cryptographic shredding for sensitive data).
6. **Transparency (Be Open About Logging)**:
    - **Principle**: Inform users about what data is being collected, why it's being collected, how it's used, and for how long it will be retained.
    - **Best Practice**: Clearly articulate logging practices in privacy policies, terms of service, or user notices. Provide mechanisms for users to understand and potentially control their data where appropriate (e.g., data subject access requests).

Final Task

Subtask:

Summarize the entire process, including the initial logging script, identified privacy flaws, the improved logging script, and the principles of privacy-aware logging.

Summary:

Data Analysis Key Findings

- **Initial Logging Script Functionality**: A basic Python script was successfully created to log user activity, including username, IP address, and timestamp, to a file named `user_activity.log`.
- **Identified Privacy Risks**: The initial script posed significant privacy risks due to:
    - Logging Personally Identifiable Information (PII) such as usernames and IP addresses in plain text.
    - Lack of explicit security measures for the log file, including no encryption or access control, making it vulnerable to unauthorized access.
    - Potential for detailed user tracking and profiling by correlating logged data over time.
    - Absence of log retention policies, leading to indefinite storage of potentially sensitive data.
- **Improved Logging Script Implementation**: A revised script, `log_activity_secure`, was developed to mitigate these risks by:
    - Anonymizing usernames through SHA256 hashing (e.g., "alice_secure" became a hash).
    - Masking IP addresses by obscuring the last octet (e.g., "192.168.1.101" became "192.168.1.XXX").
    - Writing to a new, conceptually more secure log file named `secure_user_activity.log`.
- **Key Principles of Privacy-Aware Logging**: The analysis outlined essential principles for secure and private logging:

Gemini

Task: Use an AI tool to generate a Python script that logs user activity (username, IP address, timestamp). Analyze: Examine whether sensitive data is logged unnecessarily or insecurely. Expected Output: • Identified privacy risks in logging. • Improved version with minimal, anonymized, or masked logging. • Explanation of privacy-aware logging principles.

Okay, I will create a plan to generate a Python script for logging user activity, then analyze it for privacy risks, provide an improved version, and finally explain privacy-aware logging principles.

- **Generate Initial Logging Script**: Generate a basic Python script that logs user activity, including username, IP address, and timestamp.
- **Analyze Logging Script for Privacy Risks**: Examine the generated logging script to identify potential privacy risks related to sensitive data being logged unnecessarily or insecurely.
- **Generate Improved Logging Script**: Create a revised version of the logging script that addresses identified privacy risks, incorporating techniques such as minimal logging, anonymization, or data masking.
- **Explain Privacy-Aware Logging Principles**: Provide a detailed explanation of key principles and best practices for privacy-aware logging.
- **Final Task**: Summarize the entire process, including the initial logging script, identified privacy flaws, the improved logging script, and the principles of privacy-aware logging.

Generating...
- Preparing
- Generate Initial Logging Script
- Analyze Logging Script for Privacy Risks
- Generate Improved Logging Script
- Explain Privacy-Aware Logging Principles
- Final Task

Explain the security flaws in the current login system code | Revise the login system code to use password hashing | Show best practices for secure authentication

What can I help you build?

Gemini 2.5 Flash

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

- Anonymizing usernames through SHA256 hashing (e.g., "alice_secure" became a hash).
- Masking IP addresses by obscuring the last octet (e.g., "192.168.1.101" became "192.168.1.XXX").
- Writing to a new, conceptually more secure log file named `secure_user_activity.log`.

- **Key Principles of Privacy-Aware Logging**: The analysis outlined essential principles for secure and private logging:
  - **Data Minimization**: Log only the essential data.
  - **Anonymization/Pseudonymization**: Obscure identities through hashing or masking.
  - **Access Control**: Restrict who can view log data.
  - **Secure Storage**: Encrypt logs at rest and in transit.
  - **Retention Policies**: Define and enforce time limits for log storage.
  - **Transparency**: Inform users about logging practices.

Insights or Next Steps

- The comparison between the initial and improved logging scripts highlights that implementing simple techniques like hashing and masking can significantly reduce privacy exposure without completely sacrificing the utility of logs for operational insights.
- Further enhancements could include implementing encryption for the log file itself, integrating robust access control mechanisms, and automating log rotation and deletion based on defined retention policies to ensure compliance and data hygiene.

Gemini

Task: Use an AI tool to generate a Python script that logs user activity (username, IP address, timestamp). Analyze: Examine whether sensitive data is logged unnecessarily or insecurely. Expected Output: • Identified privacy risks in logging. • Improved version with minimal, anonymized, or masked logging. • Explanation of privacy-aware logging principles.

Okay, I will create a plan to generate a Python script for logging user activity, then analyze it for privacy risks, provide an improved version, and finally explain privacy-aware logging principles.

- **Generate Initial Logging Script**: Generate a basic Python script that logs user activity, including username, IP address, and timestamp.
- **Analyze Logging Script for Privacy Risks**: Examine the generated logging script to identify potential privacy risks related to sensitive data being logged unnecessarily or insecurely.
- **Generate Improved Logging Script**: Create a revised version of the logging script that addresses identified privacy risks, incorporating techniques such as minimal logging, anonymization, or data masking.
- **Explain Privacy-Aware Logging Principles**: Provide a detailed explanation of key principles and best practices for privacy-aware logging.
- **Final Task**: Summarize the entire process, including the initial logging script, identified privacy flaws, the improved logging script, and the principles of privacy-aware logging.

Generating...

- Preparing
- Generate Initial Logging Script
- Analyze Logging Script for Privacy Risks
- Generate Improved Logging Script
- Explain Privacy-Aware Logging Principles
- Final Task

Explain the security flaws in the current login system code | Revise the login system code to use password hashing | Show best practices for secure authentication

What can I help you build?

Gemini 2.5 Flash

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