

# AI ASSISTED CODING

ROHITH REDDY

2303A51111

BATCH – 03

30 – 01 – 2026

## ASSIGNMENT – 5.5

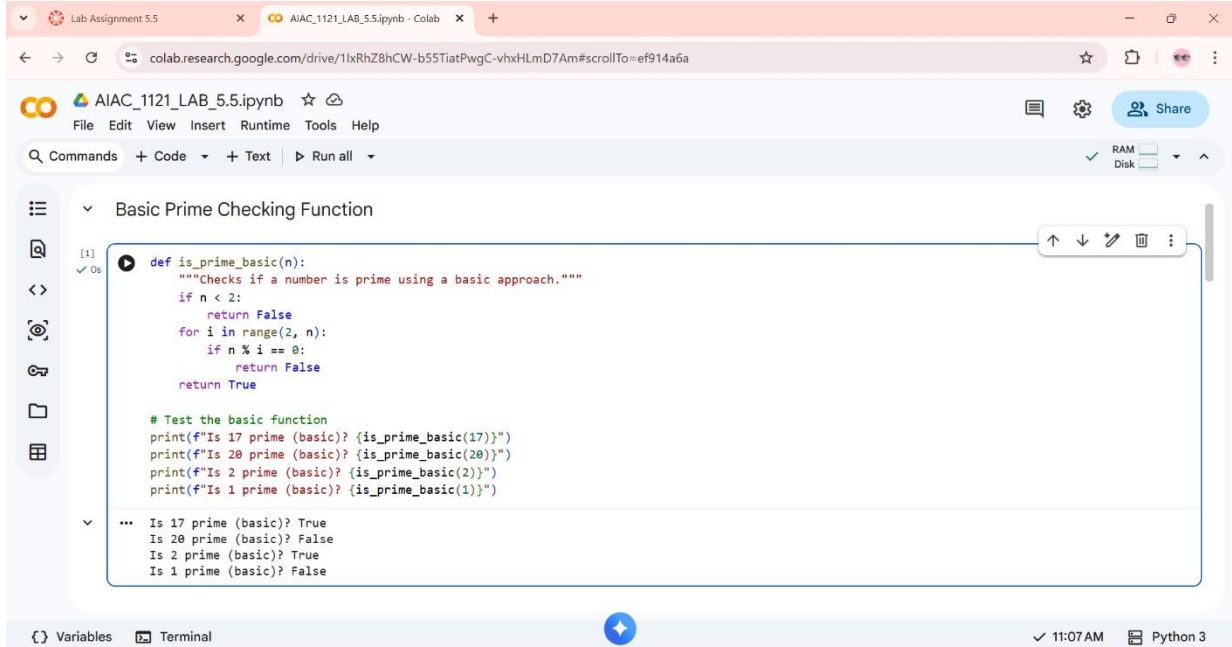
**Lab 5:** Ethical Foundations – Responsible AI Coding Practices.

**TASK - 01 :** (Transparency in Algorithm Optimization)

**Prompt :** Generate Python code for two prime-checking methods and explain how the optimized version improves performance.

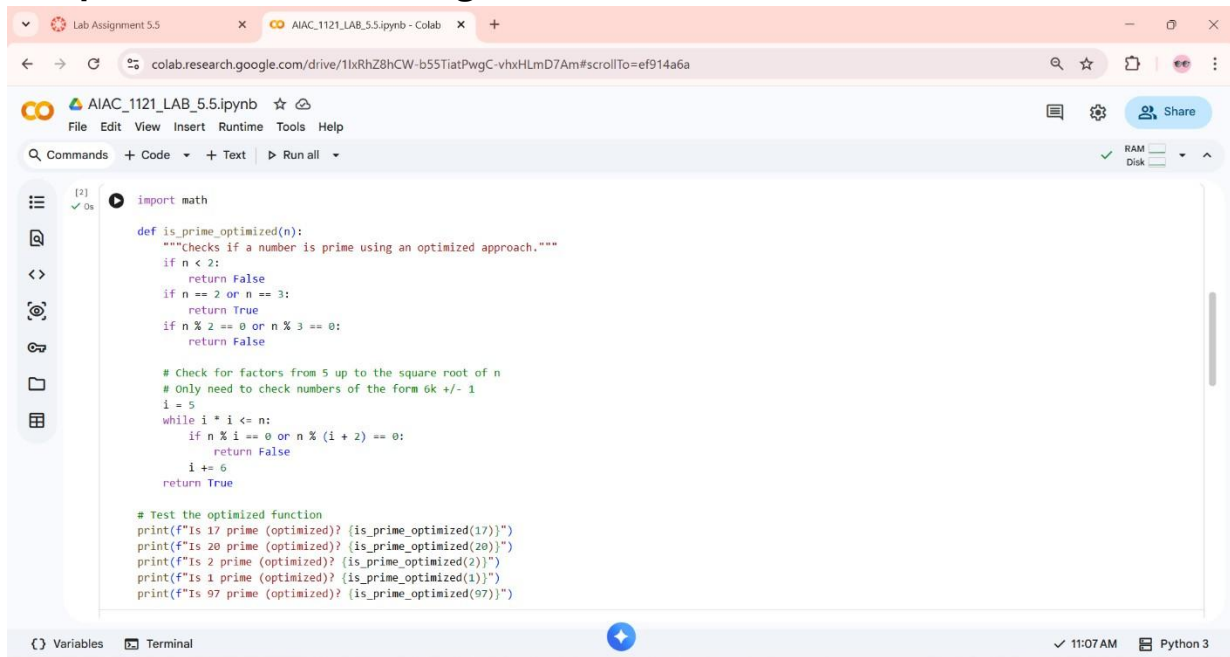
**Code:**

### 1. Basic Prime Checking Function



```
[1] def is_prime_basic(n):  
    """Checks if a number is prime using a basic approach."""  
    if n < 2:  
        return False  
    for i in range(2, n):  
        if n % i == 0:  
            return False  
    return True  
  
# Test the basic function  
print(f"Is 17 prime (basic)? {is_prime_basic(17)}")  
print(f"Is 20 prime (basic)? {is_prime_basic(20)}")  
print(f"Is 2 prime (basic)? {is_prime_basic(2)}")  
print(f"Is 1 prime (basic)? {is_prime_basic(1)}")  
  
... Is 17 prime (basic)? True  
Is 20 prime (basic)? False  
Is 2 prime (basic)? True  
Is 1 prime (basic)? False
```

## 2. Optimized Prime Checking Function



The screenshot shows a Google Colab notebook titled "AIAC\_1121\_LAB\_5.5.ipynb". The code defines a function `is_prime_optimized(n)` that checks if a number is prime using an optimized approach. The function includes comments explaining the logic: it checks for small primes (2, 3), then checks for factors from 5 up to the square root of `n`, only testing numbers of the form `6k +/- 1`. Below the function, there are test cases for `is_prime_optimized(17)`, `is_prime_optimized(20)`, `is_prime_optimized(2)`, `is_prime_optimized(1)`, and `is_prime_optimized(97)`.

```
[2] import math

def is_prime_optimized(n):
    """Checks if a number is prime using an optimized approach."""
    if n < 2:
        return False
    if n == 2 or n == 3:
        return True
    if n % 2 == 0 or n % 3 == 0:
        return False

    # Check for factors from 5 up to the square root of n
    # Only need to check numbers of the form 6k +/- 1
    i = 5
    while i * i <= n:
        if n % i == 0 or n % (i + 2) == 0:
            return False
        i += 6
    return True

# Test the optimized function
print(f"Is 17 prime (optimized)? {is_prime_optimized(17)}")
print(f"Is 20 prime (optimized)? {is_prime_optimized(20)}")
print(f"Is 2 prime (optimized)? {is_prime_optimized(2)}")
print(f"Is 1 prime (optimized)? {is_prime_optimized(1)}")
print(f"Is 97 prime (optimized)? {is_prime_optimized(97)}")
```

### Transparent Explanation:

Naive Method Time Complexity:  $O(n)$  → Checks all numbers from 2 to  $n-1$ .

Optimized Method Time Complexity:  $O(\sqrt{n})$  → Only checks up to square root of  $n$ .

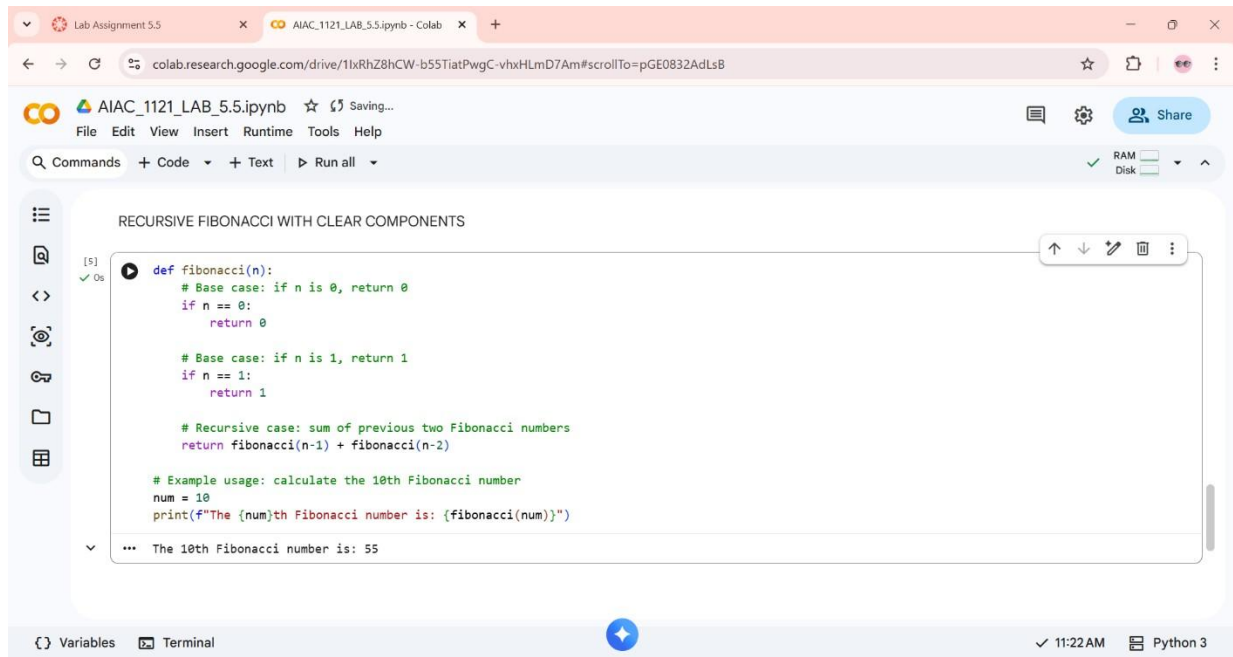
### Comparison :

Method	Time Complexity	Performance
Naive	$O(n)$	Slower
Optimized	$O(\sqrt{n})$	Faster

**Task – 02 :** Transparency in Recursive Algorithms.

**Prompt :** Give me the Recursive Fibonacci code with clear comments.

## Code:



```
RECURSIVE FIBONACCI WITH CLEAR COMPONENTS

def fibonacci(n):
    # Base case: if n is 0, return 0
    if n == 0:
        return 0

    # Base case: if n is 1, return 1
    if n == 1:
        return 1

    # Recursive case: sum of previous two Fibonacci numbers
    return fibonacci(n-1) + fibonacci(n-2)

# Example usage: calculate the 10th Fibonacci number
num = 10
print(f"The {num}th Fibonacci number is: {fibonacci(num)}")

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

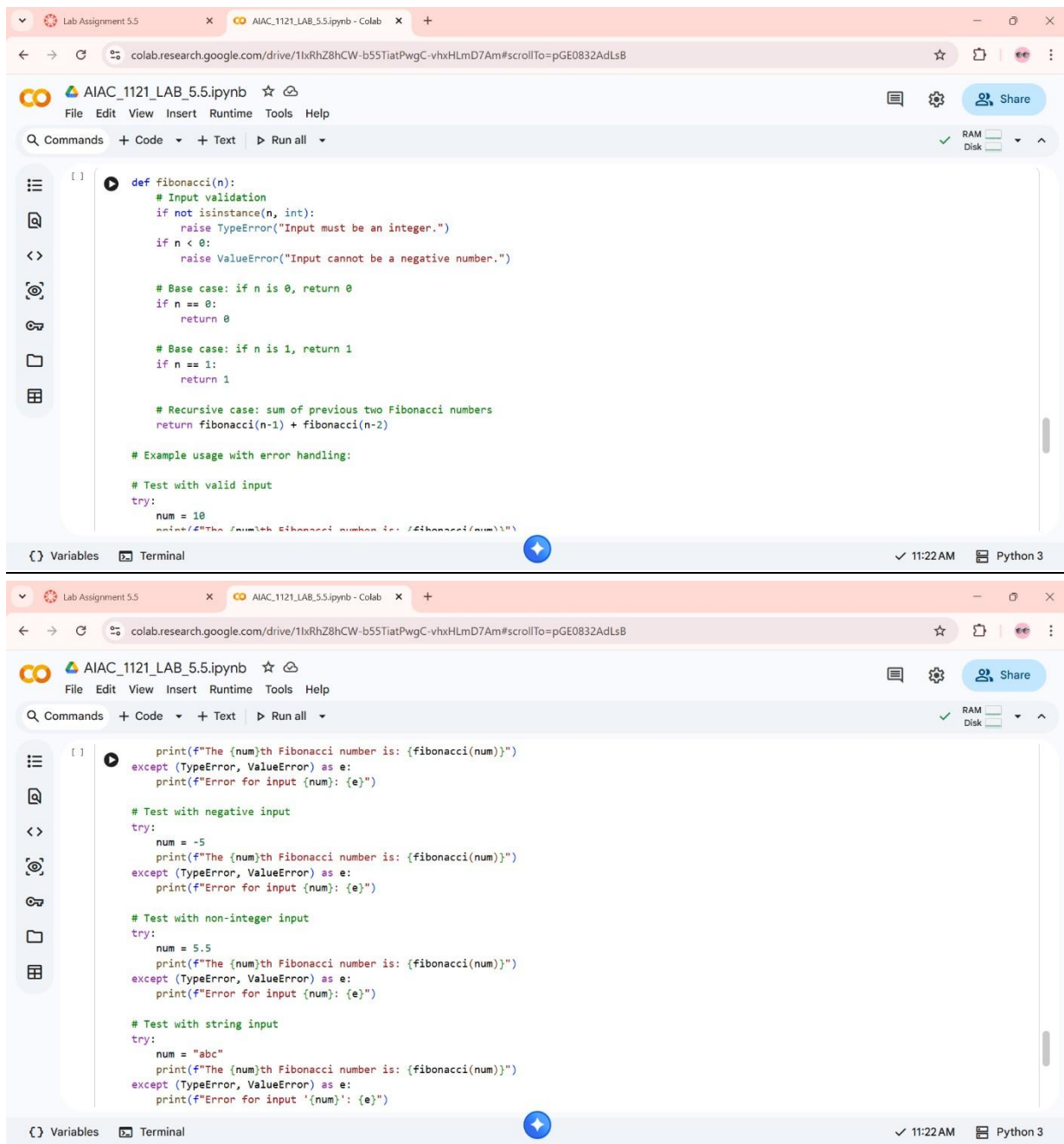
## Explanation:

- Base Cases:
  - $\text{fibonacci}(0) \rightarrow 0$  ◦  $\text{fibonacci}(1) \rightarrow 1$
- Recursive Call:
  - $\text{fibonacci}(n) = \text{fibonacci}(n-1) + \text{fibonacci}(n-2)$

## Task – 03 : Transparency in Error Handling.

**Prompt :** Generate code with proper error handling and clear explanations for each exception.

## Code:



The image shows two screenshots of a Google Colab notebook titled "AIAC\_1121\_LAB\_5.5.ipynb". The first screenshot displays the definition of a `fibonacci` function with input validation and recursive logic. The second screenshot shows test cases for the function, including valid input, negative input, non-integer input, and string input, all wrapped in try-except blocks to handle errors.

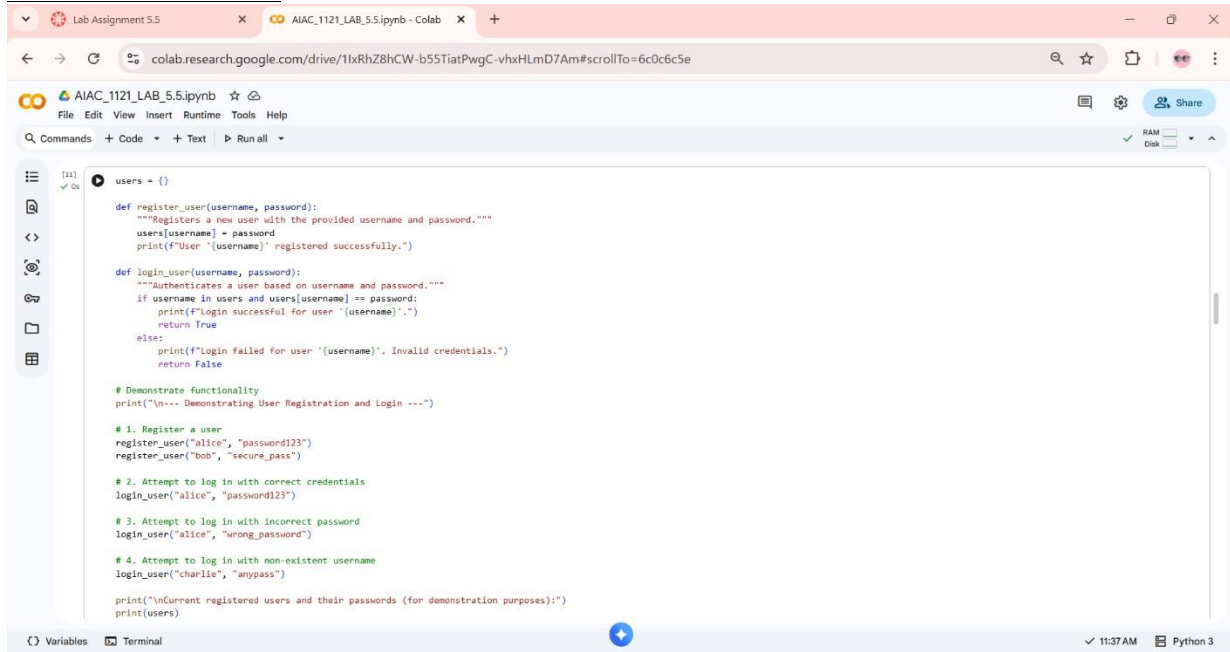
```
[ ] def fibonacci(n):  
    # Input validation  
    if not isinstance(n, int):  
        raise TypeError("Input must be an integer.")  
    if n < 0:  
        raise ValueError("Input cannot be a negative number.")  
  
    # Base case: if n is 0, return 0  
    if n == 0:  
        return 0  
  
    # Base case: if n is 1, return 1  
    if n == 1:  
        return 1  
  
    # Recursive case: sum of previous two Fibonacci numbers  
    return fibonacci(n-1) + fibonacci(n-2)  
  
# Example usage with error handling:  
  
# Test with valid input  
try:  
    num = 10  
    print(f"The {num}th Fibonacci number is: {fibonacci(num)}")  
  
print(f"The {num}th Fibonacci number is: {fibonacci(num)}")  
  
# Test with negative input  
try:  
    num = -5  
    print(f"The {num}th Fibonacci number is: {fibonacci(num)}")  
except (TypeError, ValueError) as e:  
    print(f"Error for input {num}: {e}")  
  
# Test with non-integer input  
try:  
    num = 5.5  
    print(f"The {num}th Fibonacci number is: {fibonacci(num)}")  
except (TypeError, ValueError) as e:  
    print(f"Error for input {num}: {e}")  
  
# Test with string input  
try:  
    num = "abc"  
    print(f"The {num}th Fibonacci number is: {fibonacci(num)}")  
except (TypeError, ValueError) as e:  
    print(f"Error for input '{num}': {e}")
```

## Explaining the Errors:

Exception	Meaning
FileNotFoundError	File does not exist

PermissionError No permission to read file Exception  
Any other unknown error **Task – 04 : Security in User Authentication. Code:**

## Insecure Version:



```
[11]: users = {}

def register_user(username, password):
    """Registers a new user with the provided username and password."""
    users[username] = password
    print(f"User '{username}' registered successfully.")

def login_user(username, password):
    """Authenticates a user based on username and password."""
    if username in users and users[username] == password:
        print(f"Login successful for user '{username}'.")
        return True
    else:
        print(f"Login failed for user '{username}'. Invalid credentials.")
        return False

# Demonstrate functionality
print("\n--- Demonstrating User Registration and Login ---")

# 1. Register a user
register_user("alice", "password123")
register_user("bob", "secure_pass")

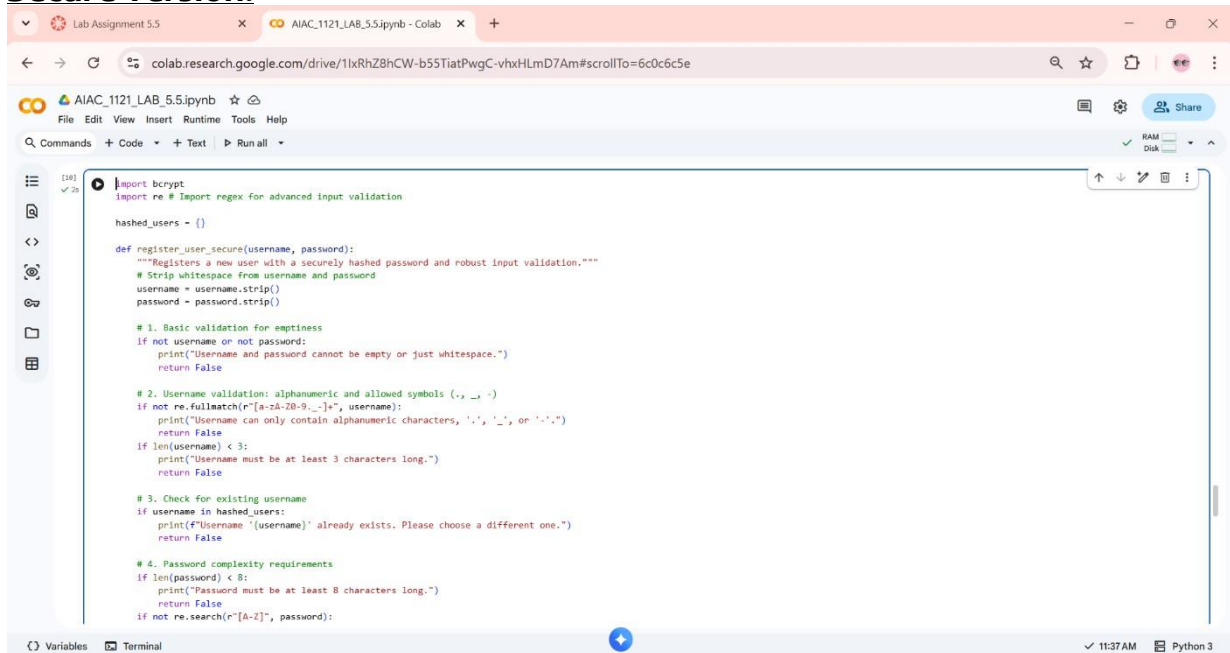
# 2. Attempt to log in with correct credentials
login_user("alice", "password123")

# 3. Attempt to log in with incorrect password
login_user("alice", "wrong_password")

# 4. Attempt to log in with non-existent username
login_user("charlie", "anypass")

print("\nCurrent registered users and their passwords (for demonstration purposes):")
print(users)
```

## Secure Version:



```
[10]: import bcrypt
import re # Import regex for advanced input validation

hashed_users = {}

def register_user_secure(username, password):
    """Registers a new user with a securely hashed password and robust input validation."""
    # Strip whitespace from username and password
    username = username.strip()
    password = password.strip()

    # 1. Basic validation for emptiness
    if not username or not password:
        print("Username and password cannot be empty or just whitespace.")
        return False

    # 2. Username validation: alphanumeric and allowed symbols (. _ -)
    if not re.fullmatch(r"[a-zA-Z0-9_.-]+", username):
        print("Username can only contain alphanumeric characters, '.', '_', or '-'")
        return False
    if len(username) < 3:
        print("Username must be at least 3 characters long.")
        return False

    # 3. Check for existing username
    if username in hashed_users:
        print(f"Username '{username}' already exists. Please choose a different one.")
        return False

    # 4. Password complexity requirements
    if len(password) < 8:
        print("Password must be at least 8 characters long.")
        return False
    if not re.search(r"[A-Z]", password):
```

The image displays two screenshots of a Google Colab notebook titled 'AIAC\_1121\_LAB\_5.5.ipynb'. The first screenshot shows the initial code for user registration and login. The second screenshot shows the code after execution, displaying the output of the registration and login attempts.

```
def login_user_secure(username, password):
    """Authenticates a user against their securely hashed password with input stripping."""
    # Strip whitespace from username and password
    username = username.strip()
    password = password.strip()

    if username not in hashed_users:
        print("Login failed: Invalid credentials.") # Generic message for security
        return False

    # Check the provided password against the stored hash
    if bcrypt.checkpw(password.encode('utf-8'), hashed_users[username]):
        print(f"Login successful for user '{username}'.")
        return True
    else:
        print("Login failed: Invalid credentials.") # Generic message for security
        return False

# Demonstrate functionality with enhanced secure system
print("\n--- Demonstrating Enhanced Secure User Registration and Login ---")

# 1. Register users with new validations
register_user_secure("jane_doe", "StrongPass1!")
register_user_secure("user with space", "ValidPass2@") # Invalid username
register_user_secure("anotherUser", "ValidPass3#") # Invalid username
register_user_secure("bob", "weak") # Password too short
register_user_secure("cui", "onlylowercase") # Missing uppercase, digit, special
register_user_secure("david", "SecurePass4") # Missing special character
register_user_secure("emily", "emily123!") # Valid password, but username exists
register_user_secure("emily", "EmilyP4ss") # Valid registration

# 2. Demonstrate stripping whitespace
register_user_secure(" padded_user ", " PaddedPass5$ ") # Should register 'padded_user'
login_user_secure("padded_user", "PaddedPass5$")
login_user_secure(" padded_user ", "PaddedPass5$") # Login with padded username
login_user_secure("padded_user", " PaddedPass5$ ") # Login with padded password

# 3. Attempt to log in with correct credentials
login_user_secure("jane_doe", "StrongPass1!")

# 4. Attempt to log in with incorrect password
login_user_secure("jane_doe", "wrong_password")

# 5. Attempt to log in with non-existent username
login_user_secure("frank", "anypass")

print("\nCurrent registered users (hashed passwords stored, not displayed for security):")
print(f"Users registered: {list(hashed_users.keys())}")

--- Demonstrating Enhanced Secure User Registration and Login ---
User 'jane_doe' registered securely.
Username can only contain alphanumeric characters, '_', '-' or '.'.
Username can only contain alphanumeric characters, '_', '-' or '.'.
Password must be at least 8 characters long.
Password must contain at least one special character (!@#$%^&*()).
```

## Explanation :

- Always hash passwords
- Never store plain-text passwords
- Validate user input
- Use strong hashing algorithms

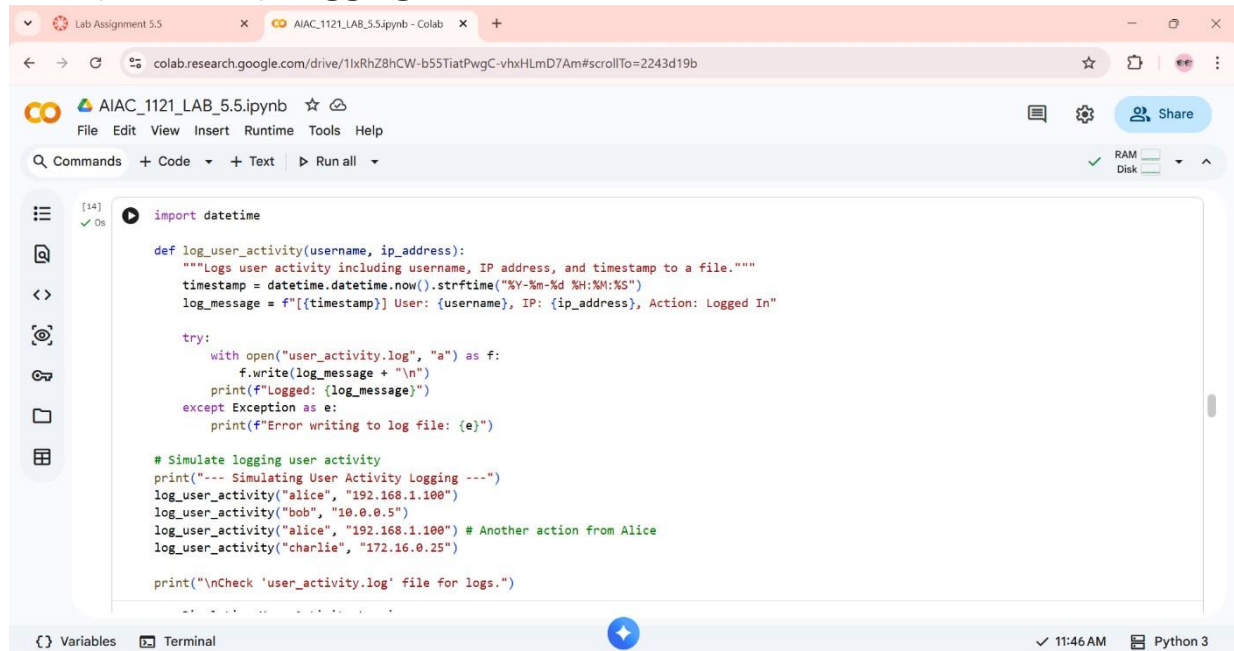
## Task – 05 : Privacy in Data Logging.



**Prompt – 01 :** Create a basic Python script that simulates logging user activity, including username, IP address, and timestamp, to a file or console.

**Code:**

**Privacy and Risky Logging:**



The screenshot shows a Google Colab notebook titled "AIAC\_1121\_LAB\_5.5.ipynb". The code in the notebook is as follows:

```
[14] import datetime

def log_user_activity(username, ip_address):
    """Logs user activity including username, IP address, and timestamp to a file."""
    timestamp = datetime.datetime.now().strftime("%Y-%m-%d %H:%M:%S")
    log_message = f"[{timestamp}] User: {username}, IP: {ip_address}, Action: Logged In"

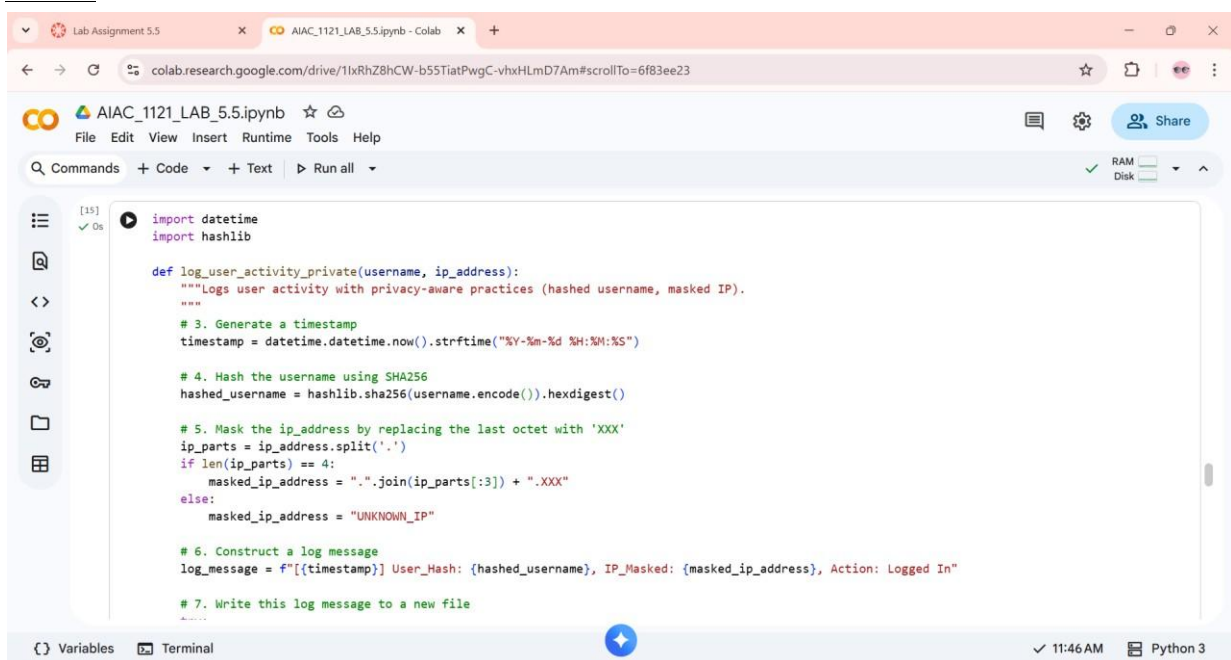
    try:
        with open("user_activity.log", "a") as f:
            f.write(log_message + "\n")
        print(f"Logged: {log_message}")
    except Exception as e:
        print(f"Error writing to log file: {e}")

# Simulate logging user activity
print("--- Simulating User Activity Logging ---")
log_user_activity("alice", "192.168.1.100")
log_user_activity("bob", "10.0.0.5")
log_user_activity("alice", "192.168.1.100") # Another action from Alice
log_user_activity("charlie", "172.16.0.25")

print("\nCheck 'user_activity.log' file for logs.")
```

**Prompt – 02 :** Examine the initial logging script to identify specific privacy risks associated with logging sensitive data like usernames and IP addresses directly. Detail potential negative impacts.

**Code:**



The screenshot shows a Google Colab notebook titled "AIAC\_1121\_LAB\_5.5.ipynb". The code in the notebook is as follows:

```
[15] import datetime
import hashlib

def log_user_activity_private(username, ip_address):
    """Logs user activity with privacy-aware practices (hashed username, masked IP)."""
    # 3. Generate a timestamp
    timestamp = datetime.datetime.now().strftime("%Y-%m-%d %H:%M:%S")

    # 4. Hash the username using SHA256
    hashed_username = hashlib.sha256(username.encode()).hexdigest()

    # 5. Mask the ip_address by replacing the last octet with 'XXX'
    ip_parts = ip_address.split('.')
    if len(ip_parts) == 4:
        masked_ip_address = ".".join(ip_parts[:3]) + ".XXX"
    else:
        masked_ip_address = "UNKNOWN_IP"

    # 6. Construct a log message
    log_message = f"[{timestamp}] User_Hash: {hashed_username}, IP_Masked: {masked_ip_address}, Action: Logged In"

    # 7. Write this log message to a new file
```

```
# 7. Write this log message to a new file
try:
    with open("user_activity_private.log", "a") as f:
        f.write(log_message + "\n")
    print(f"Logged (Private): {log_message}")
except Exception as e:
    print(f"Error writing to private log file: {e}")

# 8. Call the log_user_activity_private function with several example usernames and IP addresses
print("\n--- Simulating Privacy-Enhanced User Activity Logging ---")
log_user_activity_private("alice", "192.168.1.100")
log_user_activity_private("bob", "10.0.0.5")
log_user_activity_private("alice", "192.168.1.100") # Another action from Alice
log_user_activity_private("charlie", "172.16.0.25")
log_user_activity_private("diana", "203.0.113.42")

print("\nCheck 'user_activity_private.log' file for privacy-enhanced logs.")

...

--- Simulating Privacy-Enhanced User Activity Logging ---
Logged (Private): [2026-01-30 06:16:19] User_Hash: 2bd806c97f0e00af1a1fc3328fa763a9269723c8db8fac4f93af71db186d6e90, IP_Masked: 192.168.1.XXX, Action:
Logged (Private): [2026-01-30 06:16:19] User_Hash: 81b637d8fcd2c6da6359e6963113a1170de795e4b725b84d1e0b4cfd9ec58ce9, IP_Masked: 10.0.0.XXX, Action:
Logged (Private): [2026-01-30 06:16:19] User_Hash: 2bd806c97f0e00af1a1fc3328fa763a9269723c8db8fac4f93af71db186d6e90, IP_Masked: 192.168.1.XXX, Action:
Logged (Private): [2026-01-30 06:16:19] User_Hash: b9dd968c1753459a78115d3cb845a57d924b6877e885b08bd01886ccdf34433c, IP_Masked: 172.16.0.XXX, Action:
Logged (Private): [2026-01-30 06:16:19] User_Hash: 1b5f0044c1421413089060f2f037a47331a0577d43eb78830cd017551c7db, IP_Masked: 203.0.113.XXX, Action:
```

## **Explanation :**

Mask or anonymize sensitive data

Log only what is necessary

Avoid storing personal identifiers

Protect log files from unauthorized access

**THANK YOU!!**