Assignment_5.5

NAME: A. SATHWIK

HALL TICKET NO: 2303A51178

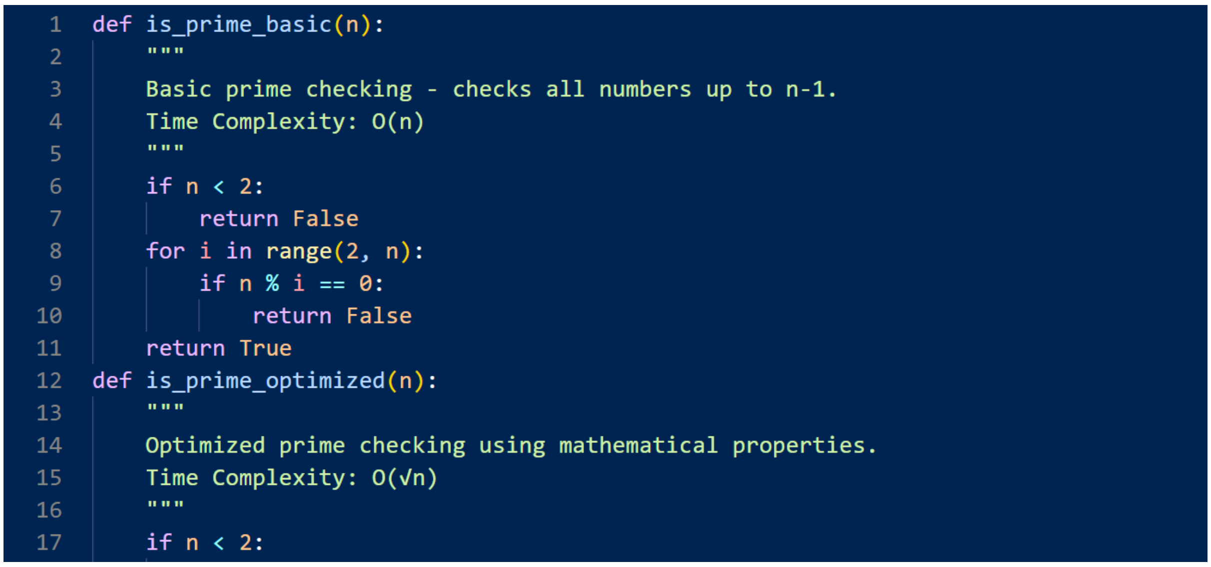
BATCH :29

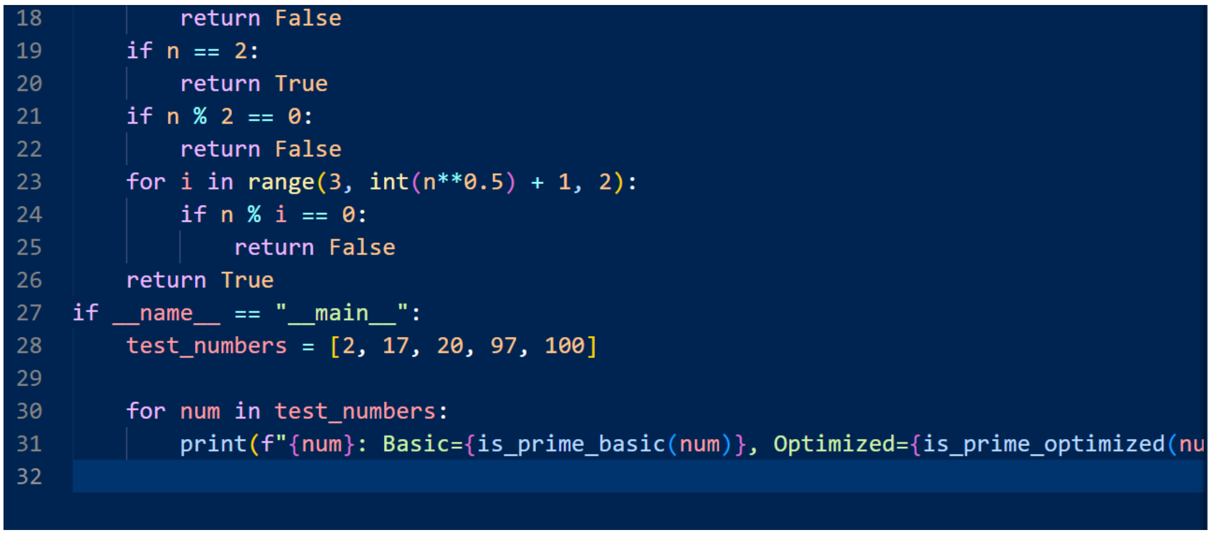
Task 1: Use AI to generate two solutions for checking prime numbers:

Prompt:

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

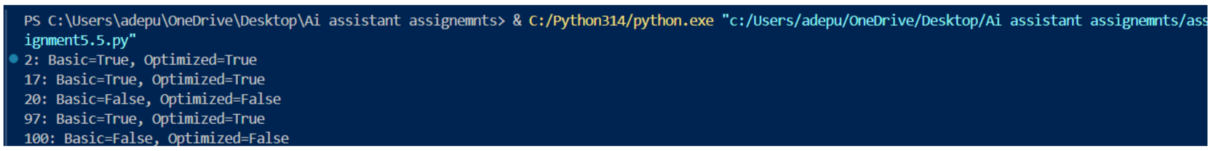
Code:





```python
def is_prime_basic(n):
    """
    Basic prime checking - checks all numbers up to n-1.
    Time Complexity: O(n)
    """
    if n < 2:
        return False
    for i in range(2, n):
        if n % i == 0:
            return False
    return True
def is_prime_optimized(n):
    """
    Optimized prime checking using mathematical properties.
    Time Complexity: O(√n)
    """
    if n < 2:
        return False
    if n == 2:
        return True
    if n % 2 == 0:
        return False
    for i in range(3, int(n**0.5) + 1, 2):
        if n % i == 0:
            return False
    return True
if __name__ == "__main__":
    test_numbers = [2, 17, 20, 97, 100]

    for num in test_numbers:
        print(f"{num}: Basic={is_prime_basic(num)}, Optimized={is_prime_optimized(nu

```

Output:



```
PS C:\Users\adepu\OneDrive\Desktop\Ai assistant assignemnts> & C:/Python314/python.exe "c:/Users/adepu/OneDrive/Desktop/Ai assistant assignemnts/ass
ignment5.5.py"
2: Basic=True, Optimized=True
17: Basic=True, Optimized=True
20: Basic=False, Optimized=False
97: Basic=True, Optimized=True
100: Basic=False, Optimized=False
```

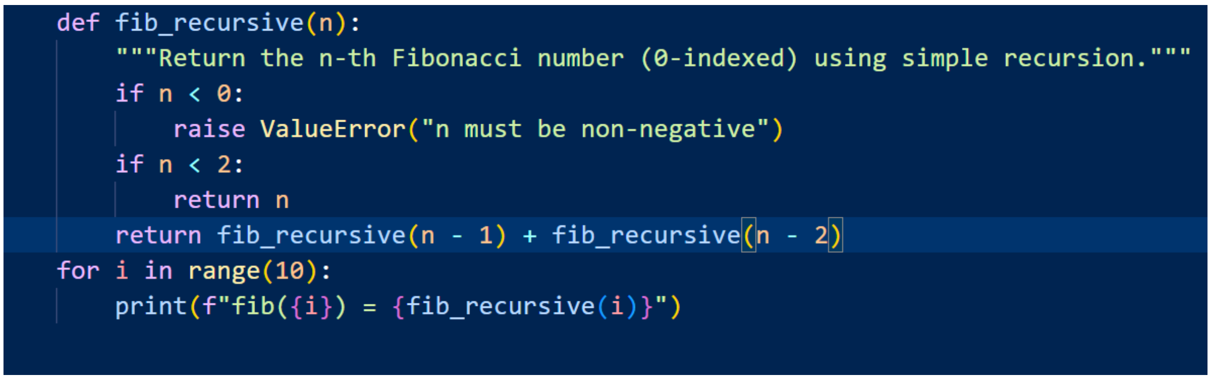
Observation:

The given program checks whether a number is prime using two approaches: a basic method and an optimized method. Both methods produce the same and correct results for all the tested numbers. Prime numbers such as 2, 17, and 97 are correctly identified, while non-prime numbers like 20 and 100 are also correctly detected. The basic method checks all numbers up to n−1, which increases the number of calculations and makes it slower for larger values. In contrast, the optimized method reduces unnecessary checks by testing divisibility only up to the square root of the number and skipping even numbers. Due to this reduction in computations, the optimized method performs faster and is more efficient. Hence, the optimized approach is more suitable for practical and large-scale applications.

Task Description #2 (Transparency in Recursive Algorithms)

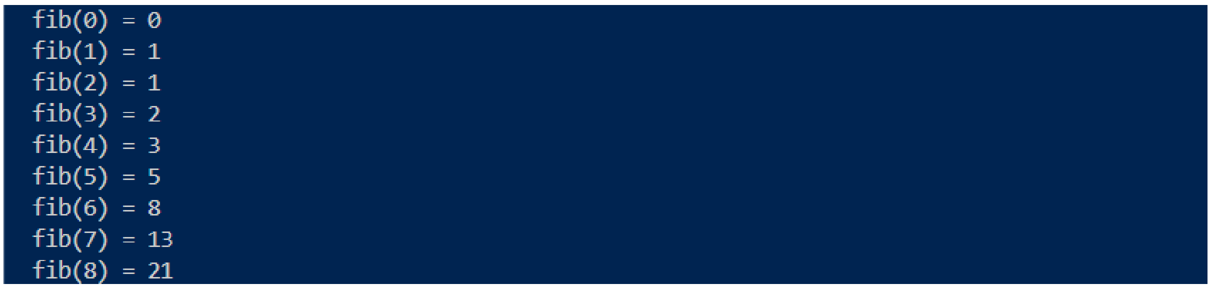
Objective: Use AI to generate a recursive function to calculate Fibonacci numbers.

Code:

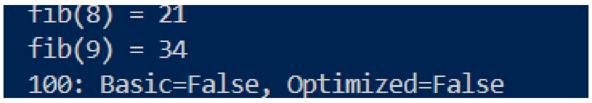


```python
def fib_recursive(n):
    """Return the n-th Fibonacci number (0-indexed) using simple recursion."""
    if n < 0:
        raise ValueError("n must be non-negative")
    if n < 2:
        return n
    return fib_recursive(n - 1) + fib_recursive(n - 2)
for i in range(10):
    print(f"fib({i}) = {fib_recursive(i)}")
```

Output:



```
fib(0) = 0
fib(1) = 1
fib(2) = 1
fib(3) = 2
fib(4) = 3
fib(5) = 5
fib(6) = 8
fib(7) = 13
fib(8) = 21
```



```
fib(8) = 21
fib(9) = 34
100: Basic=False, Optimized=False
```

Observation:

The given program computes Fibonacci numbers using a simple recursive approach. For each input value, the function calls itself to calculate the previous two Fibonacci numbers until it reaches the base cases. The base cases handle the smallest inputs, where the function directly returns the value for 0 and 1, ensuring the recursion stops correctly. The output shows that the function generates the correct Fibonacci sequence from fib (0) to fib (9). However, due to repeated function calls for the same values, this recursive method performs many redundant calculations. As a result, while the program is easy to understand and demonstrates the concept of recursion clearly, it becomes inefficient for larger input values. Therefore, this approach is suitable for learning recursion but not ideal for large-scale or performance-critical applications

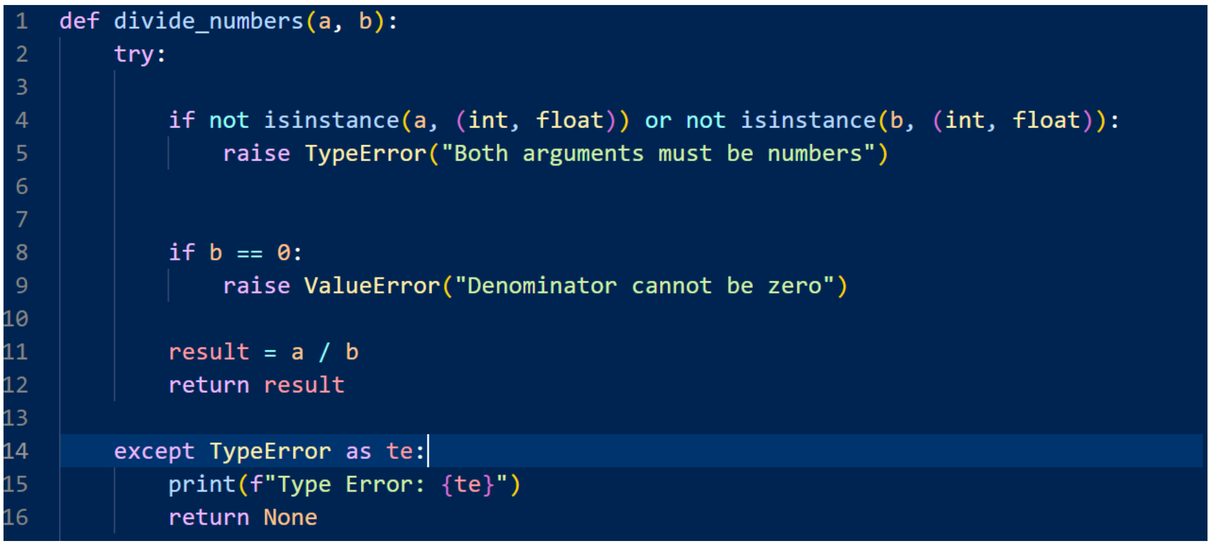
Task Description #3 (Transparency in Error Handling)

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

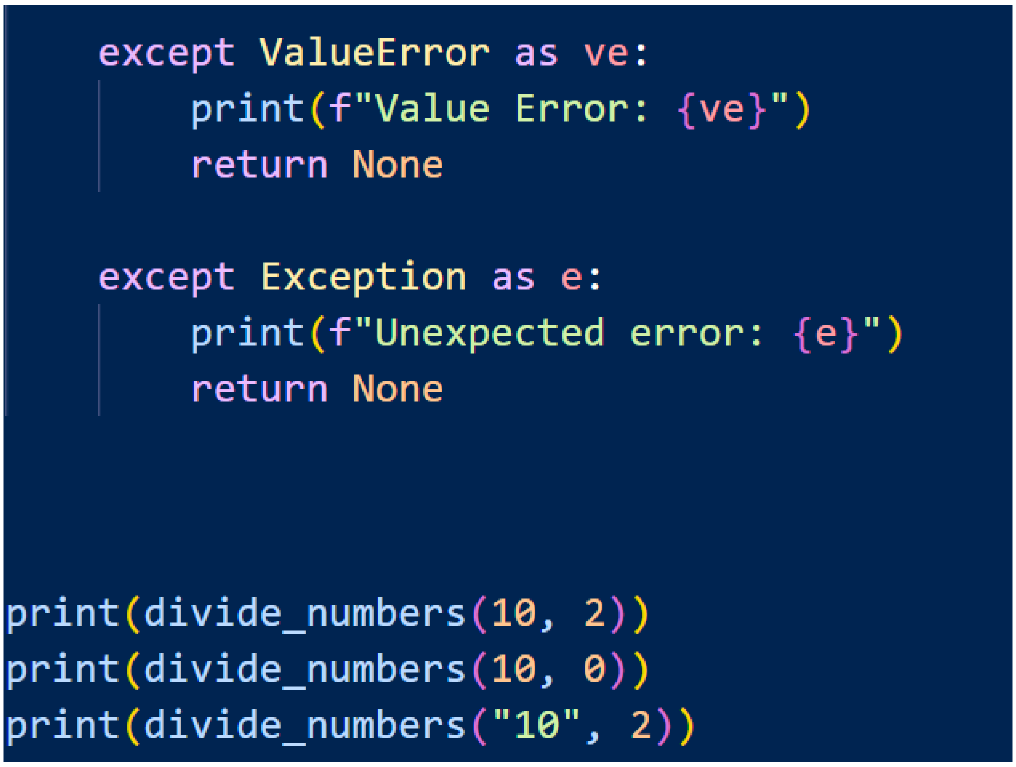
Prompt:

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

Code:

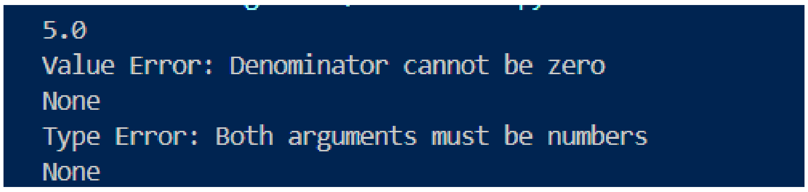


```python
def divide_numbers(a, b):
    try:

        if not isinstance(a, (int, float)) or not isinstance(b, (int, float)):
            raise TypeError("Both arguments must be numbers")


        if b == 0:
            raise ValueError("Denominator cannot be zero")

        result = a / b
        return result

    except TypeError as te:
        print(f"Type Error: {te}")
        return None
```



```python
    except ValueError as ve:
        print(f"Value Error: {ve}")
        return None

    except Exception as e:
        print(f"Unexpected error: {e}")
        return None



print(divide_numbers(10, 2))
print(divide_numbers(10, 0))
print(divide_numbers("10", 2))
```

Output:



```
5.0
Value Error: Denominator cannot be zero
None
Type Error: Both arguments must be numbers
None
```

Observation:

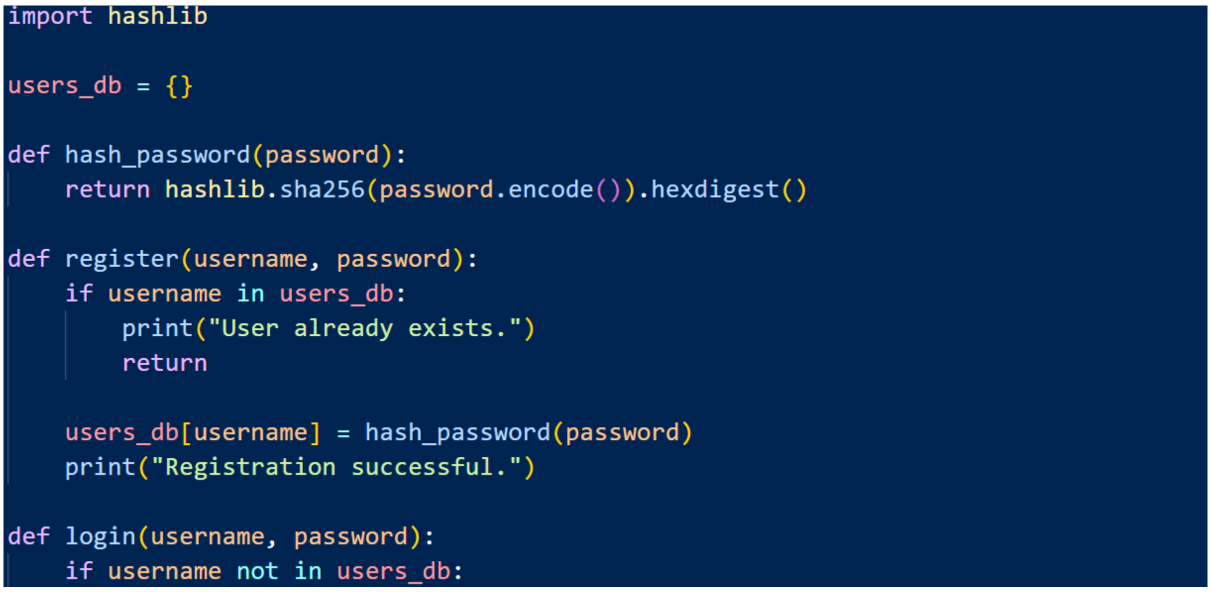
The given Python program demonstrates effective use of exception handling to safely perform division operations while managing possible runtime errors. When valid numeric inputs are provided and the denominator is non-zero, the function successfully performs the division and returns the correct result. If the denominator is zero, the program proactively raises a Value Error, preventing a runtime crash and displaying a clear error message before returning None. Similarly, when non-numeric input is passed, a Type Error is raised and handled gracefully with an informative message. The use of multiple except blocks ensures that different error conditions are identified and handled separately, improving code robustness and readability. Overall, the program ensures reliable execution by validating inputs and handling exceptional cases in a controlled manner.

Task Description #4 (Security in User Authentication)

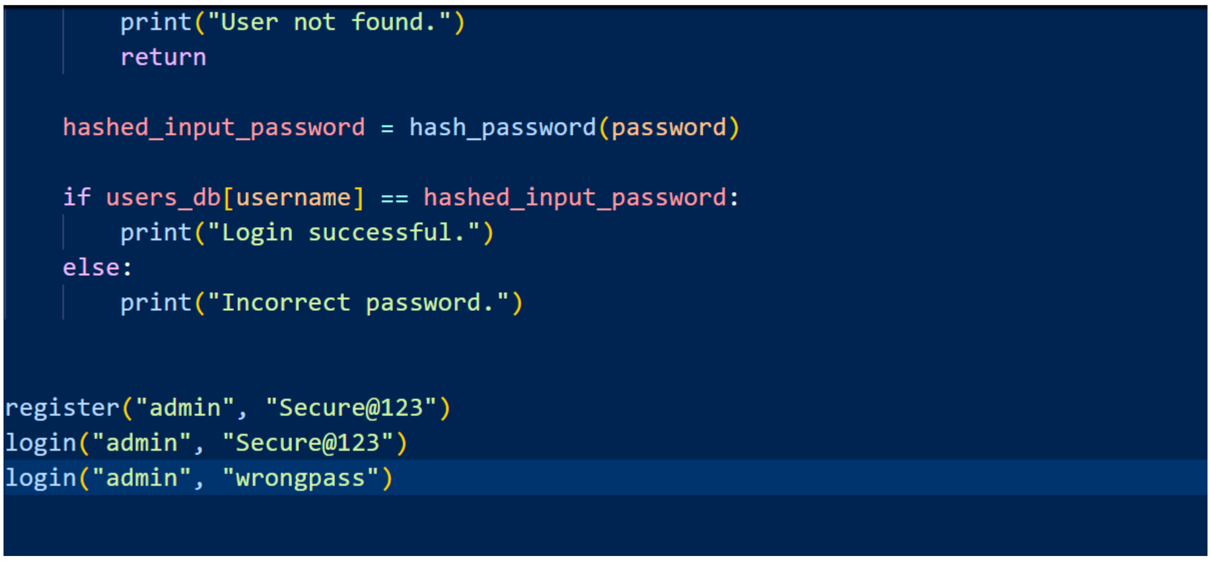
Task: Use an AI tool to generate a Python-based login system.

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

Code:

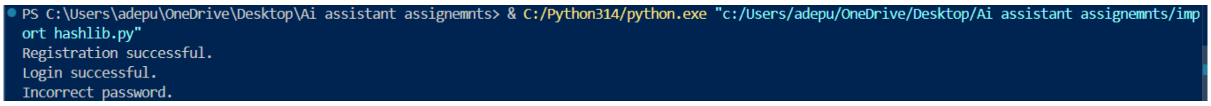


```python
import hashlib

users_db = {}

def hash_password(password):
    return hashlib.sha256(password.encode()).hexdigest()

def register(username, password):
    if username in users_db:
        print("User already exists.")
        return

    users_db[username] = hash_password(password)
    print("Registration successful.")

def login(username, password):
    if username not in users_db:
```



```
        print("User not found.")
        return

    hashed_input_password = hash_password(password)

    if users_db[username] == hashed_input_password:
        print("Login successful.")
    else:
        print("Incorrect password.")


register("admin", "Secure@123")
login("admin", "Secure@123")
login("admin", "wrongpass")
```

Output:



```
PS C:\Users\adepu\OneDrive\Desktop\Ai assistant assignemnts> & C:/Python314/python.exe "c:/Users/adepu/OneDrive/Desktop/Ai assistant assignemnts/imp
ort hashlib.py"
Registration successful.
Login successful.
Incorrect password.
```

Observation:

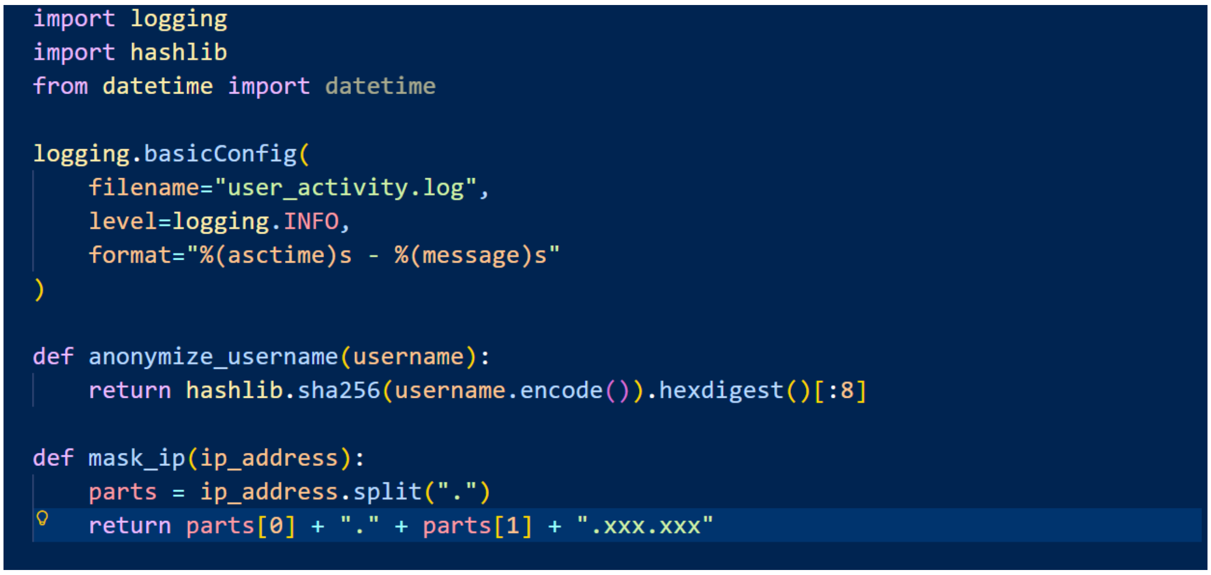
The given Python program demonstrates a basic yet secure approach to user authentication by implementing password hashing instead of plain-text password storage. During user registration, the password is converted into a hashed value using the SHA-256 algorithm before being stored in the simulated database, ensuring that sensitive credentials are not exposed. During login, the entered password is hashed again and compared with the stored hash, which prevents direct password comparison and enhances security. The program also includes validation for existing users and incorrect login attempts, improving reliability. Overall, the code follows essential security practices for authentication systems by protecting user passwords through hashing, thereby reducing the risk of credential leakage and unauthorized access.

Task Description #5 (Privacy in Data Logging)

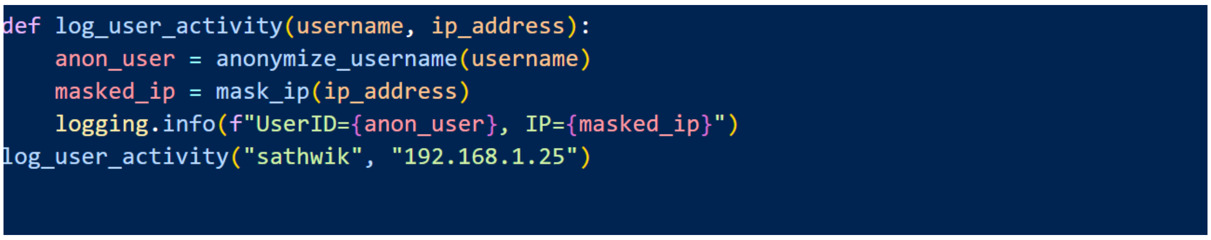
Task: Use an AI tool to generate a Python script that logs user activity (username, IP address, timestamp).

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

Code:



```python
import logging
import hashlib
from datetime import datetime

logging.basicConfig(
    filename="user_activity.log",
    level=logging.INFO,
    format="%(asctime)s - %(message)s"
)

def anonymize_username(username):
    return hashlib.sha256(username.encode()).hexdigest()[:8]

def mask_ip(ip_address):
    parts = ip_address.split(".")
    return parts[0] + "." + parts[1] + ".xxx.xxx"
```



```python
def log_user_activity(username, ip_address):
    anon_user = anonymize_username(username)
    masked_ip = mask_ip(ip_address)
    logging.info(f"UserID={anon_user}, IP={masked_ip}")
log_user_activity("sathwik", "192.168.1.25")
```

Output:

This code does not print value or output in the vs terminal it just print the log_user_activity("sathwik", "192.168.1.25")

Observation:

The given Python script demonstrates a privacy-aware approach to logging user activity by avoiding the storage of sensitive personal information in its raw form. Instead of recording actual usernames, the program anonymizes them using a hashing technique, ensuring that user identities cannot be directly revealed from the log file. Similarly, IP addresses are partially masked to prevent precise location tracking while still retaining useful network information for monitoring purposes. By logging only minimal and non-identifiable data along with timestamps, the script follows key privacy principles such as data minimization and anonymization. Overall, the implementation reduces privacy risks while maintaining effective system auditing and accountability.