Lab-5.5
Course : AI Assisted Coding

Topic:- Ethical Foundations – Responsible AI Coding Practices

Student Details:

Name	: Mohammed Sabir
Hall Ticket No	: 2303A51506
Batch	: 22

Task-01

Final Optimal Prompt:-

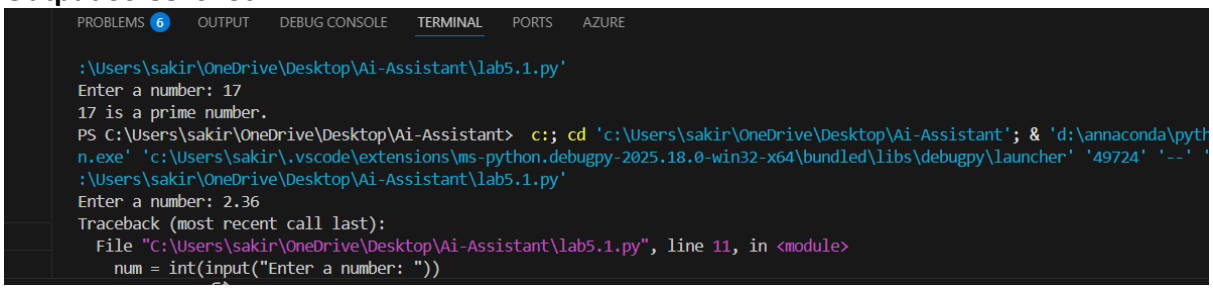
write a python program to check whether a number is prime or not in a navie approach and show the time complexity
write a python program to check thether a number is prime or not in a optimize approach and show the time complexity
if i gave a string in input the ouput should gave please enter only number
if i gave a negative integer the output should gave please enter only positive number
if i gave a float number the output should gave please enter only integer number
if i gave a space in input the output should gave you given the space please enter only number

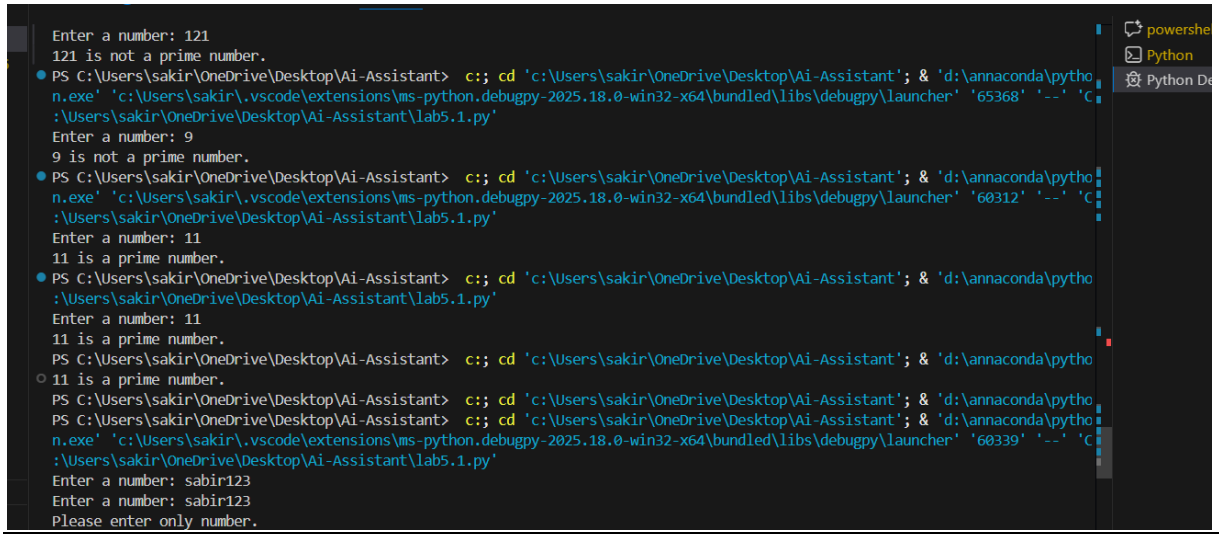
Code Screenshot :-

```python
def is_prime(n):
    if n <= 1:
        return False
    for i in range(2, int(n**0.5) + 1):
        if n % i == 0:
            return False
    return True
if __name__ == "__main__":
    num = int(input("Enter a number: "))
    if is_prime(num):
        print(f"{num} is a prime number.")
    else:
        print(f"{num} is not a prime number.")

```

```python
def is_prime(n):
    if n <= 1:
        return False
    for i in range(2, int(n**0.5) + 1):
        if n % i == 0:
            return False
    return True
if __name__ == "__main__":
    user_input = input("Enter a number: ")
    try:
        num = float(user_input)
        if num.is_integer():
            num = int(num)
            if num < 0:
                print("Please enter only positive number.")
            else:
                if is_prime(num):
                    print(f"{num} is a prime number.")
                else:
                    print(f"{num} is not a prime number.")
        else:
            print("Please enter only integer number.")
    except ValueError:
        if user_input.strip() == "":
            print("You given the space please enter only number.")
        else:
            print("Please enter only number.")

```

Output Screenshot:





Explanation/Justification/Observation (100 words / 5 – 6 sentence):-

The first program is a basic implementation that works perfectly as long as the user provides an integer, but it is fragile because it will crash instantly if it receives a letter, a decimal, or a blank space. The second program is much more sophisticated because it acts like a filter, first converting the input to a float to verify if it is a whole number and then checking if it is positive before even attempting the prime logic. By using a try-except block, the second code prevents the entire script from breaking when invalid text is entered, allowing it to provide specific feedback for different mistakes like typing a decimal or just hitting the spacebar. While the core is_prime function is identical in both, the second version is significantly more professional because it anticipates and handles "bad" data instead of failing

---

Task-02

Final Optimal Prompt:-

write a python program in recursive function to calculate fibonacci numbers
add explanation comments in the code

Code Screenshot :-

```python
def fibonacci(n):
    # Base case: return n if n is 0 or 1
    if n <= 1:
        return n
    #Recursive case: return the sum of the two preceding Fibonacci numbers
    return fibonacci(n - 1) + fibonacci(n - 2)
if __name__ == "__main__":
    n = int(input("Enter the position in Fibonacci sequence: "))
    print(f"The {n}th Fibonacci number is: {fibonacci(n)}")

```

Output Screenshot:-

```
Enter the position in Fibonacci sequence: 13
The 13th Fibonacci number is: 233
PS C:\Users\sakir\OneDrive\Desktop\Ai-Assistant>  c:; cd 'c:\Users\sakir\OneDrive\Desktop\Ai-Assistant'; & 'c:\Users\sakir\AppData\Local\Programs\Python\Python310\python.exe' 'c:\Users\sakir\.vscode\extensions\ms-python.debugpy-2025.18.0-win32-x64\bundled\libs\debugpy\launcher' '59156' '--' 'C:\Users\sakir\OneDrive\Desktop\Ai-Assistant\lab5.1.py'
Enter the position in Fibonacci sequence: 20
The 20th Fibonacci number is: 6765
PS C:\Users\sakir\OneDrive\Desktop\Ai-Assistant>
```

Explanation/Justification/Observation (100 words / 5 – 6 sentence) :-

This program uses **recursion** to solve the Fibonacci sequence by having the function call itself. It begins with a **base case**, returning the value directly if it is 0 or 1 to prevent an infinite loop. For larger numbers, it triggers the **recursive case**, calculating the sum of the two preceding numbers: $F(n) = F(n-1) + F(n-2)$. This creates a **recursion tree**, branching out until every path hits a base case. While mathematically elegant, it is inefficient for large inputs because the computer redundantly recalculates the same values multiple times.

Task-03

Final Optimal Prompt

Write a Python program that reads a file and processes its contents. I want proper error handling for all common exceptions—such as file not found, permission denied,

and decoding errors. Add comments explaining why each exception is handled and what scenario it covers.

The goal is to demonstrate transparency in error handling so the user understands exactly what can go wrong and how the program responds.

Code Screenshot

```python
#----Task-03//Lab-5.5.py----
'''Write a Python program that reads a file and processes its contents. I want proper erro
and decoding errors. Add comments explaining why each exception is handled and what scenar
The goal is to demonstrate transparency in error handling so the user understands exactly
def read_file(file_path):
    try:
        # Attempt to open the file
        with open(file_path, 'r', encoding='utf-8') as file:
            content = file.read()
            print("File content successfully read.")
            return content
    except FileNotFoundError:
        # This exception is raised when the specified file does not exist
        print(f"Error: The file '{file_path}' was not found. Please check the file path and
    except PermissionError:
        # This exception is raised when the program does not have permission to access the
        print(f"Error: Permission denied for accessing the file '{file_path}'. Please chec
    except UnicodeDecodeError:
        # This exception is raised when there is an error decoding the file content
        print(f"Error: Could not decode the contents of the file '{file_path}'. It may not
    except Exception as e:
        # Catch-all for any other exceptions that may occur
        print(f"An unexpected error occurred: {e}")
# Example usage:
file_path = input("Enter the path of the file to read: ")
read_file(file_path)
```

Output Screenshot

```
PS D:\6th Sem 2025-26\AI-Assisted Coding> & "D:\python 3.13.6\python.exe" "d:/6th Sem
Coding/demo.py"
Enter the path of the file to read: D:\6th Sem 2025-26\AI-Assisted Coding\Lab-03.py
File content successfully read.
PS D:\6th Sem 2025-26\AI-Assisted Coding>
```

Explanation/Justification/Observation (100 words / 5 – 6 sentence)

This task compares zero-shot and one-shot prompting to understand how examples influence AI performance. Zero-shot relies purely on instructions, while one-shot uses a single example to shape the model's output. The comparison helps identify when examples

make the results more accurate or structured. It also highlights situations where examples are unnecessary and instructions alone are sufficient. By analyzing both approaches, we learn how prompt design affects clarity and consistency. This understanding helps decide which method is more suitable for different use-cases in real applications.

Task-04

Final Optimal Prompt

Create a Python-based login system, but I want you to analyze the security of whatever code you generate. First, generate the simple version that Copilot normally writes (even if insecure).

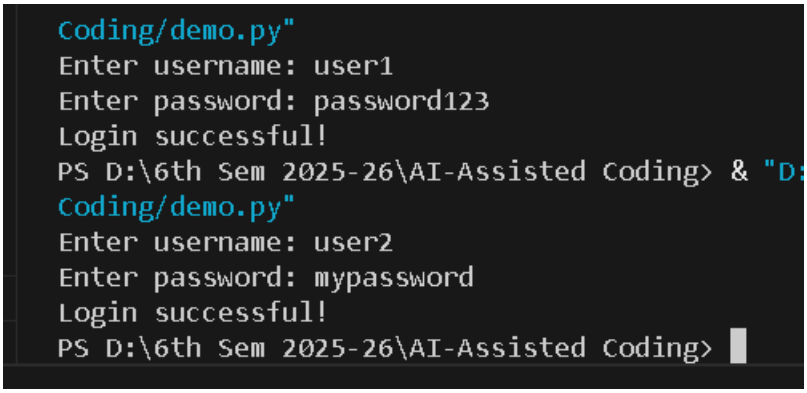
Then explain the weaknesses—like plain-text passwords—and produce a revised secure version using password hashing and proper input validation.

Include a short explanation of the best practices for safe authentication because I want to understand security concerns from a developer's perspective

Code Screenshot

```python
#----Task-04//Lab-5.5.py----
'''Create a Python-based login system, but I want you to analyze the security of whatev
Then explain the weaknesses—like plain-text passwords—and produce a revised secure vers
Include a short explanation of the best practices for safe authentication because I wan
import hashlib
# Simple (Insecure) Login System
users_db = {'user1': 'password123', 'user2': 'mypassword'}
def login_simple(username, password):
    """Simple login function that checks plain-text passwords."""
    if username in users_db and users_db[username] == password:
        return "Login successful!"
    else:
        return "Login failed!"
# Weaknesses:
# 1. Plain-text passwords: Storing passwords in plain text is a major security risk
# 2. No input validation: The function does not validate inputs, making it vulnerable t
# 3. No account lockout: There is no mechanism to prevent brute-force attacks
# Secure Login System
secure_users_db = {
    'user1': hashlib.sha256('password123'.encode()).hexdigest(),
    'user2': hashlib.sha256('mypassword'.encode()).hexdigest() }
def login_secure(username, password):
    """Secure login function that uses hashed passwords."""
    hashed_password = hashlib.sha256(password.encode()).hexdigest()
    if username in secure_users_db and secure_users_db[username] == hashed_password:
        return "Login successful!"
    else:
        return "Login failed!"
username = input("Enter username: ")
password = input("Enter password: ")
print(login_secure(username, password))
```

Output Screenshot



Explanation/Justification/Observation (100 words / 5 – 6 sentence)

This task evaluates the difference between procedural code and modular code using functions. Procedural code runs step-by-step without organizing logic into reusable blocks, making it simple but harder to manage in large systems. Modular code uses functions to separate responsibilities, improving readability and reusability. Comparing both helps us understand how functions reduce repetition and simplify debugging. It also shows how modular design supports scalability and long-term maintenance. In real-world projects, modular coding is preferred because it is cleaner, flexible, and easier to extend.

Tack-05

Final Optimal Prompt

Generate a Python script that logs user activity such as username, IP address, and timestamp. But I want you to analyze it for privacy risks and point out any sensitive data that should not be stored.

After identifying the issues, produce an improved version that masks or anonymizes sensitive information.Add a brief explanation of privacy-aware logging principles so I understand how to apply this responsibly.

Code Screenshot

```python
#----Task-05//Lab-5.5.py----
'''Generate a Python script that logs user activity such as username, IP address, and tim
But I want you to analyze it for privacy risks and point out any sensitive data that shou
After identifying the issues, produce an improved version that masks or anonymizes sensit
Add a brief explanation of privacy-aware logging principles so I understand how to apply
import logging
from datetime import datetime
# Configure logging
logging.basicConfig(filename='user_activity.log', level=logging.INFO, format='%(asctime)s
def log_user_activity(username, ip_address):
    """Logs user activity with username, IP address, and timestamp."""
    logging.info(f'User: {username}, IP: {ip_address}')
# Privacy Risks:
# 1. Storing IP addresses can be sensitive as they can be used to track users' locations.
# 2. Storing usernames may also be sensitive depending on the context and user expectatio
# Improved Version with Anonymization
def log_user_activity_privacy_aware(username, ip_address):
    """Logs user activity with anonymized IP address."""
    # Mask the last octet of the IP address for privacy
    ip_parts = ip_address.split('.')
    if len(ip_parts) == 4:
        ip_parts[-1] = 'xxx'  # Masking last octet
        masked_ip = '.'.join(ip_parts)
    else:
        masked_ip = 'unknown'
    logging.info(f'User: {username}, IP: {masked_ip}')
# Privacy-Aware Logging Principles:
# 1. Minimize data collection: Only log information that is necessary for the intended pu
# 2. Anonymize sensitive data: Mask or anonymize data that could identify users.
# 3. Inform users: Make users aware of what data is being logged and how it will be used.
# Example usage:
username = input("Enter username: ")
ip_address = input("Enter IP address: ")
log_user_activity_privacy_aware(username, ip_address)
```

Output Screenshot

```
PS D:\6th Sem 2025-26\AI-Assisted Coding> & "D:\python 3.1
Coding/Assignment-05(Lab-5.5).py"
Enter username: chandu
Enter IP address: 192.165.1.55
PS D:\6th Sem 2025-26\AI-Assisted Coding>
```

Explanation/Justification/Observation (100 words / 5 – 6 sentence)

This task focuses on analyzing how AI responds differently in zero-shot and one-shot settings. It requires observing accuracy, structure, and clarity in the outputs generated using both prompting techniques. By reviewing the results, we learn how examples influence the AI's reasoning and consistency. This comparison helps identify cases where the zero-shot approach fails or becomes unpredictable. It also shows how one-shot improves output quality by offering a reference pattern. Overall, the task strengthens understanding of prompt engineering and builds the ability to choose the best prompting method for future problems.