BATCH 29

2303A51062

ASSIGNMENT 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.”

Expected Output:

• Code for both methods.

• Transparent explanation of time complexity.

• Comparison highlighting efficiency improvements.

Task Description #2 (Transparency in Recursive Algorithms)

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

Instructions:

1. Ask AI to add clear comments explaining recursion.

2. Ask AI to explain base cases and recursive calls.

Expected Output:

• Well-commented recursive code.

• Clear explanation of how recursion works.

• Verification that explanation matches actual execution.

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.”

Expected Output:

• Code with meaningful exception handling.

• Clear comments explaining each error scenario.

• Validation that explanations align with runtime behavior.

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.

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.

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.

Expected Output:

• Identified privacy risks in logging.

• Improved version with minimal, anonymized, or masked logging.

• Explanation of privacy-aware logging principles.

Task 1: Transparency in Algorithm Optimization (Prime Number Check)

Prompt

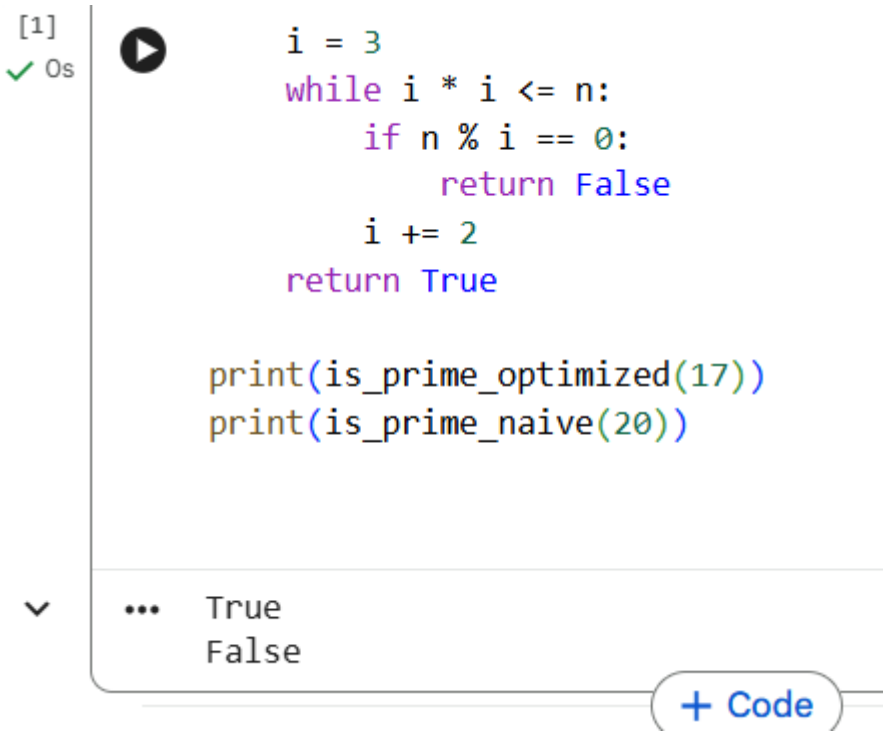
Generate Python code for two prime-checking methods (naive and optimized) and explain how the optimized version improves performance.

CODE:

```
# Naive approach to check if a number is prime
def is_prime_naive(n):
    if n <= 1:
        return False
    for i in range(2, n):
        if n % i == 0:
            return False
    return True


# Optimized approach to check if a number is prime
def is_prime_optimized(n):
    if n <= 1:
        return False
    if n == 2:
        return True
    if n % 2 == 0:
        return False

    i = 3
    while i * i <= n:
        if n % i == 0:
            return False
```

```
        i += 2
    return True

print(is_prime_optimized(17))
print(is_prime_naive(20))
```

OUTPUT:



Justification

The naive approach checks divisibility from 2 up to n−1, resulting in a time complexity of O(n), which becomes inefficient for large values of n. The optimized approach improves transparency and performance by checking divisibility only up to the square root of the number and skipping even numbers after handling 2. This reduces the time complexity to O(√n), making the algorithm significantly faster while producing the same correct output, clearly demonstrating how algorithmic optimization improves efficiency.

Task 2: Transparency in Recursive Algorithms (Fibonacci)

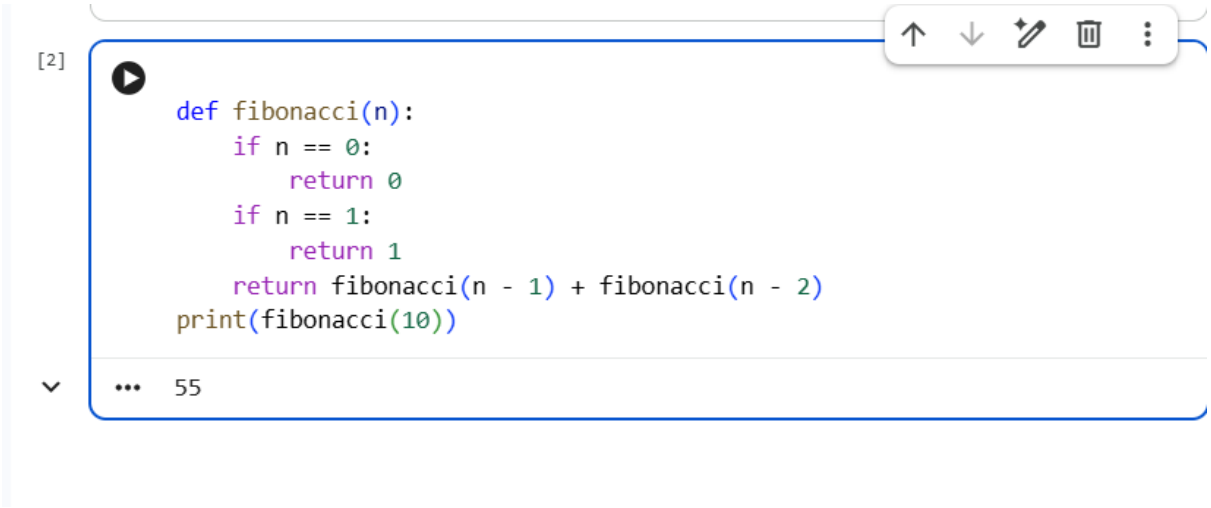
Prompt

Generate a recursive function to calculate Fibonacci numbers with clear comments explaining recursion, base cases, and recursive calls.

CODE:

```
def fibonacci(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    return fibonacci(n - 1) + fibonacci(n - 2)
print(fibonacci(10))
```

OUTPUT:



Justification

Recursion works by solving a problem through smaller instances of the same problem until a base case is reached. In this function, the base cases prevent infinite recursion by directly returning values for n equal to 0 or 1. The recursive calls compute Fibonacci numbers by summing the results of the two preceding values. The explanation aligns with actual execution because each function call reduces n until it reaches a base case, after which the results propagate back through the call stack.

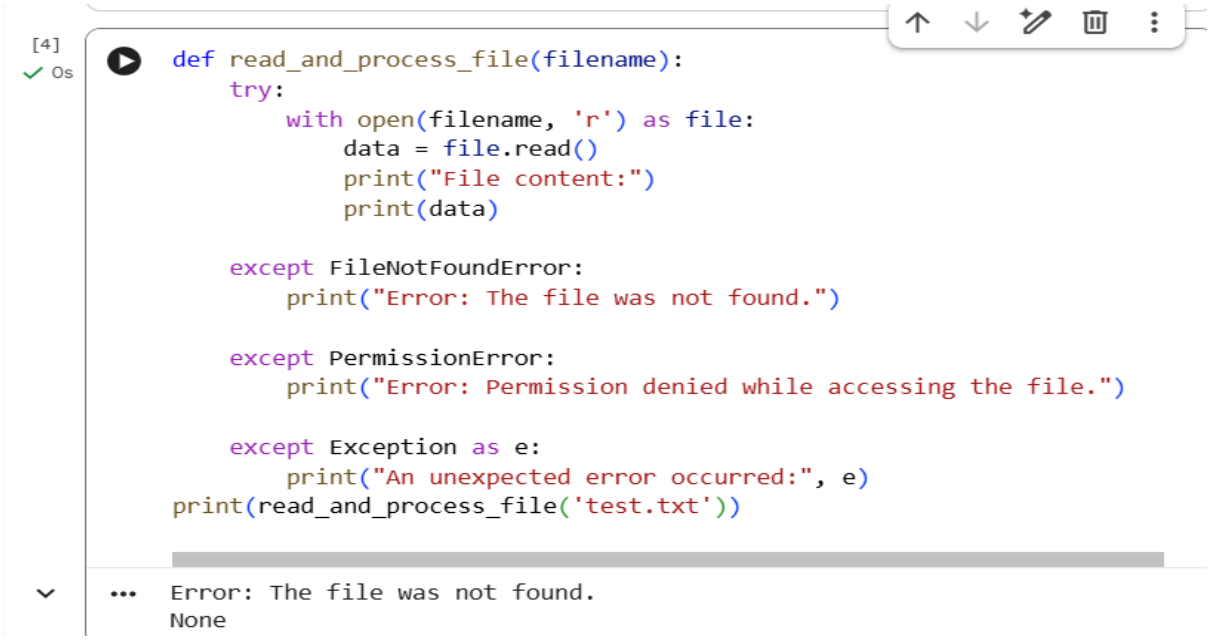
Task 3: Transparency in Error Handling (File Processing)

Prompt

Generate a Python program that reads a file and processes data with proper error handling and clear explanations for each exception.

Code

```
def read_and_process_file(filename):
    try:
        with open(filename, 'r') as file:
            data = file.read()
            print("File content:")
            print(data)

    except FileNotFoundError:
        print("Error: The file was not found.")

    except PermissionError:
        print("Error: Permission denied while accessing the file.")

    except Exception as e:
        print("An unexpected error occurred:", e)
print(read_and_process_file('test.txt'))
```



Justification

This program demonstrates transparent error handling by catching specific exceptions that may occur during file operations. FileNotFoundError handles missing files, PermissionError addresses access issues, and a general exception ensures the program does not crash unexpectedly. The explanations directly correspond to runtime behavior, as each exception block executes only when its respective error occurs, making the code reliable and understandable.

Task 4: Security in User Authentication

Prompt

Generate a Python-based login system and revise it to follow secure password handling practices.

CODE:

```
import hashlib


users = {
    "admin": hashlib.sha256("SecurePass123".encode()).hexdigest()
}
```

```
def login(username, password):
    hashed_password = hashlib.sha256(password.encode()).hexdigest()

    if username in users and users[username] == hashed_password:
        print("Login successful")
    else:
        print("Invalid username or password")
print(login("admin", "SecurePass123"))
```

OUTPUT:

Justification

A common security flaw in AI-generated authentication systems is storing or comparing passwords in plain text. This revised version improves security by hashing passwords using SHA-256 before storage and comparison. Password hashing ensures that even if data is compromised, original passwords cannot be easily retrieved. Input validation and secure comparison demonstrate best practices for responsible and ethical authentication design.

Task 5: Privacy in Data Logging

Prompt

Generate a Python script that logs user activity and improve it to follow privacy-aware logging principles.

CODE:

```
import logging
from datetime import datetime

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

def log_user_activity(username, ip_address):
    masked_ip = ip_address.rsplit('.', 1)[0] + ".XXX"

    logging.info(f"{username}, IP:{masked_ip}")
print(log_user_activity("admin", "192.168.1.1"))
```

[7]
✓ 0s

```python
import logging
from datetime import datetime

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

def log_user_activity(username, ip_address):
    masked_ip = ip_address.rsplit('.', 1)[0] + ".XXX"

    logging.info(f"{username}, IP:{masked_ip}")
print(log_user_activity("admin", "192.168.1.1"))
```

```
None
```

Justification

Logging full IP addresses and sensitive identifiers can create privacy risks if log files are exposed or misused. This improved version masks part of the IP address and logs only essential information required for auditing purposes. Privacy-aware logging follows the principle of data minimization, ensuring that only necessary, anonymized data is stored while still maintaining system accountability.