AI ASSISTANT CODING ASSIGNMENT-5.5

Name: SANJAY KARUPOTHULA

Enrollment no: 2303A52337

Batch: 34

Semester: VI

Branch: CSE(AIML)

Lab 5: Ethical Foundations – Responsible AI Coding Practices

Task Description #1 (Transparency in Algorithm Optimization)

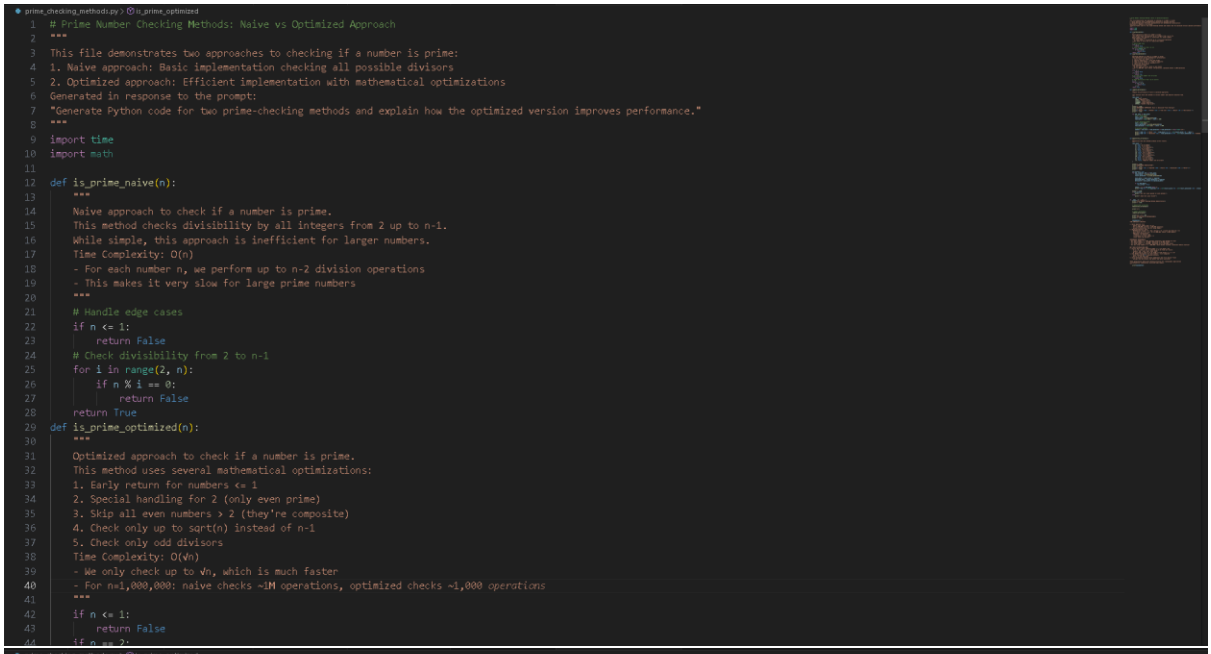
Task: Use AI to generate two solutions for checking prime numbers:

• Naive approach(basic)

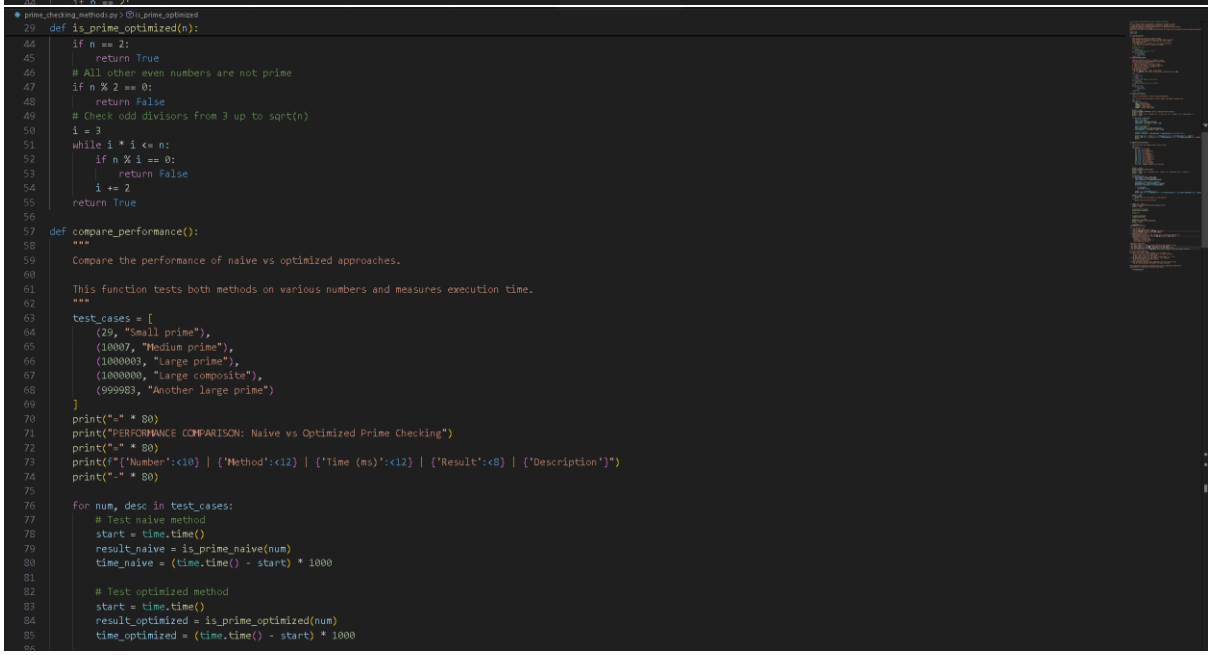
• Optimized approach

Prompt:

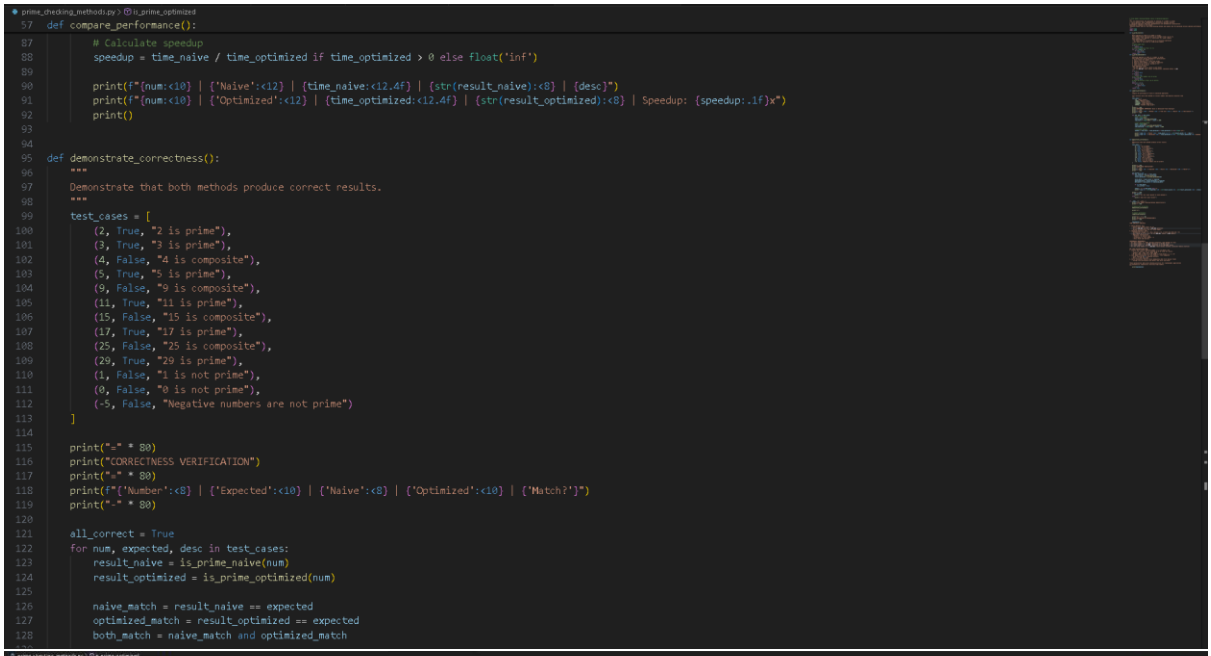
“Generate Python code for two prime-checking methods and explain how the optimized version improves performance.”



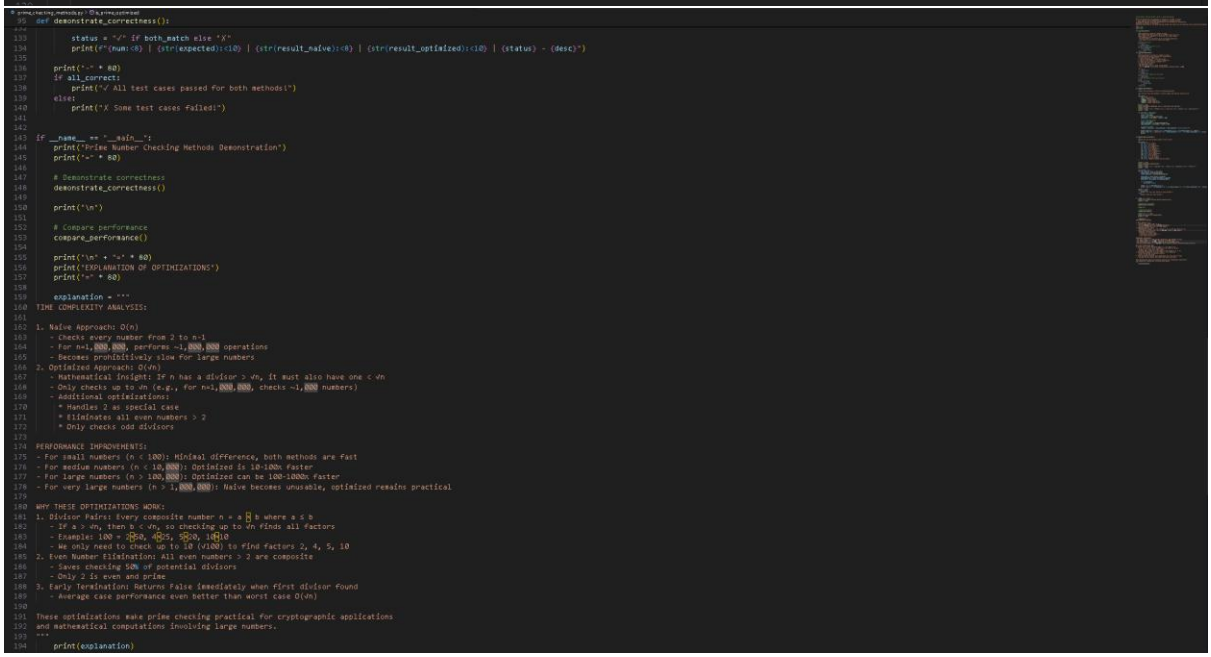
```
prime_checking_methods.py > is_prime_optimized
# Prime Number Checking Methods: Naive vs Optimized Approach
"""
This file demonstrates two approaches to checking if a number is prime:
1. Naive approach: Basic implementation checking all possible divisors
2. Optimized approach: Efficient implementation with mathematical optimizations
Generated in response to the prompt:
"Generate Python code for two prime-checking methods and explain how the optimized version improves performance."
"""
import time
import math

def is_prime_naive(n):
    """
    Naive approach to check if a number is prime.
    This method checks divisibility by all integers from 2 up to n-1.
    While simple, this approach is inefficient for larger numbers.
    Time Complexity: O(n)
    - For each number n, we perform up to n-2 division operations
    - This makes it very slow for large prime numbers
    """
    # Handle edge cases
    if n <= 1:
        return False
    # Check divisibility from 2 to n-1
    for i in range(2, n):
        if n % i == 0:
            return False
    return True
def is_prime_optimized(n):
    """
    Optimized approach to check if a number is prime.
    This method uses several mathematical optimizations:
    1. Early return for numbers <= 1
    2. Special handling for 2 (only even prime)
    3. Skip all even numbers > 2 (they're composite)
    4. Check only up to sqrt(n) instead of n-1
    5. Check only odd divisors
    Time Complexity: O(√n)
    - We only check up to √n, which is much faster
    - For n=1,000,000: naive checks ~1M operations, optimized checks ~1,000 operations
    """
    if n <= 1:
        return False
    if n == 2:
```



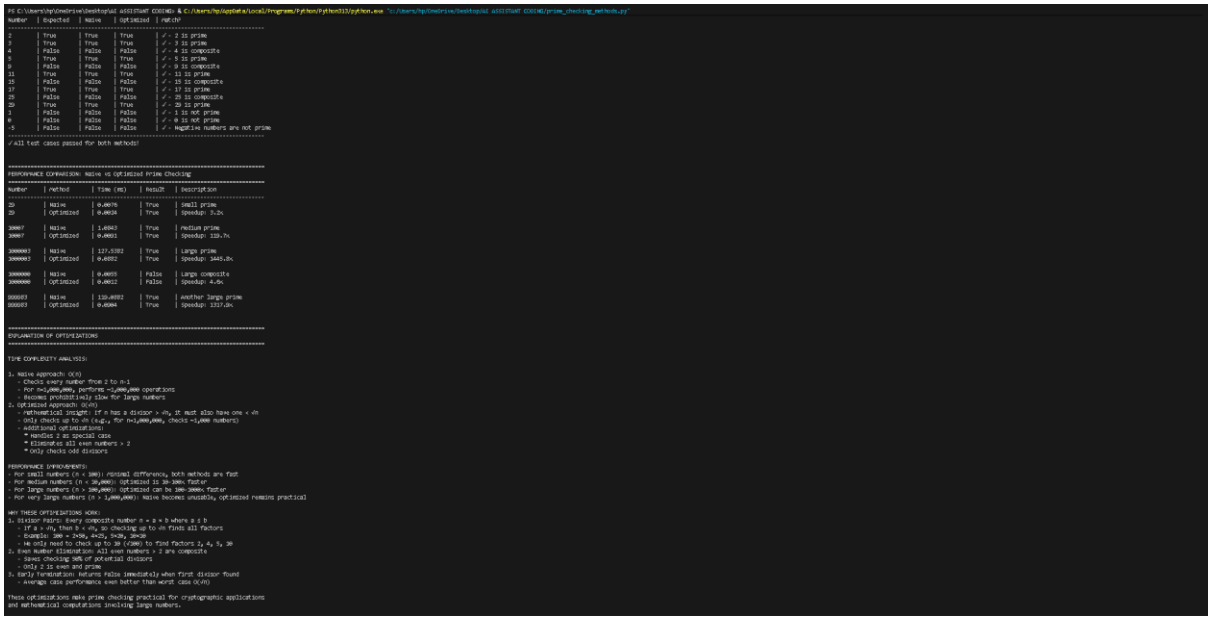
```
prime_checking_methods.py > is_prime_optimized
def is_prime_optimized(n):
    if n == 2:
        return True
    # All other even numbers are not prime
    if n % 2 == 0:
        return False
    # Check odd divisors from 3 up to sqrt(n)
    i = 3
    while i * i <= n:
        if n % i == 0:
            return False
        i += 2
    return True

def compare_performance():
    """
    Compare the performance of naive vs optimized approaches.

    This function tests both methods on various numbers and measures execution time.
    """
    test_cases = [
        (29, "Small prime"),
        (10007, "Medium prime"),
        (1000003, "Large prime"),
        (1000000, "Large composite"),
        (999983, "Another large prime")
    ]
    print("=" * 80)
    print("PERFORMANCE COMPARISON: Naive vs Optimized Prime Checking")
    print("=" * 80)
    print(f"{'Number':<10} | {'Method':<12} | {'Time (ms)':<12} | {'Result':<8} | {'Description'}")
    print("-" * 80)

    for num, desc in test_cases:
        # Test naive method
        start = time.time()
        result_naive = is_prime_naive(num)
        time_naive = (time.time() - start) * 1000

        # Test optimized method
        start = time.time()
        result_optimized = is_prime_optimized(num)
        time_optimized = (time.time() - start) * 1000
```



```
prime_checking_methods.py > is_prime_optimized
def compare_performance():
        # Calculate speedup
        speedup = time_naive / time_optimized if time_optimized > 0 else float('inf')

        print(f"{num:<10} | {'Naive':<12} | {time_naive:<12.4f} | {str(result_naive):<8} | {desc}")
        print(f"{num:<10} | {'Optimized':<12} | {time_optimized:<12.4f} | {str(result_optimized):<8} | Speedup: {speedup:.1f}x")
        print()


def demonstrate_correctness():
    """
    Demonstrate that both methods produce correct results.
    """
    test_cases = [
        (2, True, "2 is prime"),
        (3, True, "3 is prime"),
        (4, False, "4 is composite"),
        (5, True, "5 is prime"),
        (9, False, "9 is composite"),
        (11, True, "11 is prime"),
        (15, False, "15 is composite"),
        (17, True, "17 is prime"),
        (25, False, "25 is composite"),
        (29, True, "29 is prime"),
        (1, False, "1 is not prime"),
        (0, False, "0 is not prime"),
        (-5, False, "Negative numbers are not prime")
    ]

    print("=" * 80)
    print("CORRECTNESS VERIFICATION")
    print("=" * 80)
    print(f"{'Number':<8} | {'Expected':<10} | {'Naive':<8} | {'Optimized':<10} | {'Match?'}")
    print("-" * 80)

    all_correct = True
    for num, expected, desc in test_cases:
        result_naive = is_prime_naive(num)
        result_optimized = is_prime_optimized(num)

        naive_match = result_naive == expected
        optimized_match = result_optimized == expected
        both_match = naive_match and optimized_match
```



```
prime_checking_methods.py > demonstrate_correctness
def demonstrate_correctness():
        status = "✓" if both_match else "✗"
        print(f"{num:<8} | {str(expected):<10} | {str(result_naive):<8} | {str(result_optimized):<10} | {status} - {desc}")

    print("-" * 80)
    if all_correct:
        print("✓ All test cases passed for both methods!")
    else:
        print("✗ Some test cases failed!")


if __name__ == "__main__":
    print("Prime Number Checking Methods Demonstration")
    print("=" * 80)

    # Demonstrate correctness
    demonstrate_correctness()

    print("\n")

    # Compare performance
    compare_performance()

    print("\n" + "=" * 80)
    print("EXPLANATION OF OPTIMIZATIONS")
    print("=" * 80)

    explanation = """
TIME COMPLEXITY ANALYSIS:

1. Naive Approach: O(n)
   - Checks every number from 2 to n-1
   - For n=1,000,000, performs ~1,000,000 operations
   - Becomes prohibitively slow for large numbers
2. Optimized Approach: O(√n)
   - Mathematical insight: If n has a divisor > √n, it must also have one < √n
   - Only checks up to √n (e.g., for n=1,000,000, checks ~1,000 numbers)
   - Additional optimizations:
     * Handles 2 as special case
     * Eliminates all even numbers > 2
     * Only checks odd divisors

PERFORMANCE IMPROVEMENTS:
- For small numbers (n < 100): Minimal difference, both methods are fast
- For medium numbers (n < 10,000): Optimized is 10-100x faster
- For large numbers (n > 100,000): Optimized can be 100-1000x faster
- For very large numbers (n > 1,000,000): Naive becomes unusable, optimized remains practical

WHY THESE OPTIMIZATIONS WORK:
1. Divisor Pairs: Every composite number n = a × b where a ≤ b
   - If a > √n, then b < √n, so checking up to √n finds all factors
   - Example: 100 = 2×50, 4×25, 5×20, 10×10
   - We only need to check up to 10 (√100) to find factors 2, 4, 5, 10
2. Even Number Elimination: All even numbers > 2 are composite
   - Saves checking 50% of potential divisors
   - Only 2 is even and prime
3. Early Termination: Returns False immediately when first divisor found
   - Average case performance even better than worst case O(√n)

These optimizations make prime checking practical for cryptographic applications
and mathematical computations involving large numbers.
"""
    print(explanation)
```



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.

```python
def fibonacci(n):
    # Base case 1: If n is 0, return 0
    if n == 0:
        return 0
    # Base case 2: If n is 1, return 1
    if n == 1:
        return 1
    # Recursive case: For n > 1, return the sum of the two preceding Fibonacci numbers
    # This is the recursive call: fibonacci(n-1) + fibonacci(n-2)
    return fibonacci(n - 1) + fibonacci(n - 2)

# Example usage
if __name__ == "__main__":
    # Test the function with some values
    print("Fibonacci(0):", fibonacci(0))  # Should print 0
    print("Fibonacci(1):", fibonacci(1))  # Should print 1
    print("Fibonacci(5):", fibonacci(5))  # Should print 5 (0,1,1,2,3,5)
    print("Fibonacci(10):", fibonacci(10))  # Should print 55
    explanation="""
Clear Explanation of How Recursion Works:

Recursion is a technique where a function calls itself to solve a problem by breaking it into smaller sub-problems.
In the Fibonacci sequence:
- Each number is the sum of the previous two numbers
- This relationship is naturally suited for recursion
Base Cases:
- fibonacci(0) returns 0
- fibonacci(1) returns 1
These base cases are essential because they terminate the recursive calls.
Recursive Case:
For n > 1, the function returns:
- fibonacci(n-1) + fibonacci(n-2)
Each recursive call reduces the value of n, moving closer to the base cases.
Eventually, all calls reach a base case
"""
    print(explanation)
```

```
PS C:\Users\hp\OneDrive\Desktop\AI ASSISTANT CODING> & C:/Users/hp/AppData/Local/Programs/Python/Python313/python.exe "c:/Users/hp/OneDrive/Desktop/AI ASSISTANT CODING/fibonacci_recursive.py"
Fibonacci(0): 0
Fibonacci(1): 1
Fibonacci(5): 5
Fibonacci(10): 55

Clear Explanation of How Recursion Works:

Recursion is a technique where a function calls itself to solve a problem by breaking it into smaller sub-problems.
In the Fibonacci sequence:
- Each number is the sum of the previous two numbers
- This relationship is naturally suited for recursion
Base Cases:
- fibonacci(0) returns 0
- fibonacci(1) returns 1
These base cases are essential because they terminate the recursive calls.
Recursive Case:
For n > 1, the function returns:
- fibonacci(n-1) + fibonacci(n-2)
Each recursive call reduces the value of n, moving closer to the base cases.
Eventually, all calls reach a base case

PS C:\Users\hp\OneDrive\Desktop\AI ASSISTANT CODING>
```

Task Description #3 (Transparency in Error Handling)

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

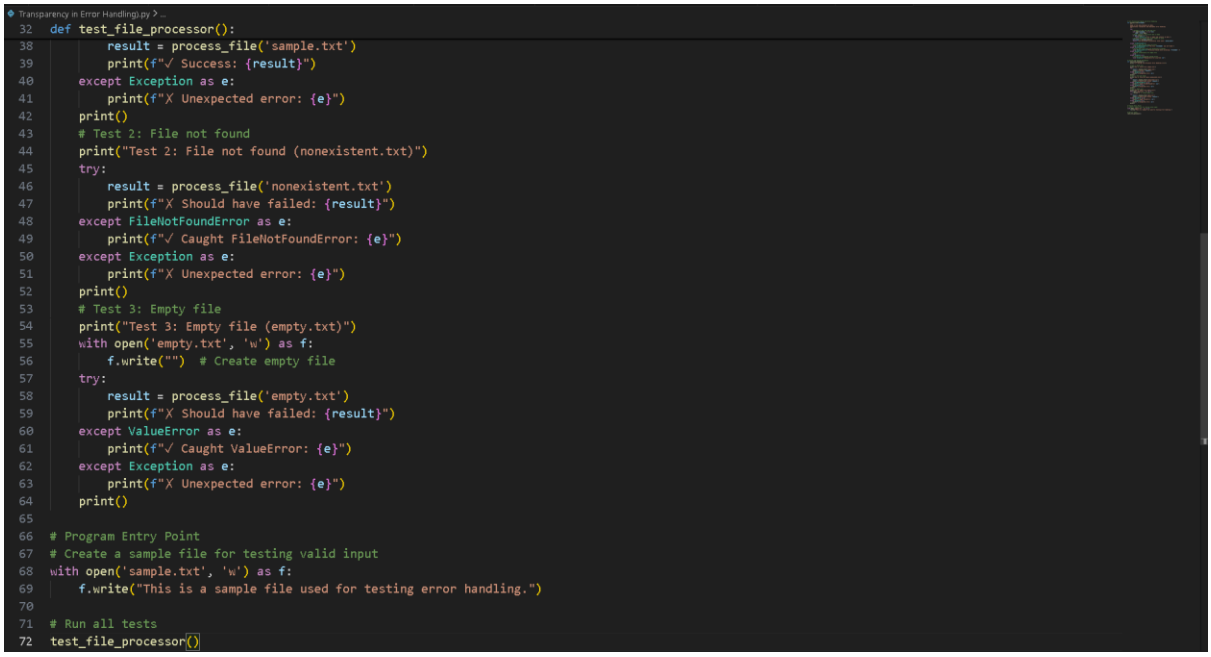
Prompt:

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

```python
# File Processing Program with Error Handling
def process_file(filename):
    """
    Reads a file and processes its data.
    Demonstrates transparent and meaningful error handling.
    """
    try:
        # Attempt to open and read the file
        with open(filename, 'r') as file:
            data = file.read()
        # Error scenario: file exists but is empty
        if data.strip() == "":
            raise ValueError("File is empty and contains no data.")
        # Example data processing: count number of words
        word_count = len(data.split())
        return f"File processed successfully. Word count: {word_count}"

    except FileNotFoundError:
        # Occurs when the file does not exist
        raise FileNotFoundError(f"The file '{filename}' was not found.")
    except PermissionError:
        # Occurs when file exists but permission is denied
        raise PermissionError(f"Permission denied while accessing '{filename}'.")
    except ValueError:
        # Raised intentionally for empty file
        raise
    except Exception as e:
        # Catches any unexpected runtime errors
        raise Exception(f"Unexpected error occurred: {e}")

# Testing and Runtime Validation
def test_file_processor():
    print("=== Testing File Processor Error Handling ===\n")

    # Test 1: Valid file
    print("Test 1: Valid file (sample.txt)")
```

‘

```
PS C:\Users\hp\OneDrive\Desktop\AI ASSISTANT CODING> & C:/Users/hp/AppData/Local/Programs/Python/Python313/python.exe "c:/Users/hp/OneDrive/Desktop/AI ASSISTANT CODING/Transparency in
"
=== Testing File Processor Error Handling ===

Test 1: Valid file (sample.txt)
✓ Success: File processed successfully. Word count: 10

Test 2: File not found (nonexistent.txt)
✓ Caught FileNotFoundError: The file 'nonexistent.txt' was not found.

Test 3: Empty file (empty.txt)
✓ Caught ValueError: File is empty and contains no data.

PS C:\Users\hp\OneDrive\Desktop\AI ASSISTANT CODING>
```



```
Transparency in Error Handling).py > ...
def test_file_processor():
        result = process_file('sample.txt')
        print(f"✓ Success: {result}")
    except Exception as e:
        print(f"✗ Unexpected error: {e}")
    print()
    # Test 2: File not found
    print("Test 2: File not found (nonexistent.txt)")
    try:
        result = process_file('nonexistent.txt')
        print(f"✗ Should have failed: {result}")
    except FileNotFoundError as e:
        print(f"✓ Caught FileNotFoundError: {e}")
    except Exception as e:
        print(f"✗ Unexpected error: {e}")
    print()
    # Test 3: Empty file
    print("Test 3: Empty file (empty.txt)")
    with open('empty.txt', 'w') as f:
        f.write("")  # Create empty file
    try:
        result = process_file('empty.txt')
        print(f"✗ Should have failed: {result}")
    except ValueError as e:
        print(f"✓ Caught ValueError: {e}")
    except Exception as e:
        print(f"✗ Unexpected error: {e}")
    print()

# Program Entry Point
# Create a sample file for testing valid input
with open('sample.txt', 'w') as f:
    f.write("This is a sample file used for testing error handling.")

# Run all tests
test_file_processor()
```

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.

```python
import hashlib
import re

# Secure user database with hashed passwords
users = {
    "admin": hashlib.sha256("Admin@123".encode()).hexdigest(),
    "user": hashlib.sha256("User@123".encode()).hexdigest()
}
def validate_input(username, password):
    """
    Validates username and password format
    """
    if not username or not password:
        return False, "Username and password cannot be empty"
    if len(password) < 8:
        return False, "Password must be at least 8 characters long"
    # Password complexity check
    if not re.search(r"[A-Z]", password) or not re.search(r"[0-9]", password):
        return False, "Password must contain at least one uppercase letter and one number"
    return True, ""

def login(username, password):
    """
    Secure login function using hashed passwords
    """
    # Input validation
    valid, message = validate_input(username, password)
    if not valid:
        return message
    # Hash the entered password
    hashed_password = hashlib.sha256(password.encode()).hexdigest()
    # Compare hashed passwords
    if username in users and users[username] == hashed_password:
        return "Login successful"
    else:
        return "Invalid username or password"

username = input("Enter username: ")
password = input("Enter password: ")
print(login(username, password))

explanation = """
Short Note: Best Practices for Secure Authentication
- Always hash passwords, never store them in plain text
- Use strong hashing algorithms like bcrypt, Argon2, or PBKDF2
- Enforce minimum password length and complexity
- Validate all user inputs to prevent attacks
- Limit login attempts to prevent brute-force attacks
- Use HTTPS for transmitting credentials"""
print(explanation)
```

```
PS C:\Users\hp\OneDrive\Desktop\AI ASSISTANT CODING> & C:/Users/hp/AppData/Local/Programs/Python/Python313/python.exe "c:/Users/hp/OneDrive/Desktop/AI ASSISTANT CODING/Security in
.py"
Enter username: admin
Enter password: Admin@123
Login successful

Short Note: Best Practices for Secure Authentication
- Always hash passwords, never store them in plain text
- Use strong hashing algorithms like bcrypt, Argon2, or PBKDF2
- Enforce minimum password length and complexity
- Validate all user inputs to prevent attacks
- Limit login attempts to prevent brute-force attacks
- Use HTTPS for transmitting credentials
PS C:\Users\hp\OneDrive\Desktop\AI ASSISTANT CODING>
```

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.

Privacy in Data Logging.py > ...

```python
import datetime
import hashlib

def mask_ip(ip_address):
    """
    Masks the last octet of an IPv4 address
    Example: 192.168.1.24 → 192.168.1.***
    """
    parts = ip_address.split(".")
    parts[-1] = "***"
    return ".".join(parts)
def anonymize_username(username):
    """
    Hashes the username so it cannot be reversed
    """
    return hashlib.sha256(username.encode()).hexdigest()[:10]
def log_user_activity(username, ip_address):
    """
    Logs user activity using anonymized and minimal data
    """
    timestamp = datetime.datetime.now().strftime("%Y-%m-%d %H:%M:%S")
    masked_ip = mask_ip(ip_address)
    anon_user = anonymize_username(username)
    log_entry = f"{timestamp} | UserID: {anon_user} | IP: {masked_ip}\n"
    with open("activity.log", "a") as log_file:
        log_file.write(log_entry)
# Example usage
log_user_activity("sathwik", "192.168.1.24")

explanation = """
```

```
PS C:\Users\hp\OneDrive\Desktop\AI ASSISTANT CODING> & C:/Users/hp/AppData/Local/Programs/Python/Python313/python.exe "c:/Users/hp/OneDrive/Desktop/AI ASSISTANT CODING/Privacy in Data Logging.py"
Enter username: sathwik
Enter IP address: 192.168.1.24
User activity logged securely.

    Identified Privacy Risks
- Plain-text usernames: Directly identifies a specific user (PII).
- Full IP address logging: IP addresses can reveal location and user identity.
- No data minimization: Logs more data than required for monitoring.
- No anonymization or masking: Data can be misused if log files are leaked.
- Unlimited retention: No control over how long sensitive data is stored.
    Privacy-Aware Logging Principles (Short Note)
- Data minimization: Log only what is necessary
- Anonymization: Remove or hash personal identifiers
- Masking: Obscure sensitive parts of data (IP, IDs)
- Least privilege: Restrict access to log files
- Retention control: Avoid storing logs longer than needed
- Compliance awareness: Align with privacy laws (GDPR, DPDP)
PS C:\Users\hp\OneDrive\Desktop\AI ASSISTANT CODING> & C:/Users/hp/AppData/Local/Programs/Python/Python313/python.exe "c:/Users/hp/OneDrive/Desktop/AI ASSISTANT CODING/Privacy in Data Logging.py"
```