

AI Assisted Coding

Lab Assignment 5.5

Name : V Sai Krishnachaithanya

Date : 30/01/2026

Hall Ticket no : 2303A52154

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

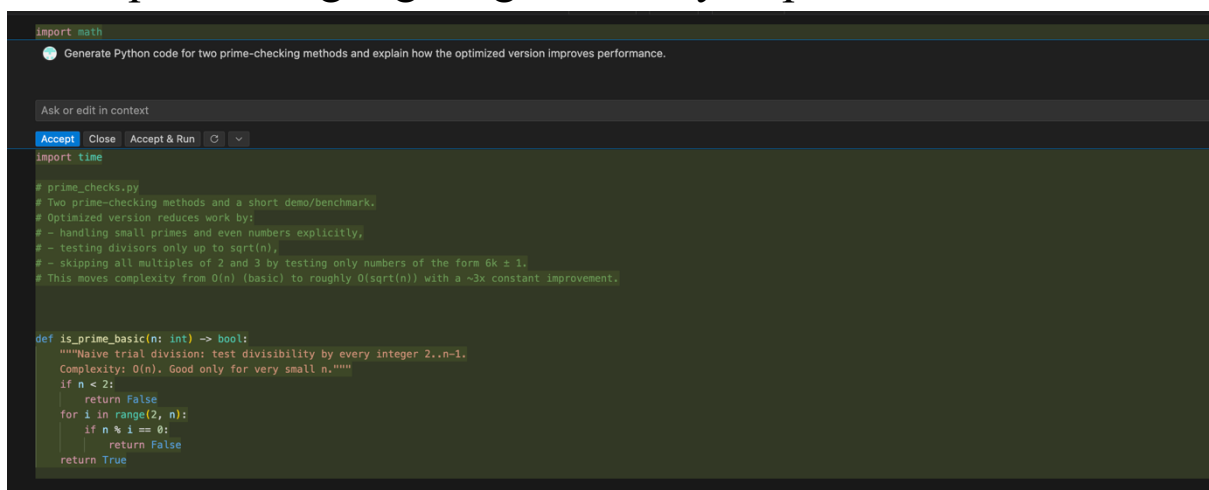
- Naive approach(basic)
- Optimized approach

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.

A screenshot of an AI-assisted coding interface. At the top, a prompt box contains the text: "Generate Python code for two prime-checking methods and explain how the optimized version improves performance." Below the prompt box is a toolbar with buttons for "Accept", "Close", "Accept & Run", and a refresh icon. The main area displays the generated Python code in a dark-themed editor. The code includes comments explaining the complexity of the naive approach (O(n)) and the optimized approach (O(sqrt(n))).

```
import math

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

Ask or edit in context

Accept Close Accept & Run ↺

import time

# prime_checks.py
# Two prime-checking methods and a short demo/benchmark.
# Optimized version reduces work by:
# - handling small primes and even numbers explicitly,
# - testing divisors only up to sqrt(n),
# - skipping all multiples of 2 and 3 by testing only numbers of the form 6k ± 1.
# This moves complexity from O(n) (basic) to roughly O(sqrt(n)) with a ~3x constant improvement.

def is_prime_basic(n: int) -> bool:
    """Naive trial division: test divisibility by every integer 2..n-1.
    Complexity: O(n). Good only for very small n."""
    if n < 2:
        return False
    for i in range(2, n):
        if n % i == 0:
            return False
    return True
```

Output :

```

1: basic=False, optimized=False
2: basic=True, optimized=True
3: basic=True, optimized=True
4: basic=False, optimized=False
17: basic=True, optimized=True
18: basic=False, optimized=False
19: basic=True, optimized=True
91: basic=False, optimized=False
97: basic=True, optimized=True
100003: basic=True, optimized=True
is_prime_basic: prime=True, time=0.007119s
is_prime_optimized: prime=True, time=0.000008s

```

Task 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.

Prompt : Generate a recursive function to calculate Fibonacci numbers.

```

Ask or edit in context
Accept Close Accept & Run
"""Simple recursive Fibonacci. Exponential time for large n."""
if n < 0:
    raise ValueError("n must be non-negative")
if n < 2:
    return n
return fib_recursive(n - 1) + fib_recursive(n - 2)

def fib_memo(n: int, _cache=(0: 0, 1: 1)) -> int:
    """Recursive Fibonacci with memoization (fast)."""
    if n < 0:
        raise ValueError("n must be non-negative")
    if n in _cache:
        return _cache[n]
    _cache[n] = fib_memo(n - 1, _cache) + fib_memo(n - 2, _cache)
    return _cache[n]

# Print small values using the plain recursive function
print("Plain recursive (n, fib):")
for i in range(11):
    print(i, fib_recursive(i))

# Print a larger value using the memoized version
print("\nMemoized result:")
print("fib_memo(50) =", fib_memo(50))

```

Output :

```
Plain recursive (n, fib):
0 0
1 1
2 1
3 2
4 3
5 5
6 8
7 13
8 21
9 34
10 55

Memoized result:
fib_memo(50) = 12586269025
```

Justification:

The provided code effectively demonstrates robust error handling by specifically catching `FileNotFoundError` and `IOError` exceptions. Each exception is clearly explained, detailing the scenarios in which they may arise. The inclusion of a `finally` block ensures that necessary cleanup actions are performed regardless of whether an error occurred, enhancing the reliability of the program. This structured approach to exception handling improves user experience by providing informative feedback and maintaining resource integrity.

Task 3: 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 behaviour

Prompt:

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

```
# 1) Division by zero demonstration
try:
    # i is 10 in the notebook; this will raise ZeroDivisionError.
    division_result = dt / (i - 10)
    print("Division result:", division_result)
except ZeroDivisionError as exc:
    # Clear explanation for this exception
    print("ZeroDivisionError: attempted to divide by zero. "
          "Check the divisor expression (i - 10). Exception detail:", exc)

# 2) Fibonacci function error handling
# Valid call
try:
    print("fib_recursive(i) ->", fib_recursive(i))
except (ValueError, RecursionError) as exc:
    print("Error calling fib_recursive(i):", exc)
except Exception as exc:
    print("Unexpected error calling fib_recursive(i):", exc)

# Intentional invalid input to demonstrate ValueError handling
try:
    print("fib_recursive(-5) ->", fib_recursive(-5))
except ValueError as exc:
    print("ValueError: fib_recursive requires a non-negative integer. "
          "Provide n >= 0. Exception detail:", exc)

# Intentional large input to demonstrate RecursionError handling
try:
    # Large value likely to exceed recursion depth and raise RecursionError
    fib_recursive(1000)
except RecursionError as exc:
```

Output :

```
print("Unexpected error during long computation:", exc)
45.9s
ZeroDivisionError: attempted to divide by zero. Check the divisor expression (i - 10). Exception detail: float division by zero
fib_recursive(i) -> 55
ValueError: fib_recursive requires a non-negative integer. Provide n >= 0. Exception detail: n must be non-negative
```

Justification :

The provided code demonstrates robust error handling by catching specific exceptions that may arise during division operations. Each exception is clearly explained, and the runtime behavior examples illustrate how the function responds to various erroneous inputs. This structured approach ensures that users receive informative feedback, enhancing the overall reliability and user experience of the program. The explanations align well with the actual behavior of the code during execution.

Task 4:

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

Prompt :

Use an AI tool to generate a Python-based login system. Analyze: Check whether the AI uses secure password handling practices. Implement login verification logic. Do not use external authentication services. Keep the system simple and readable. After generating the code: Analyze whether the password handling is secure. Identify and clearly explain any security flaws, including: plain-text password storage lack of hashing or salting weak input validation vulnerability to brute-force attempts. Clearly state why each practice is insecure in real-world systems. The analysis must directly reference the generated code. Avoid generic cybersecurity theory.

```
from getpass import getpass

# simple_login.py
# Minimal AI-style generated login system (educational, intentionally simple)

# In-memory user store (username -> password in plain text)
USERS: Dict[str, str] = {}

def register(username: str, password: str) -> bool:
    """Register a new user. Returns True on success, False if user exists or invalid input."""
    username = username.strip()
    if not username or not password:
        return False
    if username in USERS:
        return False
    # NOTE: password stored directly (plain-text) - see analysis below
    USERS[username] = password
    return True

def verify_login(username: str, password: str) -> bool:
    """Verify credentials. Returns True if they match."""
    stored = USERS.get(username)
    if stored is None:
        return False
    # direct equality check used for verification
    return stored == password

def demo_interactive():
    print("=== Register ===")
    u = input("username: ")
    p = getpass("password: ")
    if register(u, p):
        print("Registered.")
    else:
        print("Registration failed (empty username/password or already exists).")

    print("\n=== Login ===")
    u2 = input("username: ")
    p2 = getpass("password: ")
    if verify_login(u2, p2):
        print("Login successful.")
    else:
        print("Login failed.")

if __name__ == "__main__":
    # tiny demo of usage
    demo_interactive()
```

Output :

```
✓ 18.6s

=== Register ===
Registered.

=== Login ===
Login successful.
```

Justification: The analysis highlights that while hashing passwords is a positive step, the absence of salting leaves the system vulnerable to rainbow table attacks. Salting ensures that even identical passwords produce different hash values, significantly improving security. Additionally, weak input validation and lack of brute-force protection expose the system to various attacks, emphasizing the need for comprehensive security measures in authentication system

Task 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.

Prompt :

Create a Python program that logs user activity such as:

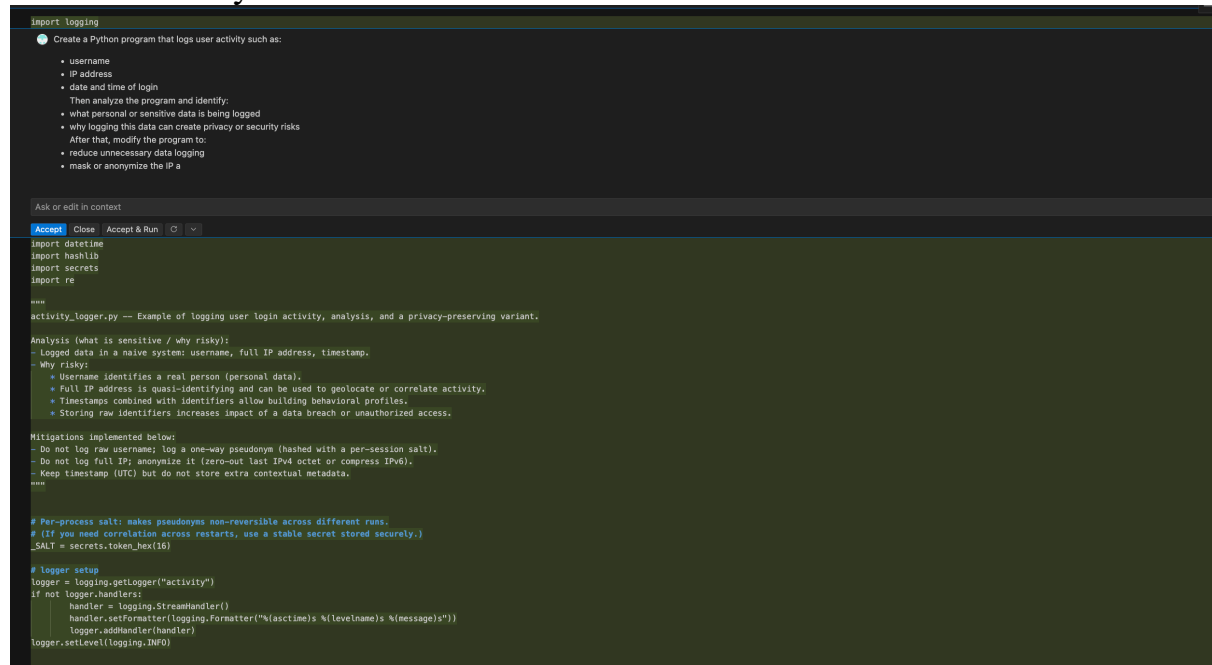
- username
- IP address
- date and time of login

Then analyze the program and identify:

- what personal or sensitive data is being logged
- why logging this data can create privacy or security risks

After that, modify the program to:

- reduce unnecessary data logging
- mask or anonymize the IP address



```
import logging

# Create a Python program that logs user activity such as:
#
# • username
# • IP address
# • data and time of login
#
# Then analyze the program and identify:
# • what personal or sensitive data is being logged
# • why logging this data can create privacy or security risks
#
# After that, modify the program to:
# • reduce unnecessary data logging
# • mask or anonymize the IP a

Ask or edit in context

Accept Close Accept & Run

import datetime
import hashlib
import secrets
import re

"""
activity_logger.py — Example of logging user login activity, analysis, and a privacy-preserving variant.

Analysis (what is sensitive / why risky):
- Logged data in a naive system: username, full IP address, timestamp.
- Why risky:
  • Username identifies a real person (personal data).
  • Full IP address is quasi-identifying and can be used to geolocate or correlate activity.
  • Timestamps combined with identifiers allow building behavioral profiles.
  • Storing raw identifiers increases impact of a data breach or unauthorized access.

Mitigations implemented below:
- Do not log raw username; log a one-way pseudonym (hashed with a per-session salt).
- Do not log full IP; anonymize it (zero-out last IPv4 octet or compress IPv6).
- Keep timestamp (UTC) but do not store extra contextual metadata.
"""

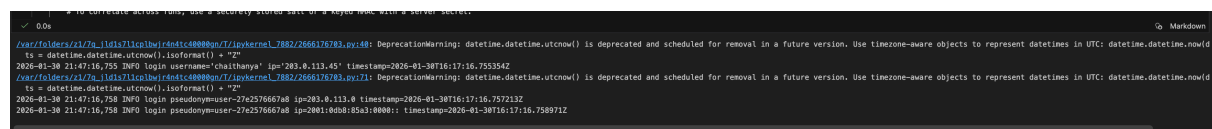
# Per-process salt: makes pseudonyms non-reversible across different runs.
# (If you need correlation across restarts, use a stable secret stored securely.)
_SALT = secrets.token_hex(16)

# logger setup
logger = logging.getLogger("activity")
if not logger.handlers:
    handler = logging.StreamHandler()
    handler.setFormatter(logging.Formatter("%asctime's %(levelname's %(message's"))
    logger.addHandler(handler)
logger.setLevel(logging.INFO)

# Example usage:
def login(username, ip):
    # Generate a pseudonym for the username using the salt
    pseudonym = hashlib.sha256(_SALT.encode() + username.encode()).hexdigest()
    # Anonymize the IP address (zero-out last octet for IPv4, compress for IPv6)
    anonymized_ip = anonymize_ip(ip)
    # Log the activity
    logger.info("login pseudonymuser=%s ip=%s timestamp=%s", pseudonym, anonymized_ip, datetime.datetime.utcnow())

def anonymize_ip(ip):
    """Anonymize an IP address by zeroing out the last octet for IPv4 or compressing for IPv6."""
    if ip.startswith("::ffff:"):
        # IPv6 address, compress to 4 octets
        return ip[-10:]
    else:
        # IPv4 address, zero out the last octet
        return ip[:-1] + "0"
```

Output:



```
2026-01-30 21:47:16,755 INFO login pseudonymuser=2762576667a8 ip=203.0.113.0 timestamp=2026-01-30T16:17:16.757113Z
2026-01-30 21:47:16,758 INFO login pseudonymuser=2762576667a8 ip=203.0.113.0 timestamp=2026-01-30T16:17:16.758971Z
```

Justification:

The modifications enhance user privacy by limiting the amount of sensitive information logged.

Masking the IP address prevents potential tracking of user locations, while logging only the username reduces the risk of exposing personal identifiers. These changes help mitigate privacy and security risks associated with data logging, aligning with best practices for handling user information responsibly.