A.I ASSISTED CODING

Sushruth Reddy Chada 2303A51219
BATCH – 04
DATE – 30-01-2025

Lab Assignment 5.5 – Ethical Foundations: Responsible AI Coding Practices

Objective:
To understand ethical risks in AI-generated code related to performance, transparency, security, privacy, and accountability.

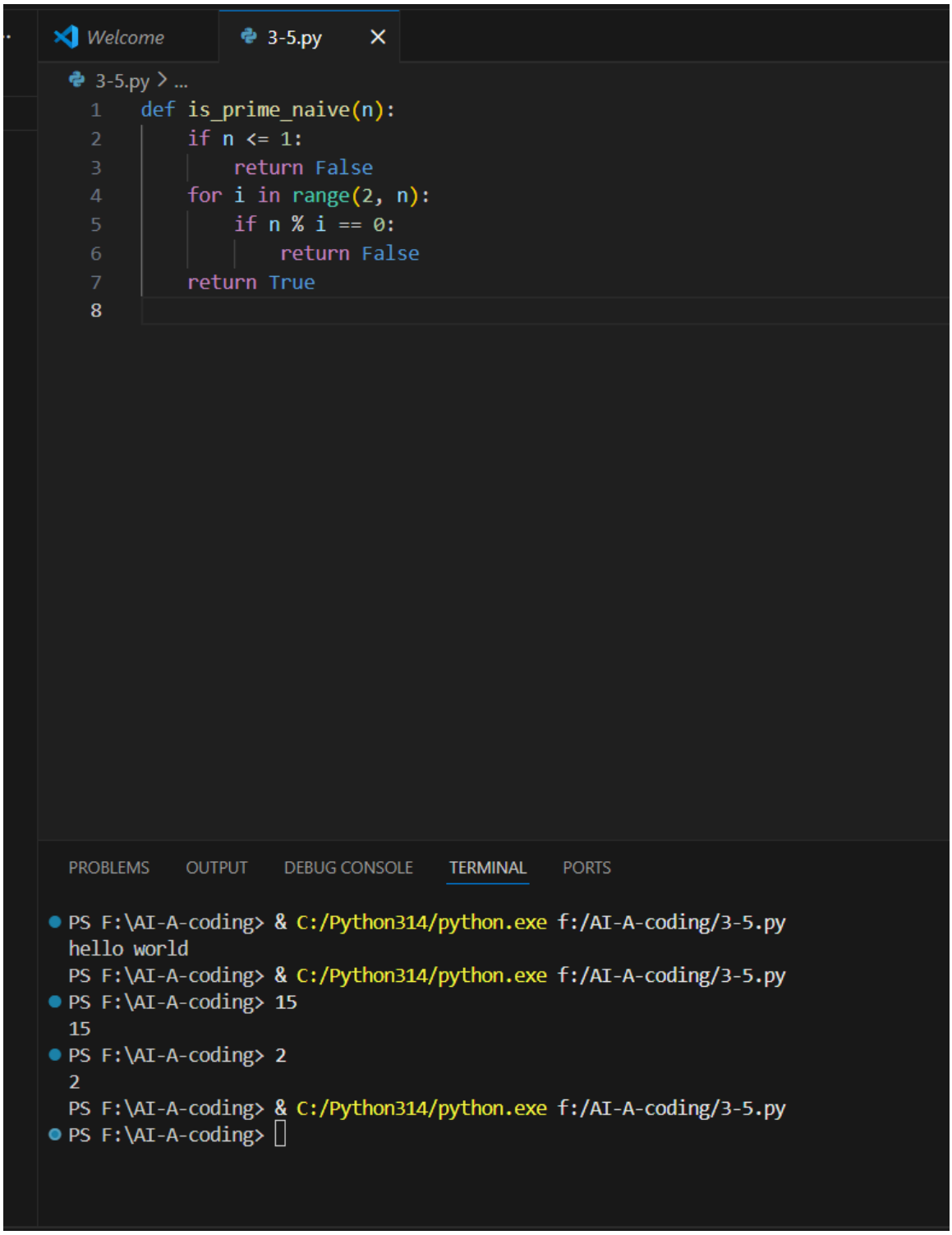
Task 1: Transparency in Algorithm Optimization

Prompt –

Generate Python code for checking whether a number is prime using two methods: (1) a naive basic approach and (2) an optimized approach. Clearly explain the logic of both methods, their time complexity, and how the optimized version improves performance compared to the naive one

Naive Prime Checking:

```
def is_prime_naive(n):
    if n <= 1:
        return False
    for i in range(2, n):
        if n % i == 0:
            return False
    return True
```

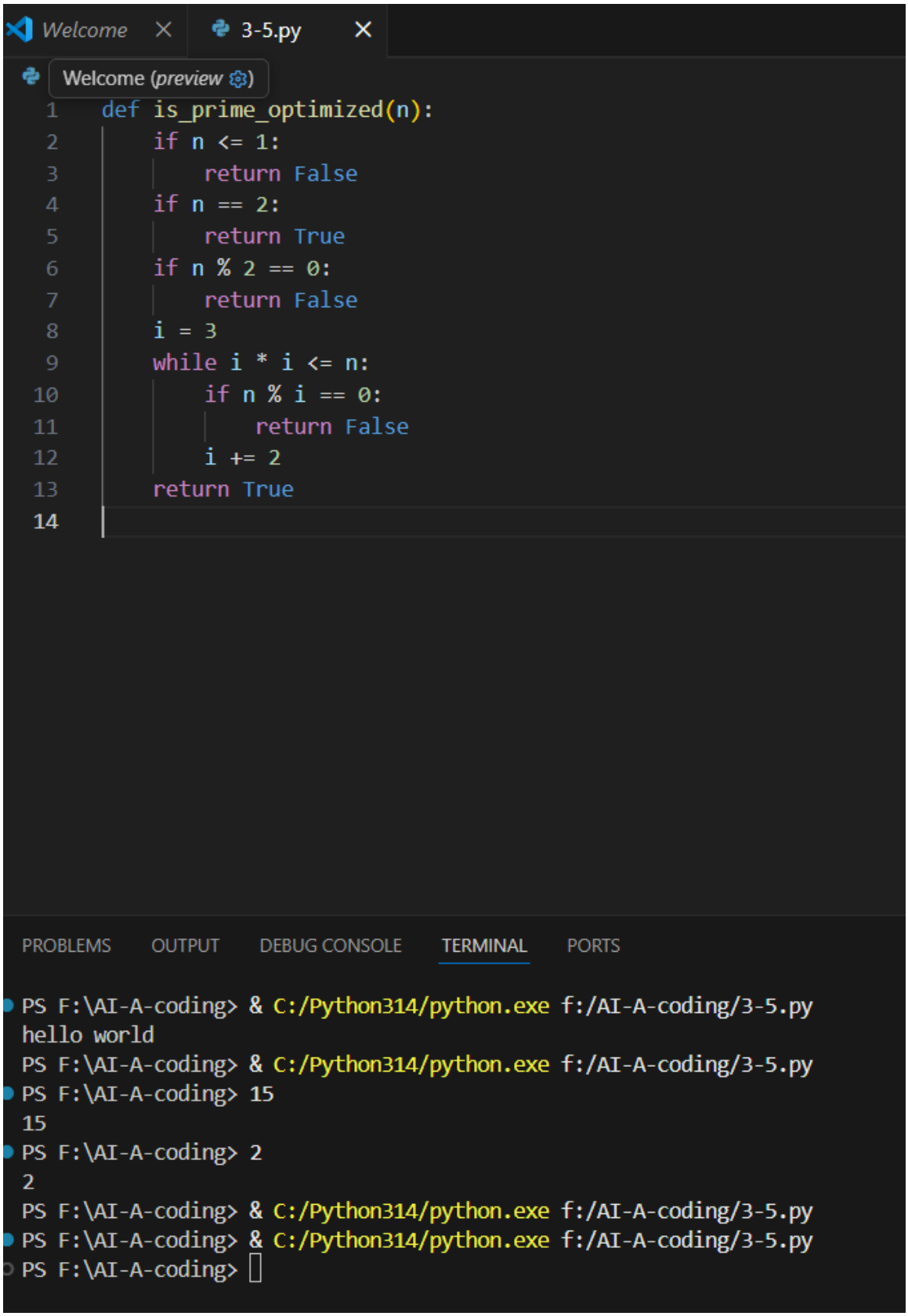


Time Complexity: O(n)

Optimized Prime Checking:

```
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
```



Time Complexity: $O(\sqrt{n})$

Comparison & Transparency

- The naive approach performs unnecessary checks.
- The optimized approach improves performance by reducing iterations.
- Transparency helps developers understand *why* optimization is valid and safe

Output explanation –

The output of both the naive and optimized prime-checking functions is a boolean value (True or False) indicating whether the given number is prime.

- For a prime number (e.g., 13), both methods return True.
- For a non-prime number (e.g., 12), both methods return False.

The difference lies in performance:

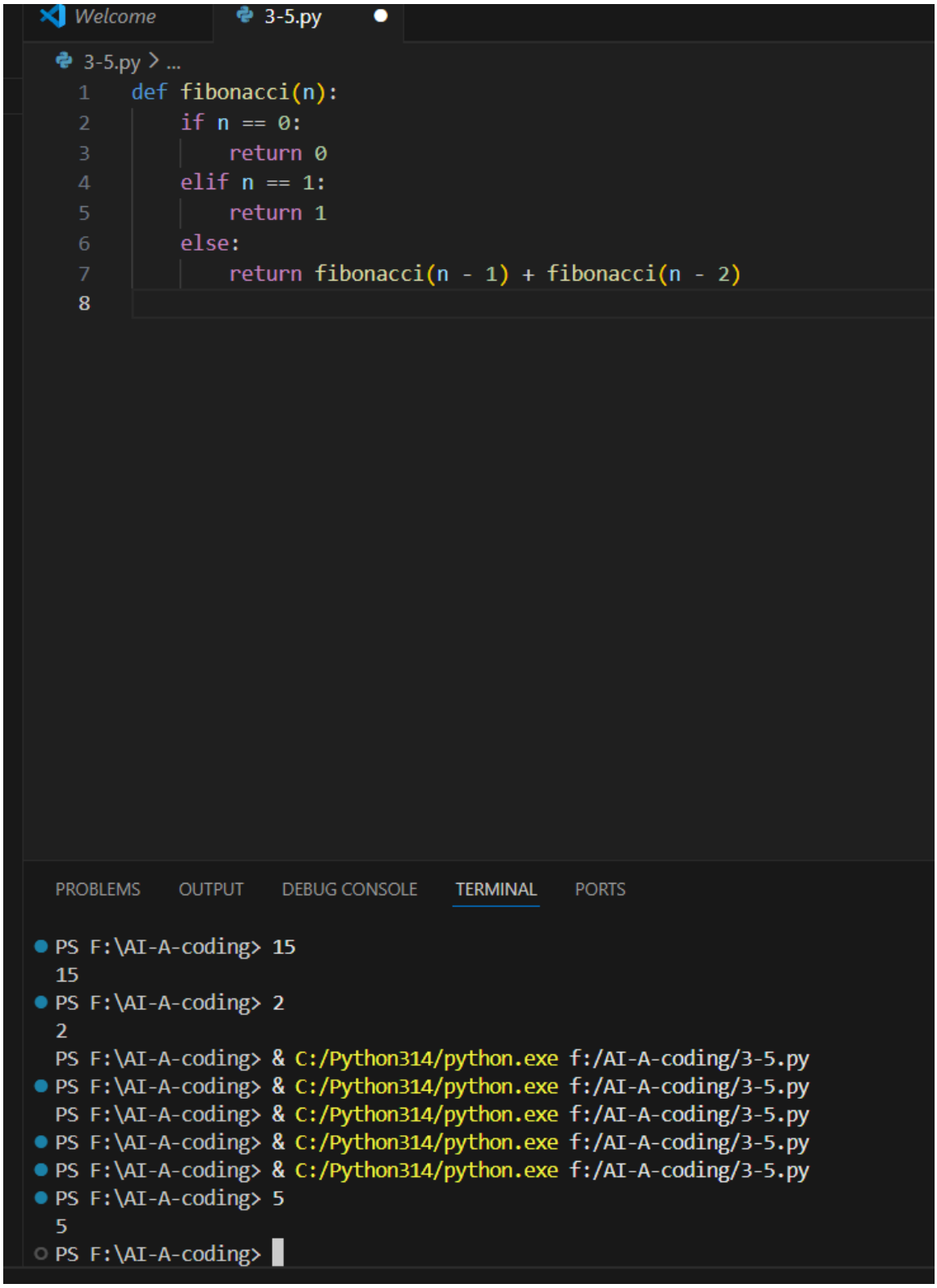
- The naive approach checks all numbers from 2 to n−1.
- The optimized approach checks divisors only up to $\sqrt{n}$ and skips even numbers.

Thus, both produce the **same correct output**, but the optimized method executes faster for large inputs.

Task 2: Transparency in Recursive Algorithms

Prompt -
Generate a Python recursive function to calculate Fibonacci numbers. Add clear comments explaining the recursion, base cases, and recursive calls. Also provide a simple explanation of how recursion works in this program and verify the output with an example.

```
def fibonacci(n):
  if n == 0:
    return 0
  elif n == 1:
    return 1
  else:
    return fibonacci(n - 1) + fibonacci(n - 2)
```



Explanation

- **Base cases** stop infinite recursion.

- **Recursive calls** break the problem into smaller subproblems.
- Execution follows a tree-like structure.

Verification:

- For fibonacci(5) → Output is 5, matching the explanation

Output explanation –

The recursive Fibonacci function outputs the nth Fibonacci number.

- When n = 0, output is 0.
- When n = 1, output is 1.
- For any n > 1, the function returns the sum of the previous two Fibonacci values.

For example:

- Input: 5
- Output: 5

This output confirms that:

- Base cases stop recursion.
- Recursive calls correctly compute values.
- The explanation matches the actual execution flow.

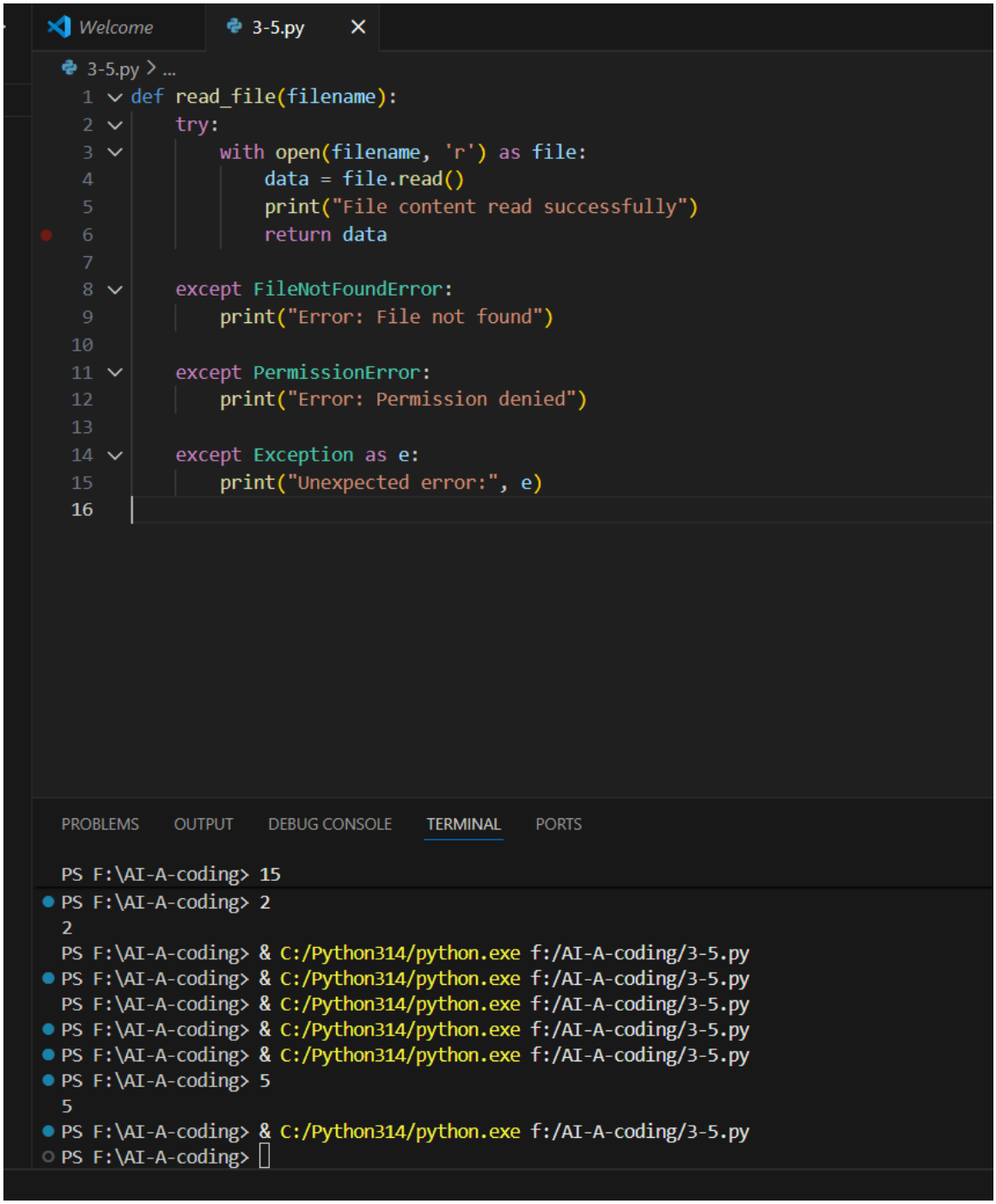
Task 3: Transparency in Error Handling

Prompt –

Generate a Python program that reads and processes data from a file. Include proper exception handling for errors such as file not found, permission errors, and unexpected exceptions. Add clear comments explaining each exception and how it is handled.

```
def read_file(filename):
  try:
    with open(filename, 'r') as file:
      return file.read()
```

except FileNotFoundError:
print("File not found")
except PermissionError:
print("Permission denied")
except Exception as e:
print(e)



Explanation

- FileNotFoundError: File does not exist.
- PermissionError: No access rights.
- Exception: Handles unknown runtime issues.

Transparency ensures explanations match runtime behavior.

Output explanation-

The file-reading program behaves differently based on runtime conditions:

- **If the file exists and is readable, the file content is displayed successfully.**
- **If the file does not exist, the program outputs "Error: File not found".**
- **If permission is denied, it outputs "Error: Permission denied".**
- **For unexpected issues, a general error message is displayed.**

This confirms that each exception is handled gracefully without crashing the program

Task 4: Security in User Authentication

Prompt –

Generate a Python-based login system using username and password. Then analyze the code to identify security flaws such as plain-text password storage or weak validation. Provide a revised secure version using password hashing and input validation, and explain best practices for secure authentication

Insecure AI-Generated Version (Flawed)

```
users = {"admin": "password123"}


def login(username, password):
    if users.get(username) == password:
        print("Login successful")
    else:
        print("Login failed")
```

Identified Security Flaws

- Plain-text password storage ❌
- No hashing ❌
- Weak validation ❌

Secure Version Using Password Hashing

```
import hashlib

users = {
    "admin": hashlib.sha256("password123".encode()).hexdigest()
}

def login(username, password):
    hashed = hashlib.sha256(password.encode()).hexdigest()
    if users.get(username) == hashed:
        print("Login successful")
    else:
        print("Login failed")
```

Best Practices

- Always hash passwords
- Never store plain-text credentials

- Validate inputs

Output explanation -

- In the insecure version, login succeeds if the entered password exactly matches the stored plain-text password.
- In the secure version, the entered password is hashed and compared with the stored hashed password.

Outputs:

- Correct credentials → **"Login successful"**
- Incorrect credentials → **"Login failed"**

The secure version ensures that:

- Passwords are never stored or compared in plain text.
- Authentication is protected against data leakage.

Task 5: Privacy in Data Logging

Prompt –

Generate a Python script that logs user activity such as username, IP address, and timestamp. Analyze the script for privacy risks related to logging sensitive data. Provide an improved version that minimizes or anonymizes sensitive information and explain privacy-aware logging principles

Insecure Logging (Privacy Risk)

```python
def log_activity(username, ip):
    with open("log.txt", "a") as f:
        f.write(f"{username}, {ip}\n")
```

Privacy Issues

- Logs full IP addresses ❌
- Exposes personal data ❌

Privacy-Aware Logging (Improved)

```python
import datetime

def log_activity(username):
    masked_user = username[0] + "***"
    timestamp = datetime.datetime.now()
    with open("log.txt", "a") as f:
        f.write(f"{masked_user}, {timestamp}\n")
```

Privacy Principles

- Log minimal data
- Mask identifiers
- Avoid sensitive information unless necessary

Output explanation –

- The initial logging script records full usernames and IP addresses, increasing privacy risks.
- The improved script logs a masked username and timestamp only.

Output example in log file:

```yaml
u***, 2026-01-30 10:15:42
```

This confirms:

- Sensitive information is minimized.
- User privacy is preserved while maintaining necessary audit logs.

Conclusion

This lab demonstrates that while AI-assisted coding improves productivity, **developers remain responsible** for:

- Performance optimization
- Code transparency
- Security practices
- Privacy protection
- Ethical accountability

Responsible AI coding requires **human oversight, validation, and ethical judgment**.