

## LAB TEST-04

### **SET-D**

#### **Question 1:** Refactor Legacy File Processing Code

Task: You are given the following outdated script that reads a file and counts words:

```
file = open("data.txt", "r")
content = file.read()
words = content.split(" ")
print("Word count:", len(words))
file.close()
```

Use an AI-assisted coding tool to refactor the code to:

1. Use context managers (with statement).
2. Handle exceptions (like missing file).
3. Optimize the logic for accurate word counting.

Instructions:


- Test with a sample text file.
- Document any code smells the AI helped identify.

#### **Prompt:**

Refactor the following legacy Python script to make it more efficient and modern:

```
file = open("data.txt", "r")
content = file.read()
words = content.split(" ")
print("Word count:", len(words))
file.close()
```

## Code:

```
 # Create a sample file for testing
file_content = "This is a sample file.\nIt has multiple lines\nand some words."
with open("data.txt", "w") as file:
    file.write(file_content)

try:
    # Use a context manager (with statement) for safe file handling
    with open("data.txt", "r") as file:
        content = file.read()
        # Optimize word counting by splitting by any whitespace and filtering empty strings
        words = content.split()
        print("Word count:", len(words))
except FileNotFoundError:
    print("Error: data.txt not found.")
except Exception as e:
    print(f"An unexpected error occurred: {e}")

... Word count: 12
```

## Explanation:

This code snippet first creates a sample file named data.txt with some content. Then, it attempts to open and read this file using a with statement, which ensures the file is automatically closed even if errors occur. It splits the content into words by splitting on any whitespace (spaces, tabs, newlines) and filters out any empty strings that might result from multiple spaces. Finally, it prints the count of these words. The try...except block handles potential errors, specifically catching FileNotFoundError if data.txt doesn't exist, and a general Exception for any other unexpected errors.

## Question 2: Optimize a Data Processing Script

Task: A legacy script processes a list of numbers inefficiently:

```
data = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
even_numbers = []
for i in data:
    if i % 2 == 0:
        even_numbers.append(i)
print("Even numbers:", even_numbers)
```

Use AI-assisted refactoring to:

1. Improve the efficiency using modern Python techniques.
2. Add logging and modular functions.
3. Ensure code adheres to PEP8 standards.

Instructions:

- Run the refactored code and verify the same output.
- Summarize the AI's role in detecting inefficiencies.

### **Prompt:**

Refactor the following legacy Python script to make it more efficient and modern:

```
data = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
even_numbers = []
for i in data:
    if i % 2 == 0:
        even_numbers.append(i)
print("Even numbers:", even_numbers)
```

## Code:

```
import logging

# Configure logging
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s - %(message)s')

def find_even_numbers(number_list):
    """Finds even numbers in a list."""
    # Use a list comprehension to find even numbers, adhering to PEP8 spacing
    return [num for num in number_list if num % 2 == 0]

# Create a sample list of numbers
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# Log the input data, adhering to PEP8 line length if necessary
logging.info(f"Input numbers: {numbers}")

# Call the function to find even numbers
even_numbers = find_even_numbers(numbers)

# Log the resulting even numbers
logging.info(f"Even numbers found: {even_numbers}")

# Print the list of even numbers (optional, for immediate output)
print(even_numbers)
```

[2, 4, 6, 8, 10]

## Explanation:

Certainly! The code in cell a2c03ca2 is the refactored script for finding even numbers. Here's a breakdown:

1. **import logging:** This line imports the logging module, which is used for tracking events that happen while the program is running.
2. **logging.basicConfig(...):** This configures the basic settings for logging. It sets the logging level to INFO, meaning it will record informational messages and above, and defines the format of the log messages to include the timestamp, level, and message.
3. **def find\_even\_numbers(number\_list):**: This defines a function called `find_even_numbers` that takes one argument, `number_list`.
4. **"""Finds even numbers in a list."""**: This is a docstring, which explains what the function does.
5. **return [num for num in number\_list if num % 2 == 0]:** This is a list comprehension, a concise way to create lists. It iterates through the `number_list`, and for each `num`, it checks if the remainder when divided by 2 is 0 (`num % 2 == 0`). If it is (meaning the number is even), it includes that number in the new list that is returned by the function.
6. **numbers = [...]:** This creates a sample list of numbers.

7. **logging.info(f"Input numbers: {numbers}")**: This line logs the input list of numbers at the INFO level.
8. **even\_numbers = find\_even\_numbers(numbers)**: This calls the `find_even_numbers` function with the `numbers` list and stores the returned list of even numbers in the `even_numbers` variable.
9. **logging.info(f"Even numbers found: {even\_numbers}")**: This line logs the resulting list of even numbers at the INFO level.
10. **print(even\_numbers)**: This line prints the list of even numbers to the console.

In essence, the code defines a function to efficiently find even numbers using a list comprehension and uses logging to track the input and output of the process, all while adhering to PEP8 style guidelines.