AI ASSIGNMENT - 13

NAME: K.JASHUVA

HALLTICKET NO: 2503A51L16

BATCHNO: 19

TASK 1:

Task Description : Remove Repetition

Task: Provide AI with the following redundant code and ask it to refactor

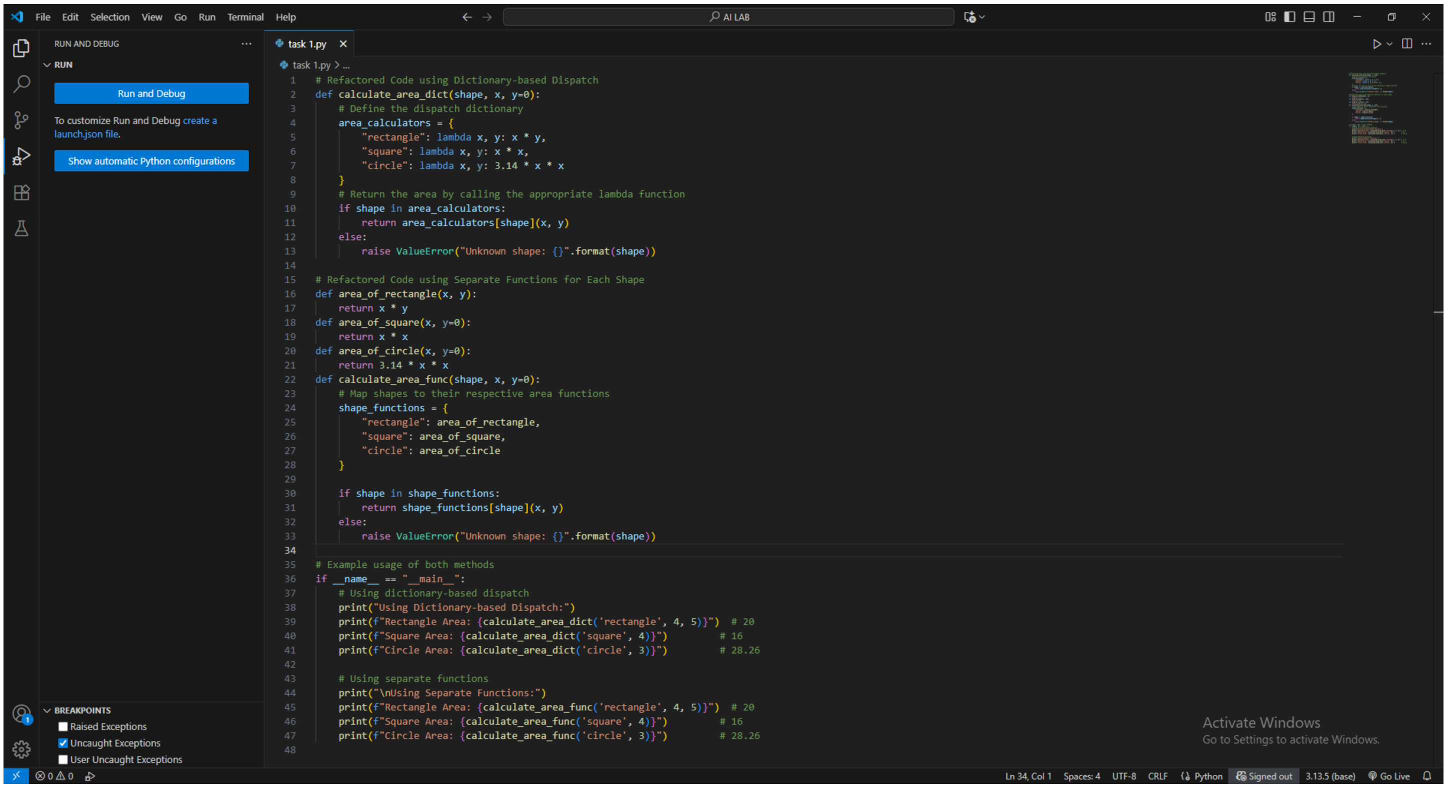
Python Code:

```
def calculate_area(shape, x, y=0):
if shape == "rectangle":
return x * y
elif shape == "square":
return x * x
elif shape == "circle":
return 3.14 * x * x
```

PROMPT:

Refactor the following redundant Python function that calculates shape areas using if-elif; make it cleaner using a dictionary-based dispatch or modular functions.

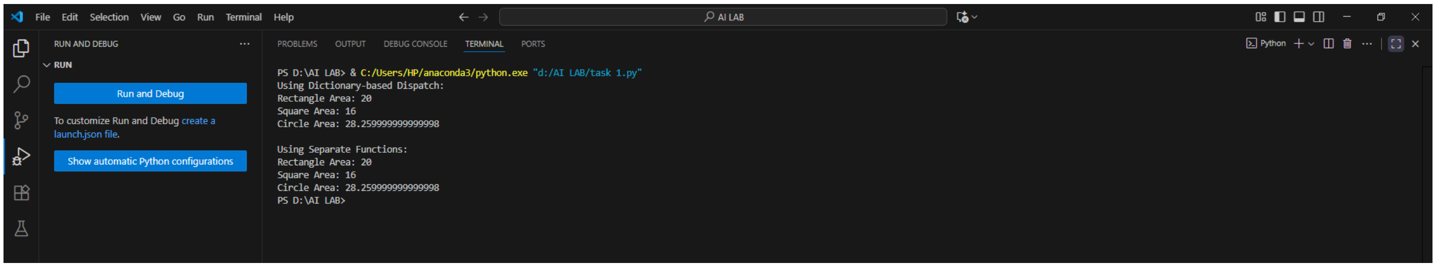
CODE:



```python
# Refactored Code using Dictionary-based Dispatch
def calculate_area_dict(shape, x, y=0):
    # Define the dispatch dictionary
    area_calculators = {
        "rectangle": lambda x, y: x * y,
        "square": lambda x, y: x * x,
        "circle": lambda x, y: 3.14 * x * x
    }
    # Return the area by calling the appropriate lambda function
    if shape in area_calculators:
        return area_calculators[shape](x, y)
    else:
        raise ValueError("Unknown shape: {}".format(shape))

# Refactored Code using Separate Functions for Each Shape
def area_of_rectangle(x, y):
    return x * y
def area_of_square(x, y=0):
    return x * x
def area_of_circle(x, y=0):
    return 3.14 * x * x
def calculate_area_func(shape, x, y=0):
    # Map shapes to their respective area functions
    shape_functions = {
        "rectangle": area_of_rectangle,
        "square": area_of_square,
        "circle": area_of_circle
    }

    if shape in shape_functions:
        return shape_functions[shape](x, y)
    else:
        raise ValueError("Unknown shape: {}".format(shape))

# Example usage of both methods
if __name__ == "__main__":
    # Using dictionary-based dispatch
    print("Using Dictionary-based Dispatch:")
    print(f"Rectangle Area: {calculate_area_dict('rectangle', 4, 5)}")  # 20
    print(f"Square Area: {calculate_area_dict('square', 4)}")          # 16
    print(f"Circle Area: {calculate_area_dict('circle', 3)}")          # 28.26

    # Using separate functions
    print("\nUsing Separate Functions:")
    print(f"Rectangle Area: {calculate_area_func('rectangle', 4, 5)}")  # 20
    print(f"Square Area: {calculate_area_func('square', 4)}")          # 16
    print(f"Circle Area: {calculate_area_func('circle', 3)}")          # 28.26
```

File Edit Selection View Go Run Terminal Help
AI LAB
RUN AND DEBUG
RUN
Run and Debug
To customize Run and Debug create a launch.json file.
Show automatic Python configurations
task 1.py
BREAKPOINTS
□ Raised Exceptions
☑ Uncaught Exceptions
□ User Uncaught Exceptions
Activate Windows
Go to Settings to activate Windows.
Ln 34, Col 1 Spaces: 4 UTF-8 CRLF Python Signed out 3.13.5 (base) Go Live

OUTPUT:



```
PS D:\AI LAB> & C:/Users/HP/anaconda3/python.exe "d:/AI LAB/task 1.py"
Using Dictionary-based Dispatch:
Rectangle Area: 20
Square Area: 16
Circle Area: 28.259999999999998

Using Separate Functions:
Rectangle Area: 20
Square Area: 16
Circle Area: 28.259999999999998
PS D:\AI LAB>
```

File Edit Selection View Go Run Terminal Help
AI LAB
RUN AND DEBUG
RUN
Run and Debug
To customize Run and Debug create a launch.json file.
Show automatic Python configurations
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
Python

OBSERVATIONS:

- The function uses multiple if-elif conditions, leading to repetitive and verbose code.
- Hardcoded logic for each shape reduces scalability — adding new shapes requires modifying the main function.
- The code lacks modularity and separation of concerns; shape-specific calculations are mixed in one function.
- No use of Python features like dictionaries or separate functions to simplify dispatching logic.
- Potential for errors or duplication if area formulas need updates or expansion.

TASK 2

Task Description – Error Handling in Legacy Code

Task: Legacy function without proper error handling

Python Code:

def read_file(filename):

f = open(filename, "r")

data = f.read()

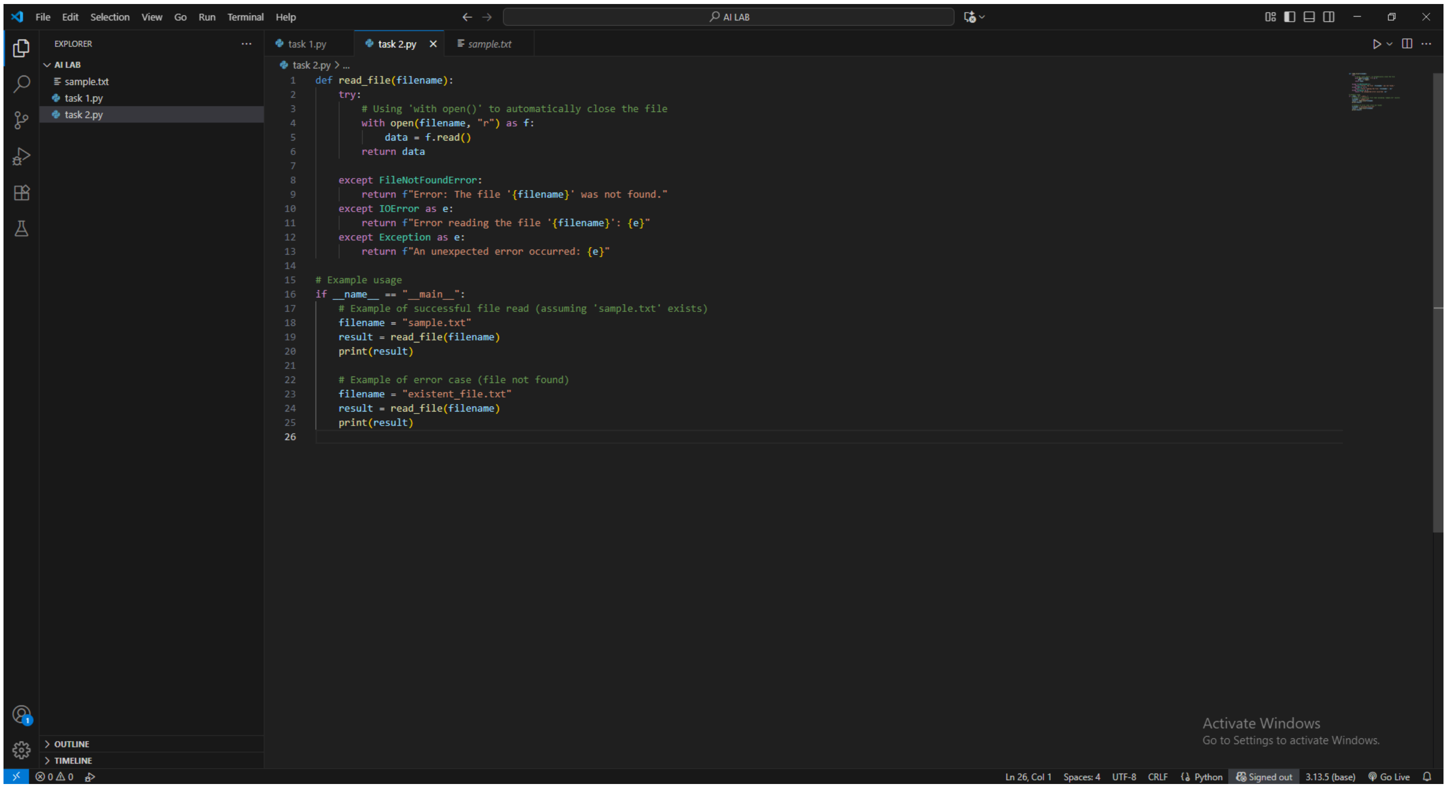
f.close()

return data

PROMPT:

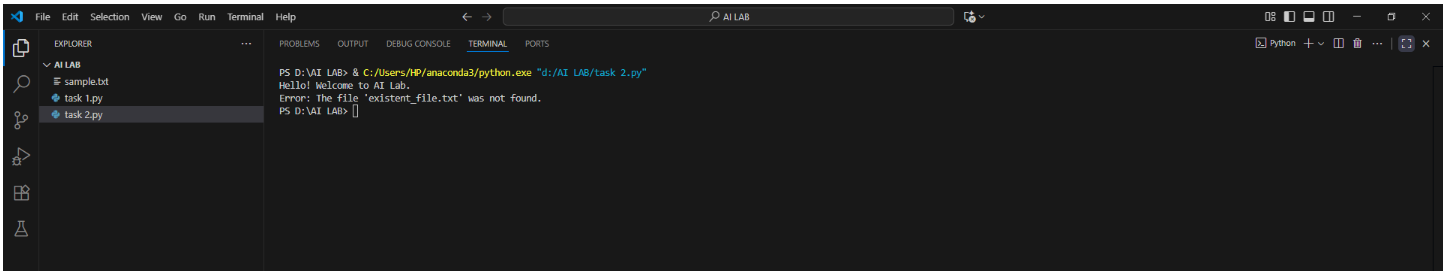
Refactor the following legacy function to use with open() and add proper try-except error handling for file-related exceptions.

CODE:



```python
def read_file(filename):
    try:
        # Using 'with open()' to automatically close the file
        with open(filename, "r") as f:
            data = f.read()
        return data

    except FileNotFoundError:
        return f"Error: The file '{filename}' was not found."
    except IOError as e:
        return f"Error reading the file '{filename}': {e}"
    except Exception as e:
        return f"An unexpected error occurred: {e}"

# Example usage
if __name__ == "__main__":
    # Example of successful file read (assuming 'sample.txt' exists)
    filename = "sample.txt"
    result = read_file(filename)
    print(result)

    # Example of error case (file not found)
    filename = "existent_file.txt"
    result = read_file(filename)
    print(result)
```

OUTPUT:



```
PS D:\AI LAB> & C:/Users/HP/anaconda3/python.exe "d:/AI LAB/task 2.py"
Hello! Welcome to AI Lab.
Error: The file 'existent_file.txt' was not found.
PS D:\AI LAB> 
```

OBSERVATIONS:

- The function manually opens and closes the file, which risks leaving the file open if an exception occurs.
- No error handling; potential exceptions (e.g., file not found, permission denied) will cause the program to crash.
- Lack of resource management best practices; should use with open() to ensure the file is properly closed.
- No feedback or handling for I/O errors, reducing robustness and user-friendliness.
- The function assumes the file exists and is readable, making it fragile in real-world scenarios.

TASK 3

Task Description – Complex Refactoring

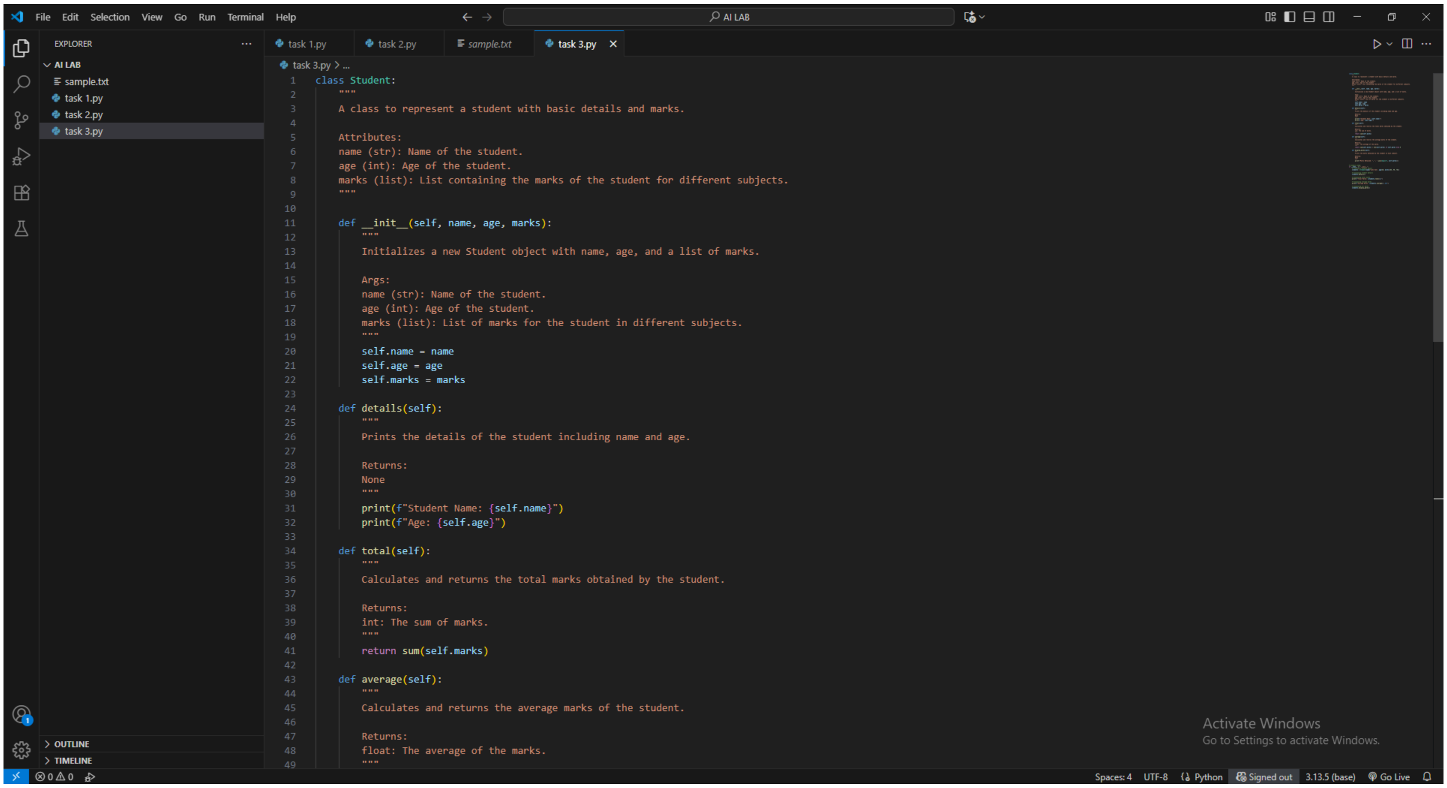
Task: Provide this legacy class to AI for readability and modularity improvements:

Python Code:

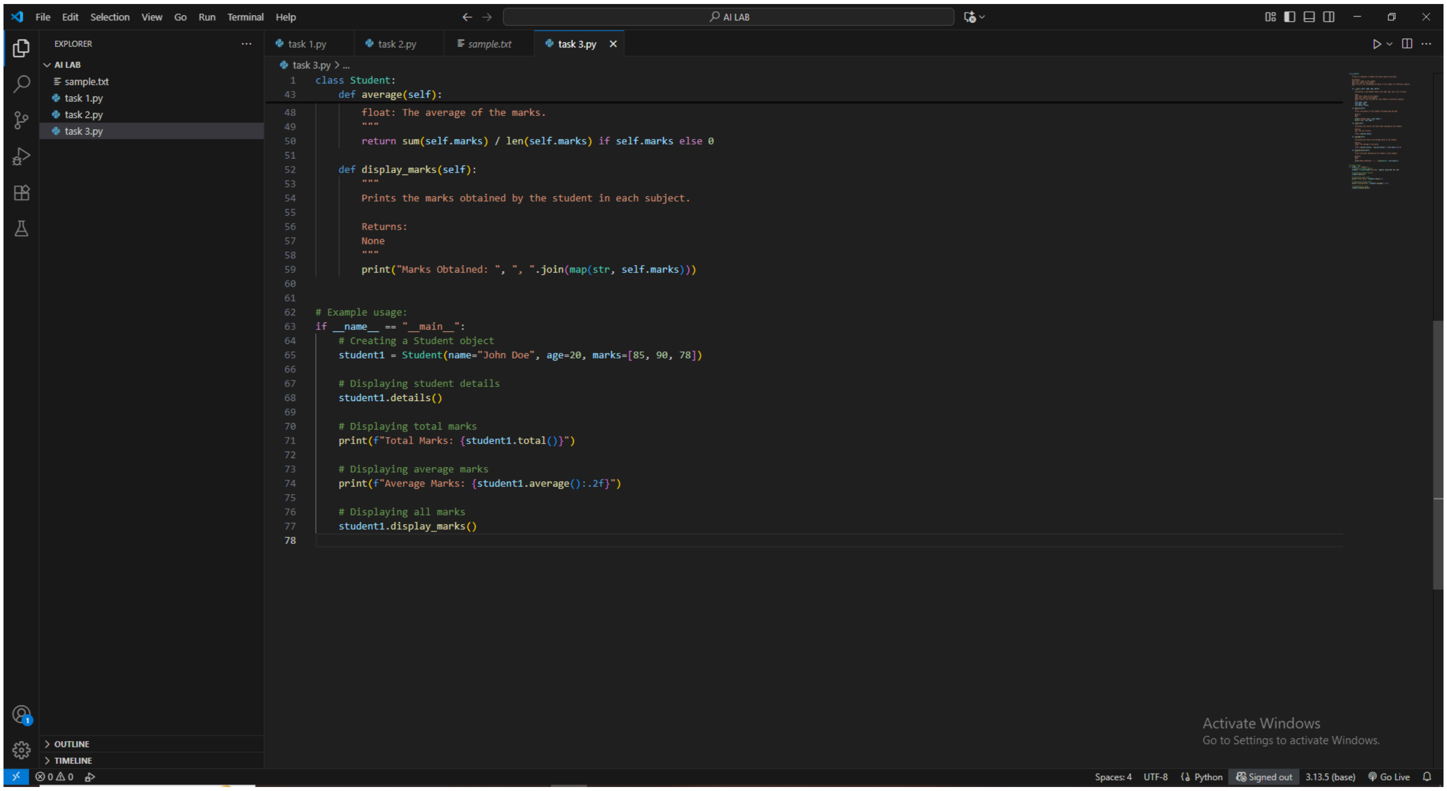
```
class Student:
def __init__(self, n, a, m1, m2, m3):
self.n = n
self.a = a
self.m1 = m1
self.m2 = m2
self.m3 = m3
def details(self):
print("Name:", self.n, "Age:", self.a)
def total(self):
return self.m1+self.m2+self.m3
```

PROMPT: Refactor this legacy Student class to improve readability by using clear variable names, adding docstrings, and storing marks in a list for modularity.

CODE:

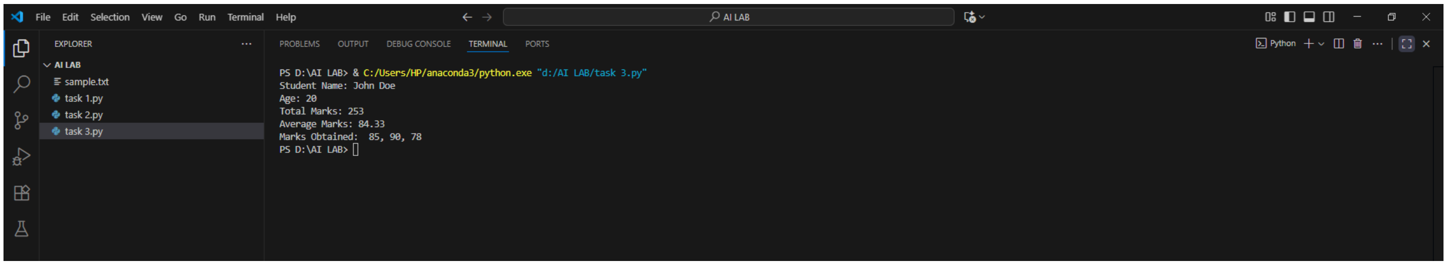


```python
class Student:
    """
    A class to represent a student with basic details and marks.

    Attributes:
    name (str): Name of the student.
    age (int): Age of the student.
    marks (list): List containing the marks of the student for different subjects.
    """

    def __init__(self, name, age, marks):
        """
        Initializes a new Student object with name, age, and a list of marks.

        Args:
        name (str): Name of the student.
        age (int): Age of the student.
        marks (list): List of marks for the student in different subjects.
        """
        self.name = name
        self.age = age
        self.marks = marks

    def details(self):
        """
        Prints the details of the student including name and age.

        Returns:
        None
        """
        print(f"Student Name: {self.name}")
        print(f"Age: {self.age}")

    def total(self):
        """
        Calculates and returns the total marks obtained by the student.

        Returns:
        int: The sum of marks.
        """
        return sum(self.marks)

    def average(self):
        """
        Calculates and returns the average marks of the student.

        Returns:
        float: The average of the marks.
        """
```



```python
class Student:
    def average(self):
        float: The average of the marks.
        """
        return sum(self.marks) / len(self.marks) if self.marks else 0

    def display_marks(self):
        """
        Prints the marks obtained by the student in each subject.

        Returns:
        None
        """
        print("Marks Obtained: ", ", ".join(map(str, self.marks)))


# Example usage:
if __name__ == "__main__":
    # Creating a Student object
    student1 = Student(name="John Doe", age=20, marks=[85, 90, 78])

    # Displaying student details
    student1.details()

    # Displaying total marks
    print(f"Total Marks: {student1.total()}")

    # Displaying average marks
    print(f"Average Marks: {student1.average():.2f}")

    # Displaying all marks
    student1.display_marks()

```

OUTPUT:



```
PS D:\AI LAB> & C:/Users/HP/anaconda3/python.exe "d:/AI LAB/task 3.py"
Student Name: John Doe
Age: 20
Total Marks: 253
Average Marks: 84.33
Marks Obtained:  85, 90, 78
PS D:\AI LAB> 
```

OBSERVATIONS:

- Variable names (n, a, m1, m2, m3) are unclear and non-descriptive, reducing code readability.
- Marks are stored as separate attributes rather than in a collection (e.g., list), limiting scalability and increasing code repetition.
- The class lacks docstrings and inline comments, making it harder to understand the purpose of methods and attributes.
- The details() method uses a basic print statement with multiple arguments, resulting in less readable output formatting.
- No methods to handle or manipulate marks beyond calculating the total, limiting functionality.
- The design is not modular, which would make adding features (like average marks or more subjects) cumbersome.

TASK 4

Task Description – Inefficient Loop Refactoring

Task: Refactor this inefficient loop with AI help

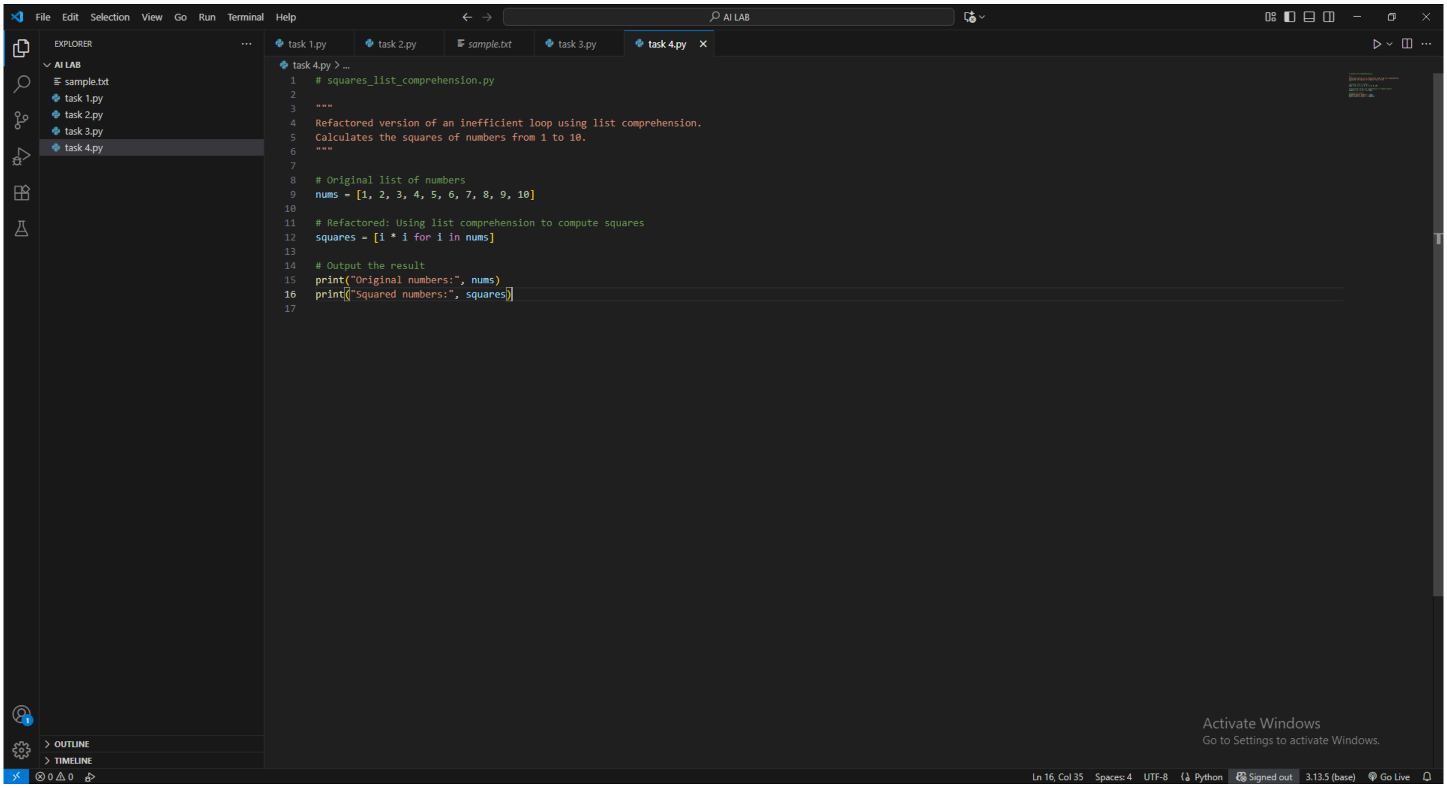
Python Code:

```
nums = [1,2,3,4,5,6,7,8,9,10]

squares = []

for i in nums:

squares.append(i * i)
```

PROMPT:

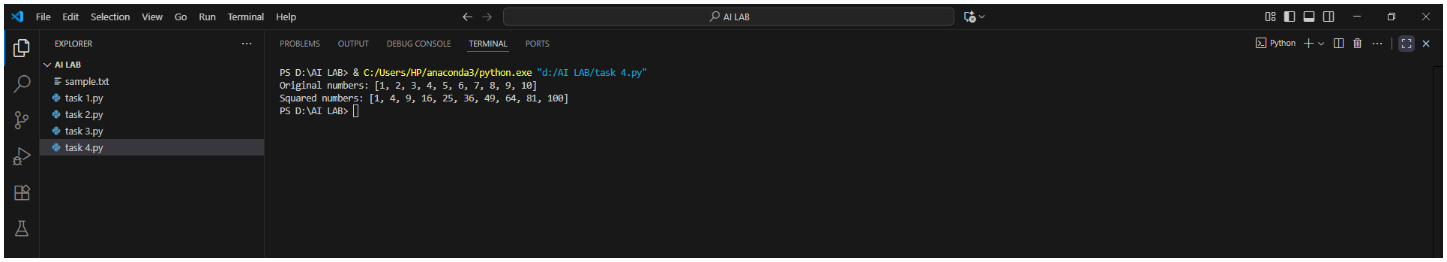
Refactor the following inefficient loop that appends squares of numbers to a list into a more concise and efficient version using list comprehension.

CODE:



```python
# squares_list_comprehension.py

"""
Refactored version of an inefficient loop using list comprehension.
Calculates the squares of numbers from 1 to 10.
"""

# Original list of numbers
nums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

# Refactored: Using list comprehension to compute squares
squares = [i * i for i in nums]

# Output the result
print("Original numbers:", nums)
print("Squared numbers:", squares)
```

OUTPUT:



```
PS D:\AI LAB> & C:/Users/HP/anaconda3/python.exe "d:/AI LAB/task 4.py"
Original numbers: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
Squared numbers: [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
PS D:\AI LAB> 
```

OBSERVATIONS:

- The current code uses an explicit for loop with .append(), which is verbose and less Pythonic.
- List comprehension offers a cleaner, faster, and more readable way to generate the list of squares.
- Using list comprehension improves maintainability and aligns with Python’s idiomatic style.