AI ASSISTED CODING

LAB-13.2

NAME: B. NISHANT

ENROLLNO:2403A52089

BATCH:04

TASK-01:

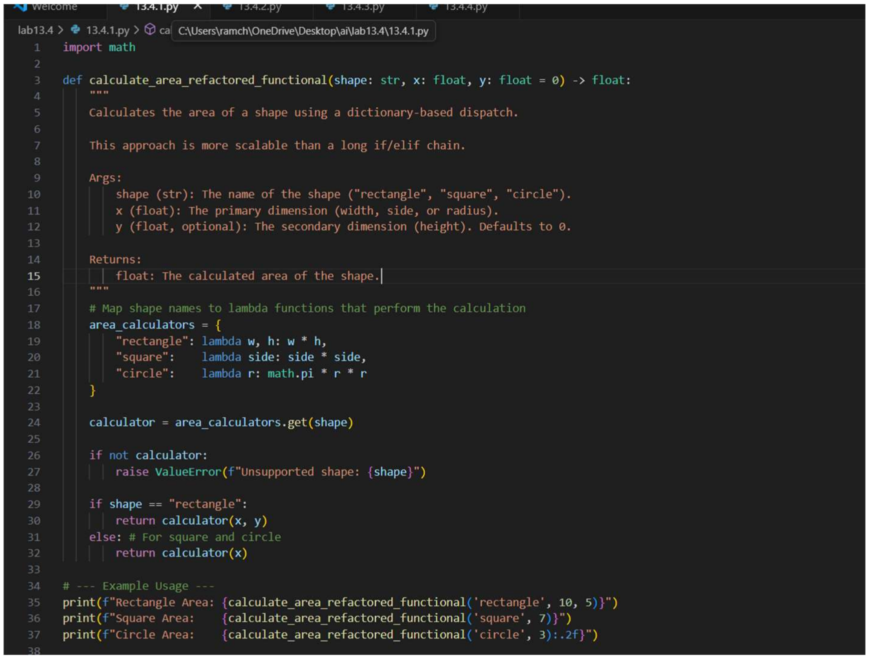
Remove Repetition.

PROMPT:

Refactor the following redundant code

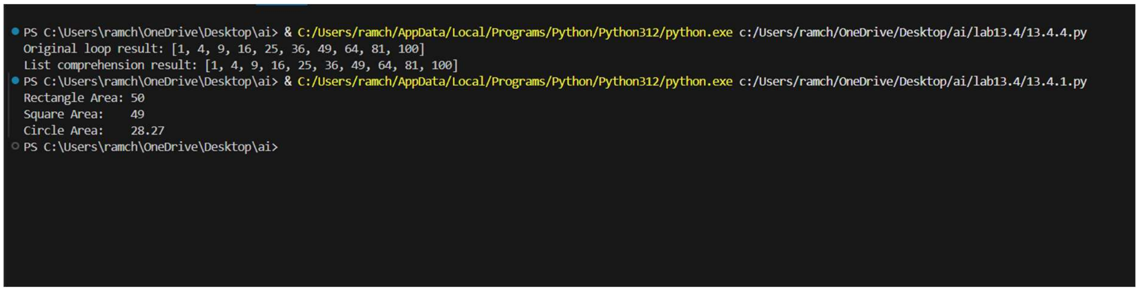
def calculate_area(shape, x, y=0):
if shape == "rectangle":
return x * y
elif shape == "square":
return x * x
elif shape == "circle":
return 3.14 * x * x

CODE:



```python
import math

def calculate_area_refactored_functional(shape: str, x: float, y: float = 0) -> float:
    """
    Calculates the area of a shape using a dictionary-based dispatch.

    This approach is more scalable than a long if/elif chain.

    Args:
        shape (str): The name of the shape ("rectangle", "square", "circle").
        x (float): The primary dimension (width, side, or radius).
        y (float, optional): The secondary dimension (height). Defaults to 0.

    Returns:
        float: The calculated area of the shape.
    """
    # Map shape names to lambda functions that perform the calculation
    area_calculators = {
        "rectangle": lambda w, h: w * h,
        "square":    lambda side: side * side,
        "circle":    lambda r: math.pi * r * r
    }

    calculator = area_calculators.get(shape)

    if not calculator:
        raise ValueError(f"Unsupported shape: {shape}")

    if shape == "rectangle":
        return calculator(x, y)
    else: # For square and circle
        return calculator(x)

# --- Example Usage ---
print(f"Rectangle Area: {calculate_area_refactored_functional('rectangle', 10, 5)}")
print(f"Square Area:    {calculate_area_refactored_functional('square', 7)}")
print(f"Circle Area:    {calculate_area_refactored_functional('circle', 3):.2f}")
```

OUTPUT:



```
● PS C:\Users\ramch\OneDrive\Desktop\ai> & C:/Users/ramch/AppData/Local/Programs/Python/Python312/python.exe c:/Users/ramch/OneDrive/Desktop/ai/lab13.4/13.4.4.py
  Original loop result: [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
  List comprehension result: [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
● PS C:\Users\ramch\OneDrive\Desktop\ai> & C:/Users/ramch/AppData/Local/Programs/Python/Python312/python.exe c:/Users/ramch/OneDrive/Desktop/ai/lab13.4/13.4.1.py
  Rectangle Area: 50
  Square Area:    49
  Circle Area:    28.27
○ PS C:\Users\ramch\OneDrive\Desktop\ai>
```

OBSERVATION:

The function calculate_area computes the area of a rectangle, square, or circle based on the given dimensions. It converts the shape name to lowercase for consistency, uses x as the main dimension and y as an optional width for rectangles, and calculates the area accordingly. For rectangles, both x and y are required, squares use x**2, and circles use math.pi * x**2. It raises an error if the shape is unknown or if required dimensions are missing. The docstring explains its usage and parameters.

TASK-02:

Error Handling in Legacy Code.

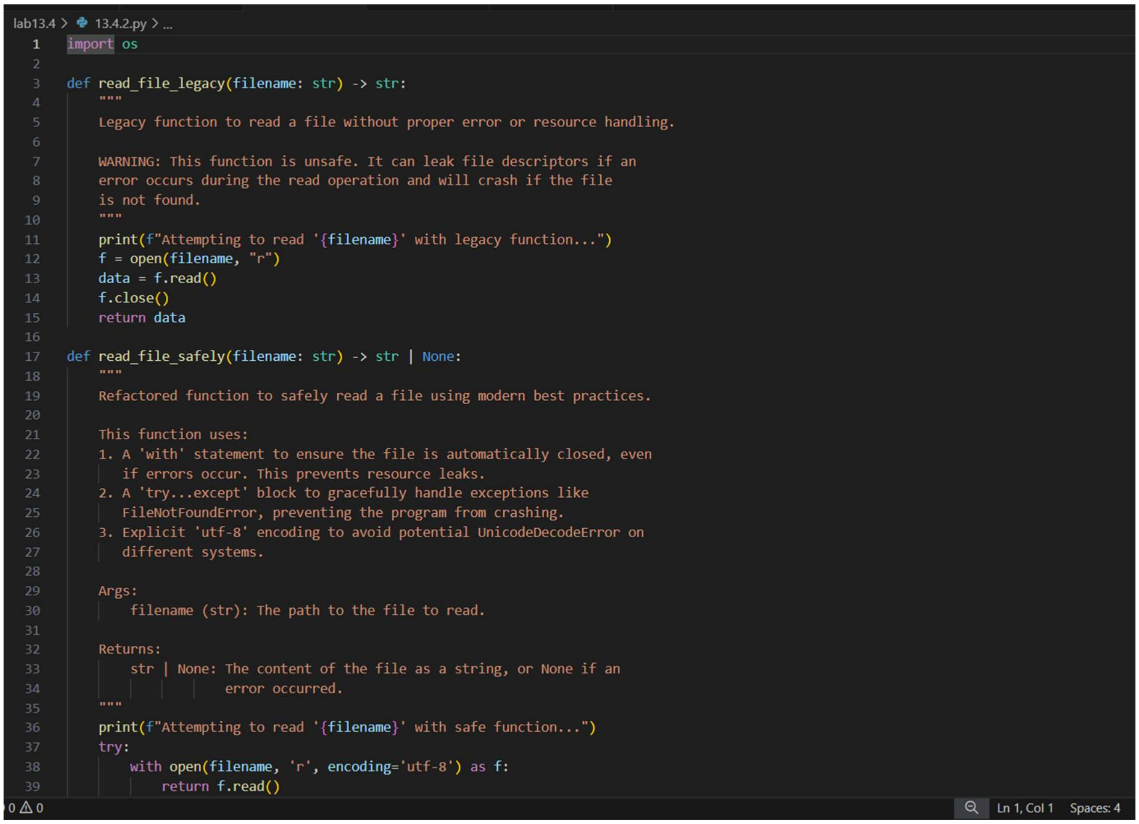
PROMPT:

The following python code that reads the file but it doesn’t handle the errors . rewrite the code by correcting all the errors.

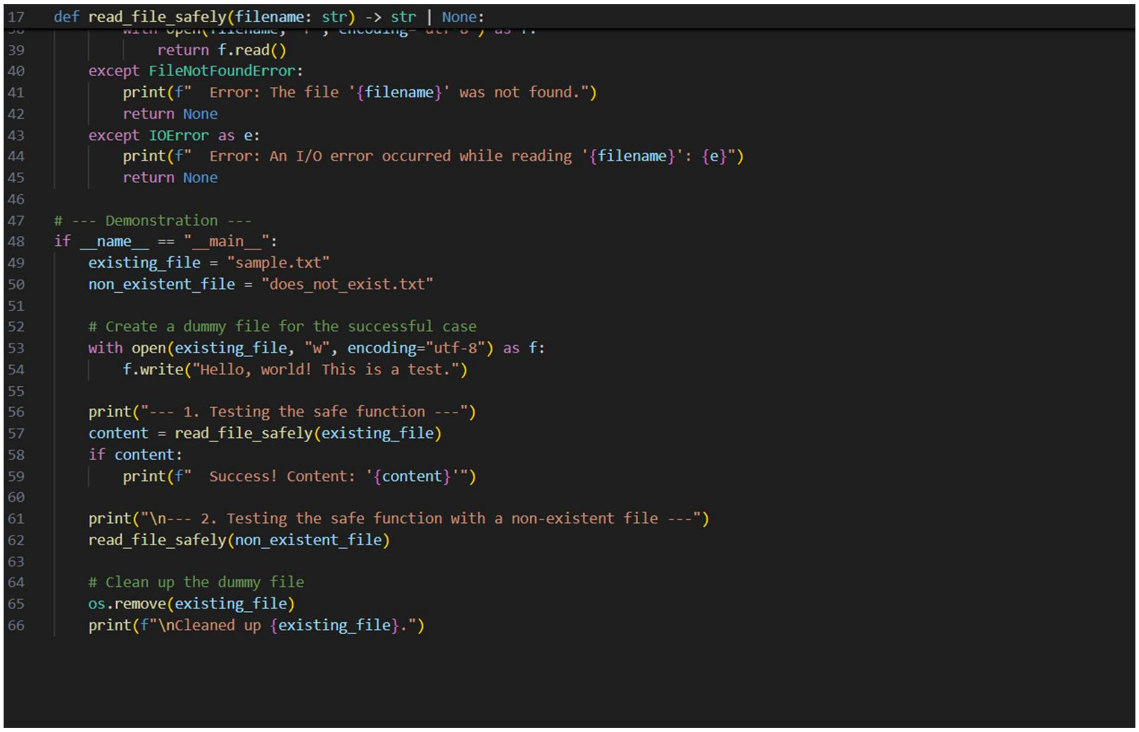
def read_file(filename):
f = open(filename, "r")
data = f.read()

f.close()
return data

CODE:

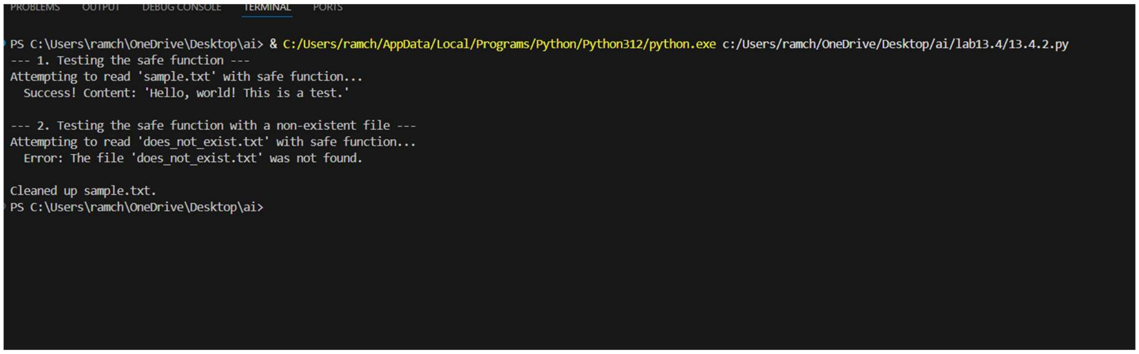


```python
import os

def read_file_legacy(filename: str) -> str:
    """
    Legacy function to read a file without proper error or resource handling.

    WARNING: This function is unsafe. It can leak file descriptors if an
    error occurs during the read operation and will crash if the file
    is not found.
    """
    print(f"Attempting to read '{filename}' with legacy function...")
    f = open(filename, "r")
    data = f.read()
    f.close()
    return data

def read_file_safely(filename: str) -> str | None:
    """
    Refactored function to safely read a file using modern best practices.

    This function uses:
    1. A 'with' statement to ensure the file is automatically closed, even
       if errors occur. This prevents resource leaks.
    2. A 'try...except' block to gracefully handle exceptions like
       FileNotFoundError, preventing the program from crashing.
    3. Explicit 'utf-8' encoding to avoid potential UnicodeDecodeError on
       different systems.

    Args:
        filename (str): The path to the file to read.

    Returns:
        str | None: The content of the file as a string, or None if an
                    error occurred.
    """
    print(f"Attempting to read '{filename}' with safe function...")
    try:
        with open(filename, 'r', encoding='utf-8') as f:
            return f.read()
```



```python
def read_file_safely(filename: str) -> str | None:
        with open(filename, "r", encoding="utf-8") as f:
            return f.read()
    except FileNotFoundError:
        print(f"  Error: The file '{filename}' was not found.")
        return None
    except IOError as e:
        print(f"  Error: An I/O error occurred while reading '{filename}': {e}")
        return None

# --- Demonstration ---
if __name__ == "__main__":
    existing_file = "sample.txt"
    non_existent_file = "does_not_exist.txt"

    # Create a dummy file for the successful case
    with open(existing_file, "w", encoding="utf-8") as f:
        f.write("Hello, world! This is a test.")

    print("--- 1. Testing the safe function ---")
    content = read_file_safely(existing_file)
    if content:
        print(f"  Success! Content: '{content}'")

    print("\n--- 2. Testing the safe function with a non-existent file ---")
    read_file_safely(non_existent_file)

    # Clean up the dummy file
    os.remove(existing_file)
    print(f"\nCleaned up {existing_file}.")
```

OUTPUT:



```
PS C:\Users\ramch\OneDrive\Desktop\ai> & C:/Users/ramch/AppData/Local/Programs/Python/Python312/python.exe c:/Users/ramch/OneDrive/Desktop/ai/lab13.4/13.4.2.py
--- 1. Testing the safe function ---
Attempting to read 'sample.txt' with safe function...
  Success! Content: 'Hello, world! This is a test.'

--- 2. Testing the safe function with a non-existent file ---
Attempting to read 'does_not_exist.txt' with safe function...
  Error: The file 'does_not_exist.txt' was not found.

Cleaned up sample.txt.
PS C:\Users\ramch\OneDrive\Desktop\ai>
```

OBSERVATION:

The refactored function safely reads a file using with open(), ensuring the file is automatically closed, and uses try-except to handle errors like missing files or read failures. It provides clear error messages instead of crashing, making the code more robust and reliable.

Name: B. Nishant
Batch: 04
Enrol: 2403A52089

TASK-03:

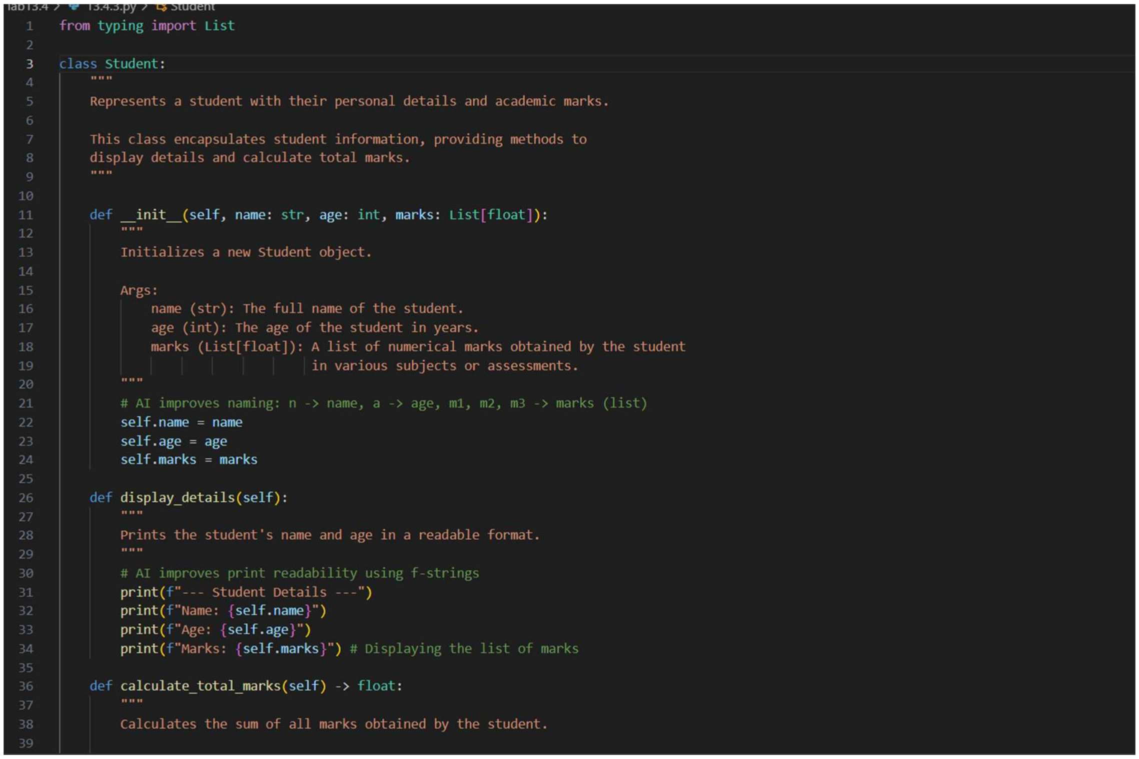
Complex refactoring

PROMPT:

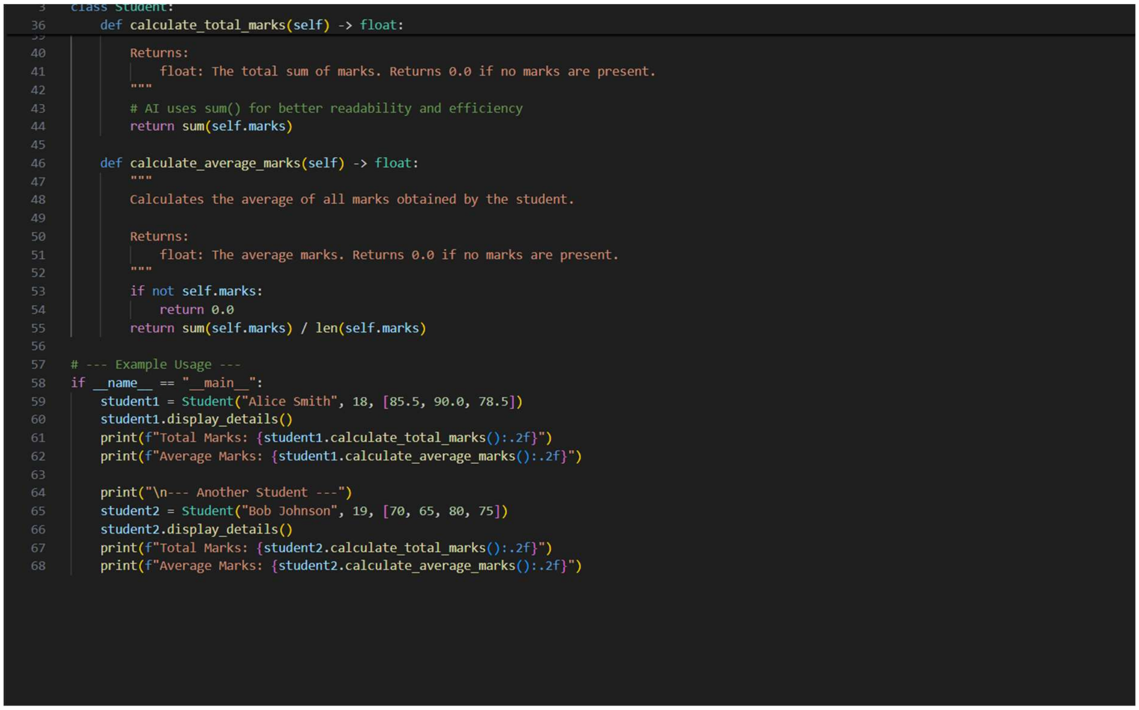
Rewrite the following code by adding the proper variable names and refactor it in a proper way.

```
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
```

CODE:

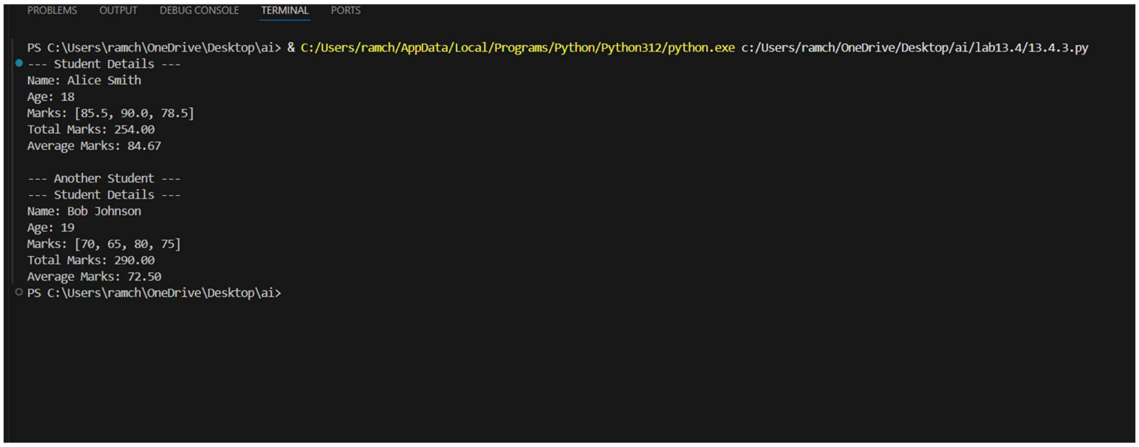


```python
from typing import List

class Student:
    """
    Represents a student with their personal details and academic marks.

    This class encapsulates student information, providing methods to
    display details and calculate total marks.
    """

    def __init__(self, name: str, age: int, marks: List[float]):
        """
        Initializes a new Student object.

        Args:
            name (str): The full name of the student.
            age (int): The age of the student in years.
            marks (List[float]): A list of numerical marks obtained by the student
                                 in various subjects or assessments.
        """
        # AI improves naming: n -> name, a -> age, m1, m2, m3 -> marks (list)
        self.name = name
        self.age = age
        self.marks = marks

    def display_details(self):
        """
        Prints the student's name and age in a readable format.
        """
        # AI improves print readability using f-strings
        print(f"--- Student Details ---")
        print(f"Name: {self.name}")
        print(f"Age: {self.age}")
        print(f"Marks: {self.marks}") # Displaying the list of marks

    def calculate_total_marks(self) -> float:
        """
        Calculates the sum of all marks obtained by the student.

```



```
class Student:
    def calculate_total_marks(self) -> float:

        Returns:
            float: The total sum of marks. Returns 0.0 if no marks are present.
        """
        # AI uses sum() for better readability and efficiency
        return sum(self.marks)

    def calculate_average_marks(self) -> float:
        """
        Calculates the average of all marks obtained by the student.

        Returns:
            float: The average marks. Returns 0.0 if no marks are present.
        """
        if not self.marks:
            return 0.0
        return sum(self.marks) / len(self.marks)

# --- Example Usage ---
if __name__ == "__main__":
    student1 = Student("Alice Smith", 18, [85.5, 90.0, 78.5])
    student1.display_details()
    print(f"Total Marks: {student1.calculate_total_marks():.2f}")
    print(f"Average Marks: {student1.calculate_average_marks():.2f}")

    print("\n--- Another Student ---")
    student2 = Student("Bob Johnson", 19, [70, 65, 80, 75])
    student2.display_details()
    print(f"Total Marks: {student2.calculate_total_marks():.2f}")
    print(f"Average Marks: {student2.calculate_average_marks():.2f}")
```

OUTPUT:



```
PS C:\Users\ramch\OneDrive\Desktop\ai> & C:/Users/ramch/AppData/Local/Programs/Python/Python312/python.exe c:/Users/ramch/OneDrive/Desktop/ai/lab13.4/13.4.3.py
--- Student Details ---
Name: Alice Smith
Age: 18
Marks: [85.5, 90.0, 78.5]
Total Marks: 254.00
Average Marks: 84.67

--- Another Student ---
--- Student Details ---
Name: Bob Johnson
Age: 19
Marks: [70, 65, 80, 75]
Total Marks: 290.00
Average Marks: 72.50
PS C:\Users\ramch\OneDrive\Desktop\ai>
```

OBSERVATION:

The refactored Student class improves readability and modularity by using meaningful names, storing marks in a list, and adding docstrings. The details method prints information clearly with formatted strings, and total efficiently sums marks using sum(). The design is now more flexible and easier to extend.

Name: B. Nishant
Batch: 04
Enrol: 2403A52089

TASK-04:

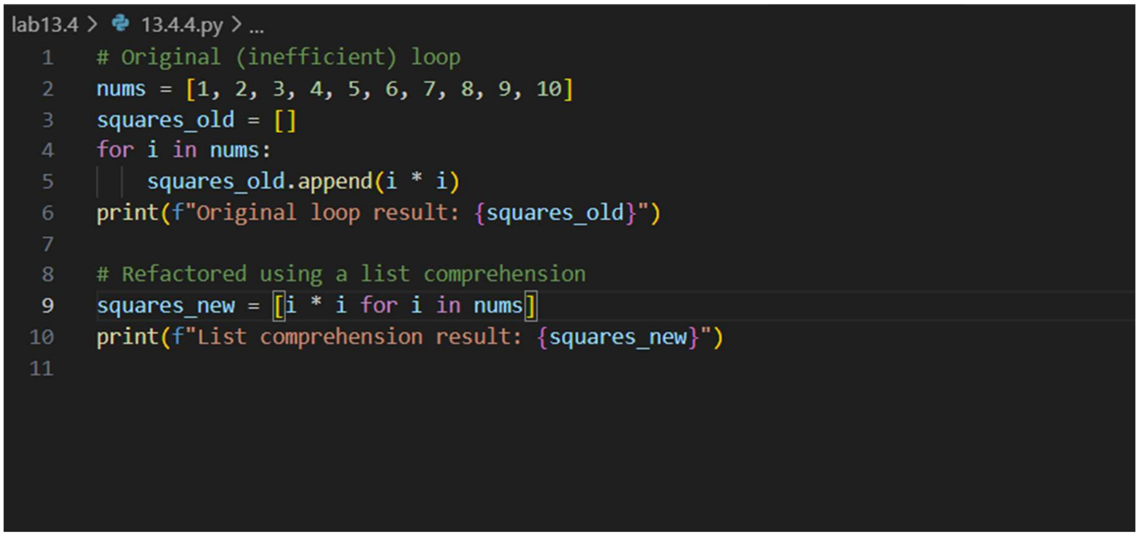
Inefficient Loop Refactoring

PROMPT:

I have a Python loop that computes squares of numbers and appends them to a list, but it seems inefficient. Can you rewrite it in a shorter, more Pythonic way and explain why it’s better?

nums = [1,2,3,4,5,6,7,8,9,10]
squares = []
for i in nums:
squares.append(i * i

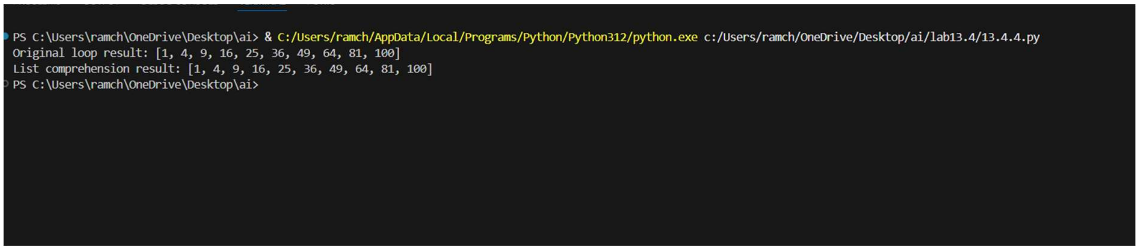
CODE:



lab13.4 > 13.4.4.py > ...

```python
# Original (inefficient) loop
nums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
squares_old = []
for i in nums:
    squares_old.append(i * i)
print(f"Original loop result: {squares_old}")

# Refactored using a list comprehension
squares_new = [i * i for i in nums]
print(f"List comprehension result: {squares_new}")
```

OUTPUT:



```
PS C:\Users\ramch\OneDrive\Desktop\ai> & C:/Users/ramch/AppData/Local/Programs/Python/Python312/python.exe c:/Users/ramch/OneDrive/Desktop/ai/lab13.4/13.4.4.py
Original loop result: [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
List comprehension result: [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
PS C:\Users\ramch\OneDrive\Desktop\ai>
```

OBSERVATION:

The list comprehension makes the code shorter, more readable, and efficient by replacing the explicit loop and append() method with a single expression.

Name: B. Nishant
Batch: 04
Enrol: 2403A52089