NAME : MERUGU RAKSHITH

ENROLLMENT NO : 2403A52007

BATCH NO: 02

SUBJECT: AI ASSISTANT CODING

Assignment-13.2

Task Description #1 – Remove Repetition

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

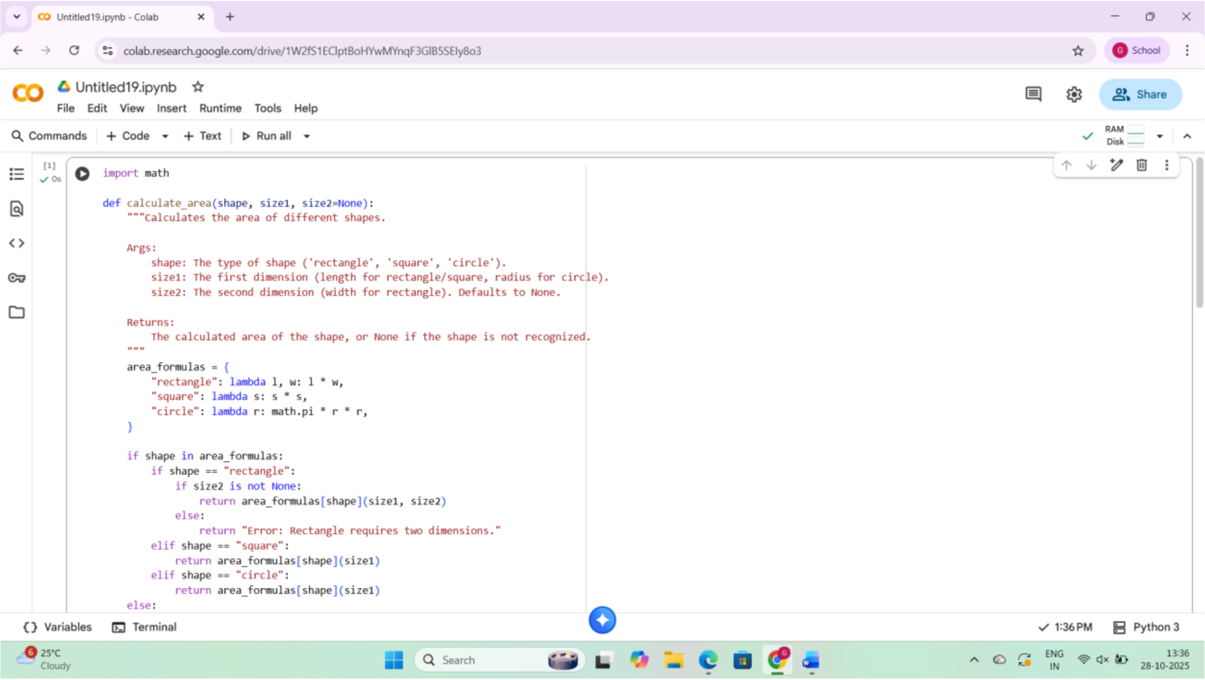
Python Code

```
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
```

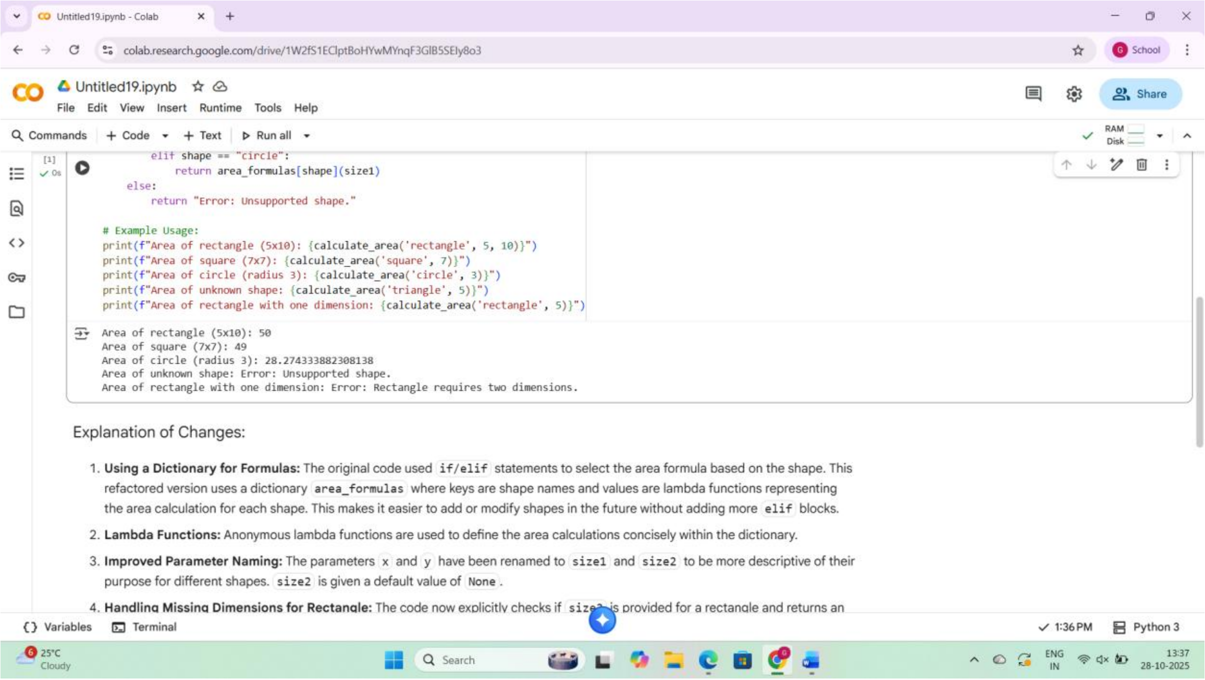
Prompt

Refactor the following Python code to remove redundancy and make it more efficient.
Make the code clean, reusable, and follow Python best practices.
Use a cleaner approach like a dictionary or helper functions to avoid repeating the same formulas.
Also, explain how your version improves readability and maintainability.

Code and output



```
import math

def calculate_area(shape, size1, size2=None):
    """Calculates the area of different shapes.

    Args:
        shape: The type of shape ('rectangle', 'square', 'circle').
        size1: The first dimension (length for rectangle/square, radius for circle).
        size2: The second dimension (width for rectangle). Defaults to None.

    Returns:
        The calculated area of the shape, or None if the shape is not recognized.
    """
    area_formulas = {
        "rectangle": lambda l, w: l * w,
        "square": lambda s: s * s,
        "circle": lambda r: math.pi * r * r,
    }

    if shape in area_formulas:
        if shape == "rectangle":
            if size2 is not None:
                return area_formulas[shape](size1, size2)
            else:
                return "Error: Rectangle requires two dimensions."
        elif shape == "square":
            return area_formulas[shape](size1)
        elif shape == "circle":
            return area_formulas[shape](size1)
    else:
```



```
        elif shape == "circle":
            return area_formulas[shape](size1)
    else:
        return "Error: Unsupported shape."

# Example Usage:
print(f"Area of rectangle (5x10): {calculate_area('rectangle', 5, 10)}")
print(f"Area of square (7x7): {calculate_area('square', 7)}")
print(f"Area of circle (radius 3): {calculate_area('circle', 3)}")
print(f"Area of unknown shape: {calculate_area('triangle', 5)}")
print(f"Area of rectangle with one dimension: {calculate_area('rectangle', 5)}")
```

```
Area of rectangle (5x10): 50
Area of square (7x7): 49
Area of circle (radius 3): 28.274333882308138
Area of unknown shape: Error: Unsupported shape.
Area of rectangle with one dimension: Error: Rectangle requires two dimensions.
```

Explanation of Changes:

1. **Using a Dictionary for Formulas:** The original code used `if/elif` statements to select the area formula based on the shape. This refactored version uses a dictionary `area_formulas` where keys are shape names and values are lambda functions representing the area calculation for each shape. This makes it easier to add or modify shapes in the future without adding more `elif` blocks.
2. **Lambda Functions:** Anonymous lambda functions are used to define the area calculations concisely within the dictionary.
3. **Improved Parameter Naming:** The parameters `x` and `y` have been renamed to `size1` and `size2` to be more descriptive of their purpose for different shapes. `size2` is given a default value of `None`.
4. **Handling Missing Dimensions for Rectangle:** The code now explicitly checks if `size` is provided for a rectangle and returns an

Observations

Certainly! Here are some observations about the refactored code:

- **Improved Organization:** The use of the area_formulas dictionary makes the code much more organized and easier to understand. All the shape-specific logic is centralized.

- **Increased Maintainability:** Adding a new shape now only requires adding an entry to the area_formulas dictionary and potentially a check for required dimensions. This is much simpler than adding another elif block.
- **Clearer Intent:** The variable names size1 and size2 are more descriptive than x and y, making the function's parameters easier to understand in the context of different shapes.
- **More Accurate Calculations:** Using math.pi provides a more accurate result for the circle's area compared to using a fixed approximation.
- **Robust Error Handling:** The explicit checks for missing dimensions for the rectangle and the return of informative error strings make the function more robust and easier to debug.
- **Use of Lambda Functions:** The use of lambda functions within the dictionary provides a concise way to define the area calculations.

Overall, the refactored code is a significant improvement in terms of readability, maintainability, and robustness compared to the original version.

Task Description #2 – Error Handling in Legacy Code

Task: Legacy function without proper error handling

Python Code

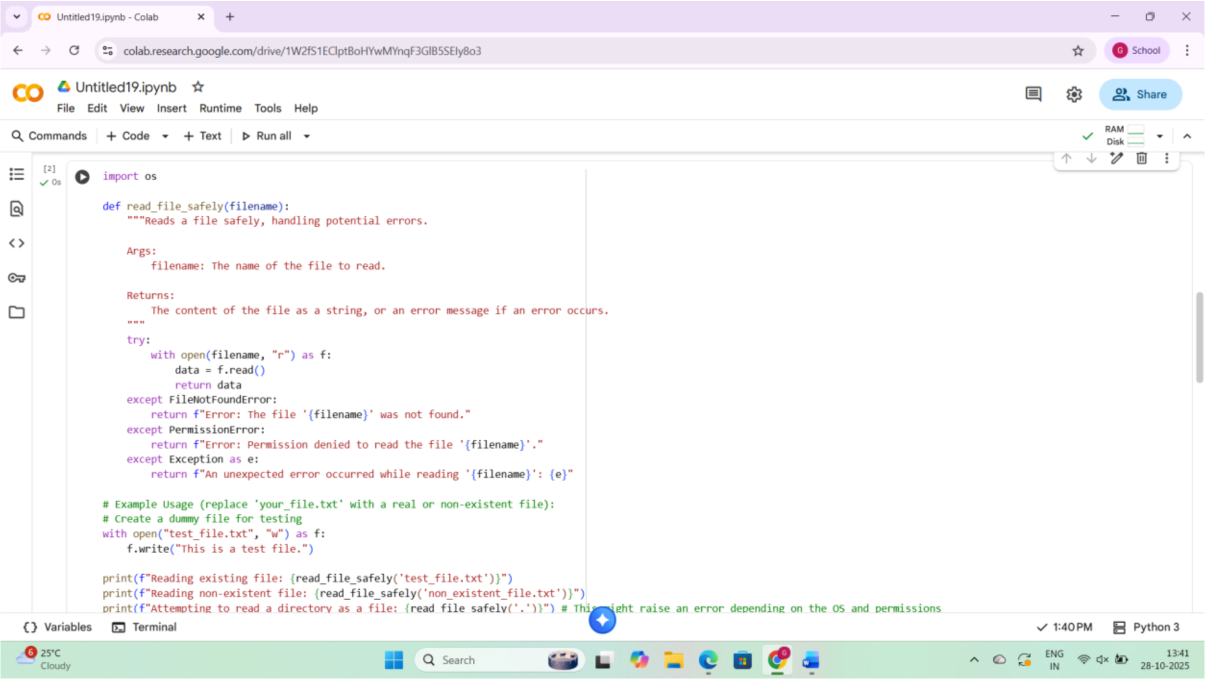
```
def read_file(filename):
f = open(filename, "r")
data = f.read()

.close()
return data
```

Prompt

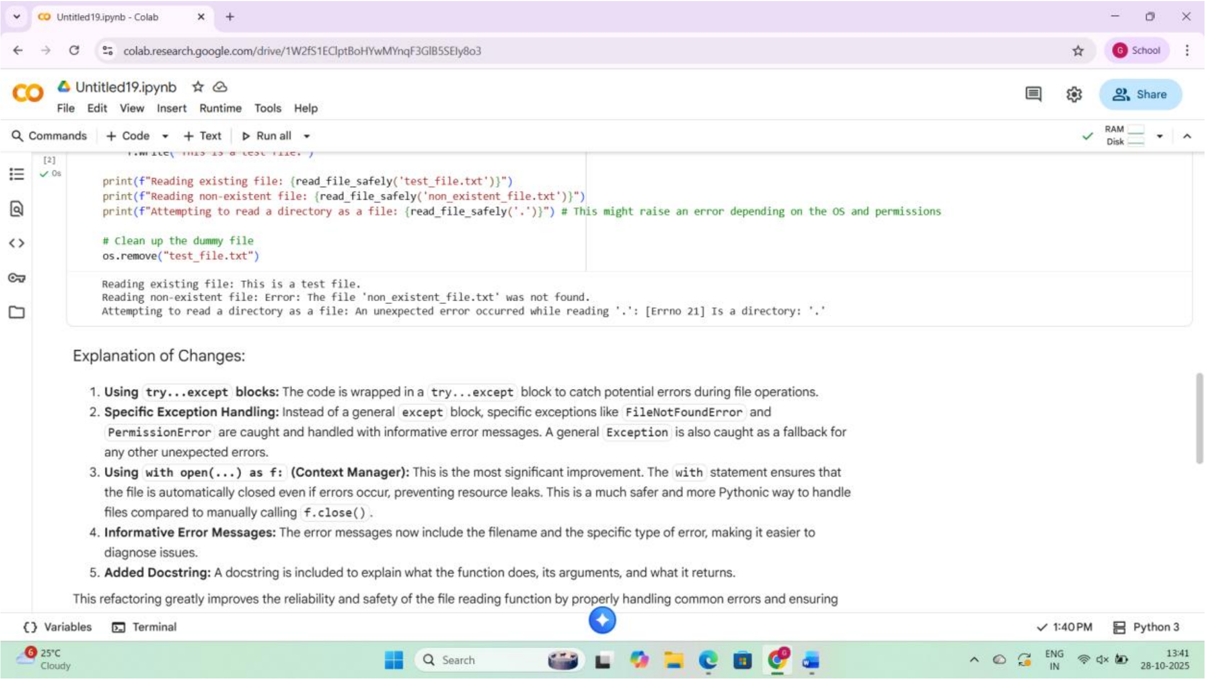
The following legacy Python function lacks proper error handling and best practices. Refactor it to safely handle errors such as missing files or permission issues.
Make sure to use modern and Pythonic approaches (like context managers) and explain how your version improves reliability and safety.

```
def read_file(filename):

  f = open(filename, "r")

  data = f.read()

  f.close()

  return data
```

code and output



```python
import os

def read_file_safely(filename):
    """Reads a file safely, handling potential errors.

    Args:
        filename: The name of the file to read.

    Returns:
        The content of the file as a string, or an error message if an error occurs.
    """
    try:
        with open(filename, "r") as f:
            data = f.read()
            return data
    except FileNotFoundError:
        return f"Error: The file '{filename}' was not found."
    except PermissionError:
        return f"Error: Permission denied to read the file '{filename}'."
    except Exception as e:
        return f"An unexpected error occurred while reading '{filename}': {e}"

# Example Usage (replace 'your_file.txt' with a real or non-existent file):
# Create a dummy file for testing
with open("test_file.txt", "w") as f:
    f.write("This is a test file.")

print(f"Reading existing file: {read_file_safely('test_file.txt')}")
print(f"Reading non-existent file: {read_file_safely('non_existent_file.txt')}")
print(f"Attempting to read a directory as a file: {read_file_safely('.')}") # This might raise an error depending on the OS and permissions
```



```python
print(f"Reading existing file: {read_file_safely('test_file.txt')}")
print(f"Reading non-existent file: {read_file_safely('non_existent_file.txt')}")
print(f"Attempting to read a directory as a file: {read_file_safely('.')}") # This might raise an error depending on the OS and permissions

# Clean up the dummy file
os.remove("test_file.txt")
```

```
Reading existing file: This is a test file.
Reading non-existent file: Error: The file 'non_existent_file.txt' was not found.
Attempting to read a directory as a file: An unexpected error occurred while reading '.': [Errno 21] Is a directory: '.'
```

Explanation of Changes:

1. **Using `try...except` blocks:** The code is wrapped in a `try...except` block to catch potential errors during file operations.
2. **Specific Exception Handling:** Instead of a general `except` block, specific exceptions like `FileNotFoundError` and `PermissionError` are caught and handled with informative error messages. A general `Exception` is also caught as a fallback for any other unexpected errors.
3. **Using `with open(...) as f:` (Context Manager):** This is the most significant improvement. The `with` statement ensures that the file is automatically closed even if errors occur, preventing resource leaks. This is a much safer and more Pythonic way to handle files compared to manually calling `f.close()`.
4. **Informative Error Messages:** The error messages now include the filename and the specific type of error, making it easier to diagnose issues.
5. **Added Docstring:** A docstring is included to explain what the function does, its arguments, and what it returns.

This refactoring greatly improves the reliability and safety of the file reading function by properly handling common errors and ensuring

Observation

- **Enhanced Safety:** The use of with open(...) ensures that the file is always closed properly, even if errors occur. This prevents resource leaks and potential issues with file corruption.

- **Specific Error Messages:** By catching specific exceptions (FileNotFoundError, PermissionError), the function can provide more informative error messages to the user, making it easier to diagnose the problem.

- **Graceful Failure:** Instead of crashing when an error occurs (like the original function would for a missing file), the refactored function returns an informative error string, allowing the calling code to handle the error gracefully.
- **Improved Readability:** The try...except block clearly separates the code that might raise an error from the code that handles those errors.
- **Pythonic Approach:** Using the with statement is the standard and recommended way to handle file operations in Python.

Task Description #3 – 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.m3ginal read_file function.
```

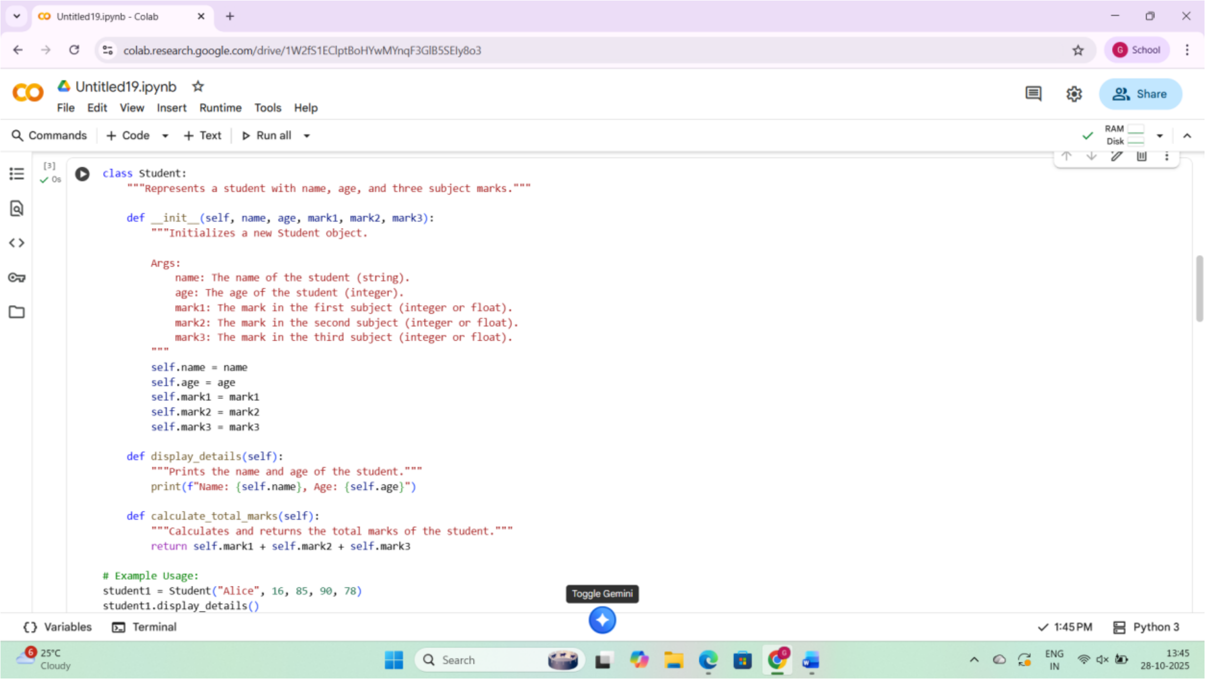
Prompt

Refactor the following legacy Python class to improve readability, modularity, and maintainability.
Use clear variable names, follow Python naming conventions, and separate concerns properly.
Also, explain how your refactored version enhances code readability and reusability.

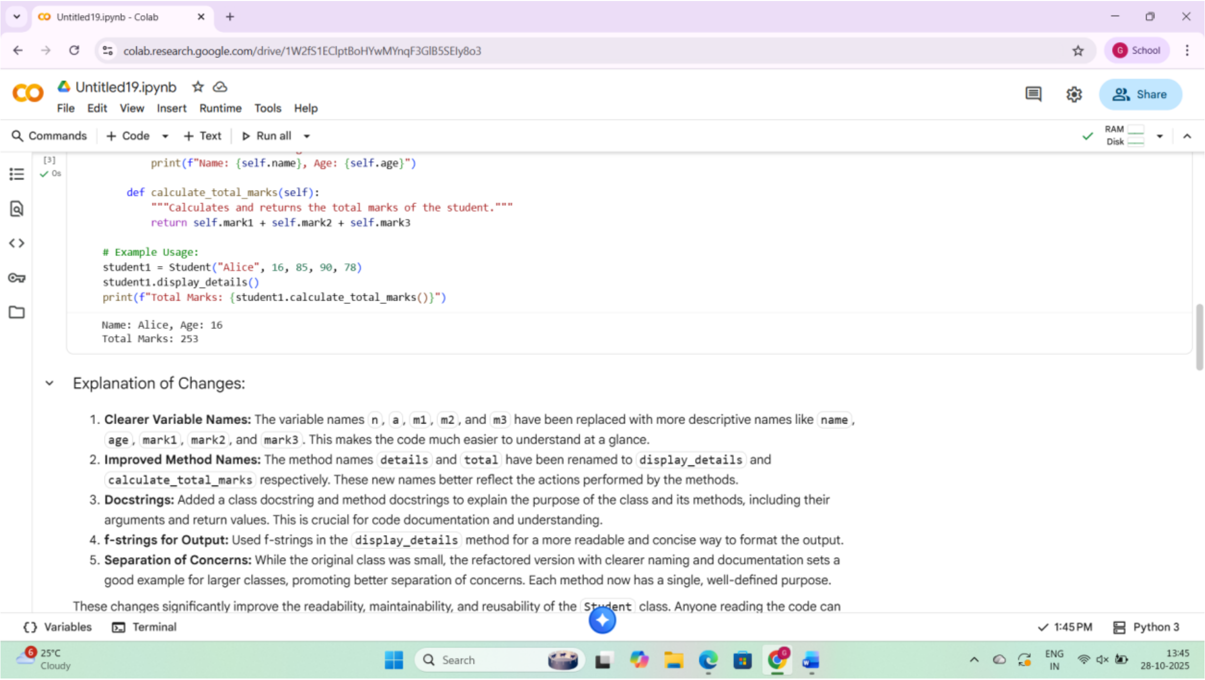
```
class Student:

  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 and output



```
class Student:
    """Represents a student with name, age, and three subject marks."""

    def __init__(self, name, age, mark1, mark2, mark3):
        """Initializes a new Student object.

        Args:
            name: The name of the student (string).
            age: The age of the student (integer).
            mark1: The mark in the first subject (integer or float).
            mark2: The mark in the second subject (integer or float).
            mark3: The mark in the third subject (integer or float).
        """
        self.name = name
        self.age = age
        self.mark1 = mark1
        self.mark2 = mark2
        self.mark3 = mark3

    def display_details(self):
        """Prints the name and age of the student."""
        print(f"Name: {self.name}, Age: {self.age}")

    def calculate_total_marks(self):
        """Calculates and returns the total marks of the student."""
        return self.mark1 + self.mark2 + self.mark3

# Example Usage:
student1 = Student("Alice", 16, 85, 90, 78)
student1.display_details()
```



```
        print(f"Name: {self.name}, Age: {self.age}")

    def calculate_total_marks(self):
        """Calculates and returns the total marks of the student."""
        return self.mark1 + self.mark2 + self.mark3

# Example Usage:
student1 = Student("Alice", 16, 85, 90, 78)
student1.display_details()
print(f"Total Marks: {student1.calculate_total_marks()}")
```

```
Name: Alice, Age: 16
Total Marks: 253
```

Explanation of Changes:

1. **Clearer Variable Names:** The variable names `n`, `a`, `m1`, `m2`, and `m3` have been replaced with more descriptive names like `name`, `age`, `mark1`, `mark2`, and `mark3`. This makes the code much easier to understand at a glance.
2. **Improved Method Names:** The method names `details` and `total` have been renamed to `display_details` and `calculate_total_marks` respectively. These new names better reflect the actions performed by the methods.
3. **Docstrings:** Added a class docstring and method docstrings to explain the purpose of the class and its methods, including their arguments and return values. This is crucial for code documentation and understanding.
4. **f-strings for Output:** Used f-strings in the `display_details` method for a more readable and concise way to format the output.
5. **Separation of Concerns:** While the original class was small, the refactored version with clearer naming and documentation sets a good example for larger classes, promoting better separation of concerns. Each method now has a single, well-defined purpose.

These changes significantly improve the readability, maintainability, and reusability of the `Student` class. Anyone reading the code can

Observations

- **Enhanced Readability:** The use of clear and descriptive variable and method names (e.g., name, age, calculate_total_marks) makes it much easier to understand the purpose of each part of the class at a glance.
- **Improved Maintainability:** The code is now more organized and self-documenting due to the clear naming and docstrings. This makes it easier to modify or extend the class in the future.
- **Better Reusability:** With clear documentation and well-defined methods, the Student class is more easily reusable in other parts of a program or in different projects.
- **Adherence to Python Conventions:** The refactored code follows standard Python naming conventions (e.g., using lowercase with underscores for method names), which is good practice.
- **Clearer Output:** Using f-strings in the display_details method provides a more concise and readable way to format the output.

Overall, the refactored Student class is a good example of how clear naming, documentation, and separation of concerns can significantly improve the quality of the code.

Task Description #4 – Inefficient Loop Refactoring

Task: Refactor this inefficient loop with AI help

Python Code

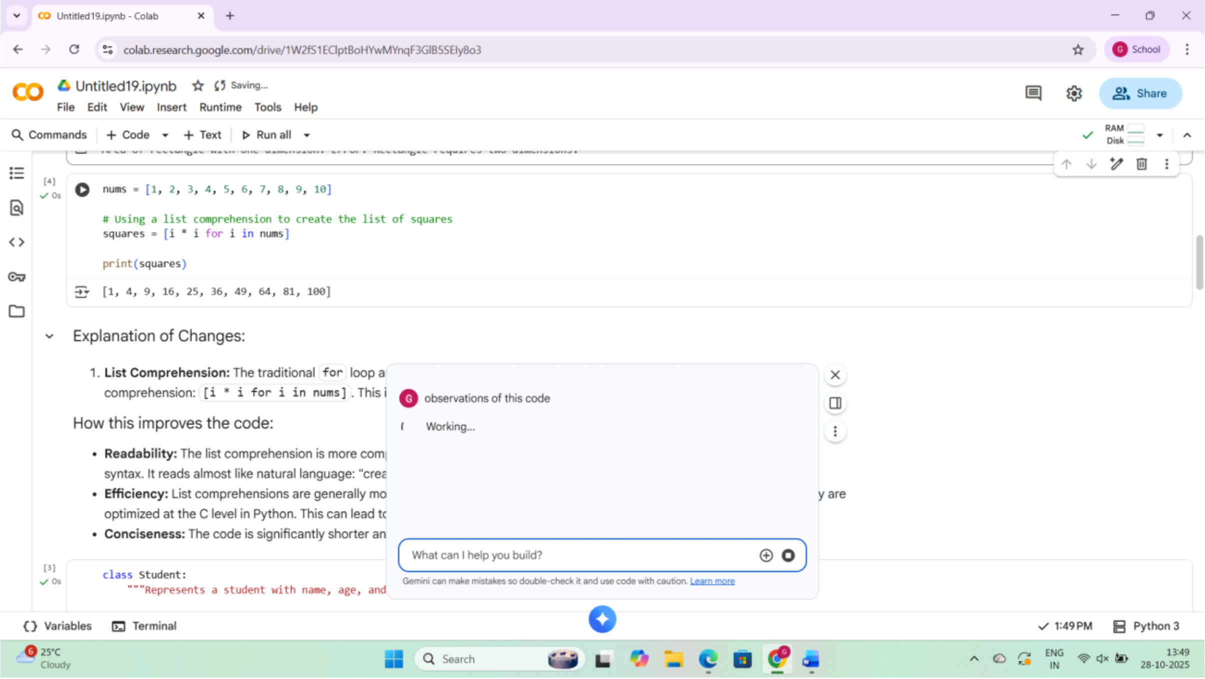
```
nums = [1,2,3,4,5,6,7,8,9,10]
squares = []
for i in nums:
squares.append(i * i)
```

Prompt

Refactor the following Python code to make it more efficient and Pythonic.
Use modern language features like list comprehensions or functional programming where appropriate.
Also, explain how your refactored version improves performance and readability.

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

squares = []

for i in nums:

  squares.append(i * i)
```

code and output



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

# Using a list comprehension to create the list of squares
squares = [i * i for i in nums]

print(squares)
```

```
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Explanation of Changes:

1. **List Comprehension:** The traditional `for` loop a
comprehension: `[i * i for i in nums]`. This i

How this improves the code:

- **Readability:** The list comprehension is more com
syntax. It reads almost like natural language: "crea
- **Efficiency:** List comprehensions are generally mo … y are
optimized at the C level in Python. This can lead t
- **Conciseness:** The code is significantly shorter an

```python
class Student:
    """Represents a student with name, age, and
```

observations of this code

Working...

What can I help you build?

Gemini can make mistakes so double-check it and use code with caution. Learn more

Observation

- **Conciseness and Readability:** The list comprehension squares = [i * i for i in nums] is significantly shorter and more expressive than the traditional for loop with append. It clearly communicates the intent of creating a list of squares.
- **Pythonic Style:** Using list comprehensions is a highly recommended and Pythonic way to create lists based on existing iterables. It's a common pattern that experienced Python developers recognize immediately.
- **Potential Performance Improvement:** While for very small lists the difference might be negligible, list comprehensions are generally more efficient than for loops with append because they are optimized at a lower level in Python.
- **Direct Creation:** The list is created directly in one line, without needing to initialize an empty list and then append to it in a separate step.