

**PROGRAM** : B.TECH/CSE  
**SPECIALIZATION** : AIML  
**COURSE TITLE** : AI ASSISTED CODING  
**COURSE CODE** : 24CS101PC214  
**SEMESTER** : 3RD  
**NAME OF THE STUDENT** : KARNAKANTI GODADEVI  
**ENROLLMENT NO** : 2403A52003  
**BATCH NO** : 01

**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
```

**Expected Output**

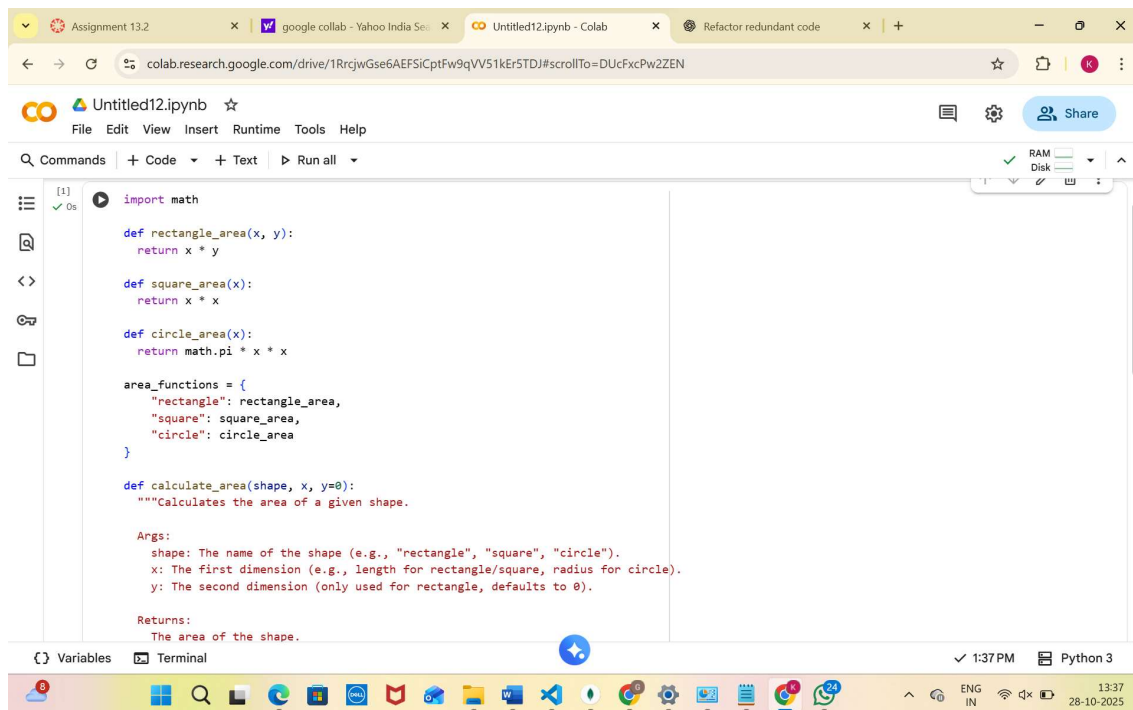
- Refactored version with dictionary-based dispatch or separate functions.
- Cleaner and modular design.

**Prompt:**

Refactor the following redundant Python code to remove repetition and make it cleaner and modular. use a dictionary-based dispatch or separate functions for each shape.

```
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&Output:**



The screenshot shows a Google Colab notebook titled 'Untitled12.ipynb'. The code defines three functions: `rectangle_area(x, y)`, `square_area(x)`, and `circle_area(x)`. These are stored in a dictionary `area_functions`. A `calculate_area(shape, x, y=0)` function is also defined with a docstring explaining its arguments and return value.

```
[1] import math

def rectangle_area(x, y):
    return x * y

def square_area(x):
    return x * x

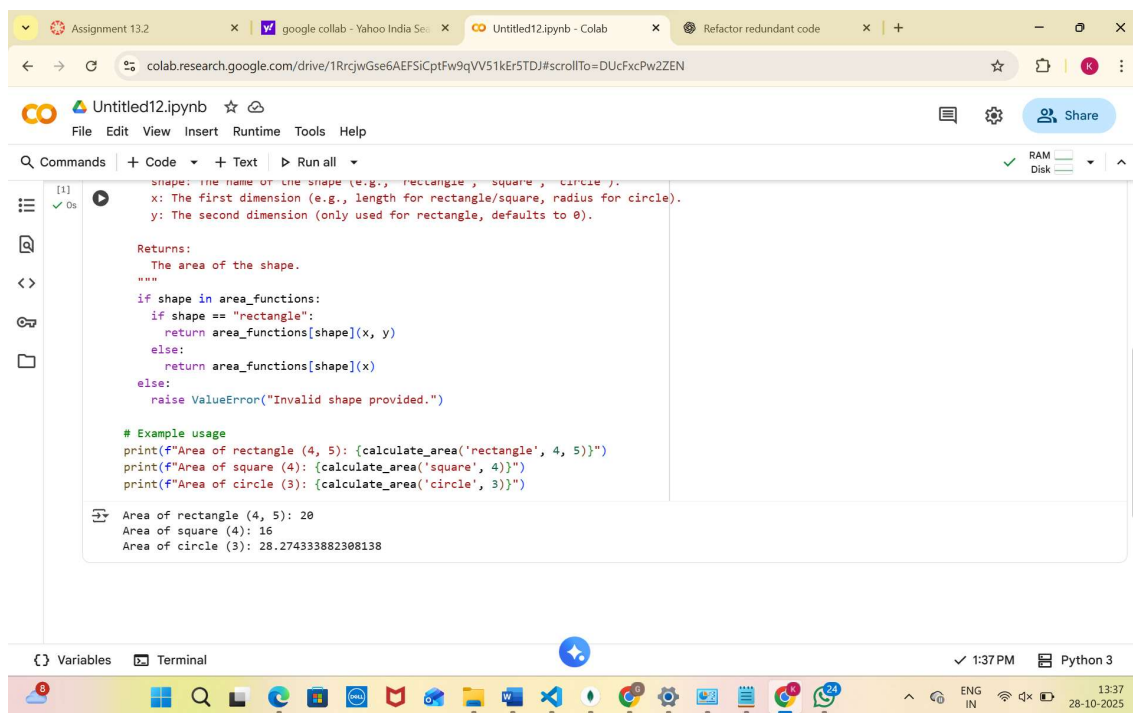
def circle_area(x):
    return math.pi * x * x

area_functions = {
    "rectangle": rectangle_area,
    "square": square_area,
    "circle": circle_area
}

def calculate_area(shape, x, y=0):
    """Calculates the area of a given shape.

    Args:
        shape: The name of the shape (e.g., "rectangle", "square", "circle").
        x: The first dimension (e.g., length for rectangle/square, radius for circle).
        y: The second dimension (only used for rectangle, defaults to 0).

    Returns:
        The area of the shape.
```



The screenshot shows the same notebook after running the code. The `calculate_area` function is updated to use the `area_functions` dictionary. The output shows the area calculations for a rectangle, square, and circle.

```
shape: The name of the shape (e.g., "rectangle", "square", "circle").
x: The first dimension (e.g., length for rectangle/square, radius for circle).
y: The second dimension (only used for rectangle, defaults to 0).

Returns:
    The area of the shape.
    """
if shape in area_functions:
    if shape == "rectangle":
        return area_functions[shape](x, y)
    else:
        return area_functions[shape](x)
else:
    raise ValueError("Invalid shape provided.")

# Example usage
print(f"Area of rectangle (4, 5): {calculate_area('rectangle', 4, 5)}")
print(f"Area of square (4): {calculate_area('square', 4)}")
print(f"Area of circle (3): {calculate_area('circle', 3)}")
```

Area of rectangle (4, 5): 20  
Area of square (4): 16  
Area of circle (3): 28.274333882308138

## Observation:

`import math`: This line imports the math module, which is needed to access the value of pi (`math.pi`) for the circle area calculation.

`rectangle_area(x, y)`: This function calculates the area of a rectangle given its length (`x`) and width (`y`).

`square_area(x)`: This function calculates the area of a square given its side length (x).

`circle_area(x)`: This function calculates the area of a circle given its radius (x) using the formula  $\pi r^2$ .

`area_functions = { ... }`: This creates a dictionary called `area_functions` that maps shape names (strings like "rectangle", "square", "circle") to their corresponding area calculation functions. This is a way to dispatch or call the correct function based on the shape name.

`calculate_area(shape, x, y=0)`: This is the main function that takes the shape name, the first dimension (x), and an optional second dimension (y) as input.

It checks if the provided shape is a key in the `area_functions` dictionary.

If it's a "rectangle", it calls the `rectangle_area` function with both x and y.

For other shapes ("square" and "circle"), it calls the corresponding function from the `area_functions` dictionary with only x.

If the shape is not found in the dictionary, it raises a `ValueError`.

Example usage: The lines at the end demonstrate how to use the `calculate_area` function to find the areas of a rectangle, a square, and a circle and print the results.

## **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()
```

```
    f.close()  
    return data
```

### **Expected Output:**

All refactors with `with open()` and `try-except`:

### **Prompt:**

Refactor the following legacy Python code to include proper error handling and use modern best practices such as `with open()`.

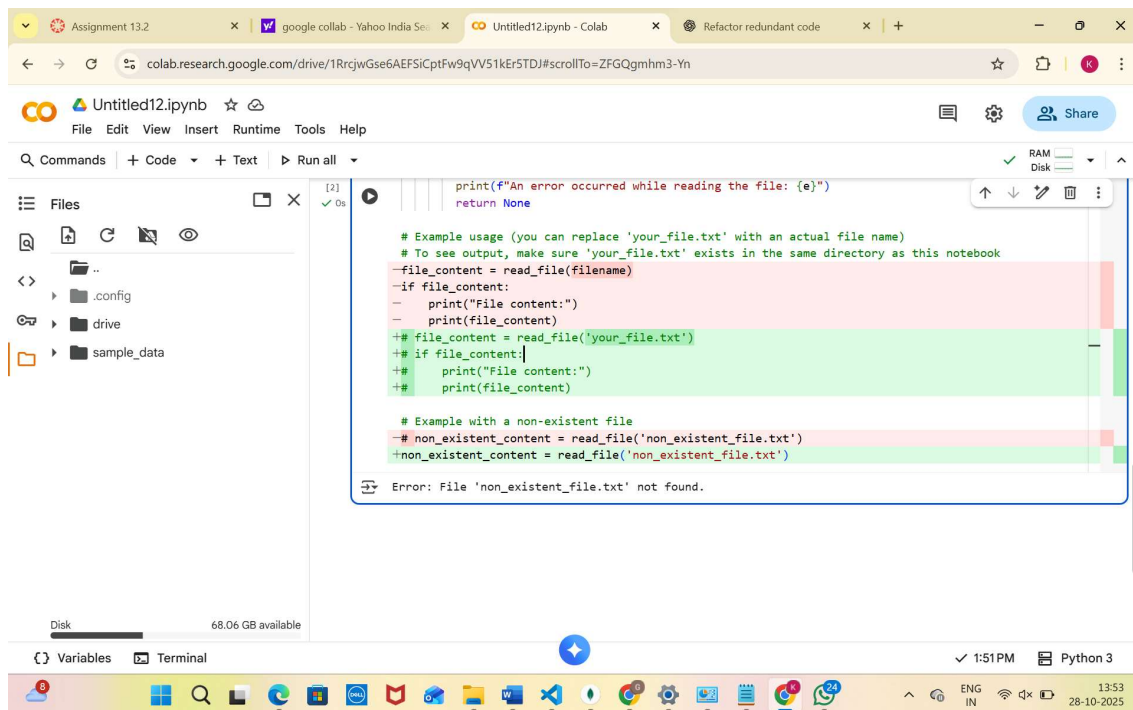
```
def read_file(filename):  
    f = open(filename, "r")
```

```
data = f.read()
```

```
f.close()
```

```
return data
```

## Code&Output:



The screenshot shows a Google Colab notebook titled 'Untitled12.ipynb'. The left sidebar displays a file explorer with a directory structure: '..' (parent), '.config', 'drive', and 'sample\_data'. The main code area contains a function `read_file(filename)` that attempts to read a file. The function has two example calls: one for a file that exists and one for a file that does not exist. The execution of the second call results in an error: 'Error: File 'non\_existent\_file.txt' not found.' The error message is displayed in a blue box at the bottom of the code cell. The notebook interface includes a top menu bar with options like File, Edit, View, Insert, Runtime, Tools, and Help. The bottom status bar shows the system time as 1:51 PM and the Python version as Python 3.

```
[2] ✓ 0s  
print(f"An error occurred while reading the file: {e}")  
return None  
  
# Example usage (you can replace 'your_file.txt' with an actual file name)  
# To see output, make sure 'your_file.txt' exists in the same directory as this notebook  
-file_content = read_file(filename)  
-if file_content:  
-    print("File content:")  
-    print(file_content)  
+## file_content = read_file('your_file.txt')  
+## if file_content:  
+##     print("File content:")  
+##     print(file_content)  
  
# Example with a non-existent file  
-# non_existent_content = read_file('non_existent_file.txt')  
+non_existent_content = read_file('non_existent_file.txt')  
  
Error: File 'non_existent_file.txt' not found.
```

```
[2] ✓ 0s
print(f"An error occurred while reading the file: {e}")
return None

# Example usage (you can replace 'your_file.txt' with an actual file name)
# To see output, make sure 'your_file.txt' exists in the same directory as this notebook
file_content = read_file(filename)
if file_content:
    print("File content:")
    print(file_content)
+## file_content = read_file('your_file.txt')
+## if file_content:
+##     print("File content:")
+##     print(file_content)

# Example with a non-existent file
-# non_existent_content = read_file('non_existent_file.txt')
+non_existent_content = read_file('non_existent_file.txt')

Error: File 'non_existent_file.txt' not found.
```

## Observation:

- `def read_file(filename)::` This line defines the function `read_file` that takes one argument, `filename`, which is the name of the file to be read.
- `Docstring:` The text within the triple quotes explains what the function does, its arguments, and what it returns.
- `try...except` block: This block is used for error handling.
  - `try::` The code inside this block is attempted to be executed.
  - `with open(filename, "r") as f::` This opens the file specified by `filename` in read mode ("r"). The `with` statement ensures that the file is automatically closed even if errors occur. The opened file object is assigned to the variable `f`.
  - `data = f.read():` This reads the entire content of the file and stores it in the `data` variable.
  - `return data:` If the file is read successfully, the function returns the file content.
  - `except FileNotFoundError::` If a `FileNotFoundError` occurs (meaning the file doesn't exist), the code inside this block is executed. It prints an error message indicating that the file was not found.

- except Exception as e:: This catches any other potential exceptions that might occur during file reading. It prints a generic error message along with the specific error that occurred.
- return None: If any exception occurs, the function returns None.

The commented-out sections show examples of how to use the read\_file function with an existing file and a non-existent file.

### 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.m3

#### Expected Output:

- AI improves naming (name, age, marks).
- Adds docstrings.
- Improves print readability.
- Possibly uses sum(self.marks) if marks stored in a list

#### Prompt:

Refactor the following legacy Python class to improve readability, naming, and modularity.

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
```

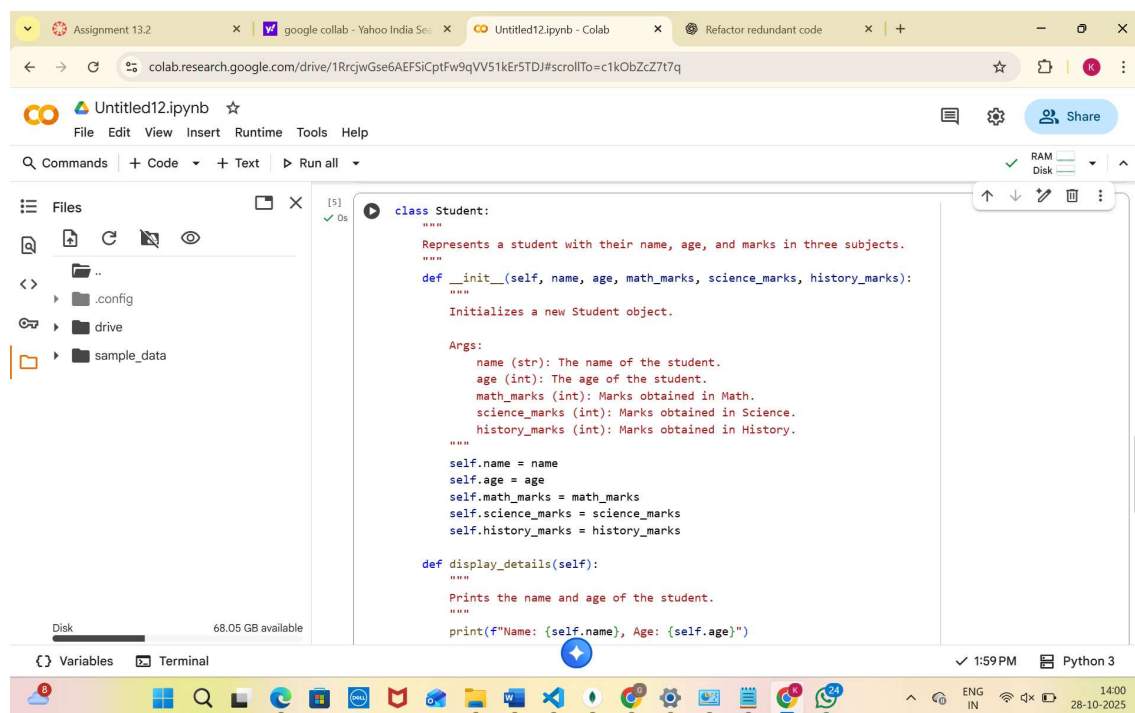
```
def details(self):
```

```
    print("Name:", self.n, "Age:", self.a)
```

```
def total(self):
```

```
    return self.m1 + self.m2 + self.m3
```

## Code & Output:



The screenshot shows a Google Colab notebook interface. The browser address bar displays the URL: `colab.research.google.com/drive/1RrcjwGse6AEFSICptFw9qVV51kEr5TDJ#scrollTo=c1kObZcZ7l7q`. The notebook title is "Untitled12.ipynb". The left sidebar shows a file explorer with folders named `..`, `.config`, `drive`, and `sample_data`. The main editor area contains the following Python code:

```
class Student:
    """
    Represents a student with their name, age, and marks in three subjects.
    """
    def __init__(self, name, age, math_marks, science_marks, history_marks):
        """
        Initializes a new Student object.

        Args:
            name (str): The name of the student.
            age (int): The age of the student.
            math_marks (int): Marks obtained in Math.
            science_marks (int): Marks obtained in Science.
            history_marks (int): Marks obtained in History.
        """
        self.name = name
        self.age = age
        self.math_marks = math_marks
        self.science_marks = science_marks
        self.history_marks = history_marks

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

The bottom status bar indicates the time is 1:59 PM, the environment is Python 3, and the date is 28-10-2025.

```
class Student:
    def __init__(self, name, age, math_marks, science_marks, history_marks):
        self.name = name
        self.age = age
        self.math_marks = math_marks
        self.science_marks = science_marks
        self.history_marks = history_marks

    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 the total marks obtained by the student.

        Returns:
            int: The total marks.
        """
        return self.math_marks + self.science_marks + self.history_marks

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

Name: Alice, Age: 16  
Total Marks: 253

## Observation:

- `class Student::` This line defines the beginning of the class.
- `__init__(self, name, age, math_marks, science_marks, history_marks):` This is the constructor method. It's called when you create a new Student object. It initializes the object's attributes (`self.name, self.age, self.math_marks, self.science_marks, self.history_marks`) with the values passed as arguments.
- `display_details(self):` This method prints the student's name and age to the console.
- `calculate_total_marks(self):` This method calculates the sum of the student's marks in math, science, and history and returns the total.

The commented-out section at the end shows an example of how to create a Student object, display its details, and calculate its total marks.



## 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)
```

**Expected Output: AI suggested a list comprehension**

**Prompt:**

Refactor the following inefficient Python loop to make it more concise and Pythonic.

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

```
squares = []
```

```
for i in nums:
```

```
    squares.append(i * i)
```

## Code&Output:

The screenshot shows a Google Colab notebook interface. The top bar includes tabs for 'Assignment 13.2', 'google collab - Yahoo India Se...', 'Untitled12.ipynb - Colab', and 'Refactor redundant code'. The address bar shows the Colab URL. The notebook's menu bar includes 'File', 'Edit', 'View', 'Insert', 'Runtime', 'Tools', and 'Help'. On the left, a file explorer shows folders like '.config', 'drive', and 'sample\_data'. The main code area contains two cells. The first cell, labeled '[5]', shows a function call: `total_marks = student1.calculate_total_marks()` followed by `print(f"Total Marks: {total_marks}")`. Below the code, the output is displayed: 'Name: Alice, Age: 16' and 'Total Marks: 253'. The second cell, labeled '[6]', is titled 'Gemini' and shows the refactored code: `nums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]`, `squares = [i * i for i in nums]`, and `print(squares)`. The output of this cell is the list `[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]`. The bottom status bar shows '2:05 PM', 'Python 3', and the date '28-10-2025'.

**Observation:**

- `nums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]`: This line initializes a list named `nums` containing integers from 1 to 10.
- `squares = [i * i for i in nums]`: This is a list comprehension. It's a concise way to create lists. It iterates through each element `i` in the `nums` list and creates a new list `squares` where each element is the square of `i` (`i * i`).
- `print(squares)`: This line prints the `squares` list to the console.