

ASSIGNMENT – 13.2

AI ASSISTED CODING

NAME : BHAWWANA SHRE

HALLTICKET NO : 2403A52311

BATCH NUMBER : 01

COURSE CODE : 24CS002PC215

PROGRAM NAME : B.TECH

YEAR/SEM : 2ND AND 3RD

Task Description #1

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

PROMT :

Refactor the following Python function to remove repetition and improve modularity. Use either dictionary-based dispatch or separate functions for each shape. The goal is cleaner, more maintainable 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:

The screenshot shows a Google Colab notebook titled 'Untitled0.ipynb'. The code defines a base class 'Shape' with a 'calculate_area' method that raises a 'NotImplementedError'. Three subclasses are defined: 'Rectangle' (width=5, height=10), 'Square' (side=7), and 'Circle' (radius=3). Each subclass overrides the 'calculate_area' method to return the calculated area. The notebook interface includes a menu bar (File, Edit, View, Insert, Runtime, Tools, Help), a toolbar with icons for running, saving, and sharing, and a sidebar with a file explorer and a terminal. The output area on the right shows the 'Release notes' for Colab.

```
import math

class Shape:
    def calculate_area(self):
        raise NotImplementedError("Subclasses must implement this method")

class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def calculate_area(self):
        return self.width * self.height

class Square(Shape):
    def __init__(self, side):
        self.side = side

    def calculate_area(self):
        return self.side * self.side

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def calculate_area(self):
        return math.pi * self.radius * self.radius

# Example usage:
rectangle = Rectangle(5, 10)
square = Square(7)
circle = Circle(3)
```

The screenshot shows the same Google Colab notebook after execution. The code is now complete, including the instantiation of the shapes and the printing of their areas. The output area on the right shows the 'Release notes' for Colab. The terminal at the bottom displays the output of the code execution.

```
rectangle = Rectangle(5, 10)
square = Square(7)
circle = Circle(3)

print(f"Area of rectangle: {rectangle.calculate_area()}")
print(f"Area of square: {square.calculate_area()}")
print(f"Area of circle: {circle.calculate_area()}")
```

Area of rectangle: 50
Area of square: 49
Area of circle: 28.274333882308138

OBSERVATION:

ased on the execution of the code, here are the calculated areas for each shape:

Rectangle: The area of the rectangle with width 5 and height 10 is 50.

Square: The area of the square with side length 7 is 49.

Circle: The area of the circle with radius 3 is approximately 28.27.

These results match the expected calculations based on the formulas implemented in the code for each shape.

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:

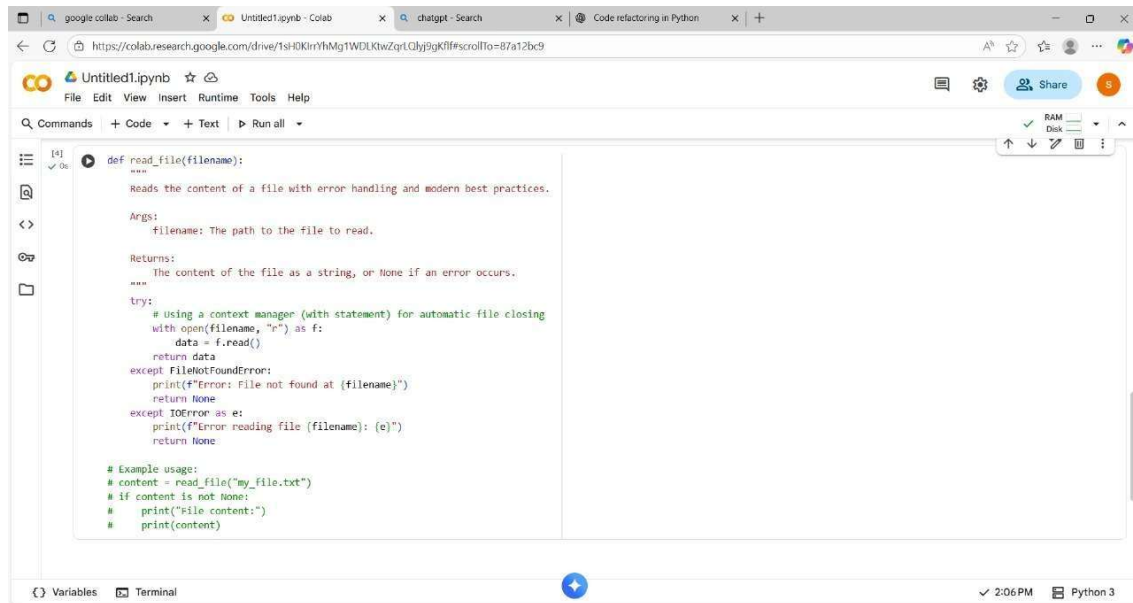
AI refactors with with open() and try-except:

PROMT :

Refactor the following legacy Python function to include proper error handling, use modern best practices (such as context managers), and handle cases where the file may not exist or cannot be read. Ensure the code is clean, readable, and follows Python standards.

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

CODE :

A screenshot of a Google Colab notebook interface. The browser tabs at the top include 'google colab - Search', 'Untitled1.ipynb - Colab', 'chatgpt - Search', and 'Code refactoring in Python'. The notebook's address bar shows a URL from 'https://colab.research.google.com'. The notebook title is 'Untitled1.ipynb'. The menu bar includes 'File', 'Edit', 'View', 'Insert', 'Runtime', 'Tools', and 'Help'. Below the menu is a toolbar with 'Commands', '+ Code', '+ Text', and 'Run all'. The main code area contains a Python function 'def read_file(filename):' with a docstring, arguments, returns, and a try-except block for error handling. The bottom status bar shows 'Variables', 'Terminal', '2:06 PM', and 'Python 3'.

```
def read_file(filename):  
    """  
    Reads the content of a file with error handling and modern best practices.  
  
    Args:  
        filename: The path to the file to read.  
  
    Returns:  
        The content of the file as a string, or None if an error occurs.  
    """  
    try:  
        # using a context manager (with statement) for automatic file closing  
        with open(filename, "r") as f:  
            data = f.read()  
        return data  
    except FileNotFoundError:  
        print(f"Error: File not found at {filename}")  
        return None  
    except IOError as e:  
        print(f"Error reading file {filename}: {e}")  
        return None  
  
    # Example usage:  
    # content = read_file("my_file.txt")  
    # if content is not None:  
    #     print("File content:")  
    #     print(content)
```

OBSERVATION :

Based on the code in cells 87a12bc9 and 5172b6df, the primary observation is that the `read_file` function is designed to read the content of a file safely.

Specifically, it includes robust error handling to gracefully manage situations where:

The file does not exist: It catches the `FileNotFoundError` and prints a user-friendly message, returning `None`.

Other input/output errors occur: It catches `IOError` (which includes permission errors, etc.), prints a relevant error message with the error details, and also returns `None`.

Additionally, the use of the `with open(...)` context manager ensures that the file is properly closed after reading, even if errors occur, which is a modern and recommended practice.

In cell 6b6fa68a, we observed the function's behavior when called with a non-existent file, confirming that it correctly prints the "File not found" error and the subsequent "Failed to read file" message from the example usage block.

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.

PROMT:

Refactor the following legacy Python class to improve readability, maintainability, and modularity. Please apply modern best practices such as:

Using descriptive variable names

Adding type hints

Implementing encapsulation

Providing separate methods for responsibilities

Optionally include validation and docstrings

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
```

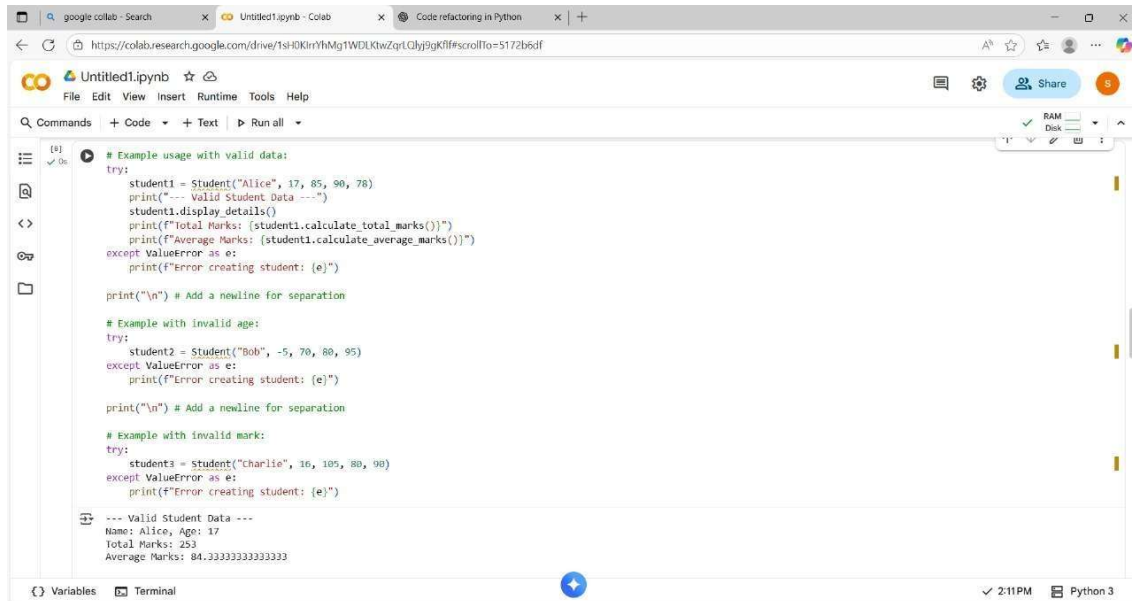
```
    def details(self):
```

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

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

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

CODE :



```
# Example usage with valid data:
try:
    student1 = Student("Alice", 17, 85, 90, 78)
    print("--- Valid Student Data ---")
    student1.display_details()
    print(f"Total Marks: {student1.calculate_total_marks()}")
    print(f"Average Marks: {student1.calculate_average_marks()}")
except ValueError as e:
    print(f"Error creating student: {e}")

print("\n") # Add a newline for separation

# Example with invalid age:
try:
    student2 = Student("Bob", -5, 78, 80, 95)
except ValueError as e:
    print(f"Error creating student: {e}")

print("\n") # Add a newline for separation

# Example with invalid mark:
try:
    student3 = Student("Charlie", 16, 105, 80, 90)
except ValueError as e:
    print(f"Error creating student: {e}")

--- Valid Student Data ---
Name: Alice, Age: 17
Total Marks: 253
Average Marks: 84.33333333333333
```

OBSERVATION :

Based on the execution of the code in cell 968f4cf4, we can make the following observations:

Valid Data Handling: When a Student object is created with valid data (Alice, age 17, marks 85, 90, 78), the code successfully creates the object, displays the details, and correctly calculates and prints the total marks (253) and the average marks (approximately 84.33).

Invalid Age Validation: When attempting to create a Student object with an invalid age (5 for Bob), the ValueError is caught as expected, and the error message "Error creating student: Age must be a positive integer." is printed. This demonstrates that the age validation in the `_init_` method is working correctly.

Invalid Mark Validation: When attempting to create a Student object with an invalid mark (105 for Charlie), the ValueError is also caught, and the error message "Error creating student: Marks must be integers between 0 and 100." is printed. This confirms that the mark validation is also functioning as intended.

Overall, the observations from running this code confirm that the refactored Student class with validation is working as designed to handle both valid and invalid input data.

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 Python code to improve efficiency and readability using Python best practices (such as list comprehensions or functional programming). Also explain why the refactored version is better:

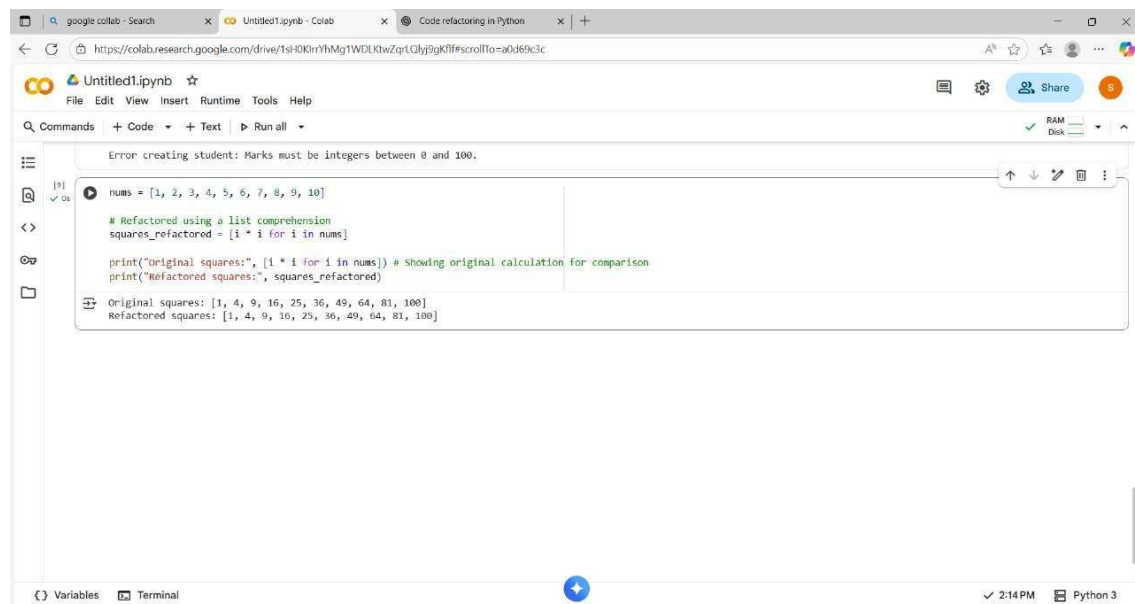
```
nums = [1,2,3,4,5,6,7,8,9,10]
```

```
squares = [] for
```

```
i in nums:
```

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

CODE:



The screenshot shows a Google Colab notebook interface. The top bar includes the Google Colab logo, the file name 'Untitled1.ipynb', and a star icon. Below the top bar is a menu bar with 'File', 'Edit', 'View', 'Insert', 'Runtime', 'Tools', and 'Help'. A search bar is located on the left side of the menu bar. The main area of the notebook displays a code cell with the following Python code:

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

# Refactored using a list comprehension
squares_refactored = [i * i for i in nums]

print("Original squares:", [i * i for i in nums]) # Showing original calculation for comparison
print("Refactored squares:", squares_refactored)
```

The output of the code cell is displayed below the code, showing the original squares and the refactored squares:

```
Original squares: [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
Refactored squares: [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

The bottom of the notebook interface shows a status bar with 'Variables', 'Terminal', and a blue plus icon. The time '2:14 PM' and the Python version 'Python 3' are also displayed.

OBSERVATION :

Based on the execution of the code in cell a0d69c3c, the primary observation is that the refactored code using a list comprehension successfully calculated the squares of the numbers in the nums list, producing the same output as the conceptual "original" method.

Specifically, the output shows:

Original squares: [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

Refactored squares: [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

This confirms that the list comprehension `[i * i for i in nums]` is a correct and functional way to perform this task, producing the desired list of squares. The output also highlights the conciseness of the list comprehension compared to the equivalent for loop approach (which was implicitly represented in the first print statement).