# NAME : G.SAI AMRUTHA

# ENROLLMENT NO : 2403A52017

# 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

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

## 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

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

## 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

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

### 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

A screenshot of a computer

AI-generated content may be incorrect.

### 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.