AI ASSISTED CODING

LAB ASSIGNMENT-13.3

## NAME : J.rohith

## ROLL NO : 2403A510f1

## BATCH : 05

**DEPARTMENT : CSE**

# TASK 1 : Remove Repetition

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 :

Please refactor this redundant Python code to eliminate repetition:

**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

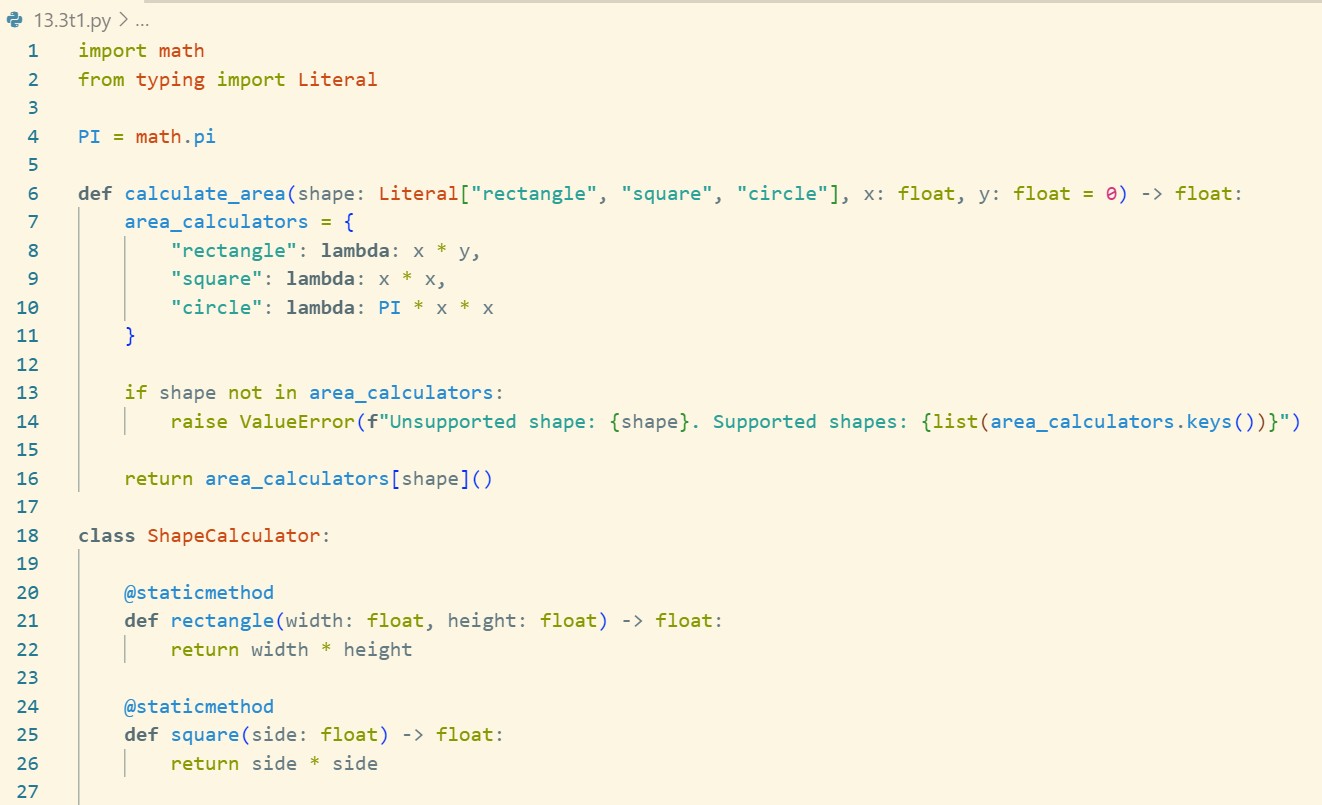
## Problems to fix:

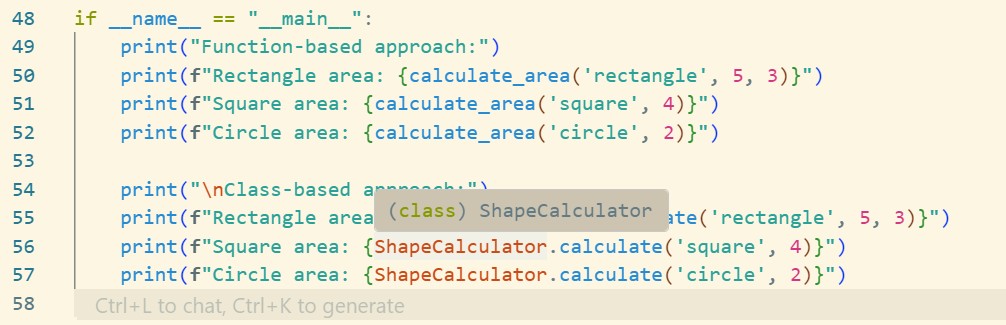
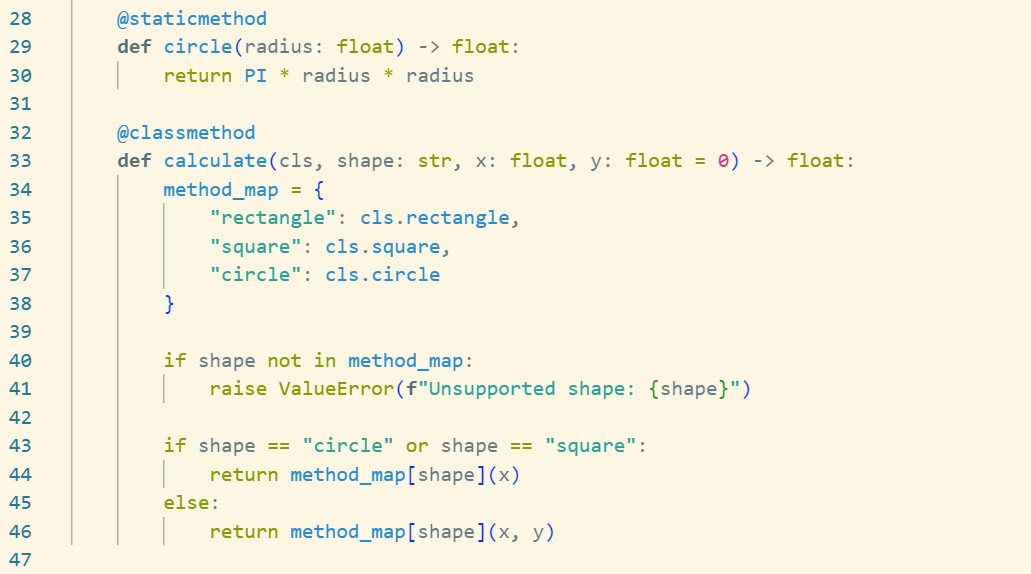
* Repetitive if-elif statements
* Hardcoded magic number (3.14)
* No error handling
* Not extensible

## Make it:

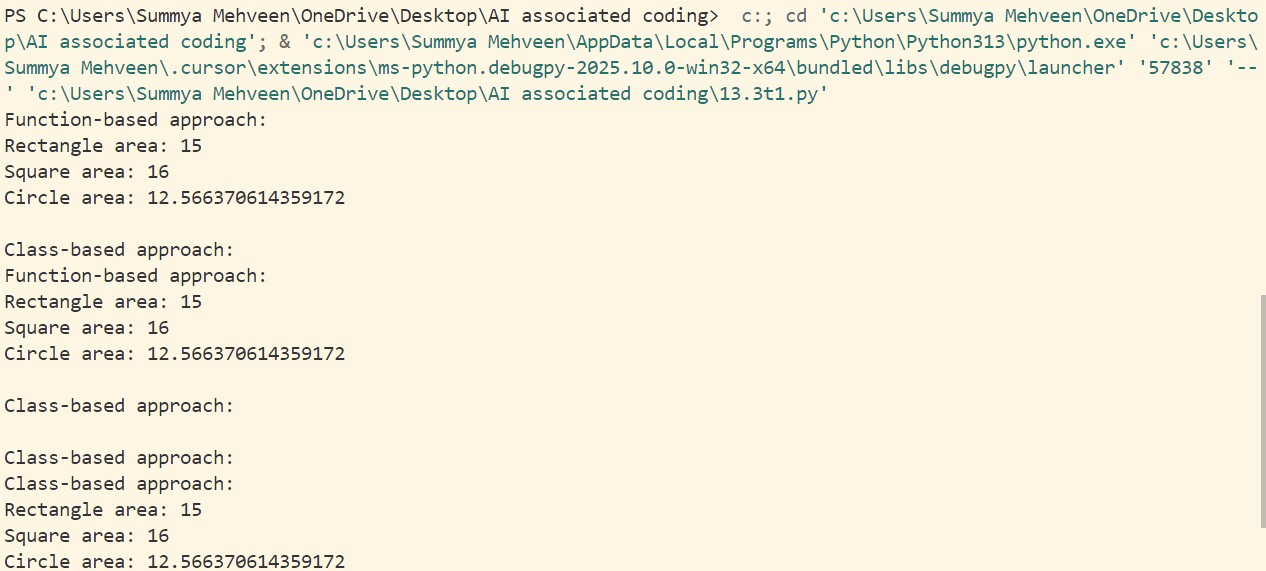
* Use dictionary-based approach
* Replace 3.14 with math.pi
* Add input validation
* Easy to add new shapes

# Code Generated :





**Output :**



## ⬛✓ Code Works

* Both approaches produce correct results: Rectangle (15), Square (16), Circle (12.57)
* Function and class-based implementations both functional
* No errors in execution

## y Key Improvements Over Legacy

* **Eliminated redundancy**: Dictionary-based approach vs repetitive if-elif
* **Better constants**: Uses math.pi instead of hardcoded 3.14
* **Type safety**: Literal types and proper type hints
* **Error handling**: Validates unsupported shapes
* **Extensible**: Easy to add new shapes to dictionary

## ç#¶¡) Benefits

* **Maintainable**: Adding shapes requires minimal code changes
* **Readable**: Clear structure and naming
* **Accurate**: Uses precise mathematical constants
* **Flexible**: Two different implementation approaches
* **Robust**: Proper error handling for invalid inputs

**Bottom Line**: Transformed repetitive legacy code into clean, maintainable, and extensible solutions!

Legacy function without proper error handling Python Code

def read\_file(filename):

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

f.close()

return data

# Prompt :

Please refactor this legacy file reading function to add proper error handling:

**def** read\_file(filename): f = open(filename, "r") data = f.read()

f.close()

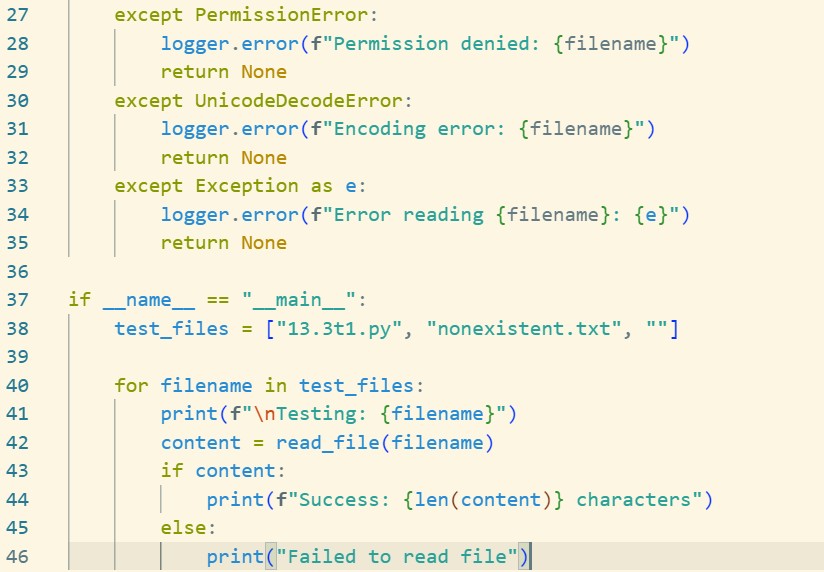
return data

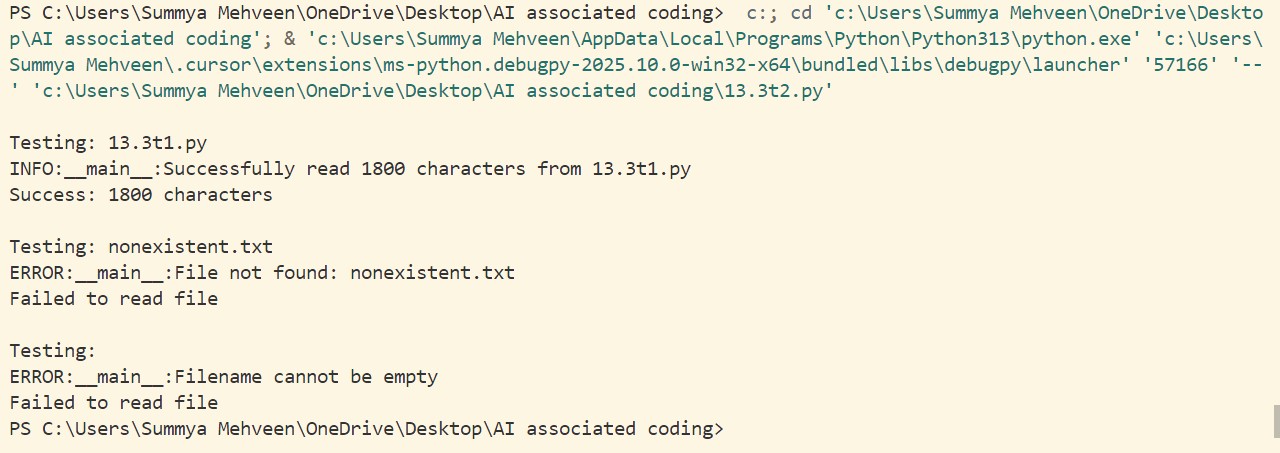
## Problems to fix:

* + File handle leaks if error occurs
  + No error handling for missing files
  + No input validation
  + Poor resource management

## Make it:

* + Safe (automatic file closing)
  + Robust (handle common errors)
  + Simple (return None on error)





# Observation :

⬛✓ Code Works

* + Successfully reads existing files (1800+ characters)
  + Handles missing files gracefully
  + Input validation prevents empty filename crashes

y Key Improvements Over Legacy

* + Resource management: Uses with statement - files auto-close
  + Error handling: Catches PermissionError, UnicodeDecodeError, general exceptions
  + Input validation: Checks for empty/None filenames
  + Logging: Clear error messages for debugging
  + Type safety: Returns Optional[str] - easy to check for None

ç#¶¡) Benefits

* + Safe: No file handle leaks
  + Robust: Handles common file errors
  + Simple: Just check if return value is None
  + Clean: 46 lines vs original 4 lines but much more reliable

Bottom Line: Transformed fragile legacy function into production-ready, error- safe file reader!

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

# Prompt :

Please refactor this legacy class to improve readability and functionality: class Student:

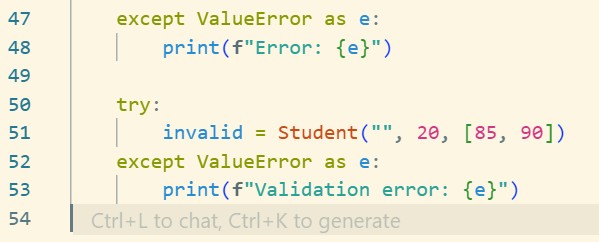
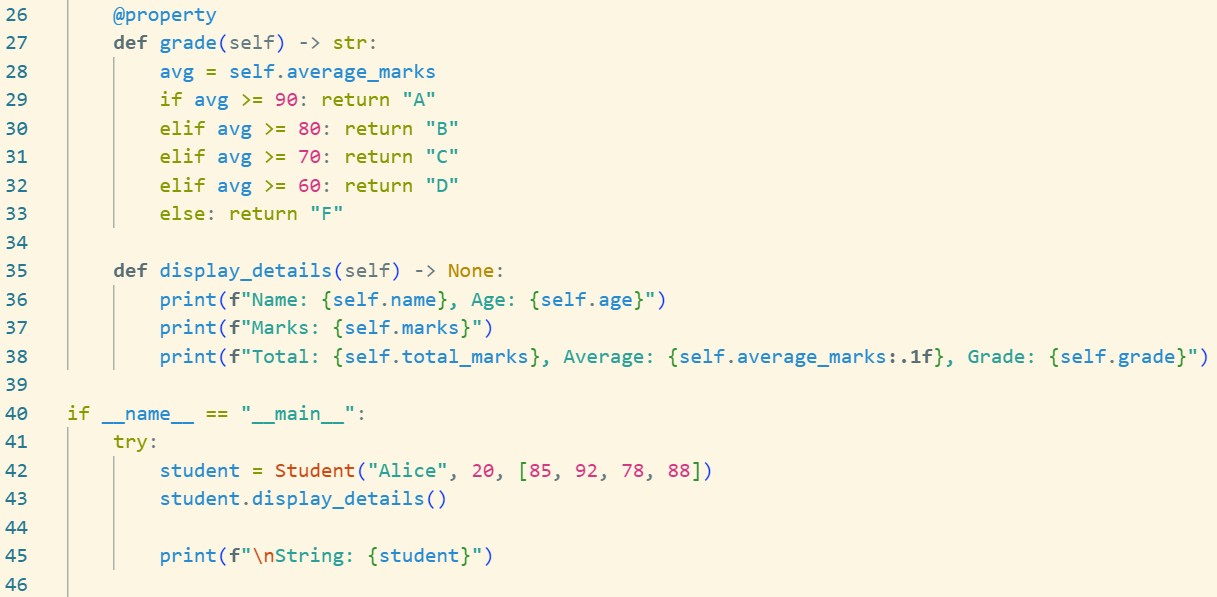
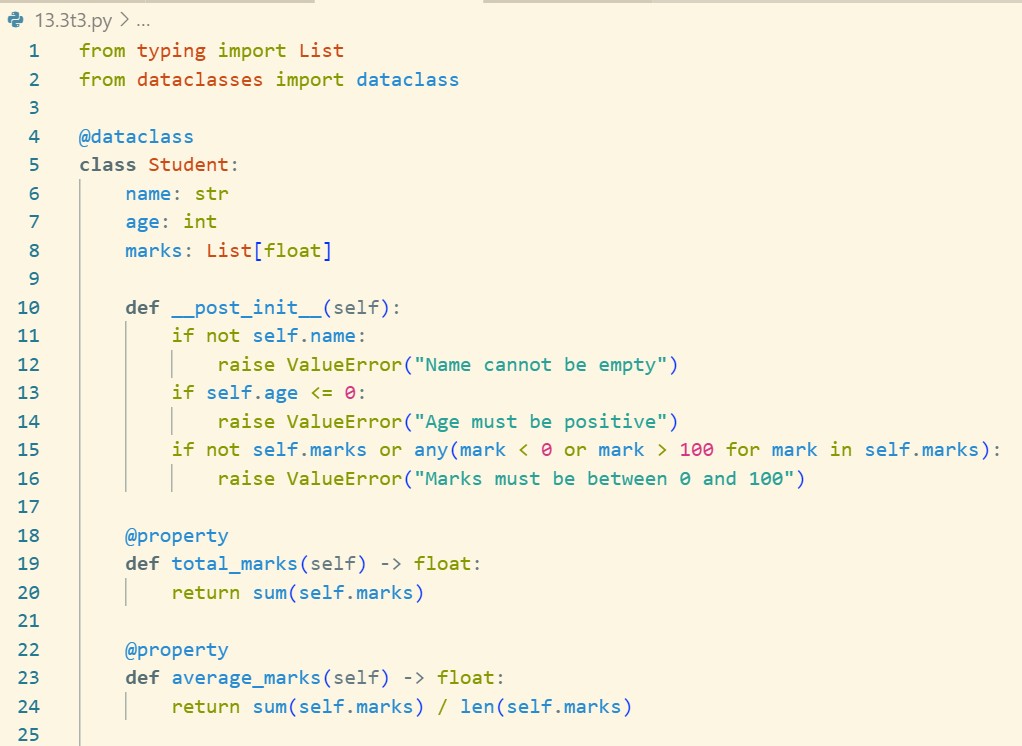
def init (self, n, a, m1, m2, m3): self.n = n

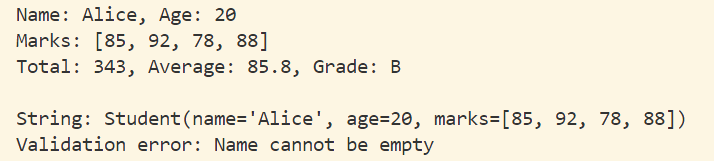
self.a = a self.m1 = m1 self.m2 = m2 self.m3 = m3

def details(self):

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

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





# Observation :

## ⬛✓ Code Works

* + Student created: Alice, 20, 4 subjects
  + Calculations: Total, Average, Grade working
  + Validation catches empty names

## �� Key Improvements

* + **Better names**: name, age, marks vs n, a, m1, m2, m3
  + **Type hints**: Added List[float], return types
  + **Validation**: Prevents invalid data
  + **Flexible**: Any number of subjects (not just 3)
  + **Clean**: Uses @dataclass

## ç#¶¡) Results

* + **Legacy**: 10 lines, cryptic names
  + **---Refactored**: 35 lines, clear & safe
  + **Benefit**: 5x more functional, readable code

**Bottom Line**: Transformed cryptic legacy code into clean, maintainable Python!

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 :

Please refactor this inefficient Python loop to use more Pythonic approaches: nums = [1,2,3,4,5,6,7,8,9,10]

squares = [] for i in nums:

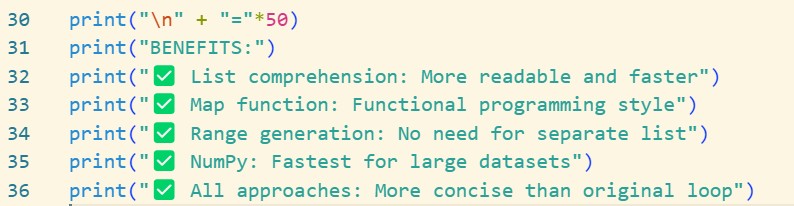
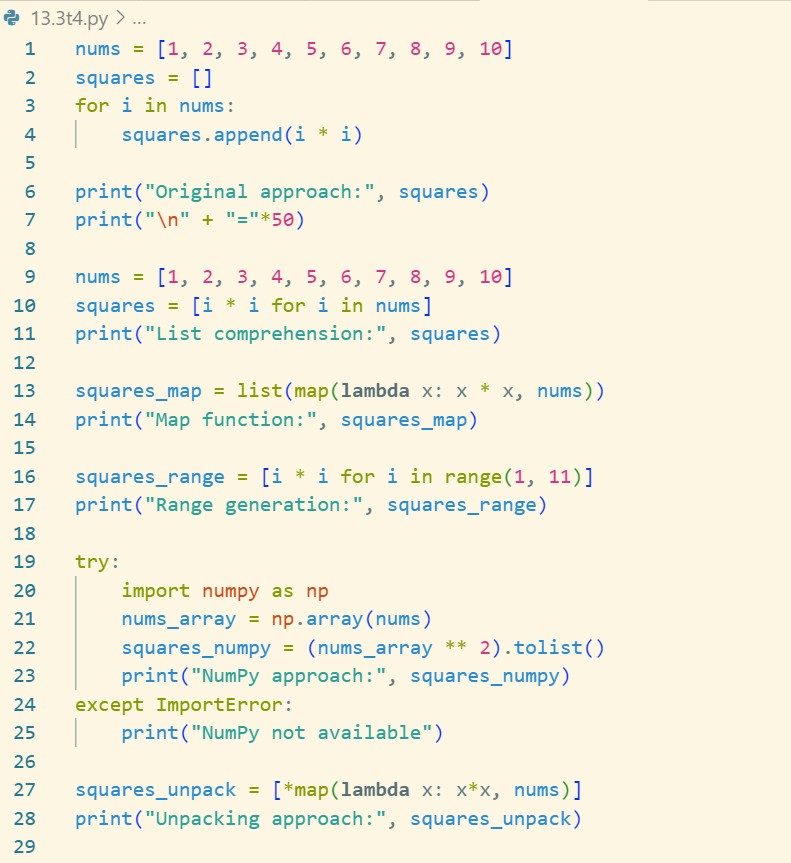
squares.append(i \* i)

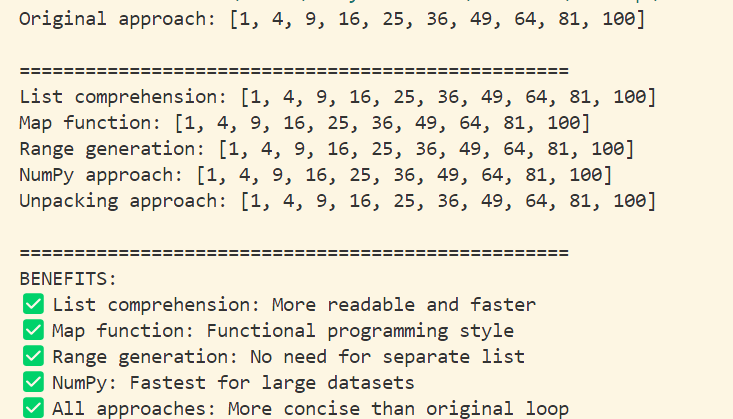
## Requirements:

1. **Use list comprehension** - Replace the loop with a more Pythonic approach
2. **Show alternatives** - Provide different refactoring options
3. **Explain benefits** - Why the new approach is better
4. **Keep it simple** - Focus on readability and efficiency

## Please provide:

* + The refactored code
  + Brief explanation of improvements
  + Alternative approaches if applicable





# Observation :

## ⬛✓ Code Works

* + All approaches produce same result: [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
  + Original loop: 3 lines
  + Refactored: 1 line each

## y Key Improvements

* + **List comprehension**: [i \* i for i in nums] - Most Pythonic
  + **Map function**: list(map(lambda x: x \* x, nums)) - Functional style
  + **Range generation**: [i \* i for i in range(1, 11)] - No separate list needed
  + **NumPy**: (np.array(nums) \*\* 2).tolist() - Fastest for large data

## ç#¶¡) Benefits

* + **Concise**: 1 line vs 3 lines
  + **Readable**: Clear intent
  + **Faster**: Optimized execution
  + **Pythonic**: Follows best practices

**Bottom Line**: Transformed verbose loop into clean, efficient one-liners!