

AI ASSISTED CODING

LAB ASSIGNMENT-13.3

NAME : Neha Shazneen

ROLL NO : 2403A510A1

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 :

```
13.3t1.py > ...
1  import math
2  from typing import Literal
3
4  PI = math.pi
5
6  def calculate_area(shape: Literal["rectangle", "square", "circle"], x: float, y: float = 0) -> float:
7      area_calculators = {
8          "rectangle": lambda: x * y,
9          "square": lambda: x * x,
10         "circle": lambda: PI * x * x
11     }
12
13     if shape not in area_calculators:
14         raise ValueError(f"Unsupported shape: {shape}. Supported shapes: {list(area_calculators.keys())}")
15
16     return area_calculators[shape]()
17
18 class ShapeCalculator:
19
20     @staticmethod
21     def rectangle(width: float, height: float) -> float:
22         return width * height
23
24     @staticmethod
25     def square(side: float) -> float:
26         return side * side
27
```

```

28     @staticmethod
29     def circle(radius: float) -> float:
30         return PI * radius * radius
31
32     @classmethod
33     def calculate(cls, shape: str, x: float, y: float = 0) -> float:
34         method_map = {
35             "rectangle": cls.rectangle,
36             "square": cls.square,
37             "circle": cls.circle
38         }
39
40         if shape not in method_map:
41             raise ValueError(f"Unsupported shape: {shape}")
42
43         if shape == "circle" or shape == "square":
44             return method_map[shape](x)
45         else:
46             return method_map[shape](x, y)
47
48 if __name__ == "__main__":
49     print("Function-based approach:")
50     print(f"Rectangle area: {calculate_area('rectangle', 5, 3)}")
51     print(f"Square area: {calculate_area('square', 4)}")
52     print(f"Circle area: {calculate_area('circle', 2)}")
53
54     print("\nClass-based approach:")
55     print(f"Rectangle area: {(class) ShapeCalculator.calculate('rectangle', 5, 3)}")
56     print(f"Square area: {ShapeCalculator.calculate('square', 4)}")
57     print(f"Circle area: {ShapeCalculator.calculate('circle', 2)}")
58

```

Ctrl+L to chat, Ctrl+K to generate

Output :

```

PS C:\Users\Summya Mehveen\OneDrive\Desktop\AI associated coding> c;; cd 'c:\Users\Summya Mehveen\OneDrive\Desktop\AI associated coding'; & 'c:\Users\Summya Mehveen\AppData\Local\Programs\Python\Python313\python.exe' 'c:\Users\Summya Mehveen\.cursor\extensions\ms-python.debugpy-2025.10.0-win32-x64\bundled\libs\debugpy\launcher' '57838' '--'
'c:\Users\Summya Mehveen\OneDrive\Desktop\AI associated coding\13.3t1.py'
Function-based approach:
Rectangle area: 15
Square area: 16
Circle area: 12.566370614359172

Class-based approach:
Function-based approach:
Rectangle area: 15
Square area: 16
Circle area: 12.566370614359172

Class-based approach:

Class-based approach:
Class-based approach:
Rectangle area: 15
Square area: 16
Circle area: 12.566370614359172

```

Observation :

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

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

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!

TASK 2 : Error Handling in Legacy Code

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)

Code Generated :

```
13.3t2.py > ...
1   import logging
2   from pathlib import Path
3   from typing import Optional
4
5   logging.basicConfig(level=logging.INFO)
6   logger = logging.getLogger(__name__)
7
8   def read_file(filename: str, encoding: str = 'utf-8') -> Optional[str]:
9
10      if not filename:
11          logger.error("Filename cannot be empty")
12          return None
13
14      file_path = Path(filename)
15
16      try:
17          if not file_path.exists():
18              logger.error(f"File not found: {filename}")
19              return None
20
21          with open(file_path, 'r', encoding=encoding) as f:
22              data = f.read()
23
24              logger.info(f"Successfully read {len(data)} characters from {filename}")
25              return data
26
27      except PermissionError:
28          logger.error(f"Permission denied: {filename}")
29          return None
30      except UnicodeDecodeError:
31          logger.error(f"Encoding error: {filename}")
32          return None
33      except Exception as e:
34          logger.error(f"Error reading {filename}: {e}")
35          return None
36
37  if __name__ == "__main__":
38      test_files = ["13.3t1.py", "nonexistent.txt", ""]
39
40      for filename in test_files:
41          print(f"\nTesting: {filename}")
42          content = read_file(filename)
43          if content:
44              print(f"Success: {len(content)} characters")
45          else:
46              print("Failed to read file")
```

Output :

```
PS C:\Users\Summya Mehveen\OneDrive\Desktop\AI associated coding> c:; cd 'c:\Users\Summya Mehveen\OneDrive\Desktop\AI associated coding'; & 'c:\Users\Summya Mehveen\AppData\Local\Programs\Python\Python313\python.exe' 'c:\Users\Summya Mehveen'.cursor\extensions\ms-python.debugpy-2025.10.0-win32-x64\bundled\libs\debugpy\launcher' '57166' '--' 'c:\Users\Summya Mehveen\OneDrive\Desktop\AI associated coding\13.3t2.py'

Testing: 13.3t1.py
INFO:__main__:Successfully read 1800 characters from 13.3t1.py
Success: 1800 characters

Testing: nonexistent.txt
ERROR:__main__:File not found: nonexistent.txt
Failed to read file

Testing:
ERROR:__main__:Filename cannot be empty
Failed to read file
PS C:\Users\Summya Mehveen\OneDrive\Desktop\AI associated coding>
```

Observation :

Code Works

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

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

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!

TASK 3 : Complex Refactoring

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
```

Code Generated :

```
13.3t3.py > ...
1   from typing import List
2   from dataclasses import dataclass
3
4   @dataclass
5   class Student:
6       name: str
7       age: int
8       marks: List[float]
9
10  def __post_init__(self):
11      if not self.name:
12          raise ValueError("Name cannot be empty")
13      if self.age <= 0:
14          raise ValueError("Age must be positive")
15      if not self.marks or any(mark < 0 or mark > 100 for mark in self.marks):
16          raise ValueError("Marks must be between 0 and 100")
17
18      @property
19      def total_marks(self) -> float:
20          return sum(self.marks)
21
22      @property
23      def average_marks(self) -> float:
24          return sum(self.marks) / len(self.marks)
25
26      @property
27      def grade(self) -> str:
28          avg = self.average_marks
29          if avg >= 90: return "A"
30          elif avg >= 80: return "B"
31          elif avg >= 70: return "C"
32          elif avg >= 60: return "D"
33          else: return "F"
34
35      def display_details(self) -> None:
36          print(f"Name: {self.name}, Age: {self.age}")
37          print(f"Marks: {self.marks}")
38          print(f"Total: {self.total_marks}, Average: {self.average_marks:.1f}, Grade: {self.grade}")
39
40  if __name__ == "__main__":
41      try:
42          student = Student("Alice", 20, [85, 92, 78, 88])
43          student.display_details()
44
45          print(f"\nString: {student}")
46
47      except ValueError as e:
48          print(f"Error: {e}")
49
50      try:
51          invalid = Student("", 20, [85, 90])
52      except ValueError as e:
53          print(f"Validation error: {e}")
54
```

Ctrl+L to chat, Ctrl+K to generate

Output :

```
Name: Alice, Age: 20
Marks: [85, 92, 78, 88]
Total: 343, Average: 85.8, Grade: B

String: Student(name='Alice', age=20, marks=[85, 92, 78, 88])
Validation error: Name cannot be empty
```

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

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!

TASK 4 : Inefficient Loop Refactoring

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

Code Generated :

```
13.3t4.py > ...
1     nums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
2     squares = []
3     for i in nums:
4         squares.append(i * i)
5
6     print("Original approach:", squares)
7     print("\n" + "="*50)
8
9     nums = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
10    squares = [i * i for i in nums]
11    print("List comprehension:", squares)
12
13    squares_map = list(map(lambda x: x * x, nums))
14    print("Map function:", squares_map)
15
16    squares_range = [i * i for i in range(1, 11)]
17    print("Range generation:", squares_range)
18
19    try:
20        import numpy as np
21        nums_array = np.array(nums)
22        squares_numpy = (nums_array ** 2).tolist()
23        print("NumPy approach:", squares_numpy)
24    except ImportError:
25        print("NumPy not available")
26
27    squares_unpack = [*map(lambda x: x*x, nums)]
28    print("Unpacking approach:", squares_unpack)
29
30    print("\n" + "="*50)
31    print("BENEFITS:")
32    print("✓ List comprehension: More readable and faster")
33    print("✓ Map function: Functional programming style")
34    print("✓ Range generation: No need for separate list")
35    print("✓ NumPy: Fastest for large datasets")
36    print("✓ All approaches: More concise than original loop")
```

Output :

```
Original approach: [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
=====
List comprehension: [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
Map function: [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
Range generation: [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
NumPy approach: [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
Unpacking approach: [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

=====
BENEFITS:
✓ List comprehension: More readable and faster
✓ Map function: Functional programming style
✓ Range generation: No need for separate list
✓ NumPy: Fastest for large datasets
✓ All approaches: More concise than original loop
```

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

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

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!