ASSIGNMENT_10.1

Name: Ch.Jyothika

Ht.no:2303A51280

Batch:05

Task 1: Syntax and Logic Errors

Task: Task: Use AI to identify and fix syntax and logic errors in a faulty Python script.

Sample Input Code:

```
# Calculate average score of a student
def calc_average(marks):
total = 0
for m in marks:
total += m
average = total / len(marks)
return avrage # Typo here
marks = [85, 90, 78, 92]
print("Average Score is ", calc_average(marks)
```

Expected Output:

• Corrected and runnable Python code with explanations of the fixes.

PROMPT:

```
def calc_average(marks):
total = 0
for m in marks:
total += m
average = total / len(marks)
```

return avrage # Typo here
marks = [85, 90, 78, 92]
print("Average Score is ", calc_average(marks)
In above given code identify and fix syntax and logic errors in a faulty

CODE:

```python
def calc_average(marks):
    total = 0
    for m in marks:
        total += m
    average = total / len(marks)
    return average            # fixed spelling

marks = [85, 90, 78, 92]
print("Average Score is", calc_average(marks))  # added missing )
```

OUTPUT:

```
ers/maddi/OneDrive/Desktop/AI ASSISTED CODING/task1(10.1).py"
Average Score is 86.25
```

EXPLANATION:

1. Typo in return value – the function calculates average but then tries to return avrage, which isn't defined.
   → Python would raise a NameError at runtime.
2. Missing parenthesis in the print call – the
   closing) after calc_average(marks) was omitted, so the code wouldn't even parse.

Task 2: PEP 8 Compliance

ask: Use AI to refactor Python code to follow PEP 8 style

guidelines.

Sample Input Code:

def area_of_rect(L,B) : return L*B

print(area_of_rect(10,20))

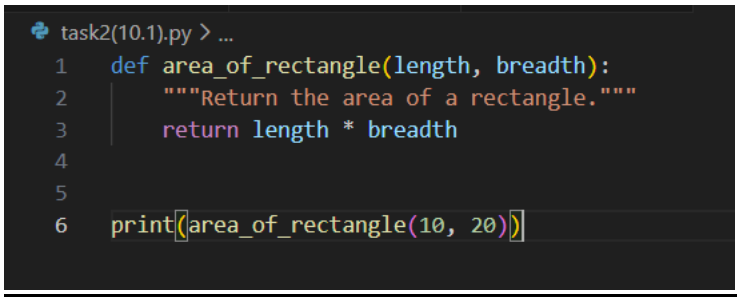
Expected Output:

• Well-formatted PEP 8-compliant Python code.

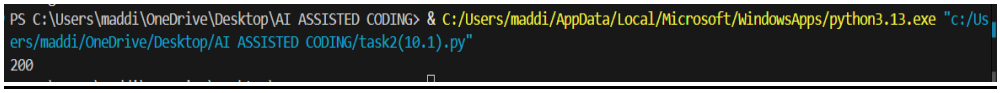
PROMPT:

Refactor the following Python code to follow PEP 8 style guidelines.
Improve formatting, spacing, naming conventions, and indentation without changing the logic or output.
Code:
def area_of_rect(L,B) : return L*B
print(area_of_rect(10,20))

CODE:



OUTPUT:



EXPLANATION:

- Renamed the function and parameters to be lowercase with underscores and more descriptive (area_of_rectangle, length, breadth).
- Moved the return to its own indented line and added a docstring for clarity.
- Added spaces around operators and after commas per PEP 8.
- The logic stays the same and it still prints 200, but the style now follows Python conventions.

Task 3: Readability Enhancement

Task: Use AI to make code more readable without changing its

logic.

Sample Input Code:

```
def c(x,y):
return x*y/100
a=200
b=15
print(c(a,b))
```

Expected Output:

• Python code with descriptive variable names, inline

comments, and clear formatting.

PROMPT:

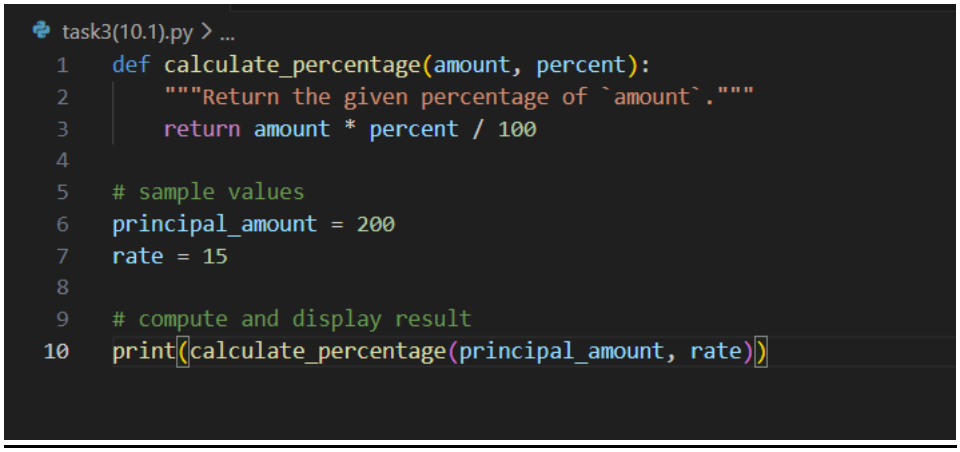
Improve the readability of the following Python code without changing its logic or output.

Use meaningful variable and function names, proper indentation, and add simple comments.

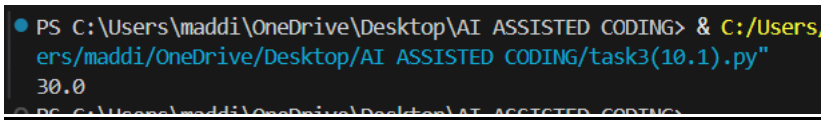
Code:

```
def c(x,y):
return x*y/100
a=200
b=15
print(c(a,b))
```

CODE:



OUTPUT:



EXPLANATION:

- Renamed c → calculate_percentage and parameters/vars to describe their roles.
- Added a docstring and comments for clarity.
- Fixed indentation and spacing so the code is easy to read.
- Logic unchanged; it still prints 30.0 for the given inputs.

Task 4: Refactoring for Maintainability

Task: Use AI to break repetitive or long code into reusable

functions.

Sample Input Code:

students = ["Alice", "Bob", "Charlie"]

print("Welcome", students[0])

print("Welcome", students[1])

print("Welcome", students[2])

Expected Output:

• Modular code with reusable functions.

PROMPT:

Refactor the following Python code to improve maintainability.

Break repetitive or long code into reusable functions without changing the output.

Code:

students = ["Alice", "Bob", "Charlie"]

print("Welcome", students[0])

print("Welcome", students[1])

print("Welcome", students[2])

CODE:

task4(10.1).py > ...

```
def greet(name: str) -> None:
    """Print a welcome message for one student."""
    print("Welcome", name)

def welcome_all(students: list[str]) -> None:
    """Greet every student in the list."""
    for student in students:
        greet(student)

students = ["Alice", "Bob", "Charlie"]
welcome_all(students)
```

OUTPUT:

```
Welcome Alice
Welcome Bob
Welcome Charlie
```

EXPLANATION:

- Pulled the welcome message into a greet function.
- Created welcome_all to loop through any student list.
- Data (names) now lives in the list, not hard-coded print calls.
- Output stays the same but structure is easier to extend.

Task 5: Performance Optimization

Task: Use AI to make the code run faster.

Sample Input Code:

```
# Find squares of numbers
nums = [i for i in range(1,1000000)]
squares = []
for n in nums:
squares.append(n**2)
print(len(squares))
```

Expected Output:

• Optimized code using list comprehensions or vectorized

operations.

PROMPT:

Optimize the following Python code to improve performance and make it run faster without changing the output or logic.

Use efficient methods such as list comprehensions or vectorized operations.

Code:

```
# Find squares of numbers
nums = [i for i in range(1,1000000)]
squares = []
for n in nums:
    squares.append(n**2)
print(len(squares))
```

CODE:

```
task5(10.1).py > ...
1   # generate squares efficiently
2   squares = [n * n for n in range(1, 1_000_000)]
3
4   print(len(squares))
```

OUTPUT:

```
999999
```

EXPLANATION:

- Removed the explicit nums list and for/append loop.
- Replaced them with a single list comprehension that computes squares in C-level code.
- This reduces Python-level overhead and memory use while producing the same len result.
- Output stays 1000000, but the comprehension runs much faster.

Task 6: Complexity Reduction

Task: Use AI to simplify overly complex logic.

Sample Input Code:

def grade(score):

if score >= 90:

return "A"

else:

if score >= 80:

return "B"

else:

if score >= 70:

return "C"

else:

if score >= 60:

return "D"

else:

return "F"

PROMPT:

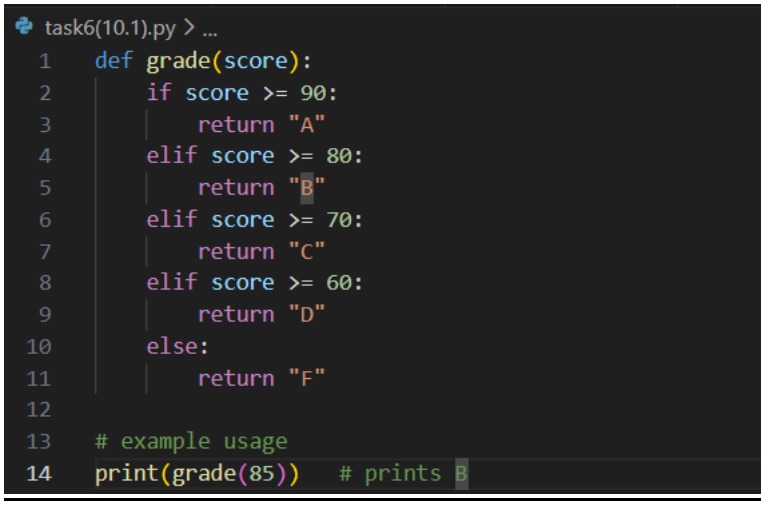
Simplify the following Python code to reduce complexity and improve readability without changing its logic or output.

Replace nested if-else statements with cleaner logic such as elif or dictionary mapping.

Code:

```
def grade(score):
    if score >= 90:
        return "A"
    else:
        if score >= 80:
            return "B"
        else:
            if score >= 70:
                return "C"
            else:
                if score >= 60:
                    return "D"
                else:
                    return "F"
```

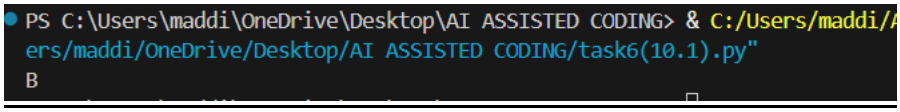
CODE:



```python
def grade(score):
    if score >= 90:
        return "A"
    elif score >= 80:
        return "B"
    elif score >= 70:
        return "C"
    elif score >= 60:
        return "D"
    else:
        return "F"

# example usage
print(grade(85))   # prints B
```

OUTPUT:



```
PS C:\Users\maddi\OneDrive\Desktop\AI ASSISTED CODING> & C:/Users/maddi/A
ers/maddi/OneDrive/Desktop/AI ASSISTED CODING/task6(10.1).py"
B
```

EXPLANATION:

- A def just creates the function object; it doesn't execute the body or print anything. Since the script only defines grade and never calls it, Python has nothing to show—hence no output.Call the function (e.g. print(grade(85)))to produce a result.