

Assignment – 2.5

Name: N.Hasini Rao

Roll Number: 2303A510G3

Batch – 03

AI Assisted Coding

16-01-2026

Task 1: Refactoring Odd/Even Logic (List Version)

❖ Scenario:

You are improving legacy code.

❖ Task:

Write a program to calculate the sum of odd and even numbers in a list, then refactor it using AI.

❖ Expected Output:

❖ Original and improved code

Task 1: Refactoring Odd/Even Logic

The task involves refactoring a legacy code snippet to calculate the sum of odd and even numbers in a list. The original code uses a while loop to iterate through the list, checking each element's index to determine if it's odd or even. The improved code uses list comprehensions and the built-in `sum` function.

Original Code (Legacy Style):

```

# Task 1: Refactoring Odd/Even Logic (List Version)
# Scenario:
# You are improving Legacy code.
# Task:
# Write a program to calculate the sum of odd and even numbers in a list,
# then refactor it using AI.
# Expected Output:
# Original and Improved code

# Original code (Legacy Style)
def calculate_sums_original(numbers):
    odd_sum = 0
    even_sum = 0
    i = 0
    while i < len(numbers):
        if numbers[i] % 2 == 0:
            even_sum = even_sum + numbers[i]
        else:
            odd_sum = odd_sum + numbers[i]
        i = i + 1
    return odd_sum, even_sum

# Test the original code
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
odd, even = calculate_sums_original(numbers)
print("Original code:")
print(f"Sum of odd numbers: {odd}")
print(f"Sum of even numbers: {even}")

```

Improved/Refactored Code:

```

# Improved Code (Refactored)
def calculate_sums_improved(numbers):
    """
    Calculate the sum of odd and even numbers in a list.
    Args:
        numbers: List of integers
    Returns:
        tuple: (sum_of_odd_numbers, sum_of_even_numbers)
    """
    odd_sum = sum(num for num in numbers if num % 2 != 0)
    even_sum = sum(num for num in numbers if num % 2 == 0)
    return odd_sum, even_sum

# Alternative improved version using filter
def calculate_sums_alternative(numbers):
    """Alternative refactored version using filter."""
    odd_sum = sum(filter(lambda x: x % 2 != 0, numbers))
    even_sum = sum(filter(lambda x: x % 2 == 0, numbers))
    return odd_sum, even_sum

# Test the improved code
numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
odd, even = calculate_sums_improved(numbers)
print("Improved Code:")
print(f"Sum of odd numbers: {odd}")
print(f"Sum of even numbers: {even}")

# Test alternative version
odd2, even2 = calculate_sums_alternative(numbers)
print("Alternative Improved Code:")
print(f"Sum of odd numbers: {odd2}")
print(f"Sum of even numbers: {even2}")

```

Task 2: Area Calculation Explanation ♦

Scenario:

You are onboarding a junior developer.

❖ Task:

Ask Gemini to explain a function that calculates the area of different shapes.

❖ Expected Output:

➤ Code

➤ Explanation

ask 2: Area Calculation Explanation

❖ Scenario:

You are onboarding a junior developer.

❖ Task:

Ask Gemini to explain a function that calculates the area of different shapes.

❖ Expected Output:

➤ Code

➤ Explanation

The screenshot shows the Gemini Code Assist interface within the Google Cloud Platform. The main area displays two tabs: 'lab1.py' and 'AIAC2.5.py'. The 'AIAC2.5.py' tab contains the following Python code:

```
def calculate_sums(numbers):
    even_sum = 0
    odd_sum = 0

    for num in numbers:
        # Check if the number is divisible by 2
        if num % 2 == 0:
            even_sum += num
        else:
            odd_sum += num

    return even_sum, odd_sum

# Example usage:
my_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
evens, odds = calculate_sums(my_list)

print(f"List: {my_list}")
print(f"Sum of even numbers: {evens}")
print(f"Sum of odd numbers: {odds}")

my_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
even_sum = 0
odd_sum = 0
for num in my_list:
    if num % 2 == 0:
        even_sum += num
    else:
        odd_sum += num
```

Below the code editor, there are tabs for 'PROBLEMS', 'OUTPUT', and 'Code'. The 'OUTPUT' tab is selected. A tooltip at the bottom right says 'Describe what to build next'.

```

File Edit Selection View Go Run Terminal Help ← →
GMINI CODE ASSIST CH0
Preview Channel
task2.py x
Assessment2 > task2.py > ...
import math
...
def calculate_area(shape, **kwargs):
    """
    Calculates the area of different shapes.
    Args:
        shape (str): The name of the shape (e.g., 'rectangle', 'circle', 'triangle').
        **kwargs: Additional keyword arguments required for the calculation.
            For 'rectangle', expects 'length' and 'width'.
            For 'circle', expects 'radius'.
            For 'triangle', expects 'base' and 'height'.
    Returns:
        float: The calculated area of the shape.
        str: An error message if the shape is not supported or dimensions are missing.
    """
    if shape == "rectangle":
        if "length" in kwargs and "width" in kwargs:
            return kwargs["length"] * kwargs["width"]
        else:
            return "Error: Missing length or width for rectangle."
    elif shape == "circle":
        if "radius" in kwargs:
            return math.pi * (kwargs["radius"] ** 2)
        else:
            return "Error: Missing radius for circle."
    elif shape == "triangle":
        if "base" in kwargs and "height" in kwargs:
            return 0.5 * kwargs["base"] * kwargs["height"]
        else:
            return "Error: Missing base or height for triangle."
    else:
        return "Error: Unsupported shape."
...
# Example Usage:
rectangle_area = calculate_area("rectangle", length=10, width=5)
print("The area of the rectangle is: ", rectangle_area)
circle_area = calculate_area("circle", radius=7)
print("The area of the circle is: ", circle_area)
triangle_area = calculate_area("triangle", base=8, height=4)
print("The area of the triangle is: ", triangle_area)
# Example of an error
square_area = calculate_area("square", side=5)
print("The area of the square is: ", square_area)

```

The function uses `if`, `elif` (else if), and `else` statements to determine which shape has been requested and then performs the correct calculation.

- `if shape == "rectangle":`
 - If the shape is a "rectangle", the code checks if the necessary dimensions, `length` and `width`, are provided in the `kwargs` dictionary.
 - If they are present, it calculates the area by multiplying `length` and `width`.
 - If they are missing, it returns an error message.
- `elif shape == "circle":`
 - If the shape is a "circle", it checks for a `radius`.
 - If found, it calculates the area using the formula πr^2 , importing the `math` module to get an accurate value of `math.pi`.
 - If the `radius` is missing, it returns an error.
- `elif shape == "triangle":`
 - If the shape is a "triangle", it checks for a `base` and `height`.
 - If they are present, it calculates the area using the formula $0.5 \times \text{base} \times \text{height}$.
 - If they are missing, it returns an error.
- `else:`
 - If the provided shape name does not match any of the supported shapes ("rectangle", "circle", "triangle"), the function returns "Unsupported shape" error.

Example Usage:

The lines at the end of the script demonstrate how to use the function and print the results for a rectangle, a circle, a triangle, and an unsupported shape to show how the error handling works.

Explanation

This Python script defines a single function `calculate_area` that is designed to be a flexible tool for finding the area of several different geometric shapes.

Function Definition:

```
def calculate_area(shape, **kwargs):
```

This defines a function named `calculate_area` that accepts two arguments: `shape`: A string that tells the function which shape's area to calculate (e.g., "rectangle").

`**kwargs`: This is a special Python feature that allows the function to accept any number of keyword arguments (like `length=10` or `radius=5`). These arguments are collected into a dictionary named `kwargs`. This makes the function adaptable, as different shapes require different parameters for area calculation.

Conditional Logic:

The function uses `if`, `elif` (else if), and `else` statements to determine which shape has been requested and then performs the correct calculation. `if shape == "rectangle":`

If the shape is a "rectangle", the code checks if the necessary dimensions, `length` and `width`, are provided in the `kwargs` dictionary.

If they are present, it calculates the area by multiplying `length` and `width`.

If they are missing, it returns an error message.

elif shape == "circle":

If the shape is a "circle", it checks for a radius.

If found, it calculates the area using the formula $\pi * r^2$, importing the math module to get an accurate value of math.pi. If the radius is missing, it returns an error.

elif shape == "triangle":

If the shape is a "triangle", it checks for a base and height.

If they are present, it calculates the area using the formula $0.5 * \text{base} * \text{height}$.

If they are missing, it returns an error.

else:

If the provided shape name does not match any of the supported shapes ("rectangle", "circle", "triangle"), the function returns an "Unsupported shape" error.

Example Usage:

The lines at the end of the script demonstrate how to use the function and print the results for a rectangle, a circle, a triangle, and an unsupported shape to show how the error handling works.

Task 3: Prompt Sensitivity Experiment

❖ **Scenario:**

You are testing how AI responds to different prompts.

❖ **Task:**

Use Cursor AI with different prompts for the same problem and observe code changes.

❖ **Expected Output:**

➤ **Prompt list**

➤ **Code variations**

A screenshot of the Visual Studio Code (VS Code) interface. The top menu bar includes File, Edit, Selection, View, Go, Run, Terminal, Help, and an Upgrade to Pro button. The title bar shows "task3.py - AI-A-coding-v2 - Cursor". The left sidebar displays a file tree with files like task1.py, task2.py, task3.py, task4.py, task5_iterative.py, and task5_recursive.py. The main editor area contains the following Python code:

```
# Write a Python function to reverse an integer.
def reverse(x: int) -> int:
    INT_MAX = 2**31 - 1
    INT_MIN = -2**31
    sign = -1 if x < 0 else 1
    x = abs(x)
    rev = 0
    while x != 0:
        rev = rev * 10 + x % 10
        x //= 10
    rev *= sign
    if rev < INT_MIN or rev > INT_MAX:
        return 0
    return rev
```

The status bar at the bottom shows the path "E:\sem6\AI-A-coding-v2>" and the file "task1.py lines 1-9". The terminal tab is active, displaying the following command-line session:

```
PS E:\sem6\AI-A-coding-v2> & 'c:\Python314\python.exe' 'c:\Users\sprus\cursor\extensions\ms-python.on.debug-2025.18.0-win32-x64\bundle\libs\debug\launcher' '59879' '--' 'e:\sem6\AI-A-coding-v2\Assessment2.5\task3.py'
PS E:\sem6\AI-A-coding-v2> 9547
# E:\sem6\AI-A-coding-v2> & 'c:\Python314\python.exe' 'c:\Users\sprus\cursor\extensions\ms-python.on.debug-2025.18.0-win32-x64\bundle\libs\debug\launcher' '62775' '--' 'e:\sem6\AI-A-coding-v2\Assessment2.5\task3.py'
# PS E:\sem6\AI-A-coding-v2> & 'c:\Python314\python.exe' 'c:\Users\sprus\cursor\extensions\ms-python.on.debug-2025.18.0-win32-x64\bundle\libs\debug\launcher' '62814' '--' 'e:\sem6\AI-A-coding-v2\Assessment2.5\task3.py'
# PS E:\sem6\AI-A-coding-v2> 8520
# PS E:\sem6\AI-A-coding-v2>
```

task3.2.py - AI-A-coding-v2 - Cursor

```

File Edit Selection View Go Run Terminal Help
task1.py u task2.py u task3.py u task32.py u
Assessment2.5 3 task32.py ...
1 def reverse_integer(x):
2     """
3         Reverse an integer
4         Args:
5             x: An integer to reverse
6         Returns:
7             The reversed integer, or 0 if the reversed integer overflows 32-bit signed integer range
8             (i.e., outside [-2^31, 2^31 - 1] = [-2147483648, 2147483647])
9     """
10    # Handle negative numbers
11    is_negative = x < 0
12    num = abs(x)
13
14    # Reverse the digits
15    reversed_num = 0
16    while num > 0:
17        reversed_num = reversed_num * 10 + num % 10
18        num /= 10
19
20    # Apply sign
21    if is_negative:
22        reversed_num = -reversed_num
23
24    # Check for 32-bit signed integer overflow
25    INT_MIN = -2^31 # -2147483648
26    INT_MAX = 2^31 - 1 # 2147483647
27
28    if reversed_num < INT_MIN or reversed_num > INT_MAX:
29        return 0
30
31    return reversed_num
32
33
34
35
36
37
38 # Test cases
39 if __name__ == "__main__":
40     # Test case 1: Positive number
41     print("reverse_integer(123) = (reverse_integer(123))" # Expected: 321
42
43     # Test case 2: Negative number
44     print("reverse_integer(-123) = (reverse_integer(-123))" # Expected: -321
45
46     # Test case 3: Number ending with zero
47     print("reverse_integer(120) = (reverse_integer(120))" # Expected: 21
48
49     # Test case 4: Single digit
50     print("reverse_integer(5) = (reverse_integer(5))" # Expected: 5
51
52     # Test case 5: Zero
53     print("reverse_integer(0) = (reverse_integer(0))" # Expected: 0
54
55     # Test case 6: Overflow case (reversed number exceeds INT_MAX)
56     print("reverse_integer(1534236489) = (reverse_integer(1534236489))" # Expected: 0 (overflow)
57
58     # Test case 7: Overflow case (reversed number exceeds INT_MIN)
59     print("reverse_integer(-2147483648) = (reverse_integer(-2147483648))" # Expected: 0 (overflow)
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
99

```

Upgrade to Pro ... ↗

Integer reversal with overflow handling

Writing a Python function to reverse an integer with 32-bit signed integer overflow handling.

Checking the current file for context.

Read task32.py

Writing a function to reverse an integer with 32-bit signed integer overflow handling.

Edit Illegal value for maxlen

Reading the file szgut

Read task32.py

Improving overflow handling checking during reversal to prevent intermediate values from exceeding 32-bit bounds

Edit Illegal value for maxlen

Reversing the file with improved overflow handling

Edit Illegal value for maxlen

Checking the file and trying a different approach:

Read task32.py

Improving overflow handling by checking during reversal instead of after

Edit Illegal value for maxlen

Testing the current implementation:

Run command: cd python

cd "e:\sem\AI-A-coding-v2\Assessment2.5" & python task3.2.py

Use AllowSet + Run

2 of 2 Undo Ctrl+N Save Ctrl+S

Problems Output Debug Console Terminal Ports

PowerShell Extension (2025.4.0)

Copyright (c) Microsoft Corporation.

<https://www.microsoft.com/vscode-powershell>

Type "help" to get help.

PS E:\sem\AI-A-coding-v2>

Ctrl+K to generate command

task3.2.py - AI-A-coding-v2 - Cursor

```

File Edit Selection View Go Run Terminal Help
task1.py u task2.py u task3.py u task32.py u
Assessment2.5 3 task32.py ...
1 # Write a Python function to reverse an integer with 32-bit signed integer overflow handling.
2
3 def reverse_integer(x):
4     """
5         Reverse an integer with 32-bit signed integer overflow handling.
6         Args:
7             x: An integer to reverse
8         Returns:
9             The reversed integer, or 0 if the reversed integer overflows 32-bit signed integer range
10            (i.e., outside [-2^31, 2^31 - 1] = [-2147483648, 2147483647])
11    """
12    # Handle negative numbers
13    is_negative = x < 0
14    num = abs(x)
15
16    # Reverse the digits
17    reversed_num = 0
18    while num > 0:
19        reversed_num = reversed_num * 10 + num % 10
20        num /= 10
21
22    # Apply sign
23    if is_negative:
24        reversed_num = -reversed_num
25
26    # Check for 32-bit signed integer overflow
27    INT_MIN = -2^31 # -2147483648
28    INT_MAX = 2^31 - 1 # 2147483647
29
30    if reversed_num < INT_MIN or reversed_num > INT_MAX:
31        return 0
32
33    return reversed_num
34
35
36
37
38 # Test cases
39 if __name__ == "__main__":
40     # Test case 1: Positive number
41     print("reverse_integer(123) = (reverse_integer(123))" # Expected: 321
42
43     # Test case 2: Negative number
44     print("reverse_integer(-123) = (reverse_integer(-123))" # Expected: -321
45
46     # Test case 3: Number ending with zero
47     print("reverse_integer(120) = (reverse_integer(120))" # Expected: 21
48
49     # Test case 4: Single digit
50     print("reverse_integer(5) = (reverse_integer(5))" # Expected: 5
51
52     # Test case 5: Zero
53     print("reverse_integer(0) = (reverse_integer(0))" # Expected: 0
54
55     # Test case 6: Overflow case (reversed number exceeds INT_MAX)
56     print("reverse_integer(1534236489) = (reverse_integer(1534236489))" # Expected: 0 (overflow)
57
58     # Test case 7: Overflow case (reversed number exceeds INT_MIN)
59     print("reverse_integer(-2147483648) = (reverse_integer(-2147483648))" # Expected: 0 (overflow)
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
79

```

Upgrade to Pro ... ↗

Integer reversal with overflow handling

Writing a Python function to reverse an integer with 32-bit signed integer overflow handling.

Read task32.py

Improving overflow handling by checking during reversal instead of after

Edit Illegal value for maxlen

Testing the current implementation:

Ran command: cd python

cd "e:\sem\AI-A-coding-v2\Assessment2.5" & python task3.2.py

At C:\Users\ps\appdata\local\temp\py-script-#1c81b-f7de-4912-aed7-7d7aaec82116.ps:17 char:43

+ cd "e:\sem\AI-A-coding-v2\Assessment2.5" & python task3.2.py

+ ~~~~~

The token 'cd' is not a valid statement separator in this version.

+ cd "e:\sem\AI-A-coding-v2\Assessment2.5" & python task3.2.py

+ ~~~~~

+ The token 'cd' is not a valid statement separator in this version.

+ cd "e:\sem\AI-A-coding-v2\Assessment2.5" & python task3.2.py

+ ~~~~~

Failing the PowerShell command:

Ran command: cd python

cd "e:\sem\AI-A-coding-v2\Assessment2.5"; python task3.2.py

reverse_integer(123) = 321

reverse_integer(-123) = -321

reverse_integer(120) = 21

reverse_integer(5) = 5

reverse_integer(0) = 0

reverse_integer(1534236489) = 0

reverse_integer(-2147483648) = 0

Use AllowSet +

The function is implemented and working. It:

1. Reverses an integer by extracting digits and building the reversed number
2. Handles negative numbers by preserving the sign
3. Checks for 32-bit signed integer overflow — returns 0 if the reversed number is outside [-2^31, 2^31 - 1]

Test cases:

- 331 → 133
- -331 → -133
- 338 → 833 (leading zeros removed)
- 5 → 5
- 0 → 0
- 1534236489 → 946483243153 (overflow)
- -2147483648 → -8 (overflow)

The function correctly handles edge cases and overflow. The implementation checks bounds after reversal, which works since Python integers are unbounded and we only need to validate the final result against 32-bit limits.

Would you like me to optimize it to check for overflow during reversal, or is the current implementation sufficient?

1 file

Plan, ⌂ for context, / for commands

Ctrl+K to generate command

Cursor Tab

Un 56 Col 103 Spaces: 4 UTF-8 CRLF Python Python 3.14 (64-bit) Go Live

The screenshot displays two instances of the Visual Studio Code (VS Code) interface. Both instances show the same Python file, `task3.py`, which is intended to reverse an integer while performing strict overflow checks. The code includes various comments and logic to handle edge cases and overflow conditions.

Left Editor Content:

```

1 # Write an optimized Python solution for reversing an integer with strict overflow checks for coding interviews.
2
3 def reverse_integer(x):
4     """
5         Optimized solution to reverse an integer with strict 32-bit signed integer overflow handling.
6
7             Time Complexity: O(log(n)) - where n is the input number
8             Space Complexity: O(1) - constant extra space
9
10        Args:
11            x: An Integer to reverse
12
13        Returns:
14            The reversed integer, or 0 if the reversed integer overflows 32-bit signed integer range
15            (i.e., outside [-2^31, 2^31 - 1] = [-2147483648, 2147483647])
16
17        # Strictly aligned integer bounds
18        INT_MIN = -2**31 # -2147483648
19        INT_MAX = 2**31 - 1 # 2147483647
20
21        result = 0
22
23        while x != 0:
24
25            # Extract the last digit
26            digit = x % 10 > 0 else -(x % 10) # Handle negative modulo correctly
27            x = x // 10 if x > 0 else -(x // 10) # Handle negative division correctly
28
29            # Strict overflow check BEFORE multiplication
30            # Check if result * 10 would overflow
31            # For positive overflow: result * 10 + digit > INT_MAX
32            # For negative overflow: result * 10 + digit < INT_MIN
33            if result > INT_MAX // 10 or (result == INT_MAX // 10 and digit > INT_MAX % 10):
34                return 0
35
36            # Safe to perform the operation
37            result = result * 10 + digit
38
39        return result
40
41
42    # Alternative optimized version (more Pythonic and cleaner)
43    # ...
44    # Alternative optimized solution - cleaner approach handling sign separately.
45    # ...
46
47    # Alternative optimized version (more Pythonic and cleaner)
48    # ...
49    # Alternative optimized solution - cleaner approach handling sign separately.
50    # ...
51
52    Time Complexity: O(log(n))
53
54    Problems Output Debug Console Terminal Help
55
56    Assessment25: task3.py ...
57
58    PowerShell Extension v2025.4.0
59    Copyright (c) Microsoft Corporation.
60
61    https://aka.ms/vscode-powershell
62    Type 'help' to get help.
63
64    PS E:\vscode\AI-A-coding-v2>

```

Right Editor Content:

The right editor shows the same code with several annotations and a terminal window below it.

- Annotations:**
 - # Write an optimized Python solution for reversing an integer with strict overflow checks for coding interviews.
 - Checking the current file, then implementing an optimized solution with strict overflow checks.
 - Writing an optimized solution with strict overflow checks during reversal.
 - Testing the optimized solution:
- Terminal:**

```

cd "e:\vscode\AI-A-coding-v2\Assessment2.5"; python task3.py
Use Allowlist: ✓ Run ⌂

```

Task 4: Tool Comparison Reflection

❖ Scenario:

You must recommend an AI coding tool.

❖ **Task:**

Based on your work in this topic, compare Gemini, Copilot, and Cursor AI for usability and code quality.

❖ **Expected Output:**

Short written reflection

Based on my experience using Gemini, GitHub Copilot, and Cursor AI during this topic, I observed clear differences in both usability and code quality.

Gemini is useful for understanding concepts and generating explanations, but it often produces generic code unless very strict constraints are provided. It is better suited for learning and problem understanding rather than competitive or production-level coding.

GitHub Copilot integrates smoothly with IDEs like VS Code and provides fast, context-aware code suggestions. However, its outputs sometimes assume the developer will handle edge cases, so overflow handling and constraints may be missed unless explicitly guided.

Cursor AI provided the best balance of usability and code quality. It allows direct interaction with the codebase, understands existing files, and responds well to detailed prompts. When constraints are clearly mentioned, Cursor AI consistently generated correct, optimized, and readable code, making it ideal for real development and debugging tasks.

Conclusion:

For learning → Gemini

For quick coding assistance → Copilot

For serious development and prompt-based experimentation → Cursor AI