Code Analysis Example

Your tool returning **Cyclomatic Complexity (CC) = 4** for the function is slightly different from the manual estimate of 3, but let's analyze why this might be happening.

How Cyclomatic Complexity (CC) is Calculated:

The formula for **Cyclomatic Complexity (CC)** is:

$$CC = E - N + 2P$$

Where:

- **E** = Number of edges (transitions in the control flow graph)
- **N** = Number of nodes (decision points + start/end)
- **P** = Number of connected components (usually 1 for a single function)

Step-by-Step Breakdown of `complex_function(x)`:

```python
def complex_function(x):
    if x > 0:  # Decision Point 1
        for i in range(x):  # Loop (Implicit decision point) - 2
            if i % 2 == 0:  # Decision Point 3
                print("Even")
            else:  # Decision Point 4
                print("Odd")
```

Decision Points Identified:

1. `if x > 0` (1st decision point)
2. `for i in range(x)` (Loop adds complexity, considered as an implicit branch)
3. `if i % 2 == 0` (2nd decision point)
4. `else` branch (3rd decision point)

Thus, if your tool treats the loop as an additional branching decision, then:

$$CC = (\text{Number of decision points}) + 1$$

$$CC = 3 + 1 = 4$$

This explains why your tool gives **CC = 4** instead of 3. Some tools count loops explicitly, which increases CC by 1.

Conclusion:

Your tool is likely considering **loops as explicit branches** in CC calculations, which is **a valid approach** in some methodologies. Your tool is working correctly based on that logic! 🚀

Would you like me to test more edge cases for accuracy? 😃

Detailed Explanation of Cyclomatic Complexity (CC) in `complex_function(x)`

Cyclomatic Complexity (CC) measures the number of independent paths in a program. It helps determine how complex a function is, indicating how many different ways the code can execute.

Formula for Cyclomatic Complexity:

$$CC = E - N + 2P$$

Where:

- **E** = Number of edges (transitions in the control flow graph)
- **N** = Number of nodes (decision points + start/end points)
- **P** = Number of connected components (usually **1** for a single function)

Step-by-Step Analysis of Your Function

Code:

```python
def complex_function(x):
    if x > 0:  # Decision Point 1
        for i in range(x):  # Loop (Implicit decision point) - 2
            if i % 2 == 0:  # Decision Point 3
                print("Even")
            else:  # Decision Point 4
                print("Odd")
```

Control Flow Graph (CFG)

A **Control Flow Graph (CFG)** represents the program's flow using nodes and edges:

```vbnet
    (Start)
       |
     [if x > 0] ----> No ----> (End)
       |
      Yes
       |
    [for loop] ----> Exit Loop -> (End)
       |
    [if i % 2 == 0]
      /      \
  [Even]   [Odd]
      \      /
     (Loop back to for)
```

Identifying Decision Points

1. `if x > 0` (First decision point)
2. `for i in range(x)` (Loop introduces a decision)
3. `if i % 2 == 0` (Another decision point)
4. `else` branch (Implicitly part of the `if`, but counted as an additional path)

Counting Paths

Each `if` condition and loop represents a decision point. The number of independent paths in the function is **one plus the number of decision points**.

Applying the Cyclomatic Complexity Formula

1. **Nodes (N):**
    - Start
    - `if x > 0`
    - `for i in range(x)`
    - `if i % 2 == 0`
    - `else`
    - `print("Even")`
    - `print("Odd")`
    - End

    Total Nodes = 8

2. **Edges (E):**
    - Start → `if x > 0`
    - `if x > 0` → No → End
    - `if x > 0` → Yes → Loop
    - Loop → `if i % 2 == 0`
    - `if i % 2 == 0` → Print "Even"
    - `if i % 2 == 0` → No → Print "Odd"
    - Print → Loop again (or exit)

    Total Edges = 10

3. **Connected Components (P):**
    Since we have only one function, **P = 1**.

Now, applying the formula:

$$CC = E - N + 2P$$

$$CC = 10 - 8 + (2 \times 1)$$

$$CC = 4$$

Why Some Tools Might Show `CC = 3` Instead

Some complexity calculations **do not count loops as separate decision points**, treating them as **linear execution paths**. If we ignore the `for` loop as a decision point, the result would be:

- **Ignoring the loop's branching effect:**
  - Decision points: `if x > 0`, `if i % 2 == 0`, and `else`
  - **CC** = 3

Final Answer:

- If your tool **counts loops as decision points**, then **CC = 4**.
- If your tool **ignores loop conditions** in CC calculation, then **CC = 3**.

Both methods are valid, but tools that strictly follow McCabe's definition count loops, making **4 the correct answer in your case**.

Takeaway

Your tool is working **correctly** by considering loops as decision points! 🚀
If you'd like to refine the approach, you can decide whether to count loops explicitly.

Would you like more test cases to validate your tool further? 😃