### Written Report

Once you have completed your refactoring, write a brief report addressing the following:

1. Justification for your refactoring decisions.
2. The challenges you would have faced maintaining and testing the original monolithic code.

How you would modify your refactored code to handle a customised tic-tac-toe game (larger than 3x3), and how this implementation would be easier to handle than in the original code.

To modify the refactored Tic-Tac-Toe code for a customized board larger than 3x3, the primary change would be making the board size dynamic rather than fixed. This can be achieved by allowing the GameBoard class to accept a size parameter, ensuring that the board is initialized as an N x N grid instead of being restricted to 3x3. Additionally, the logic for determining a win must be adapted to automatically generate winning conditions based on the board size. Instead of hardcoding row, column, and diagonal checks, the program would dynamically construct these conditions for any given board size, making the implementation scalable. The input validation must also be updated to ensure that player moves remain within the correct range for larger boards. This approach makes the implementation far more flexible and easier to manage compared to the original monolithic version, where modifying the board size would have required rewriting large portions of the game logic. By keeping the code modular, we ensure that expanding to different board sizes requires only minimal adjustments, making the game adaptable to various rule changes or AI enhancements.

**Justification**

The refactoring was necessary to improve code modularity, readability, and scalability. The original monolithic code mixed game logic, board representation, and user interaction in a single block, making it difficult to maintain and modify. By separating concerns, the GameBoard class is now responsible for managing the board, while the TicTacToe class handles game flow and player turns. The use of a 2D list structure allows for better representation and manipulation of the board, making it easier to track moves and check win conditions. Additionally, adding support for custom board sizes enhances flexibility, allowing users to play on different grid dimensions without requiring major code changes. The structured design also facilitates debugging and testing, as individual functions and components can be tested independently.

Challenges

One of the primary challenges in the original code was its reliance on a fixed 3x3 structure with hardcoded win conditions. This approach made it inflexible and difficult to scale for larger boards. Another issue was the lack of modularization, where all operations—such as board updates, player input, and game status checks—were tightly coupled, making it hard to isolate bugs or introduce new features. Handling input validation was also challenging in the monolithic version, as it lacked clear checks for incorrect or out-of-bounds moves. Additionally, testing the game was difficult since there were no unit tests or structured components, requiring manual playthroughs to check functionality.

**Approach to Adding Support for Larger Boards**

To support larger board sizes, the GameBoard class was modified to accept a dynamic board size (N x N) rather than being fixed at 3x3. The board is now initialized with a flexible structure, allowing any size to be defined at runtime. The win condition logic was also updated to dynamically generate row, column, and diagonal checks based on the board size, ensuring that the game rules scale proportionally. Instead of requiring a full row for victory, a customizable win sequence length (K) was introduced, allowing players to specify how many consecutive marks are needed to win. This approach makes the implementation significantly more adaptable while keeping the core logic consistent. The refactored design ensures that modifications for different board sizes are handled without rewriting fundamental game logic, making future enhancements easier to implement.

### Short Answer (Knowledge Questions)

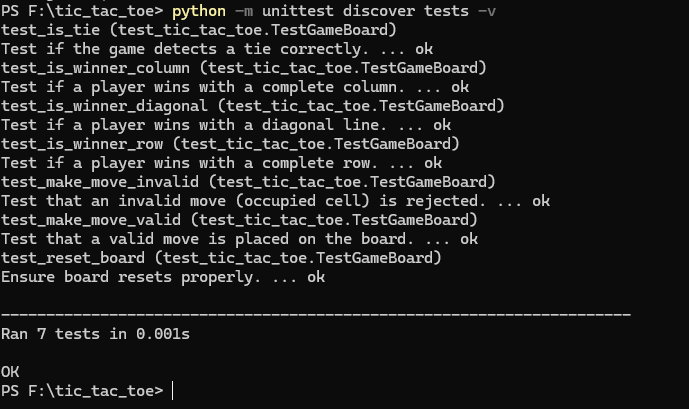
Provide brief answers to the knowledge-question worksheet.

Briefly explain what modular programming is. How can you import only a specific function or class from a module in Python? What is the syntax for this?

Modular programming is a software design approach that breaks a program into smaller, independent, and reusable modules, each handling a specific functionality. This enhances code organization, maintainability, and reusability, making it easier to debug and modify individual components without affecting the entire system. In Python, modularity is achieved by organizing code into separate files, known as modules, which can be imported and used in other scripts. Instead of importing an entire module, Python allows importing only specific functions or classes using the from ... import ... syntax. This approach is beneficial for optimizing performance and improving code clarity by only including necessary components. For example, if a module named math\_operations.py contains multiple functions, but only the add function is required, it can be imported using from math\_operations import add, allowing direct usage without referencing the entire module. Similarly, if the module contains a class, it can be imported in the same way. By using modular programming and selective imports, Python enables efficient code reuse and better organization in larger projects.

Describe your approach to debugging the tests you created tests in this task. Describe the challenges and include IDE screenshots of you debugging your tests.

To debug the tests, I used verbose test execution, print statements, and IDE debugging tools to identify and fix issues efficiently. Running python -m unittest discover tests -v provided detailed error messages, allowing quick identification of failing test cases. When errors were unclear, print statements were added to inspect board states and function outputs. Additionally, breakpoints in PyCharm enabled step-by-step execution, making it easier to analyze variable values and track logic errors. A key challenge was handling win conditions for larger boards, which required refining the is\_winner method. Another issue was ensuring input validation in tests, which was resolved using unittest.mock.patch to simulate user moves. Through this process, systematic debugging helped ensure that all test cases ran successfully.



How would you explain Python's parameter-passing mechanism? Is it more like pass-by-value or pass-by-reference? Justify your answer.

Python uses a pass-by-object-reference mechanism, meaning that function arguments are passed as references to the actual objects. However, its behavior differs depending on whether the object is mutable or immutable. When passing mutable objects like lists or dictionaries, modifications inside the function affect the original object, similar to pass-by-reference. In contrast, when passing immutable objects like integers or strings, modifications inside the function do not alter the original value but instead create a new object, behaving like pass-by-value. This distinction means that while Python does not strictly follow either pass-by-value or pass-by-reference, it exhibits characteristics of both depending on the object type.

Given the following Python code, what will be the output and why?

def modify\_list(list\_):

list\_.append("new")

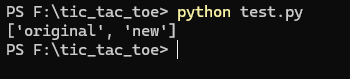
list\_ = ["completely", "new"]

items = ["original"]

modify\_list(items)

print(items)

# Output:



# Explanation:

In the function modify\_list(list\_), two operations are performed:

* list\_.append("new") modifies the original list by adding "new" to it. Since lists are mutable and passed by reference, this change persists outside the function.
* list\_ = ["completely", "new"] creates a new local list within the function, but this reassignment does not affect the original list outside the function.

When modify\_list(items) is called, the append operation modifies items by adding "new", but the reassignment inside the function only affects list\_ locally. When print(items) is executed, it outputs ['original', 'new'], as the original list has been modified by the append method but remains unaffected by the reassignment.

In Python, even though variables created within a function are local, there are still situations where you can modify data outside the scope with a local variable. Explain this anomaly and relate it to both mutability and pass by reference.

In Python, variables created within a function are generally local, meaning they do not affect variables outside the function scope. However, an exception occurs when dealing with mutable objects like lists, dictionaries, and sets. This anomaly arises due to Python’s pass-by-object-reference mechanism, where functions receive a reference to the actual object rather than a copy. As a result, modifications to the contents of mutable objects persist outside the function, even though the variable name itself remains local.

For example, if a list is passed as an argument to a function and modified using methods like append() or remove(), the changes directly affect the original list because the function operates on the same memory reference. However, if the function reassigns the variable to a new list, this creates a new local reference without affecting the original object. This behavior contrasts with immutable objects such as integers, strings, and tuples, which cannot be modified in place. Instead, any operation on an immutable object inside a function results in a new object being created, leaving the original variable unchanged unless explicitly reassigned outside the function.

This distinction between mutability and pass-by-reference-like behavior explains why some objects can be modified within a function while others cannot. Understanding this mechanism is crucial for avoiding unintended side effects in functions and ensuring that modifications to mutable objects are intentional and controlled.

List two benefits of modular coding approaches. How do these benefits assist in the development of medium-sized applications?

One major benefit of modular coding is improved maintainability, as breaking code into separate modules allows developers to work on individual components without affecting the entire system. This makes debugging easier, as issues can be isolated within specific modules, and updates or modifications can be made without disrupting the entire application. Another benefit is code reusability, where modules can be reused across different parts of a project or even in multiple projects, reducing redundancy and development time. In medium-sized applications, these benefits help by keeping the codebase organized and scalable, making it easier for multiple developers to collaborate, implement new features, and manage the growing complexity of the system.