**Worksheet: Analysing Python Code**

**Objective:**  
You are tasked with analysing and understanding the provided Python code. The code implements a game where players use mathematical operations to match targets by selecting from a set of numbers. The game can be played in two modes: a "training game" with fixed numbers and targets, and a random game with randomly generated numbers and targets.

**Part 1: General Understanding**

1. **What is the main function of the program?**
   * Describe in your own words the purpose of the program.
   * What does the user need to input to start the game? What happens when they input "y"?
2. **Identify the two modes of the game.**
   * What are the differences between the training game mode and the random game mode?
   * How does the number of targets and available numbers vary between these two modes?
3. **What is Reverse Polish Notation (RPN)?**
   * The code includes a function ConvertToRPN. What does it do, and why is RPN used in this game?

**Part 2: Functions and Logic**

1. **Main Function:**
   * Describe the role of the Main() function in the program.
   * What values are set for MaxNumber and MaxTarget depending on whether the user selects the training game or not?
2. **Game Flow:**
   * Describe the game loop inside the PlayGame function. What happens in each iteration of the loop?
   * What conditions cause the game to end?
3. **Target Matching Logic:**
   * In the CheckIfUserInputEvaluationIsATarget function, what happens when the user’s input evaluates to a target? How is the score updated?
   * How are targets removed from the list once they are matched?
4. **Handling User Input:**
   * The function CheckIfUserInputValid checks if the user's input is a valid mathematical expression. What kind of expression does it accept?
   * How does the program convert the user input into RPN (Reverse Polish Notation)?

**Part 3: Code Behaviour and Edge Cases**

1. **Invalid User Input:**
   * What happens if the user enters an invalid mathematical expression? How does the program handle this?
2. **Edge Case Handling:**
   * What happens if the user runs out of available numbers to use in the game?
   * How does the program ensure that numbers are only used once?
3. **Scoring System:**
   * How does the scoring work in this game? What happens when the user correctly matches a target? What happens when they enter an invalid expression or fail to use available numbers?

**Part 4: Code Improvements**

1. **Improving Game Flow:**
   * What improvements can be made to the game’s user interface to enhance player experience? (Consider input prompts, output formatting, etc.)
2. **Error Handling:**
   * How could the error handling be improved in the code? For example, what could be done to handle unexpected inputs more gracefully?
3. **Modularization:**
   * Are there any parts of the code that could be further modularized or optimized? If so, explain how.

Part 4 Validation

**1. Validating the User’s Initial Input (Choice)**

* In the Main() function, the user is asked to input "y" to start the training game or anything else for a random game. However, there’s no validation to ensure that the user inputs something valid.
* **Improvement:** Add a loop that repeatedly asks for input until a valid response is received (either "y" or something else).

**2. Validating User Input (Expression) in CheckIfUserInputValid**

* The current implementation checks if the user's input matches the regex pattern ^([0-9]+[\+\-\\*\/])+[0-9]+$, but it does not handle cases like division by zero or invalid combinations of operators (e.g., 3++4 or 3\*/4).
* **Improvement:**
  + Ensure no division by zero.
  + Prevent consecutive operators like ++, //, etc.
  + Ensure that expressions are well-formed (e.g., no operator at the start or end).

**3. Ensuring User Input Uses Only Available Numbers (CheckNumbersUsedAreAllInNumbersAllowed)**

* The function CheckNumbersUsedAreAllInNumbersAllowed ensures that the numbers used in the user input are among the allowed numbers. However, there is no explicit validation to check if the user uses more numbers than allowed or uses numbers that are no longer available.
* **Improvement:**
  + Check that the user is not using more numbers than are available in the NumbersAllowed list.
  + Warn the user if they are attempting to use a number that has already been used up.

**4. Validating RemoveNumbersUsed (Ensuring Proper Removal)**

* The function RemoveNumbersUsed takes user input in RPN format and removes the used numbers from the list of available numbers. However, it doesn’t ensure that the number is actually in the list before removing it. If the number isn't found in NumbersAllowed, there could be unintended behavior or errors.
* **Improvement:** Add validation to ensure that only numbers found in NumbersAllowed are removed.

**5. Handling Invalid Targets (In UpdateTargets)**

* In the UpdateTargets function, the code assumes that the list Targets is always valid and non-empty, but if all targets are set to -1 (i.e., no targets are left), the program might not handle it correctly.
* **Improvement:** Add a check to handle the case where all targets have been matched and set to -1.

**6. Ensuring Proper Numerical Range in CheckValidNumber**

* The CheckValidNumber function checks whether a number is valid based on the MaxNumber but doesn't validate if the number is negative or zero. Since the game only accepts positive integers within a range, we should explicitly check for positive numbers.
* **Improvement:** Modify the function to handle negative or zero values explicitly.

**7. Ensuring the User’s Input is Not Empty in GetNumberFromUserInput**

* In the GetNumberFromUserInput function, the program doesn't check if the user input is empty or malformed. If an empty input is given or if a non-number is entered, the program might break.
* **Improvement:** Add validation to ensure that user input is not empty or non-numeric.

Code Fixes

**1. Logic of Removing Numbers in RemoveNumbersUsed**

* **Potential issue:** The RemoveNumbersUsed function removes numbers from NumbersAllowed after each valid user input. However, the function assumes that the numbers in the expression are integers and checks if they're available in the NumbersAllowed list. If the ConvertToRPN method doesn't convert the user input correctly or if it doesn't properly handle the order of operators and operands, the removal might not work as expected.
* **Suggested Fix:** Double-check the ConvertToRPN function to ensure that it's properly handling all types of user input and converting them to Reverse Polish Notation (RPN) correctly. Additionally, confirm that RemoveNumbersUsed accurately removes the numbers that are used and doesn't inadvertently skip any.

**2. Handling Division by Zero**

* **Potential issue:** The program does not explicitly handle division by zero in the EvaluateRPN function. If the user tries to divide by zero, this will cause a runtime exception (ZeroDivisionError), and the game will crash.
* **Suggested Fix:** Add error handling for division by zero within the EvaluateRPN function.

**3.Target Updates (and Game Ending)**

* **Potential issue:** The condition to check if the game is over relies on the Targets list, where if the first target is not -1, it considers the game over. However, the game might not end if all targets are marked as -1. The loop may continue even though there are no targets left to hit.
* **Suggested Fix:** Ensure that UpdateTargets properly handles when all targets have been achieved (i.e., all targets are -1). This could be handled in the loop in PlayGame:

**4. Random Number Generation Logic**

* **Potential issue:** The random number generation in the function FillNumbers (for generating numbers allowed in the game) could cause an issue. If the MaxNumber is too small, the list NumbersAllowed might not fill up properly, and the game might fail to work as expected. For example, in the training game mode, the numbers [2, 3, 2, 8, 512] are hardcoded, which might not align with the expected behavior of having numbers randomly chosen.
* **Suggested Fix:** Ensure that the random number generation works as expected for both the training and random game modes. It would be better to have some validation or at least a more diverse random range when generating the numbers.

**5. Handling Invalid User Input in ConvertToRPN**

* **Potential issue:** The ConvertToRPN function may not handle user input as expected in cases where there are multiple operators in a row (e.g., 3++4 or 5--2). The code as it stands assumes valid input, but it doesn't validate consecutive operators or invalid characters that are non-numeric.
* **Suggested Fix:** Add validation to ensure that the user input contains valid sequences (no consecutive operators, no invalid characters).

**6. Edge Case: Empty Target List**

* **Potential issue:** If there are no targets left in the Targets list (e.g., all values are -1), the game could continue, or it might not update properly, leading to unexpected behavior. If all targets are achieved, the game should end or reset accordingly.
* **Suggested Fix:** Explicitly handle the case where no targets remain, as mentioned earlier in the UpdateTargets and PlayGame functions.

**Summary of Key Issues:**

1. **Division by zero:** Should be handled in EvaluateRPN to prevent crashes.
2. **Target list and game-ending condition:** Needs better handling when all targets are matched (-1).
3. **User input validation:** Needs to handle invalid or malformed input (e.g., 3++4, division by zero).
4. **Handling random number generation:** Ensure no duplicates and that it works properly across different game modes.
5. **Consecutive operator validation in ConvertToRPN.**

By fixing these issues, the game should work as expected, providing a smooth and error-free experience for the user.

Extensions

**1. Multiplayer Mode**

* **Feature Idea:** Allow two or more players to take turns playing the game, with scores being tracked individually.
* **Implementation Idea:** You could add a PlayerCount variable and alternate turns between players, showing their individual scores and updating them each time they take a turn. You could even have a "best of X rounds" format or allow players to compete until all targets are achieved.

def MultiplayerGame(Targets, NumbersAllowed, MaxTarget, MaxNumber, PlayerCount):

scores = [0] \* PlayerCount

GameOver = False

current\_player = 0

while not GameOver:

DisplayState(Targets, NumbersAllowed, scores[current\_player])

UserInput = input(f"Player {current\_player + 1}, enter an expression: ")

if CheckIfUserInputValid(UserInput):

UserInputInRPN = ConvertToRPN(UserInput)

if CheckNumbersUsedAreAllInNumbersAllowed(NumbersAllowed, UserInputInRPN, MaxNumber):

IsTarget, scores[current\_player] = CheckIfUserInputEvaluationIsATarget(Targets, UserInputInRPN, scores[current\_player])

if IsTarget:

NumbersAllowed = RemoveNumbersUsed(UserInput, MaxNumber, NumbersAllowed)

NumbersAllowed = FillNumbers(NumbersAllowed, True, MaxNumber)

current\_player = (current\_player + 1) % PlayerCount # Alternate players

if all(target == -1 for target in Targets): # All targets hit, game over

GameOver = True

print(f"Game over! Player 1: {scores[0]}, Player 2: {scores[1]}")

**2. Difficulty Levels**

* **Feature Idea:** Introduce multiple difficulty levels with different ranges for targets and the number of allowed numbers.
* **Implementation Idea:** You can offer a choice of difficulty (easy, medium, hard) at the beginning of the game, which modifies the maximum target value and number range. The number of targets could also be adjusted based on the difficulty level.

def SetDifficulty():

difficulty = input("Choose difficulty (easy/medium/hard): ").lower()

if difficulty == 'easy':

return 10, 20 # MaxTarget, MaxNumber

elif difficulty == 'medium':

return 50, 100

elif difficulty == 'hard':

return 100, 1000

else:

print("Invalid choice, defaulting to medium.")

return 50, 100

**3. Hint System**

* **Feature Idea:** Provide hints to the player when they are stuck, either by offering suggestions or by showing a possible correct answer.
* **Implementation Idea:** After a certain number of incorrect attempts or after a player requests help, a hint could be shown, such as a partial expression that could lead to the target.

def OfferHint(Targets, NumbersAllowed, UserInputInRPN, MaxTarget):

# Simple hint that suggests a valid operator or number

hint\_number = GetNumber(MaxTarget)

print(f"Hint: Try using {hint\_number} to approach a target!")

**4. Timed Mode**

* **Feature Idea:** Add a timed mode where players must complete the game within a certain time limit.
* **Implementation Idea:** Implement a countdown timer that gives players a limited time to solve each target or a total time to complete the game. The timer could be displayed in the console, and if the time runs out, the game ends automatically.

import time

def TimedGame(Targets, NumbersAllowed, MaxTarget, MaxNumber, time\_limit):

start\_time = time.time()

GameOver = False

while not GameOver:

elapsed\_time = time.time() - start\_time

remaining\_time = time\_limit - elapsed\_time

if remaining\_time <= 0:

print("Time's up! Game Over!")

break

DisplayState(Targets, NumbersAllowed, 0) # Display current score

print(f"Time left: {int(remaining\_time)} seconds")

UserInput = input("Enter an expression: ")

if CheckIfUserInputValid(UserInput):

UserInputInRPN = ConvertToRPN(UserInput)

if CheckNumbersUsedAreAllInNumbersAllowed(NumbersAllowed, UserInputInRPN, MaxNumber):

IsTarget, \_ = CheckIfUserInputEvaluationIsATarget(Targets, UserInputInRPN, 0)

if IsTarget:

NumbersAllowed = RemoveNumbersUsed(UserInput, MaxNumber, NumbersAllowed)

NumbersAllowed = FillNumbers(NumbersAllowed, True, MaxNumber)

if all(target == -1 for target in Targets): # All targets hit, game over

GameOver = True

print("Game Over!")

**5. Leaderboard System**

* **Feature Idea:** Track the player's high scores and maintain a leaderboard.
* **Implementation Idea:** You could save the player's scores to a file or database and compare them each time the game is played to see how they rank against their own previous bests or other players' scores.

def SaveScore(score):

with open("leaderboard.txt", "a") as f:

f.write(f"Score: {score}\n")

print("Your score has been saved to the leaderboard!")

def DisplayLeaderboard():

print("Leaderboard:")

with open("leaderboard.txt", "r") as f:

for line in f:

print(line.strip())

**6. Multiple Operators or Advanced Expressions**

* **Feature Idea:** Introduce more advanced mathematical expressions like exponents (^), square roots (sqrt), or even parentheses.
* **Implementation Idea:** Update the ConvertToRPN function and expression evaluation to handle more complex operations beyond just addition, subtraction, multiplication, and division. This would also involve expanding the list of valid operators and ensuring proper precedence and parentheses handling.

def ConvertToRPN(UserInput):

# Add support for exponents and parentheses

Precedence = {"^": 6, "+": 2, "-": 2, "\*": 4, "/": 4}

# Add parsing logic to handle "^" and parentheses

**7. Achievements and Rewards**

* **Feature Idea:** Introduce an achievement system, rewarding players for completing challenges (e.g., solving 10 targets in a row, using specific numbers).
* **Implementation Idea:** Track achievements and reward the player with badges or extra points. For example, completing a series of target numbers consecutively could grant bonus points.

def CheckAchievements(score, Targets):

# Achievement: Solve all targets in a single round

if all(target == -1 for target in Targets):

print("Achievement unlocked: Perfect round!")

score += 5

return score

**8. Customizable Targets**

* **Feature Idea:** Let the player choose specific targets or create their own set of targets.
* **Implementation Idea:** Instead of generating random targets, provide the option for the player to input their own set of targets at the start of the game.

def CustomTargets(MaxTarget):

custom\_targets = []

number\_of\_targets = int(input("How many targets do you want to set? "))

for i in range(number\_of\_targets):

target = int(input(f"Enter target {i + 1}: "))

if target <= MaxTarget:

custom\_targets.append(target)

return custom\_targets

**9. Save and Load Game State**

* **Feature Idea:** Allow players to save their progress and return later to continue the game.
* **Implementation Idea:** Save the current game state (e.g., the remaining targets, available numbers, score) to a file so the player can pick up where they left off.

def SaveGameState(Targets, NumbersAllowed, Score):

with open("game\_state.txt", "w") as file:

file.write(f"{Targets},{NumbersAllowed},{Score}")

def LoadGameState():

with open("game\_state.txt", "r") as file:

saved\_state = file.read().split(",")

Targets = saved\_state[0].split(":") # Parse into list

NumbersAllowed = list(map(int, saved\_state[1].split(":"))) # Convert to list of integers

Score = int(saved\_state[2])

return Targets, NumbersAllowed, Score

**10. Visual User Interface (GUI)**

* **Feature Idea:** Move the game to a graphical user interface (GUI) to improve interactivity.
* **Implementation Idea:** Use a Python GUI library like Tkinter or Pygame to create a more interactive experience, where users can drag and drop numbers or see a graphical representation of their score, targets, and allowed numbers.

**Summary of Features to Extend/Implement:**

* **Multiplayer mode** for competitive play.
* **Difficulty levels** for varying challenge.
* **Hint system** to assist players when stuck.
* **Timed mode** to add a sense of urgency.
* **Leaderboard and achievements** to track progress and encourage replay.