CMP5361 COMPUTER MATHEMATICS AND DECLARATIVE PROGRAMMING LOGBOOK FARKLE COMMAND LINE GAME

By Ammad Amin Raja
Student ID 20171842

Contents

Introduction

Farkle is a traditional dice game that mixes luck and strategy. Each player rolls six dice in turn, trying to make certain combinations to score points. The game is well-known for its accessibility to a broad spectrum of players due to its simplicity, yet it still has enough depth to appeal to strategic thinkers. In an exciting balance between risk and reward, players must choose whether to bank their points or risk them by rolling the dice once more, so anyone looking for some fun can play.

Overview of the Project

I will plan, design, implement and test a command-line game of Farkle using Python from start to finish. The objective is too ensure the game is well-structured implementing Farkle's rules and scoring system accurately.

Rules of the game

Farkle requires six standard dice. Rolling specific combinations earns points. The rules are easy. The game involves 2 players who roll 6 dice per turn. If any points are available, the turn player can bank them or keep rolling. The remaining dice scored zero.

Scoring System

Points are awarded based on dice combinations:

Rolling a 1: 100 points

Rolling a 5: 50 points

Three 1s: 1,000 points

Straight (1-2-3-4-5-6): 1,500 points

Three pairs: 1,500 points

The complete scoring details will be printed in the game

T1: Planning and Designing an Interactive ‘Farkle’ command line game

Requirement Analysis

This analysis identifies that functional and non-functional needs of the Farkle game. Functionally, the game must properly initialise, manage gaming mechanics, handle user interactions, regulate game flow, and declare a winner, while non-functional requirements include maintainability and testability.

User Behaviour Specifications

Explanation of User Behaviour Specifications and Their Importance

User behaviour specifications determine a software system's end-user behaviour. They establish how the system should handle user inputs and interactions. Effective communication, documentation, analysis, testing, and ensuring the product meets user expectations increases software quality and satisfaction. Gherkin Specifications can explain each feature using scenarios and programme behaviour. They are written in plain English so even non developers can understand. Gherkin Specifications have two paths:

Happy Paths: Successful scenarios with good user behaviour and valid inputs.

Sad Paths: Erroneous inputs or unexpected behaviours produce errors to validate the system's error handling and provide user feedback.

Gherkin Scenarios

System Initialization and Greeting

Scenario	Steps	Type	Variables & Example Output
System Loads and Greets the User	**Given** the system is started **When** the system loads **Then** I should see a greeting message **And** I should see the message "Reach 10,000 points to win!" **And** I should see the rules of the game printed out	Happy	**greeting_message**: "Welcome to Farkle!", **objective_message**: "Reach 10,000 points to win!", **rules_message**: "Rules: ..."

Displaying the Rules	**Given** the system is started **When** the system loads **Then** I should see the rules of the game: ...	Happy	**rules_message**: "Rules of Farkle: 1. Players take turns rolling six dice. ..."
System Fails to Load Rules	**Given** the system is started **When** the system loads **Then** an error occurs, and the rules are not displayed	Sad	**error_message**: "Error: Failed to load game rules. Please restart the application"

Setting Up Players

Scenario	Steps	Type	Variables & Example Output
Inputting Player Names	**When** the program asks for Player 1's name **Then** I input 'Test Player 1' **And** the program asks for Player 2's name **Then** I input 'Test Player 2' **And** the game should start with Player 1 and Player 2	Happy	**player1_name**: "Test Player 1", **player2_name**: "Test Player 2", **start_message**: "Players have been named: ..."
Inputting Empty Player Names	**When** the program asks for Player 1's name **And** I input nothing **Then** I should see an error message 'Player name cannot be empty. Please enter a valid name.'	Sad	**error_message**: "Player name cannot be empty. Please enter a valid name."

Starting the Game

Scenario	Steps	Type	Variables & Example Output
Game Start	**Given** there are two players	Happy	**start_message**: "The game has started!", **player_turn_message**: "Test Player 1's turn! Press Enter to roll the dice."

	When both players have been named **Then** the game should start **And** Player 1 should roll the dice first		

Rolling the Dice

Scenario	Steps	Type	Variables & Example Output
Initial Roll	**Given** it is my turn **When** I roll the dice **Then** I should roll six dice **And** the result should be displayed	Happy	**dice_roll**: "(2, 3, 4, 5, 6, 1)"
Attempting to Roll with Invalid Input	**Given** it is my turn **When** I am prompted to roll the dice **And** I input an invalid command **Then** I should see an error message 'Invalid input. Please press Enter to roll the dice.'	Sad	**invalid_input**: "x", **error_message**: "Invalid input. Please press Enter to roll the dice."

Scoring System and Combinations

Scenario	Steps	Type	Variables & Example Output
Understanding the scoring system	**Given** I have rolled the dice **When** I see the results **Then** each die should have a different point value **And** the points should be calculated as follows:	Happy	**die_face_points**: {"1": 100, "2": 0, "3": 0, "4": 0, "5": 50, "6": 0}
Special dice such as 1 and 5	**Given** I have rolled the dice **When** I see a 1 or a 5 **Then** I should know that 1 is worth 100	Happy	**dice_roll**: "(1, 1, 5, 2, 4, 3)", **roll_score**: 250, **turn_score**: 250, **remaining_dice**: 3, **special_points**: {"1": 100, "5": 50}

	points and 5 is worth 50 points **And** I should see the points calculated as follows:		
Triplets and Higher Combinations	**Given** it is my turn **When** I roll 3 or more of the same number (other than ones or fives) **Then** I should see the points calculated based on Farkle rules **And** these points can be banked or I can continue rolling	Happy	**dice_roll**: "(3, 3, 3, 4, 2, 6)", **roll_score**: 300, **turn_score**: 300, **remaining_dice**: 3, **bank_choice_message**: "Do you want to bank the points and end your turn (b), or roll the remaining dice (r)?"
Banking Points and Continuing to Roll	**Given** I have banked 250 points and have 3 dice remaining **When** I roll the remaining 3 dice **Then** any new points or combinations are added to the banked points **And** I can choose to 'Bank In' or continue rolling	Happy	**dice_roll**: "(2, 4, 5)", **roll_score**: 50, **turn_score**: 300, **remaining_dice**: 2, **bank_choice_message**: "Do you want to bank the points and end your turn (b), or roll the remaining dice (r)?"

Saving Points

Scenario	Steps	Type	Variables & Example Output
Saving points and continuing to roll	**Given** I have rolled the dice **And** I have scored points **When** I choose to save my points and continue	Happy	**roll_score**: 200, **turn_score**: 200, **remaining_dice**: 4, **bank_choice_message**: "Do you want to bank the points and end your turn (b), or roll the remaining dice (r)?"

	Then my points should be added to my potential score **And** I should roll the remaining dice **And** if I Farkle on the next roll, my saved points should not be added to my total score		

Handling a Farkle

Scenario	Steps	Type	Variables & Example Output
Rolling a Farkle	**Given** it is my turn **When** I roll the dice **And** none of the remaining dice score points **Then** I should be notified that I Farkled **And** my turn should end immediately **And** the points accumulated if any during this turn should not be added to my total score **And** the turn should pass to the next player	Sad	**dice_roll**: "(2, 4, 6, 3, 2, 5)", **farkle_message**: "FARKLE! You lose all points for this turn."

Turn Transition or Ending Turn

Scenario	Steps	Type	Variables & Example Output
Turn Ends after Player Banks Points	**Given** it is Player 1's turn **When** Player 1 decides to 'Bank In' their points **Then** Player 1's turn ends	Happy	**banked_points**: 300, **player_turn_message**: "Player 2's turn! Press Enter to roll the dice."

	And Player 2's turn begins		
Turn Ends after Farkle	**Given** it is Player 1's turn **When** Player 1 rolls a Farkle **Then** all points banked in that turn are lost **And** Player 1's turn ends **And** Player 2's turn begins	Sad	**farkle_roll**: (2, 4, 6, 3, 2, 5), **farkle_message**: "FARKLE! You lose all points for this turn.", **player_turn_message**: "Player 2's turn! Press Enter to roll the dice."

Winning the Game

Scenario	Steps	Type	Variables & Example Output
Reaching the winning score	**Given** I have been playing the game **And** my score is close to the winning score **When** my total score reaches 10,000 points **Then** I should be declared the winner **And** the game should end **And** I should see a congratulatory message	Happy	**winning_message**: "Test Player 1 wins with a score of 10,000! Congratulations!"

Data modelling

Input Arguments Model

Component	Data Type	Example	Mathematical Definition
Player Names	String	"Test Player"	Let P be the set of player names. P = { "Player1", "Player2" } This means P has a cardinality of 2, representing the two players.
Dice Count	Integer	6	Let D be the set of dice faces. D = { 1, 2, 3, 4, 5, 6 } The cardinality of D is 6, representing the six faces of a die. A roll is an element of D^6, representing all possible combinations of six dice rolls.
Player Choices	String	"r" (roll), "b" (bank)	Represents the player's decision to roll or bank.

Dice Model

Component	Data Type	Example	Mathematical Definition
Dice Roll	Tuple of integers	(2, 5, 3, 6, 1, 4)	Let R be the set of all possible rolls of six dice. $R \in D^6$ R (a specific roll of six dice) is an element of the set D^6 (the set of all possible combinations of six dice rolls) so we use the D^6 to define all possible outcomes and then R is identified as of one of these outcomes. Example: R = (1, 5, 2, 3, 5, 6)
Dice Count	Integer	6	The number of dice to be rolled.

Score Model

Component	Data Type	Example	Mathematical Definition
Score Calculation	Integer	1500	Let S be the set of scores. S = { (Player1, 0), (Player2, 0) } . This set represents the initial scores of the two players in the game, where both players start with a score of 0. The cardinality of this set is 2, as it contains scores for two players. Score Function: f:D→N. The score function f maps each dice face to a score. f(1)=100, f(5)=50, f(d)=0 for d∈{2,3,4,6}. Depending on the roll points are awarded so 1 is 100 points, 5 is 50 points and combination of 2,3,4,6 is 0 points.
Remaining Dice	Integer	2	2 ∈ D, where D is the set of possible remaining dice.

Player Model

Component	Data Type	Example	Mathematical Definition
Player Name	String	"Player1"	"Player1" ∈ P, where P is the set of all possible player names.
Player Score	Integer	3500	3500 ∈ S, where S = {x ∈ Z | x ≥ 0} (the set of non-negative integers).
Accumulated Points	Boolean	True	True ∈ B, where B = {True, False}.

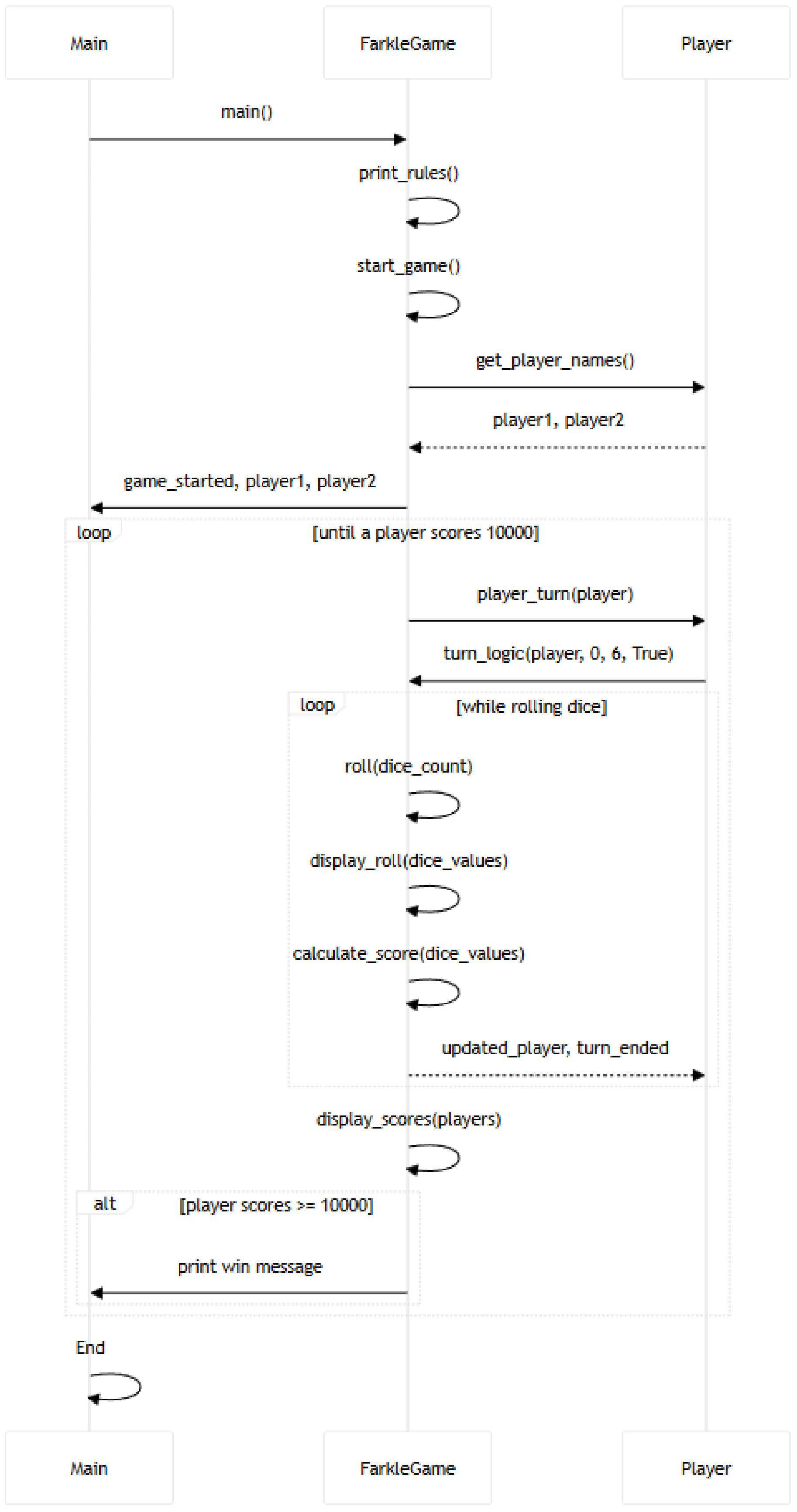
Axiomatic Definitions and Functions

Function	Description	Mathematical Definition
get_player_names()	Handles input for player names, ensuring non-empty values.	Let P be the set of players. P={"$Player1$","$Player2$"}
display_scores()	Displays the current scores of all players.	Let S be the set of scores. S={($Player$1,0),($Player$2,0)}
roll(dice_count: int)	Simulates rolling a specified number of dice.	Let D be the set of dice faces. D={1,2,3,4,5,6} A roll is an element of D^6. Example: R=(1,5,2,3,5,6)
get_player_names()	Parses user inputs and ensures they are valid.	Input validation ensures that $P \neq \emptyset$. The set of player names must have at least one element and it cannot be empty.
roll(dice_count: int)	Initializes a new set of dice for rolling.	Let D be the set of dice faces. D={1,2,3,4,5,6}
main()	Checks if any player has won the game by reaching the target score.	Winning Condition: Define a predicate $Win(p,s)$ which is true if player p's score s is 10,000 or more. $Win(p,s) \iff s \geq 10{,}000$ Win(p,s) is true if and only if $s \geq$ 10,000

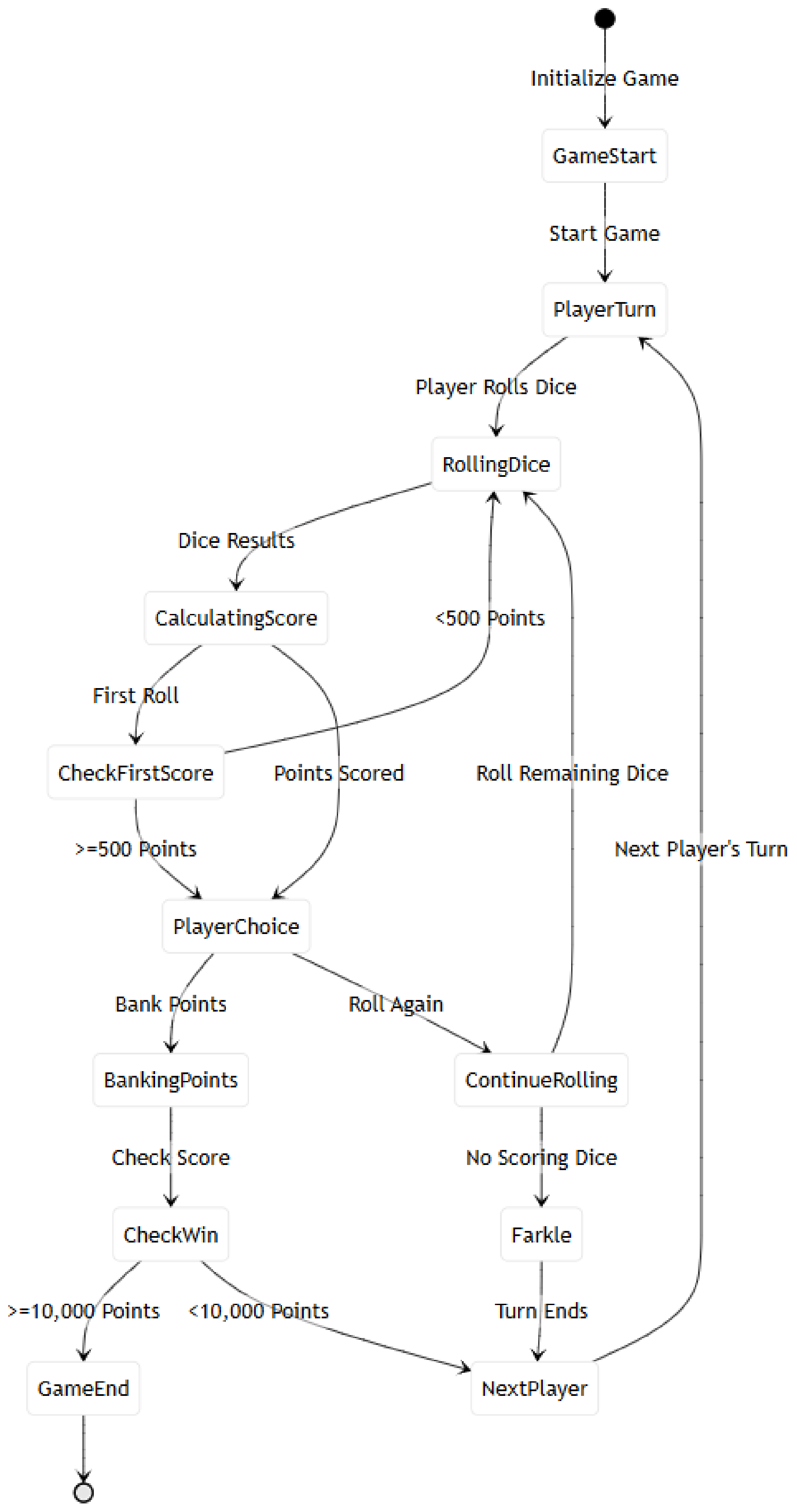
Player Turn Example

Aspect	Description	Example
Initial State	Initial state with Player1 and score 0.	Let p = $Player$, s = 0, and roll R=(1,5,2,3,5,6).
Score Calculation	Calculation based on rolled values.	Score for roll = $f(1) + f(5) + f(5)$=100+50+50=200.
Bank or Continue to Roll	Updating score based on player action.	Turn Function: Depending on whether the player decides to bank the points or continue rolling. $turn(p,s)$={($Player1$,200) if Bank, ($Player1$,0) if Farkle on n ext roll. If the player banks the points, their score becomes 200. If the player Farkles on the next roll, their score remains 0.

Sequence Diagram



State Transition Diagram



T2 Implementation

I chose Python for developing my Farkle game because it is the language I am most comfortable with. In my implementation, I used immutability and a modular approach to divide the scoring logic into many, specialised functions, each responsible for a specific aspect of the scoring process. By using pure functions, I assured that each function provides the same output for the same input and has no side effects.

Imports

Modules and functions that are imported

```
import random
from collections import Counter
from dataclasses import dataclass, replace
from typing import List, Tuple
```

random is a module that provides functions for generating random numbers, this is used for rolling the dices.

Collections is a module that implements specialized container datatypes, such as **Counter** which is a class that counts the occurrences of elements in a list or iterable., so this would be used to count the occurrences of each die value in a dice roll.

Dataclass is a module used to automatically generate special methods like __**init**__ and __**repr**__ for classes, in this case it is for the player class. The **replace** function creates new instances of the **dataclass**, replacing specific fields with new values, this is a way to keep my player class immutable as the original instance for it will be unchained.

List and **tuple** are imported from the **typing** module for type hinting, this makes it clear what types of inputs a function expects and what type of output it returns.

Player class

```
@dataclass(frozen=True)
class Player:
    name: str
    score: int = 0
    accumulated_points: bool = False
```

The **player** class is defined using the **@dataclass** decorator with **(froze=true)** which makes it immutable. Thanks to this the state of the player cannot be changed unexpectedly. It has 3 fields which are all type annotated. This prevents any side effects and thus makes the code immutable.

Score Functions

```
def gain_score(player: Player, points: int) -> Player:
    return replace(player, score=player.score + points)

def set_accumulated_points(player: Player, value: bool) -> Player:
    return replace(player, accumulated_points=value)
```

The type hint **player: Player** specifies that the player parameter is expected to be an instance of the Player class, and **-> Player** indicates that these functions return a new instance of the Player class. This is where the replace function is used. These two functions allow the state of the player to be updated without changing the original instance, strengthening the immutability of the code.

Print Rules

```python
def print_rules() -> None:
    """
    Prints the rules of the game.
    """
    rules = """
    print(rules)
```

This is a simple function that prints out all the rules of the game. It does not modify any state or produce side effects other than printing to the console. It has a type hint with 'None' indicating that it does not return any value.

Roll Dice

```python
def roll(dice_count: int) -> Tuple[int, ...]:
   #Simulates the rolling of dice and returns it as a tuple.
    return tuple(random.randint(1, 6) for _ in range(dice_count))
```

This function manages the roll of the dice. It takes the number of dice to roll as argument and returns a tuple of integers representing the dice roll. Random integers from 1 to 6 are generated thanks to random import. This function is essentially immutable as the dice rolled are in a tuple meaning they cannot be changed once created.

Scoring Rules

```python
def apply_scoring_rules(count: Counter, scoring_rules: Dict[Tuple[int, ...],
Any]) -> Tuple[int, Counter, int]:
    score = 0
    temp_count = count.copy()
    used_dice = 0

    for key in sorted(scoring_rules.keys(), key=lambda x: -len(x)):
        while all(temp_count[num] >= key.count(num) for num in key):
            score += scoring_rules[key](temp_count)
            for num in key:
                temp_count[num] -= key.count(num)
            used_dice += len(key)

    return score, temp_count, used_dice
```

The **apply_scoring_rules** function takes a **Counter** object and a dictionary of scoring rules to calculate the score based on predefined dice combinations. The score is initialized to 0. It returns a **tuple** containing the accumulated score, an updated Counter reflecting the remaining dice and the number of dice used. The function creates a copy of the **input Counter** to ensure immutability and iterates over sorted scoring rules to prioritize larger combinations. For each key, it checks if the dice roll contains the specified combination, applies the corresponding score, and updates the counts and used dice accordingly, this is done through 2 nested loops an “outer” for loop and an “inner” while loop.

Special Dice Calculations

```
def adjust_for_single_1s_and_5s(count: Counter) -> Tuple[int, int]:
    score = 0
    single_ones = count[1] % 3
    single_fives = count[5] % 3
    score += single_ones * 100
    score += single_fives * 50
    used_dice = single_ones + single_fives

    return score, used_dice
```

The **adjust_for_single_1s_and_5s** function handles scoring for single 1s and 5s that are not part of larger combinations. It takes a **Counter** object of remaining dice calculates the score for any single 1s (worth 100 points each) and single 5s (worth 50 points each) and returns a tuple of the score and the number of dice used. The score for all 1s and 5s is directly calculated and is multiplied based on the amount of 1s and 5s, then the total number of 1s and 5s that have been found are subtracted by the total number of dice. The function ensures immutability by not modifying the input Counter directly. This function ensures that any remaining single 1s and 5s are scored appropriately after applying the primary scoring rules for higher combinations.

Calculate Score

```
def calculate_score(dice_roll: Tuple[int, ...]) -> Tuple[int, int]:
    count = Counter(dice_roll)


    if all(count[d] == 1 for d in range(1, 7)):
        return 1500, 0
    if len(count) == 3 and all(v == 2 for v in count.values()):
        return 1500, 0


    scoring_rules = {
        (1, 1, 1): lambda _: 1000,
        (2, 2, 2): lambda _: 200,
        (3, 3, 3): lambda _: 300,
        (4, 4, 4): lambda _: 400,
        (5, 5, 5): lambda _: 500,
        (6, 6, 6): lambda _: 600,
        (1, 1, 1, 1): lambda _: 1000,
        (2, 2, 2, 2): lambda _: 1000,
        (3, 3, 3, 3): lambda _: 1000,
        (4, 4, 4, 4): lambda _: 1000,
        (5, 5, 5, 5): lambda _: 1000,
        (6, 6, 6, 6): lambda _: 1000,
        (1, 1, 1, 1, 1): lambda _: 2000,
        (2, 2, 2, 2, 2): lambda _: 2000,
        (3, 3, 3, 3, 3): lambda _: 2000,
        (4, 4, 4, 4, 4): lambda _: 2000,
        (5, 5, 5, 5, 5): lambda _: 2000,
        (6, 6, 6, 6, 6): lambda _: 2000,
        (1, 1, 1, 1, 1, 1): lambda _: 3000,
        (2, 2, 2, 2, 2, 2): lambda _: 3000,
        (3, 3, 3, 3, 3, 3): lambda _: 3000,
        (4, 4, 4, 4, 4, 4): lambda _: 3000,
        (5, 5, 5, 5, 5, 5): lambda _: 3000,
        (6, 6, 6, 6, 6, 6): lambda _: 3000,
    }

    score, temp_count, used_dice = apply_scoring_rules(count, scoring_rules)
    single_score, single_used_dice = adjust_for_single_1s_and_5s(temp_count)
    score += single_score
    used_dice += single_used_dice

    remaining_dice = len(dice_roll) - used_dice


    remaining_dice = max(remaining_dice, 0)

    return score, remaining_dice
```

This function only takes a single argument **dice_roll**, which is a tuple of integers representing the rolled dice values, and returns a tuple of two integers: the score and the number of remaining dice. The "counter" objects counts the value of each dice occurrences in the tuple.

The first check done by this function is for the special combinations first which are straight and three pairs. **if all(count[d] == 1 for d in range (1, 7)):** checks if each die value from 1 to 6 appears exactly once. If true, it returns a score of 1500 and 0 remaining dice, while the one below if **len(count) == 3 and all (v == 2 for v in count.values()): checks** if there are exactly three different pairs. If true, it also returns a score of 1500 and 0 remaining dice. If these special combinations are not found, the function defines a dictionary**, scoring_rules**, containing lambda functions for various dice combinations and their corresponding scores, it then calls the **apply_scoring_rules** function with count and **scoring_rules** to calculate the score for primary combinations and update the remaining dice count. Next, it calls the **adjust_for_single_1s_and_5s** function to handle any remaining single 1s and 5s, adding their scores to the total score and updating the used dice count.

The **remaining_dice** is calculated by subtracting the total number of used dice from the length of **dice_roll** and ensuring it is not negative. Lastly the final return statement returns a tuple containing the calculated score and remaining dice.

Display Score

```
def display_scores(players: List[Player]) -> None:
    scores = "\n".join(f"{player.name}: {player.score} points" for player in
players)
    print(f"\nScores:\n{scores}\n")
```

This function only expects a single argument, **players**, which is a list of **Player** objects. It returns no value as described by the **none**. It generates a string with each player's name and score on a new line from a list of Player objects. With the title "Scores:", the command line prints this string showcasing the name of the player and their total score. This function does not have a side effect apart from printing to the console thus it can be considered a pure function because it always produces the same output given the same input.

Display Roll

```
def display_roll(dice_values: Tuple[int, ...]) -> None:
    print(f"Dice roll: {dice_values}")
```

This function is responsible for displaying the value of the rolled dice onto the console. It takes a tuple of integers as input and constructs a formatted string that includes these values. It returns no value. This function can also be considered immutable as it does not modify any input data or external state, it only prints the value of each rolled dice.

Turn Logic

```
def turn_logic(player: Player, turn_score: int, num_dice: int, first_roll:
bool) -> Tuple[Player, bool]:
    if num_dice == 0:
        return gain_score(player, turn_score), True

    input("Press Enter to roll the dice.")
    dice_values = roll(num_dice)
    display_roll(dice_values)
    roll_score, remaining_dice = calculate_score(dice_values)

    if roll_score == 0:
        print("FARKLE! You lose all points for this turn.")
        return player, True

    turn_score += roll_score
    print(f"Roll score: {roll_score}, Turn score: {turn_score}, Remaining
dice: {remaining_dice}")

    if first_roll and turn_score < 500 and not player.accumulated_points:
        print("You need at least 500 points to start accumulating points.")
        return turn_logic(player, turn_score, remaining_dice, first_roll)
    else:
        player = set_accumulated_points(player, True)

    while True:
```

```python
        choice = input("Do you want to bank the points and end your turn (b),
or roll the remaining dice (r)? ").lower()
        if choice in ('b', 'r'):
            break
        else:
            print("Invalid choice. Please enter 'b' to bank the points or 'r'
to roll the remaining dice.")

    if choice == 'b':
        return gain_score(player, turn_score), True
    elif remaining_dice == 0:
        return turn_logic(player, turn_score, 6, False)
    else:
        return turn_logic(player, turn_score, remaining_dice, False)
```
This is another important function in my implementation as it manages the turn of a player. This function takes 4 arguments which are: **player: Player, turn_score: int, num_dice: int, first_roll: bool.** The return type of this function is a tuple containing the updated player object and a Boolean which indicates whether the turn has ended. Tuples are immutable so this is one way to make my implementation immutable. If no dices are left to roll the function updates the player’s total score by adding the accumulated turn score and ends the turn since the Boolean is switched to **True**.

The next statement prompts the user to press “enter” to roll the dice. Rolling the dice, displaying the roll, calculating score are the functions that are called.

The **farkle** check works if the roll score contains no matching lambda functions or special rolls such as 1s and 5s. If the turn score is 0 after rolling all dices, the turn is ended, and the user is informed that they have Farkled.

The turn score will be updated with each new roll score until the user has no dice remaining or choose to bank their points and finish the turn.

The next statement is a **conditional statement** that determines whether a player has scored at least 500 points once in a turn. As in Farkle, a player can only begin accumulating points after scoring 500 points; once they have satisfied this requirement, they can begin banking points for the remainder of the game.

Next, I have a **while** loop which gives the player two options after successfully rolling, these two options are either keep rolling the remaining dices or bank any points accumulated and end turn, keep in mind that this loop will only work after a player has accumulated at least 500 points once. The user must input either **“r”** to continue rolling or **“b”** to bank points. Depending on the input post loop conditions are called. If the user picks “b” then the turn is ended, and all current turn points are added to the total score. While if the user picks “r” the remaining dice are rolled again and the calculate score function is called, it user scores again the loop repeats where the user can then decide to bank the points or keep rolling with any left dices, it is important to note that if the user does not score again then the farkle handling logic is called, the accumulated turn points are reset to 0 and the turn is ended.

This function maintains immutability by not directly modifying the player object. Instead, it creates new instances of the Player object with updated attributes such as the total score.

These new instances are returned and used, ensuring that the original player object remains unchanged.

Player Turn

```python
def player_turn(player: Player) -> Player:
    print(f"\n{player.name}'s turn!")
    updated_player, turn_ended = turn_logic(player, 0, 6, True)
    if not player.accumulated_points and updated_player.score < 500:
        # Reset turn score if the player has not accumulated the required 500
points
        updated_player = Player(player.name, player.score,
player.accumulated_points)
    return updated_player
```

This function is required to control the turn of a single player. It takes a single argument, **Player**, and returns an updated Player object once the turn is complete. This function prints out which player's turn it is. The **turn_logic** function is used here. There is also a conditional statement to determine whether the player has acquired the required 500 points. Finally, the modified player object is returned. This object is a new instance of the player object; thus it does not modify it directly, ensuring immutability.

Get Player Name

```python
def get_player_names() -> Tuple[str, str]:
    player1_name = ""
    while not player1_name:
        player1_name = input("Enter name of Player 1: ")
        if not player1_name:
            print("Player name cannot be empty. Please enter a valid name.")

    player2_name = ""
    while not player2_name:
        player2_name = input("Enter name of Player 2: ")
        if not player2_name:
            print("Player name cannot be empty. Please enter a valid name.")

    return player1_name, player2_name
```

This function does not take any arguments and it returns a tuple containing 2 strings, which are essentially the names for player 1 and player 2. Each player's name is initialized with and empty string. There is a while loop for each player ensuring that the player's name input cannot be empty. Finally, it returns the tuple containing both player 1 and player 2 names.

Start Game

```
def start_game() -> Tuple[bool, Player, Player]:
    player1_name, player2_name = get_player_names()
    player1 = Player(player1_name)
    player2 = Player(player2_name)
    print("Players have been named:")
    print(player1)
    print(player2)
    input("If both players are ready, press Enter to begin...")
    return True, player1, player2
```

This function does not take any arguments and returns a tuple which includes a boolean value indicating if the game has started, and two Player objects representing Player 1 and Player 2. The **get_player_names** function is called in here and then 2 new instances of the player 1 and player 2 are created with the selected names. It also announces the players and asks for player input on whether they are ready for the game. Finally it returns a tuple indicating that the game has started (True), along with the Player objects for Player 1 and Player 2.

Main Function

```
from FarkleGame import print_rules, start_game, player_turn, display_scores

def main() -> None:
    print_rules()

    game_started, player1, player2 = start_game()

    if game_started:
        print("The game has started!")
        players = [player1, player2]
        while all(player.score < 10000 for player in players):
            for i, player in enumerate(players):
                players[i] = player_turn(player)
                display_scores(players)
                if players[i].score >= 10000:
                    print(f"{players[i].name} wins with a score of
{players[i].score}!")
                    return

if __name__ == "__main__":
    main()
```

For this function some important imports are necessary which are import **print_rules, start_game, player_turn, display_scores.** This ensures that this function can call the imported functions when running the code.

This function does not take any arguments and does not have a return type.

The rules are printed as the first action, then the **start_game** function is called to initialize the game and get the players ready.

There is a conditional statement to check whether the game has started and if it has it prints out the game start notification. If this is true, then the players are initialized.

The main game loop continues while all players have less than 10,000 points; it iterates over each player, calling the **player_turn** function to manage their turn and then the **display_scores** function to update and display the current scores.

The if conditional statement checks for a winner by determining whether the current player's score has reached or exceeded 10,000 points; if true, it prints a winner message and terminates the loop. This is how the game ends.

I have finally added an entry check point to ensure the script behaves correctly when run directly and when imported as a module.

In general, this function ensures immutability by generating new instances of player objects instead of altering existing ones.

Reflection on Implementation

I improved my Python programming skills by focusing on immutability and modular design while making my Farkle game. Dataclasses and replace prevented unexpected side effects by ensuring immutability. This helped me understand pure functions better also the modular approach into smaller functions improved code readability and maintainability and I understand the value of having the functions split into smaller bits rather than having a big chunky function that is hard to read and manage. I also have a better understanding of loops and conditional statements. I learned to implement lamda functions.

Although there have been significant developments, I admit that there are many areas in which I can improve such as getting even a better understanding of functional programming. I believe my mindset on how to approach problems has developed. Overall, I think the implementation was decent. I will try more projects to gain more experience and knowledge.

T3: Testing an Interactive ‘Farkle’ command line game.

Manual Testing

This testing will involve manually interacting with the project to ensure certain functions work as intended.

Test Case ID: 1 **Pass**/Fail

Feature Tested: System Loads and Greets the User

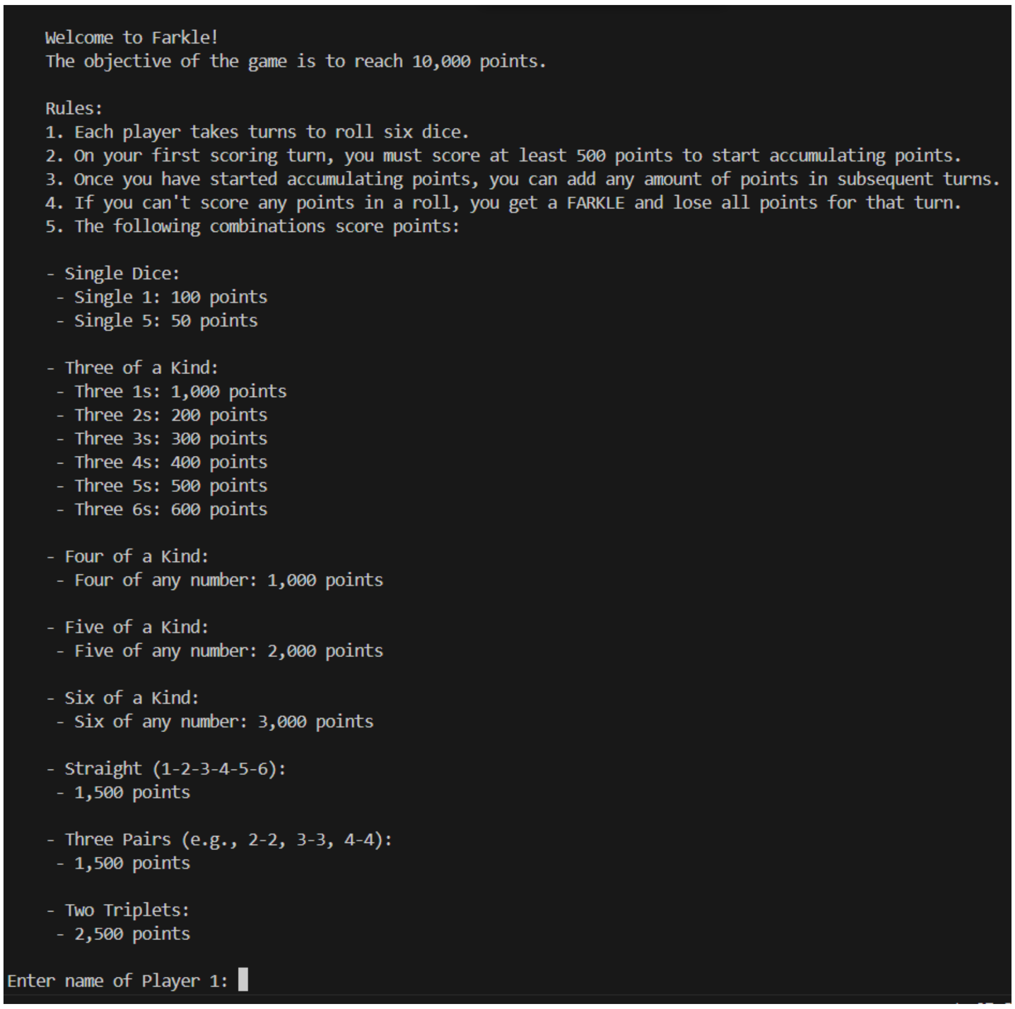
Steps

Run the program from the main.py

Expected Output

When the Farkle game starts successfully and the system loads, the user should see a welcoming message "Welcome to Farkle!", followed by the objective "Reach 10,000 points to win!", and the complete game rules should be displayed to provide an overview of how to play the game.

Actual Output



```
    Welcome to Farkle!
    The objective of the game is to reach 10,000 points.

    Rules:
    1. Each player takes turns to roll six dice.
    2. On your first scoring turn, you must score at least 500 points to start accumulating points.
    3. Once you have started accumulating points, you can add any amount of points in subsequent turns.
    4. If you can't score any points in a roll, you get a FARKLE and lose all points for that turn.
    5. The following combinations score points:

    - Single Dice:
     - Single 1: 100 points
     - Single 5: 50 points

    - Three of a Kind:
     - Three 1s: 1,000 points
     - Three 2s: 200 points
     - Three 3s: 300 points
     - Three 4s: 400 points
     - Three 5s: 500 points
     - Three 6s: 600 points

    - Four of a Kind:
     - Four of any number: 1,000 points

    - Five of a Kind:
     - Five of any number: 2,000 points

    - Six of a Kind:
     - Six of any number: 3,000 points

    - Straight (1-2-3-4-5-6):
     - 1,500 points

    - Three Pairs (e.g., 2-2, 3-3, 4-4):
     - 1,500 points

    - Two Triplets:
     - 2,500 points

Enter name of Player 1:
```

Test Case ID: 2 **Pass**/Fail

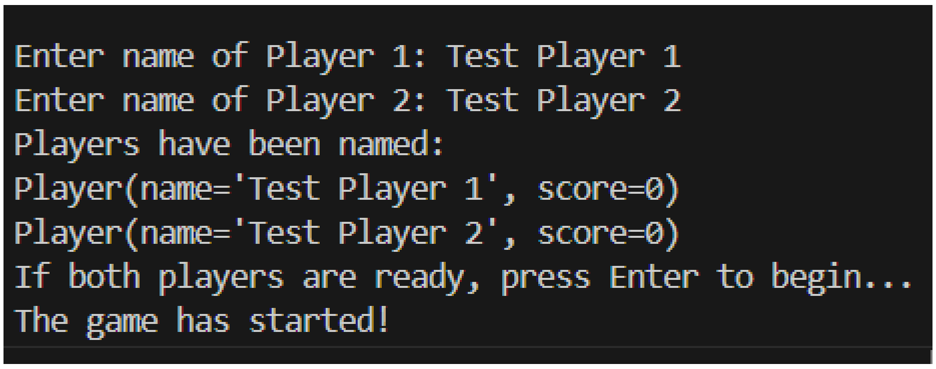
Feature Tested: Inputting Player Names

Steps
Run the program from the main.py
Enter Player 1 name
Enter Player 2 name
Press enter to confirm if both players are ready to go

Expected Output

The game prompts for Player 1's name, and after entering "Test Player 1", it prompts for Player 2's name. After entering "Test Player 2", the game should start with these names, confirming the players have been successfully named.

Actual Output



```
Enter name of Player 1: Test Player 1
Enter name of Player 2: Test Player 2
Players have been named:
Player(name='Test Player 1', score=0)
Player(name='Test Player 2', score=0)
If both players are ready, press Enter to begin...
The game has started!
```

Test Case ID: 3 **Pass**/Fail

Feature Tested: Game Start and Initial Roll

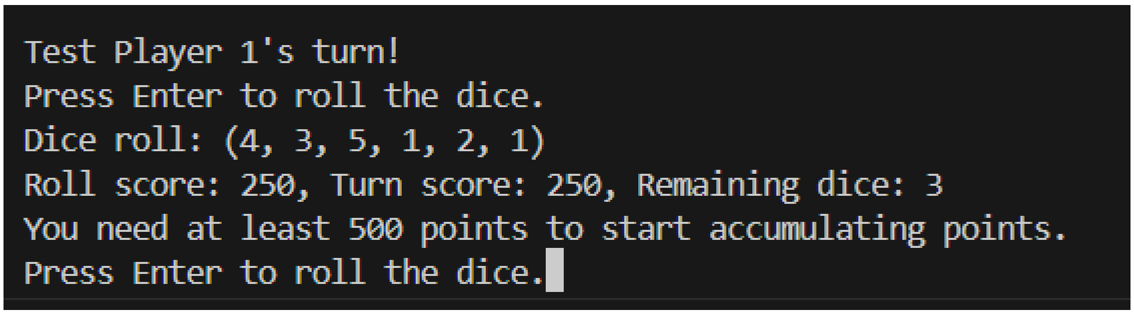
Steps

Run the program from the main.py
Enter Player 1 name
Enter Player 2 name
Press enter to confirm if both players are ready to go then the game will start
Player 1’s turn will begin
Player 1 will enter to roll dice
Player 1’s turn score will be displayed along with roll score

Expected Output

With both players named, the game begins. Player 1 should be prompted to roll the dice first with a message like "Player 1's turn! Press Enter to roll the dice," indicating the start of the gameplay and at the start of a turn, the player rolls six dice. The result of the roll should be displayed on the screen, showing the values of the six dice to the player.

Actual Output



```
Test Player 1's turn!
Press Enter to roll the dice.
Dice roll: (4, 3, 5, 1, 2, 1)
Roll score: 250, Turn score: 250, Remaining dice: 3
You need at least 500 points to start accumulating points.
Press Enter to roll the dice.
```

Test Case ID: 4 **Pass**/Fail

Feature Tested: Understanding the Scoring System

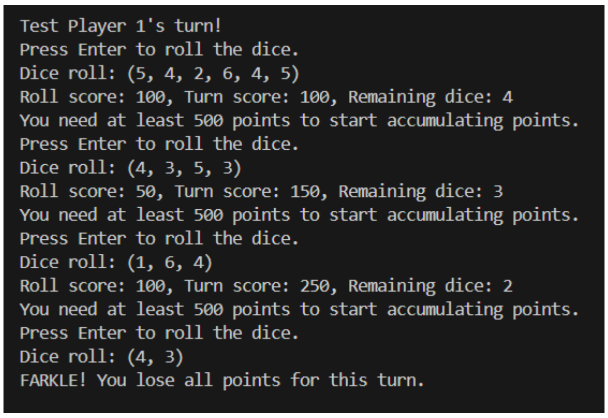
Steps

During a player's turn they will roll and score points
These points are calculated based on different combinations
There are special dice such as 1s and 5s
The system will correctly display the calculated points
The player should then see an option to bank these points or continue rolling, showing a message like "Do you want to bank the points and end your turn (b), or roll the remaining dice (r)?", assuming they have initially scored 500 or more points

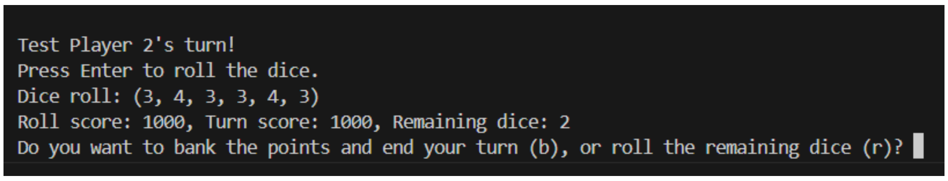
Expected Output

After rolling the dice, each die should display a different point value according to Farkle rules. The points should be calculated and shown, helping the player understand how their roll translates to points. Each player will roll and different dice combinations will be calculated. Such as simple 1s and 5s, and 3 or higher of a kind combinations. Additionally, if a player hasn't scored 500 yet they will be prompted to roll again if any dice are left otherwise their turn will end with 0 points.

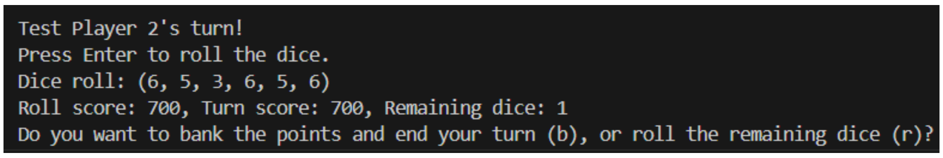
Actual Output



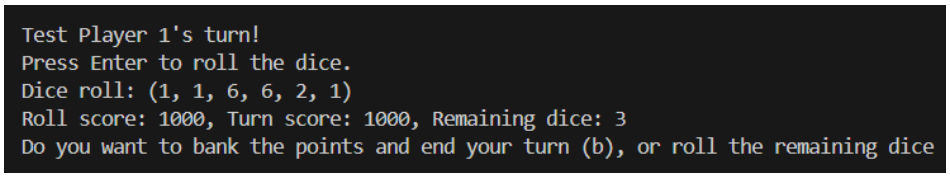
```
Test Player 1's turn!
Press Enter to roll the dice.
Dice roll: (5, 4, 2, 6, 4, 5)
Roll score: 100, Turn score: 100, Remaining dice: 4
You need at least 500 points to start accumulating points.
Press Enter to roll the dice.
Dice roll: (4, 3, 5, 3)
Roll score: 50, Turn score: 150, Remaining dice: 3
You need at least 500 points to start accumulating points.
Press Enter to roll the dice.
Dice roll: (1, 6, 4)
Roll score: 100, Turn score: 250, Remaining dice: 2
You need at least 500 points to start accumulating points.
Press Enter to roll the dice.
Dice roll: (4, 3)
FARKLE! You lose all points for this turn.
```



```
Test Player 2's turn!
Press Enter to roll the dice.
Dice roll: (3, 4, 3, 3, 4, 3)
Roll score: 1000, Turn score: 1000, Remaining dice: 2
Do you want to bank the points and end your turn (b), or roll the remaining dice (r)?
```



```
Test Player 2's turn!
Press Enter to roll the dice.
Dice roll: (6, 5, 3, 6, 5, 6)
Roll score: 700, Turn score: 700, Remaining dice: 1
Do you want to bank the points and end your turn (b), or roll the remaining dice (r)?
```



```
Test Player 1's turn!
Press Enter to roll the dice.
Dice roll: (1, 1, 6, 6, 2, 1)
Roll score: 1000, Turn score: 1000, Remaining dice: 3
Do you want to bank the points and end your turn (b), or roll the remaining dice
```

Ammad Raja - 20171842

Test Case ID: 5 **Pass**/Fail

Feature Tested: Banking Points and Continuing to Roll

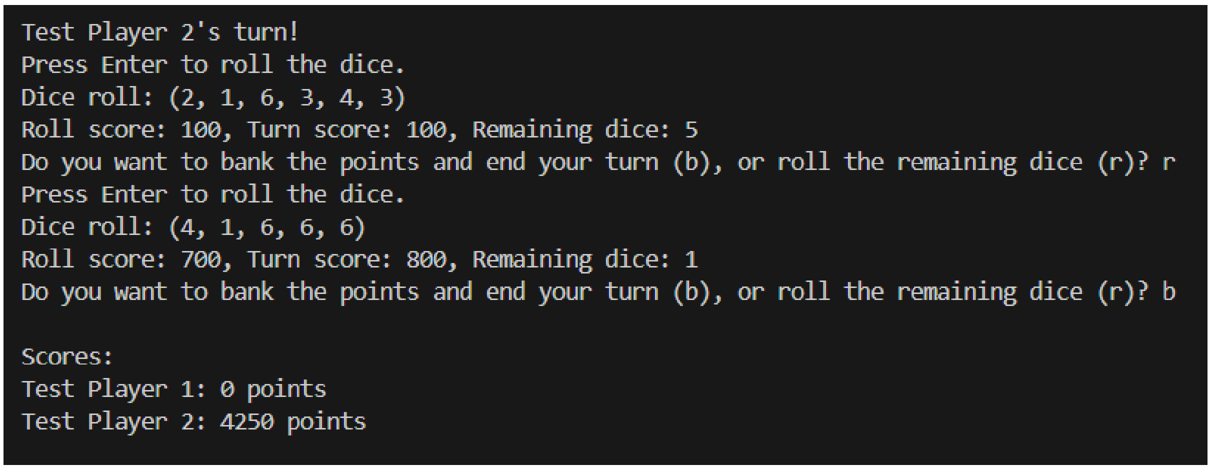
Steps

During a player's turn they can bank any points scored
Then that player can roll again if any dice are left
Points will be calculated again depending on new roll
The player can repeat the rolling process until they decide to bank the points or they don't score
The turn will end

Expected Output

After banking the scored points and rolling the remaining three dice, any new points or combinations should be added to the banked points. The player should be able to choose to bank the accumulated points or continue rolling the remaining dice. When the player is done with banking points or no dice are left to roll, the turn will end and the total score for each player will be shown.

Actual Output



```
Test Player 2's turn!
Press Enter to roll the dice.
Dice roll: (2, 1, 6, 3, 4, 3)
Roll score: 100, Turn score: 100, Remaining dice: 5
Do you want to bank the points and end your turn (b), or roll the remaining dice (r)? r
Press Enter to roll the dice.
Dice roll: (4, 1, 6, 6, 6)
Roll score: 700, Turn score: 800, Remaining dice: 1
Do you want to bank the points and end your turn (b), or roll the remaining dice (r)? b

Scores:
Test Player 1: 0 points
Test Player 2: 4250 points
```

Test Case ID: 6 **Pass**/Fail

Feature Tested: Rolling a Farkle

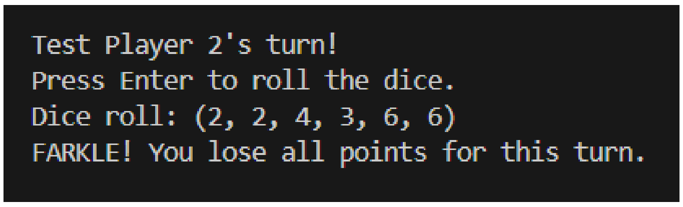
Steps

A player will roll the dice
They will not score any points
The turn will end

Expected Output

If a roll results in no points (a Farkle), the player is notified with a message like "FARKLE! You lose all points for this turn." The turn ends immediately, and no points are added to the total score for that turn.

Actual Output



```
Test Player 2's turn!
Press Enter to roll the dice.
Dice roll: (2, 2, 4, 3, 6, 6)
FARKLE! You lose all points for this turn.
```

Test Case ID: 7 **Pass**/Fail

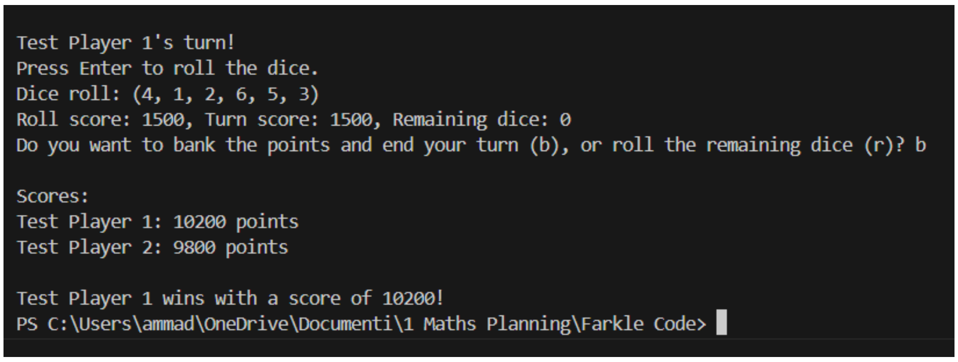
Feature Tested: Reaching the Winning Score

Steps
A player will roll the dice
They will score points which they will bank
Now that players points are over 10,000 or more
They win
The game ends

Expected Output

When a player's score reaches 10,000 points, they are declared the winner. The game ends, and a congratulatory message is displayed, such as "Test Player 1 wins with a score of 10,000! Congratulations!", indicating the successful completion of the game

Actual Output



```
Test Player 1's turn!
Press Enter to roll the dice.
Dice roll: (4, 1, 2, 6, 5, 3)
Roll score: 1500, Turn score: 1500, Remaining dice: 0
Do you want to bank the points and end your turn (b), or roll the remaining dice (r)? b

Scores:
Test Player 1: 10200 points
Test Player 2: 9800 points

Test Player 1 wins with a score of 10200!
PS C:\Users\ammad\OneDrive\Documenti\1 Maths Planning\Farkle Code>
```

Automated Testing

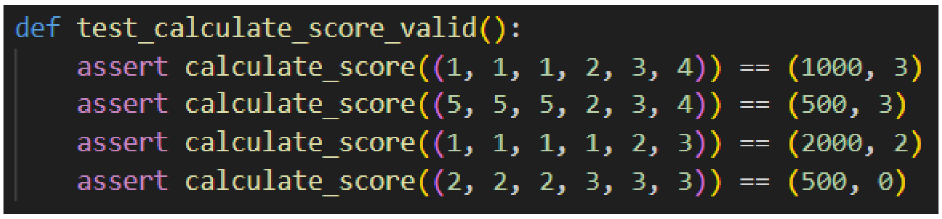
Is done by using scripts and tools to execute test cases automatically to ensure that each function behaves as expected and consistently produces the correct output.

Unit Testing

Unit tests automatically test programme components and functions to ensure they work as expected. They isolate each code section and test it separately to find issues. For the Farkle game, unit tests were written to verify individual functions, such as scoring logic and player turn management and more. I will use Pytest.

Test Name : 1 Test Calculate Score Valid

Code Snippet

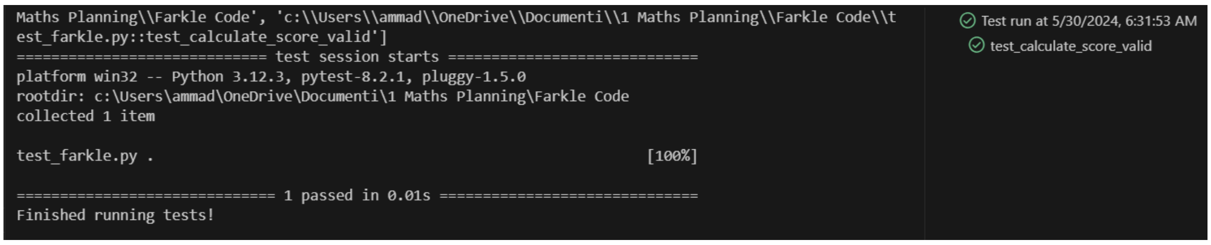


```python
def test_calculate_score_valid():
    assert calculate_score((1, 1, 1, 2, 3, 4)) == (1000, 3)
    assert calculate_score((5, 5, 5, 2, 3, 4)) == (500, 3)
    assert calculate_score((1, 1, 1, 1, 2, 3)) == (2000, 2)
    assert calculate_score((2, 2, 2, 3, 3, 3)) == (500, 0)
```

Explanation: Verifies the correct score and remaining dice for valid scoring combinations.

Justification: Ensures scoring logic is correct for common scenarios.

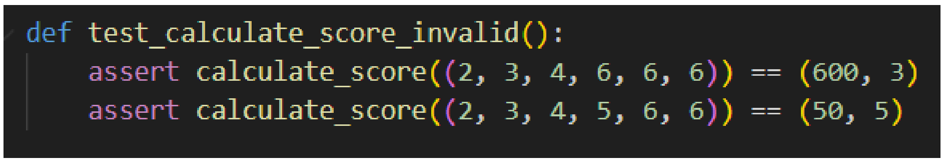
Returned Output



```
Maths Planning\\Farkle Code', 'c:\\Users\\ammad\\OneDrive\\Documenti\\1 Maths Planning\\Farkle Code\\t
est_farkle.py::test_calculate_score_valid']
=========================== test session starts ============================
platform win32 -- Python 3.12.3, pytest-8.2.1, pluggy-1.5.0
rootdir: c:\Users\ammad\OneDrive\Documenti\1 Maths Planning\Farkle Code
collected 1 item

test_farkle.py .                                                      [100%]

============================ 1 passed in 0.01s =============================
Finished running tests!
```

Test Name: 2 Test Calculate Score Invalid

Code Snippet

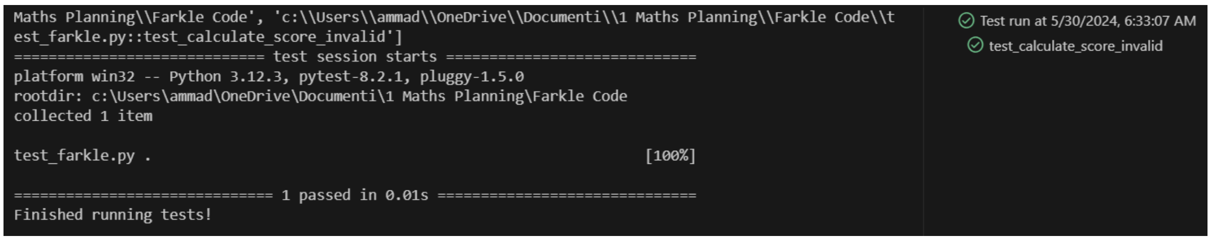


```python
def test_calculate_score_invalid():
    assert calculate_score((2, 3, 4, 6, 6, 6)) == (600, 3)
    assert calculate_score((2, 3, 4, 5, 6, 6)) == (50, 5)
```

Explanation: Verifies handling of non-scoring dice combinations.

Justification: Ensures game identifies and scores invalid combinations correctly.

Returned Output

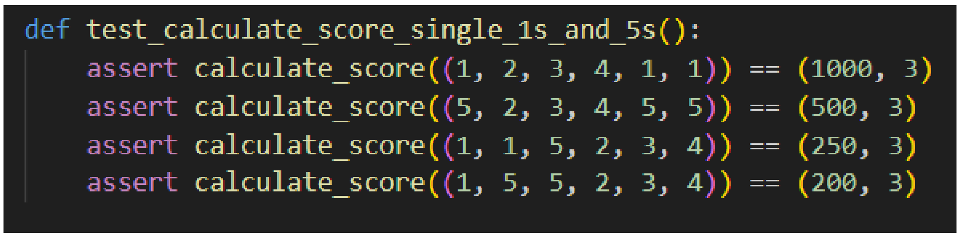


```
Maths Planning\\Farkle Code', 'c:\\Users\\ammad\\OneDrive\\Documenti\\1 Maths Planning\\Farkle Code\\t
est_farkle.py::test_calculate_score_invalid']
=========================== test session starts ============================
platform win32 -- Python 3.12.3, pytest-8.2.1, pluggy-1.5.0
rootdir: c:\Users\ammad\OneDrive\Documenti\1 Maths Planning\Farkle Code
collected 1 item

test_farkle.py .                                                      [100%]

============================ 1 passed in 0.01s =============================
Finished running tests!
```

Test run at 5/30/2024, 6:33:07 AM
test_calculate_score_invalid

Test Name: 3 Test Calculate Score Single 1s and 5s

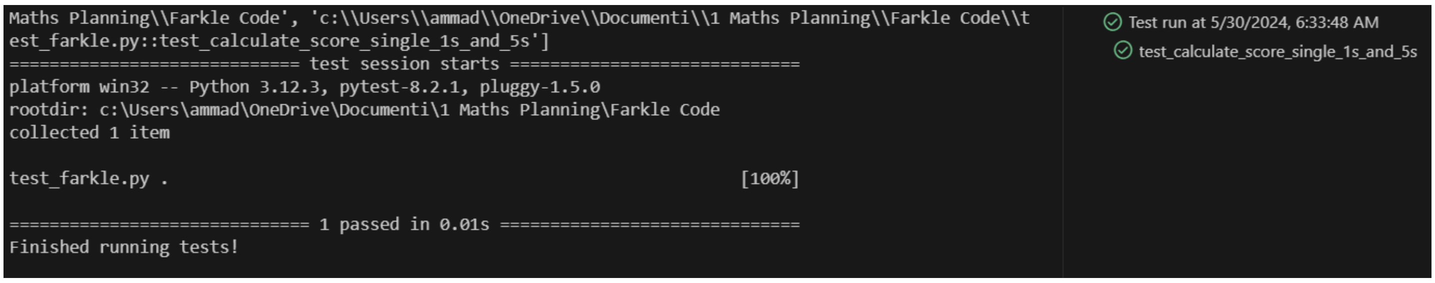
Code Snippet



```
def test_calculate_score_single_1s_and_5s():
    assert calculate_score((1, 2, 3, 4, 1, 1)) == (1000, 3)
    assert calculate_score((5, 2, 3, 4, 5, 5)) == (500, 3)
    assert calculate_score((1, 1, 5, 2, 3, 4)) == (250, 3)
    assert calculate_score((1, 5, 5, 2, 3, 4)) == (200, 3)
```

Explanation: Verifies that the calculate_score function correctly scores single 1s and 5s along with other dice.
Justification: Ensures accurate scoring for single high-value dice, which are common in gameplay.

Returned Output

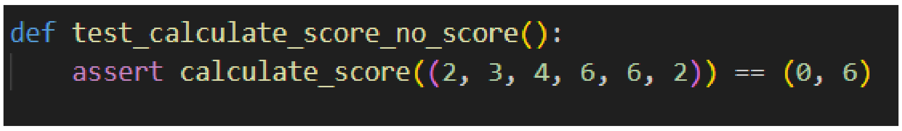


```
Maths Planning\\Farkle Code', 'c:\\Users\\ammad\\OneDrive\\Documenti\\1 Maths Planning\\Farkle Code\\t
est_farkle.py::test_calculate_score_single_1s_and_5s']
============================= test session starts =============================
platform win32 -- Python 3.12.3, pytest-8.2.1, pluggy-1.5.0
rootdir: c:\Users\ammad\OneDrive\Documenti\1 Maths Planning\Farkle Code
collected 1 item

test_farkle.py .                                                         [100%]

============================== 1 passed in 0.01s ==============================
Finished running tests!
```

✓ Test run at 5/30/2024, 6:33:48 AM
✓ test_calculate_score_single_1s_and_5s

Test Name: 4 Test Calculate Score No Score

Code Snippet

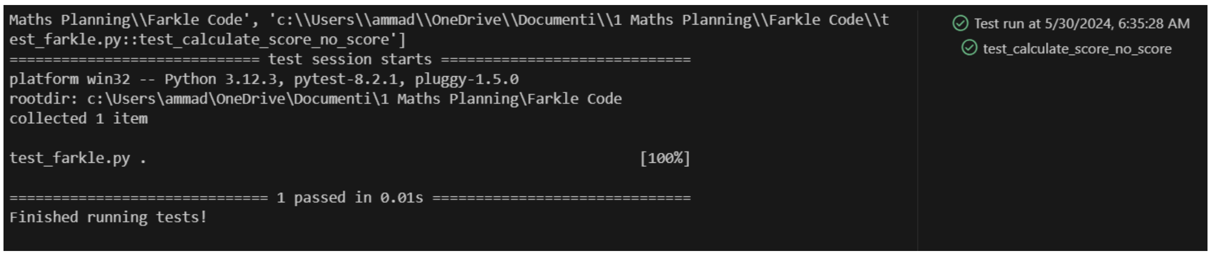


```
def test_calculate_score_no_score():
    assert calculate_score((2, 3, 4, 6, 6, 2)) == (0, 6)
```

Explanation: Verifies that no points are scored for non-scoring combinations.

Justification: Ensures the game correctly identifies and handles rolls with no scoring potential.

Returned Output

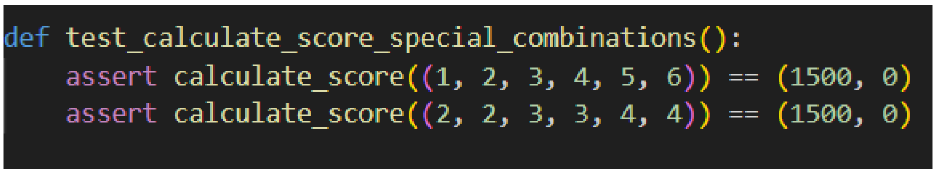


```
Maths Planning\\Farkle Code', 'c:\\Users\\ammad\\OneDrive\\Documenti\\1 Maths Planning\\Farkle Code\\t
est_farkle.py::test_calculate_score_no_score']
============================= test session starts =============================
platform win32 -- Python 3.12.3, pytest-8.2.1, pluggy-1.5.0
rootdir: c:\Users\ammad\OneDrive\Documenti\1 Maths Planning\Farkle Code
collected 1 item

test_farkle.py .                                                         [100%]

============================== 1 passed in 0.01s ==============================
Finished running tests!
```

✓ Test run at 5/30/2024, 6:35:28 AM
✓ test_calculate_score_no_score

Test Name: 5 Test Calculate Score Special Combinations

Code Snippet

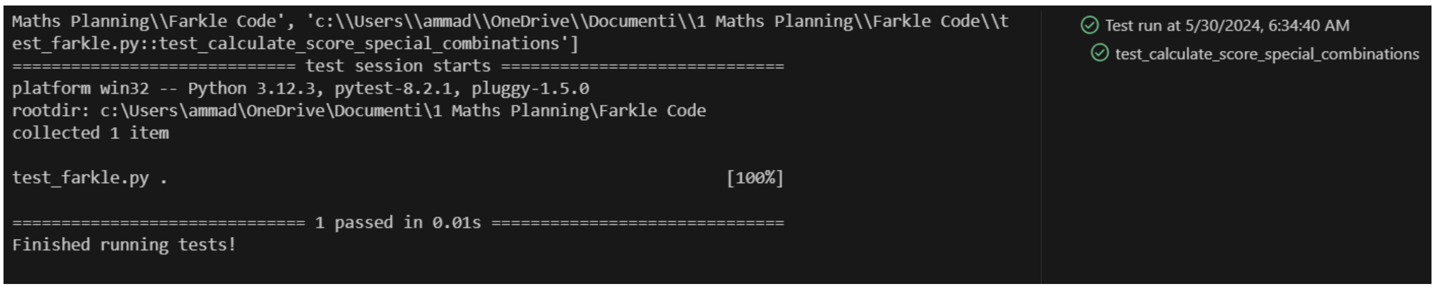


```python
def test_calculate_score_special_combinations():
    assert calculate_score((1, 2, 3, 4, 5, 6)) == (1500, 0)
    assert calculate_score((2, 2, 3, 3, 4, 4)) == (1500, 0)
```

Explanation: Verifies that special combinations like straights and three pairs are scored correctly.

Justification: Ensures the game correctly handles and scores these special combinations.

Returned Output

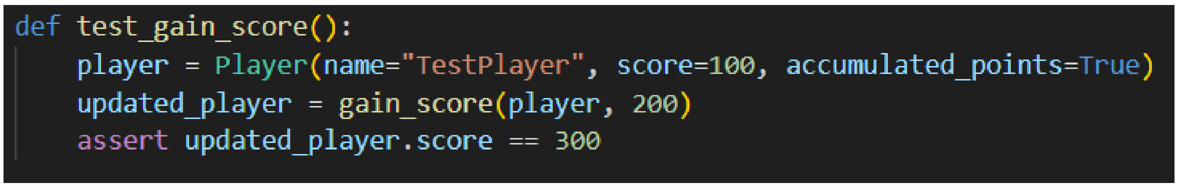


```
Maths Planning\\Farkle Code', 'c:\\Users\\ammad\\OneDrive\\Documenti\\1 Maths Planning\\Farkle Code\\t
est_farkle.py::test_calculate_score_special_combinations']
============================ test session starts ============================
platform win32 -- Python 3.12.3, pytest-8.2.1, pluggy-1.5.0
rootdir: c:\Users\ammad\OneDrive\Documenti\1 Maths Planning\Farkle Code
collected 1 item

test_farkle.py .                                                        [100%]

============================= 1 passed in 0.01s =============================
Finished running tests!
```

Test run at 5/30/2024, 6:34:40 AM
test_calculate_score_special_combinations

Test Name: 6 Test Gain Score

Code Snippet

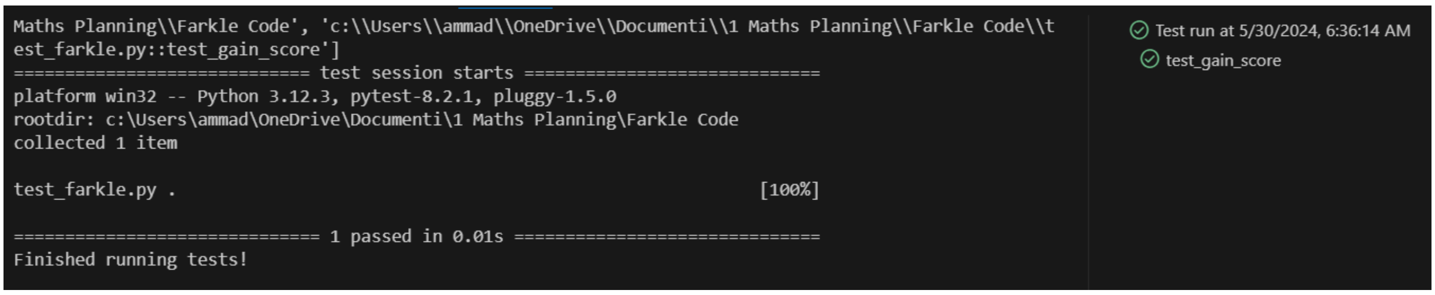


```python
def test_gain_score():
    player = Player(name="TestPlayer", score=100, accumulated_points=True)
    updated_player = gain_score(player, 200)
    assert updated_player.score == 300
```

Explanation: Verifies that the player's score is correctly updated when points are gained.

Justification: Ensures that the game correctly adds points to a player's score.

Returned Output

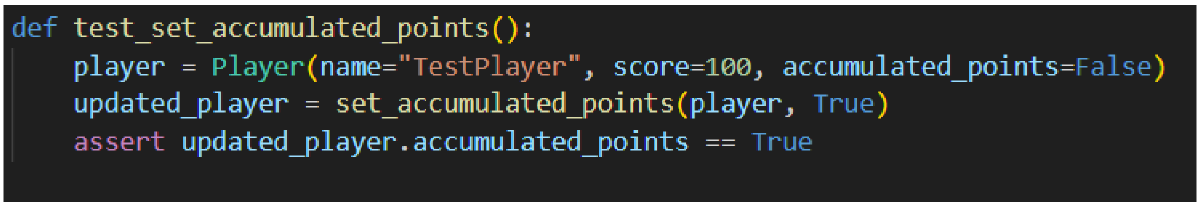


```
Maths Planning\\Farkle Code', 'c:\\Users\\ammad\\OneDrive\\Documenti\\1 Maths Planning\\Farkle Code\\t
est_farkle.py::test_gain_score']
============================ test session starts ============================
platform win32 -- Python 3.12.3, pytest-8.2.1, pluggy-1.5.0
rootdir: c:\Users\ammad\OneDrive\Documenti\1 Maths Planning\Farkle Code
collected 1 item

test_farkle.py .                                                        [100%]

============================= 1 passed in 0.01s =============================
Finished running tests!
```

Test run at 5/30/2024, 6:36:14 AM
test_gain_score

Test Name: 7 Test Set Accumulated Points

Code Snippet

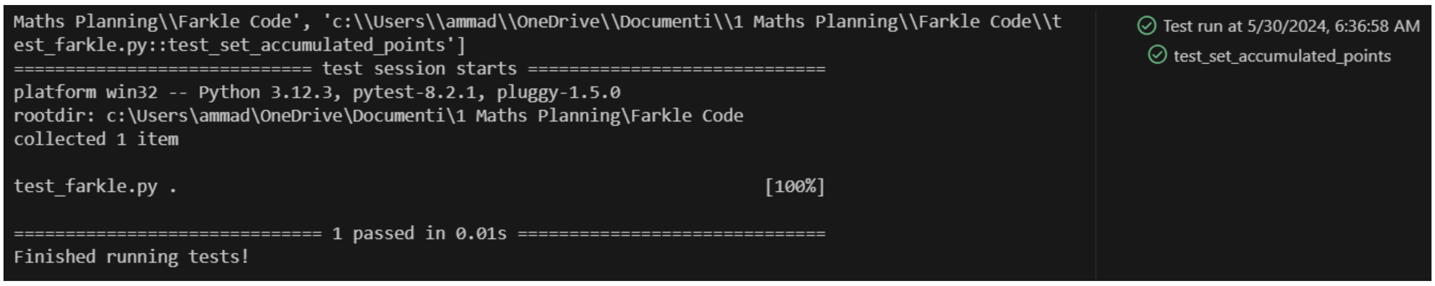


```python
def test_set_accumulated_points():
    player = Player(name="TestPlayer", score=100, accumulated_points=False)
    updated_player = set_accumulated_points(player, True)
    assert updated_player.accumulated_points == True
```

Explanation: Verifies that the accumulated points status is correctly updated.

Justification: Ensures the game correctly tracks whether a player has started accumulating points.

Returned Output

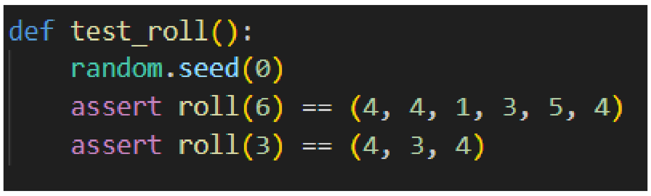


```
Maths Planning\\Farkle Code', 'c:\\Users\\ammad\\OneDrive\\Documenti\\1 Maths Planning\\Farkle Code\\t
est_farkle.py::test_set_accumulated_points']
=========================== test session starts ============================
platform win32 -- Python 3.12.3, pytest-8.2.1, pluggy-1.5.0
rootdir: c:\Users\ammad\OneDrive\Documenti\1 Maths Planning\Farkle Code
collected 1 item

test_farkle.py .                                                     [100%]

============================ 1 passed in 0.01s =============================
Finished running tests!
```

Test run at 5/30/2024, 6:36:58 AM
test_set_accumulated_points

Test Name: 8 Test Roll

Code Snippet

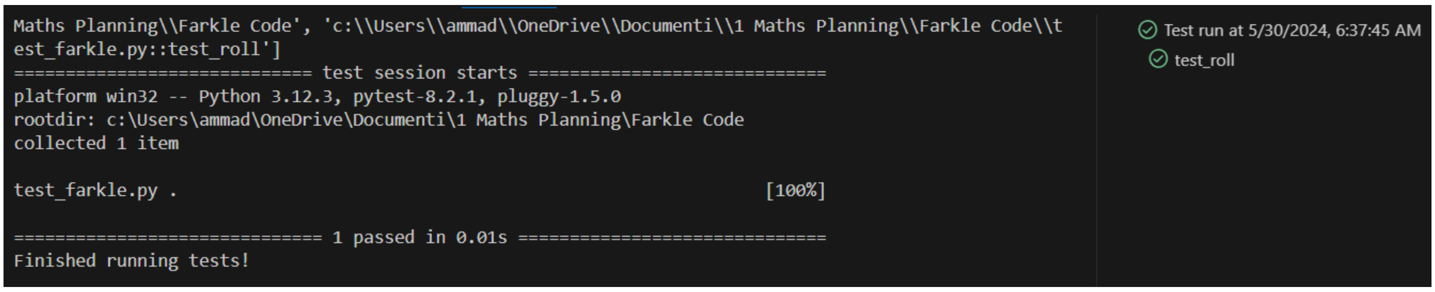


```python
def test_roll():
    random.seed(0)
    assert roll(6) == (4, 4, 1, 3, 5, 4)
    assert roll(3) == (4, 3, 4)
```

Explanation: Verifies that the roll function generates the expected dice values.

Justification: Ensures the consistency and predictability of dice rolls during testing.

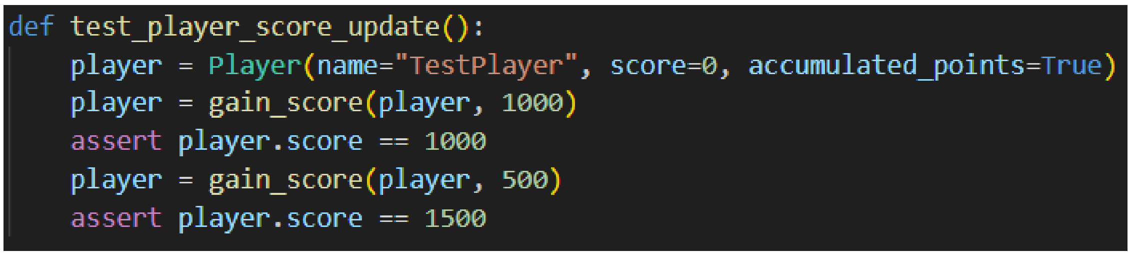
Returned Output



```
Maths Planning\\Farkle Code', 'c:\\Users\\ammad\\OneDrive\\Documenti\\1 Maths Planning\\Farkle Code\\t
est_farkle.py::test_roll']
=========================== test session starts ============================
platform win32 -- Python 3.12.3, pytest-8.2.1, pluggy-1.5.0
rootdir: c:\Users\ammad\OneDrive\Documenti\1 Maths Planning\Farkle Code
collected 1 item

test_farkle.py .                                                     [100%]

============================ 1 passed in 0.01s =============================
Finished running tests!
```

Test Name: 9 Test Player Score Update

Code Snippet

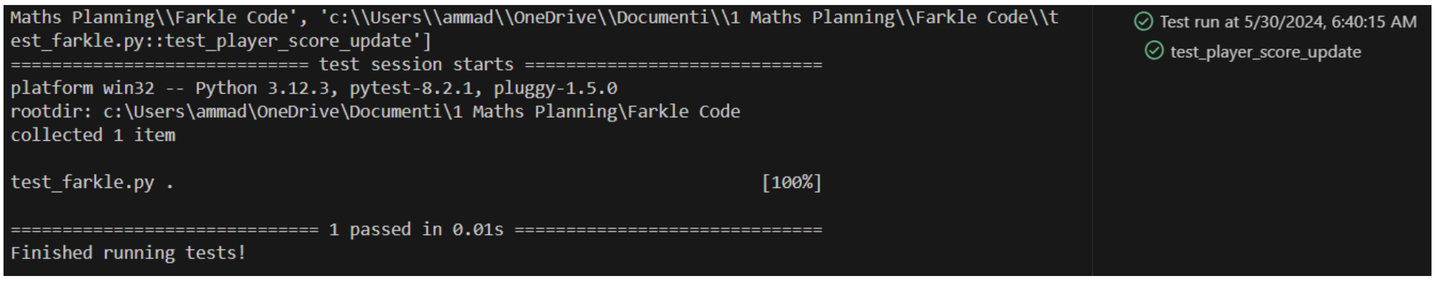


```python
def test_player_score_update():
    player = Player(name="TestPlayer", score=0, accumulated_points=True)
    player = gain_score(player, 1000)
    assert player.score == 1000
    player = gain_score(player, 500)
    assert player.score == 1500
```

Explanation: Verifies that the player's score is correctly updated multiple times.

Justification: Ensures the game correctly tracks score updates over multiple turns.

Returned Output

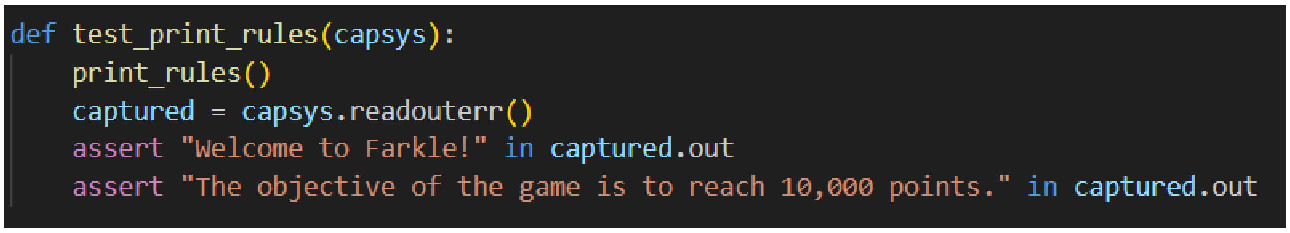


```
Maths Planning\\Farkle Code', 'c:\\Users\\ammad\\OneDrive\\Documenti\\1 Maths Planning\\Farkle Code\\t
est_farkle.py::test_player_score_update']
============================= test session starts =============================
platform win32 -- Python 3.12.3, pytest-8.2.1, pluggy-1.5.0
rootdir: c:\Users\ammad\OneDrive\Documenti\1 Maths Planning\Farkle Code
collected 1 item

test_farkle.py .                                                         [100%]

============================== 1 passed in 0.01s ==============================
Finished running tests!
```

✓ Test run at 5/30/2024, 6:40:15 AM
✓ test_player_score_update

Test Name:10 Test Print Rules

Code Snippet

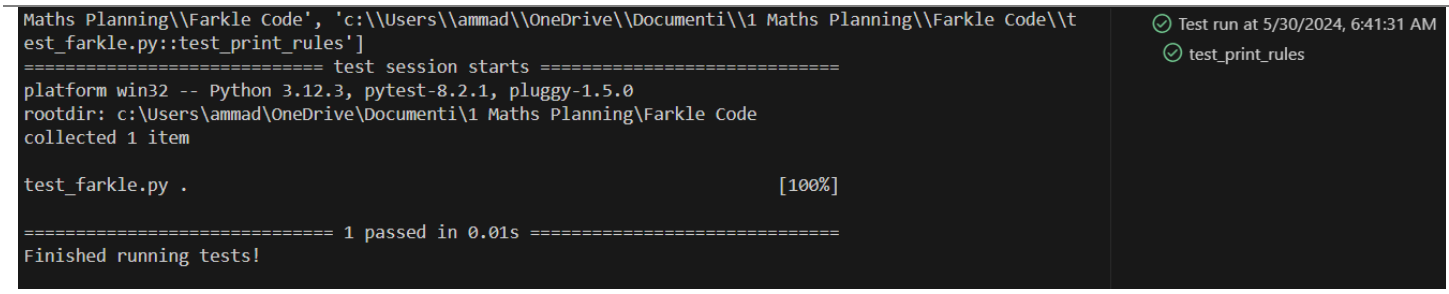


```python
def test_print_rules(capsys):
    print_rules()
    captured = capsys.readouterr()
    assert "Welcome to Farkle!" in captured.out
    assert "The objective of the game is to reach 10,000 points." in captured.out
```

Explanation: Verifies that the rules are printed correctly.

Justification: Ensures that players receive the correct game instructions.

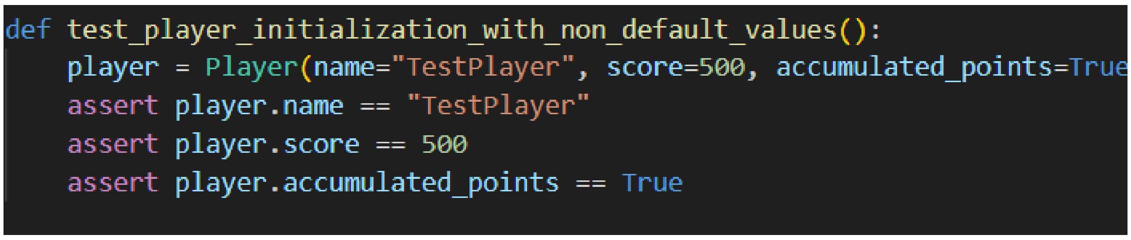
Returned Output



```
Maths Planning\\Farkle Code', 'c:\\Users\\ammad\\OneDrive\\Documenti\\1 Maths Planning\\Farkle Code\\t
est_farkle.py::test_print_rules']
============================= test session starts ==============================
platform win32 -- Python 3.12.3, pytest-8.2.1, pluggy-1.5.0
rootdir: c:\Users\ammad\OneDrive\Documenti\1 Maths Planning\Farkle Code
collected 1 item

test_farkle.py .                                                         [100%]

============================== 1 passed in 0.01s ===============================
Finished running tests!
```

Test Name: 11 Test Player Initialization with Non-Default Values

Code Snippet

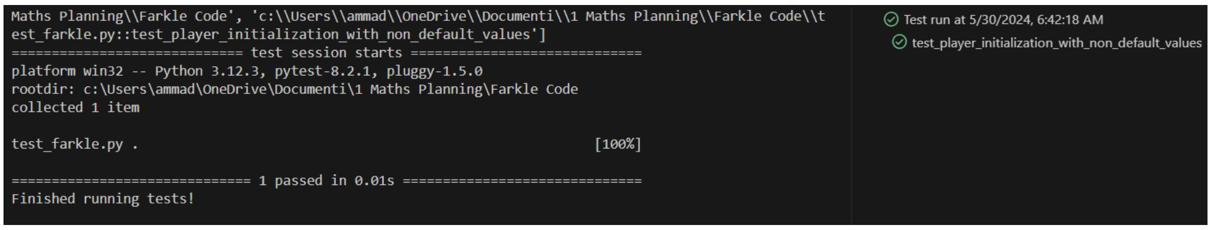


```python
def test_player_initialization_with_non_default_values():
    player = Player(name="TestPlayer", score=500, accumulated_points=True
    assert player.name == "TestPlayer"
    assert player.score == 500
    assert player.accumulated_points == True
```

Explanation: Verifies that a Player object is correctly initialized with non-default values.

Justification: Ensures that player initialization works correctly with provided values.

Returned Output

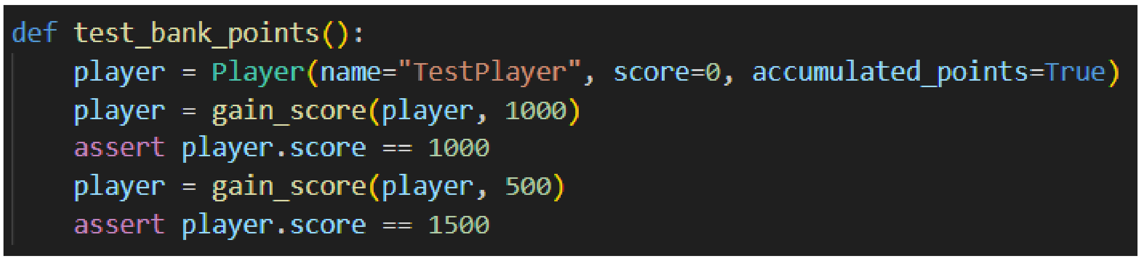


```
Maths Planning\\Farkle Code', 'c:\\Users\\ammad\\OneDrive\\Documenti\\1 Maths Planning\\Farkle Code\\t
est_farkle.py::test_player_initialization_with_non_default_values']
============================= test session starts ==============================
platform win32 -- Python 3.12.3, pytest-8.2.1, pluggy-1.5.0
rootdir: c:\Users\ammad\OneDrive\Documenti\1 Maths Planning\Farkle Code
collected 1 item

test_farkle.py .                                                         [100%]

============================== 1 passed in 0.01s ===============================
Finished running tests!
```

✓ Test run at 5/30/2024, 6:42:18 AM
✓ test_player_initialization_with_non_default_values

Test Name: 12 Test Bank Points

Code Snippet

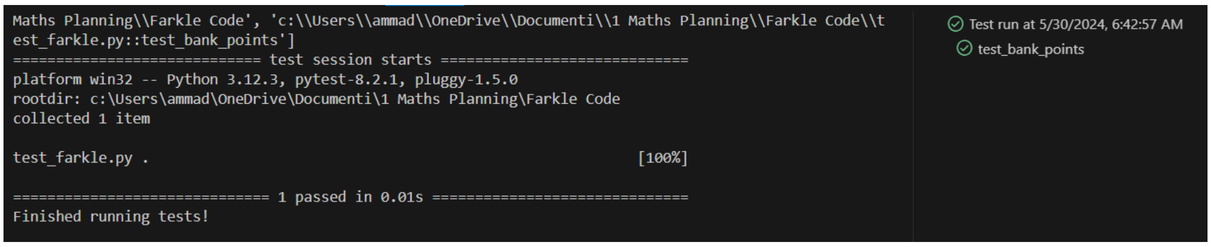


```python
def test_bank_points():
    player = Player(name="TestPlayer", score=0, accumulated_points=True)
    player = gain_score(player, 1000)
    assert player.score == 1000
    player = gain_score(player, 500)
    assert player.score == 1500
```

Explanation: Verifies that points are correctly banked and the player's score is updated.

Justification: Ensures that banking points functionality is correctly implemented.

Returned Output



```
Maths Planning\\Farkle Code', 'c:\\Users\\ammad\\OneDrive\\Documenti\\1 Maths Planning\\Farkle Code\\t
est_farkle.py::test_bank_points']
=========================== test session starts ============================
platform win32 -- Python 3.12.3, pytest-8.2.1, pluggy-1.5.0
rootdir: c:\Users\ammad\OneDrive\Documenti\1 Maths Planning\Farkle Code
collected 1 item

test_farkle.py .                                                      [100%]

============================ 1 passed in 0.01s =============================
Finished running tests!
```

Test run at 5/30/2024, 6:42:57 AM
test_bank_points

Test Name: 13 Test Farkle Handling

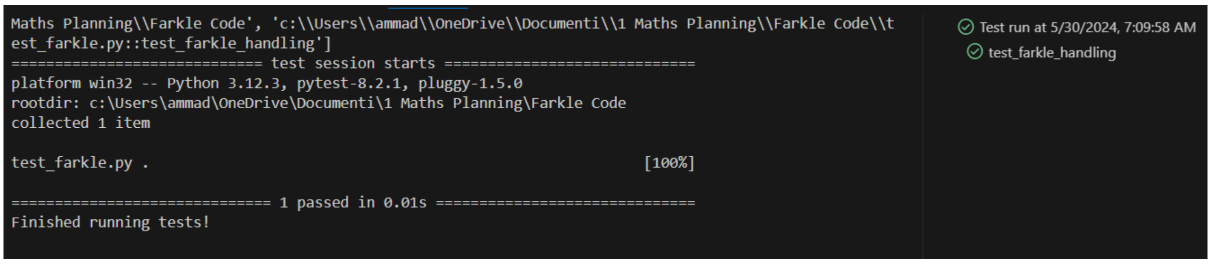
Code Snippet

```python
def test_farkle_handling():
    player = Player(name="TestPlayer", score=1500, accumulated_points=True)
    roll_score, remaining_dice = calculate_score((2, 3, 4, 6, 6, 2))
    assert roll_score == 0
    assert remaining_dice == 6
```

Explanation: Ensures that a turn ends with no points added if a Farkle is rolled.

Justification: Verifies that Farkle scenarios are correctly handled by the game logic.

Returned Output

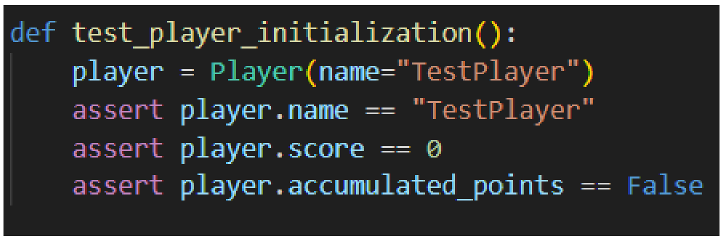


```
Maths Planning\\Farkle Code', 'c:\\Users\\ammad\\OneDrive\\Documenti\\1 Maths Planning\\Farkle Code\\t
est_farkle.py::test_farkle_handling']
=========================== test session starts ============================
platform win32 -- Python 3.12.3, pytest-8.2.1, pluggy-1.5.0
rootdir: c:\Users\ammad\OneDrive\Documenti\1 Maths Planning\Farkle Code
collected 1 item

test_farkle.py .                                                      [100%]

============================ 1 passed in 0.01s =============================
Finished running tests!
```

Test run at 5/30/2024, 7:09:58 AM
test_farkle_handling

Test Name: 14 Test Player Initialization

Code Snippet

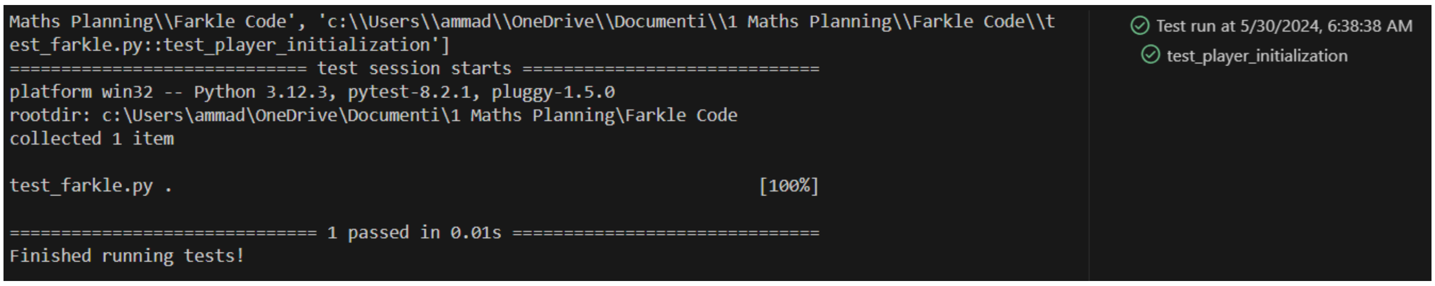


```python
def test_player_initialization():
    player = Player(name="TestPlayer")
    assert player.name == "TestPlayer"
    assert player.score == 0
    assert player.accumulated_points == False
```

Explanation: Verifies that a Player object is correctly initialized with default values.

Justification: Ensures that new players start with the correct initial state.

Returned Output

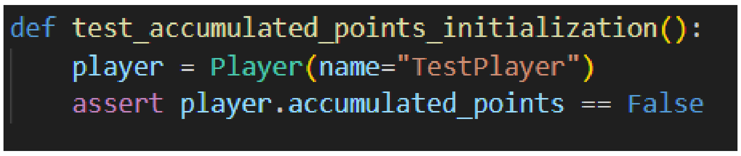


```
Maths Planning\\Farkle Code', 'c:\\Users\\ammad\\OneDrive\\Documenti\\1 Maths Planning\\Farkle Code\\t
est_farkle.py::test_player_initialization']
============================ test session starts ============================
platform win32 -- Python 3.12.3, pytest-8.2.1, pluggy-1.5.0
rootdir: c:\Users\ammad\OneDrive\Documenti\1 Maths Planning\Farkle Code
collected 1 item

test_farkle.py .                                                       [100%]

============================= 1 passed in 0.01s =============================
Finished running tests!
```

✓ Test run at 5/30/2024, 6:38:38 AM
✓ test_player_initialization

Test Name:15 Test Accumulated Points Initialization

Code Snippet

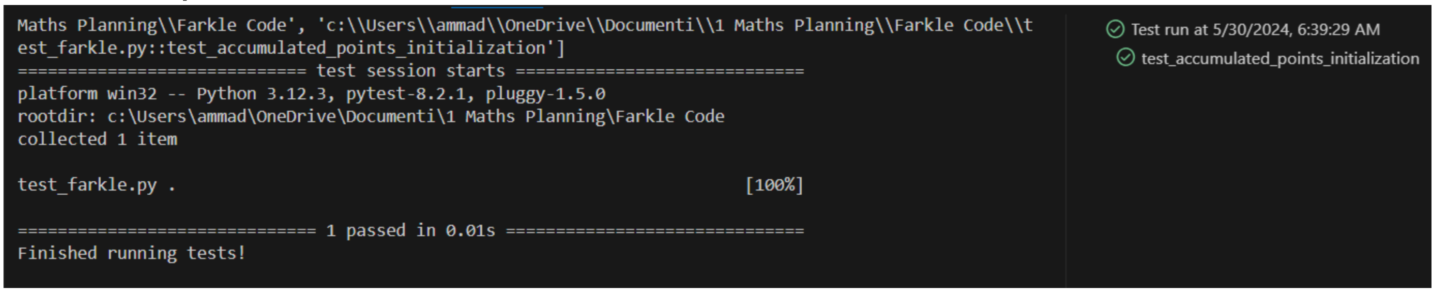


```python
def test_accumulated_points_initialization():
    player = Player(name="TestPlayer")
    assert player.accumulated_points == False
```

Explanation: Verifies that the accumulated points attribute initializes correctly.

Justification: Ensures that new players do not start with accumulated points.

Returned Output



```
Maths Planning\\Farkle Code', 'c:\\Users\\ammad\\OneDrive\\Documenti\\1 Maths Planning\\Farkle Code\\t
est_farkle.py::test_accumulated_points_initialization']
============================ test session starts ============================
platform win32 -- Python 3.12.3, pytest-8.2.1, pluggy-1.5.0
rootdir: c:\Users\ammad\OneDrive\Documenti\1 Maths Planning\Farkle Code
collected 1 item

test_farkle.py .                                                       [100%]

============================= 1 passed in 0.01s =============================
Finished running tests!
```

✓ Test run at 5/30/2024, 6:39:29 AM
✓ test_accumulated_points_initialization

BDD Testing

Overview

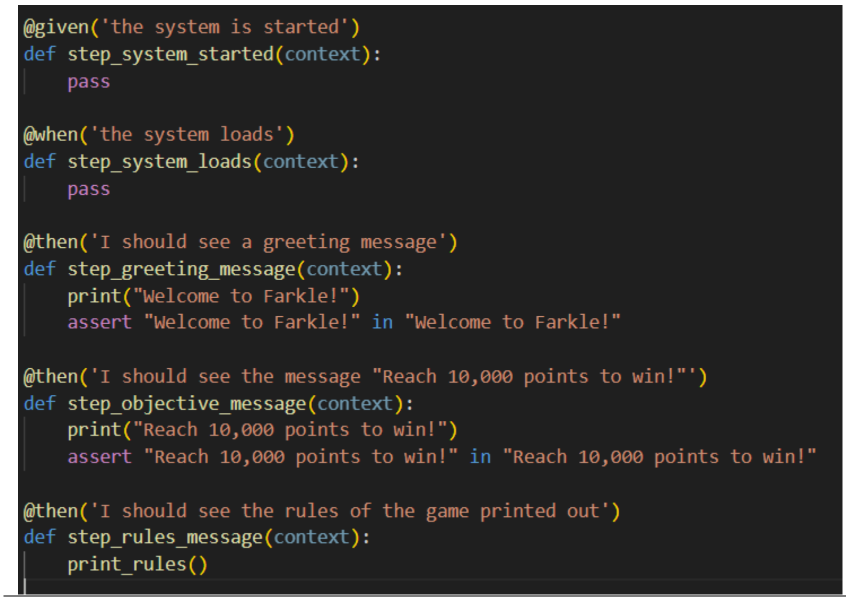
Writing scenarios in Gherkin syntax describes the game's expected behaviour in different situations. These scenarios are then executed using a tool like Behave to ensure the game meets the specified requirements.

Installing behave requires having pip and python already installed, in a command prompt or terminal run **pip install behave.** To use Behave, navigate to your project directory and create a features folder containing **your .feature** files and **steps** implementations containing code for testing. Type **Behave** in command prompt to run it, it will execute all the features.

Test System Greeting and Rules Display

Verifies that the game initializes properly and informs the player about the game's objectives and rules.

Code Snippet and Scenario Feature



```python
@given('the system is started')
def step_system_started(context):
    pass

@when('the system loads')
def step_system_loads(context):
    pass

@then('I should see a greeting message')
def step_greeting_message(context):
    print("Welcome to Farkle!")
    assert "Welcome to Farkle!" in "Welcome to Farkle!"

@then('I should see the message "Reach 10,000 points to win!"')
def step_objective_message(context):
    print("Reach 10,000 points to win!")
    assert "Reach 10,000 points to win!" in "Reach 10,000 points to win!"

@then('I should see the rules of the game printed out')
def step_rules_message(context):
    print_rules()
```

Scenario: System Loads and Greets the User

Given the system is started

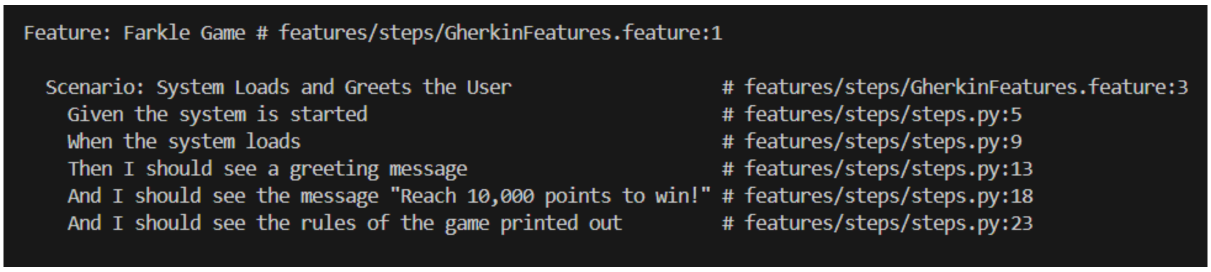
When the system loads

Then I should see a greeting message

And I should see the message "Reach 10,000 points to win!"

And I should see the rules of the game printed out

Result

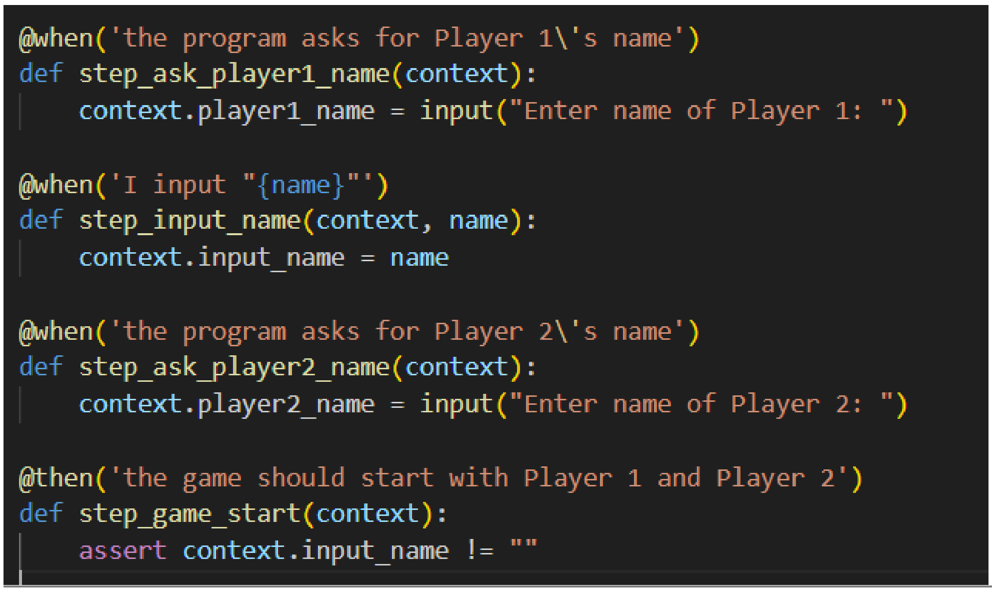


```
Feature: Farkle Game # features/steps/GherkinFeatures.feature:1

  Scenario: System Loads and Greets the User                        # features/steps/GherkinFeatures.feature:3
    Given the system is started                                     # features/steps/steps.py:5
    When the system loads                                           # features/steps/steps.py:9
    Then I should see a greeting message                            # features/steps/steps.py:13
    And I should see the message "Reach 10,000 points to win!" # features/steps/steps.py:18
    And I should see the rules of the game printed out              # features/steps/steps.py:23
```

Test Inputting Player Names

Ensures that the game correctly handles player name input and initiates the game with two players.

Code Snippet and Scenario Feature



```python
@when('the program asks for Player 1\'s name')
def step_ask_player1_name(context):
    context.player1_name = input("Enter name of Player 1: ")

@when('I input "{name}"')
def step_input_name(context, name):
    context.input_name = name

@when('the program asks for Player 2\'s name')
def step_ask_player2_name(context):
    context.player2_name = input("Enter name of Player 2: ")

@then('the game should start with Player 1 and Player 2')
def step_game_start(context):
    assert context.input_name != ""
```

Scenario: Inputting Player Names

When the program asks for Player 1's name

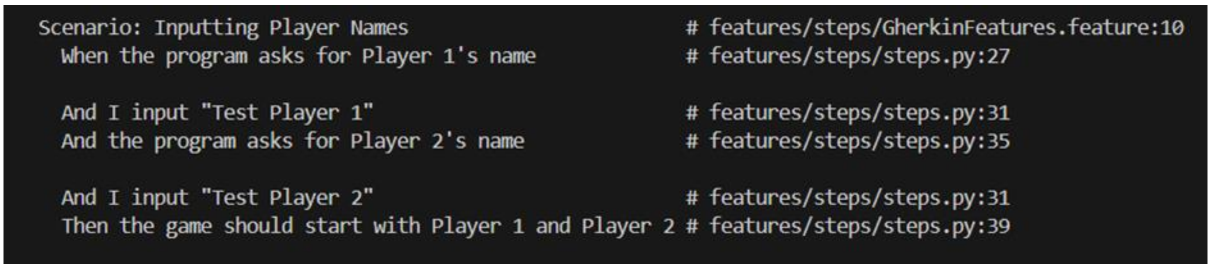
And I input "Test Player 1"

And the program asks for Player 2's name

And I input "Test Player 2"

Then the game should start with Player 1 and Player 2

Result

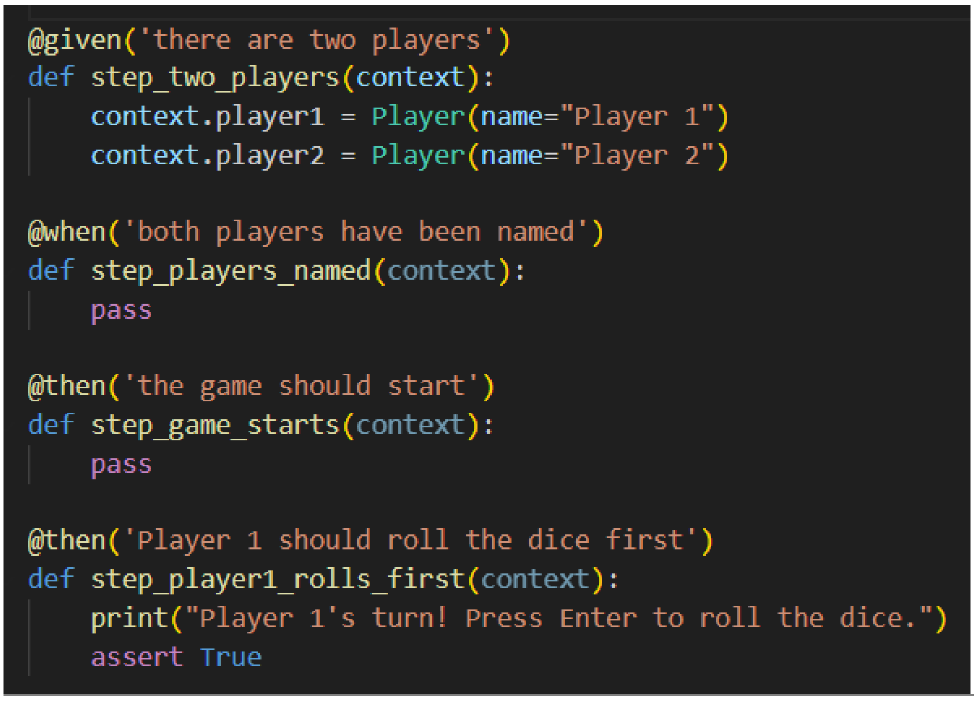


```
Scenario: Inputting Player Names                          # features/steps/GherkinFeatures.feature:10
  When the program asks for Player 1's name              # features/steps/steps.py:27

  And I input "Test Player 1"                            # features/steps/steps.py:31
  And the program asks for Player 2's name               # features/steps/steps.py:35

  And I input "Test Player 2"                            # features/steps/steps.py:31
  Then the game should start with Player 1 and Player 2 # features/steps/steps.py:39
```

Test Game Start

Verifies that the game initializes correctly with both players and starts with Player 1 rolling the dice

Code Snippet and Scenario Feature



```python
@given('there are two players')
def step_two_players(context):
    context.player1 = Player(name="Player 1")
    context.player2 = Player(name="Player 2")

@when('both players have been named')
def step_players_named(context):
    pass

@then('the game should start')
def step_game_starts(context):
    pass

@then('Player 1 should roll the dice first')
def step_player1_rolls_first(context):
    print("Player 1's turn! Press Enter to roll the dice.")
    assert True
```

Scenario: Game Start

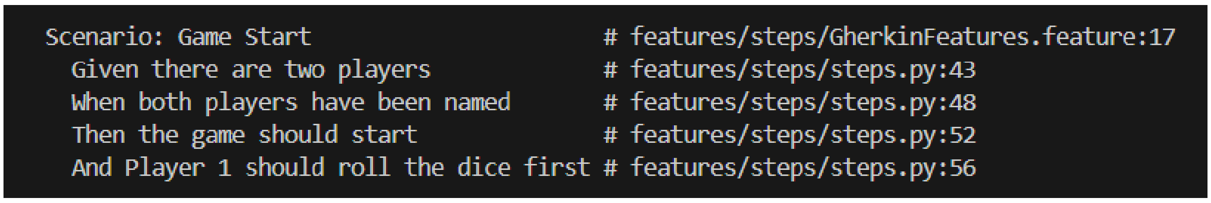
Given there are two players

When both players have been named

Then the game should start

And Player 1 should roll the dice first

Result

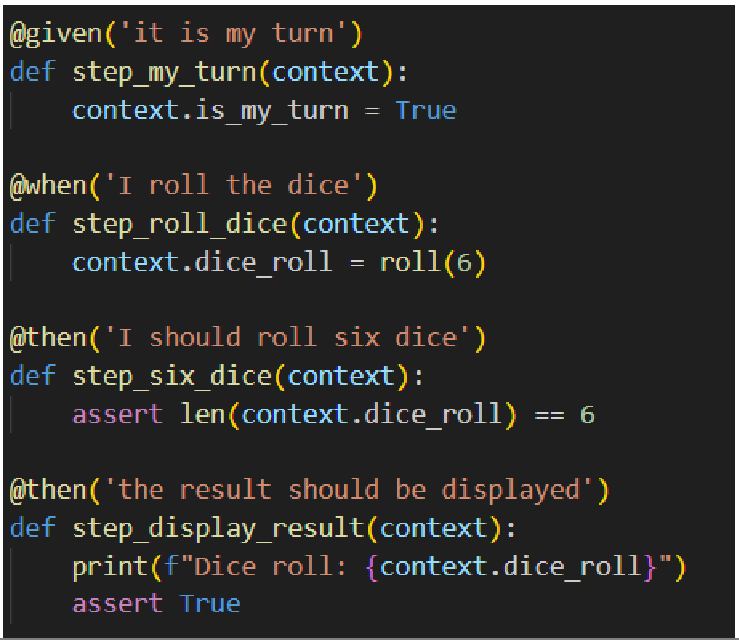


```
Scenario: Game Start                      # features/steps/GherkinFeatures.feature:17
  Given there are two players             # features/steps/steps.py:43
  When both players have been named       # features/steps/steps.py:48
  Then the game should start              # features/steps/steps.py:52
  And Player 1 should roll the dice first # features/steps/steps.py:56
```

Test Initial Roll

Verifies that the player rolls the correct number of dice and the results are displayed.

Code Snippet and Scenario Feature



```python
@given('it is my turn')
def step_my_turn(context):
    context.is_my_turn = True

@when('I roll the dice')
def step_roll_dice(context):
    context.dice_roll = roll(6)

@then('I should roll six dice')
def step_six_dice(context):
    assert len(context.dice_roll) == 6

@then('the result should be displayed')
def step_display_result(context):
    print(f"Dice roll: {context.dice_roll}")
    assert True
```

Scenario: Initial Roll

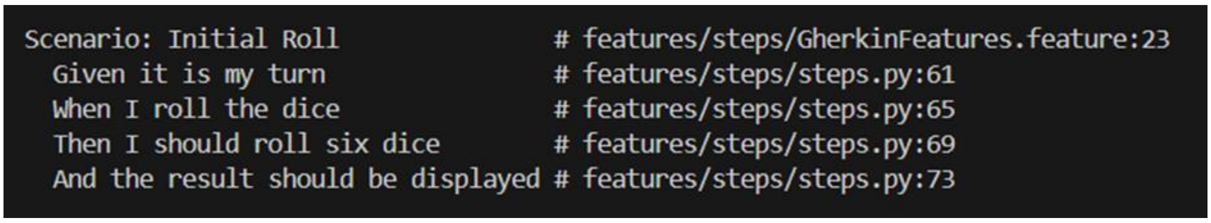
Given it is my turn

When I roll the dice

Then I should roll six dice

And the result should be displayed

Result

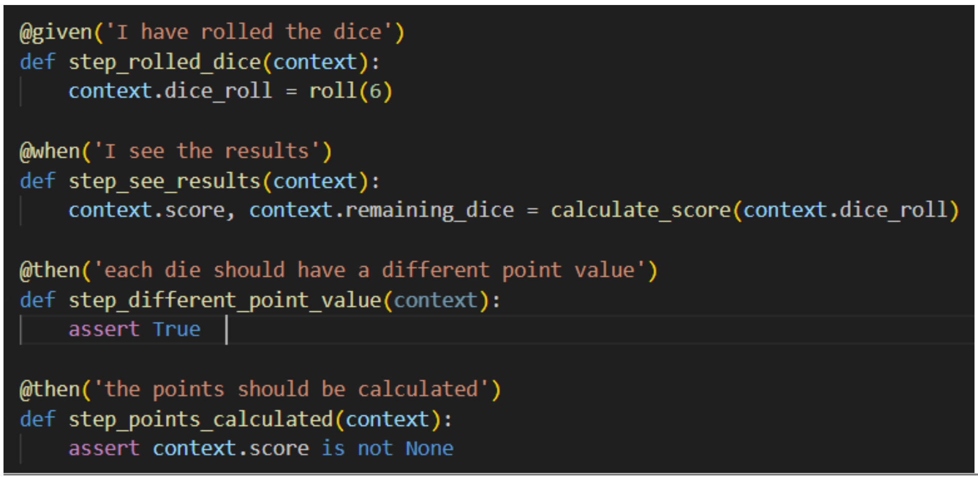


```
Scenario: Initial Roll                  # features/steps/GherkinFeatures.feature:23
  Given it is my turn                   # features/steps/steps.py:61
  When I roll the dice                  # features/steps/steps.py:65
  Then I should roll six dice           # features/steps/steps.py:69
  And the result should be displayed # features/steps/steps.py:73
```

Test Scoring System

Verifies that the scoring system calculates the points correctly for each die

Code Snippet and Scenario Feature



```python
@given('I have rolled the dice')
def step_rolled_dice(context):
    context.dice_roll = roll(6)

@when('I see the results')
def step_see_results(context):
    context.score, context.remaining_dice = calculate_score(context.dice_roll)

@then('each die should have a different point value')
def step_different_point_value(context):
    assert True

@then('the points should be calculated')
def step_points_calculated(context):
    assert context.score is not None
```

Scenario: Understanding the Scoring System

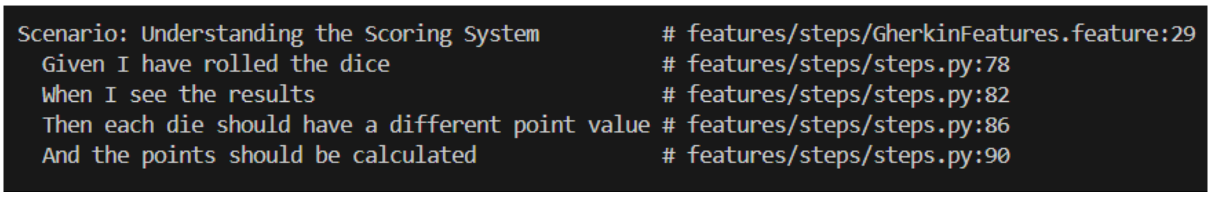
Given I have rolled the dice

When I see the results

Then each die should have a different point value

And the points should be calculated

Result

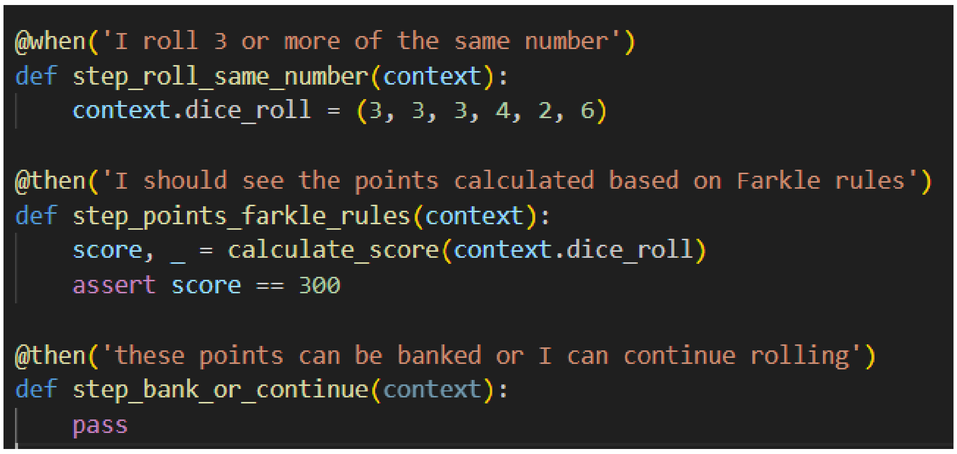


```
Scenario: Understanding the Scoring System          # features/steps/GherkinFeatures.feature:29
  Given I have rolled the dice                      # features/steps/steps.py:78
  When I see the results                            # features/steps/steps.py:82
  Then each die should have a different point value # features/steps/steps.py:86
  And the points should be calculated               # features/steps/steps.py:90
```

Test Triplets and Higher Combinations

Ensures that the game calculates points correctly for triplets and higher combinations

Code Snippet and Scenario Feature



```
@when('I roll 3 or more of the same number')
def step_roll_same_number(context):
    context.dice_roll = (3, 3, 3, 4, 2, 6)

@then('I should see the points calculated based on Farkle rules')
def step_points_farkle_rules(context):
    score, _ = calculate_score(context.dice_roll)
    assert score == 300

@then('these points can be banked or I can continue rolling')
def step_bank_or_continue(context):
    pass
```

Scenario: Triplets and Higher Combinations

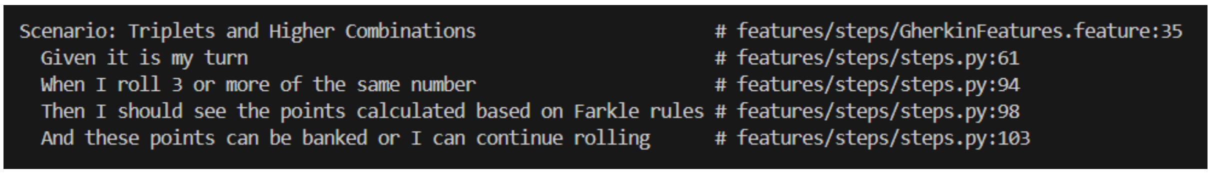
Given it is my turn

When I roll 3 or more of the same number

Then I should see the points calculated based on Farkle rules

And these points can be banked or I can continue rolling

Result

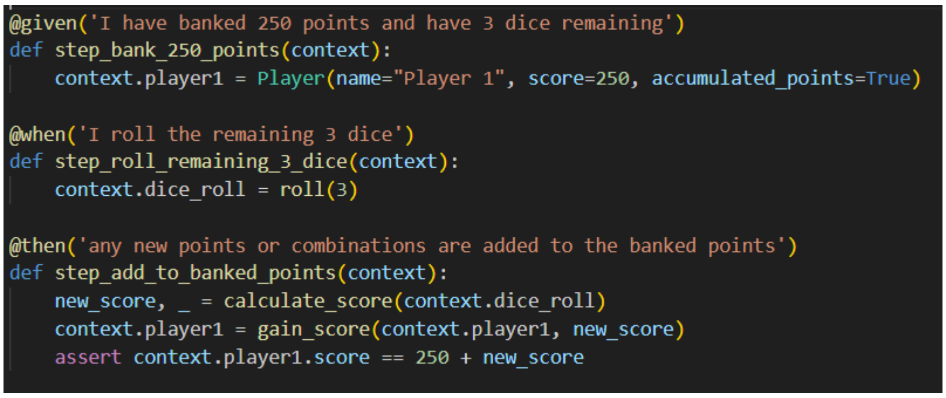


```
Scenario: Triplets and Higher Combinations                     # features/steps/GherkinFeatures.feature:35
  Given it is my turn                                          # features/steps/steps.py:61
  When I roll 3 or more of the same number                     # features/steps/steps.py:94
  Then I should see the points calculated based on Farkle rules # features/steps/steps.py:98
  And these points can be banked or I can continue rolling     # features/steps/steps.py:103
```

Test Banking Points and Continuing to Roll

Ensures that new points are correctly added to the previously banked points when rolling remaining dice.

Code Snippet and Scenario Feature



```
@given('I have banked 250 points and have 3 dice remaining')
def step_bank_250_points(context):
    context.player1 = Player(name="Player 1", score=250, accumulated_points=True)

@when('I roll the remaining 3 dice')
def step_roll_remaining_3_dice(context):
    context.dice_roll = roll(3)

@then('any new points or combinations are added to the banked points')
def step_add_to_banked_points(context):
    new_score, _ = calculate_score(context.dice_roll)
    context.player1 = gain_score(context.player1, new_score)
    assert context.player1.score == 250 + new_score
```

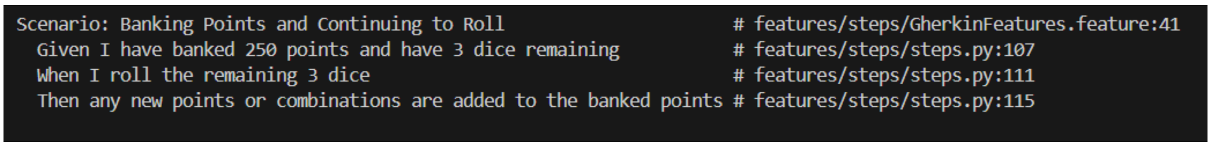
Scenario: Banking Points and Continuing to Roll

Given I have banked 250 points and have 3 dice remaining

When I roll the remaining 3 dice

Then any new points or combinations are added to the banked points

Result

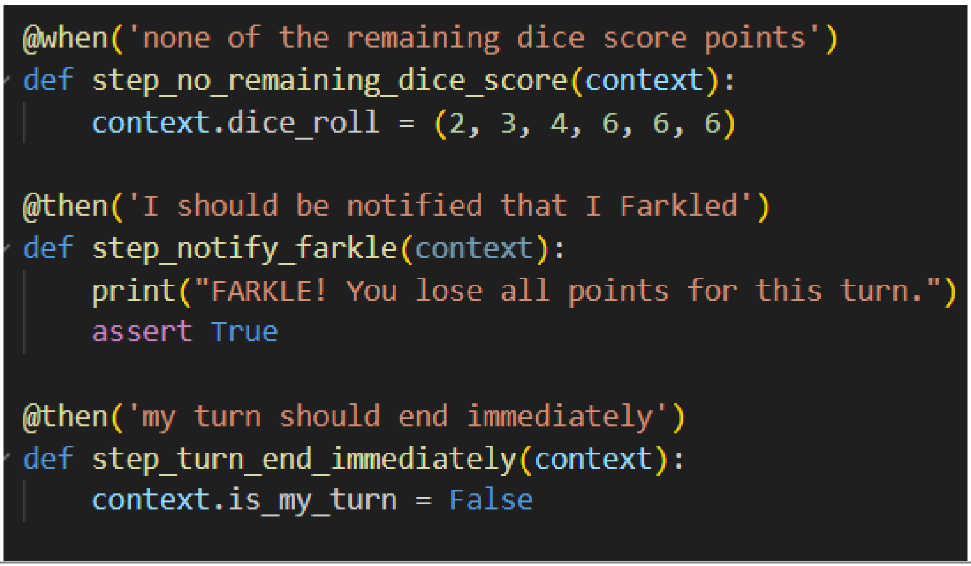


```
Scenario: Banking Points and Continuing to Roll                          # features/steps/GherkinFeatures.feature:41
  Given I have banked 250 points and have 3 dice remaining               # features/steps/steps.py:107
  When I roll the remaining 3 dice                                       # features/steps/steps.py:111
  Then any new points or combinations are added to the banked points     # features/steps/steps.py:115
```

Test Rolling a Farkle

Verifies that Farkle scenarios are correctly handled by the game logic

Code Snippet and Scenario Feature



```python
@when('none of the remaining dice score points')
def step_no_remaining_dice_score(context):
    context.dice_roll = (2, 3, 4, 6, 6, 6)

@then('I should be notified that I Farkled')
def step_notify_farkle(context):
    print("FARKLE! You lose all points for this turn.")
    assert True

@then('my turn should end immediately')
def step_turn_end_immediately(context):
    context.is_my_turn = False
```

Scenario: Rolling a Farkle

Given it is my turn

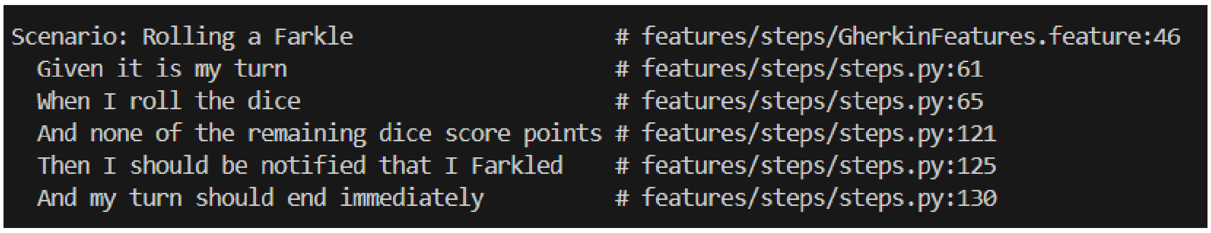
When I roll the dice

And none of the remaining dice score points

Then I should be notified that I Farkled

And my turn should end immediately

Result

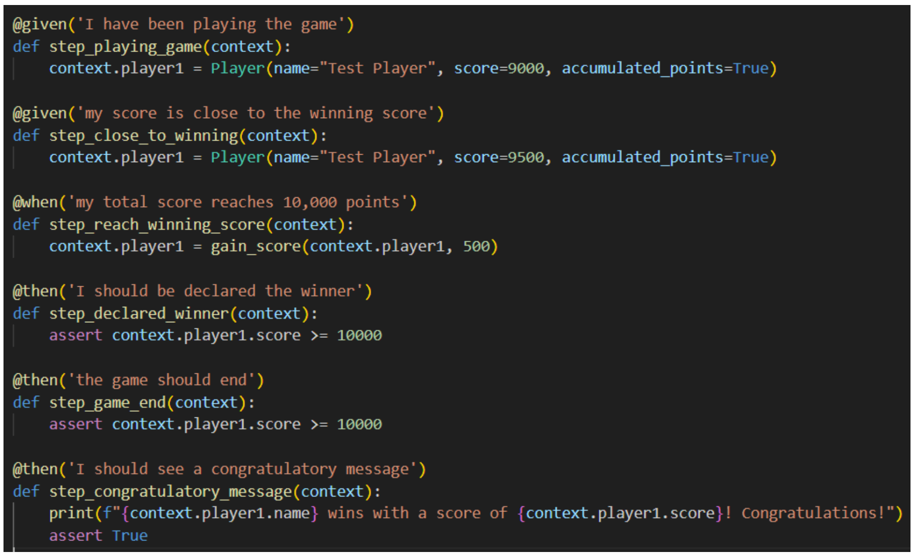


```
Scenario: Rolling a Farkle                             # features/steps/GherkinFeatures.feature:46
  Given it is my turn                                  # features/steps/steps.py:61
  When I roll the dice                                 # features/steps/steps.py:65
  And none of the remaining dice score points          # features/steps/steps.py:121
  Then I should be notified that I Farkled             # features/steps/steps.py:125
  And my turn should end immediately                   # features/steps/steps.py:130
```

Test Reaching the Winning Score

Ensures that the game correctly identifies when a player reaches the winning score and ends the game appropriately.

Code Snippet and Scenario Feature



```python
@given('I have been playing the game')
def step_playing_game(context):
    context.player1 = Player(name="Test Player", score=9000, accumulated_points=True)

@given('my score is close to the winning score')
def step_close_to_winning(context):
    context.player1 = Player(name="Test Player", score=9500, accumulated_points=True)

@when('my total score reaches 10,000 points')
def step_reach_winning_score(context):
    context.player1 = gain_score(context.player1, 500)

@then('I should be declared the winner')
def step_declared_winner(context):
    assert context.player1.score >= 10000

@then('the game should end')
def step_game_end(context):
    assert context.player1.score >= 10000

@then('I should see a congratulatory message')
def step_congratulatory_message(context):
    print(f"{context.player1.name} wins with a score of {context.player1.score}! Congratulations!")
    assert True
```

Scenario: Reaching the Winning Score

Given I have been playing the game

And my score is close to the winning score

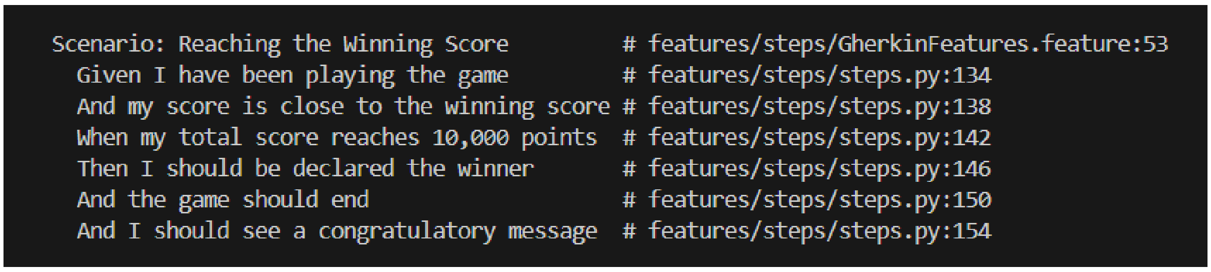
When my total score reaches 10,000 points

Then I should be declared the winner

And the game should end

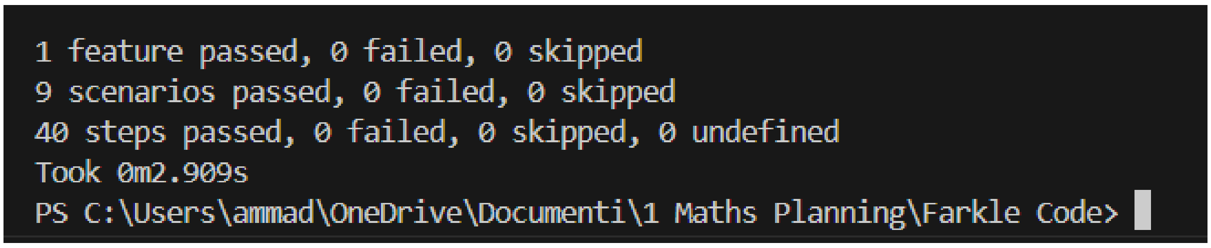
And I should see a congratulatory message

Result



```
  Scenario: Reaching the Winning Score          # features/steps/GherkinFeatures.feature:53
    Given I have been playing the game          # features/steps/steps.py:134
    And my score is close to the winning score # features/steps/steps.py:138
    When my total score reaches 10,000 points  # features/steps/steps.py:142
    Then I should be declared the winner        # features/steps/steps.py:146
    And the game should end                     # features/steps/steps.py:150
    And I should see a congratulatory message  # features/steps/steps.py:154
```

All results



```
1 feature passed, 0 failed, 0 skipped
9 scenarios passed, 0 failed, 0 skipped
40 steps passed, 0 failed, 0 skipped, 0 undefined
Took 0m2.909s
PS C:\Users\ammad\OneDrive\Documenti\1 Maths Planning\Farkle Code>
```

T3 Reflection

In testing my Farkle game, I discovered the importance of extensive testing for reliability. Manual testing helped me reproduce user interactions and find errors, while automated unit tests allowed testing for each component efficiently. BDD testing with Gherkin scenarios helped my requirement communication.

I have learned how to set up unit tests and use assert statements to verify the correctness of my code by targeting each function. And thanks to the BDD testing I learned how to set up and use the Behave framework to implement Gherkin scenarios for defining the expected behaviour of my game.

Overall, testing has greatly enhanced my knowledge and understanding.

T4: Understanding and exploring team / community-based software development projects: Behave.

Introduction

For this task I'll learn and explain about the Python library and testing framework Behave. Behave simplifies BDD with Gherkin User Stories and Behavioural Specifications. Simply put, it helps developers create tests in a way that even non-programmers can understand, improving team collaboration.

I'll investigate numerous aspects of the Behave project's development process to better understand team-based software development. This includes looking into the importance of having good documentation for contributors and the impact of version control tools like Git in managing multi-contributor projects.

I will also look at the project's commit history and review some pull requests to see how changes are proposed and reviewed.

Finally, I will explore how leveraging a version control system like Git can be beneficial for team-based software projects.

Project Documentation

For each software project having comprehensive documentation is a major advantage because it allows an easier onboarding process for new contributors as they can quickly understand how the project is structured and set up. This allows the new contributors to quickly get on track with everyone else on the team. Consistency is effectively achieved through documentation as it ensures everyone follows the practice and standards. Efficiency is also vital, and it is best achieved through documentation because well-documented procedures and codebases allow for faster troubleshooting and implementation of new features. Documentation also improves collaboration between the team as every contributor can understand the full project and the priorities along with what they need to do, this avoids duplicate work and a better work coordination. Another important benefit of documentation is code maintainability and scalability as it allows knowledge preservation about the project, it means new contributors can easily get an overview of the project and be ready to fix any issues, bugs and add improvements on the code if necessary.

Version Control Systems

Version Control Systems play a crucial role in software development projects as they are a solution to a lot of issues developers encounter when working in teams. Version control, commonly referred to as revision control or source control, is a systematic process that monitors and oversees changes made to files, specifically source code files, across a period of time. A detailed history is kept for all files allowing multiple developers to work on the same codebase simultaneously. This means they every developer can do their part on a project without confliction with others while also effectively sharing knowledge. (MoldStud,2024)

Git is an amazing tool as it allows teams to monitor modifications made to their codebase, work together on projects, and revert to prior versions if needed. Some major advantages of git are:

- Distributed Nature and Local Repositories, allowing every contributor to have full access to the code database offline. This also ensure no single point failure.
- Speed and Performance as most operations on Git are performed locally.
- Flexibility and Branching Models allowing developers to create new features, bug fixes and experimental changes in new lightweight branches.
- Collaboration and Code Reviews, Git allows for easy implementation of new changes and review of other contributor's code.
- Change Tracking and Accountability, thanks to this any change made on the project is tracked, it exactly shows, when the change was made, where and by whom. (Ogeto, B. 2023)

These are some of many advantages that Version Control Systems can provide to software developers. There are other version control systems such as Google Drive and Dropbox these are cloud-based ones. These offer basic version control functions and are mainly for file sharing and storage, they can be a good choice for small personal projects however for more complex ones, version control like Git is the best since it is specifically designed for software development workflow. (Vikas,2020)

Git Functions

Clone

Cloning involves making a local duplicate of a remote repository, complete with all files, branches, and commits. When you clone a repository, you can work on the project on your own computer and then easily sync any changes with the remote repository at a later time. This means that a new contributor can clone the project on their local device ensuring that even if they make mistakes the actual project won't be affected, so they can easily experiment and only synchronize the projects when everything is right.

Commit

A commit is a collection of changes that have been made to the repository. When you make a commitment, you're essentially capturing a snapshot of your changes and storing them in the local repository. Every commit is assigned a distinct identifier and accompanied by a commit message that explains the modifications made. This allows users to save the current version of the project which they can revert back to in case of issues such as bugs. These changes are only saved locally meaning the remote repository is unaffected.

Pull

Pulling is the process of bringing in and combining updates from a remote repository into your local repository. It syncs your local copy with the most recent updates from the remote repository. This necessary to update your work based on what has been updated on the remote repository by other contributors. Pulling allows you to be up to date with what other contributors. Being up to date is essential to avoid bugs and errors, and also makes sure you don't waste time working on something not updated.

Push

This is essentially the opposite of pulling, pushing It's all about sending your changes from your computer to a remote repository, making sure the remote repository is up to date with your local changes. Push happens when you are ready to integrate your local repository with the remote one. It is essential to pull the remote repository once to make sure your changes are working with the current updated remote repository, once all checks have been done it is a good idea to commit the work and then finally push it.

Branch

A branch is like a separate path of progress within a repository. Every developer can create, modify, and test changes without affecting the main remote repository. Branches can then be merged back into the main branch or other branches once the changes are good to go. This is essential as it gives developers the freedom to try new ideas and implement them without risking the entire system.

Merge

Merging involves taking the changes made in one branch and incorporating them into another branch. It combines the changes from one branch into another, creating a new commit that includes all the modifications. This is how developers can add their new implementations to the main code.

Pull Request

It's basically a way to let others know about any changes you've made and pushed to a repository on the hosting service in this case GIT. It lets collaborators have a look and talk about the changes before merging them into the main codebase. Pull requests are a great way to encourage code reviews and build collaboration among team members. Pull requests also ensure that your work is checked by others to make sure it's error free and that it can easily merge with the main codebase without it crashing, this why pull requests should carefully be examined. Prajapati, A. (2019)

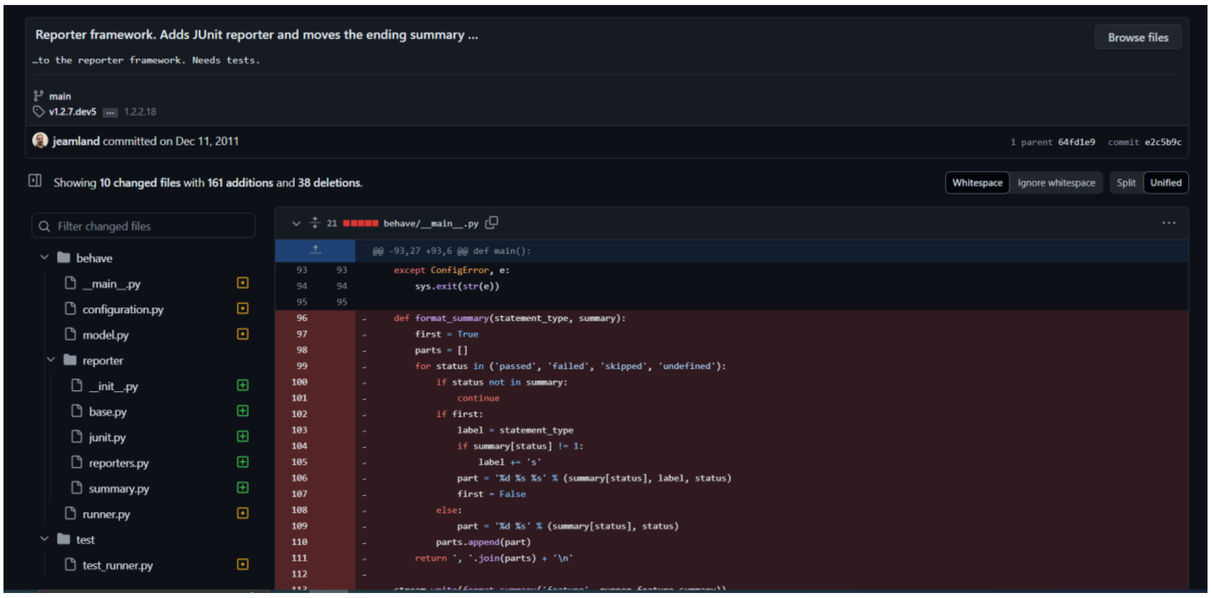
Behave Project Commit History

Early Major Commit

This commit (ID e2c5b9c) was done on 11 December 2011by jeamland, a contributor to Behave. It introduced a new Reporter framework with a base reporter class and specific implementations like the JUnit-like reporter. 10 files were changed with a total of 161 additions and 38 deletions.

In red are all lines of code that were removed. While in green it is all lines of code newly added.

The **JUnitReporter** class in behave/reporter/junit.py is specifically designed to generate **XML reports** that are compatible with JUnit. These reports are essential for easily integrating Behave's test results with CI systems and other tools. The class captures every detail of test execution, like test cases, how long they took, and whether they passed, failed, or were skipped. It writes this information into XML files in a format that **JUnit** parsers can understand. This was essentially the main change that happened here, it changed how Behave handles reporting by adding a flexible reporter framework. A new Reporter framework with a base class for creating custom reporters **(behave/reporter/base.py)** and a **SummaryReporter** class for generating test summaries **(behave/reporter/summary.py)** were all added in this commit.



Reporter framework. Adds JUnit reporter and moves the ending summary ...

…to the reporter framework. Needs tests.

main
v1.2.7.dev5 … 1.2.2.18

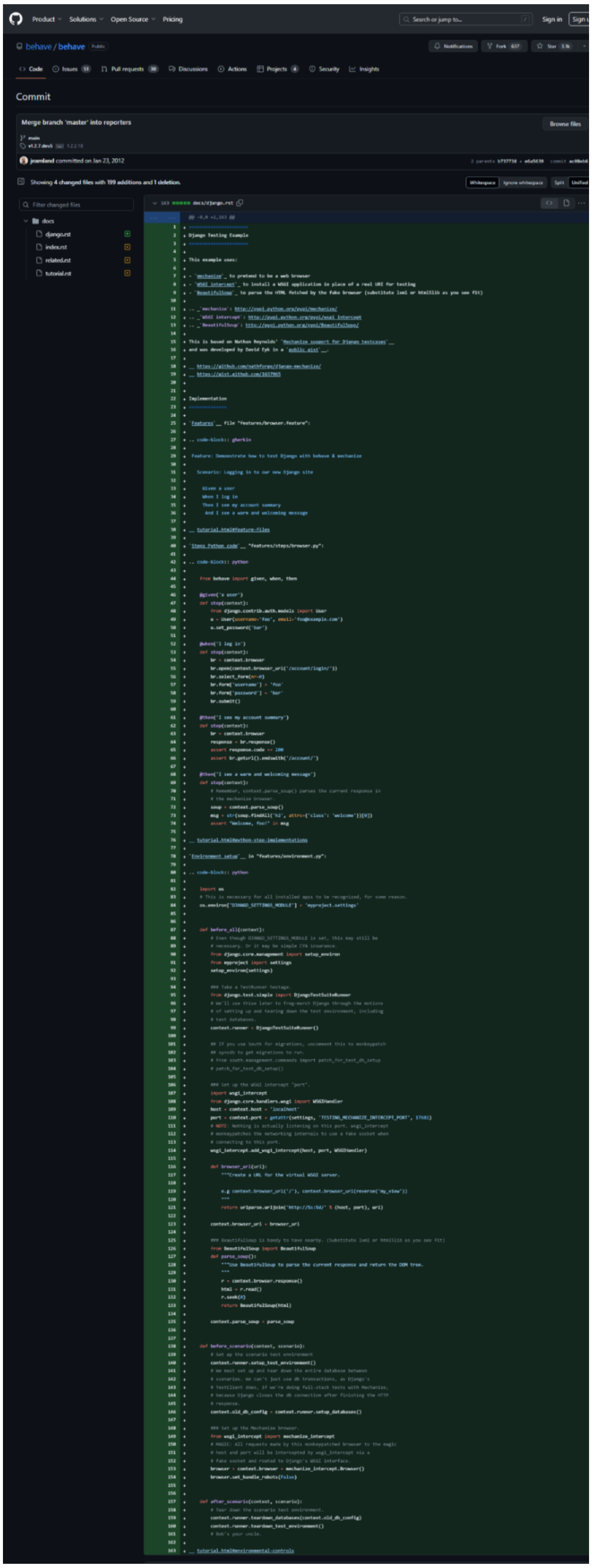
jeamland committed on Dec 11, 2011 — 1 parent 64fd1e9 — commit e2c5b9c

Showing **10 changed files** with **161 additions** and **38 deletions**.

21 behave/__main__.py

```
@@ -93,27 +93,6 @@ def main():
     except ConfigError, e:
         sys.exit(str(e))

-    def format_summary(statement_type, summary):
-        first = True
-        parts = []
-        for status in ('passed', 'failed', 'skipped', 'undefined'):
-            if status not in summary:
-                continue
-            if first:
-                label = statement_type
-                if summary[status] != 1:
-                    label += 's'
-                part = '%d %s %s' % (summary[status], label, status)
-                first = False
-            else:
-                part = '%d %s' % (summary[status], status)
-            parts.append(part)
-        return ', '.join(parts) + '\n'
-
```

https://github.com/behave/behave/commits/main/?since=2011-12-11&until=2011-12-11. The rest of the code changes are there. I couldn't fit it all in images.



Commit

Merge branch 'master' into reporters

main
v1.2.7.dev5 ... 1.2.2.18

jeamland committed on Jan 23, 2012 — 2 parents b737738 + e6a5639 commit ac0beb6

Showing 4 changed files with 199 additions and 1 deletion.

- docs
  - django.rst
  - index.rst
  - related.rst
  - tutorial.rst

```
163 docs/django.rst
@@ -0,0 +1,163 @@
+ ====================
+ Django Testing Example
+ ====================
+
+ This example uses:
+
+ - `mechanize`_ to pretend to be a web browser
+ - `WSGI intercept`_ to install a WSGI application in place of a real URI for testing
+ - `BeautifulSoup`_ to parse the HTML fetched by the fake browser (substitute lxml or html5lib as you see fit)
+
+ .. _`mechanize`: http://pypi.python.org/pypi/mechanize/
+ .. _`WSGI intercept`: http://pypi.python.org/pypi/wsgi_intercept
+ .. _`BeautifulSoup`: http://pypi.python.org/pypi/BeautifulSoup/
+
+ This is based on Nathan Reynolds' `Mechanize support for Django testcases`__
+ and was developed by David Eyk in a `public gist`__.
+
+ __ https://github.com/nathforge/django-mechanize/
+ __ https://gist.github.com/1637965
+
+
+ Implementation
+ ==============
+
+ `Features`__ file "features/browser.feature":
+
+ .. code-block:: gherkin
+
+  Feature: Demonstrate how to test Django with behave & mechanize
+
+    Scenario: Logging in to our new Django site
+
+      Given a user
+      When I log in
+      Then I see my account summary
+       And I see a warm and welcoming message
+
+ __ tutorial.html#feature-files
+
+ `Steps Python code`__ "features/steps/browser.py":
+
+ .. code-block:: python
+
+     from behave import given, when, then
+
+     @given('a user')
+     def step(context):
+         from django.contrib.auth.models import User
+         u = User(username='foo', email='foo@example.com')
+         u.set_password('bar')
+
+     @when('I log in')
+     def step(context):
+         br = context.browser
+         br.open(context.browser_url('/account/login/'))
+         br.select_form(nr=0)
+         br.form['username'] = 'foo'
+         br.form['password'] = 'bar'
+         br.submit()
+
+     @then('I see my account summary')
+     def step(context):
+         br = context.browser
+         response = br.response()
+         assert response.code == 200
+         assert br.geturl().endswith('/account/')
+
+     @then('I see a warm and welcoming message')
+     def step(context):
+         # Remember, context.parse_soup() parses the current response in
+         # the mechanize browser.
+         soup = context.parse_soup()
+         msg = str(soup.findAll('h2', attrs={'class': 'welcome'})[0])
+         assert "Welcome, foo!" in msg
+
+ __ tutorial.html#python-step-implementations
+
+ `Environment setup`__ in "features/environment.py":
+
+ .. code-block:: python
+
+     import os
+     # This is necessary for all installed apps to be recognized, for some reason.
+     os.environ['DJANGO_SETTINGS_MODULE'] = 'myproject.settings'
+
+
+     def before_all(context):
+         # Even though DJANGO_SETTINGS_MODULE is set, this may still be
+         # necessary. Or it may be simple CYA insurance.
+         from django.core.management import setup_environ
+         from myproject import settings
+         setup_environ(settings)
+
+         ### Take a TestRunner hostage.
+         from django.test.simple import DjangoTestSuiteRunner
+         # We'll use thise later to frog-march Django through the motions
+         # of setting up and tearing down the test environment, including
+         # test databases.
+         context.runner = DjangoTestSuiteRunner()
+
+         ## If you use South for migrations, uncomment this to monkeypatch
+         ## syncdb to get migrations to run.
+         # from south.management.commands import patch_for_test_db_setup
+         # patch_for_test_db_setup()
+
+         ### Set up the WSGI intercept "port".
+         import wsgi_intercept
+         from django.core.handlers.wsgi import WSGIHandler
+         host = context.host = 'localhost'
+         port = context.port = getattr(settings, 'TESTING_MECHANIZE_INTERCEPT_PORT', 17681)
+         # NOTE: Nothing is actually listening on this port. wsgi_intercept
+         # monkeypatches the networking internals to use a fake socket when
+         # connecting to this port.
+         wsgi_intercept.add_wsgi_intercept(host, port, WSGIHandler)
+
+         def browser_url(url):
+             """Create a URL for the virtual WSGI server.
+
+             e.g context.browser_url('/'), context.browser_url(reverse('my_view'))
+             """
+             return urlparse.urljoin('http://%s:%d/' % (host, port), url)
+
+         context.browser_url = browser_url
+
+         ### BeautifulSoup is handy to have nearby. (Substitute lxml or html5lib as you see fit)
+         from BeautifulSoup import BeautifulSoup
+         def parse_soup():
+             """Use BeautifulSoup to parse the current response and return the DOM tree.
+             """
+             r = context.browser.response()
+             html = r.read()
+             r.seek(0)
+             return BeautifulSoup(html)
+
+         context.parse_soup = parse_soup
+
+
+     def before_scenario(context, scenario):
+         # Set up the scenario test environment
+         context.runner.setup_test_environment()
+         # We must set up and tear down the entire database between
+         # scenarios. We can't just use db transactions, as Django's
+         # TestClient does, if we're doing full-stack tests with Mechanize,
+         # because Django closes the db connection after finishing the HTTP
+         # response.
+         context.old_db_config = context.runner.setup_databases()
+
+         ### Set up the Mechanize browser.
+         from wsgi_intercept import mechanize_intercept
+         # MAGIC: All requests made by this monkeypatched browser to the magic
+         # host and port will be intercepted by wsgi_intercept via a
+         # fake socket and routed to Django's WSGI interface.
+         browser = context.browser = mechanize_intercept.Browser()
+         browser.set_handle_robots(False)
+
+
+     def after_scenario(context, scenario):
+         # Tear down the scenario test environment.
+         context.runner.teardown_databases(context.old_db_config)
+         context.runner.teardown_test_environment()
+         # Bob's your uncle.
+
+ __ tutorial.html#environmental-controls
```

Relevant Change Commit

This commit was done on 23 January 2012 by jeamland a contributor to the behave project.

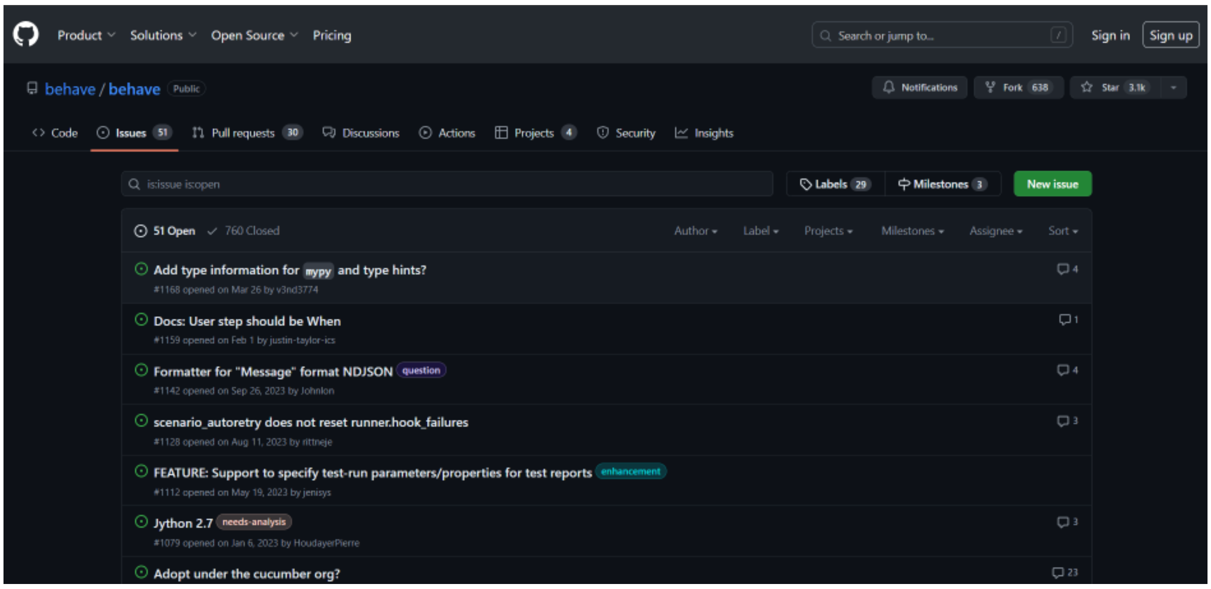
4 files were changed with 199 additions and 1 deletion.

This change added substantial enhancement to the project's documentation.

A new documentation file **(docs/django.rst)** with a detailed Django example was added while the documentation index **(docs/index.rst)** was updated to include a link to this new Django example and the related tools page **(docs/related.rst).**

This commit was not a merge from a branch or fork because it was just a direct commit to documentation of the project.

Issue Tracker



Add type information for `mypy` and type hints? 4
#1168 opened on Mar 26 by v3nd3774

Docs: User step should be When 1
#1159 opened on Feb 1 by justin-taylor-ics

Formatter for "Message" format NDJSON question 4
#1142 opened on Sep 26, 2023 by Johnlon

scenario_autoretry does not reset runner.hook_failures 3
#1128 opened on Aug 11, 2023 by rittneje

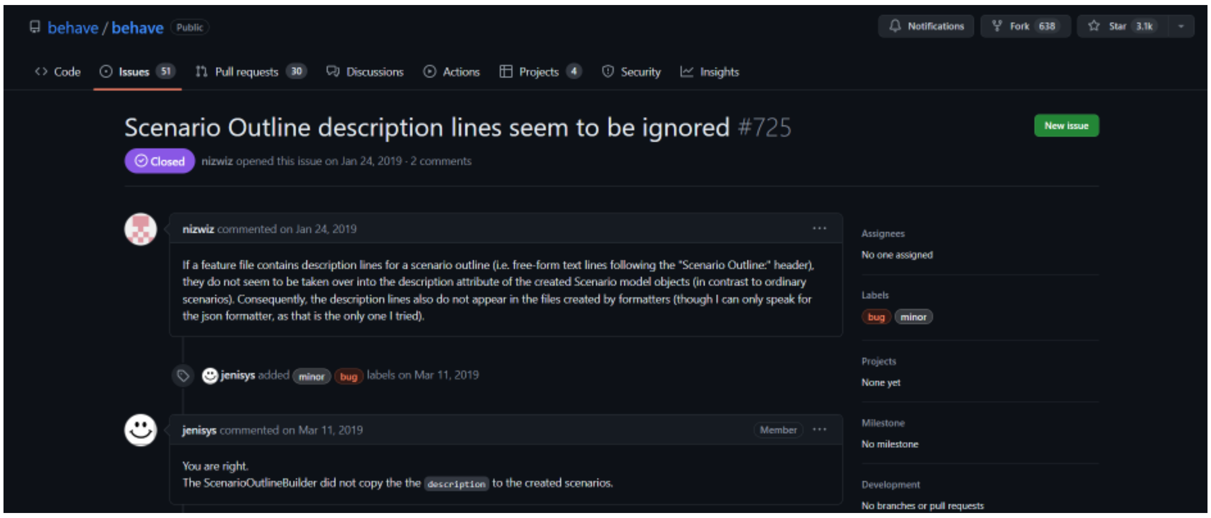
FEATURE: Support to specify test-run parameters/properties for test reports enhancement
#1112 opened on May 19, 2023 by jenisys

Jython 2.7 needs-analysis 3
#1079 opened on Jan 6, 2023 by HoudayerPierre

Adopt under the cucumber org? 23

The GitHub Issue Tracker is a tool used to manage and track issues in a project, including bug reports, feature requests, and development tasks. Anyone can report any issues they are having. This issue will be then reviewed by a contributor and labelled correctly. There are currently 51 active issues and 760 closed issues involving Behave.

Example Issue



Scenario Outline description lines seem to be ignored #725

nizwiz opened this issue on Jan 24, 2019 · 2 comments

nizwiz commented on Jan 24, 2019

If a feature file contains description lines for a scenario outline (i.e. free-form text lines following the "Scenario Outline:" header), they do not seem to be taken over into the description attribute of the created Scenario model objects (in contrast to ordinary scenarios). Consequently, the description lines also do not appear in the files created by formatters (though I can only speak for the json formatter, as that is the only one I tried).

jenisys added minor bug labels on Mar 11, 2019

jenisys commented on Mar 11, 2019

You are right.
The ScenarioOutlineBuilder did not copy the the `description` to the created scenarios.

Assignees
No one assigned

Labels
bug minor

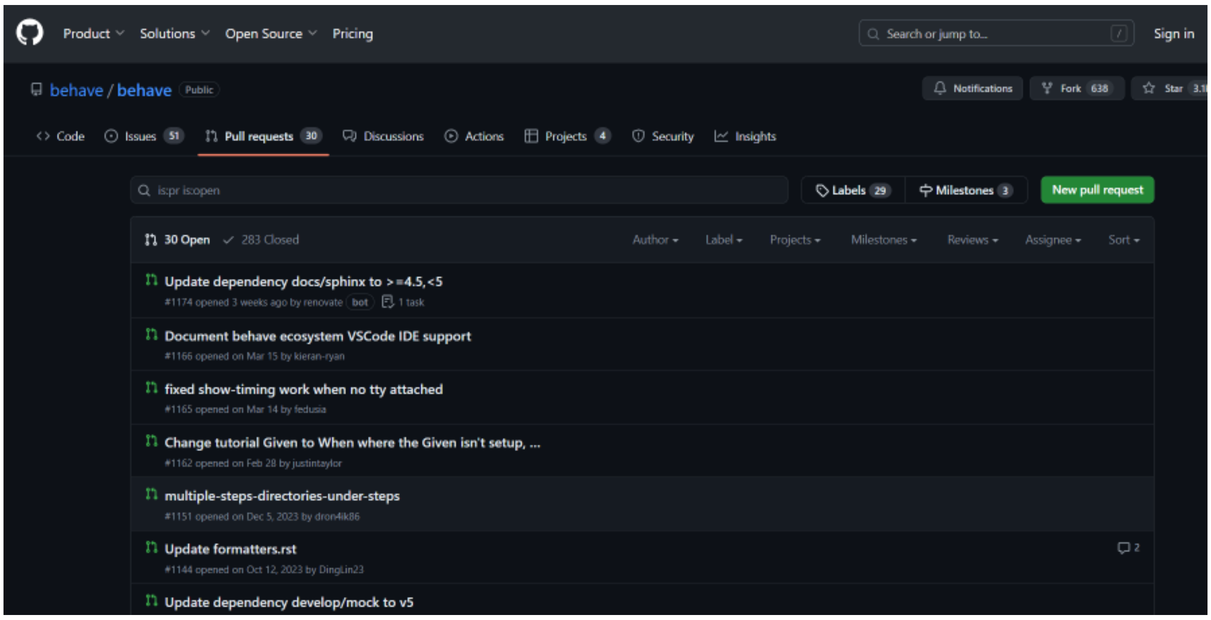
Projects
None yet

Milestone
No milestone

Development
No branches or pull requests

Description lines for scenario outlines are not being copied to the **description** attribute of **Scenario** model objects, affecting their appearance in formatted output was caused by a bug. This issue has been resolved by jenisys on 11 march 2019 it was reported by nizwiz on 24 January 2019

Pull Requests



30 Open ✓ 283 Closed

Update dependency docs/sphinx to >=4.5,<5
#1174 opened 3 weeks ago by renovate bot 1 task

Document behave ecosystem VSCode IDE support
#1166 opened on Mar 15 by kieran-ryan

fixed show-timing work when no tty attached
#1165 opened on Mar 14 by fedusia

Change tutorial Given to When where the Given isn't setup, ...
#1162 opened on Feb 28 by justintaylor

multiple-steps-directories-under-steps
#1151 opened on Dec 5, 2023 by dron4ik86

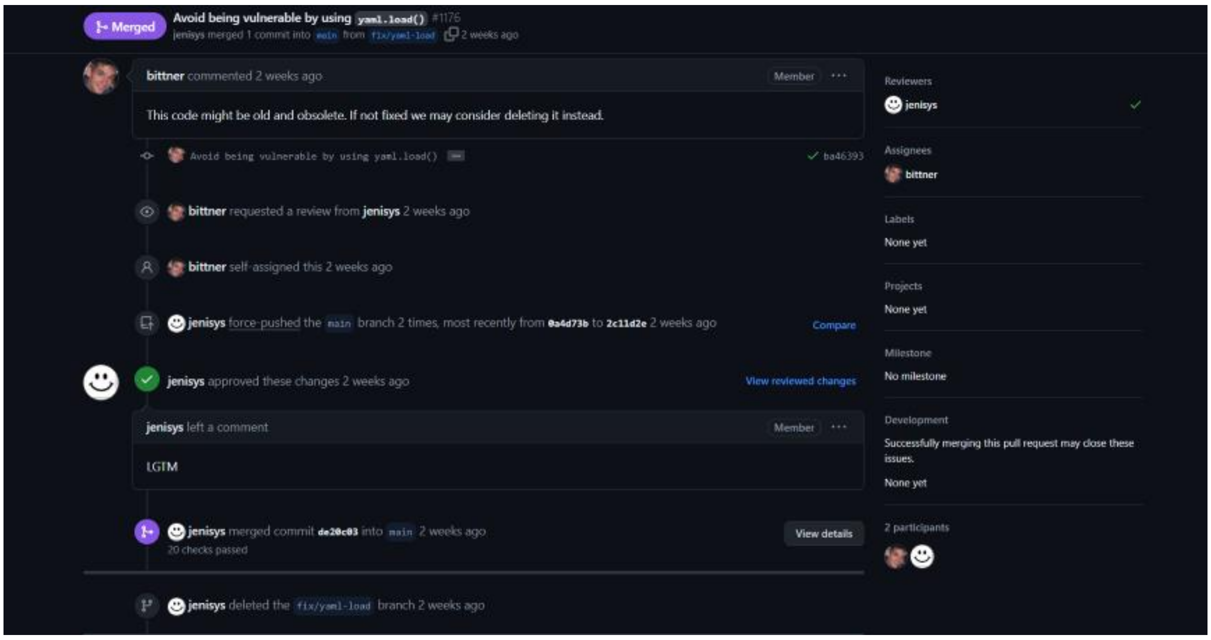
Update formatters.rst 2
#1144 opened on Oct 12, 2023 by DingLin23

Update dependency develop/mock to v5

This section is used for reviewing and merging changes into the codebase. Developers can suggest changes to improve features or fix issues. These requests get reviewed by contributors.

There are currently 30 open requests and 283 closed.

Example Pull Request



Avoid being vulnerable by using `yaml.load()` #1176
jenisys merged 1 commit into `main` from `fix/yaml-load` 2 weeks ago

bittner commented 2 weeks ago

This code might be old and obsolete. If not fixed we may consider deleting it instead.

Avoid being vulnerable by using yaml.load() ✓ ba46393

bittner requested a review from jenisys 2 weeks ago

bittner self-assigned this 2 weeks ago

jenisys force-pushed the `main` branch 2 times, most recently from 0a4d73b to 2c11d2e 2 weeks ago

jenisys approved these changes 2 weeks ago

jenisys left a comment

LGTM

jenisys merged commit de20c03 into `main` 2 weeks ago
20 checks passed

jenisys deleted the `fix/yaml-load` branch 2 weeks ago

Reviewers
jenisys ✓

Assignees
bittner

Labels
None yet

Projects
None yet

Milestone
No milestone

Development
Successfully merging this pull request may close these issues.
None yet

2 participants

This request was suggested by bittner aimed to address a security vulnerability associated with the use of **yaml.load()** by modifying or replacing it to ensure safer code execution. After jenisys approved the pull request, it was integrated with all checks passing and the **fix/yaml-load** branch was deleted post-merge. This update reduces YAML parsing vulnerabilities, improving project security.

T4 Reflection

Leveraging Git would significantly enhance the development process for this coursework by managing code changes efficiently and facilitating individual progress. Any changes I make will be traceable and if I have issues I can always roll back to a previous version, I could also test features before merging them into the main branch. Git would improve organization, traceability, and lead to a more robust and well-documented project.

There are many benefits to using a version control system like Git. My plan is to learn how to effectively use all Git and Github functions so I can store my projects there in the future.

Thank you for reading. This is the end.

References

(MoldStud,2024) The importance of version control systems in software development. Available at: https://moldstud.com/articles/p-the-importance-of-version-control-systems-in-software-development.

Ogeto, B. (2023) The benefits of using version control systems like git. Available at: https://hojaleaks.com/the-benefits-of-using-version-control-systems-like-git-in-software-development.

(Vikas,2020) Vikas, Cloud Duplicate Finder, Find Duplicates In Google Drive, OneDrive, Dropbox, Box. Available at: https://www.cloudduplicatefinder.com/blog/comparing-version-control-features-of-onedrive-dropbox-google-drive/

Prajapati, A. (2019) What is git commit, Push, pull, log, aliases, fetch, Config & Clone, Medium. Available at: https://medium.com/mindorks/what-is-git-commit-push-pull-log-aliases-fetch-config-clone-56bc52a3601c