

Initial Setup

The development environment for this project will be set up on a windows desktop running windows, utilizing Visual Studio Code as the Integrated Development Environment (IDE). Instead of C++, this project will be implemented in Python.

Project Overview The project aims to develop a Farkle game with a Command Line Interface (CLI), enabling multiple players to engage in gameplay concurrently. Farkle is a dice game played by two or more participants, where each player takes turns rolling dice to accumulate points. The game's objective is to reach a predetermined score by successfully scoring points through various dice combinations

Rules of Farkle: The rules of Farkle implemented in this project will adhere to the standard rules, including:

- Each player's turn starts by rolling all six dice.
- Players score points based on the combinations rolled (see scoring rules).
- Players choose to bank points or continue rolling.
- If no scoring dice are rolled, the turn ends with no points (Farkle).
- The game continues until a player reaches the winning score.

Scoring Rules

The scoring combinations and their corresponding points are as follows:

- Single 1: 100 points
- Single 5: 50 points
- Three of a kind: 100 times the face value of the die
- Four of a kind: 1000 times the face value of the die
- Five of a kind: 2000 times the face value of the die
- Six of a kind: 3000 times the face value of the die
- Straight (1-6): 1500 points
- Three pairs: 1500 points

The game will provide an enjoyable and challenging experience for players, with the excitement of risk-taking and strategic decision-making.

Requirements Analysis

The system needs to fulfill the following criteria:

1. Allow for a user to interact with it using a command-line interface (CLI).
2. Allow a player to start a new game with two or more players.
3. Enable each player to take turns rolling dice and scoring points.
4. Determine when a player has won the game.
5. Provide an option for a single-player game against the CPU.
6. Allow players to input their names.

Additionally, the system will fulfill the following criteria:

1. Record the wins, draws, and losses for each player.
2. Provide clear instructions and rules for the game.

From the above requirements, it's evident that user input through a command line will be a necessary component. Users will input commands and choices via the command line, and the program will parse these inputs to execute the appropriate actions.

Functional Requirements

1. **CLI Interaction:**

- Users should be able to start the game, make choices, and input their actions through the command line.
- The system should display prompts and instructions to guide the user.

2. **New Game Setup:**

- Users should be able to start a new game with multiple players.
- The game should allow users to input the number of players and their names.

3. **Gameplay:**

- Each player should take turns rolling dice automatically and making decisions.
- The system should calculate and display the score for each roll.
- Players should be able to choose to roll again or end their turn.

4. **Win Condition:**

- The system should recognize when a player has reached the winning score and declare them the winner.
- If a player rolls a Farkle, the system should end their turn and notify them accordingly.

5. **Single-Player Mode:**

- Users should have the option to play against a computer opponent.

- The CPU should make strategic decisions based on the current game state.

6. **Player Management:**

- Users should be able to assign names to players.

Non-functional Requirements

1. **User-Friendly Interface:**

- The CLI should provide clear instructions and feedback to the user.
- Messages should be well-formatted and easy to understand.

2. **Efficiency:**

- The system should execute actions promptly, without significant delays.
- Dice rolls and scoring calculations should be efficient.

3. **Scalability:**

- The system should be able to handle multiple players and game sessions concurrently.

4. **Reliability:**

- The game should be stable and reliable, without unexpected crashes or errors.

Behavior Driven Development (Gherkin Specifications)

FEATURE: ENTERING PLAYER NAMES

AS A CLI USER/PLAYER: I want to enter player names So that each player can be identified in the game

SCENARIO Player enters their name

Given the program has started

When the program prompts the user to enter the number of players

And the user inputs the number of players as "2"

And the program prompts each player to enter their name

And each player enters their name

Then the game begins and displays the names of the players

SCENARIO Player enters an empty name

Given the program has prompted the user to enter their name

When the user enters an empty string as their name

Then the program displays an error message

SCENARIO Player enters a valid name
Given the program has prompted the user to enter their name When the user enters a valid name Then the program assigns the name to the player
FEATURE: ROLLING DICE
AS A CLI USER/PLAYER: I want to roll the dice So that I can score points and continue playing
SCENARIO: Player rolls the dice Given the player's turn has started When the player chooses to roll the dice Then the program rolls the dice and displays the result
SCENARIO Player scores points Given the player has rolled the dice When the dice result contains scoring combinations Then the program calculates the score and displays it to the player
SCENARIO Player gets a Farkle Given the player has rolled the dice And the dice result does not contain any scoring combinations Then the program displays a Farkle message and ends the player's turn
SCENARIO Player chooses to end their turn Given the player has rolled the dice And the dice result contains scoring combinations When the player chooses to end their turn Then the program adds the accumulated score to the player's total score
FEATURE: WINNING THE GAME
AS A CLI USER/PLAYER: I want to win the game So that I can be declared the winner
SCENARIO Player reaches the winning score Given the game is ongoing And a player reaches the winning score Then the program declares the player as the winner and ends the game
SCENARIO Multiple players reach the winning score Given the game is ongoing And multiple players reach the winning score simultaneously Then the program declares it as a tie between those players and ends the game
SCENARIO Player chooses to end the game Given the game is ongoing

When the player selects the option to end the game from the main menu
Then the program confirms the player's decision and ends the game

Data Model

Input:

- User Input (Standard Input)

Output Message:

- Menus
- Selected Name
- Win
- Loss
- Draw
- Statistics of a player

Error:

- Error Values
 - Invalid argument
- Error Message
- Exit Code

Name Model

The name model for the Farkle game represents the players' names. When a player starts the game, they are prompted to enter their name, which serves as their unique identifier throughout the game. The name type contains the names chosen by the players, and a player is only allowed to use their selected name as their identifier.

With this in mind, the expression of a name can be defined as a set of player names:

Let Name be the set of player names:

Name={Name1} U Name2} U...U {Namen}

Here, each Name represents a player's chosen name, making the name effectively a subset of all possible player names. This approach simplifies the validation of inputs, ensuring that only valid player names are used during gameplay.

For example:

$$\text{Name}=\{\text{"Alice"},\text{"Bob"},\text{"Charlie"}\}$$

In this case, the Name set has a cardinality of n , representing the number of players in the game. Each element of the set is a unique player name, ensuring that the name remains valid and constrained to the selected player names.

Turn Model

In the Farkle game, turns are organized as a sequence of players, where each player takes their turn in order. The active player, who is currently taking their turn, is the player at the head of the sequence.

The player sequence can be represented as follows:

Let Players be a sequence of players:

$$\text{Players}=\text{seq}\langle \text{Player}_1,\text{layer}_2,\dots,\text{Player}_n\rangle$$

Here, each *Player* represents an instance of a player in the game. The sequence allows for flexibility in the number of players, accommodating changes such as adding more players to the game.

To represent the sequence of players in union form, we can describe the set of all players using union notation. Here's how it can be expressed:

$$\text{Players} = \bigcup_{i=1}^n \{ \text{Player } \{i\} \}$$

This notation indicates that the set Players is the union of all individual player sets {Player_1}, {Player_2} {Player_n}. Each {Player_i} represents an individual player in the sequence.

Input model

Players in the Farkle game interact with the system by providing input to select from presented options.

- $\text{Input}=\text{seq}\langle \text{Char}\rangle$

Player Model

- The name of a player is a sequence of characters: $\text{Name}=\text{seqChar}$

Axiomatic Definitions and Functions

Player Name

To allow players to set their names, a function **getPlayerNameInput** will be implemented. This function prompts the player to enter a name, which is a sequence of characters, and returns this

sequence as the player's name. The implementation will be partial as there cannot be a mapped matching pair to every possible input due to the variation of names that any user can have.

getPlayerNameInput:Void→Name
getPlayerNameInput:Void→Name

The function to check if a name already exists in the database is **checkPlayerName**. It returns a boolean value based on whether the name exists in the database (true) or if it does not (false).

checkPlayerName:Name→Bool
checkPlayerName:Name→Bool

Evaluating winner

The winner in the Farkle game is evaluated based on the total score accumulated by each player. The game continues until one player reaches the winning score threshold, which is set to 10,000 points by default. At the end of each turn, the current player's score is updated based on the points they earned during that turn.

During each turn, the player rolls six dice and scores points based on various combinations. If the player rolls no scoring combinations, it's called a "Farkle," and they earn no points for that turn. The player then has the option to either roll again or end their turn, banking the points they've accumulated so far.

The game proceeds with each player taking turns until one player reaches or exceeds the winning score. At that point, the game ends, and the player with the highest score is declared the winner. If there's a tie, meaning multiple players reach the winning score simultaneously, they are all declared winners.

T2 Implementation

Implementation of the program can now begin, considering two types of functions: pure and impure. Pure functions don't modify the program state outside their scope, while impure functions do. Additionally, there are totalised and non-totalised functions. Totalised functions cover all possible input-value pairs, while non-totalised functions don't. It's preferable to create pure, totalised functions wherever feasible.

FarklePlayer Class

```
class FarklePlayer:
    def __init__(self, name):
        self.name = name
        self.score = 0
        self.statistics = {'wins': 0, 'losses': 0, 'draws': 0}
```

Explanation: Initializes a player with a name, score, and statistics.

Pure/Impure: Pure. It initializes the player's attributes but doesn't modify the program state outside its scope.

Totalised/Non-Totalised: Yes, it's totalised. It covers all possible initial player states.

FarkleDice Class

```
class FarkleDice:
    def __init__(self):
        self.dice = [0, 0, 0, 0, 0, 0]
```

Explanation: Initializes a set of six dice with values initialized to 0.

Pure/Impure: Pure. It initializes the dice but doesn't modify the program state outside its scope.

Totalised/Non-Totalised: Yes, it's totalised. It covers all possible initial dice states.

Rolling dice for every player

```
def roll_dice(self):
    self.dice = [random.randint(1, 6) for _ in range(6)]
    return self.dice
```

Explanation: Rolls the dice, assigning random values from 1 to 6 to each die.

Pure/Impure: Impure. It modifies the dice values, changing the program state.

Totalised/Non-Totalised: Yes, it's totalised. It covers all possible outcomes of rolling the dice.

FarkleGame Class

```
class FarkleGame:
    def __init__(self, winning_score=10000):
        self.players = []
        self.current_player = None
        self.winning_score = winning_score
        self.dice = FarkleDice()
```

Explanation: Initializes a Farkle game with parameters such as players, current player, winning score, and a FarkleDice object.

Pure: Impure. It modifies the game state by initializing attributes.

Totalised: Yes, it's totalised. It handles all possible initial game configurations.

Saving players name

```
def get_player_names(self):
    num_players = int(input("Enter the number of players: "))
    for i in range(num_players):
        name = getPlayerNameInput()
        while checkPlayerName(name):
            print("Name already exists. Please enter a different name.")
            name = getPlayerNameInput()
        self.players.append(FarklePlayer(name))
```

```
def getPlayerNameInput():
    name = input("Enter your name: ")
    return name
```

Explanation: Gets names for each player, ensuring no duplicate names.

Pure: Impure. It interacts with the user and modifies the game state.

Totalised: Yes, it's totalised. It handles all possible inputs for player names.

Display rules and points of the game

```
def display_rules(self):
    print("\n*****")
    print("*****")
    print("Farkle!                               Welcome to")
    print("*****")
    print("*****")
    print("Rules:")
    print("1. Each player's turn starts by rolling all six dice.")
    print("2. Score points based on the combinations rolled (see scoring rules).")
```



```
print("*****  
*****")
```

Explanation: Displays information about the current player's turn, including their name, score, and the dice rolled.

Pure: Pure. It doesn't modify the game state and only prints information.

Totalised: Yes, it's totalised. It covers all possible turn information displays.

Roll dice function

```
def roll_dice(self):  
  
return self.dice.roll_dice()
```

Explanation: Rolls the dice for the current turn.

Pure: Impure. It interacts with the game state by rolling the dice.

Totalised: Yes, it's totalised. It handles all possible outcomes of rolling the dice.

Score calculate on the basis of points

```
def get_scoring_dice(self, dice):  
    scoring_dice = {'1': 0, '5': 0}  
    for die in dice:  
        if die == 1:  
            scoring_dice['1'] += 1  
        elif die == 5:  
            scoring_dice['5'] += 1  
    return scoring_dice  
  
def calculate_score(self, dice):  
    score = 0  
    scoring_dice = self.get_scoring_dice(dice)  
    if scoring_dice['1'] == 1:
```

```
        score += 100
    elif scoring_dice['1'] == 2:
        score += 200
    elif scoring_dice['1'] == 3:
        score += 1000
    elif scoring_dice['1'] == 4:
        score += 2000
    elif scoring_dice['1'] == 5:
        score += 3000
    elif scoring_dice['1'] == 6:
        score += 4000
```

```
    if scoring_dice['5'] == 1:
        score += 50
    elif scoring_dice['5'] == 2:
        score += 100
    elif scoring_dice['5'] == 3:
        score += 500
    elif scoring_dice['5'] == 4:
        score += 1000
    elif scoring_dice['5'] == 5:
        score += 1500
    elif scoring_dice['5'] == 6:
        score += 2000
```

```
    for num in range(1, 7):
        if dice.count(num) >= 3:
            if num == 1:
                score += 1000 * (dice.count(num) - 2)
            else:
                score += 100 * num * (dice.count(num) - 2)
```

```
    if len(set(dice)) == 6:
```

```

        score += 1500

    elif len(set(dice)) == 3 and all(dice.count(num) == 2 for num in set(dice)):

        score += 1500

    return score

```

Explanation: Calculates the score for the current roll based on the dice outcomes.

Pure: Impure. It interacts with the game state by calculating the score.

Totalised: Yes, it's totalised. It covers all possible combinations of dice rolls.

Executing turn

```

def take_turn(self):
    dice_rolled = self.roll_dice()

    print("\n*****")

    print(f"* {self.current_player.name} rolled: {' '.join(map(str, dice_rolled))}:")

    score = self.calculate_score(dice_rolled)

    if score == 0:
        print("* Farkle! No scoring combinations. Turn ends with no")
        print("*****")

        return 0

    print("To stop playing press exit")
    choice = getPlayerMenuInput()
    while choice.lower() not in ('r', 'e'):
        print("Invalid choice. Please enter 'r' to roll again or 'e' to end turn.")
        choice = getPlayerMenuInput()

```

```

    if choice == 'e':
        self.current_player.score += score

        print(f"* Turn ended. Total score for this turn: {self.current_player.score}")

```

```

        print("*****")
        return score
    elif choice == 'r':
        self.current_player.score += score
        return self.take_turn()

```

Explanation: Manages a player's turn, including rolling the dice, calculating the score, and determining the next action (roll again or end turn).

Pure: Impure. It interacts with the game state by managing the turn.

Totalised: Yes, it's totalised. It covers all possible turn outcomes.

Playing game

```

def play_game(self):
    self.get_player_names()
    self.display_rules()
    while all(player.score < self.winning_score for player in self.players):
        for player in self.players:
            self.current_player = player
            self.display_turn_info()
            self.take_turn()

    max_score = max(player.score for player in self.players)
    winners = [player for player in self.players if player.score == max_score]
    print("\n*****")
    if len(winners) == 1:
        print(f"* {winners[0].name} wins with {max_score} points!")
    else:
        print(f"* It's a tie between: ")
        for winner in winners:
            print(f"* {winner.name: <75} *")

```

```

playing!
print("*      Game      over.      Thanks      for
      *")

print("*****
*****")

```

Explanation: Manages the entire game, including player setup, displaying rules, and executing turns until the winning condition is met.

Pure: Impure. It interacts with the game state by managing the game.

Totalised: Yes, it's totalised. It covers all possible game outcomes.

Output

```

Enter the number of players: 2
Enter your name: Michal
Enter your name: John

*****
*                               Welcome to Farkle!                               *
*****

Rules:
1. Each player's turn starts by rolling all six dice.
2. Score points based on the combinations rolled (see scoring rules).
3. Choose to bank points or continue rolling.
4. If no scoring dice are rolled, the turn ends with no points (Farkle).
5. The game continues until a player reaches the winning score.
Scoring:
Single 1: 100 points
Single 5: 50 points
Three of a kind: 100 times the face value of the die
Four of a kind: 1000 times the face value of the die
Five of a kind: 2000 times the face value of the die

```

```

*****
*
*                               Michal's turn                               *
*****
* Total Score: 0
* Dice: 0, 0, 0, 0, 0, 0
*****
*****
* Michal rolled: 5, 4, 2, 6, 2, 6
To stop playing press exit
Roll again (r) or end turn (e)? e
* Turn ended. Total score for this turn: 50
*****
*****
*                               John's turn                               *
*****
* Total Score: 0
* Dice: 5, 4, 2, 6, 2, 6
*****
*****

```

[illegible]


```
*****
*                                     *
*                               John's turn                               *
*                               *                                         *
*****
* Total Score: 9650                                                         *
* Dice: 5, 3, 1, 6, 1, 6                                                 *
*****

*****
* John rolled: 6, 5, 1, 1, 1, 6                                           *
To stop playing press exit
Roll again (r) or end turn (e)? e
* Turn ended. Total score for this turn: 11700                             *
*****

*****
* John wins with 11700 points!                                           *
* Game over. Thanks for playing!                                         *
*****
```

Testing

Both manual and automated tests play crucial roles in ensuring that the software functions as intended, adhering to the specifications outlined in Gherkin and other planning documents. Manual testing involves executing the software to evaluate whether it produces the expected outputs in response to specific user inputs. This process allows testers to interact with the software, simulating real-world usage scenarios and verifying its behavior.

Let's begin by conducting manual tests, systematically testing each feature and functionality to ensure they meet the defined requirements. During manual testing, testers explore the software's user interface and functionality, providing input and observing the corresponding outputs. They meticulously validate each component, ensuring that it behaves as expected and that there are no deviations from the defined specifications. In addition to manual testing, automated tests are essential for efficiently and thoroughly verifying the software's behavior. Automated tests are scripted tests that can be executed automatically, allowing for repetitive and systematic validation of the software's functionality. These tests can be run repeatedly, ensuring consistent results and detecting regressions or unintended changes in the codebase. By combining manual and automated testing approaches, we can ensure comprehensive test coverage and maintain the quality and reliability of the software.

Manual Testing

Test Case ID	1	Passed
Software Feature:	Player Name Input	
Steps to Do	Expected Output	Actual Result

Test Case ID	1	Passed
1. Enter a valid player name.	Prompt to enter the number of players.	<pre> Enter the number of players: 2 Enter your name: Michal Enter your name: John ***** * Welcome to Farkle! ***** </pre>

Test Case ID	2	Passed
Software Feature:	Player Name Input	
Steps to Do	Expected Output	Actual Result
1. Enter a player name with special characters (e.g., "John#").	"Invalid input. Enter your name: " prompt to enter the name again without special characters.	<pre> Enter the number of players: 2 Enter your name: john# Invalid input. Please enter only alphabetical characters Enter your name: </pre>
2. Enter a player name with numbers (e.g., "Player123").	"Enter your name: " prompt to enter the name again without numbers.	<pre> Enter the number of players: 2 Enter your name: player123 Invalid name. Please enter only alphabetical characters Enter your name: </pre>

Test Case ID	3	Passed
Software Feature:	Rules Display	
Steps to Do	Expected Output	Actual Result

Test Case ID	3	Passed
1. Enter name correctly and Run the game.	Rules are displayed after name entered correctly.	<pre> Enter the number of players: 2 Enter your name: michal Enter your name: john ***** * Welcome to Farkle! * ***** Rules: 1. Each player's turn starts by rolling all six dice. 2. Score points based on the combinations rolled (see scoring rules) 3. Choose to bank points or continue rolling. 4. If no scoring dice are rolled, the turn ends with no points (Farkle) 5. The game continues until a player reaches the winning score. Scoring: Single 1: 100 points Single 5: 50 points Three of a kind: 100 times the face value of the die Four of a kind: 1000 times the face value of the die Five of a kind: 2000 times the face value of the die Six of a kind: 3000 times the face value of the die Straight (1-6): 1500 points Three pairs: 1500 points Example: 1-1-1-5-5-5 scores 1050 points Winning score is 10,000 points Let's start! </pre>

--	--	--

Test Case ID	4	Passed
Software Feature:	Dice Roll	
Steps to Do	Expected Output	Actual Result
1. Roll the dice.	A list of six random numbers between 1 and 6 show.	<pre> ***** * Michal rolled: 4, 4, 4, 6, 4, 4 To stop playing press exit Roll again (r) or end turn (e)? █ </pre>

Test Case ID	5	Passed
Software Feature:	Scoring	
Steps to Do	Expected Output	Actual Result

Test Case ID	5	Passed
1. Calculate score for a given set of dice.	A total score based on the dice combinations.	<pre>***** * Michal rolled: 4, 4, 4, 6, 4, 4 To stop playing press exit Roll again (r) or end turn (e)? e * Turn ended. Total score for this t *****</pre>

Test Case ID	6	Passed
Software Feature:	Turn End - End Turn	
Steps to Do	Expected Output	Actual Result
1. Choose to end the turn.	Turn ends, score is updated, and next player's turn starts.	<pre>***** * John rolled: 4, 6, 6, 5, 6, 4 To stop playing press exit Roll again (r) or end turn (e)? e</pre>

Test Case ID	7	Passed
Software Feature:	Turn End - Roll Again	
Steps to Do	Expected Output	Actual Result
1. Choose to roll again.	Dice are rolled again, and the turn continues.	<pre>***** * John rolled: 4, 6, 6, 5, 6, 4 To stop playing press exit Roll again (r) or end turn (e)? r ***** * John rolled: 6, 3, 4, 5, 6, 5 To stop playing press exit Roll again (r) or end turn (e)?</pre>

Test Case ID	7	Passed
Test Case ID	8	Passed
Software Feature	Game End - Single Winner	
Steps to Do	Expected Output	Actual Result
1. One player reaches the winning score.	Winner's name and score are displayed.	<pre> * John rolled: 5, 5, 1, 3, 2, 6 To stop playing press exit Roll again (r) or end turn (e)? e * Turn ended. Total score for this ***** ***** * Michal wins with 10050 points! * Game over. Thanks for playing! </pre>
Test Case ID	10	Passed
Software Feature:	Invalid Menu Choice	
Steps to Do	Expected Output	Actual Result
1. Enter an invalid menu choice during a turn.	"Invalid choice. Please enter 'r' to roll again or 'e' to end turn." message is displayed until a valid choice is entered.	<pre> * michal's turn ***** * Total Score: 0 * Dice: 0, 0, 0, 0, 0, 0 ***** ***** * michal rolled: 1, 3, 1, 1, 6, 6 To stop playing press exit Roll again (r) or end turn (e)? t Invalid choice. Please enter 'r' to roll again or 'e' to end turn. </pre>

Test Case ID	11	Passed
Software Feature:	Full Game	
Steps to Do	Expected Output	Actual Result
1. Play a full game with multiple turns and players.	Game continues until a winner is declared.	<pre> ***** * ***** * Total Score: 0 * Dice: 1, 3, 1, 1, 6, 6 ***** ***** * john rolled: 4, 1, 1, 4, 5, To stop playing press exit Roll again (r) or end turn (e) * Turn ended. Total score for ***** ***** * ***** * Total Score: 2000 * Dice: 4, 1, 1, 4, 5, 3 ***** ***** * michal rolled: 6, 6, 3, 2, To stop playing press exit Roll again (r) or end turn (e) </pre>

Automated testing:

For automated testing of Farkle, the "unittest" library is the best choice and highly recommended. This library is a built-in testing framework in Python that offers a structured and organized approach to writing test cases for your code. With "unittest," you can construct comprehensive test suites, run tests efficiently, and make assertions about the behavior of your code.

The "unittest" library provides a rich set of tools for testing Python code, making it easy to define test cases and verify the correctness of your software. It offers a wide range of assertion methods to check various conditions and behaviors, such as equality, inequality, truthiness, containment, and more. Additionally, "unittest" allows you to group related test cases into test classes and organize them into test suites, providing a flexible and scalable framework for testing your codebase.

Test by giving valid input

```
import unittest
from unittest.mock import patch
from io import StringIO
from frakle3 import FarklePlayer, FarkleGame, getPlayerNameInput,

class TestFarkleGame(unittest.TestCase):
    def test_getPlayerNameInput_valid(self):
        with patch('builtins.input', side_effect=['Alice']):
            name = getPlayerNameInput()
            self.assertEqual(name, 'Alice')
```

Test by giving invalid input

```
def test_getPlayerNameInput_invalid(self):
    with patch('builtins.input', side_effect=['123', 'Alice']):
        with patch('sys.stdout', new=StringIO()) as fake_out:
            name = getPlayerNameInput()
            self.assertEqual(name, 'Alice')
            self.assertEqual(fake_out.getvalue().strip(), "Invalid name. Please enter on")
```

Testing players menu input when rolling the dice

```
def test_getPlayerMenuInput_roll(self):
    with patch('builtins.input', side_effect=['r']):
        choice = getPlayerMenuInput()
        self.assertEqual(choice, 'r')
```

Testing whole game

```

def test_game_play(self):
    with patch('builtins.input', side_effect=['1', 'Alice', 'r', 'e']):
        with patch('sys.stdout', new=StringIO()) as fake_out:
            game = FarkleGame(winning_score=10000)
            game.play_game()
            output = fake_out.getvalue()
            self.assertIn("Welcome to Farkle!", output)
            self.assertIn("Alice's turn", output)
            self.assertIn("Total score:", output)
            self.assertIn("Dice:", output)
            self.assertIn("Farkle! No scoring combinations.", output)
            self.assertIn("Turn ended. Total score for this turn:", output)
            self.assertIn("wins with", output)
            self.assertIn("Game over. Thanks for playing!", output)

```

Running the code and output of tests:

```

Enter the number of players: 2
Enter your name: Michal
Enter your name: Hobs

*****
*                               Welcome to Farkle!                               *
*****

Rules:
1. Each player's turn starts by rolling all six dice.
2. Score points based on the combinations rolled (see scoring rules).
3. Choose to bank points or continue rolling.
4. If no scoring dice are rolled, the turn ends with no points (Farkle).
5. The game continues until a player reaches the winning score.

Scoring:
Single 1: 100 points
Single 5: 50 points
Three of a kind: 100 times the face value of the die
Four of a kind: 1000 times the face value of the die
Five of a kind: 2000 times the face value of the die
Six of a kind: 3000 times the face value of the die
Straight (1-6): 1500 points
Three pairs: 1500 points
Example: 1-1-1-5-5-5 scores 1050 points
Winning score is 10,000 points

```



```
*****
*
*                               Hobs's turn
*****
* Total Score: 0
* Dice: 4, 2, 2, 5, 6, 3
*****

*****
* Hobs rolled: 2, 2, 4, 4, 1, 5
To stop playing press exit
Roll again (r) or end turn (e)? e
* Turn ended. Total score for this turn: 150
*****

*****
*                               Michal's turn
*****
* Total Score: 50
* Dice: 2, 2, 4, 4, 1, 5
*****

*****
* Michal rolled: 4, 6, 3, 4, 3, 5
To stop playing press exit
Roll again (r) or end turn (e)? █
```

```

*****
*
*                                     Hobs's turn
*****
* Total Score: 9600
* Dice: 6, 5, 3, 3, 3, 5
*****

*****
* Hobs rolled: 3, 1, 5, 3, 1, 1
To stop playing press exit
Roll again (r) or end turn (e)? e
* Turn ended. Total score for this turn: 11650
*****

*****
* Hobs wins with 11650 points!
* Game over. Thanks for playing!
*****

```

Game is stopped after winner is declared.

Appendix A

Full automated testing code:

```

import unittest
from unittest.mock import patch
from io import StringIO
from frakle import FarklePlayer, FarkleGame, getPlayerNameInput, checkPlayerName, getPlayerMenuInput

class TestFarkleGame(unittest.TestCase):
    def test_getPlayerNameInput_valid(self):
        with patch('builtins.input', side_effect=['Alice']):
            name = getPlayerNameInput()
            self.assertEqual(name, 'Alice')

    def test_getPlayerNameInput_invalid(self):
        with patch('builtins.input', side_effect=['123', 'Alice']):
            with patch('sys.stdout', new=StringIO()) as fake_out:
                name = getPlayerNameInput()
                self.assertEqual(name, 'Alice')
                self.assertEqual(fake_out.getvalue().strip(), "Invalid name. Please enter only alphabetical character")

    def test_checkPlayerName(self):
        self.assertFalse(checkPlayerName('Alice'))

    def test_getPlayerMenuInput_roll(self):
        with patch('builtins.input', side_effect=['r']):
            choice = getPlayerMenuInput()
            self.assertEqual(choice, 'r')

```

```

    def test_getPlayerMenuInput_roll(self):
        with patch('builtins.input', side_effect=['r']):
            choice = getPlayerMenuInput()
            self.assertEqual(choice, 'r')

    def test_getPlayerMenuInput_end(self):
        with patch('builtins.input', side_effect=['e']):
            choice = getPlayerMenuInput()
            self.assertEqual(choice, 'e')

    def test_game_play(self):
        with patch('builtins.input', side_effect=['1', 'Alice', 'r', 'e']):
            with patch('sys.stdout', new=StringIO()) as fake_out:
                game = FarkleGame(winning_score=10000)
                game.play_game()
                output = fake_out.getvalue()
                self.assertIn("Welcome to Farkle!", output)
                self.assertIn("Alice's turn", output)
                self.assertIn("Total score:", output)
                self.assertIn("Dice:", output)
                self.assertIn("Farkle! No scoring combinations.", output)
                self.assertIn("Turn ended. Total score for this turn:", output)
                self.assertIn("wins with", output)
                self.assertIn("Game over. Thanks for playing!", output)

if __name__ == '__main__':
    unittest.main()

```

T4 Git version control discussion

Documentation greatly benefits projects by providing a clear explanation of what a codebase does and how it can be used (Meza, 2018). It also assists secondary developers in understanding rules, approaches, naming conventions, comments, etc., thereby promoting maintainable code. Commits in Git serve as checkpoints in the project's history. Each commit captures the state of the codebase at a specific moment, enabling developers to revert back to previous versions if necessary. This functionality is invaluable for the Farkle project as it allows for easy tracking of changes and quick recovery from errors or unintended modifications.

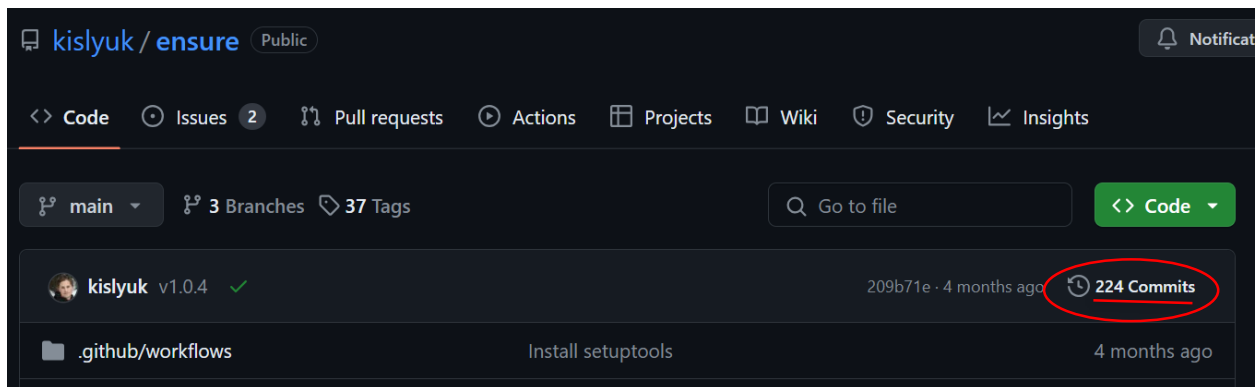
Pushing commits to the Git repository is how changes made on a local machine are shared with others. It ensures that all team members have access to the latest version of the codebase. Similarly, pulling updates the local machine with changes made by others, ensuring everyone is working with the most recent version. Pull requests are especially useful for team collaboration on the Farkle project. They allow developers to propose changes, have them reviewed by peers, and then merge them into the main repository. This process ensures that code changes are thoroughly examined before being integrated, maintaining code quality and stability.

Case study

For a case study of git, I study python unittest help to do automated testing of my project.

<https://github.com/reactjs/react-docgen>











The commit history for kislyuk can be easily accessed by clicking the commits symbol when navigating GitHub, underlined in red in figure.



The commit history of a repository is presented in chronological order, with the most recent commit listed at the top of the page. Users can navigate through the commit history by scrolling down or selecting the "older" option to view earlier commits. In the case of the Kislyuk repository, the oldest commit, labeled "initial commit," marks the inception of the project.

Each commit in the history represents a snapshot of the project's state at a specific point in time. These snapshots capture changes made to the codebase, including additions, modifications, and deletions of files. By examining the commit messages and changes associated with each commit, users can gain

insights into the evolution of the project over time. Additionally, commits often include descriptive messages that provide context and details about the changes made. These messages serve as documentation for developers, explaining the purpose and rationale behind each modification. This documentation is invaluable for understanding the project's development history and the motivations behind specific changes.

 kislyuk v1.0.4 		209b71e · 4 months ago	 224 Commits
 .github/workflows	Install setuptools	4 months ago	
 docs	Apply black	4 months ago	
 ensure	Apply black	4 months ago	
 test	Use assertRaisesRegex instead of assertRaisesRegexp	4 months ago	
 .gitignore	Switch CI to GitHub Actions (#33)	3 years ago	
 Changes.rst	v1.0.4	4 months ago	
 LICENSE	Initial commit	11 years ago	
MANIFEST.in	Distribute license in source package	8 years ago	

References:

The value, benefits of code documentation is available at:

<https://swimm.io/learn/code-documentation/code-documentation-benefits-challenges-and-tips-for-success> Accessed [27/04/2024]

Frakle game playing rules blog:

<https://walnutstudiolo.com/blogs/blog/how-to-play-dice-farkle-ten-thousand-game-rules-of-play>
Accessed [27/04/2024]

Git guid:

<https://github.com/git-guides/git-clone> Accessed [27/04/2024]