**Initial Setup**

The development environment for this project will be set up on a windwos 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), allowing for multiple players to play simultaneously. Farkle is a dice game played with two or more players, where each player takes turns throwing dice to score points.

**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*={*Name*1​,*Name*2​,...,*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:

**Name={"Alice","Bob","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:

**𝑃𝑙𝑎𝑦𝑒𝑟𝑠=𝑠𝑒𝑞<𝑃𝑙𝑎𝑦𝑒𝑟1,𝑙𝑎𝑦𝑒𝑟2,...,𝑃𝑙𝑎𝑦𝑒𝑟𝑛>**

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.

For example:

**𝑃𝑙𝑎𝑦𝑒𝑟𝑠=𝑠𝑒𝑞<𝑃𝑙𝑎𝑦𝑒𝑟𝐴,𝑃𝑙𝑎𝑦𝑒𝑟𝐵>**

**Input model**

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

* *Input*=*seq*<*Char*>

**Player Model**

* The name of a player is a sequence of characters: 𝑁𝑎𝑚𝑒=𝑠𝑒𝑞𝐶ℎ𝑎𝑟*Name*=*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*

**Statistics**

Each player has a set of statistics associated with them based on their play history. Initially, these values are set to 0, and each win, loss, or draw by the specified player increments the value of the relevant parameter by 1.

The incrementing functions are named **addWin**, **addLoss**, and **addDraw** respectively.

𝑎𝑑𝑑𝑊𝑖𝑛:𝐼𝑛𝑡→𝐼𝑛𝑡*addWin*:*Int*→*Int*  
𝑎𝑑𝑑𝐿𝑜𝑠𝑠:𝐼𝑛𝑡→𝐼𝑛𝑡*addLoss*:*Int*→*Int*  
𝑎𝑑𝑑𝐷𝑟𝑎𝑤:𝐼𝑛𝑡→𝐼𝑛𝑡*addDraw*:*Int*→*Int*

It is also possible to get and set these values. Below is an axiomatic definition for the win value:

𝑔𝑒𝑡𝑊𝑖𝑛𝑠:𝑆𝑡𝑎𝑡𝑖𝑠𝑡𝑖𝑐𝑠→𝐼𝑛𝑡*getWins*:*Statistics*→*Int*  
𝑠𝑒𝑡𝑊𝑖𝑛:(𝐼𝑛𝑡,𝑆𝑡𝑎𝑡𝑖𝑠𝑡𝑖𝑐𝑠)→𝑆𝑡𝑎𝑡𝑖𝑠𝑡𝑖𝑐𝑠*setWins*:(*Int*,*Statistics*)→*Statistics*

**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("\*                             Welcome to Farkle!                                 \*")

        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("3. Choose to bank points or continue rolling.")

        print("4. If no scoring dice are rolled, the turn ends with no points (Farkle).")

        print("5. The game continues until a player reaches the winning score.")

        print("Scoring:")

        print("Single 1: 100 points")

        print("Single 5: 50 points")

        print("Three of a kind: 100 times the face value of the die")

        print("Four of a kind: 1000 times the face value of the die")

        print("Five of a kind: 2000 times the face value of the die")

        print("Six of a kind: 3000 times the face value of the die")

        print("Straight (1-6): 1500 points")

        print("Three pairs: 1500 points")

        print("Example: 1-1-1-5-5-5 scores 1050 points")

        print("Winning score is 10,000 points")

        print("Let's start!\n")

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

Explanation: Displays the rules of the Farkle game.

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

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

**Display players name whose turn**

    def display\_turn\_info(self):

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

        print(f"\*                                 {self.current\_player.name}'s turn                               \*")

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

        print(f"\* Total Score: {self.current\_player.score: <72} \*")

        print(f"\* Dice: {', '.join(map(str, self.dice.dice)): <68} \*")

        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)): <59} \*")

        score = self.calculate\_score(dice\_rolled)

        if score == 0:

            print("\* Farkle! No scoring combinations. Turn ends with no points.                \*")

            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: <51} \*")

            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("\* It's a tie between:                                                            \*")

            for winner in winners:

                print(f"\* {winner.name: <75} \*")

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

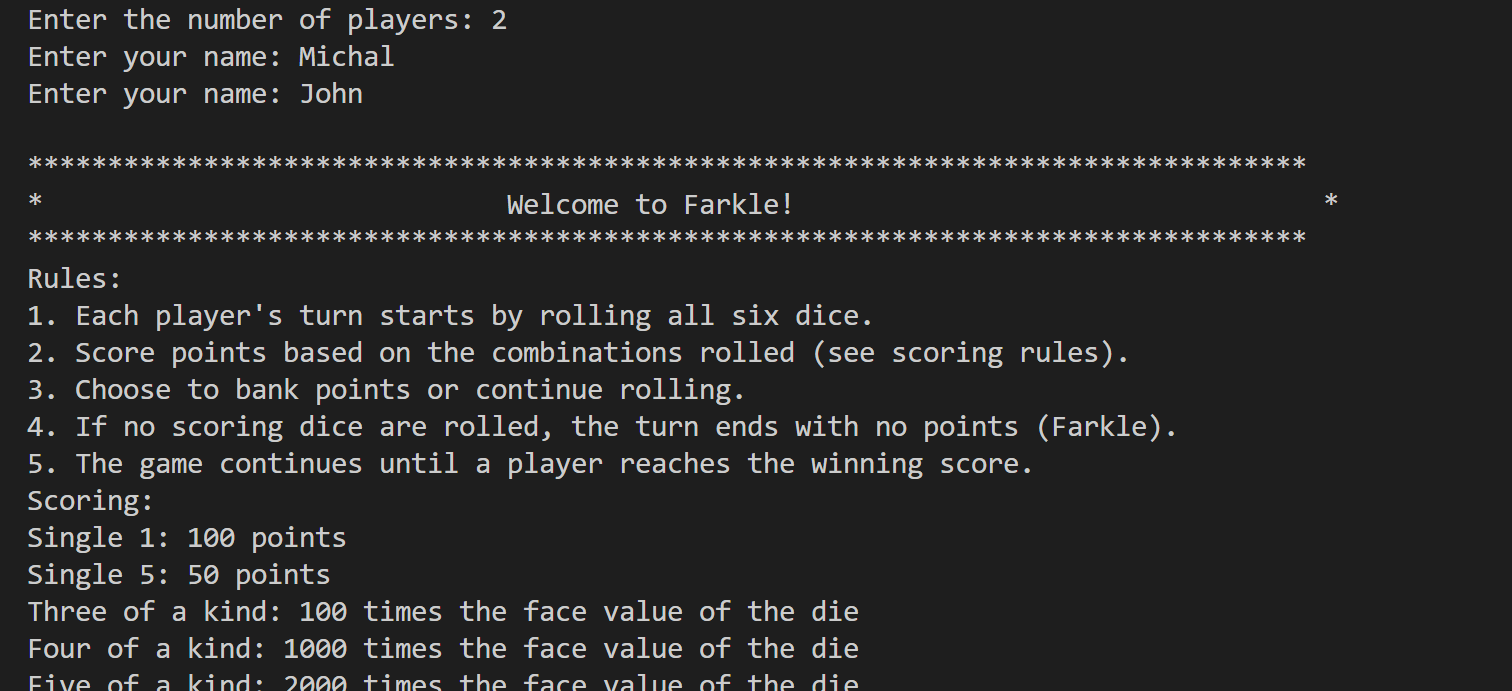
        print("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

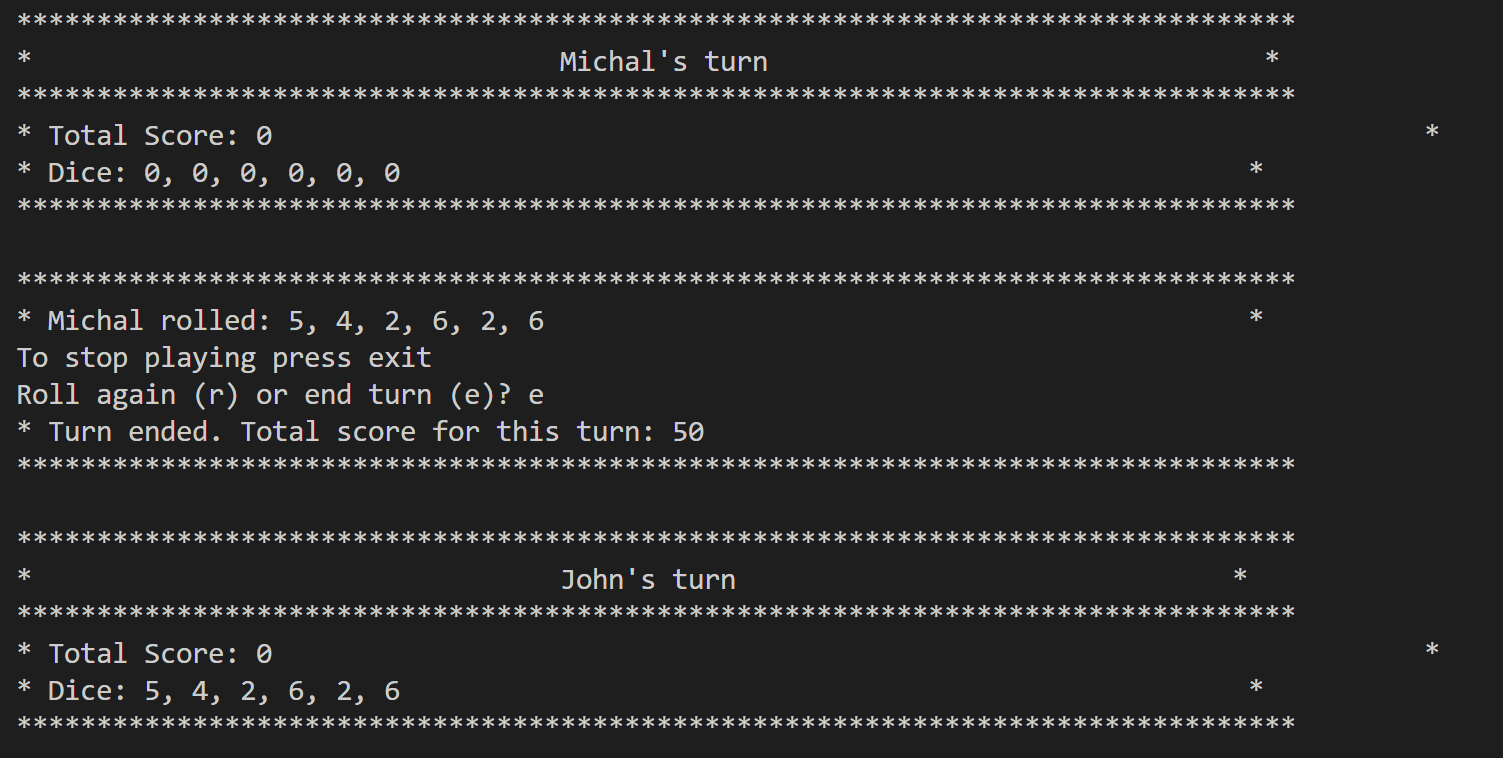
Explanation: Manages the entire game, including player setup, displaying rules, and executing turns until the winning condition is met.

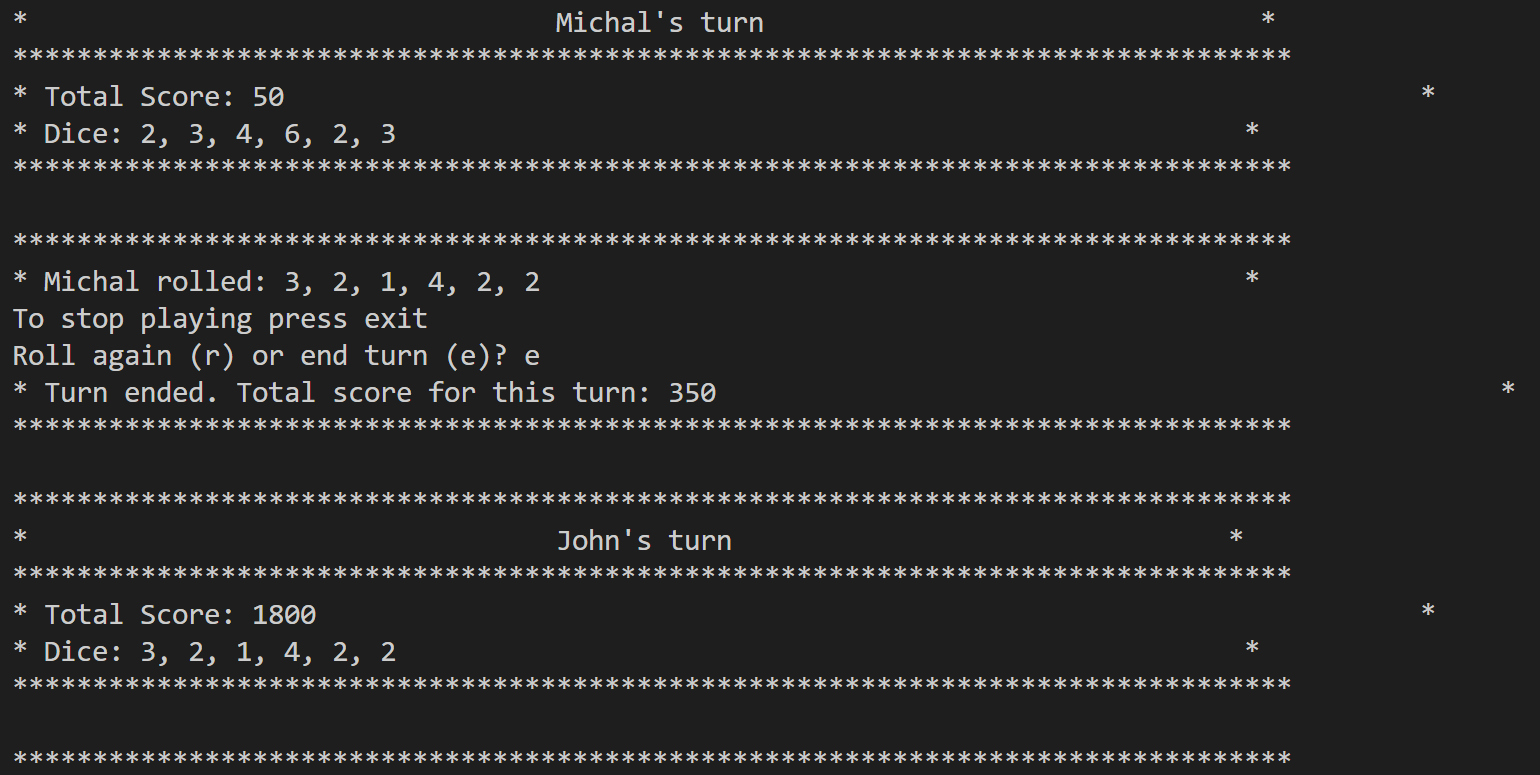
Pure: Impure. It interacts with the game state by managing the game.

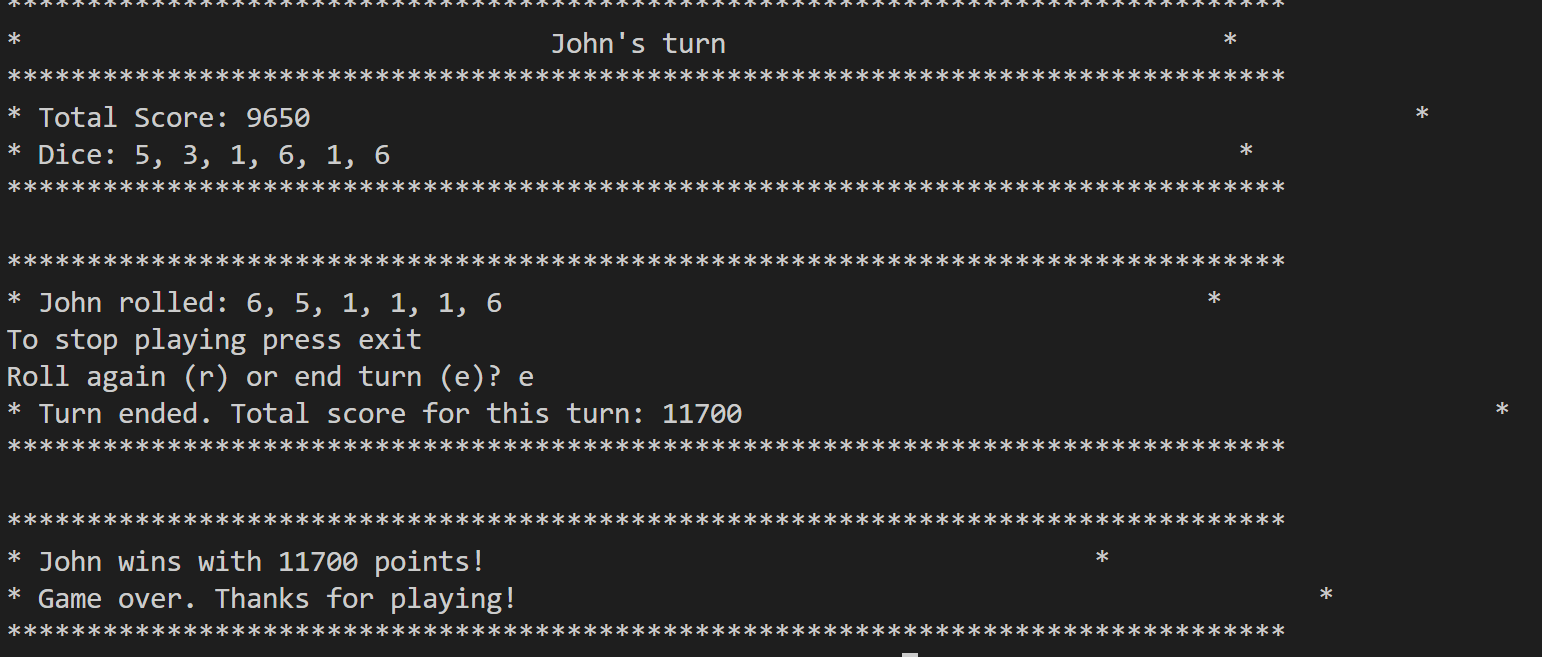
Totalised: Yes, it's totalised. It covers all possible game outcomes.

**Output**









**Testing**

Both manual and automated tests are necessary to ensure that the features work as expected according to the Gherkin specifications and other planning components. Manual testing involves using the software implementation to assess whether expected outputs are returned for specific user inputs. Lets start one by one.

**Manual Testing**

| **Test Case ID** | **1** | **Passed** |
| --- | --- | --- |
| ****Software Feature:**** | Player Name Input |  |
| ****Steps to Do**** | ****Expected Output**** | ****Actual Result**** |
| 1. Enter a valid player name. | Prompt to enter the number of players. |  |
|  |  |  |
|  |  |  |

| **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. |  |
| 2. Enter a player name with numbers (e.g., "Player123"). | "Enter your name: " prompt to enter the name again without numbers. |  |

| **Test Case ID** | **3** | **Passed** |
| --- | --- | --- |
| ****Software Feature:**** | Rules Display |  |
| ****Steps to Do**** | ****Expected Output**** | ****Actual Result**** |
| 1. Enter name correctly and Run the game. | Rules are displayed after name entered correctly. |  |
|  |  |  |

| **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. |  |
|  |  |  |

| **Test Case ID** | **5** | **Passed** |
| --- | --- | --- |
| ****Software Feature:**** | Scoring |  |
| ****Steps to Do**** | ****Expected Output**** | ****Actual Result**** |
| 1. Calculate score for a given set of dice. | A total score based on the dice combinations. |  |

| **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. |  |
|  |  |  |

| **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. |  |
|  |  |  |

| **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. |  |

| **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. |  |
|  |  |  |

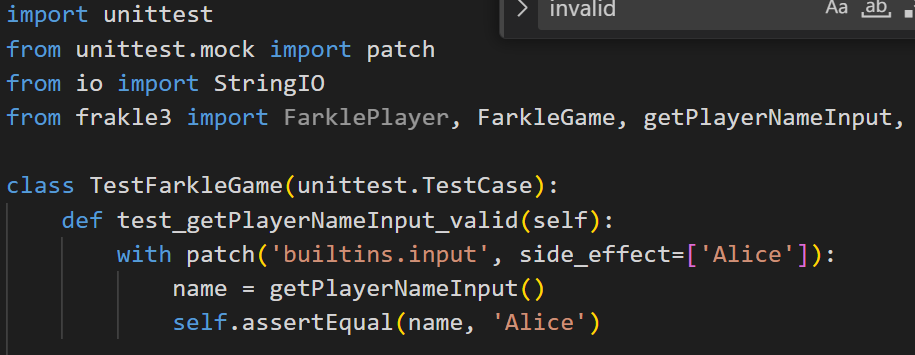
| **Test Case ID** | **11** | **Passed** |
| --- | --- | --- |
| ****Software Feature:**** | Full Game |  |
| ****Steps to Do**** | ****Expected Output**** | ****Actual Result**** |
| 1. Play a full game with multiple turns and players. | Game continues until a winner is declared. |  |

**Automated testing:**

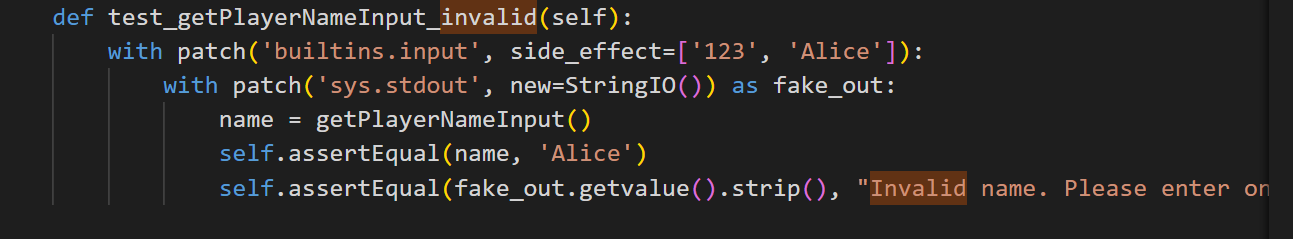
For automated testing of frakle “unittest” library is best and using it.

The unittest library in Python is a built-in testing framework that allows you to write test cases for your code in a structured and organized manner. It provides a set of tools for constructing and running tests, as well as making assertions about the behavior of your code.

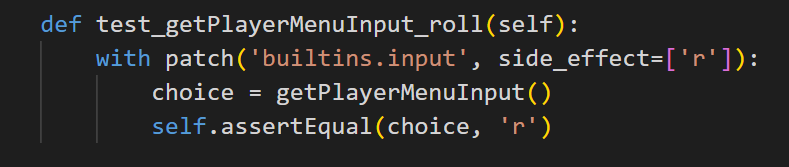
Test by giving valid input



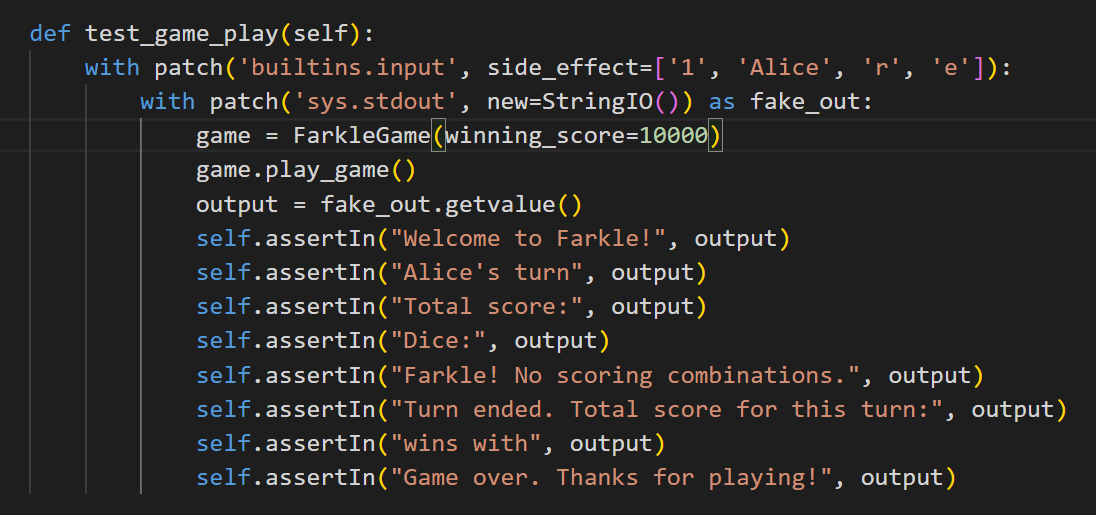
Test by giving invalid input



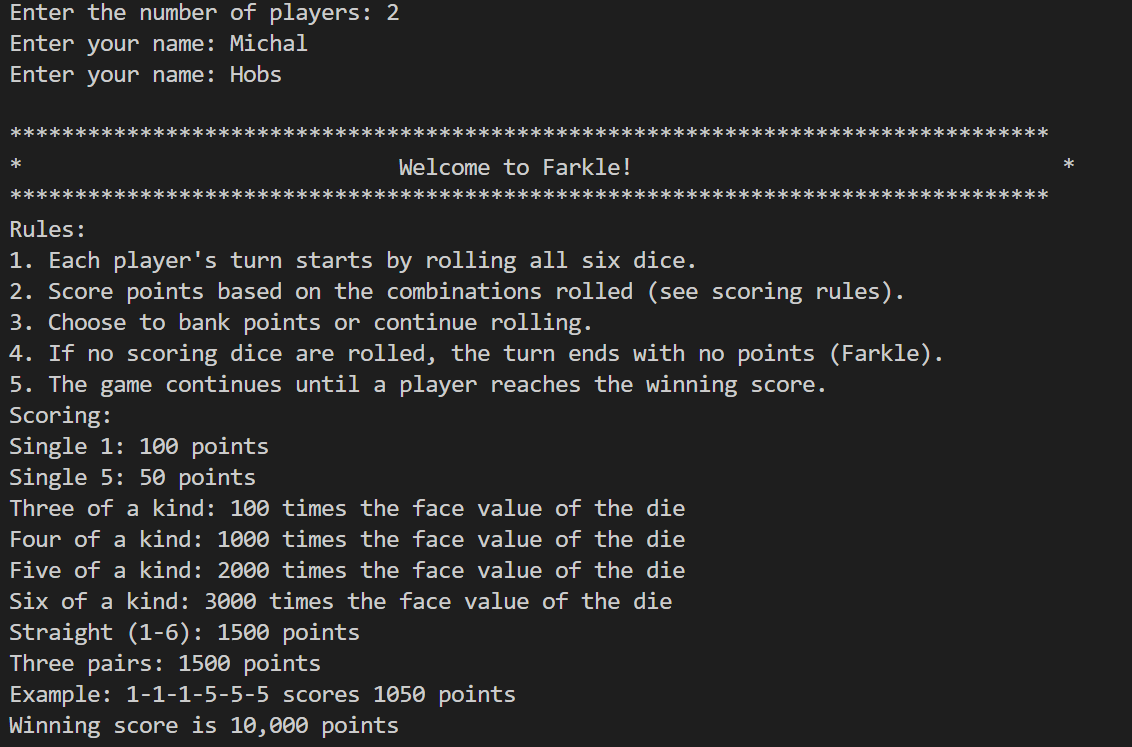
Testing players menu input when rolling the dice

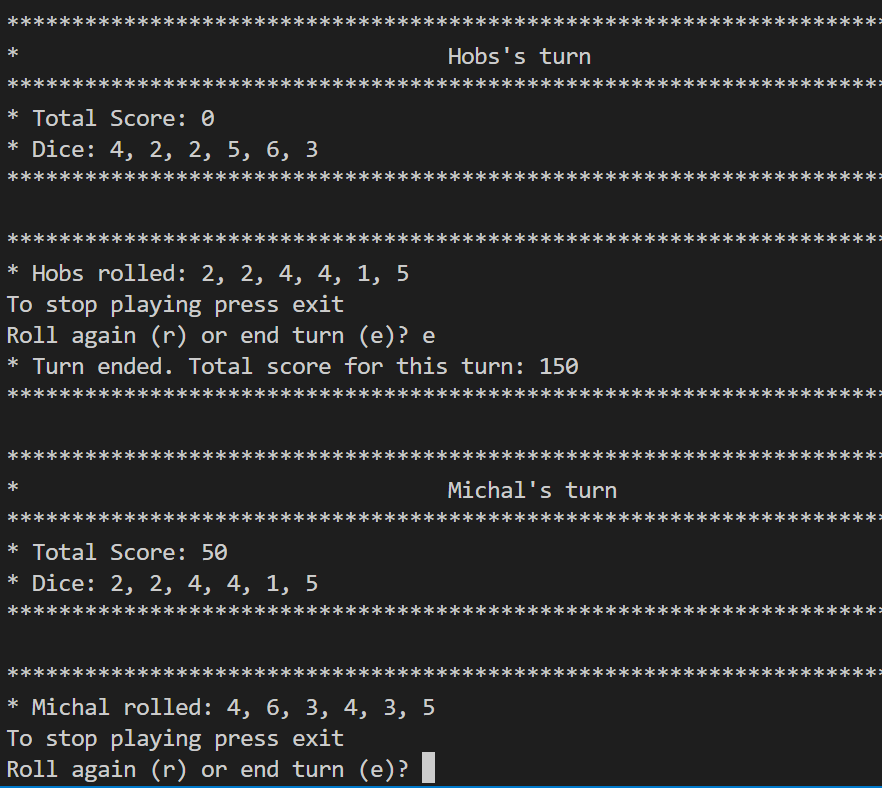


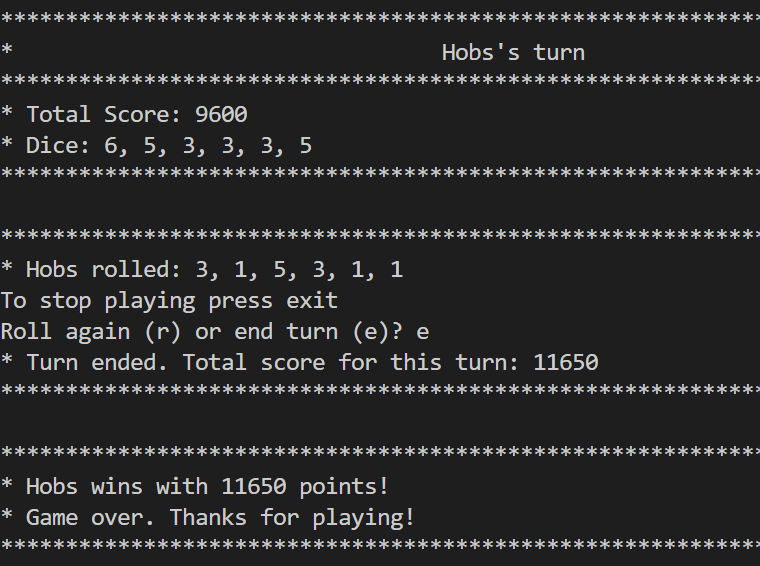
**Testing whole game**



Running the code and output of tests:







Game is stopped after winner is declared.

**Appendix A**

**Full automated testing code:**





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.

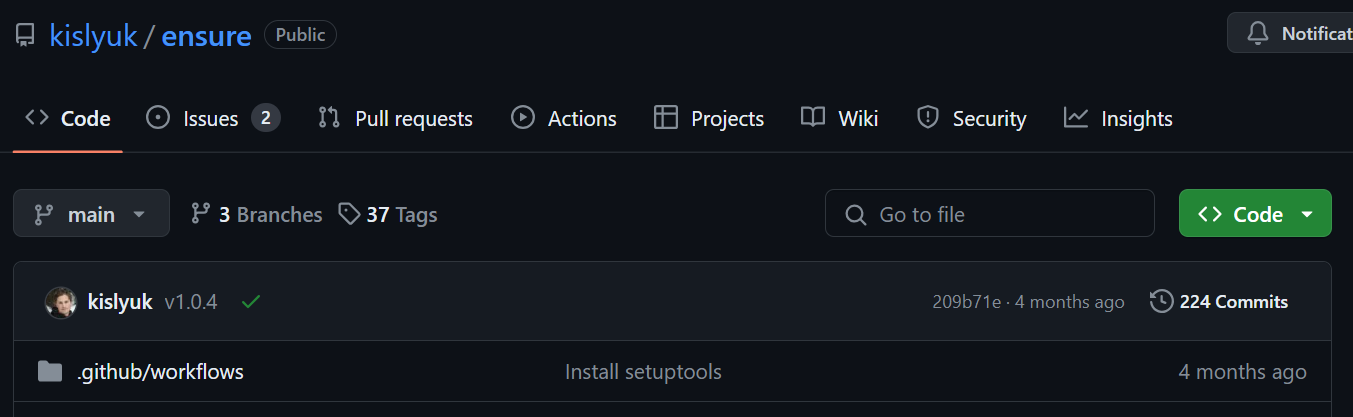
Pull requests are especially useful for team collaboration on the Farkle project. They allow developers to propose changes, have them reviewed by peers, and then merge them into the main repository. This process ensures that code changes are thoroughly examined before being integrated, maintaining code quality and stability.

**Case study**

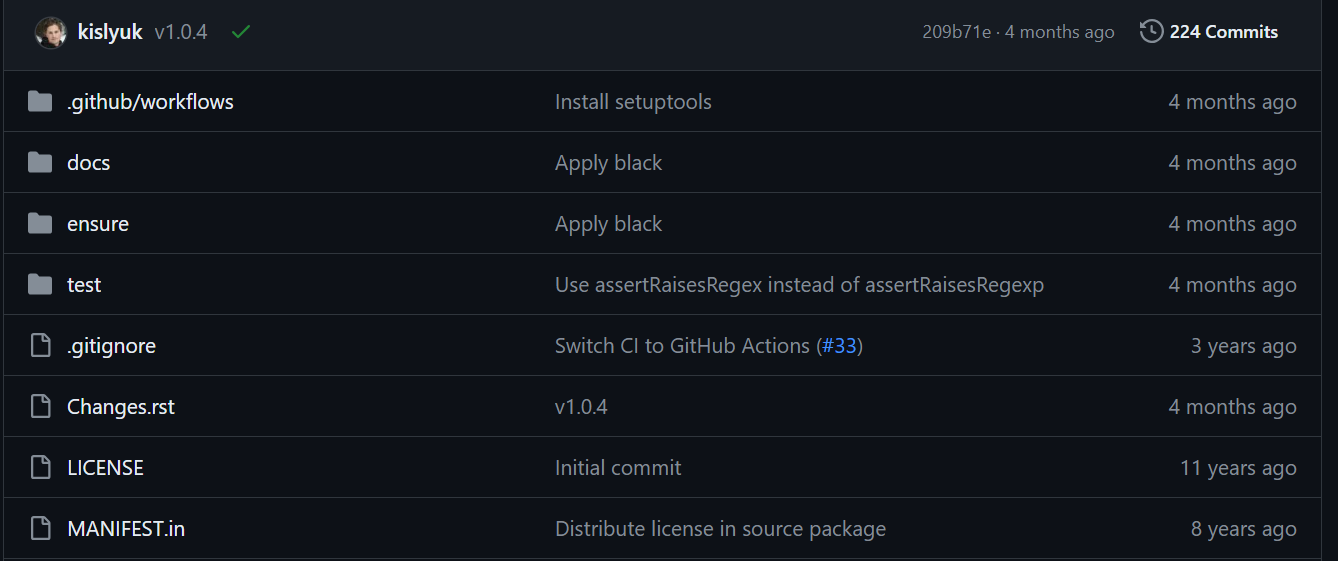
For a case study of git, I study python unittest help to do automated testing of my project.

<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.



Commits are displayed in chronological order, with the latest commit appearing at the top of the list. By scrolling down or selecting "older," you can view earlier commits in the repository. In the case of the Kislyuk repository, the oldest commit is titled "initial commit."



**Refrences:**

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]