**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 Adventure Game is a captivating text-based interactive adventure that immerses players in a thrilling journey of exploration and discovery. Within this game, players embark on an adventure through a series of intricately designed rooms, each with its unique atmosphere, challenges, and secrets waiting to be unveiled. The game unfolds across three main rooms: the Start Room, the Hallway, and the Kitchen, each meticulously crafted to offer a distinct experience and push players to unravel the mysteries hidden within.

At the heart of the Adventure Game's allure lies its captivating visual presentation, brought to life through ASCII art boxes that serve as immersive representations of each room. These ASCII art boxes provide players with a vivid visual backdrop, allowing them to visualize their surroundings and immerse themselves fully in the game's narrative. From the cozy confines of the Start Room to the mysterious depths of the Hallway and the bustling activity of the Kitchen, each room's ASCII art design captures its unique ambiance and sets the stage for thrilling adventures.As players navigate through the rooms, they encounter various challenges, puzzles, and interactive elements that engage their minds and fuel their curiosity. From collecting items scattered throughout the rooms to using them strategically to overcome obstacles and unlock new areas, the Adventure Game offers a rich tapestry of gameplay experiences that keep players engaged and entertained. With every decision they make and every discovery they uncover, players delve deeper into the immersive world of the game, forging their path and shaping their adventure in captivating ways.

**Rules of Text adventure game**: The rules of Text adventure gameimplemented in this project will adhere to the standard rules, including:

**Exploration:** Players navigate through different rooms by using directional commands such as "go north", "go south", etc.

**Item Collection:** Players can pick up items found in each room using the command "take <item>". Items can be stored in the player's inventory for later use.

**Item Usage:** Players can use items in their inventory or within the room by typing "use <item>". This action can trigger events, solve puzzles, or unlock new areas.

**Room Exits:** Each room has exits leading to other rooms. Players can explore these exits by typing "go <direction>".

**Room Descriptions:** Upon entering a room, the game provides a description of the room, including any items present and the available exits.

**Inventory Management:** Players can view their inventory at any time to see which items they are currently carrying.

**Game Loop:** The game continues until the player decides to quit by typing "quit". Otherwise, the player can keep exploring, collecting items.

**Help Command:** Typing "help" displays a list of available commands and instructions on how to play the game.

**Interaction Feedback:** The game provides feedback to the player after each action, confirming the success or failure of the action and updating the game state accordingly.

**Visual Representation:** Rooms are visually represented using ASCII art, providing a unique and immersive experience for the player.

Requirements Analysis for Text Adventure Project:

**User Interface:**

1. The system should have a command-line interface (CLI) for user interaction.
2. Players should be able to input commands to interact with the game.

**Game Initialization:**

1. The system should allow players to start a new game.
2. Players should have the option to play solo or with multiple players.

**Player Interaction:**

1. Each player should be able to input commands during their turn.
2. Commands should include actions like moving between rooms, taking items, and using items.

**Game Mechanics:**

1. Players should navigate between different rooms using directional commands (e.g., north, south).
2. Each room should contain items that players can interact with.
3. Players should be able to pick up items found in rooms.
4. Items should have uses and effects when used by the player.

**Player Management:**

1. Players should be able to input their names at the start of the game.
2. The system should support multiple players, each with their own inventory.

**Game Progression:**

1. The game should determine win conditions for each player.
2. Win conditions might include finding a specific item, reaching a certain room, or solving a puzzle.

**Game State Management:**

1. The system should keep track of the current room for each player.
2. Inventory items for each player should be managed and tracked.
3. Used items should be recorded and updated accordingly.

**Feedback and Output:**

1. The system should provide descriptive feedback after each player action.
2. Feedback should include information on the current room, available exits, and any items found or used.

**Help and Guidance:**

1. Players should have access to a help command that provides instructions and available commands.
2. The system should guide players through the game's objectives and mechanics.

**Scalability and Extensibility:**

1. The system should be easily extensible to add new rooms, items, and game mechanics.
2. It should support future updates and expansions without major modifications to the codebase.

**Behavior Driven Development (Gherkin Specifications)**

|  |
| --- |
| Feature: Exploring Rooms in Adventure Game |
| AS A CLI USER/PLAYER: I want to enter player names So that each player can be identified in the game |
| Scenario: Player starts the game  Given the game has started  When the game initializes  Then the game displays the current room with its description |
| Scenario: Player moves to a valid adjacent room |
| Given the player is in the start room  When the player chooses to go north  Then the player moves to the hallway room And the game displays the hallway room with its description |
| Scenario: Player moves to an invalid room |
| Given the player is in the kitchen room  When the player chooses to go east  Then the game displays a message that there is no room in that direction |
| Scenario: Player picks up an item |
| Given the player is in the start room And there is a key in the start room  When the player chooses to take the key  Then the player picks up the key And the game displays the start room without the key |
| Scenario: Player tries to pick up a non-existent item |
| Given the player is in the hallway room  When the player tries to take a sword  Then the game displays a message that there is no sword here |
| Scenario: Player uses an item from inventory |
| Given the player has a key in their inventory And the player is in the hallway room  When the player chooses to use the key  Then the game displays a message that the player used the key |
| Scenario: Player uses an item from the room |
| Given the player is in the kitchen room And there is a knife in the kitchen room  When the player chooses to use the knife  Then the game displays a message that the player used the knife |
| Scenario: Player tries to use a non-existent item |
| Given the player is in the start room  When the player tries to use a hammer  Then the game displays a message that the player doesn't have a hammer |
| Scenario: Player tries to use a non-existent item |
| Given the player is in the start room  When the player enters a command "jump"  Then the game displays a message that the command is invalid |
| Scenario: Player enters an invalid command |
| Given the player is in the start room  When the player enters a command "jump"  Then the game displays a message that the command is invalid |
| Scenario: Player requests help |
| Given the player is in the game  When the player types "help" command  Then the game displays a list of available commands and their descriptions |
| Scenario: Player quits the game |
| Given the player is in the game  When the player chooses to quit  Then the game displays a farewell message |
| Scenario: Player tries to move without specifying direction |
| Given the player is in the hallway room  When the player tries to go without specifying direction  Then the game displays a message that the direction is missing |

Data Model

**Input:**

* User Input (Standard Input)

**Output Message:**

* Welcome to the adventure game!
* Type 'help' for instructions.
* Current Room: [Room Name]
* [Room ASCII Art]
* Description: [Room Description]
* Exits: [Available Exits]
* You picked up the [item\_name]
* There is no [item\_name] here.
* You used the [item\_name]
* You don't have a [item\_name]
* Invalid command. Type 'help' for instructions.
* Available commands: go <direction>, take <item>, use <item>, help, quit
* Good luck!

**Error:**

* Error Values
  + Invalid argument
* Error Message
* Exit Code

**Room Model:**

Each room in the game is represented by a Room object.

Attributes:

name: Name of the room (string).

description: Description of the room (string).

exits: Dictionary representing exits from the room, where keys are directions (strings) and values are names of adjacent rooms (strings).

items: List of items present in the room (list of strings).

used\_items: List of items already used in the room (list of strings).

**Item Model:**

Items within the game are represented by strings.

**Player Model:**

Each player in the game is represented by a Player object.

Attributes:

inventory: List of items the player is carrying (list of strings).

**Game Model:**

The game itself is represented by a Game object.

Attributes:

player: Instance of the Player class.

current\_room: Current room the player is in (instance of Room).

rooms: Dictionary mapping room names to Room objects.

**Input Model:**

Players interact with the game by providing input, which is a sequence of characters.

Input: seq<char>

**Output Message Model:**

Output messages generated by the game during gameplay.

**Output\_Message=Ui=1n ​{Message[i]​}**

For example:

Output\_Message = {"Welcome to the adventure game!", "Type 'help' for instructions.", "Invalid command. Type 'help' for instructions.", ...}

**Game Command Model:**

In the Adventure Game, players have access to a variety of commands that enable them to interact with the game world, navigate through different rooms, and manipulate objects to progress in their journey. These commands are essential tools that players use to explore their surroundings, solve puzzles, and uncover the secrets hidden within each room.

Command: seq<string>

For example:

Command = {"go", "take", "use", "help", "quit"}

**Axiomatic Definitions and Functions**

Axiomatic Definitions:

**Room:**

A room represents a location within the game environment.

Each room has:

A name: Identifies the room.

A description: Provides information about the room.

Exits: Defines the possible directions the player can move from the room.

Items: Objects present in the room that the player can interact with.

Used items: Items that the player has interacted with in the room.

**Player:**

A player is the character controlled by the user.

Each player has:

An inventory: A collection of items the player is carrying.

**Game:**

The game manages the flow of the game and interactions between the player and the game environment.

It keeps track of:

* The current room: The room where the player currently is.
* Collection of rooms: All the rooms available in the game.

Functions:

**display\_info (Room):**

Displays the details of the current room, including ASCII art representation, description, and available exits.

**add\_item (Player):**

Adds an item to the player's inventory.

**remove\_item (Player):**

Removes an item from the player's inventory.

**add\_room (Game):**

Adds a new room to the game.

**start\_game (Game):**

Initializes the game by selecting a random room as the starting point and begins gameplay.

**display\_current\_room (Game):**

Displays information about the current room.

**move (Game):**

Moves the player to a new room based on the given direction.

**take\_item (Game):**

Allows the player to take an item from the current room and add it to their inventory.

use\_item (Game):

Allows the player to use an item from their inventory or the current room.

**play (Game):**

Main loop of the game where the player can input commands.

Handles movement, item interaction, and quitting the game.

T2 Implementation

Implementation of the program can now commence, focusing on two types of functions: pure and impure. Pure functions do not alter the program state outside their scope, while impure functions do. Additionally, there are totalized and non-totalized functions. Totalized functions handle all possible input-value pairs, whereas non-totalized functions do not. Whenever possible, it is preferable to create pure, totalized functions.

**Class: Room**

Description: Represents a room in the game environment.

class Room:

    def \_\_init\_\_(self, name, description, exits, items=None):

        self.name = name

        self.description = description

        self.exits = exits

        self.items = items if items else []

        self.used\_items = []

**Function: display\_info()**

 def display\_info(self):

        # Define the ASCII art for each room

        room\_art = {

            "start": [

                "+-------------------------------------------------------+",

                "|    Start room                                         |",

                "|                                                       |",

                "|   Items:                                              |",

                "|    -" + "\n|   - ".join(self.items).center(17) + "    ",

                "|                                                       |",

                "|   Used Items:                                         |",

                "|    -" + "\n|   - ".join(self.used\_items).center(17) +"",

                "|                                                       |",

                "+-------------------------------------------------------+"

            ],

            "hallway": [

                "+------------------------------------------------------+",

                "|   Hallway                                            |",

                "|                                                      |",

                "|   Items:                                             |",

                "|    -" + "\n|   - ".join(self.items).center(17) + "   ",

                "|                                                      |",

                "|   Used Items:                                        |",

                "|    -" + "\n|   - ".join(self.used\_items).center(17)+"",

                "|                                                      |",

                "+------------------------------------------------------+"

            ],

            "kitchen": [

                "+-----------------------------------------------------+",

                "|    Kitchen                                          |",

                "|                                                     |",

                "|   Items:                                            |",

                "|    -" + "\n|   - ".join(self.items).center(17) + "  ",

                "|                                                     |",

                "|   Used Items:                                       |",

                "|    -" + "\n|   - ".join(self.used\_items).center(17) +"",

                "|                                                     |",

                "+------------------------------------------------------+"

            ]

        }

        # Print the room's ASCII art

        print("\n".join(room\_art[self.name]))

        # Print room description and exits

        print("Description:", self.description)

        print("Exits:", ", ".join(self.exits.keys()))

Description

The function provides detailed information about the current room, including an ASCII art representation, a textual description, and the available exits. It enhances the player's understanding of their current location and surroundings within the game.

Purity

This function is non-pure as it interacts with the external environment by displaying output to the player. Non-pure functions generally have side effects, such as printing to the console, which is the case here.

Totality

The function is totalized, meaning it will always produce output regardless of the state or input provided. It consistently provides a complete set of information about the current room, ensuring the player always receives the necessary details to proceed with the game.

**Class: Player**

Description: Represents the player in the game.

Attributes:

* inventory: List of items the player is carrying.

class Player:

 def \_\_init\_\_(self):

        self.inventory = []

**add\_item(item): Adds an item to the player's inventory.**

  def add\_item(self, item):

        self.inventory.append(item)

**Purity**

The function is impure because it modifies the state of the program. In this context, "modifies state" refers to the function's ability to change the internal state of the game, such as adding an item to the player's inventory, altering room conditions, or updating game variables. Impure functions interact with or alter the program’s state outside their local scope, which can lead to side effects.

**Totality**

The function is totalized, meaning it will always successfully perform its intended operation under all conditions. Specifically, this function is designed to always add an item to the player's inventory or the game environment. No matter the state of the game or the input provided, it ensures the item is added, thus maintaining the integrity and continuity of the game mechanics. This guarantees consistent behavior and predictable outcomes whenever the function is called.

**remove\_item(item): Removes an item from the player's inventory.**

  def remove\_item(self, item):

        if item in self.inventory:

            self.inventory.remove(item)

            return item

        else:

            return None

**Purity**

The function is impure because it modifies the state of the program. In this context, modifying state means the function changes the internal data or variables of the game. For instance, when the function attempts to remove an item from the player's inventory or from a game location, it alters the game's state. Impure functions like this one have side effects because their operations extend beyond returning a value; they also impact the game’s overall state.

**Totality**

The function is non-totalized because it does not guarantee to perform its intended operation for all possible inputs. Specifically, this function is designed to remove an item, but if the item is not present in the inventory or location from which it’s supposed to be removed, the function cannot complete its operation. Therefore, while the function aims to remove an item, it may fail to do so if the item isn’t found. This lack of guarantee for successful execution under all conditions makes the function non-totalized. This aspect introduces potential variability in behavior based on the presence or absence of the item, necessitating additional checks or handling to manage these scenarios effectively.

**Class: Game**

class Game:

    def \_\_init\_\_(self):

        self.player = Player()

        self.current\_room = None

        self.rooms = {}

Description: Orchestrates the game flow.

Attributes:

* player: Instance of the Player class.
* current\_room: Current room where the player is.
* rooms: Dictionary containing all the rooms in the game.

Functions:

**add\_room(room): Adds a room to the game.**

def add\_room(self, room):

self.rooms[room.name] = room

Purity: Impure (modifies state).

Totality: Totalized (always adds a room).

**start\_game(): Starts the game by selecting a random room and initiating gameplay.**

  def start\_game(self):

        self.current\_room = random.choice(list(self.rooms.values()))

        self.play()

Purity

The function is classified as impure because it modifies the state of the program. In the context of starting a game, this involves initializing game variables, setting up the game environment, and potentially altering the initial state of various game elements. For example, when the game starts, the function might initialize the player's starting position, populate the game world with items and characters, and set the initial conditions for gameplay. These actions modify the internal state of the game, which is why the function is considered impure. Impure functions have side effects because their operations extend beyond returning a value and impact the game’s overall state.

**Totality**

The function is totalized because it always performs its intended operation of starting the game regardless of the circumstances. This means that every time the function is called, it will consistently initialize and set up the game environment without any conditions that could prevent it from executing. It guarantees that the game will begin, setting up the necessary initial state for gameplay. This reliability and predictability are key characteristics of totalized functions, ensuring that they always complete their intended task. In this case, the function will always successfully start the game, providing a stable entry point for players.

**display\_current\_room(): Displays information about the current room.**

   def display\_current\_room(self):

        print("\nCurrent Room:", self.current\_room)

        self.current\_room.display\_info()

**Purity**

The function is classified as non-pure because it displays output to the user. In this context, displaying information about the current room, including an ASCII art representation, a description, and available exits, constitutes an interaction with the outside world. Pure functions are those that produce the same output given the same input and do not cause any observable side effects outside of returning a value. Since displaying information to the screen affects the user interface and relies on the state of the game at the time, this function is considered non-pure.

**Totality**

The function is totalized because it always performs its intended operation of displaying the current room’s information. Regardless of the game state or which room the player is in, the function will consistently generate and display the room's ASCII art, description, and exits. This ensures that every invocation of the function will provide the necessary output to the player, allowing them to understand their current location and available actions. The totalized nature of the function guarantees that it will always successfully show the current room’s details, providing a reliable user experience.

**move(direction): Moves the player to another room based on the given direction.**

 def move(self, direction):

        if direction in self.current\_room.exits:

            next\_room\_name = self.current\_room.exits[direction]

            if next\_room\_name in self.rooms:

                self.current\_room = self.rooms[next\_room\_name]

                self.display\_current\_room()  # Update display

            else:

                print("There is no room in that direction.")

        else:

            print("You cannot go in that direction.")

Purity: Impure (modifies state).

Totality: Totalized (always moves the player).

**take\_item(item\_name): Allows the player to take an item from the current room and add it to their inventory.**

  def take\_item(self, item\_name):

        if item\_name in self.current\_room.items:

            item = self.current\_room.items.remove(item\_name)

            self.player.add\_item(item)

            print("You picked up the", item\_name)

            self.display\_current\_room()  # Update display

        else:

            print("There is no", item\_name, "here.")

**Purity**

The function is impure because it modifies the state of the game. Specifically, when a player takes an item, the function updates the game’s inventory or the player's inventory to reflect this change. Impure functions interact with or alter the state of the program outside their scope, and in this case, changing the inventory constitutes such a state modification.

**Totality**

The function is totalized because it always allows the player to take an item if the item is present. This means that whenever the function is called, it will consistently execute the necessary steps to add the item to the player's inventory or the game’s inventory. The totalized aspect of the function ensures that it will handle the action of taking an item in a reliable and predictable manner, without failing to complete its operation. Even if the item is not present, the function will handle the scenario appropriately (e.g., by indicating the item cannot be taken), maintaining its totalized nature by always providing a response.

**use\_item(item\_name): Allows the player to use an item from their inventory or the current room.**

 def use\_item(self, item\_name):

        if item\_name in self.player.inventory:

            print("You used the", item\_name)

            self.current\_room.used\_items.append(item\_name)  # Add the used item to used\_items list

        elif item\_name in self.current\_room.items:

            print("You used the", item\_name)

            self.current\_room.used\_items.append(item\_name)  # Add the used item to used\_items list

        else:

            print("You don't have a", item\_name)

        self.display\_current\_room()  # Update display

Purity: Impure (modifies state).

Totality: Totalized (always allows the player to use an item).

**play(): Main loop of the game where the player can input commands.**

 def play(self):

        print("Welcome to the adventure game!")

        print("Type 'help' for instructions.")

        self.display\_current\_room()  # Initial display

        while True:

            command = input("What would you like to do? ").strip().lower().split()

            if command[0] == 'go' and len(command) > 1:

                self.move(command[1])

            elif command[0] == 'take' and len(command) > 1:

                self.take\_item(command[1])

            elif command[0] == 'use' and len(command) > 1:

                self.use\_item(command[1])

            elif command[0] == 'help':

                print("Available commands: go <direction>, take <item>, use <item>, help, quit")

            elif command[0] == 'quit':

                print("Good luck!")

                break  # Stop the game

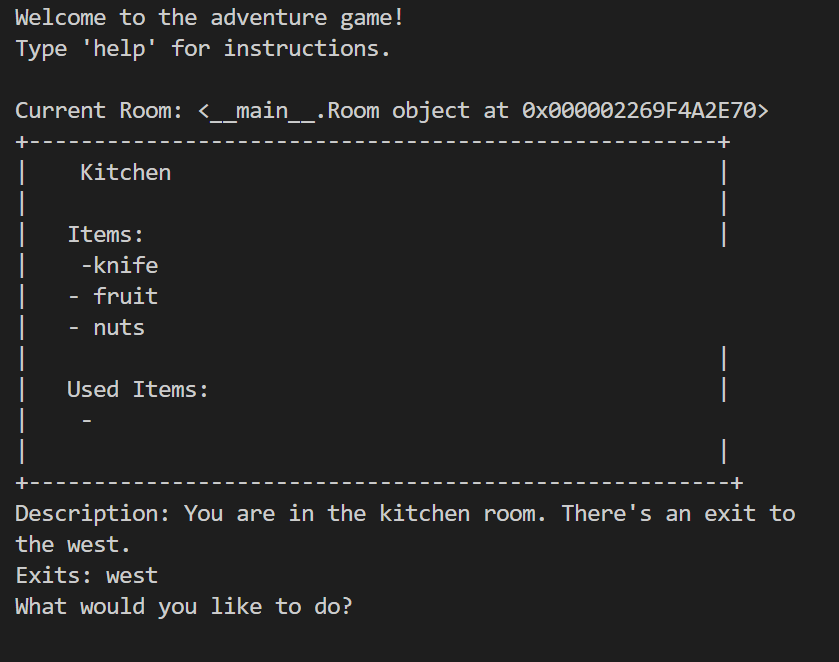
            else:

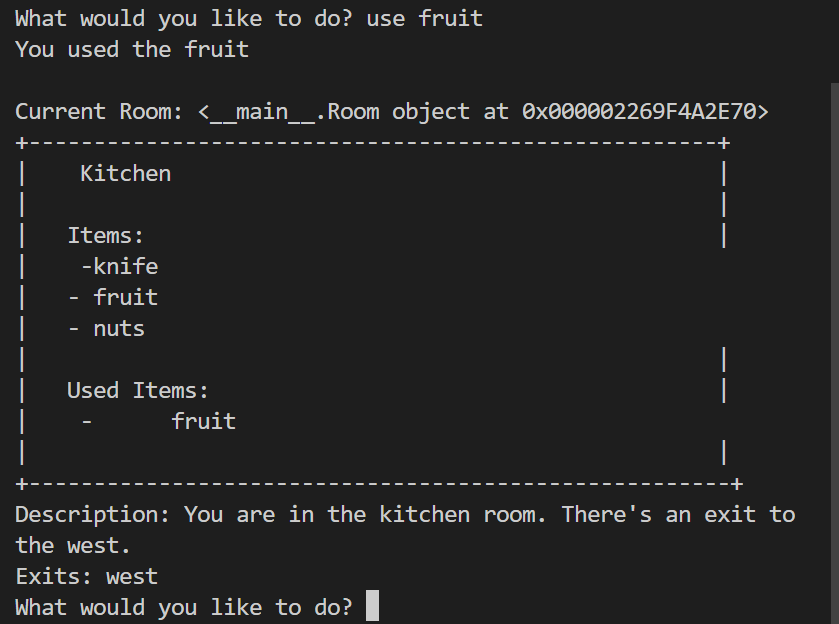
                print("Invalid command. Type 'help' for instructions.")

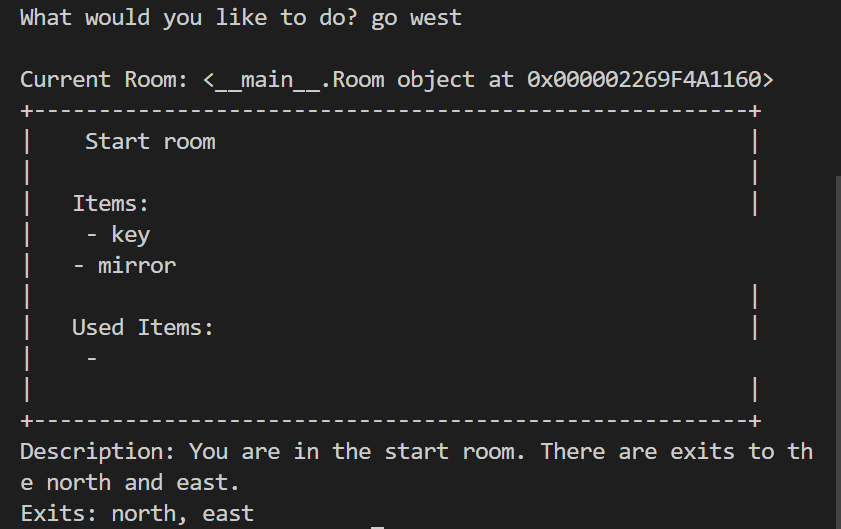
Purity: Impure (handles I/O).

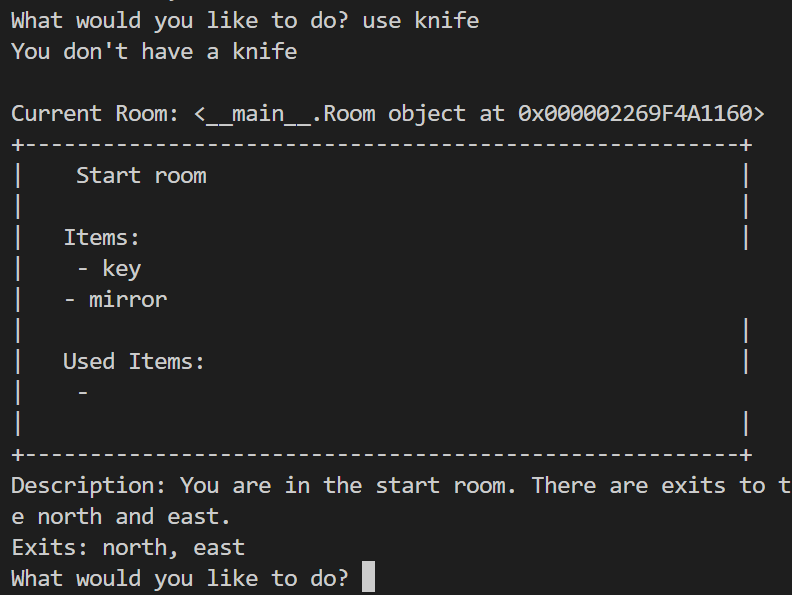
Totality: Totalized (always runs the game loop).

**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 to Test | Displaying room information |  |
| Steps to Do | Start the game | Display the current room |
| Expected Output | Start room information displayed |  |

| **Test Case ID** | **2** | **Passed/Failed** |
| --- | --- | --- |
| Software Feature to Test | Moving to another room |  |
| Steps to Do | Move to the hallway | Display the current room |
| Expected Output | Hallway room information displayed | g |

| **Test Case ID** | **3** | **Passed/Failed** |
| --- | --- | --- |
| Software Feature to Test | Taking an item |  |
| Steps to Do | Take the mirror from the start room | Display the current room |
| Expected Output | Mirror taken and Start room information displayed |  |

| **Test Case ID** | **4** | **Passed/Failed** |
| --- | --- | --- |
| Software Feature to Test | Using an item |  |
| Steps to Do | Use the mirror | Display the current room |
| Expected Output | Mirror used and Start room information displayed |  |

| **Test Case ID** | **5** | **Passed/Failed** |
| --- | --- | --- |
| Software Feature to Test | Attempting to move to a non-existent room |  |
| Steps to Do | Try to move north from the kitchen | Display error message |
| Expected Output | Error message displayed |  |

| **Test Case ID** | **6** | **Passed/Failed** |
| --- | --- | --- |
| Software Feature to Test | Attempting to take a non-existent item |  |
| Steps to Do | Try to take a chair from the start room | Display error message |
| Expected Output | Error message displayed |  |

| **Test Case ID** | **7** | **Passed/Failed** |
| --- | --- | --- |
| Software Feature to Test | Attempting to use a non-existent item |  |
| Steps to Do | Try to use a lamp | Display error message |
| Expected Output | Error message displayed |  |

| **Test Case ID** | **8** | **Passed/Failed** |
| --- | --- | --- |
| Software Feature to Test | Quitting the game |  |
| Steps to Do | Type 'quit' | Display farewell message |
| Expected Output | Farewell message displayed |  |

| **Test Case ID** | **9** | **Passed/Failed** |
| --- | --- | --- |
| Software Feature to Test | Typing an invalid command |  |
| Steps to Do | Type 'xyz' | Display error message |
| Expected Output | Error message displayed |  |

| **Test Case ID** | **10** | **Passed/Failed** |
| --- | --- | --- |
| Software Feature to Test | Typing 'help' for instructions |  |
| Steps to Do | Type 'help' | Display available commands |
| Expected Output | Available commands displayed |  |

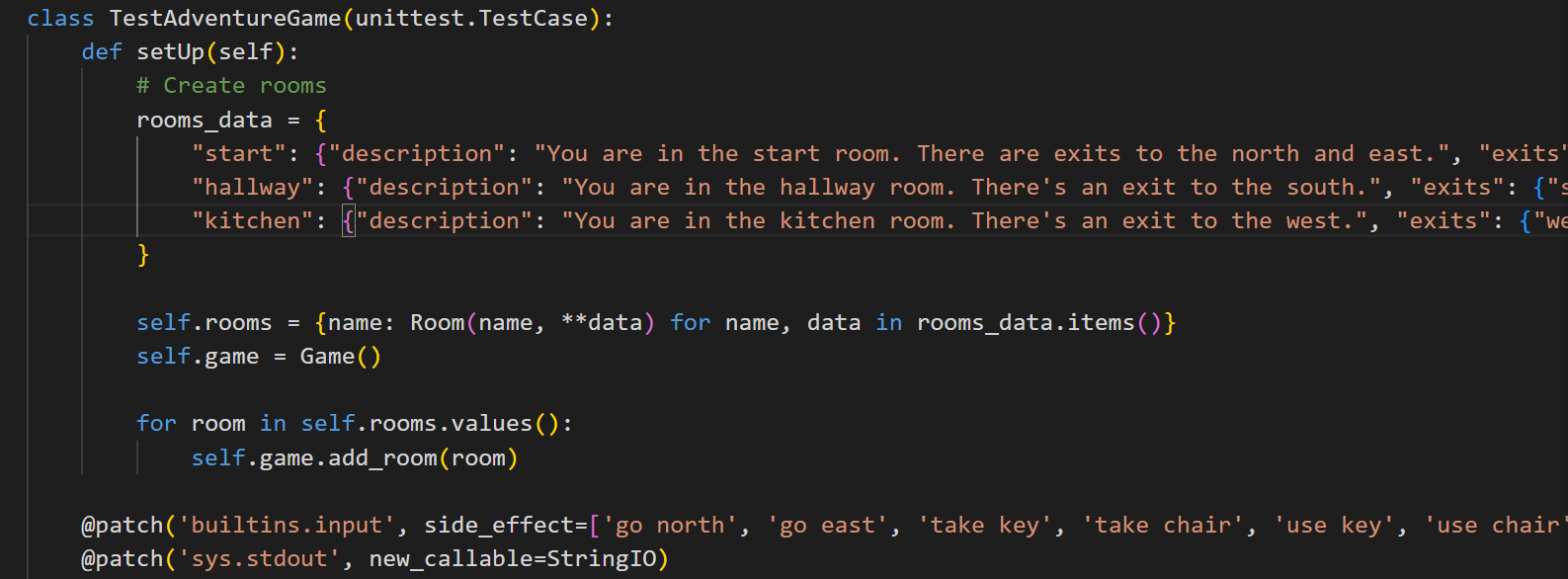
| **Test Case ID** | **11** | **Passed/Failed** |
| --- | --- | --- |
| Software Feature to Test | Taking an item from an empty room |  |
| Steps to Do | Take an item from the kitchen | Display item used message |
| Expected Output | Display item used |  |

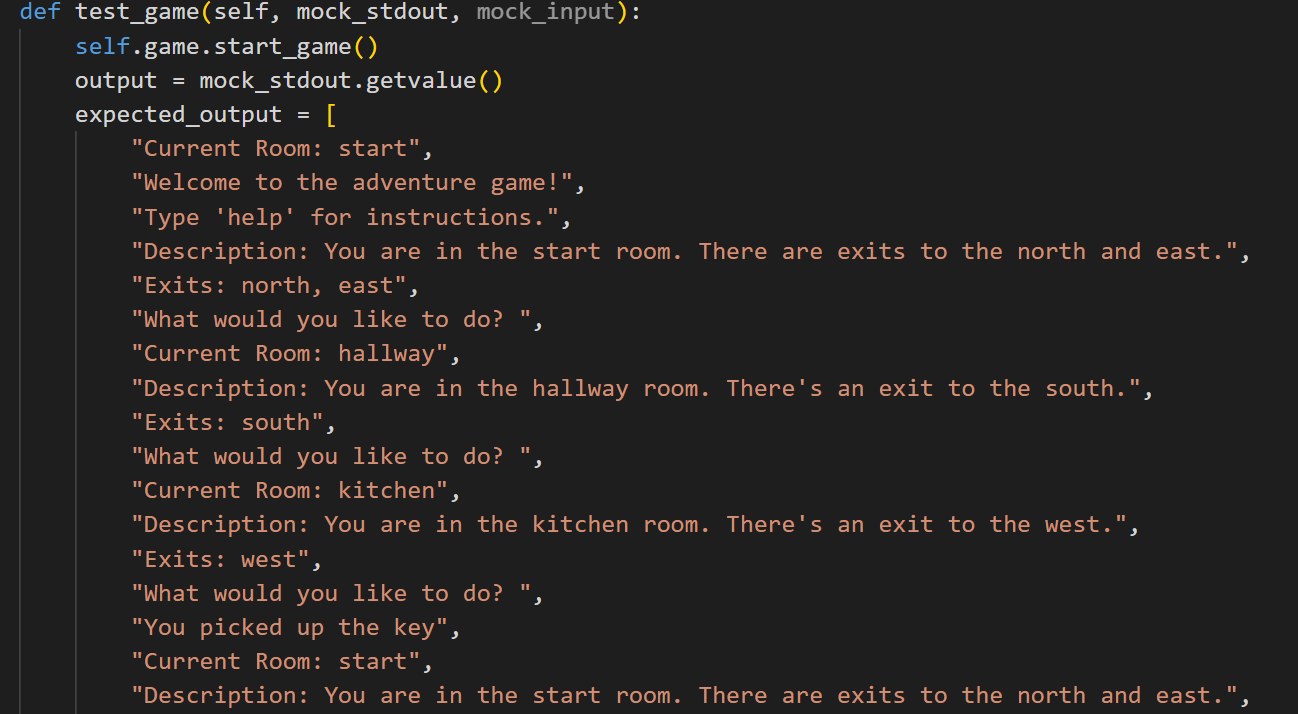
**Automated testing:**

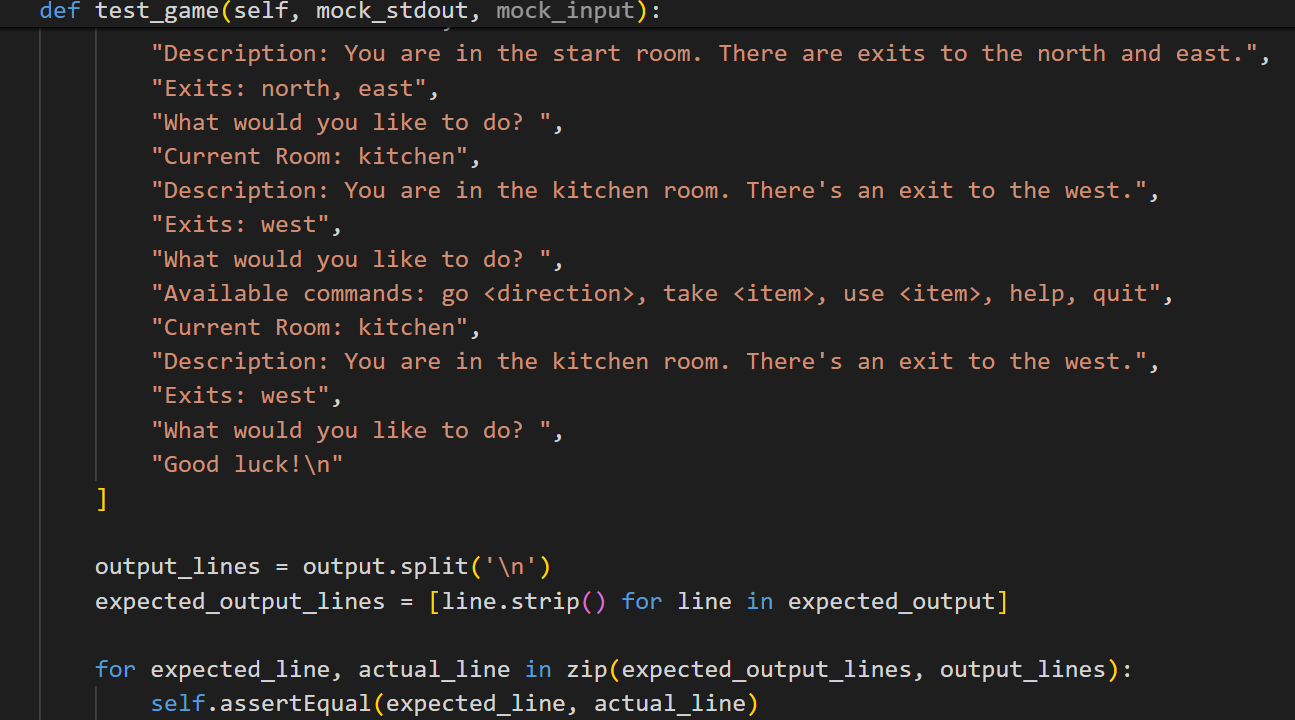
For automated testing of frakle “unittest” library is best and using it.

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.

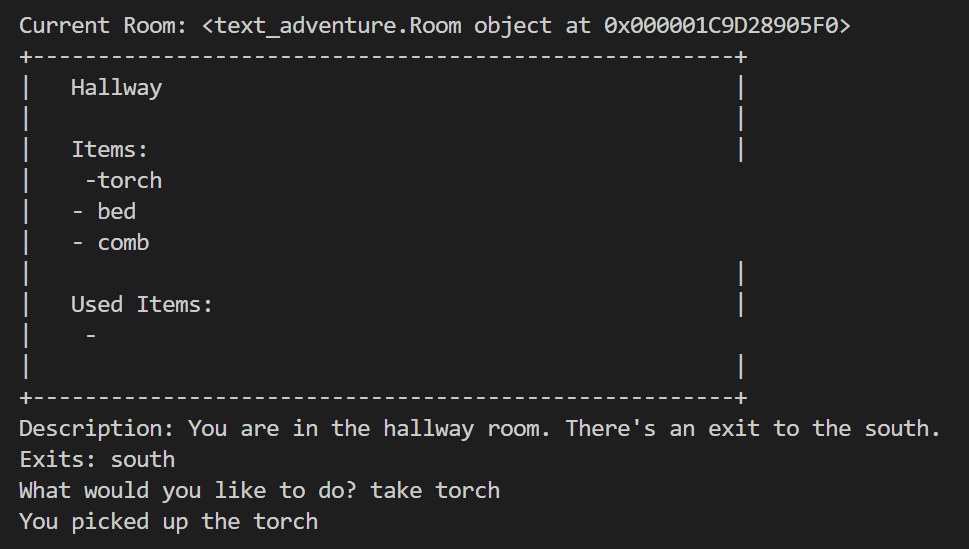
**Setup for testing**

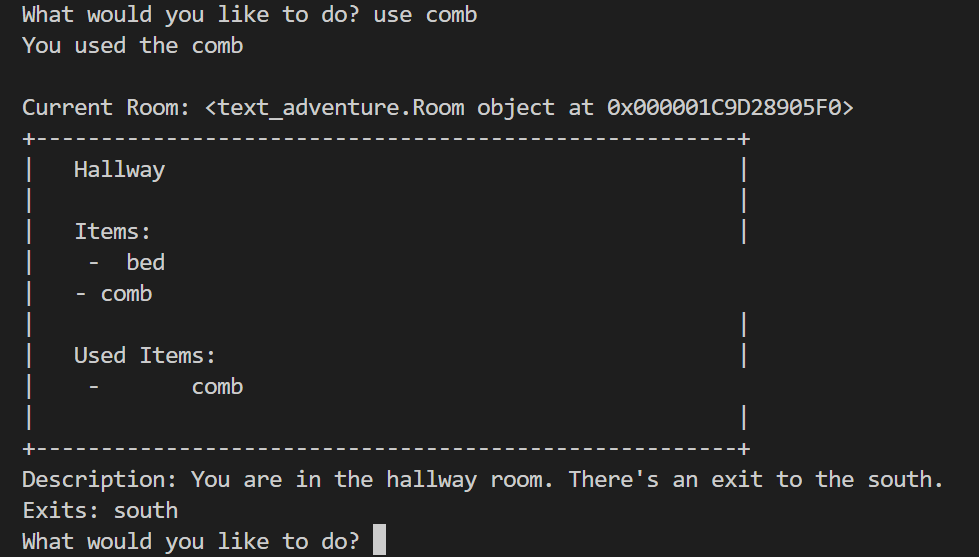


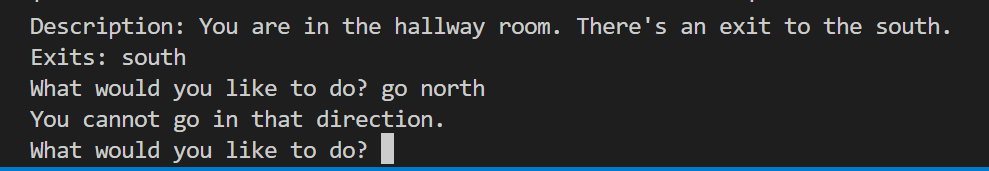
**Testing the game code** 



Running the code and output of tests:





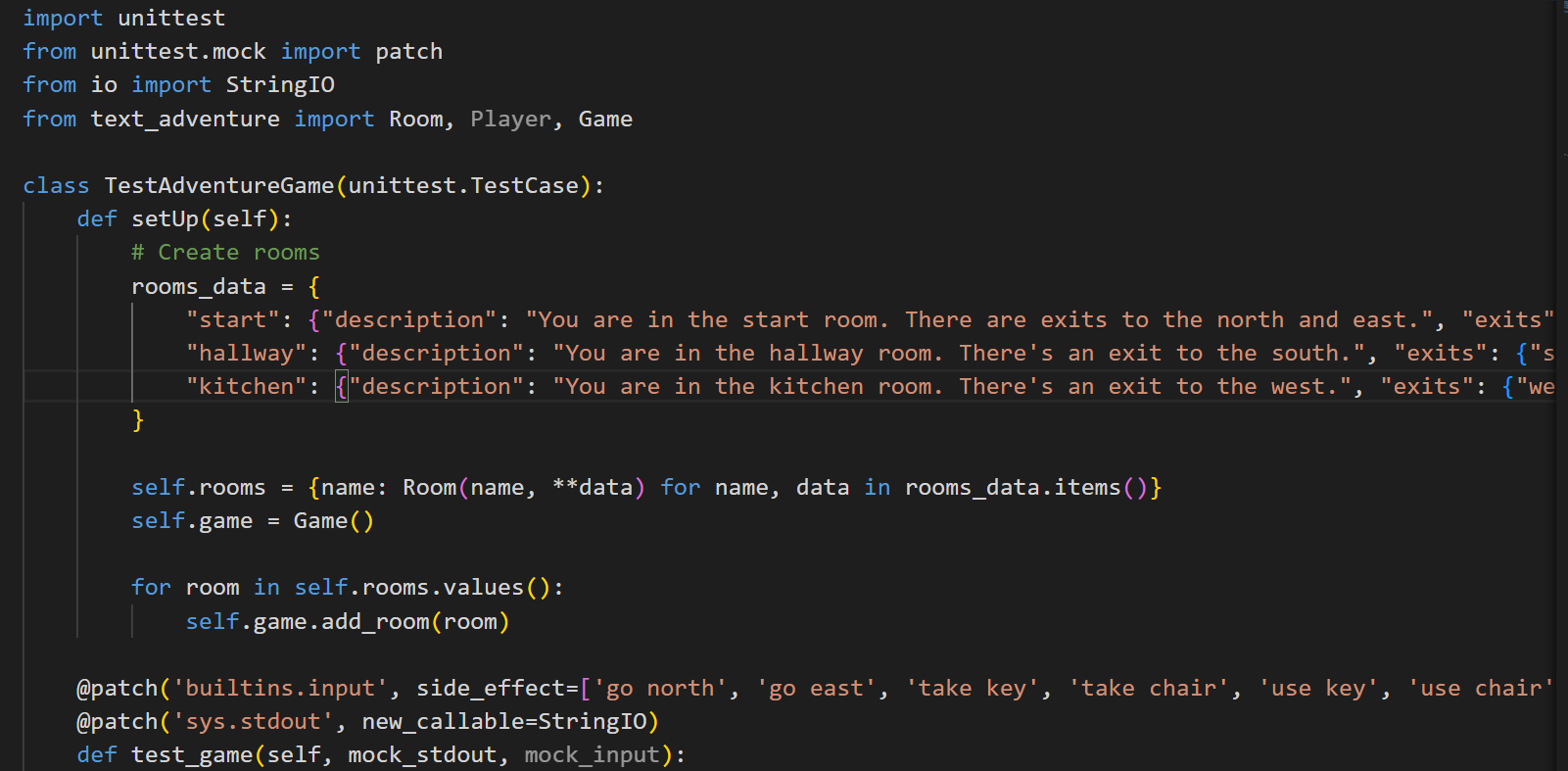


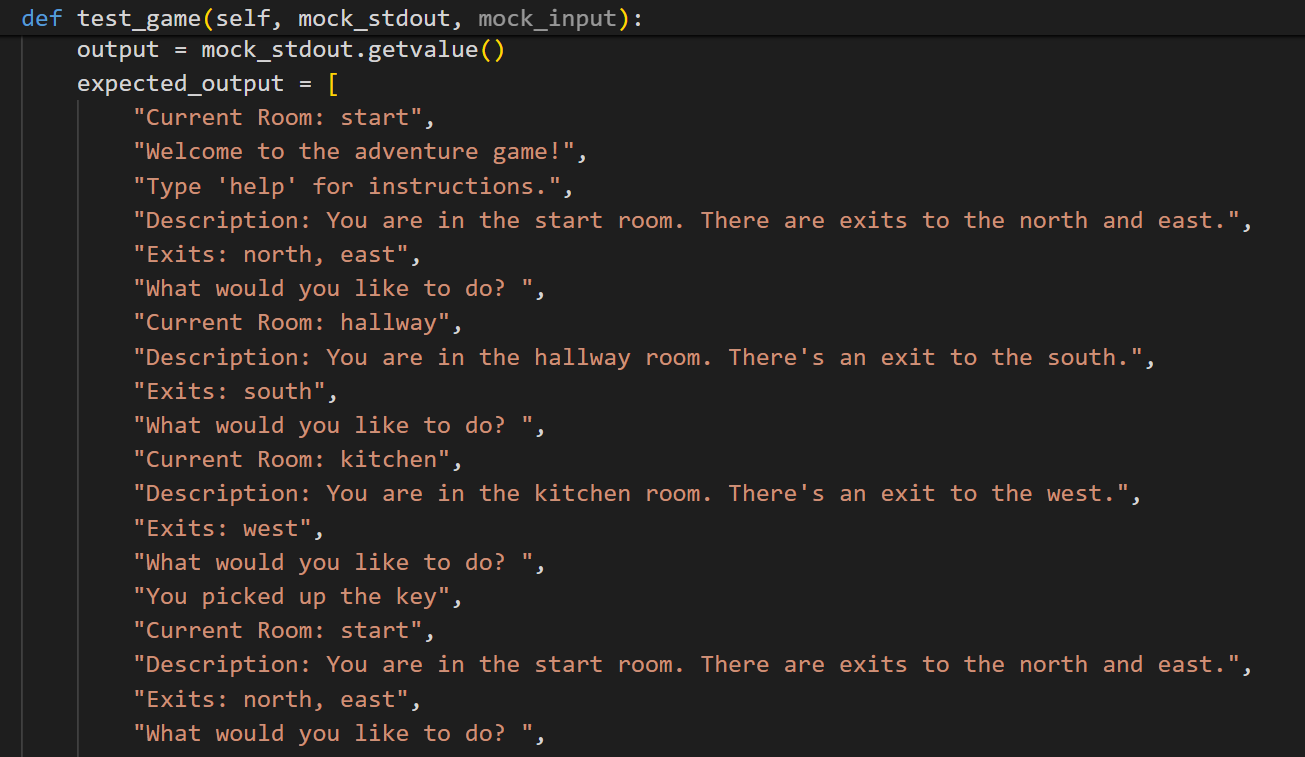


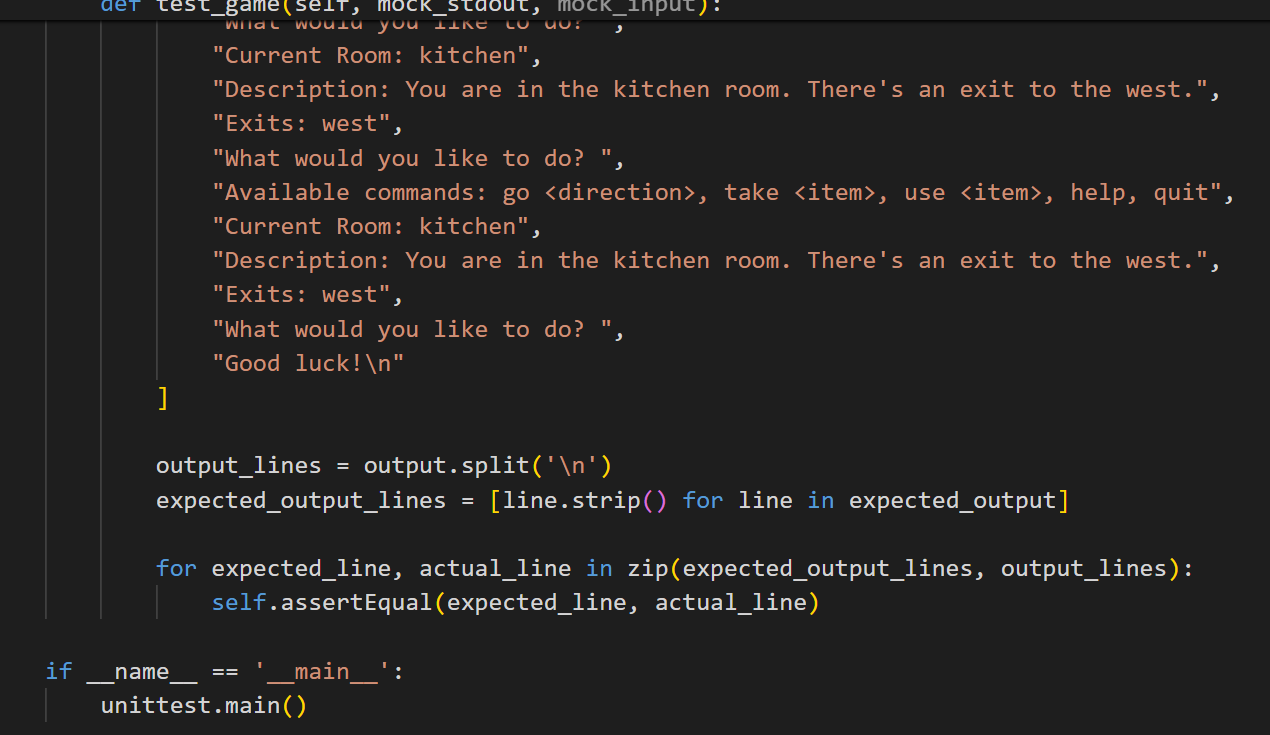
Game is stopped when user quit it.

**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.

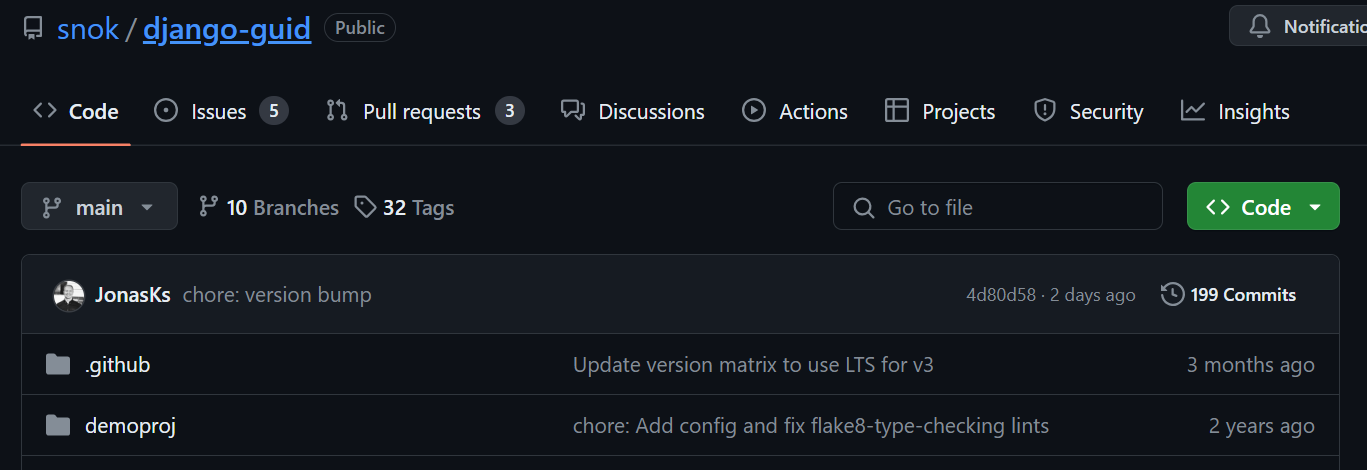
Pull requests are especially useful for team collaboration on the Farkle project. They allow developers to propose changes, have them reviewed by peers, and then merge them into the main repository. This process ensures that code changes are thoroughly examined before being integrated, maintaining code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

**Case study**

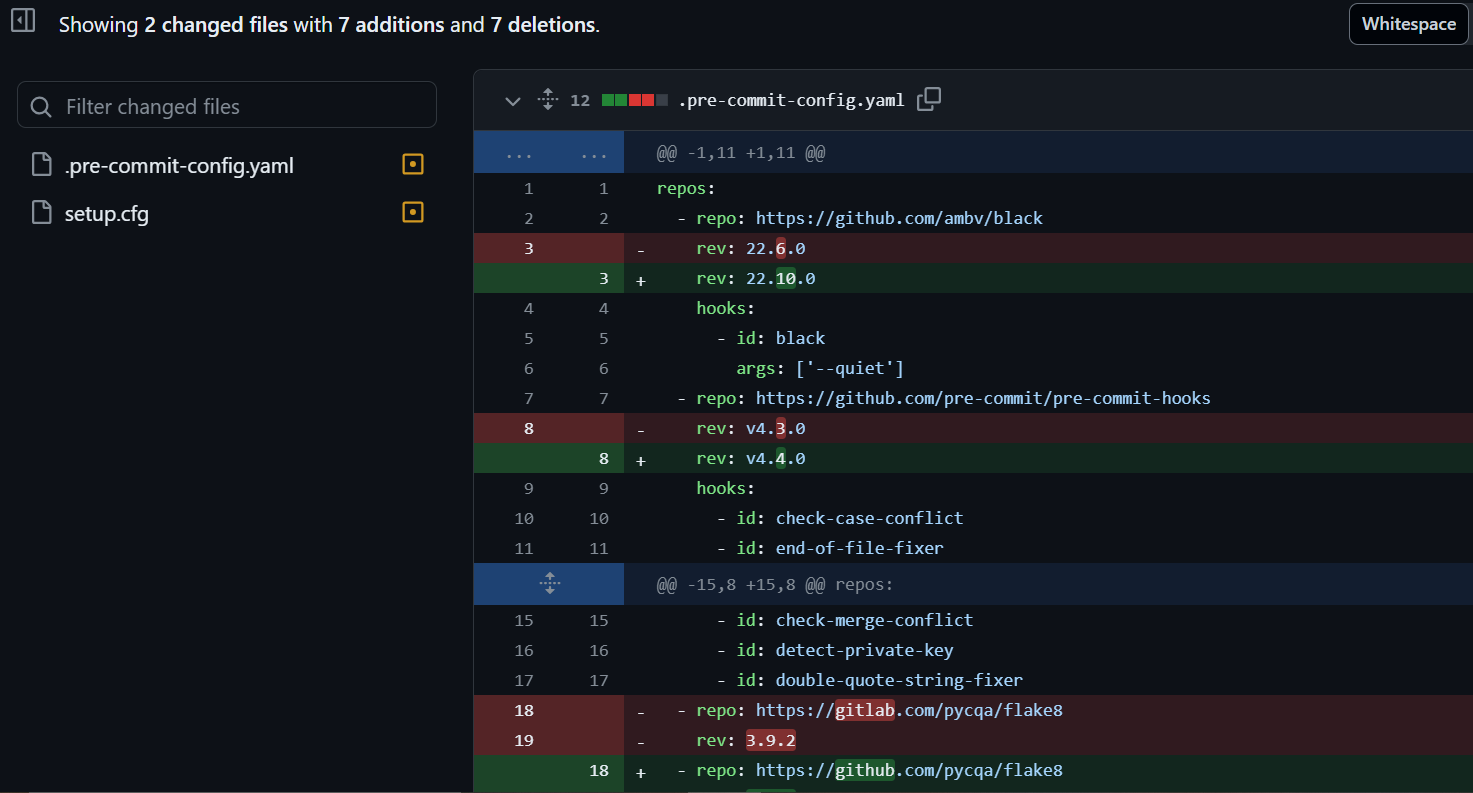
For a case study of git, I study django repsoitory.

<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 presented in a chronological sequence, with the most recent commit showcased at the top of the list. To access earlier commits, one can simply scroll down the page or utilize the "older" navigation option provided. Noteworthy changes to the project's code are visually highlighted using red and green markers, indicating lines of code that were removed (in red) or added (in green) during each commit. This color-coded representation aids in quickly identifying alterations made to the codebase over time, facilitating a clear understanding of the project's development history and evolution.



**Refrences:**

The benefits and need of documentation of code:

<https://iwconnect.com/the-benefits-of-code-documentation-why-its-a-must-have-for-your-development-team/> Accessed [28/04/2024]

Text adventure game guide:

<https://en.wikipedia.org/wiki/Text-based_game> Accessed [28/04/2024]

Git guid sample:

<https://github.com/git-guides/git-clone> Accessed [28/04/2024]