CSC111 Winter 2025 Project 1

Skye Mah-Madjar, Krisztian Drimba

June 8, 2025

Game Map

Example game map below (edit it to show your actual game map):

```
1  2  3
4  5  6
7  8  9
```

Starting location is: 1

Game solution

List of commands:
["go south", "pokemon battle", "thunder punch", "go south", "pickup usb", "go east", "pickup laptop charger", "go north", "play tenjack", "cheatcode", "go north", "go east", "investigate podiums", "solve artifact 1 riddle", "loop", "solve artifact 2 riddle", "list", "solve artifact 3 riddle", "function", "solve artifact 4 riddle", "python", "solve artifact 5 riddle", "recursion", "solve artifact 6 riddle", "tree", "solve main artifact riddle", "1234", "go west", "go west", "charge laptop", "go south", "return stone", "inventory", "combine", "yes"]

Lose condition(s)

For functionality, nearly everything in adventure.py, game_entities.py, and proj1_event_logger.py is used. However, for losing, there are some that are used more specifically. In this case, classes like EventList and Event class are obviously used to log events, but not used directly, as undoing is not necessary here.

Method 1
Description of how to lose the game:
- If the player stalls for long enough, they will eventually run out of turns and lose the game.
List of commands:
["go east", "go west", "go south", "pokemon battle", "flamethrower", "pokemon battle", "close combat", "pokemon battle", "thunder punch", "go east", "go east", "go north", "go west", "go west", "go south", "go south", "go north", "go south", "pickup usb", "go east", "go west", "go east", "go north", "go south", "pickup laptop charger", "go east", "go north", "go north", "go west", "go west", "go south", "go north", "charge laptop", "go south", "go south", "go east", "go north", "go north", "go west", "go east", "go south", "go east", "go north", "go south", "go west", "go north", "go west", "go south", "go east", "go east"]

Which parts of your code are involved in this functionality:
- AdventureGame class (and most of its methods)
- Main game loop in adventure.py (at the end of the loop is where it checks if the player ran out of turns)

- Puzzle class (and most of its methods)
- Location.conditions
- Player class

Method 2
Description of how to lose the game:
- If the player successfully reaches the end of the game after completing everything, they can refuse to submit the assignment, thus, losing the game.

List of commands:
["go south", "pokemon battle", "thunder punch", "go south", "pickup usb", "go east", "pickup laptop charger", "go north", "play tenjack", "cheatcode", "go north", "go east", "investigate podiums", "solve artifact 1 riddle", "loop", "solve artifact 2 riddle", "list", "solve artifact 3 riddle", "function", "solve artifact 4 riddle", "python", "solve artifact 5 riddle", "recursion", "solve artifact 6 riddle", "tree", "solve main artifact riddle", "1234", "go west", "go west", "charge laptop", "go south", "return stone", "inventory", "combine", "no"]

Which parts of your code are involved in this functionality:
- AdventureGame class (and most of its methods)
- Main game loop in adventure.py
- Puzzle class (and all its methods)
- Location.conditions
- Player class

Locations

For our Location class in game_entities.py, we created a method called conditions. Its purpose is to constantly add and remove the available commands of each location depending on the status of the player. For example, if a player has already beat a puzzle, they should not be able to play it again. Similarly, if they picked up an item, they should not be able to do it again. To do this most efficiently, we found a method of using lambda functions which allowed us to organize these deletions and additions most efficiently.

Inventory

We created an inventory system accessible in the menu that allows the user to interact with the items. More information in the "other commands" section of the report.

1. All location IDs that involve items in the game:
   (a) 1
   (b) 3
   (c) 4
   (d) 5
   (e) 7
   (f) 8
2. Item data:
   None of the items have a specific target location in which an automatic score is given at arrival, but some of them require an action to be done at a specific location. For these items, "Item target

location ID:" represents the location in which the action must be taken with the item present to achieve score.

(a) For Item 1:
   - Item name: "USB"
   - Item start location ID: 7
   - Item target location ID: N/A (the user just needs to have the item in their inventory)

(b) For Item 2:
   - Item name: "Laptop Charger"
   - Item start location ID: 8
   - Item target location ID: 1

(c) For Item 3:
   - Item name: "Lucky Mug"
   - Item start location ID: 5
   - Item target location ID: (user needs to combine this with G-Fuel as an action)

(d) For Item 4:
   - Item name: "G-Fuel"
   - Item start location ID: 4
   - Item target location ID: (user needs to combine this with the Mug as an action)

(e) For Item 5:
   - Item name: "Ancient Computer Scientist Stone"
   - Item start location ID: 3
   - Item target location ID: 4

3. Exact command(s) that should be used to pick up an item (choose any one item for this example), and the command(s) used to use/drop the item (can copy the list you assigned to `inventory_demo` in the `project1_simulation.py` file)

   ["go south", "pokemon battle", "thunder punch", "go south", "pickup usb", "go east", "pickup laptop charger", "inventory"]
   (The other 2 items are also gained from puzzles)

4. Which parts of your code (file, class, function/method) are involved in handling the `inventory` command:
   - adventure.py (main game loop's branch for the 'inventory' command)
   - game_entities.py (Player class, specifically the 'items' attribute and __init__ method)

Score

1. Briefly describe the way players can earn scores in your game. Include the first location in which they can increase their score, and the exact list of command(s) leading up to the score increase: Players can earn score by completing the task necessary for each item. There are 5 items in total, and to win the game they must have each item receive its necessary win condition. For example, if players "go south", "go south", "pickup usb", "score", it will show they have 1/5 score, since for the USB item, it is only necessary to pick it up.

2. Copy the list you assigned to `scores_demo` in the `project1_simulation.py` file into this section of the report:

   ["go south", "go south", "pickup usb", "go east", "pickup laptop charger", "go north", "go north", "go west", "charge laptop", "score"]

3. Which parts of your code (file, class, function/method) are involved in handling the `score` functionality:
   adventure.py (main game loop's branch for the "score" command and the AdventureGame.win method) and game_entities.py (Item class, whose status attribute is used in the score calculation)

Other Commands

1. Go [direction] was quite simple to implement, same with look, and log was quite easy too since we already had the exercise 1 code to go off of. However, undo took quite a bit of work. We went with a design where we constantly saved each game state the player made as they played so that they could undo back as much as they wanted.

2. We also decided against the command for users to drop items, we felt it was not necessary for our game and that it would be better off if the users did not have to ever drop items, instead, just have them on them or interact with them in certain ways in order to win

3. In the inventory command, player's have a few options:
   - The "combine" command for the inventory, which is not useful for the players until they collect the G-Fuel and the Lucky Mug, which they need to combine to win. Until they have both, they can try to combine, and it will give them little hints for what they need.
   - The "description" command prompts the users to input an item name, which they can then receive a description of (possibly receiving clues).

Enhancements

We found it easier to treat each puzzle as a single event, thus having only the action of taking the puzzle logged as an action, while each puzzle runs individually, separate from the game. For that same reason, users cannot see the default menu while they are in a puzzle. This worked best for our code, since none of our puzzles involved traversing the map, rather, they were all in singular places that involved the user completing the puzzle at that location before moving on. Thus, in our simulation, all non-location commands are not checked in the walkthroughs/demos.

1. ROM Podiums
   - This puzzle allows users to investigate podiums that have clues on them, where they then have to answer Computer Science riddles for each podium to finally unlock the main podium. If they win, they are rewarded with an item.
   - High Complexity Level
   - This took the longest to create out of all of the puzzles due to attempts at making it efficient. If we had to create if statements for each inspection, for each riddle answering, etc. it would be a ginormous amount of code, but we were able to simplify it massively. To do so, we have lots of code that goes into making it more efficient through string splicing, etc. making it more complex than the rest in terms of code. It does also require us to check their answers and track their progress, adding a bit of complexity. - By the way, the answers to the 7 riddles are: ['loop', 'list', 'function', 'python', 'recursion', 'tree', '1234']
   - Functionality:
     - Files: game_entities.py, adventure.py
     - Class: Puzzle
     - Methods:

    * rom_podiums(self, player: Player) → None
    * handle_artifacts(self, choice: str, player: Player, win_count: int, main_artifact_commands: dict) → int
    * handle_main_artifact(self, choice: str) → None
- Demo: ["go east", "go east", "investigate podiums", "inspect artifact 1", "solve artifact 1 riddle", "loop", "inspect artifact 2", "solve artifact 2 riddle", "list", "inspect artifact 3", "solve artifact 3 riddle", "function", "inspect artifact 4", "solve artifact 4 riddle", "python", "inspect artifact 5", "solve artifact 5 riddle", "recursion", "inspect artifact 6", "solve artifact 6 riddle", "tree", "inspect main artifact", "solve main artifact riddle", "1234", "inventory"]

2. Pokemon Battle
    - A puzzle where a player is given options of commands to take in a Pokemon-style battle, and if they choose the correct option, they are rewarded with an item.
    - Low Complexity Level
    - It is a very simple puzzle where the user gets few options, so we did not have to do much to implement other than check if they put in the correct option.
    - Functionality:
        - Files: game_entities.py, adventure.py
        - Class: Puzzle
        - Method: pokemon_battle(self, player: Player) → None
    - Demo: ["go south", "pokemon battle", "bag", "run", "close combat", "pokemon battle", "thunder punch", "inventory"]

3. Tenjack
    - A homebrew card game similar to Blackjack in which the user is dealt cards from 1 to 10 and needs to get as close to 21 without going over, and they play against the dealer. They might need to repeat multiple times, since the dealer may beat them, and the dealer is hard coded to get a random score from 15-21, something we imported a library for. If they win, they are rewarded with an item.
    - Medium Complexity Level
    - We had to create game logic for the user to be able to be dealt cards, and for the game to be able to keep track of his total, etc. It requires multiple steps to solve and needs to compare the user's decisions to this hard coded dealer, thus making it more complex. There is also a feature built-in secretly, where after entering the puzzle the user can enter 'cheatcode' which will automatically make them win. This is because it can be hard to determine the odds of winning, thus, making it unknown how many times the user might need to play this puzzle to win it. For marking purposes, this is a bypass.
    - Functionality:
        - Files: game_entities.py, adventure.py
        - Class: Puzzle
        - Method: tenjack(self, player: Player) → None
    - Demo: ["go south", "go east", "play tenjack", "cheatcode", "inventory"]

4. Ddakji
    - This puzzle is a combination of the Korean kid's game Ddakji, recently popularized by Squid Game. Ddakji is a game where players throw paper envelopes at envelopes on the floor to try and get them to flip.

- Medium Complexity Level
- This puzzle is similar to the ROM Podiums puzzle in that it involves taking certain commands to get clues about what the user should adjust in their throwing settings before they finally throw their envelope, which means they need to take multiple steps. Thus, we need to constantly adjust their settings and eventually check if they got the right one, making it a bit more complex. This puzzle also unlocks the ability for them to be able to ride the TTC, i.e. travel between the locations with id 2 and 9, adding some intricacy as it was not super simple to have them unlock abilities like that and have them not be able to do it beforehand.
- Functionality:
  - Files: game_entities.py, adventure.py
  - Class: Puzzle
  - Methods:
    - ddakji(self, player: Player) → None
    - handle_ddakji_choice(self, choice: str, player: Player, state: dict) → None
    - update_ddakji_state(self, choice: str, state: dict) → None
    - evaluate_throw(self, player: Player, state: dict) → None
- Demo: ["go east", "play ddakji", "observe tv", "read poster", "observe athlete", "eavesdrop", "observe book", "current form", "set high power", "set side down", "throw"]

Other Classes

These are some of the Other Classes we created that are not necessarily Puzzles or "Enhancements", but took a portion of our work up due their uses.

1. Puzzle Class

   - The Puzzle class was created to handle all the puzzle-related data and logic in the game. It plays a central role in managing the puzzle's state and loading the puzzle data from a JSON file.
   - This class was a significant portion of the game_entities.py file and required lots of work. Using the JSON file, introducing randomness in the puzzles, implementing puzzle-solving logic, and managing the addition and removal of various commands was very strenuous and made most puzzles quite complex to create, with the exception of the Pokemon battle, which was very easy to create.

2. Player Class

   - The Player class was designed to track the player's status throughout the game, including their inventory, and other attributes that change as the game progresses.
   - The Player Class itself does not have much internal logic, given that it has no methods itself. Rather, its important as it allowed us to create a system of storing attributes of what the game has vs. what the player has. For example, the game stores its items in a list, but the player should have their own list of the items they currently have. This allowed us to properly create the player's inventory system, for example.

Chat GPT

There were a few things we needed to explicitly use Chat GPT for, such as the doctest in load_game_state since we could not figure out a good doctest, as well as knowing how to encode the JSON file we were loading in _load_game_data in UTF-8, since for some reason we were getting errors with that. ChatGPT was not at all used for any creative decisions within the game, and majority of the code was written without it other than few a instances where we had issues we needed assistance with.