A Clueful(less) Adventure

Submission to website: Monday, April 25, 10pm

Checkoff by LA/TA: Tuesday, April 26, 10pm

This lab assumes you have Python 2.7 installed on your machine. Please use the Chrome web browser.

Welcome To Boddy Estate

Welcome to Boddy Estate! We wish you were arriving on happier times – but we need your help. We were having quite the social gathering and next thing we knew… there's been a murder in the mansion! All we know is that Mr. Albatross was killed last night – but we need your help to determine who the murderer is, what the murder weapon was, and where Mr. Albatross was killed. We're trapped in the mansion until we figure it out, and if we fail to accuse the murderer correctly, Mr. Albatross might not be the last victim of the evening!

Brief Me

For those of you who have never played Clue before, let's go over how the game works, and how this lab is both similar and different than the classic board game. Clue is a multiplayer game. In order to win the game, a player must deduce who the suspect was that killed Mr. Albatross, where the crime took place, and with what weapon. These details are sealed in a secret envelope at the beginning of the game.

The details of the game are represented as three categories of `cards` : Suspects, Weapons and Rooms. Every player begins with a set of clues, which is a subset of cards from the aforementioned three categories. What does this mean? It means that if a player is holding a specific card, then it cannot be found in the secret envelope, and therefore cannot be a detail of the crime! So the players go in a round robin, travelling to rooms and making guesses to gather information until an accusation is made and the game is over.

On any given turn, a player can:

- `TRAVEL` to another room in the Mansion. All rooms are connected. It's a magical mansion, indeed. A player can only travel for themselves. This means that a travel request is always referring to the player whose turn it is currently. **In order to make a guess involving a certain room, a player must be present in that room**. More on this below. Travelling is also important when playing the text-based-adventure side of this lab via the online UI.

There are certain key words that you can discover that will depend on the player's current location.

- Make a `GUESS` about the crime details to the player on their right. We call this player the "adversary." A guess takes the form of "guess suspect in the room with the weapon." Like the real game of clue, a player must be present in the room he's making a guess about. For example, if Mario is in the ballroom, he can't guess "scarlett in the hall with the wrench" but he can guess "scarlett in the ballroom with the wrench". In answering the guess, the adversary must follow these rules:
    - If the player guessing is absentminded and asks about a room that they're not in, the adversary does not have to (and won't) answer.
    - If the adversary has exactly one of the cards named, he must show that card.
    - If a player already has shown a specific card, he will always default to showing that card again (why?).
    - If a player has multiple of the requested cards (or has shown multiple of the requested cards), he will default to showing a suspect card before a weapon card, and a weapon card before a room card.
    - If the adversary has a card to show, the adversary only shows the current player that is guessing (no other players see the card)
- Lastly, a player can make an `ACCUSATION` about the crime details. If their accusation is correct (meaning it matches the details in the secret envelope) then they win the game. If they aren't, then they lose. After a player makes an accusation, the game is over (This also hails from the original board game in which the envelope had to be open to check if they accusing player was correct).

Give me a Clue

Here's a brief snippet of example game play (as you'll see in the UI) to give you an idea of what is expected to happen. Note that players always begin in the foyer. In this mini example, not all of the cards are divided. Let's say that Ron has the following cards: suspects: scarlett, plum; weapons: rope, wrench; rooms: hall, foyer, and Hermione has the following cards: suspects: green, mustard, weapons: candlesticks, revolver; rooms: conservatory, library; The secret envelope is: white, foyer, leadpipe, meaning Mrs. White killed Mr. Albatross in the foyer with the leadpipe.

```
Ron and Hermione begin in the foyer.

(Ron's turn):
            input: travel to the ballroom
            result (in UI): "Ron travelled to the ballroom"

Ron is in the ballroom, Hermione is in the foyer.

(Hermione's turn):
            input: guess white in the foyer with the rope.
            result: "the card revealed was: rope"

Hermione has seen rope, rope has been revealed.

(Ron's turn):
            input: travel to the conservatory
            result: "Ron travelled to the conservatory"

Ron is in the conservatory, Hermione is in the foyer.

(Hermione's Turn):
            input: guess scarlett in the ballroom with the rope.
            result: "The guess involved an invalid room. Turn forfeited!"

(Ron's turn):
            input: accuse white in the library with the leadpipe.
            result: "Hermione wins"

Game Over, Ron accused falsely. Hermione wins.
```

Talk to Me

Much of the fun of this lab comes from exploring the game's universe. Whether that's by travelling to new rooms, and choosing to *look* around, or by discovering commands that allow you to see your *cards* or the *sheet* of your deduced information. Don't be suprised if this sheet contains the name Jarvis - that's the generic name that comes on the sheet for one's opponents. He's like the friendly neighborhood AI. The universe is willing to *show* you a lot, but you might have to specify what you want to see!

If these hints weren't clear enough, here's how you can talk to the UI. 1. "look" will give you a glimpse into your surroundings (the player's current room) 2. Any legal move will execute that move (**using your code**). For example, if you type "travel to the ballroom," whoever's turn it is will travel to the ballroom. On their next turn, if you type "look" you'll be provided with a description of the ballroom. 3. "show card" will provide a description of whichever card you type - "show scarlett" or "show rope" are valid examples. 4. "sheet" will provide you with a look at the

progress the current player has made. It will show you the cards the player has and the cards that have been revealed to them X'ed off accordingly (give it a try!). The format mimics the original boardgame. 5. "cards" will give you a list of the current player's cards.

`lab.py` : Model the Mansion

You will implement your code in this file. You are not expected to read or write any other code provided as part of this lab. You must correctly implement the functions described in this section and the next.

In order to help determine the details of the crime, we need to first model Boddy Estate. To do so, we'll create a World class. It's up to you to decide how you want to initialize a World object, but make sure that you implement all of the required methods. As usual, you might find it helpful to create some helper methods in the World class. It's up to you to decide how you want to internally represent the players, but we recommend making a Player class as well. Think about what makes a player unique. What attributes does a player have? Which data structures can you use within the player object to store the cards that they have or the information that they know? Just make sure that you're following the spec on all of the required methods described in this section and the next. **Note that all players begin in the "foyer" of the mansion**

The first step is to decide how you want to representate the World and to initialize this representation in `lab.py` by defining the World's `__init__(state)` method. It will be passed `state` which is a dictionary with the following keys:

- `envelope` : This is the game's secret envelope, represented as a list: `[suspect, weapon, room]` (order not enforced)
- `players` : A list of players who are playing the game. The order of players in the list represents the order in which they take turns. Keep in mind that after the last player of the list goes, it starts over from the beginning of the list.

- `playersCards` : A dictionary. Its keys are the names of the players. Each player points to another dictionary, and its keys are `"weapons"` , `"rooms"` and `"suspects"` , each of which points to a list of cards that that player was dealt in the beginning of the game. For example:

```
{"Burt" : {
  "weapons": ["revolver", "candlesticks"],
  "rooms": ["library", "hall"],
  "suspects": ["green", "white"] },
"Ernie": {
  "weapons": ["leadpipe", "wrench", "knife"],
  "rooms": ["study", "ballroom", "conservatory"],
  "suspects": ["plum", "mustard", "peacock"] }}
```

It's up to you to decide how you want to transform this to represent players! Once again, we suggest that you make a Player class.

If it weren't for you meddling kids…

Now for the game play! It's your job to implement the following methods:

- `take_turn(self, action, args)` : This is where all of the action happens. This method is responsible for executing a player's turn, ending the player's turn, and returning the appropriate result for the action (as described below). An action is one of `"travel"` , `"guess"` , `"accuse"` , and the arguments are action-dependent:
  - Travel: the action is `"travel"` , and `args` is a destination (meaning a room. For instance, `"ballroom"`). You should return the string "travel" to indicate that a player successfully compeleted their turn.
  - Guess: the action is `"guess"` , and `args` is a dictionary. The keys are `"suspect"` , `"room"` and `"weapon"` and point to the contents of the guess. A player always asks their adversary their guess (remember, this is the player sitting to their right). In this game, a player's adversary is the next player in the player array. If the current player is the last in the array, then the next player loops around to be the first in the array. If the player makes a guess involving a room that they're not in, you should return "invalid room" and the player's turn is over (forfeited). If the guess is valid, and the next player has one of the cards that they are being asked to show, they *must* show it. If a player has already shown one of the cards in the guess, they always prioritize showing that card. If they have shown more than one of the cards, or have more than one of the cards (room, suspect, weapon), then a suspect card is always shown before a weapon card, which is always shown before a room card. (This hails from the original board game in which room cards are the most valuable and difficult to deduce, since they require physically travelling to different rooms by rolling the dice). Return the string name of the card that was shown, for instance `"scarlett"` . If they have none of the requested cards, then return the string `"none"` .
- Accuse: the action is, you guessed it, `"accuse"` and `args` is a dictionary like that of

Guess. Your job is to compare it with the contents of the secret envelope. If the contents match up, then return the current player as the winner. If not, then by convention return the player next in line (the adversary) to signify that they lost. The game is over after an accusation is made. You'll see this in the UI too - after an accusation is made, the game is restarted, new cards are dealt, etc.

An example call to `take_turn` would be: `take_turn("guess", {"suspect": "scarlett", "room": "hall", "weapon": "leadpipe"})`. After a turn is taken, it is your responsibility to update who the current player is according to the spec.

The rest of these methods allow the UI and test infrastructure to peek into your game state.

- `get_current_player_name(self)`: This method is responsible for returning the name of the player whose turn it is. This method may or may not be called behind the scenes for other methods on our end! Keep in mind that this method will return different values throughout game play, since the player changes after a turn is taken.
- `get_players(self)`: Return a list of the game players (as strings). Order matters, and it should be the same order as was originally passed to you in state for initialization.
- `get_location(self, player_name)`: Returns the specified player's location in the mansion, i.e. which room they are currently in.
- `get_cards(self, player_name)`: Returns a dictionary of cards held by the specified player, where the keys are the card types.
- `get_revealed_cards(self)`: Get the list of cards revealed throughout game play, in the order in which they were revealed. For instance, if Burt asked Ernie: "guess scarlett in the study with the revolver" and then Ernie asked Burt "guess mustard in the library with the candlesticks" then `get_revealedcards()` would return `["study", "candlesticks"]`. Each card should appear at most once in the list.
- `get_seen_cards(self, player_name)`: Get the cards seen by this player in the order in which they were seen (as a list). For the above example, `get_seen_cards("Burt")` would return `["study"]`. Keep in mind that the only player that can see a given card is the player that made a guess and got an answer from their adversary (that's why "study" isn't in Ernie's list, regardless of the fact that it his own card).

In-Browser UI (`./server.py`)

Once your code outputs a valid result, you can visualize the output to help you debug your code. Run `server.py` and open your browser to localhost:8000. There will be a randomly generated case each time, and you can play for both players by making guesses to the console. The results returned use your code!

Unit Tester (`./test.py`)

As in the previous labs, we provide you with a `test.py` script to help you **verify** the correctness of your code. The script will call the required methods in `lab.py` and verify their output. In this lab, we encourage you to **create your own** controlled test cases in order to verify your logic and diagnose your bugs. (You may need to clear your browser cache if you create your own test cases.) The `test.py` script will run all `.in` test cases in `cases/`. There may be multiple valid, shortest paths. The tester verifies that the path is `valid` and is of the correct length.

We will only use the provided test cases to auto-grade your work. You may also wish to investigate the Python debugger (PDB) to help you find bugs efficiently.

Does your lab work? Do all tests in `test.py` pass? You're done! Submit your `lab.py` on funprog and get your lab checked off by a friendly staff member.