Important message on plagiarism

The single most important point for you to realize before the beginning of your studies at ShanghaiTech is the meaning of “plagiarism”:

Plagiarism is the practice of taking someone else's work or ideas and passing them off as one's own. It is the misrepresentation of the work of another as your own. It is academic theft; a serious infraction of a University honor code, and the latter is your responsibility to uphold. Instances of plagiarism or any other cheating will be reported to the university leadership, and will have serious consequences. Avoiding any form of plagiarism is in your own interest. If you plagiarize and it is unveiled at a later stage only, it will not only reflect badly on the university, but also on your image/career opportunities.

Plagiarism is academic misconduct, and we take it very serious at ShanghaiTech. In the past we have had lots of problems related to plagiarism especially with newly arriving students, so it is important to get this right upfront:

You may…
• … discuss with your peers about course material.
• … discuss generally about the programming language, some features, or abstract lines of code. As long as it is not directly related to any homework, but formulated in a general, abstract way, such discussion is acceptable.
• … share test cases with each other.
• … help each other with setting up the development environment etc.

You may not …
• … read, possess, copy or submit the solution code of anyone else (including people outside this course or university)!
• … receive direct help from someone else (i.e. a direct communication of some lines of code, no matter if it is visual, verbal, or written)!
• … give direct help to someone else. Helping one of your peers by letting him read your code or communicating even just part of the solution in written or in verbal form will have equal consequences.
• … gain access to another one’s account, no matter if with or without permission.
• … give your account access to another student. It is your responsibility to keep your account safe, always log out, and choose a safe password. Do not just share access to your computer with other students without prior lock--out and disabling of automatic login functionality. Do not just leave your computer on without a lock even if it is just for the sake of a 5--minute break.
• … work in teams. You may meet to discuss generally about the material, but any work on the homework is to be done individually and in privacy. Remember, you may not allow anyone to even just read your source code.

With the Internet, "paste", and "share" are easy operations. Don’t think that it is easy to hide and that we will not find you, we have just as easy to use, fully automatic and intelligent tools that will identify any potential cases of plagiarism. And do not think that being the original author will make any difference. Sharing an original solution with others is just as unethical as using someone else’s work.

SI 100B PA 3

SI 100B TA Team

November 16, 2023

0 *ONU*

Welcome to *ONU*! *ONU* is a card game played with a specially printed deck. The objective of *ONU* is to be the first player to get rid of all your cards. Firstly, let's go over the basic rules of *ONU*:

0.1 Gameplay

- By default, the number of players is set to 7, and the initial hand card count is also set to 7. Users can customize the number of players and the initial hand card count based on their needs. The rest of the cards are placed in a deck.
- The game begins. The first player could play any card he/she wants.
- There are numeric cards and special cards in the deck. The numeric cards contain color and number, while the special cards have unique effects that can be placed on the next player.
- The players take turns to play their cards. They must match the card played by the last player either by number, color, or effect.
- If a player is not prevented from playing cards by special cards, but there is no card to play due to the different color and number, he/she must draw a card from the deck, and then his/her turn ends without playing any cards.
- When a player has played all his/her hand, he/she wins the game. However, if the deck is emptied before any player has played all of their cards, the scores are calculated based on the remaining cards in each player's hand. The player with the lowest score wins the game.

0.2 Card Types

- Numeric Cards: These cards have a number on the card and also have a color. They can only be played on a card that matches by color or number.
- Change Color Cards: These cards can be placed on any card and change the current color to the color specified by the Change Color Cards.
- Ban Cards: When the player places these cards, the next player has to skip his/her turn. The Ban Cards have a color and can only be played on a card that matches by the color, or on another Ban card.
- Plus-Two Cards: When the player place these cards, the next player will have to pick up two cards and skip his/her turn. The Plus-Two Cards have a color and can only be played on a card that matches by color, or on another Plus-Two.

Special rule: If the last player plays a Plus-Two Card, and the current player also has a Plus-Two Card, the current player can play the card instead of drawing and the next player has to either draw 4 cards or also play a Plus-Two card, and so on. That is, **the Plus-Two Card can be accumulated**. It would be a good choice to record the cumulative value with a variable.

0.3 General Description

In this assignment, you will be tasked with implementing an *ONU* game. The assignment involves the following classes: `Card`, `CardSet`, `Player` and `Game`. Your objective is to complete some functions within these classes.

- The `Card` class is the basic class in the game. There are two types of cards, `NumericCard`, which includes the Numeric Cards and `SpecialCard`, which includes Change Color Cards, Ban Cards and Plus-Two Cards. They are all inherited from `Card`.
- The `CardSet` class is used to represent a set of cards and provides the operations to manipulate card sets. It has two subclasses: `Hand` and `Deck`. The `Hand` class is used for operations related to a player's hand, while the `Deck` class is used for operations related to the game's deck.
- The `Player` class represents the player agent. The only thing one player could do is to make an action and choose a card from his/her hand to play if possible.
- The `Game` class represents the *ONU* game itself and is responsible for managing the entire game process, including handling player actions, maintaining the deck, and detecting game termination.

Notes: Write your code in the designated areas and avoid modifying other parts of the code. You may add additional helper functions or attributes as needed.

0.4 Local Testing

To facilitate your testing of the function implementation, we have provided Python testing scripts. It is recommended to place the test files in the same directory as the ONU folder or the same directory as the template folder. You can also modify the import path based on your own directory structure, and instructions are provided within the test files.

Running these programs will provide you with results such as pass, fail, and other outcomes, along with the expected output and the output produced by your program. This will help you identify the reasons for any errors and facilitate the debugging process.

Good luck with your implementation of *ONU*!

1 Card (20 Points)

1.1 Description

In this programming assignment, we will start by implementing the most fundamental unit of the game, which is the `Card` class. The `__init__` method for `Card` is written for you. It has one member variable: `_color`, to store the color of each individual card object created from the `Card` class. The `color` is given by an Enum `Color`.

```
class Color(IntEnum):
    RED = 0
    YELLOW = 1
    GREEN = 2
    BLUE = 3
    CYAN = 4
    ORANGE = 5
    PURPLE = 6
    WHITE = 7
    BLACK = 8
    VIOLET = 9
```

By assigning the parameter color to the member variable `self._color`, you ensure that each individual card object has its own color value stored as part of its state. Now, it's your task to override some methods in the subclass `NumericCard` and `SpecialCard`.

You will encounter a function called `__repr__` in this part. It is a special method used to return a string representation of an object. When you use the print function or directly input an object in the interactive environment in Python, you are actually invoking the object's `__repr__` method.

1.2 Methods

- `Card.get_color()`: Return the color of the card.
- `NumericCard.__init__(color, number)`: Initialize the necessary parameters for `NumericCard`, which is the number of the card. The input would be the Enum color and an integer number. Since the `color` variable is set by `Card` from `super().__init__()`, you don't need to worry about it.
- `NumericCard.__repr__()`: Return a string for printing the information of card. For numeric cards, the function should return "Numeric card" followed by their color name and number. For example, card "red 4" gets "Numeric card RED 4". The color name is defined in Enum `Color`.
- `NumericCard.__lt__(other)`: Compare the card with another card `other`. First, the color is compared based on the order in Enum `Color`. Next, if they are in the same color, compare their numbers.

When it comes to different types in the comparison of numbers, the Numeric Cards are smaller than Special Cards. Return `True` if the card is less than `other` and return `False` otherwise.

- `NumericCard.__eq__(other)`: Compare the card with another card `other` to see whether they are the same. Return `True` if their parameters are equal and return `False` otherwise.
- `NumericCard.get_number()`: Return the number of the card.
- `SpecialCard.__init__(color, effect)`: Initialize the necessary parameters for `SpecialCard`, which is the type of the Special Card. This is given by Enum `Effect`, which is defined below:

```
class Effect(Enum):
    CHANGE_COLOR = 0
    BAN = 1
    PLUS_TWO = 2
```

- `SpecialCard.__repr__()`: Return a string for printing the information of card. For special cards, it should return the name of the special card , according to Enum `Effect`, followed by its color name according to Enum `Color` similarly. An example is given by "BAN card YELLOW".
- `SpecialCard.__lt__(other)`: Compare the card with another Card `other`. First, the color is compared based on the order in Enum `Color`. When they are in the same color, if `other` is a Numeric Card, it is smaller than the Special Card. Otherwise, the Special Cards are compared based on the order in `Effect`. Return `True` if the card is less than `other` and return `False` otherwise.
- `SpecialCard.__eq__(other)`: Compare the card with another card `other` to see whether they are the same. Return `True` if their parameters are equal and return `False` otherwise.
- `SpecialCard.get_effect()`: Return the effect of the Special Card.

By completing this task, you will have the basic building block of the game, allowing you to represent and visualize the different cards used in gameplay.

2 CardSet (20 Points)

2.1 Description

Next, let's implement the `CardSet`, `Hand` and `Deck`. The `CardSet` class is the base class that represents a set of cards. It provides common functionality and operations that can be performed on a collection of cards.

The `Hand` class is a subclass of `CardSet` and represents a player's hand cards. It provides additional methods to manipulate a player's hand. This includes actions such as adding cards to the hand and removing cards from the hand.

The `Deck` class is also a subclass of `CardSet` and represents a game deck. In addition to the common functionality inherited from `CardSet`, it provides the functionality to retrieve the top card from the deck.

Note that we do not need to override the `__init__` method for `Hand` and `Deck` because they do not have any differences from the base class `CardSet` in initialization. Therefore, the subclasses inherit the initialization and other methods from the base class directly.

2.2 Methods

- `CardSet.__init__(cards)`: Save the cards given by the list `cards` in the variable `_cards`.
- `CardSet.__repr__()`: Return a string that prints all the cards in the cardset. It would be a list that containing every card.

  Here is an example:

  ```
  card1 = NumericCard(Color.BLUE, 1)
  card2 = SpecialCard(Color.BLACK, Effect.CHANGE_COLOR)
  card3 = NumericCard(Color.GREEN, 1)

  Cards = CardSet([card1, card2, card3])
  print(Cards)
  ```

  The output would be

  ```
  [Numeric card BLUE 1, CHANGE_COLOR card BLACK, Numeric card GREEN 1]
  ```

- `CardSet.is_empty()`: Return `True` if there is no cards left in the cardset, and return `False` otherwise.
- `Hand.add_card(card)`: Add the given `card` into the end of the cards.
- `Hand.remove_card(card)`: Remove the first occurrence of the given `card` from the cards.

- `Deck.get_next_card()`: Return the top of the deck, which would be the first card in the list. After this card is returned, it is no longer in the deck.

3 Player (30 Points)

3.1 Description

Next, let's implement the `Player` class. In our game, players are considered to be willing to do anything for victory, including cheating. Therefore, their behavior is strictly controlled by the game. The player's hand is managed by the game system, and they can only make choices based on their hand and the previous player's actions. The actual gameplay actions, such as playing cards or drawing cards, are all executed by the game system. Also due to this reason, players do not have their own attributes, so the `__init__` method does not perform any operations. Your task is to assist the players in making their choices.

3.2 Methods

- `Player.sort_cards(cards)`: Return a sorted version of the input list `cards` in descending order, which means from largest to smallest.

- `Player.action(cards, last_card, is_last_player_drop)`: The inputs consist of the player's hand, the card last played, and whether the last player has played any card. They are helpful when making the decisions. The returned value is a tuple with two items to store the actions. The first element of the tuple is an Enum `ActionType` representing the actions. The second element represents the card to play. If the player could not play any card, the second element would be `None`.

  ```
  class ActionType(Enum):
      DRAW = 0
      DROP = 1
      PASS = 2
  ```

  To be detailed, the player should first check the status based on the actions of the previous player.

  If the previous card played was a Ban Card, the player could do nothing, so return `ActionType.PASS` and `None`.

  If the card played was a Plus-Two Card, and the player has no Plus-Two Card to play, similarly, the player could do nothing and returns `ActionType.PASS` and `None`. The penalty of drawing cards in this case will be enforced externally, so we return ”PASS” to indicate the ”penalized” state. If the player has one or more Plus-Two Cards to avoid being penalized, just choose a larger one to play, which returns `ActionType.DROP` and the card.

  If a player is not influenced by any special cards but has no playable card in his hand to play, he/she needs to draw a card instead. Return `ActionType.DRAW` and `None`. Note that the `ActionType.DRAW` action is specific to the scenario where a player has no playable cards and is not influenced by any special cards.

If a player can play a card, return `ActionType.DROP` and the chosen card. To choose a card to play, you always play the largest card that is available. Here the definition of ”largest” is the same as the method `__lt__` in the Cards.

One may be curious about why `Player` needs to know `is_last_player_drop`. For example, if the previous player played a Ban card, the current player is prevented from playing any cards, and the variable `is_last_player_drop` would be set to `False`. On the next player’s turn, he/she knows that the `last_card` Ban card has taken effect through this variable, and he/she can play a normal game turn. You should consider all similar situations.

4 Game Start! (30 Points)

4.1 Description

Next, let's implement some methods in the `Game` class to prepare for the start of the game. Here are some attributes defined in `Game.__init__()`.

`Game._deck` represents the deck of cards used in the game. It is initialized with the `Deck` object, which is created from the provided cards list.

The players are initialized as a list `Game._players`. The number of players is set in `num_player` and it defaults to 7 if not provided.

`Game._current_player_id` represents the index of the current player. It is initialized with the value of `dealer_id` parameter minus 1. This assumes that the `dealer_id` is a zero-based index representing the starting player. The subtraction of one (dealer_id - 1) is done to return the index of the player when the next player is calculated for the first time.

`Game._last_card` represents the last card played in the game. It is initially set to None and will be updated whenever a player plays a card.

`Game._is_last_player_drop` represents whether the last player had played any card. It is initially set to `False`. It can be used to determine whether the last card played was played by the last player.

`Game._plus_two_cnt` represents the count of unexecuted Plus-Two cards played. It may be helpful when treating the accumulation of consecutive Plus-Two Cards. It is initially set to 0 and can be incremented when a player plays a Plus-Two Card. After one or more Plus-Two Cards are executed on a player, this value should be reset to zero.

After all the basic components are built, our game is ready to start! Players take turns playing cards or drawing cards until a player has no cards left in his/her hand, indicating victory. If the deck is empty, the game is also stopped. All these operations are implemented in the method `Game.turn()`.

4.2 Methods

- `Game.__init__(cards, num_player, hand_card_num, dealer_id)`: Most of the initialization is done by the template. The only thing you need to do is to deal `hand_card_num` cards to each player as their initial hand. After one player has finished taking the initial hand, it is then the next player's turn to take. The hand are recorded in `Game._player_hands`. The deck of cards contains more cards than the initial hand size, so there is no need to worry about the deck running out of cards before the game starts.

- `Game.is_end()`: This function is used to check whether the game is ended. Return `True` if there is any player with no hand cards, or if the deck is empty. Otherwise, return `False`. You may notice that there is a method `Game.is_not_end()`. It is used for the return in `Game.turn()` to improve code readability.

- `Game.current_player_drop_card(card)`: To handle the process of a player playing a card, remove the played card from the player's hand. Besides, update the relevant attributes based on the played card.
- `Game.get_scores()`: Calculate the scores for each player and return the scores as a list.

  The score consists of two parts. The first part is the sum of the number of all Numeric Cards in the hand. The second part is the product of 10 (for the punishment of keeping Special Cards) and the sum of the Effect values plus 1 of all Special Cards. The two parts are added together to get a total score.

  Here is an example. If the hand contains 4 cards, with 2 Numeric Cards ”blue 7” and ”red 5”, and 2 Special Cards ”Ban card green” and ”Change color card yellow”, the total score for this player would be (7+5) + 10*((1+1)+(0+1)) = 42.
- `Game.get_winner()`: Return the index of the player who wins the game. If the game ends because the deck is depleted and no player has finished their hand, you can calculate the scores for all players, and the player with the lowest score wins. If there is a tie, the player with the lowest index is considered the winner.
- `Game.turn()`: The player's actions are executed through the `Game` agent. It retrieves the next player's information and allows them to perform their actions. Based on the player's action, such as drawing a card, playing a card, or being passed, `Game` handles the execution of these actions within the game. Additionally, in the case of Plus-Two Cards, the player may have to draw multiple cards as specified by the card's effect. Ensure that the cards are inserted in the order they were drawn.

  After a special card has affected a player, subsequent players are no longer impacted by that card's effect. Instead, they continue to play their turns as usual, without any special restrictions or requirements imposed by the previously played special card.

  Once `Game` has processed the action, return `action`, `Game.get_info()`, `Game.is_not_end()` as the result of this round.