**ASSIGNEMENT 2 REPORT**

**SUMMARY**

In the implementation of Assisgnment 2, the NotFreeCell game is coded using 3 different class implementations, namely:

* Card
* Deck
* NotFreeCell

The the first two classes Card and Deck are created as stand alone classes and hence are not associated with the game NotFreeCell. Because of this both these classes have not been created with any restrictions based on the standard 52 deck card. The class Freecell then uses these classes to instantiate meaning into the deck and cards created (defining face, suit, color, number of cards, hierarchy of cards etc) and then creates a general model for the game that can further be played according to the set rules.

The following section of the document provides details on some of the design implementations of the above mentioned classes.

**CLASSES**

**Class: Card**

**Imports: None**

|  |  |  |
| --- | --- | --- |
| Method | Arguments | Description |
| \_\_init\_\_ | face, suit, color (optional) | constructor |
| \_\_str\_\_ | none | string conversion for object of class card |
| get\_face | none | returns the value of the face of the card |
| get\_suit | none | returns the value of the suit of the card |
| get\_color | none | returns the value of the color of the card |
| change\_face | face | changes the face of the card |
| change\_suit | suit | changes the suit of the card |
| change\_color | color | changes the color of the card |

This class is created keeping in mind that a card must have a face and a suit with an optional argument for color. If the color for the card is stated in constructing an object of class Card the color is assigned that speficific value otherwise a default value of None is set.

There is no restriction on any of the argument entries type (e.g. Integer, string etc) as they would be saved as string.

When printing an object of class Card the object is displayed with its full face, suit and color seperated by a colon, for example an ace of spade might be displayed as ‘ace:spade:None’ or ‘ace:spade:black’ depending on whether the color was set on instantiation or not. A shorter version such as ‘A:s:b’ is not done since the card can accept any type of arguments. If the arguments are in the form of integers both cards of faces of integer 1 and 11 with suit 11 and 1 will be shown as the same ‘1:1:N’ (for 1:11:None) and ‘1:1:N’ (for 11:1:None) in which case differentiating between the two would prove impossible.

In addition to this the standard functions for accessing all three attributes of the card (face, suit and color) are created as well as their mutators in case any of these are to be changed.

**Class: Deck**

**Imports: (a) Card (b) randint from Random**

|  |  |  |
| --- | --- | --- |
| Method | Argument | Description |
| \_\_init\_\_ | start\_val, end\_val, nbr\_of\_suits | constructor |
| \_\_str\_\_ | none | string conversion of object of class deck |
| card\_list | none | returns a list containing all cards (objects) |
| shuffle\_cards | none | shuffles the positions of cards in deck |
| check\_deck | face, suit | returns true if card of face and suit exists else returns false |
| check\_index | face, suit | returns the index of the card with the given face and suit |
| add\_card | face, suit, index (optional) | adds a card of the given face and suit if it doesn’t exist at given index else at end |
| add\_card\_rand | face, suit | adds a card af the given face and suit at a random index |
| add\_card\_instance | card, index (optional) | adds to deck an existing object of class card if a card of that face and suit doesn’t exist at the given index else at end |
| add\_card\_instance\_rand | card | adds to deck an existing object of class card if a card of that face and suit doesn’t exist at a random index |
| draw\_card | face, suit | draws a particular card |
| draw\_top | none | draws the last card in the deck list since the top card is the one entered last |
| draw\_bottom | none | draws the first card in the deck list since the bottom card is the one entered first |
| draw\_rand | none | draws a random card from deck |

The Deck class creates a deck containing a number of objects of the class Card. The deck class does not have any restrictions on the maximum number of cards/suits in a deck. A deck can be created with any number of cards with any number of suits. To keep this functionality the deck creates cards having faces and suits of integers. Such as 1:2:None where the face is 1, the suit is 2 and the color is None.

The Deck however does not define the color of the cards and they are assigned the default value of None. This is because the deck itself does not contain meaning for the card face, suit or color. It depends on the game using the deck which defines what the meaning of each card is in its game.

Inorder to create the deck class this implementation makes use of the class Card as well as randint function from Random. To be precise:

* The Card class is imported inorder to fill the deck with the required number of the card objects.
* The randint function from random is used to create a method to shuffle the deck of cards.

The Deck class uses a total of three arguments as Integers:

1. Starting value for the deck; the starting face value for the first card in a suit
2. Ending value for the deck; the ending value for the last card in a suit
3. Number of suits in the deck; defines the total number of suits to be created in a deck

Keeping this in mind a deck created using the following arguments: Deck (1, 4, 2) would create the following:

1:2:None

2:2:None

3:2:None

4:2:None

The magic function \_\_str\_\_ is defined to print an object of class Deck in the same manner as above. All cards in the deck are printed as their print form (face:suit:color) in separate lines.

A deck of cards is used in a large number of card games – every card game has a unique style of playing with deck. In order to properly make the deck class usable in most of the games the deck class had to be made with multiple methods to provide it with flexibility of use.

Once the deck is created shuffling a deck, checking whether a card exists in a deck and its position are vital for playing cards. In addition to these the two main senarious that could occur are adding a card to the deck or drawing a card from it. The latter two senarios can then be split in multiple other senarios.

Adding a card to the deck

* A card can be created and then added
  + In a specific position
  + In a random position
* An existing card can be added to a deck (if a game uses two decks and wants to draw a card from one and then place it into another deck). The position of adding this card can vary hence the card can be inserted
  + In a specific position
  + In a random position
* Adding a random card into a deck was not possible to implement as there can me infited number of possibilities of inserting the type of card (depends on the game being played)

Cards can be drawn from a deck in various ways

* A specific card needs to be drawn from a deck (for example I want to draw an Ace of Spades)
* A card can be drawn from the top of the deck
* A card can be drawn from the bottom of the deck
* A card can be drawn from a random position in the deck (for example in a magic tricks: ‘pick any card’)

**Class: NotFreeCell**

**Imports: (a) Deck**

|  |  |  |
| --- | --- | --- |
| Method | Arguments | Description |
| \_\_init\_\_ | none | 1. creates deck 2. assigns meaning to deck (color, faces, suits) 3. creates dictionary for card heirarchy 4. shuffles deck 5. creates lists for cascades, foundations, cell slots 6. fills cascades using deck |
| \_\_str\_\_ | none | prints the face of the game in freecell form |
| card\_val | card | checks a dictionary that contains the hierarchy integer value of the card ( 5 will have a higher value than 4 according to freecell) and returns the value of that card |
| move\_card | from\_list, to\_list | moves a card (if possible) from a cascade, foundation, or cell slot to another cascade, foundation, or cell slot according to the freecell rules |
| game\_won | none | checks the current game and returns true if won else false |
| play | none | 1. prints game 2. prints menu for user inputs 3. takes inputs from users 4. moves cards according to inputs 5. prints error messages if any or illegal moves 6. prints game again |

The NotFreeCell class uses a deck class to create a deck and give it meaning according to a standard 52 deck. It also maintains a dictionary for each card with its integer value. At the back end the NotFreecell class maintains a total of 16 lists for the 8 cascades, 4 cell slots and 4 foundations. The cascades are then dealt the 52 deck cards and from there entire game is played by shifting the 52 cards amongst the 16 lists of the game according to certain freecell rules.

In order to print the game the \_\_str\_\_ method is used to print the required contents of the 16 lists onto the output screen according to the freecell rules making them easier to understand for the user. The phrase required contents refers to for example showing only the top item on the foundation list and not the entire list to the user.

This class only makes use of one accessor which is the card value accessor. This returns the the integer value of a card according to its heirarchy. Accessors for individual lists such as the first cascade or the second cell slot etc. are not created as the game only has meaning when viewed as a whole and not as individual columns or lists.

The move card method takes the list to move a card from and the list to move a card to as arguments and moves the card if it is possible according to FreeCell rules otherwise it prints an error message.

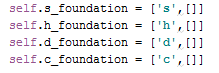
The game\_won method just checks the condition if all the foundations have all 13 cards. If they do then returns true as the game has been won, else it returns False.

The Play method is just a large while loop that keeps looping and taking inputs from the user and moving cards until either the game has been won or the user inputs the quit key. The play method for each move identifies the from\_list and the to\_list by printing options and taking inputs from users. After obtaining both the lists it inputs them into the move method to move the cards and prints the new position of the game and loops again.

**Detailed Analysis:**

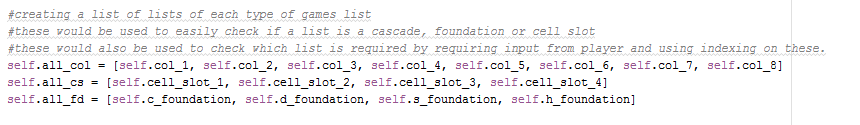
**Foundations in \_\_init\_\_ method:**

The NotFreecell class keeps track of 16 lists. Each of these lists are arrays containing cards except for 4 of the foundation lists. The four foundation lists are created as a list containing the string representation of the suit and the list containing cards.



The foundations are created in such a way to easily check with a card if the card is of the same suit or not. Hence before inputing a card in the foundation the move method compares the suit of the card being moved with the string at index 0 of the foundation to check if they are the same. If it is then it appends the card into the inner list at index 1 of the foundation list.

**List containing all relative lists:**



A list containing for each cascades, cell slots and foundations are created. These lists are then used for two purposes:

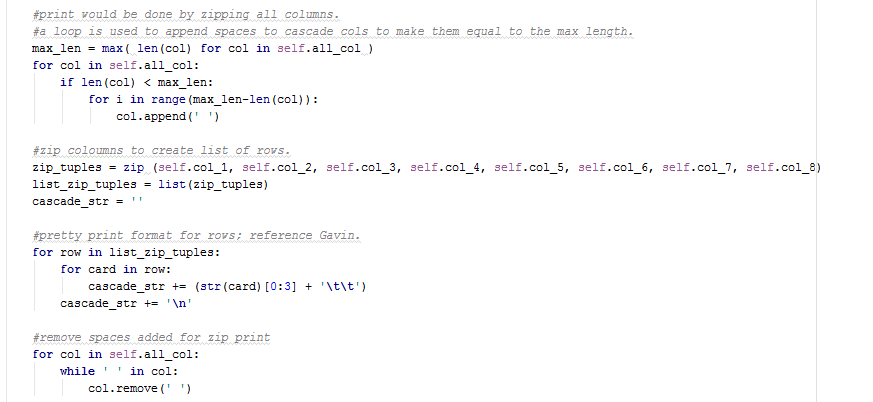
* to easily identify whether a list is of a cascades, cell slot or foundation
* to identify the list the user trys to input during the menu while playing the game using the method Play().

**Card\_value:**

This dictionary is vital for the code to understand the correct sequence in which the card hierarchy works. The value is assigned according to the face of the card regardless of its suit or color. The card values are as follows:

|  |  |
| --- | --- |
| Card Face | Card Value |
| A | 1 |
| 2 | 2 |
| 3 | 3 |
| 4 | 4 |
| 5 | 5 |
| 6 | 6 |
| 7 | 7 |
| 8 | 8 |
| 9 | 9 |
| 10 | 10 |
| J | 11 |
| Q | 12 |
| K | 13 |

**Printing cascades in \_\_str\_\_:**



**A**

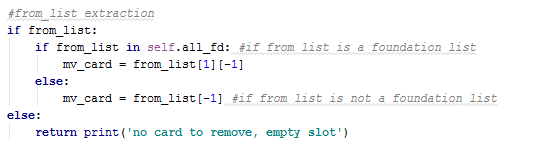
**D**

**C**

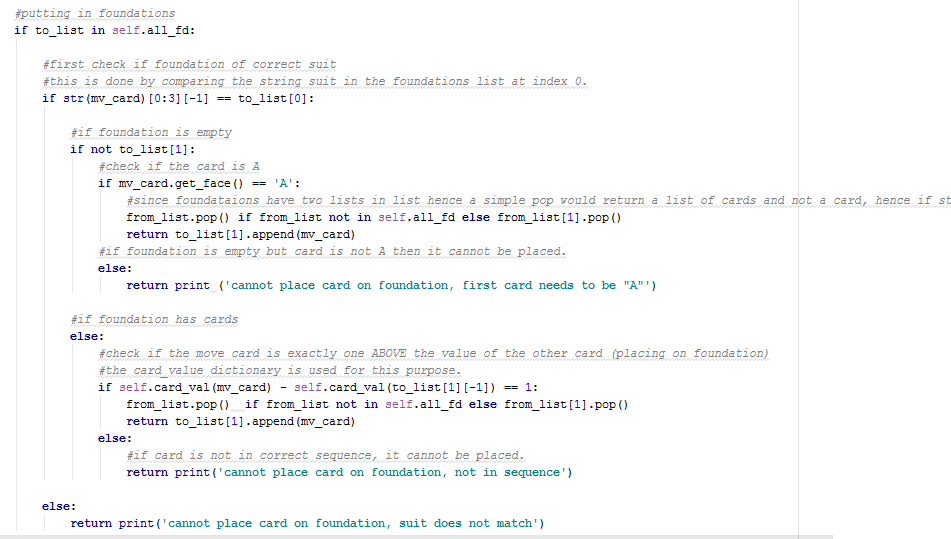
**B**

1. The 8 cascades are converted into the same length as the length of the biggest casacde. This is done by appending white space into the list until its length is equal to that of the largest one.
2. Once the lengths of all cascades are the same zip() is used to zip the 8 cascades together to form lists of rows (list1[0], list2[0], list3[0] etc.)
3. A new string is created by adding into it each row the cards. The cards are displayed using only the first 3 characters of the card (the face which is one character, the colon, and the first char of the suit)
4. Once the string is created the added white spaces are removed from the cascade lists to return them into their original form.

**Moving a Card using move\_card (from\_list, to\_list):**



The code first stores (but does not delete) the last card (card that needs to be moved) from the to\_list. The move card is not deleted as in case the card cannot be placed the card will have to be placed back, hence to avoid that scenario the card is removed once the checks are done. Since the foundations have a different format from the rest of the lists and if statement is used to correctly store the last card if its in the foundation or in another list.



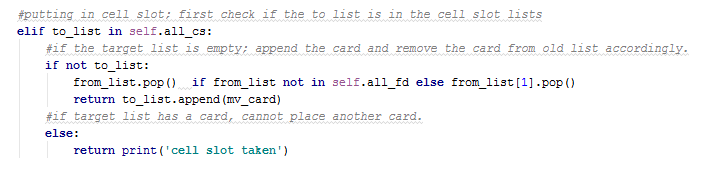
**A**

**C**

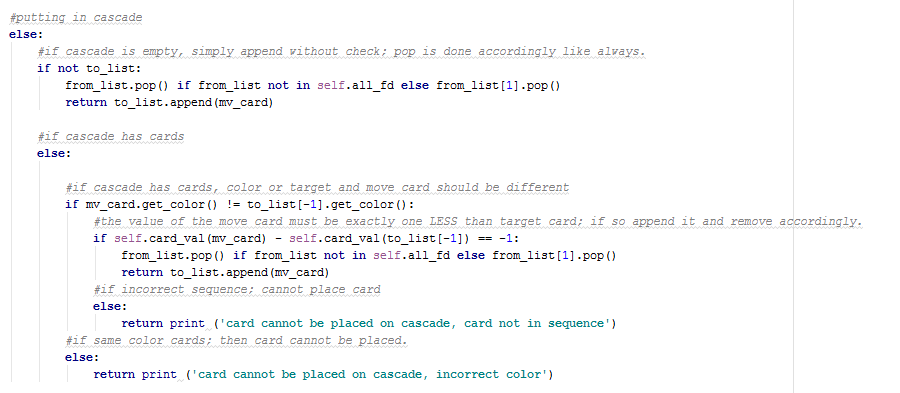
**B**

Once the move card is stored. The code checks if the target list is a foundation list. If it is then the following checks are done:

1. The code checks wheter the card and the foundation are of the same suit.
2. If the foundation is empty the the code checks if the card is an ‘A’ and added only if it is true. The first card of the foundation is always an ‘A’ in freecell.
3. If the foundation is not empty then the card\_value of the move card and the last card of the foundation are compared. If the move card is one HIGHER than the other then it is added.



The only check made for the cell slots is if the cell slot is empty. If that is the case the move card is added otherwise not.



**B**

**C**

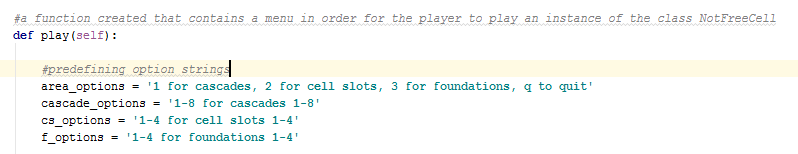
**A**

When adding to cascades checks:

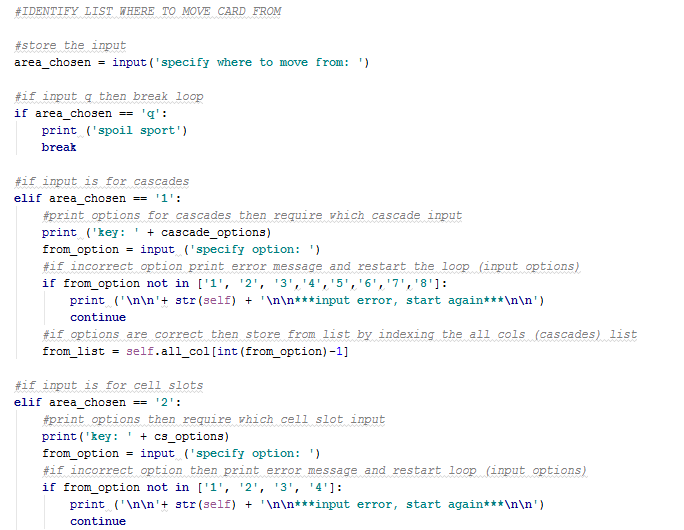
1. Whether the cascade is empty if it is then adds it
2. If it is not it checks if the color of the move card and the color of the last card in the target list are DIFFERENT.
3. If the colors are different the card value of the two cards is compared and the card is added only if the value of the move card is lower than the other cards.

In all the checks since the move card was not removed from the from\_list in the beginning before adding the card in the other list, the move card is removed from the original list.

**Play Method:**



The play method makes use of defined strings that it prints as menu directions for the users. It tells the user the key for the columns and uses it to take input and identify the list from which the user wants to move the card from and the list the user wants to move the card to.



**D**

**C**

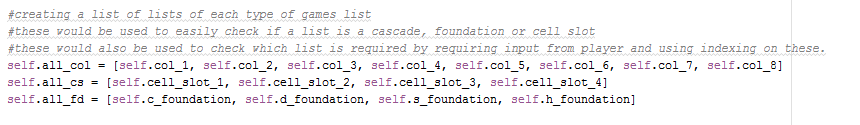
**B**

**A**

1. To identify the list from where the user wants to move the card from the code first takes it as input and stores it. (e.g. 1 for cascades, 2 for cell slots and 3 for foundations)

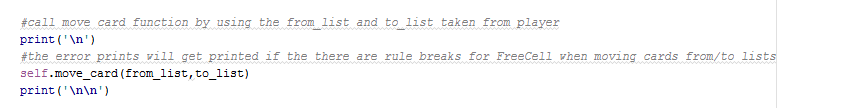
Once the option is choses multiple if statements check whethere the options chosen is for cascades, cell slots or foundations.

1. According to the area choses another option is printed to take input the column in that area. (e.g. column 1 of cascade, or the 2nd cell slot etc). These options are kept in such a way that the option can then be used to identify the column name by indexing the all column lists created shown below.



1. If the column entered is incorrect an error message is printed and the loop is restated using the CONTINUE command. When the continue command is reached the user has to redo his selection from the beginning.
2. According to the column chosen the from coloumn is identified using the indexing from the all cascade, cell slot and foudation lists.

The same method is used to identify the target column where the card needs to be moved to. After identifying both the columns the columns are entered into the move method to move the card as shown below.



After evey move done the play method makes use of the game won funtion to identify if the last move caused the player to win the game. Hence before the loop restarts the game funtion is called as shown below:

