

Assignment2:NotFreecell

Anushree Mahajan ID:28948203

FIT9133: Introduction to Python

9/21/17

TABLE OF CONTENTS

Introduction	2
1.1 The Rules	2
1.2 The Requirements	2
Solution Design	3
Card Class:	3
Deck Class:	5
Attributes and Methods of Deck class:	5
__repr__ and __str__:	6
Testing Deck Class:	6
NotFreecell	7
Card Movements:	9

INTRODUCTION

The purpose of this report is to create a game Not Freecell which is clone of the Freecell game developed by Microsoft corp. Freecell game is made up of cards of 4 suits i.e Diamond, Heart, Spade and Club each consisting of 13 cards.

1.1 THE RULES

The game follows the following rules:

- Only one card may be moved at a time.
- The four foundations must be built starting from the Ace of the appropriate suit, followed by the 2, then the 3 etc. until the King is placed.
- If a card is placed on a cascade, it must be placed on a card of the opposite colour, and of a suit that is one higher than itself. E.g. a red 2 can be placed on a black 3, but not a black 4.
- Any card may be placed in an empty cascade.
- Cards from the foundations may be placed back onto a cascade, or an empty cascade slot.
- Only one card at a time can occupy a cell slot.
- Victory is achieved when all four foundations are filled with their respective suits from Ace to King.

1.2 THE REQUIREMENTS

As part of your program, you will need the following:

- A class called Card that represents a card in play. Card must also contain the following:
 - The attribute card_face and card_suit that contain the face value and suit of the card respectively.
- A class called Deck that is made up of cards. Deck must also contain the following:
 - An __init__(self, value_start, value_end, number_of_suits) so that a deck of cards can be constructed using a known start value, a known end value, and a known number of suits. E.g. Deck(1,11,1) will generate a deck consisting of Ace to 9 and Jack in a single suit, thus a deck of 11 cards. A shuffle method to randomize the cards. A function to add a card to the deck. A function to draw a card.
- A class called NotFreecell that uses a deck and implements the rules of freecell.
- NotFreeCell should be playable, and thus is able to create new games and know when the game has ended as a result of victory.

SOLUTION DESIGN

Solution consists of three classes-

- Card class which is a blueprint to create cards.
- Deck class which is made using instances of Card class.
- NotFreecell class which implements all the game rules associated with the game.

CARD CLASS:

A class defines a real-world entity. A class is associated with attributes i.e data and behavior i.e method so that it should resemble the real world entity as closely as possible. The card class created has following four attributes- face, suit, card value and card color.

When we initialize the Card class we set up face and suit value of the instance.

Eg: card1=Card('1','Spade')

__init__ method:

When we initialize the class by using the above statement, an object called card1 is created. Whenever we create an instance of a class __init__ method is run. The __init__ method is like a constructor which sets up the default parameters whenever an instance is created. By default the first parameter of __init__ methods is an argument called self which signifies it uses the values passed by the instance which is created and is different for two objects. If we create an object called card2 with same values i.e card2=Card('1','Spade') and if we compare both the objects then it wont be equal. Python treats them as separate instances even though they have same value.

Attributes declared within this method -face value and suit value.

Property decorators for card value and card colors:

There are certain attributes which can be derived from other attributes which are specified in the __init__ method. We can create such attributes by defining methods for them or by using property decorators. Property decorator allows us to access a method like an attribute. It functions similar to getter or setter in Java.

Eg: Suits Spade and Club have black color and suits Diamond and Heart have red color.

If we define color within __init__ method as follows:

```
def __init__(self,face,suit):
    self.face=face
    self.suit=suit
    if self.suit.capitalize() in ('Heart','Diamond'):
        self.color='Red'
    elif self.suit.capitalize() in ('Club','Spade'):
        color='Black'
```

card1=('1','Spade') will have Black color. But now if we change the suit value of the instance to Heart or Diamond by setting the attribute individually then the suit color would remain same as the color attribute is defined in __init__ method which is only run when a new instance is created. However if we create a method for it, lets say set_color(self) then it wouldn't be a normal data access you'll have to access the method card1.set_color(),this means if you have used color attribute of the Class card anywhere else

you'll have to make changes at every place by changing the way the attribute is accessed. To avoid such hassles we use property decorators which allow us to access the methods as normal data attributes without parenthesis.

Similarly a property decorator is created for setting up the card values based on their faces.

Ace will have card_value of 1, Jack→card value→11 and so on Basically card value signifies the rank of the cards.

Operator overloading (__eq__ and __lt__):

As discussed earlier if we create two objects with the same values, Python treats them as separate entities even if they are actually the same card. In order to do comparison between a card object and another card object or string we use operator overloading. In operator overloading we redefine the criteria for comparison in this case so that two objects can be compared easily.

```
def __eq__(self,other):  
    if isinstance(other,Card):  
        return self.face==other.face and self.suit==other.suit  
    elif isinstance(other,str):  
        return str(self)==other
```

So in this case if two instances have same suit and face value then they would no longer be considered not equal to each other. Similarly we have used operator overloading to find whether the card is less than other.

__repr__:

This method is called when we run repr() on instance of the class created. It is an unambiguous representation of an object and is meant to be seen by other developers. Have returned the value of the class instance.

__str__:

This method is called when we run str() on instance of the class created. It is readable representation of the object and is meant to be seen by the end user. Have returned face value and first letter of suit in it.

Testing Card class:

```
def main():  
    card1=Card('1','Spade')  
    card2=Card('1','Spade')  
    print(card1==card2)  
    print(card1<card2)  
    print (repr(card1))  
    print (card1)  
    print(card1.colors)
```

Output

```
In [6]: runfile('C:/Users/Dell1/Desktop/learn_python/Assignment2/assignment2/cards.py',
wdir='C:/Users/Dell1/Desktop/learn_python/Assignment2/assignment2')
True
False
Card(1,Spade)
1:S
Black
```

DECK CLASS

Deck is made up of cards. Deck class created is a generalized deck which can be used for creating deck of any card games. Deck has to be initialized by using start value, end value and the number of suits required in a game. Ex: Deck(1,10,5) will have 10 cards each of 5 different suits. However I have created a customized suit which is consisting of 4 suits(Spade, Ace, Heart and Diamond) having 13 cards each which can be used for the freecell game. Inorder to create that deck, I have created 2 lists namely face_list and suit_list which has values of all the faces and suits respectively of a normal deck which is used to create instances of the Card class object which can later be used to create deck for Freecell.

Attributes and Methods of Deck class:

__init__ method:

The values declared in __init__ method are start_value, end_value and suit_number which are used to initialize the class. Apart from these values, card_list(to create a list of the cards), cards_played and card_draw (to keep a track of the played cards which can be used to monitor the history of the moves as the game progresses.)

Method to add cards to the deck(add_card()):

This method is used to create a deck of class using the Card class. As mentioned earlier, there are two types of deck created – one which can be used for any game and another a customized deck to be used by freecell(when suit_number=4). The resultant deck is stored in a list called card_list.

Method to keep a track of the moves played(draw_card(card)):

This method is used to keep a track of the moves played and is used when we want to undo the move which was played earlier. This method is currently not used in the NotFreecell but can be implemented later to improve the game play by keeping a track of its move. This method removes the card from cards_played list and appends it to the cards_draw list.

Method to shuffle cards(shuffle_card()):

Deck of cards when dealt in any game are shuffled so that each deck is unique. The card list is shuffled using random module.

Method to deal cards(deal_card()):

The deck class i.e the card list generated is used to play the game. This method is used to remove cards from the card_list and append it to the cards_played which has the cards to be used in the current_play.

__repr__ and __str__:

As discussed earlier, repr gives display of representation of the object instance and str gives the list of all the objects(cards) and no. of cards which are to be used by the end user.

Testing Deck Class:

Creating a deck with 5suits consisting of 5 cards each:

```
def main():
    deck1=Deck(1,5,5)
    deck1.add_card()
    print(repr(deck1))
    print(deck1)
```

Output:

```
In [2]: runfile('C:/Users/Dell1/Desktop/learn_python/Assignement2/assignment2/deck.py',
wdir='C:/Users/Dell1/Desktop/learn_python/Assignement2/assignment2')
Reloaded modules: cards
Deck(1,5,5)
No. of Cards: 25
Cards in deck:
[Card(1,0), Card(2,0), Card(3,0), Card(4,0), Card(5,0), Card(1,1), Card(2,1), Card(3,1),
Card(4,1), Card(5,1), Card(1,2), Card(2,2), Card(3,2), Card(4,2), Card(5,2), Card(1,3),
Card(2,3), Card(3,3), Card(4,3), Card(5,3), Card(1,4), Card(2,4), Card(3,4), Card(4,4),
Card(5,4)]
```

Customized Deck created for the game Freecell:

```
def main():
    deck1=Deck(1,13,4)
    deck1.add_card()
    print(repr(deck1))
    print(deck1)
```

Output:

```
In [3]: runfile('C:/Users/Dell1/Desktop/learn_python/Assignement2/assignment2/deck.py',
wdir='C:/Users/Dell1/Desktop/learn_python/Assignement2/assignment2')
Reloaded modules: cards
Deck(1,13,4)
No. of Cards: 52
Cards in deck:
[Card(Ace,Diamond), Card(2,Diamond), Card(3,Diamond), Card(4,Diamond), Card(5,Diamond),
Card(6,Diamond), Card(7,Diamond), Card(8,Diamond), Card(9,Diamond), Card(10,Diamond),
Card(Jack,Diamond), Card(Queen,Diamond), Card(King,Diamond), Card(Ace,Club),
Card(2,Club), Card(3,Club), Card(4,Club), Card(5,Club), Card(6,Club), Card(7,Club),
Card(8,Club), Card(9,Club), Card(10,Club), Card(Jack,Club), Card(Queen,Club),
Card(King,Club), Card(Ace,Heart), Card(2,Heart), Card(3,Heart), Card(4,Heart),
Card(5,Heart), Card(6,Heart), Card(7,Heart), Card(8,Heart), Card(9,Heart),
Card(10,Heart), Card(Jack,Heart), Card(Queen,Heart), Card(King,Heart), Card(Ace,Spade),
Card(2,Spade), Card(3,Spade), Card(4,Spade), Card(5,Spade), Card(6,Spade),
Card(7,Spade), Card(8,Spade), Card(9,Spade), Card(10,Spade), Card(Jack,Spade),
Card(Queen,Spade), Card(King,Spade)]
```

NOTFREECELL

Implementation of the rules for playing the game are defined in this class.

`__init__`:

Parameters initialized when an instance of NotFreecell class is created.

- An instance of the Deck Class.
- Add_card method of the deck class to create a list of cards.
- Shuffle method of the deck class to shuffle the cards used in the deck.
- Empty freecells: A list of cells.
- Foundation: A nested list of four lists one for cards belonging to each suit. Ex: `[[],[],[],[]]`
- Cascades: A nested list consisting of 8 lists as there are 8 cascades in the tableau region. Cards are added into each cascades using deal_card method of the Deck class.

Result:

Cascade=`[[list1(7)],[list2(7)],[list3(7)],[list4(7)],[list5(6)],[list6(6)],[list7(6)],[list8(6)]]`

`__repr__` and `__str__`:

As discussed earlier, repr gives display of representation of the object instance and str gives how the game freecell should look. The first row is reserved for freecells and foundation units. Where freecell is placed on the left and foundation on the right hand side of the game play region.

def main():

```
    freecell=NotFreecell()
    print('repr')
    print(repr(freecell))
    print('str')
    print(freecell)
```

Output

```
In [2]: runfile('C:/Users/Dell1/Desktop/learn_python/Assignement2/assignment2/
freecell.py', wdir='C:/Users/Dell1/Desktop/learn_python/Assignement2/assignment2')

Do you want to play freecell?yes
repr
NotFreecell()
str

4:C 3:C 7:S Q:S 10:S 10:H K:H 10:D
3:D 8:C A:D 2:S 4:H J:D 8:H 10:C
6:D 3:S A:H 2:C 5:C 2:D 8:D Q:C
J:S K:C 3:H 6:H 9:C K:S 7:D 9:H
5:D 9:S A:C 7:C 9:D 7:H 4:S J:C
A:S 6:S Q:D K:D 6:C J:H 2:H 5:H
8:S 5:S Q:H 4:D
```


Method for playing game(game_play()):

A method name game_play is implemented to keep the play continuous as long as the the game is not finished or we have entered Quit Command through user input. When you enter the Quit command then the game comes out of the loop and quits the console.

Output:

```
In [1]: runfile('C:/Users/Dell1/Desktop/learn_python/Assigment2/assignment2/freecell.py', wdir='C:/Users/Dell1/Desktop/learn_python/Assigment2/assignment2')
```

```
Do you want to play freecell?yes
```

```
4:D 9:C 3:S 7:S Q:H 3:H 7:C Q:S
10:H 2:D J:C K:D 6:S 6:H A:H 7:H
J:D 3:D J:S J:H 5:S 5:C 10:D 7:D
4:S 2:H 8:C A:C 9:S 5:H 10:S 8:D
2:C Q:C 6:D K:C K:H 6:C A:D 4:H
A:S 9:D 10:C 8:S K:S 8:H 9:H 2:S
4:C Q:D 3:C 5:D
```

```
Enter your move: Move 5:D to Freecell
None
```

```
5:D
```

```
4:D 9:C 3:S 7:S Q:H 3:H 7:C Q:S
10:H 2:D J:C K:D 6:S 6:H A:H 7:H
J:D 3:D J:S J:H 5:S 5:C 10:D 7:D
4:S 2:H 8:C A:C 9:S 5:H 10:S 8:D
2:C Q:C 6:D K:C K:H 6:C A:D 4:H
```

Method for finding whether the game is finished (game_over()):

If all the cards are placed in the foundation then the game is finished and returns true value.

Logic:

```
[len(stack) for stack in self.foundation]==[13,13,13,13]
```

Method to get position of a card (get_card_position(card_txt)):

Finding the position of the card is one of the important aspect of this game as it decides whether the move to be made is a valid move or not. Since only last elements of the cascades can be moved freely, it is very much important to determine whether the card is the last element or not. The command which we enter is in string format and the elements of the cascade are built up of Card objects. Input parameter for this method is card in string format and the value returned by this method is Card object and its position. Cascade is made up of lists of list ex: List_a=[[1,2],[3,4,5]] list_a[1][1]==4 first it finds out the element at the index 1 which is also a list then it finds out a no at index value [1], hence 4 is returned, similar principle is used while locating a card in cascade.

Algorithm:

Position of the card is defined as (column_no,row_no) both col_no and row_no start from index value=0 index value of the Card --->row no. where the card is placed and iteration value of the cascade--->column_no. in which the card is placed."""

Testing the card_position:

Getting the card and position of 3 of Club in the Cascade.

```
def main():  
    freecell=NotFreecell()  
    print(freecell)  
    print(freecell.get_card_position('3:C'))
```

Output:

```
In [3]: runfile('C:/Users/Dell1/Desktop/learn_python/Assignement2/assignment2/  
Card_location_test.py', wdir='C:/Users/Dell1/Desktop/learn_python/Assignement2/  
assignment2')  
Reloaded modules: deck, cards  
  
6:S 3:C Q:C 4:C 3:S 5:D 8:S 10:H  
J:H J:S 2:S 7:H 3:H J:C 9:H 8:D  
J:D A:S 8:H 6:C 3:D K:H 2:C 6:H  
K:S 9:S 9:D 4:D 5:C 10:D 10:C 7:C  
A:C Q:S 8:C 10:S 4:S K:D 4:H K:C  
2:D 9:C A:D 6:D 2:H 7:D Q:H 5:H  
A:H 5:S 7:S Q:D  
  
(Card(3,Club), (1, 0))
```

CARD MOVEMENTS:

There are four card movements defined for the Freecell game:

1. Move (cardx) to Foundation (y).
2. Move (cardx) to Freecell.
3. Move (cardx) to Sidelane (cardy).
4. Move (cardx) to Emptylane(y).
5. Quit command.

Method to differentiate the commands(find_command(command)):

This method is basically used to differentiate the different commands passed in by the user and redirect the program flow to the respective command requested by the user. This method behaves like a parser which gets the command from the end user and tells the computer which is the next set of instruction to be executed.

Movement1: Quit command

When user enters this command the program stops its execution and exits.

Movement2: Move (cardx) to Freecell (move_to_freecell(card_txt))

Freecells has a maximum limit of four cards to be placed. It cannot place a card after it is full. So the algorithm to place a card into a freecell is as follows:

Algorithm:

1. Check whether the freecells are full.
2. Check if the card entered is the last card of the stack.
3. If not then the card cannot be moved and if yes then place the card in freecell provided the first condition is false.

Testing:

Case1: Moving card which is on the last position in stack.

Output:

```
IPython console
Console 2/A x
Do you want to play freecell?yes
|

4:D 9:C 3:S 7:S Q:H 3:H 7:C Q:S
10:H 2:D J:C K:D 6:S 6:H A:H 7:H
J:D 3:D J:S J:H 5:S 5:C 10:D 7:D
4:S 2:H 8:C A:C 9:S 5:H 10:S 8:D
2:C Q:C 6:D K:C K:H 6:C A:D 4:H
A:S 9:D 10:C 8:S K:S 8:H 9:H 2:S
4:C Q:D 3:C 5:D

Enter your move: Move 5:D to Freecell
None

5:D

4:D 9:C 3:S 7:S Q:H 3:H 7:C Q:S
10:H 2:D J:C K:D 6:S 6:H A:H 7:H
J:D 3:D J:S J:H 5:S 5:C 10:D 7:D
4:S 2:H 8:C A:C 9:S 5:H 10:S 8:D
2:C Q:C 6:D K:C K:H 6:C A:D 4:H
A:S 9:D 10:C 8:S K:S 8:H 9:H 2:S
4:C Q:D 3:C

Python console History log IPython console
```

Case2: Moving card which is not on the last position in stack.

```
IPython console
Console 2/A x
|
Enter your move: Move 10:C to Freecell
Card 10:C is blocked and cannot make the move
None

5:D

4:D 9:C 3:S 7:S Q:H 3:H 7:C Q:S
10:H 2:D J:C K:D 6:S 6:H A:H 7:H
J:D 3:D J:S J:H 5:S 5:C 10:D 7:D
4:S 2:H 8:C A:C 9:S 5:H 10:S 8:D
2:C Q:C 6:D K:C K:H 6:C A:D 4:H
A:S 9:D 10:C 8:S K:S 8:H 9:H 2:S
4:C Q:D 3:C

Enter your move:
```

Case3: Moving card after freecell is full.

```
IPython console
Console 2/A x
5:D 8:H 8:S Q:D

4:D 9:C 3:S 7:S Q:H 3:H 7:C Q:S
10:H 2:D J:C K:D 6:S 6:H A:H 7:H
J:D 3:D J:S J:H 5:S 5:C 10:D 7:D
4:S 2:H 8:C A:C 9:S 5:H 10:S 8:D
2:C Q:C 6:D K:C K:H 6:C A:D 4:H
A:S 9:D 10:C K:S 9:H 2:S
4:C 3:C

Enter your move: Move 3:C to Freecell
Cannot move the card to freecell
None

5:D 8:H 8:S Q:D

4:D 9:C 3:S 7:S Q:H 3:H 7:C Q:S
10:H 2:D J:C K:D 6:S 6:H A:H 7:H
J:D 3:D J:S J:H 5:S 5:C 10:D 7:D
4:S 2:H 8:C A:C 9:S 5:H 10:S 8:D
2:C Q:C 6:D K:C K:H 6:C A:D 4:H
A:S 9:D 10:C K:S 9:H 2:S
4:C 3:C

Python console History log IPython console
```

Movement3:Move (cardx) to Sidelane (cardy) (move_to_sidelane(card_txt_x,card_txt_y)):

Cardx can be placed on card y provided both the cards are of different colors and the difference in their card values=1. Algorithm developed is as follows:

Algorithm:

1. From the find_command will find out the original(from_card) and target card(to_card)
2. First find the location of the card (both from_card and to_card)and its card instance ex: 9:S represents Card(9,'Spade') object.
3. Check whether the original and target card are at the end of the stack and the difference in card values of the original and target card=1and that they are of different color.
4. If the above condition is true remove card from the first cascade and append it to the next target

Script:



Side_cascade_test.p
y

Test Case:

Testing without imposing color criteria for purpose of testing.

Case1: Placing 3:D on 4:D `freecell.move_to_sideline('3:D','4:D')`

```
IPython console
Console 3/A
assignment2')
Reloaded modules: deck, cards

K:S Q:S J:S 10:S 9:S 8:S 7:S 6:S
5:S 4:S 3:S 2:S A:S K:H Q:H J:H
10:H 9:H 8:H 7:H 6:H 5:H 4:H 3:H
2:H A:H K:C Q:C J:C 10:C 9:C 8:C
7:C 6:C 5:C 4:C 3:C 2:C A:C K:D
Q:D J:D 10:D 9:D 8:D 7:D 6:D 5:D
4:D 3:D 2:D A:D

K:S Q:S J:S 10:S 9:S 8:S 7:S 6:S
5:S 4:S 3:S 2:S A:S K:H Q:H J:H
10:H 9:H 8:H 7:H 6:H 5:H 4:H 3:H
2:H A:H K:C Q:C J:C 10:C 9:C 8:C
7:C 6:C 5:C 4:C 3:C 2:C A:C K:D
Q:D J:D 10:D 9:D 8:D 7:D 6:D 5:D
4:D 2:D A:D
3:D

In [12]:
Python console History log IPython console
```

Case2: Placing J:D on 3:D `(freecell.move_to_sideline('J:D','3:D'))`

```
IPython console
Console 3/A
Reloaded modules: deck, cards

K:S Q:S J:S 10:S 9:S 8:S 7:S 6:S
5:S 4:S 3:S 2:S A:S K:H Q:H J:H
10:H 9:H 8:H 7:H 6:H 5:H 4:H 3:H
2:H A:H K:C Q:C J:C 10:C 9:C 8:C
7:C 6:C 5:C 4:C 3:C 2:C A:C K:D
Q:D J:D 10:D 9:D 8:D 7:D 6:D 5:D
4:D 3:D 2:D A:D

Invalid Move!

K:S Q:S J:S 10:S 9:S 8:S 7:S 6:S
5:S 4:S 3:S 2:S A:S K:H Q:H J:H
10:H 9:H 8:H 7:H 6:H 5:H 4:H 3:H
2:H A:H K:C Q:C J:C 10:C 9:C 8:C
7:C 6:C 5:C 4:C 3:C 2:C A:C K:D
Q:D J:D 10:D 9:D 8:D 7:D 6:D 5:D
4:D 3:D 2:D A:D

In [13]:
Python console History log IPython console
```

Movement4: Move cardx to empty_lane Y (move_to_empty_lane(card, lane_no)):

Algorithm:

1. Get the card and its position from the get_card_position(card).
2. Find whether the lane entered is empty or not
3. If true then find whether the card is at the end position of the stack.
4. If both the conditions are true then move the card to empty lane.

Script:



Empty_Cascade_test
.py

Testing: Creating an empty lane to test(freecell.move_to_emptylane('4:D',8))

```
IPython console
Console 3/A

K:S Q:S J:S 10:S 9:S 8:S 7:S 6:S 4:D
5:S 4:S 3:S 2:S A:S K:H Q:H J:H
10:H 9:H 8:H 7:H 6:H 5:H 4:H 3:H
2:H A:H K:C Q:C J:C 10:C 9:C 8:C
7:C 6:C 5:C 4:C 3:C 2:C A:C K:D
Q:D J:D 10:D 9:D 8:D 7:D 6:D 5:D
3:D 2:D A:D

In [17]: |
```

Movement5: Move (cardx) to Foundation

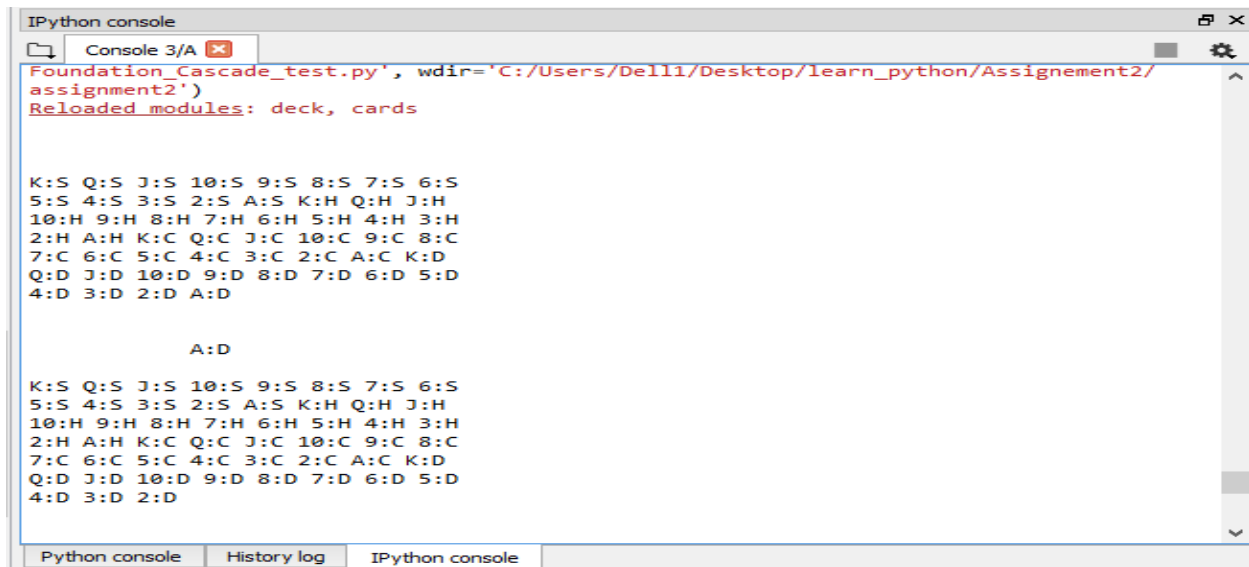
The cards which are to be placed in a foundation should be of same suit and should be placed in a sequential order.

Algorithm:

1. Get the position and the card .
2. Check whether the card is sorted. The very first card to be placed on the foundation is Ace.
3. If above two condition are true and if the card is at last position in the list then place it on the foundation.

Testing: Placing a card on the foundation

`freecell.move_to_foundation('A:D')`



The screenshot shows an IPython console window with a tab labeled 'Console 3/A'. The console output displays the state of a Freecell game. At the top, it says 'Foundation_Cascade_test.py', wdir='C:/Users/Dell/Desktop/learn_python/Assignment2/assignment2')' and 'Reloaded modules: deck, cards'. Below this, the state of the game is shown. The top row of cards is: K:S Q:S J:S 10:S 9:S 8:S 7:S 6:S. The second row is: 5:S 4:S 3:S 2:S A:S K:H Q:H J:H. The third row is: 10:H 9:H 8:H 7:H 6:H 5:H 4:H 3:H. The fourth row is: 2:H A:H K:C Q:C J:C 10:C 9:C 8:C. The fifth row is: 7:C 6:C 5:C 4:C 3:C 2:C A:C K:D. The sixth row is: Q:D J:D 10:D 9:D 8:D 7:D 6:D 5:D. The seventh row is: 4:D 3:D 2:D A:D. Below this, the state of the foundation is shown. The top row is: A:D. The second row is: K:S Q:S J:S 10:S 9:S 8:S 7:S 6:S. The third row is: 5:S 4:S 3:S 2:S A:S K:H Q:H J:H. The fourth row is: 10:H 9:H 8:H 7:H 6:H 5:H 4:H 3:H. The fifth row is: 2:H A:H K:C Q:C J:C 10:C 9:C 8:C. The sixth row is: 7:C 6:C 5:C 4:C 3:C 2:C A:C K:D. The seventh row is: Q:D J:D 10:D 9:D 8:D 7:D 6:D 5:D. The eighth row is: 4:D 3:D 2:D.

`freecell.move_to_foundation('A:D')`

`freecell.move_to_foundation('2:D')`



The screenshot shows an IPython console window with a tab labeled 'Console 3/A'. The console output displays the state of a Freecell game. The top row of cards is: 4:D 3:D 2:D A:D. Below this, the state of the game is shown. The top row is: A:D. The second row is: K:S Q:S J:S 10:S 9:S 8:S 7:S 6:S. The third row is: 5:S 4:S 3:S 2:S A:S K:H Q:H J:H. The fourth row is: 10:H 9:H 8:H 7:H 6:H 5:H 4:H 3:H. The fifth row is: 2:H A:H K:C Q:C J:C 10:C 9:C 8:C. The sixth row is: 7:C 6:C 5:C 4:C 3:C 2:C A:C K:D. The seventh row is: Q:D J:D 10:D 9:D 8:D 7:D 6:D 5:D. The eighth row is: 4:D 3:D 2:D. Below this, the state of the foundation is shown. The top row is: 2:D. The second row is: K:S Q:S J:S 10:S 9:S 8:S 7:S 6:S. The third row is: 5:S 4:S 3:S 2:S A:S K:H Q:H J:H. The fourth row is: 10:H 9:H 8:H 7:H 6:H 5:H 4:H 3:H. The fifth row is: 2:H A:H K:C Q:C J:C 10:C 9:C 8:C. The sixth row is: 7:C 6:C 5:C 4:C 3:C 2:C A:C K:D. The seventh row is: Q:D J:D 10:D 9:D 8:D 7:D 6:D 5:D. The eighth row is: 4:D 3:D.