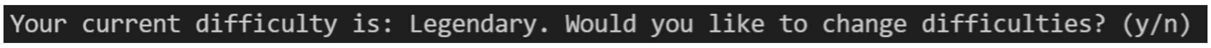
Additions to the base game

In my extension to the base game in Ass.py, I have added a set of settings that the user can adjust to customize their game for their own enjoyment. In these settings the user can adjust the number on the highest card in the deck, the number of moves they have before losing, the number of suits they are playing with and the symbols that represent those suits. The game also comes with preset difficulties: Easy, Medium, Hard, Legendary, and Custom, the last of which the user can adjust freely.

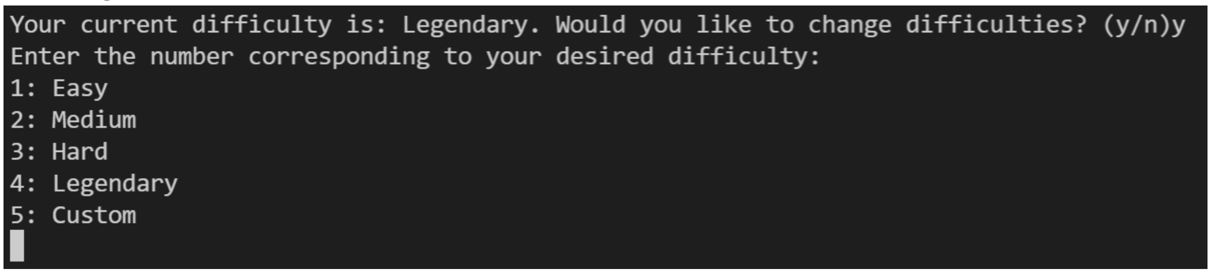
How to play

The user is first greeted with the following prompt:



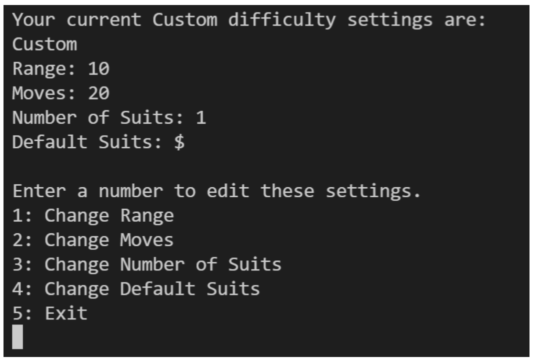
```
Your current difficulty is: Legendary. Would you like to change difficulties? (y/n)
```

Which offers them the opportunity to change their difficulty. If they enter y, they will see the following:



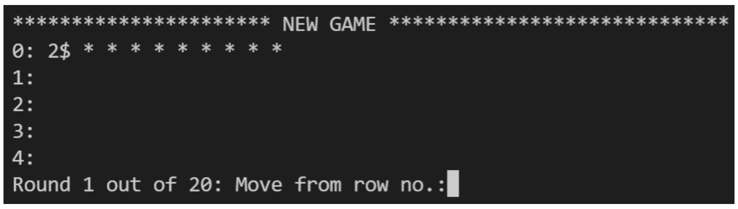
```
Your current difficulty is: Legendary. Would you like to change difficulties? (y/n)y
Enter the number corresponding to your desired difficulty:
1: Easy
2: Medium
3: Hard
4: Legendary
5: Custom
```

After any difficulty is selected, the first prompt is repeated to account for possible user error, or if the user has simply changed their mind. If the user enters n with the difficulty set to custom, they will be prompted to customize the settings to their liking, as below:



```
Your current Custom difficulty settings are:
Custom
Range: 10
Moves: 20
Number of Suits: 1
Default Suits: $

Enter a number to edit these settings.
1: Change Range
2: Change Moves
3: Change Number of Suits
4: Change Default Suits
5: Exit
```

The user can use this page to adjust any one of the four settings, or exit using 5. Once they exit, the game will begin, as shown:



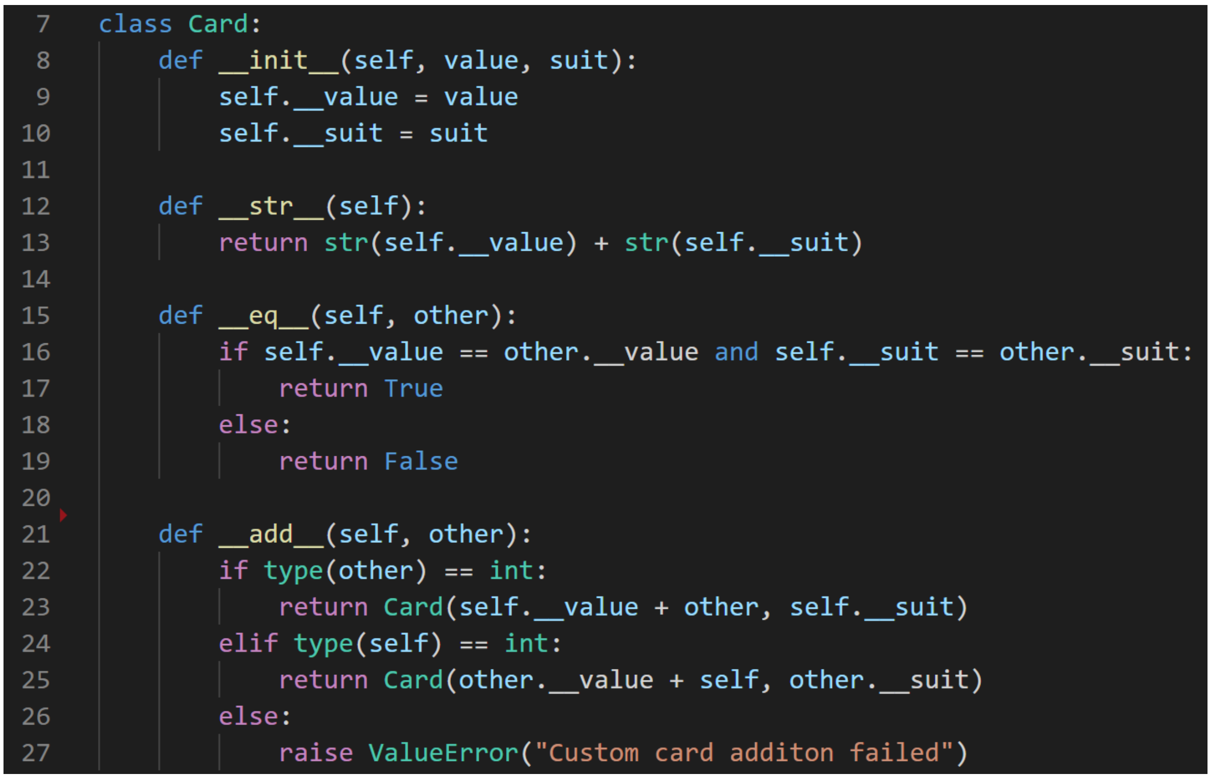
```
********************* NEW GAME ****************************
0: 2$ * * * * * * * * *
1:
2:
3:
4:
Round 1 out of 20: Move from row no.:
```

In this example we see the user has selected the symbol "$" to represent their suit.

How it works

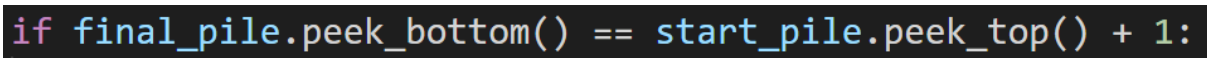
<u>Addition of Card class</u>

The first change I had to make was to create the Card class to implement the suits. The full code for the class is shown below, as it is rather compact:



```python
class Card:
    def __init__(self, value, suit):
        self.__value = value
        self.__suit = suit

    def __str__(self):
        return str(self.__value) + str(self.__suit)

    def __eq__(self, other):
        if self.__value == other.__value and self.__suit == other.__suit:
            return True
        else:
            return False

    def __add__(self, other):
        if type(other) == int:
            return Card(self.__value + other, self.__suit)
        elif type(self) == int:
            return Card(other.__value + self, other.__suit)
        else:
            raise ValueError("Custom card additon failed")
```

In this code we see that the string representation of the object has been adjusted for use in the CardPile.print_all() method, and both the = and + symbols have been overridden to take the suit of each card into account when moves are being made and their validity is being determined. For example, without taking suits into account the user could put a 2% card onto a 3# card, which is a violation of the intended rules. The + symbol has been overridden to allow addition of integers, this is because when a card is supposed to move onto another, I used the fact that the card that is stationary must be one greater than the card being moved to the right of it, and with the same suit. To simplify the code I overrode the + symbol so I could write this compactly as:



```python
if final_pile.peek_bottom() == start_pile.peek_top() + 1:
```

Within the move(self, p1, p2) method from the base game.

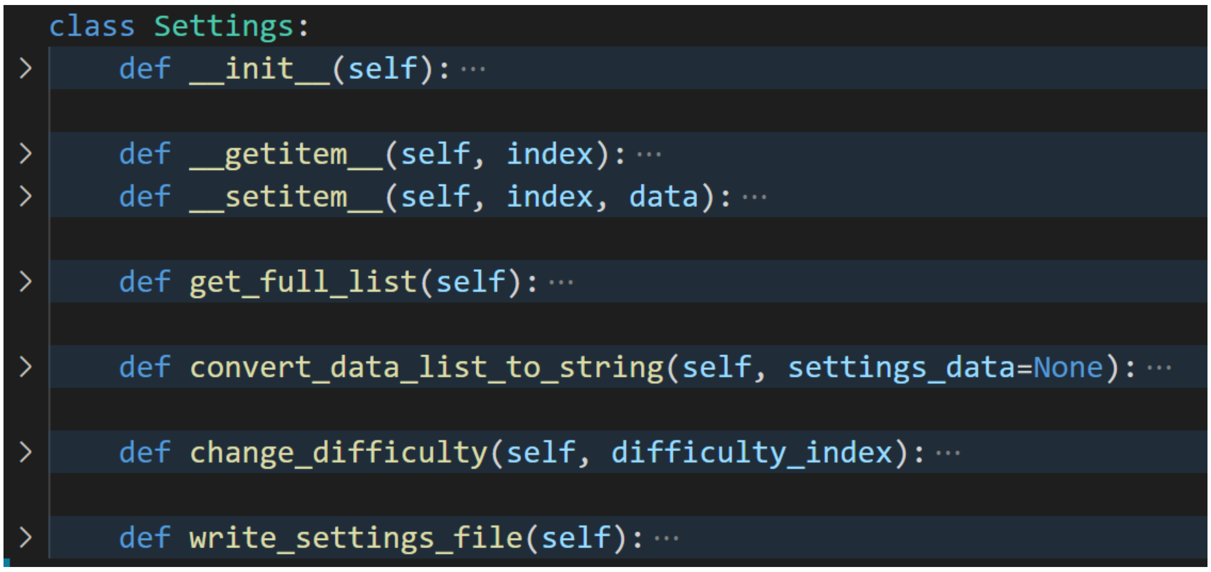
Changes to CardPile

I also created a new method for the CardPile method, called shuffle(self). This allows the shuffling of a card pile, which was necessary for the initialization of the Solitaire class, which generates a card pile with cards from 1 to the current maximum card value (chosen by difficulty) of each suit, then uses the shuffle method to randomize the game.

Addition of Settings class

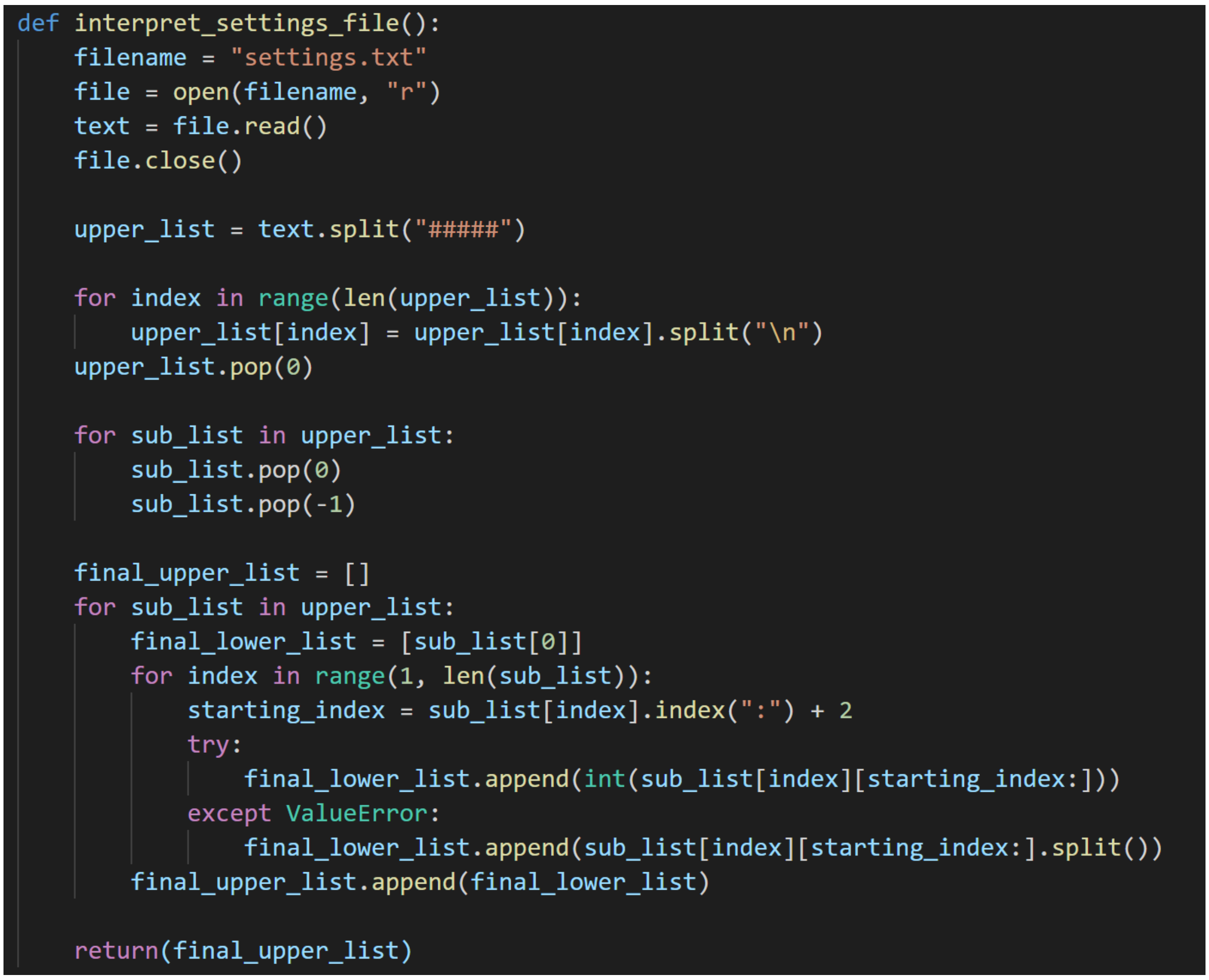
The bulk of the work on this project went into the creation of the Settings class, which allows the user to access settings from the file settings.txt, overwrite the file when their difficulty is changed, and allows the game to remember their previous difficulty using the settings.txt file.

The class Settings has the following methods:



```python
class Settings:
    def __init__(self): ...

    def __getitem__(self, index): ...
    def __setitem__(self, index, data): ...

    def get_full_list(self): ...

    def convert_data_list_to_string(self, settings_data=None): ...

    def change_difficulty(self, difficulty_index): ...

    def write_settings_file(self): ...
```

First, the Settings class intialises by using a supporting function:



```python
def interpret_settings_file():
    filename = "settings.txt"
    file = open(filename, "r")
    text = file.read()
    file.close()

    upper_list = text.split("#####")

    for index in range(len(upper_list)):
        upper_list[index] = upper_list[index].split("\n")
    upper_list.pop(0)

    for sub_list in upper_list:
        sub_list.pop(0)
        sub_list.pop(-1)

    final_upper_list = []
    for sub_list in upper_list:
        final_lower_list = [sub_list[0]]
        for index in range(1, len(sub_list)):
            starting_index = sub_list[index].index(":") + 2
            try:
                final_lower_list.append(int(sub_list[index][starting_index:]))
            except ValueError:
                final_lower_list.append(sub_list[index][starting_index:].split())
        final_upper_list.append(final_lower_list)

    return(final_upper_list)
```

The interpret_settings_file function simply reads the settings.txt file and extracts the data into a more usable list, containing the current difficulty, then the data for all of the other difficulty settings.