CSE 231 **Fall 2022**

Computer Project #10

Assignment Overview

This assignment focuses on the design, implementation and testing of a Python program which uses an instructor-supplied module to play a card game, as described below.

It is worth 55 points (5.5% of course grade) and must be **completed no later than 11:59 PM on Wednesday 11/30/2022**.
After the due date, 1 point will be deducted for every 1 hour late. No more submissions will be accepted after Thursday 12/01/2022.
If you submit by Tuesday, 11/29/2022, you will receive a 2-point bonus. Note that you need to click on the submit button to be considered as submitted.

Assignment Deliverables

The deliverable for this assignment is the following file:

`proj10.py` – the source code for your Python program

Be sure to use the specified file name and to submit it for grading via the **Coding Rooms system** before the project deadline.

Assignment Background

Klondike is one of the most popular solitaire card games. The game is played by one person with a standard 52-card deck of playing cards. The goal of the game is to build four foundations (one for each of the four suits), where all of the cards in each foundation are in order from Ace to King (with the Ace on the bottom).

Your program will allow the user to play a simplified version of Klondike, with the program managing the game. The game rules are given below.

The following website has an on-line version of Klondike (Turn One) and a short video introduction to the game:

http://worldofsolitaire.com/

Note that the game rules used for this project are simpler than the rules discussed on the website. Notably, the online game allows a multi-card pile to be moved between columns of the tableau—our project only allows one card at a time to be moved.

Game Rules

1. The deck of cards is shuffled and 28 cards are **dealt into 7 columns to form the tableau**. The cards are dealt in the following manner:

> **One card is placed in each of the 7 columns, from left to right**
> **A second card is placed in the rightmost 6 columns, from left to right**
> **A third card is placed in the rightmost 5 columns, from left to right**
> **A fourth card is placed in the rightmost 4 columns, from left to right**
> **A fifth card is placed in the rightmost 3 columns, from left to right**
> **A sixth card is placed in the rightmost 2 columns, from left to right**
> **A seventh card is placed in the rightmost column**

The **last card** placed in each column of the tableau is **turned face up**.

The **four foundations are initially empty**.
The **remaining 24 cards become the stock**. The **top card in the stock is turned over and placed face up in the waste pile** (also known as the talon).

2. Whenever an Ace is face up in the tableau or the waste (talon), it may be moved above the tableau and become the first card in that suit's foundation. After that, additional cards of that same suit may be moved into the suit's foundation: the Two on the Ace, the Three on the Two, and so on.

The objective of the game is to move all 13 cards of each suit into the appropriate foundation.

3. The top card in the talon (waste) may be moved into a foundation or into the tableau:

a) To be moved into one of the foundations, the card must be the correct suit and rank: it must be the same suit as the other cards in that foundation, and it must have a rank which is exactly one higher than the card that is currently at the top of the foundation (as above, the Two on the Ace, the Three on the Two, and so on).

b) To be moved into one of the columns in the tableau, the top card in the waste (talon) must be either a King (if the destination in the Tableau is empty), or the opposite color and exactly one rank lower than the card which is the last face-up card in that column. For example, a red Seven may be moved onto a black Eight. Hint: a red card will have suits as either 2 or 3. A black card will have suits either 1 or 4.

4. The last card in any of the 7 columns in the tableau may be moved into a foundation or into another column in the tableau:

a) To be moved into one of the foundations, the card must be the correct suit and rank (as above).

b) To be move elsewhere in the tableau, the card must be the opposite color and exactly one rank lower (as above).

5. If all of the cards are moved out of a particular column of the tableau, a King of any suit may be moved into that column. Note: a King is the only card which may be moved into an empty column.

6. At any point, the player may turn over the top card from the stock and place it face up in the talon (waste pile). When the stock becomes empty, the talon is turned over and becomes the stock.

Assignment Specifications

You will develop a program which allows the user to play Klondike according to the rules given above. The program will use the instructor-supplied **`cards.py`** module to model the cards and deck of cards.

1. The program will recognize the following commands:

`SW`	move one card from the stock to the waste (talon)
`WF N`	move one card from the waste to foundation N
`WT N`	move one card from the waste to column N in the tableau
`TF N1 N2`	move one card from column N1 of the tableau to foundation N2
`TT N1 N2`	move one card from column N1 to column N2 of the tableau
`H`	display the legal commands
`R`	restart the game (remember to shuffle)
`Q`	quit

Valid user inputs for foundation numbers range from 1 to 4, and valid user inputs for column numbers in the tableau range from 1 to 7. Note that the functions will accept as parameters the indices and not the column numbers (Hint: Python indices starts from 0 and not 1).

The program will repeatedly display the current state of the game and prompt the user to enter a command (until the user enters “q”).

The program will detect, report and recover from invalid commands. None of the data structures representing the foundations, tableau, stock or waste will be altered by an invalid command.

2. The program will use the following function to initialize the game:

`initialize() → tableau, stock, foundation, waste`

The function has no parameters and returns the starting state of the game with the four data structures initialized as described above. Check the Game rules section to see how to deal cards to each data structure. Remember that stock is of type Class `Deck`. Also, we need all tableau cards face down except last cards in each column. Cards are initially face up (as defined by the Card class) so turn all tableau cards face down. Then, turn last card in each tableau column face up (Hint: use the `flip_card()` method from the `Card` class). You will need to shuffle the stock when you create it. Use lists to create the structures.

3. The program will use the following function to move a card from the stock to the waste:

`stock_to_waste( stock, waste ) → bool`

That function has two parameters: the data structure representing the stock, and the data structure representing the waste. Hint: check that there are cards in the stock before trying to move one to the waste. Remember that stock is of type Class Deck. The function will return `True` if the move was done successfully. Otherwise, it returns `False`.

4. The program will use the following function to move a card from the waste to a foundation:

`waste_to_foundation( waste, foundation, f_num ) → bool`

That function has three parameters: the data structure representing the waste, the data structure representing the foundations, and a foundation number (the correct index in the foundation). The function will return **`True`** if the move is valid (**`False`** otherwise). If the move is valid, perform it.

5. The program will use the following function to move a card from the waste to the tableau:

`waste_to_tableau( waste, tableau, t_num ) → bool`

That function has three parameters: the data structure representing the waste, the data structure representing the tableau, and a column number (the correct index in the tableau). The function will return **`True`** if the move is valid (**`False`** otherwise). If the move is valid, perform it.

6. The program will use the following function to move a card from the tableau to a foundation:

`tableau_to_foundation( tableau, foundation, t_num, f_num ) → bool`

That function has four parameters: the data structure representing the tableau, the data structure representing the foundations, a column number, and a foundation number. The function will return **`True`** if the move is valid (**`False`** otherwise). Make sure to flip the new last card in the

tableau column from the source if the card is not face up (Hint: the method `is_face_up()` from the Card class returns `True` if card is facing up). If the move is valid, perform it.

7. The program will use the following function to move a card from one column in the tableau to another column:

`tableau_to_tableau( tableau, t_num1, t_num2 ) → bool`

That function has three parameters: the data structure representing the tableau, a source column number, and a destination column number. The function will return **`True`** if the move is valid (**`False`** otherwise). Make sure to flip the new last card in the tableau column from the source if the card is not face up (Hint: the method `is_face_up()` from the Card class returns True if card is facing up). If the move is valid, perform it.

8. **We provide a `parse_option` function** This function handles input and does error checking of input:

`parse_option(in_str) → list`

The function has one parameter: the raw input string. It returns a list which has the option specified followed by any arguments entered or returns **`None`**, if there is an error. The option returned must be upper case. All error checking of input is done in this function. Error statements are printed from this function (see specifications below). Note that if a list is returned, the list will be in the correct format with correct arguments—making your **`main`** function easier.

An example of a return (there are multiple variations):

```
return [option_str, source, destination, count]
```

where `option_str` is uppercase and the three arguments are correct integers (the column number and not the index in the list) Remember that column numbers start from 1 and not from 0.

Note that the function test we provide can check that **`None`** was returned for an error, but cannot check that a correct error message was printed—correctly printed error messages are only checked when the whole program is tested.

If an argument is an integer but is out of range,

```
print("Error in Source")
```

or

```
print("Error in Destination")
```

as appropriate.

If the `mTT` count is an integer but not positive,

```
print("Error in Count")
```

If any argument is not all digits (e.g. -1) and all other errors.

```
print("Error in option:", in_str)  # in_str is the raw input string
```
(The "not all digits" is an attempt to make some error checking easier for you.)

Remember that all errors return **`None`** after printing the error message.

9. This function checks if the game is in a winning configuration:

`check_win (stock, waste, foundation, tableau) → Boolean`

Returns **`True`** if the game is in a winning state: all cards are in the foundation, stock is empty, waste is empty and tableau is empty. Otherwise, return **`False`**.

Hint: you do not need to check that the order of the cards in the foundation is correct—you may assume that the functions that moved cards to the foundation handled that correctly.

10. We provide a display function that displays the current state of the game:

`display(tableau, stock, foundation, waste) → None`

11. Once you write all your functions, it is time to write your **`main`** function which will use the following function to manage the other functions:

`main() → None`

That function has no parameters; it controls overall execution of the program. You need to use the provided display () and parse_option() to handle inputs and display error messages and display the board.

a) Your program should start by initializing the board (tableau, stock, foundation, waste).
b) Display the MENU (given in the starter code)
c) Display the starting board (use the `display()` function).
d) Prompt for an option and check the validity of the input using the `parse_option` function.
e) If `'XX s d'`, move a card from source s to destination d depending on what is XX (XX could be TT or TF) :
    - If the move was a failure, you should print an error message: `"\nInvalid move!\n"`.
    - If a move was to the foundation and it was successful, check to see if the user won; if so print, `"You won!",` display the winning board, and stop the game. If the user did not win and the move was successful just display the board.
    - If a move was to tableau, just display the board.
f) If `'WX N'`, move a card from waste to a destination N depending on what is X (X could be T or F) :
    - If the move was a failure, you should print an error message:

`"\nInvalid move!\n"`.

- If a move was to the foundation and it was successful, check to see if the user won; if so print, `"You won!",` display the winning board, and stop the game. If the user did not win and the move was successful just display the board.
- If a move was to tableau, just display the board.

g) If `'Sw'`, move a card from Stock to Waste :
   - If a move is successful, display the board.
   - If the move was a failure, you should print an error message: `"\nInvalid move!\n"`.

h) If `'R'`, restart the game by initializing the board (after shuffling). Display the MENU and display the board.
i) If `'H'`, display the menu of choices
j) If `'Q'`, quit the game
k) If you get an error in option, error in source or error in destination (what will the `parse_option` function return in this case?), display the board and reprompt for another option.
l) The program should repeat until the user won or quit the game.

Assignment Notes

1. The coding standard for CSE 231 is posted on the course website:

   http://www.cse.msu.edu/~cse231/General/coding.standard.html

Items 1-9 of the Coding Standard will be enforced for this project.

2. Your program may not use any global variables. That is, you may not use any variables outside of functions.

3. Your program may not use any nested functions. That is, you may not nest the definition of a function inside another function definition.

4. Your program must contain the functions listed above; you may develop additional functions, as appropriate.

5. Sample output is shown on the following page.

TEST 1
See the files in the Play_Game folder

Function tests:
Check the folder Tests

Grading Rubric

Computer Project #10
Scoring Summary

General Requirements:
(4 pts) Coding Standard 1-9
(descriptive comments, mnemonic identifiers, format, etc…)

Implementations:

(6 pts) initialize
(4 pts) stock_to_waste
(6 pts) waste_to_foundation
(4 pts) waste_to_tableau
(6 pts) tableau_to_foundation
(6 pts) tableau_to_tableau
(4 pts) test flipping cards in the TF and TT functions
(4 pts) check_for_win
(9 pts) Test 1