Playing With Cards

This exercise will give you experience creating and manipulating instances of classes imported from an application-specific module, cards.py.

**Part (a):** This first part of the exercise just helps you become familiar with the cards.py module. We'll go through it together as a class.

The file paced-cardsDemo.py executes the program in cardsDemo.py (see handout) in between calling some “helper functions” imported from the library module exercise_helper.py. The helper functions allow pausing execution and showing the code from cardsDemo.py that will be executed next.

The program in cardsDemo.py imports cards.py, which defines Card and Deck classes similar to those in the cards.py module used in this week's lab and in an upcoming project. (But do *not* use this version of the module in your project, as it may not be identical to the one that will be used in grading the project.)

When instructed to do so, fill answers to what is printed at each of the pauses below.

```
pause(...) #1:

pause(...) #2:

pause(...) #3:

pause(...) #4:
```
pause(...) #5:

pause(...) #6:

pause(...) #7:

pause(...) #8:

pause(...) #9:
Part (b): In the rest of this exercise, you and a partner will write a program that plays a simplified version of the card game “Memory”.

To keep the game short and simple, we’ll use only 12 cards: the cards of rank 1 through 3 (i.e., A, 2, 3) of each suit. We’ll also assume 2 players.

Rules of the game: The 12 cards are laid face down in a 3 X 4 grid (3 rows, 4 columns). The object of the game is to find matching pairs (i.e., cards of the same rank), called tricks. A player turns over two cards. If the cards match, the player removes them from the grid and keeps the trick. If not, the player turns them back over. The players take turns until there are no cards left in the grid.

In the real game, a player’s turn keeps going until she fails to turn over a trick. For simplicity, in our game, the player’s turn ends after one try whether she earns a trick or not.

First, we’ll demo our game so you understand what your game should do. By convention, rows are numbered from 1 to 3, starting with the top row and moving downwards; and columns are numbered from 1 to 4, starting with the leftmost column and moving to the right.

Download cards.py and memoryGame.py. The former is the cards module that we imported in part (a). The latter embodies a design for one possible implementation of the memory game. Read over the docstrings and the comments of the stubbed functions. Read through the code below the function definitions. Ask any questions about anything that you don’t understand about the code and comments in memoryGame.py.

Implement the stubbed functions, testing them as you go. Then finish the code below the function definitions and play the game several times to test the game logic.

Finally, wrap the code that implements the game logic in a main function definition and add a call to main as the last line of the program.

The next page contains some implementation notes, which may be helpful.
Implementation notes:
As your first design decision, you need to decide how you will represent the grid. There are four reasonable alternatives.
1. As a (nested) list of 3 rows, where a row is represented as a list of 4 items and each item is a Card or 0.
2. As a (nested) list of 4 columns, where a column is represented as a list of 3 items and an item is a Card or 0.
3. As a (flat) list of 12 items, where an item is a Card or 0. Here, the first 4 items would make up the first row, the second 4 items would make up the second row, and the last 4 items would make up the third row.
4. As a dictionary where each key is a pair (two-tuple) of int and each value is either a Card or 0.

All of these representations use 0 to signify that one of the players has picked up the card at the associated position. (Why do you not want to just remove the card from the grid?)

This decision will affect how you want to represent grid positions. If you decide to use nested lists or a dictionary for the grid, it makes sense to represent a grid position as a 2-tuple, \((r, c)\), where \(r\) indicates a row and \(c\) indicates a column. Here, you would need to decide if \(r\) and \(c\) will be the row and column numbers that the user enters or, if using nested lists, if they will be the indices for the row and column in your grid. Either choice is fine, as long as you are consistent in doing it the same way throughout your program.

If you use a flat list for the grid, it makes sense to represent a grid position by the index into the grid, i.e., by an integer between 0 and 11, inclusive. The position (index) can be easily calculated from the row and column numbers, \(R\) and \(C\), that the user enters as \((R – 1) * 4 + (C – 1)\).

Be sure to document your design decisions as comments after the import clause and before the function definitions.

A strategy for creating the grid: Create and shuffle a deck. Initialize a variable to be an empty grid (list or dictionary). Then loop, repeatedly dealing a card and, if the card is one you want to keep, adding it to the grid. Exit the loop when the grid is full. Return the grid constructed in the loop.

Adding a card to the grid will be easy if you use the flat list representation for the grid—just append the card to the end of the grid.

Adding a card to the grid can be done in a nested loop if using any of the other representations. For instance, if you use a nested list of rows, you can collect 4 cards into a list in an inner loop and then append the list (representing a row) to the grid in the outer loop.