Playing With Cards

In the next few weeks, you will learn how to use classes to define new types. But first, this exercise illustrates the benefits of doing so. In it, you will use two application-specific types that we provide to simplify coding card games.

To start, we'll walk through execution of the program **cardsDemo.py**. This demo program imports the module **cards.py**, which defines the **Card** class and the **Deck** class. The classes define the representations of *Card* and *Deck* objects and the implementations of methods and operations for manipulating *Card* and *Deck* objects. The classes are designed so that a programmer can use a *Card* object to represent an actual playing card, a *Deck* object to represent an actual deck of playing cards, and their methods and operations to represent standard ways that card players manipulate cards and decks, such as shuffling a deck, dealing out cards, comparing cards to one another, and so on. The demo program serves to illustrate what you can do with *Card* and *Deck* objects.

We won’t execute **cardsDemo.py** directly; rather, we’ll execute another program that allows us to pause the execution and display what will be executed next. You should follow along on the handout of **cardsDemo.py**. When we reach a pause, discuss with your partner what will be printed. Put a pink tent on your monitor when you have a question and a green tent when you have an answer.

**Pause(...)** #1:

**Pause(...)** #2:

**Pause(...)** #3:

**Pause(...)** #4:

---

1 i.e., objects of type *Card* and objects of type *Deck*. 
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”.

The Memory game: The object of the game is to take the most tricks, where a trick consists of two cards of the same rank. The players shuffle and arrange the cards face down in a grid. The first 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. To keep the game short, use only cards of ranks 1 through 3 (i.e., A, 2, 3) of each suit. Arrange the cards in a 3 X 4 grid (3 rows, 4 columns). Assume only two players and that each player gets only one chance on each turn.

First, we’ll demo our game to clarify the description.

Instructions: Working with a partner, download cards.py and memoryGame.py. The former is the cards module that we imported in Part 1. The latter embodies a design for one possible implementation of the memory game. Read over and discuss the docstrings and the comments in memoryGame.py. Read through and discuss the code below the function definitions. Run the partially implemented program. Be sure you understand the design and the given code before moving on. Put up your pink tent if you have questions.

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.

Part (c): In the real Memory game, a player’s turn keeps going as long as she keeps winning tricks. Modify your program so the player’s turn does not end after earning a trick, but continues until she guesses either an illegal selection or a pair of non-matching grid positions.
Implementation notes:
Your first design decision should be how you will represent the grid. There are at least 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. Question: Why replace the card with something instead of just deleting the card from the list(s) representing 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 they will be the indices for the row and column in a list. 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. This index can be easily calculated from the row and column numbers, \(R\) and \(C\), that the user enters as \((R - 1) * 4 + (C - 1)\). Question: Explain this formula.

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 so constructed.

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 with nested loops if you use any of the other representations. For instance, for each row, the inner loop can collect cards into a list and, when a list of 4 cards is produced, the outer loop can append the list (representing a row) to the grid.