Programming Problems

For each of these problems (unless otherwise specified), write the needed code directly in the Python file, in the corresponding function definition.

All programming problems may also be checked by running 'Run File As Script' on the starter file, which calls the function `testAll()` to run test cases on all programs.

In this assignment, you will write six functions and these functions will work together to create an interactive, text-based game of **Memory**. In the game Memory you start with a deck of matched cards. Place the cards face-down on the table. In each turn you get to flip over two face-down cards. If they match, they stay face-up; if they don't, you have to flip them face-down again. The goal is to flip all the cards face-up in as few turns as possible.

If you've never played Memory before, try it out here:
https://www.helpfulgames.com/subjects/brain-training/memory.html

Note: #1-#3 can be solved with just the skills you've learned in prior units; we encourage you to write these early. #4-#6 will be covered in the Managing Large Code Projects lecture.

#1 - `generateBoard(words)` - 3pts

First, we need to set up the initial board of cards. Each card is represented by a two-item list: the first item is the card's value (a string), the second is a Boolean representing whether the card is face-down or face-up (`False` for down, `True` for up).

Write the function `generateBoard(words)` which takes a list of strings and returns a 2D list board (a list holding cards, which are 1D lists). The function should create two cards for each of the words in the list, both with initial flipped values of `False` (since we want all the cards in our game to begin face-down).

Once all cards have been added to the board, shuffle the board so the cards are in a random order. Recall that there's a built-in destructive function that can do this - `random.shuffle`.

For example, if we call `generateBoard(["dog", "cat"])`, it might return `[ ["dog", False], ["cat", False], ["cat", False], ["dog", False] ]`, though the inner lists could also be in a different random order.

Important Note: make sure that when you're generating the second card of a pair, it is not **aliased** to the first card! That can lead to problems later on.

#2 - `displayBoard(board)` - 4pts

Next, we need to be able to display the board to the user. We'll display each card as its index followed by its value if it is face-up, or the string `"???"` if it is face-down.

Write the function `displayBoard(board)` which takes a board (a 2D list containing cards as described above in `generateBoard`) and prints the board, returning `None`. Each card on the board will be printed on a separate line. The line should contain the index of the card and the card's value (or `???` if it is face-down). You can format the printed line however you like as long as those two pieces of information are included and each card is printed on a separate line.

For example, if we call `displayBoard([ ["dog", False], ["cat", True], ["cat", False], ["dog", True] ])`, (the list above, but with the first `"cat"` and second `"dog"` face-up), it might print:

```
0: ???
1: cat
2: ???
3: dog
```

#3 - `getFlippedCards(board)` - 3pts

We'll need to determine which cards on the board have been flipped to tell when the game is over. Write the function `getFlippedCards(board)` which takes the board (a 2D list containing cards as described above) and returns a 1D list of integers. This function should go through each card on the board and **if the card is face-up** add the **index** this card has in the board into a result list. The result list of indexes should be returned at the end.

For example, if we call `getFlippedCards([ ["dog", False], ["cat", True], ["cat", False], ["dog", True] ])` it will return `[ 1, 3 ]`, because the first and third elements are face-up. If no cards are face-up, just return an empty list.

#4 - `loadWords(filename)` - 10pts

Instead of providing the words for the game in a list, we'll provide the words for the game in a **file**. That file will include all the words that should be used in the game, separated by commas. For example, a file containing the text

```
owl,dog,cat
```

would be used to create a game with a six-item board - two owl cards, two dog cards, and two cat cards.

Write the function `loadWords(filename)` which takes the name of a file (a string), reads the text from that file, and returns a list of the words that occurred in that file. For example, if the file `memory2.txt` contains the text shown above, `loadWords("memory2.txt")` would return the list `[ "owl", "dog", "cat" ]`.

Important Note: to test this function, you'll need to download the files **memory1.txt**, **memory2.txt**, and **memory3.txt** from the course website and place them in the same folder as **hw5.py**. But you don't need to include the files in your submission; we'll have them in the correct place in Gradescope already.

Another note: if your code is fine but Python can't find the files, and you're sure the files are in the right place, come to office hours! We can help you get your file system set up correctly (which will be important, as many of the Hw6 projects include file reading).

#5 - `pickIndex(board)` - 10pts

To make the game interactive we'll let the user choose which cards to flip over on each turn by typing in the indexes of the cards they want to flip into the interpreter. We'll have to make sure to handle bad user inputs without crashing the program!

Write the function `pickIndex(board)` which takes a board (a 2D list containing cards as described above) and returns the index of the card to be flipped (an int). The function should ask the user to pick a card index. Assume that the user will always enter an integer index. If this index is not in the range of the board, or is the index of a card that has already been flipped, the function should print out an appropriate error message and ask the user for an index again. When the user enters a valid index, that index should be returned as an integer.

For example, if we call `pickIndex([ ["owl", False], ["dog", True], ["cat", False], ["cat", False], ["owl", False], ["dog", True] ])` we might end up with the following interaction. User input is bolded.

```
Pick a card index: -1
Out of range. Pick a valid card.
Pick a card index: 6
Out of range. Pick a valid card.
Pick a card index: 1
You've already flipped that card. Pick a different card.
Pick a card index: 2
```

The function would then return the integer `2`. Your prompts and error messages don't have to be the same as ours, but they should exist and provide relevant information.

We will manually grade this problem. To test it yourself, run `testPickIndex()` and try entering numbers out of range, and indexes that have already been flipped, as shown above. Again, assume that the input will always be an integer value.

Hint: use the board to help determine what is in range and whether the chosen index has already been flipped.
Hint 2: if you're stuck, look at our tic-tac-toe example from the Managing Large Code Projects lecture. How did we implement taking turns?

#6 - `playMemoryGame(filename)` - 10pts

Now we finally have everything we need to actually implement the game! You must use the five functions written above as **helper functions** to streamline the game code.

Write the function `playMemoryGame(filename)` which takes the name of a file that holds the words to be used in the game. The function should read the words into a list, set up a memory game based on those words, then run the game with the user until all cards have been flipped. Here's a high-level algorithm for how `playMemoryGame` works:

A. First, generate a new board by reading the words from the given file and creating a starter board based on them. Call `loadWords` and `generateBoard` for this.

B. Second, welcome the player to the game and show the starting board to the user. You'll want to call `displayBoard` for this.

C. Third, set up a loop that continues until the game is finished. The game is done when every card in the board has been flipped. To check for this, test whether the result of calling getFlippedCards on the board contains all the cards or not.

D. Fourth, in each iteration of the loop, have the user pick indexes for two cards that they want to flip face-up. After a valid index is picked, the card at that index on the board should have its flipped value set to `True` and the board should be displayed again. You'll need to do this **twice**- once for the first card and again for the second. You'll want to call `pickIndex` and `displayBoard` here (and note that you'll need to call them more than once!).

E. Fifth, check whether the two cards that were flipped match by comparing their values. If they match, leave them face-up (don't change anything); if they don't match, set both flipped values to face-down (`False`) again. In either case, print out an appropriate message to the user, then print some kind of line break to show that the next turn has begun.

F. Finally, keep track of the number of turns the user takes to clear the board. A turn is a single iteration of the loop (flipping two cards and seeing if they're a match). You should print out a congratulations message that includes the number of moves taken at the end of the game.

Once this function is finished, you're done! Play your game to test out how it works. For example, if we call `playMemoryGame("memory1.txt")`, where the file `memory1.txt` contains the text:

`dog,cat`

The following text shows what a full game might look like, with user inputs bolded. (Note that the text continues onto the next page). Your game prompts don't need to be identical to ours, but they should convey the same information.

```
Welcome to Memory!
0: ???
1: ???
2: ???
3: ???
Pick a card index: 0
0: dog
1: ???
2: ???
3: ???
Pick a card index: 2
0: dog
1: ???
2: cat
3: ???
Try again!
---
Pick a card index: 1
0: ???
1: cat
2: ???
3: ???
Pick a card index: 2
0: ???
1: cat
2: cat
3: ???
Good guess!
---
```

```
Pick a card index: 0
0: dog
1: cat
2: cat
3: ???
Pick a card index: 2
You've already flipped that card. Pick a different card.
Pick a card index: 3
0: dog
1: cat
2: cat
3: dog
Good guess!
---
Good game! You took 3 moves to clear the board.
```