CPSC 217 Assignment 3

Due: Friday November 24, 2017 at 11:55pm
Weight: 7%
Sample Solution Length: Less than 100 lines, including blank lines and some comments (not including the provided code)

Individual Work:

All assignments in this course are to be completed individually. Students are advised to read the guidelines for avoiding plagiarism located on the course website. Students are also advised that electronic tools may be used to detect plagiarism.

Late Penalty:

Late assignments will not be accepted.

Submission Instructions:

Your program must be submitted electronically to the Assignment 3 drop box in D2L. You don't need to submit `SimpleGraphics.py` or `sprites.gif` because we already have them (but it won't hurt anything if you include them).

Description

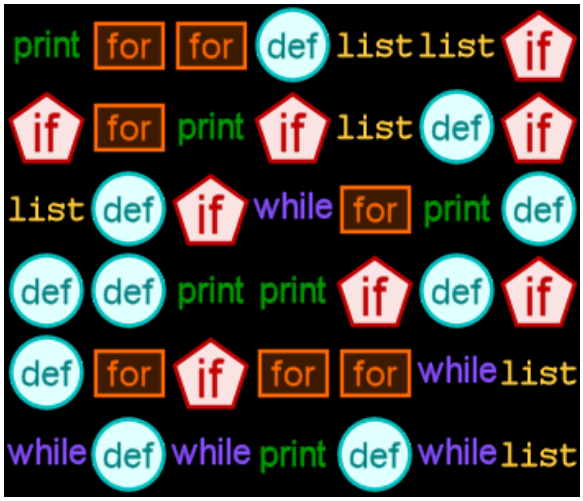
Bejeweled and Candy Crush are examples of match-three games. The goal in such games is to swap adjacent game pieces to form horizontal or vertical lines consisting of three or more identical pieces. Some swap three games also include additional game mechanics such as power-ups that clear multiple pieces from the game board, specific goals that must be fulfilled to pass a level, and time constraints, among others.

In this assignment you will add functionality to codeCrusher – a computer programming themed match-three game. I have written all of the user interface code (graphics and mouse) for the game, as well as some of the game logic, and I have created all of the necessary art assets. Your task is to write the functions that implement the remaining aspects of the game logic, resulting in a fully functional game. These functions are described in the following sections. **Note that you must follow the implementation instructions exactly. If your function has a different name, takes a different number of parameters, or returns a different value than expected then my code will not be able to call it successfully, and the game will not work.**

Part 1: Creating the Board

We will use a two dimensional list to represent the game board. Each element in the list will be an integer that indicates what type of game piece currently resides at that location. The integers 0 though to (and including) 5 are used to denote standard game pieces bearing the labels 'print', 'if', 'while', 'for', 'def' and 'list'. Empty spaces are denoted by -1 while the special game piece that clears all of the game

pieces of a particular type is denoted by 6. A game board is shown below, along with the list that represents it.




```
[[0, 3, 3, 4, 5, 5, 1],
 [1, 3, 0, 1, 5, 4, 1],
 [5, 4, 1, 2, 3, 0, 4],
 [4, 4, 0, 0, 1, 4, 1],
 [4, 3, 1, 3, 3, 2, 5],
 [2, 4, 2, 0, 4, 2, 5]]
```

Create a function named `createBoard` (notice the use of a lowercase c and an uppercase B). The function will take 3 integer parameters: the number of rows in the board, the number of columns in the board, and the number of unique game pieces that can appear on the board. Your function must return a two-dimensional list with the indicated number of rows and columns. Every value in the two-dimensional list must be a random integer r where r is greater than or equal to 0 and less than the number of unique symbols provided as a parameter to the function. The code that I have provided has already imported the randrange function, which you can use to generate a random integer within a range.

The game board above was construct by calling `createBoard` with parameters indicating 6 rows, 7 columns and 6 unique symbols. Because `createBoard` returns a random result, you will likely see different integers when you call `createBoard` with those same parameters, but the structure of the two-dimensional list you create should match what is shown above.

Run the codeCrusher game after implementing this function. My automated tests will give you feedback on whether or not you have this function working. Do not proceed to Part 2 until this function passes all of my tests. My implementation of `createBoard` is about 7 lines of code without any comments or blank lines.

Part 2: Swapping Game Pieces

Once you have implemented `createBoard` successfully you can attempt to play a game, and the game board should display successfully. Clicking on a game piece, and then clicking on one of its neighbours (either above, below, left or right) will attempt to swap the pieces. While an animation will show the pieces moving to their new locations, they will immediately jump back to their original positions. This behaviour is occurring because the body of the swap function currently only contains `pass`, which is a Python reserved word that does nothing. (This reserved word is included in Python because function, if

statement, and loop bodies cannot be empty so `pass` is placed in the body when we don't want the body to do anything but can't leave it empty).

Your next task is to provide a correct implementation for `swap`. The `swap` function takes 5 parameters: the game board, followed by the row and column for a game piece (r1 and c1), and the row and column for a second game piece (r2 and c2). Your function should modify the game board so that the pieces at (r1, c1) and (r2, c2) are swapped. Once your swap function is implemented correctly it will be possible for you to play the game by swapping adjacent pieces. In fact, the game should be rather easy because any adjacent pieces can be swapped, whether they form a line or not. In Step 4 we'll restrict the swaps to moves that form a line of at least three identical symbols.

My implementation of `swap` is only three lines long (without any comments or blank lines). Note that the swap function does not return a result – it modifies the board that is passed to it as a parameter.

Part 3: The Special Piece

When a straight line of five pieces is formed, or an L or a T involving five identical pieces is formed, the pieces are removed from the game board and a special piece is inserted in the location where the swap was performed. The special game piece is denoted by the following symbol:

This piece can be swapped with any adjacent piece (except for another special piece). When the swap is performed, the special piece is removed from the board, along with all occurrences of the other piece with which it was swapped. This is a powerful piece because it simplifies the board and makes larger matches more probable. The player is also awarded 1,000 points when the piece is initially created which is very helpful for reaching the target score.

Replace the existing implementation of `clearAll` with a new implementation that replaces all occurrences of a symbol with empty spaces. Your function will take a two-dimensional list that represents the board as the first parameter, and it will take the symbol that should be cleared as its section parameter. Your function will not return a result. Instead, it will modify the board passed as the first parameter. Recall that empty spaces are represented by -1, or better yet, use the EMPTY constant that is already defined in the starter code.

My implementation of `clearAll` is 4 lines long (without any comments or blank lines).

Part 4: Limiting the Swaps

The game becomes more interesting (and challenging) when swaps can only be performed when they form a straight line of (at least) three identical symbols. To help us implement this restriction we'll begin by creating two functions: `vLineAt` and `hLineAt`. Each of these functions takes the game board as its first parameter and a row and column representing a location on the board as its second and third parameters.

The vLineAt function returns `True` if there is a vertical line of three identical symbols that include the row and column indicated. There are three possible ways that this can occur: the indicated location is the top piece in the line, the indicate location is the middle piece in the line, or the indicated piece is the bottom piece in the line. Care must be taken to ensure that your function only examines locations that are within the board. Your function should return `False` when there is not a vertical line that includes the row and column indicated.

The `hLineAt` function behaves similarly to `vLineAt`, except that it checks for a horizontal line instead of a vertical line. As a result, the row and column indicated can be the left piece in a line, the middle piece in a line or the right piece in a line. Like `vLineAt`, the `hLineAt` function should return `False` when there is not a horizontal line that includes the row and column indicated.

The provided implementation of `canSwap` always returns `True`, permitting any pair of adjacent pieces to be swapped. Once you have implemented `vLineAt` and `hLineAt`, you should update `canSwap` so that it only allows a swap to occur when the swap forms a line of (at least) three identical symbols. The first parameter to `canSwap` is the board while the second and third parameters are the row and column of a piece on the board, and the fourth and fifth parameters the row and column for a second piece on the board. The `canSwap` function should only return `True` when swapping the indicated pieces results in a line. Otherwise it should return `False`. Note that the line that is created could include (r1, c1) or it could include (r2, c2) – both pieces involved in the swap must be checked.

Note that canSwap is only supposed to determine whether or not a swap can be performed. It isn't actually supposed to swap the pieces. As a result, if you change the board as part of determining whether or not a swap is possible, you need to change it back before returning from the function. While you are free to use any algorithm to implement `canSwap`, I used the algorithm shown below and believe that it is the easiest way to implement `canSwap` correctly.

```
Swap the pieces
If there is a horizontal line of 3 pieces involving the first location or
   there is a horizontal line of 3 pieces involving the second location or
   there is a vertical line of 3 pieces involving the first location or
   there is a vertical line of 3 pieces involving the second location
      Swap the pieces back
      Return True
Otherwise
      Swap the pieces back
      Return False
```

Your implementation of `canSwap` should call the `vLineAt` and `hLineAt` functions that you wrote earlier in this part of the assignment. My implementations of `vLineAt` and `hLineAt` are each approximately 10 lines of code (without any comments) while my implementation of `canSwap` is slightly shorter.

Part 5: Giving the Player a Hint

Sometimes the player needs a little bit of help to find a legal move on the board. Sometimes no legal move exists. The `hint` function that you will update in this part of the assignment will identify a legal swap, or it will indicate that no such swap is possible. A swap is legal if the two pieces are both in the same column in adjacent rows, or the pieces are both in the same row in adjacent columns and performing the swap forms a line of (at least) three identical symbols

The `hint` function takes the game board as its only parameter. It will return four integer values that represent the row and column of a piece, and the row and column of a second adjacent (either horizontally or vertically) piece, that can be swapped to form a line. If no such swap is possible then the function will return -1, -1, -1, -1 to indicate such.

You can test your hint function by pressing 'h' on the keyboard when playing a game. My user interface will highlight the indicated locations on the board, bringing them to the attention of the player. A message is displayed near the bottom of the window when no swap is possible. When no swap is possible the game board can be reset by pressing 'r'.

My implementation of `hint` is approximately 10 lines of code (without any blank lines or comments).

Additional Requirements:

- You must **not** modify any of the code that I have provided except for:
    - Adding your name and student number to the top of the file (and updating the comment if you feel inclined to do so)
    - Adding the `createBoard`, `vLineAt` and `hLineAt` functions
    - Updating the bodies of `swap`, `canSwap`, `hint`, and `clearAll`
- You may assume that the functions that are writing will be called with "reasonable" values. For example, it is *not* necessary to protect against scenarios such as createBoard being called with a negative number of columns.
- All lines of code that you write must be inside functions (except for the function definitions themselves and any constant definitions).
- You must create and use the functions described previously in this document.
- Do **not** define one function inside of another function.
- You must make appropriate use of loops. In particular, your program should work for game boards of various sizes. Various game board sizes can be tested by choosing different difficulty levels when starting the game.
- Include appropriate comments for each of your functions. All of your functions should begin with a comment that briefly describes the purpose of the function, along with a description of every parameter and every return value. Functions that do not return a value should be explicitly marked as such.
- Your program must **not** use global variables (except for constant values that are never changed, and in this assignment you may not even want any global constants beyond the ones that I have already defined).

- Your program must use good programming style. This includes things like appropriate variable names, good comments, minimizing the use of magic numbers, etc. Your program should begin with a comment that includes your name, student number, and a brief description of the program.
- Break and continue are generally considered bad form. As a result, you are **NOT** allowed to use them when creating your solution to this assignment. In general, their use can be avoided by using a combination of if statements and writing better conditions on your while loops.

Hints:

- In some of the functions you will need to know the number of rows and/or the number of columns in the board. These values can be determined by using the built-in `len` function. In particular, the number of rows in the board is `len(board)` and the number of columns in the board is `len(board[0])`.
- While my automated tests are reasonably thorough, they don't consider every possible case. You need to implement functions that provide the functionality described in this document, not just functions that pass the collection of provided tests.

For an A+:

The specification for the `hint` function above only requires that it return a some legal move (if such a move exists). It doesn't require `hint` to determine how many different legal moves there are, or to attempt to return the best possible move.

Improve the hint function so that it returns the best legal move, where any move that forms a line of 5, a T or an L (which will result in a special piece being placed on the board) is better than any move that forms a line of 4, which is better than any move that forms a line of 3.

If you complete the A+ portion of the assignment, please include a clear, highly conspicuous note indicating such at the top of your submission so that your TA knows to grade it.

Grading:

This assignment will be graded on a combination of functionality and style. A base grade will be determined from the general level of functionality of the program (Does it create the board correctly? Does it correctly implement swap? Does it restrict swaps to moves that form lines? Does it provide correct hints? Etc.). The base grade will be recorded as a mark out of 12.

Style will be graded on a subtractive scale from 0 to -3. For example, an assignment which receives a base grade of 12 (A), but has several stylistic problems (such as magic numbers, missing comments, etc.) resulting in a -2 adjustment will receive an overall grade of 10 (B+). Fractional marks will be rounded to the closest integer.

Total Score (Out of 12)	Letter Grade
12	A
11	A-

10	B+
9	B
8	B-
7	C+
6	C
5	C-
4	D+
3	D
0-2	F