COMP 120 - Problem Solving Assignment 5

Due: Thursday, May 7 @ 10:00PM

Required Background:

- No additional background required

Assignment Overview:

For this assignment you will write a GUI, event-driven program that plays a game called greedy snake. In the game, a snake, controlled by the user, moves around a grid, eating food that is randomly placed within the grid. The game player controls the snake using the four keyboard arrow keys. The game ends when the head of the snake runs into the body of the snake or runs off the board if the game is not in wraparound mode. A point is earned for each piece of food eaten.

Total number of points for the assignment is 100.

Initial Setup

1. Both you and your partner will need to get the starter code for your group using Git. Remember that you have a new partner starting with this assignment. Your new group number is on Blackboard, and in "PSAs -> PSA Group Numbers", in the "PSA group numbers 2.pdf" file.

2. In VS Code, open the command palette and select the "Git Clone" option.

3. When prompted for the repository URL, enter the following, with X replaced by your group number (e.g. 7 or 12).

   ssh://git@code.sandiego.edu/comp120-sp20-psa5-groupX

4. Choose the "Open Repository" option in the window that pops up in the lower-right corner of the screen. The repository should contain the files:
   - `lifeIteration1.py1`, `lifeIteration2.py`, ..., `lifeIteration6.py`, which contain source code for the six iterations of the game of life program, with version 6 being the final version.
   - `snake.py`, in which you will write your solution to the greedy snake program.

Also, remember when VS Code closes your repository when you exit and restart it. Use the "Open Recent" option (in the "File" menu) to reopen it if you can't find it.

We also recommend that you stage changes, commit those changes, and sync the changes every time you finish one of the functions you write. This ensures that you won't lose any of your work in case your computer gets lost or a file gets accidentally deleted.

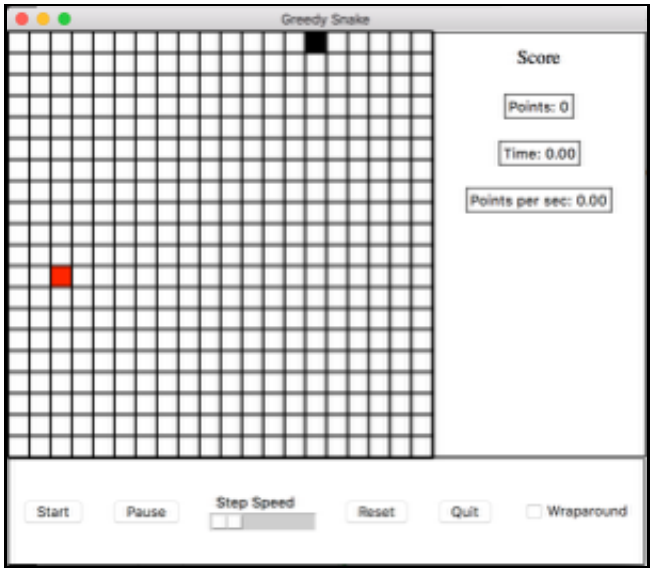
Problem

In a the file named `snake.py`, write a GUI, event-driven program that plays a version (described below) of the classic video game Greedy Snake. In the game, a snake, controlled by the user using the four keyboard arrow keys, moves around a grid, eating food that is randomly placed within the grid.

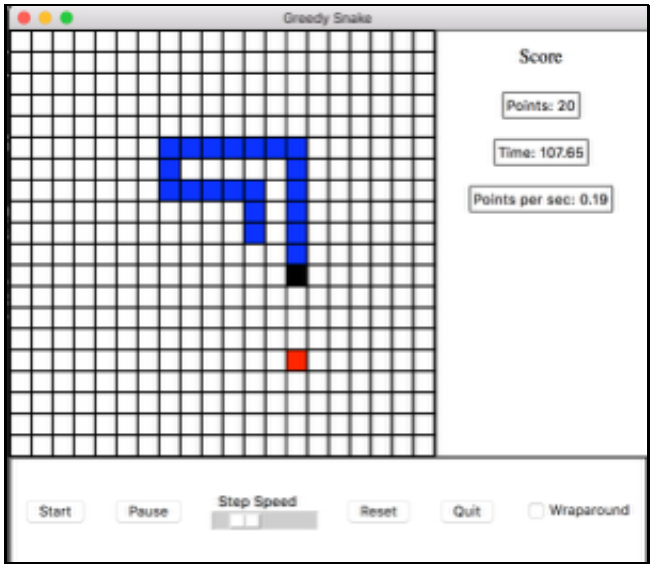
Initially, the snake is one square long, and its length increases by 1 with each piece of food that it eats. As the snake moves around the grid, each segment of the snake must follow the same path that the head of the snake takes. The game ends when the head of the snake runs into the body of the snake or runs off the board if the game is not in wraparound mode. If the game is in wraparound mode, and (for example) if the snake goes off the right side of the grid, it appears on the left side of the grid in the same row. The speed that the snake moves can be adjusted. A point is earned for each piece of food eaten, but the real measure of the player's skill is the points per second that are earned.

Here are details on how your program should look and function: (Be sure to read this entire document before starting to code, as following the instructions I give you extended guidance and rules on how to go about developing your program.)

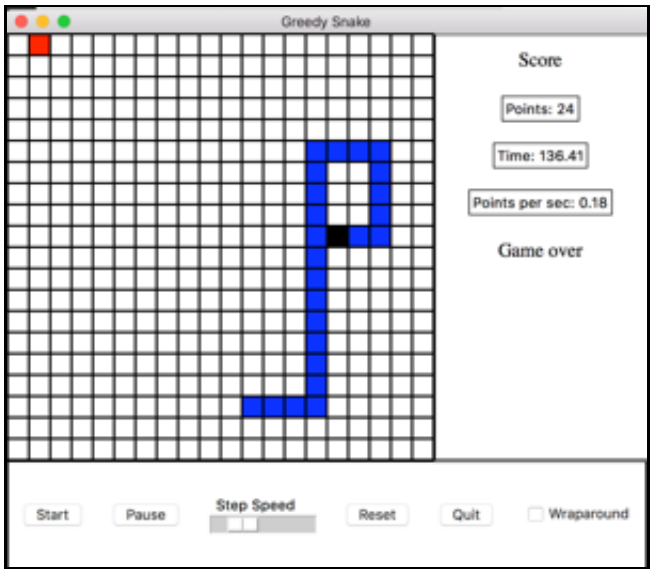
Here is picture showing how the window of your program should look at the start of the game (before the user interacts with any of the widgets):



And here is a picture showing the window after the game has been played for a while:



And here is a picture showing the window after the completion of a game:



Here is how your program should work:

1. The game window should have three frames:
   - A grid frame in the upper left on which the game is played.
     Your program should have constants defined that specify the number of rows in the grid (set it to 30), the number of columns in the grid (set it to 30), and the size (in pixels) of each cell in the grid (set it to 20). The boundaries of the cells should be drawn in black.
   - A control frame with widgets that control the game. The behavior of these widgets is described in detail further down. The height of the control frame should be defined as constant (set it to 100).
   - A score frame in which the user's current score is displayed. Details on this frame are given further down. The width of the score frame should be defined as a constant (set it to 200).

   The relative positions of the frames should be as shown in the pictures above.

2. Refer to the first picture above, which shows the game window when it starts to execute (and before the user does anything). The black square is the head of the snake. At the start of the game the snake is just one square long, and so the head of the snake is the entire snake at the start. Your program should place the snake head at a random position in the grid (all cells equally likely). The snake always has a direction in which it is moving, and the initial direction should be in the direction of the grid boundary that is furthest away. The red square represents food, and the initial location for the food is also randomly chosen (all cells, except for the cell the head of the snake is on, are equally likely).

3. The user starts the game by clicking on the start button. At this point the game updates every timestep according to the following rules:
    - The snake head moves one cell in its current direction (north, east, south, or west). Each cell in the snake body should advance one cell also, following the same path as the snake head.
    - If the timestep would cause the snake head to occupy the same cell as one cell of the snake body, the game terminates. See the third picture above. When the game terminates, the string “Game over” should appear in the score frame.
    - If the game is not in wraparound mode (details further down), and if the timestep would cause the snake head to go off one side of the grid, then the game terminates (again, displaying the “Game over” text in the score frame.
    - If the timestep causes the snake head to enter the same cell as the food, then the snake “eats” the food and its length is increased by one cell. To increase the length, the cell head occupies the cell contained by the food, and the new cell is placed at the end of the snake, in the same position as the last cell of the snake in the previous timestep. Eating the food increases the game score by 1. And when the food is eaten, another piece of food is placed at a random location in the grid (all cells except for cells taken by the snake are equally likely).
4. The default timestep is 1000ms (1 second). The user can speed up the game by moving the “Step Speed” slider.
5. The user can change the direction of the snake by using the arrow keys on the keyboard. The direction remains changed until a different arrow is clicked on.
6. The score pane shows the information illustrated in the pictures above:
    - The points earned so far. This is just the number of pieces of food eaten by the snake so far.
    - The “wall clock” that has elapsed so far, not counting the time when the game is paused. This should be displayed in seconds, with two digits after the decimal point.
    - The points earned per second of game time. (How quickly can the user earn points.)
7. The user can pause the game by clicking on the pause button. The game is restarted by clicking on the start button. The state of the game should be the same as when it was paused.
8. The user can reset the game by clicking on the reset button. The resets the game to the first picture shown above.

9. The user can enter wraparound mode by clicking the wraparound check button. In wraparound mode, if the snake tries to move off one side of the grid, it wraps around

to the other side and continues on, without terminating the game. (For example, the right side wraps around to the left side, in the same row.) Clicking on the wraparound button causes the game to enter wraparound mode immediately, even in the middle of a game.

Guidance to help with your program design and development

In class we covered developing GUI programs using Python's Tkinter toolkit, and we wrote a small example GUI program. But your snake program will be significantly larger, in terms of lines of code, than any of your previous programs. So to help you out, you will first study the design and development of a program that shares many features of your snake program, and that was developed using the same techniques that you will be required to use for your snake program. The program is an implementation of Conway's Game of Life (see https://en.wikipedia.org/wiki/Conway's_Game_of_Life), and no, this is not the Hasbro board game).

So first, go to life.html, where you will study the code for the Game of Life, and how it was developed. We will also talk about it in lab, but this is a document that you can refer back to as you are working on your snake program.

Once you have studied the life program, you will have noticed that there are a lot of similarities in the features the life program and of the snake program - the frame containing the rectangular grid of cells, and the frame containing the control widgets (the start, pause, step, reset, and quit buttons; and the step slider). This suggests that there is code in the life program that you can use in your snake program. And in fact, you should use as much code from life as you can. (This is what working programmers do - they reuse existing code as much as they can, as long as they don't violate copyrights.) This is part of the purpose of this assignment - to read, understand, and reuse parts of code that have been written for other projects.

Program design and development

Having studied the life program, here are the requirements regarding the design and development of your snake program.

1. You should use the model-view-controller design pattern that was used in the life program. You will have three classes: a SnakeModel class that contains data and methods for keeping track of the state of the game; a SnakeView class that manages the display of the window; and a Snake class that is the controller that coodinates the game. The Snake class has the methods that respond to events, and it talks to the SnakeModel class and the SnakeView class. The SnakeModel class should not be aware of anything outside of it, and the SnakeView class should also not be aware of anything outside of it.

2. You will write 7 versions of your program, one corresponding to each iteration of the development process, as described below. The names of the files containing the versions should be `snake1.py`, `snake2.py`, ..., `snake7.py`.

3. You should develop your program using incremental development, as was done for

the life program. The iterations that you should have map to the iterations of the life program, but with one additional iteration associated with the additional frame in the snake window. The iterations that you should have are:

- Iteration 1: A program that displays the game window with the three frames (grid frame, control frame, score frame), each simply a solid color with no widgets. This will be similar to Iteration 1 of the life program, but you will have one more frame, and a different arrangement of frames. The program for this iteration should be in a file named `snake1.py`.

  Run the program and visually validate that it is working correctly.

- Iteration 2: Put widgets in the grid frame. The widgets here will be the cells of the grid. This will be very similar to Iteration 2 of the life program.
  The program for this iteration should be in a file named `snake2.py`.

  Run the program and visually validate that it is working correctly.

- Iteration 3: Put widgets in the control frame. These will be the start, pause, reset, and quit buttons; the step speed slider, and the wraparound check button. This will be very similar to Iteration 3 of the game of life. You have not seen the Tkinter CheckButton widget. YOu can read about it at any one of a number of websites. Just google "Tkinter checkbutton". One that I found is here. (Checkbuttons are the widget of choice to provide for turning on or off some feature.) The program for this iteration should be in a file named `snake3.py`.

  Run the program and visually validate that it is working correctly.

- Iteration 4: Put widgets in the score frame. You will be adding labels (for "Score" and "Game over"), and frames with labels inside for "Points", "Time", and "Points per sec". There was not a corresponding frame in the game of life, but these widgets you've worked with before. Be sure to position them as you see them in the pictures above. The program for this iteration should be in a file named `snake4.py`.

  Run the program and visually validate that it is working correctly.

- Iteration 5: Connect events in the SnakeView widgets to stub event handler functions in the Snake class (that is, the controller). Have the stub function print a short message saying that they have been called. This will be similar to Iteration 4 of the life program. There will be events associated with each of the widgets in the control frame.

  Run the program and interact with all of the widgets (the buttons and the slider in the control frame, and a sampling of the cells in the grid), making sure that the corresponding stub functions are being called. The program for this iteration should be in a file named `snake5.py`.

- Iteration 6: Add the model to your program. Here, you will add a SnakeModel class to your program, similar to the way Iteration 5 of the life program adds a LifeModel class. What does the model need to keep track of? (That is, what instance variables will you need?) See the section below on.

The snake model cannot be tested visually, so you should validate it using unit testing. (See the LifeModel class in Iteration 5 of the life program for an example of unit testing.)

The program for this iteration should be in a file named `snake6.py`.

- Iteration 7: Fill in the event handlers. See Iteration 6 of the life program for examples of this. The handlers will be in the Snake class - that is, the controller. As in the game of life, these handlers will talk to both the model and the view. See the section below on suggestions.

  This is the final version of the program. You can validate it by playing the game and interacting with it in various ways.

  The program for this iteration should be in a file named `snake7.py`.

4. Note that style guidelines that say no function should have more than 30 lines of code (not counting comments). This rule helps to enforce procedural abstraction. Most of your methods/functions should be short, as in the life program.

Comments/Suggestions

The purpose of this section is not to dictate to you how to accomplish things in your program, but just to provide hints/suggestions that might make the going a bit easier for you.

1. In the SnakeModel class, you will have to decide what instance variables are needed to keep track of the state of the game. Things you definitely need to remember are

   - The number of rows and columns in the grid.
   - The location of the cell that the current piece of food is in.
   - The locations of the cells that the snake is in, with some indication of where the head of the snake is.
     (A list of the locations with the head at the front can work well.)
   - The current direction of the snake.
   - The current number of points earned by the user.

   You will probably want to add other instance variable(s) to make methods of the class more efficient. For example, since every time the snake eats a piece of food, another piece of food is placed in a randomly chosen open cell, it might be useful to have a list of the locations of open cells.

   The locations of cells can be stored in a tuple, (row, col).

2. The random module in Python has a method called choice that returns a random element from a list. This can be useful if you keep a list of open cells in the grid.

3. Your snake program advances time in discrete timesteps, just like in the life program. And just like in the life program, you schedule a timer event to happen at the start of the next timestep. See the `continue_simulation` method of the Life class to see

how to schedule a timer event. And if you stop the game, you need to be able to cancel the next timer event. See the `pause_handler` method of the Life class to see how to do this.

4. Your snake program needs to be able to respond to the user pressing the arrow keys. You will set up handlers for these events (and there are four of them - one for each arrow key) in the same way as the mouse click event in a cell was set up in the life program. See the `set_cell_click_handler` method in the LifeView class, and the `cell_click_handler` method in the Life class. One difference of course is the name of the events. For the arrow keys, the event names are `<Left>`, `<Right>`, `<Up>`, and `<Down>`.

File header and style guidelines

Be sure to properly comment and document the code in your `snake.py` file, and use the style guidelines listed below.

1. Place the following information in a comment at the top of the file (i.e., the "header"):
    - The name of the file
    - The date you first created the file.
    - Both of your names and e-mail addresses.
    - A short description of the contents of the file.

2. Place docstring documentation at the beginning of your each function that you write in the file.

3. When naming variables and functions, use descriptive names, and use all lowercase letters with underscores separating words.

4. Each function or method should do one (and only one) well-defined, easy to describe task.

5. No function or method should be longer than 30 lines of code, not counting lines that are either blank or contain only comments. Longer than this and the logic of the method can be hard to follow. If a function/method does person just one easy to describe task, but it is too long, use procedural abstraction to break down the complexity.

6. Use comments within a function or method to explain what a block of code is about to do. Comment an individual line of code only if it needs explaining.

Submission Instructions

Important: To be safe, you should run your final code on both you and your partner's computers. This will ensure that you are not relying on any special setup on your own computer for the code to work.

To submit your code, you will need to synchronize it using Git. To make sure your changes are saved and synchronized, follow these steps.

1. Open the “Source Control” menu, either by clicking on the 3rd icon on the left (right under the magnifying glass) *or* by going to “View” and “SCM”.

2. Your `snake.py` file should show up under the “Changes” section. Click on the “+” icon to the right of the `snake.py` filename to “stage” the changes. This should move the file to a section named “Staged Changes.”

3. Commit your changes by typing in a descriptive message (e.g. “finished solve stack function”) into the “Message” textbox (above the Staged Changes area). After entering the message, click the checkmark icon above the message box to perform the commit. This should remove the `maze_solver.py` and `priority_queue_linked` files from the Staged Changes section.

4. Finally, Sync your commit(s) to the server by clicking the “…” (to the right of the checkmark from the last step) and select the “Sync” option. This will likely result in a pop-up notifying you that the sync will do a push and a pull. Select “OK” (or “OK and don’t ask again”) to complete the sync.

   If you run into problems, make sure you are properly connected to the Internet and try the Sync again. If you are still running into problems, check Piazza and ask a new question there if the answer doesn’t already exist.

To make sure that your changes were synced correctly, have your partner do the final step above (namely “…” and then “Sync”). This should fetch the changes you made. You can then test on their computer to make sure it works exactly the same as on your computer. If your partner has trouble accessing and/or running the file, it is likely that the grader will also have problems and your grade will be negatively impacted.

Finally, when you are ready for me to check your code, post a note to the #psa3_submit folder on Piazza.

Grading

Once you post to Piazza asking to have your program graded, I will test all versions of your snake program. If your program works correctly, I will grade it for the program design requirements and the pair programming requirement (both described above) and give you a grade. Your grade will be 100 minus any program design penalties, minus any pair programming requirement penalty, minus any late penalties (see below).

If not all iterations of your program work as required, I will kick it back to you for continued work.

Late Penalties

Note that these penalties differ from the syllabus (but to your benefit):

1. If you sync by the due date, no penalty. Else,
2. if you sync by 10PM on Tuesday, May 12, 10 points late penalty. Else,
3. if you sync by 10PM on Friday, May 15, 20 points total late penalty. Else,
4. Not accepted after May 15

Academic Integrity

Please review the portion of the syllabus that talks about academic integrity with regard to writing programs. In summary, do not share or show your code to any other team in class, and do not turn in any code that you did not write yourself. Don't look at the code from another team, or from any other source. The point of these programs is for you to develop your coding skills.