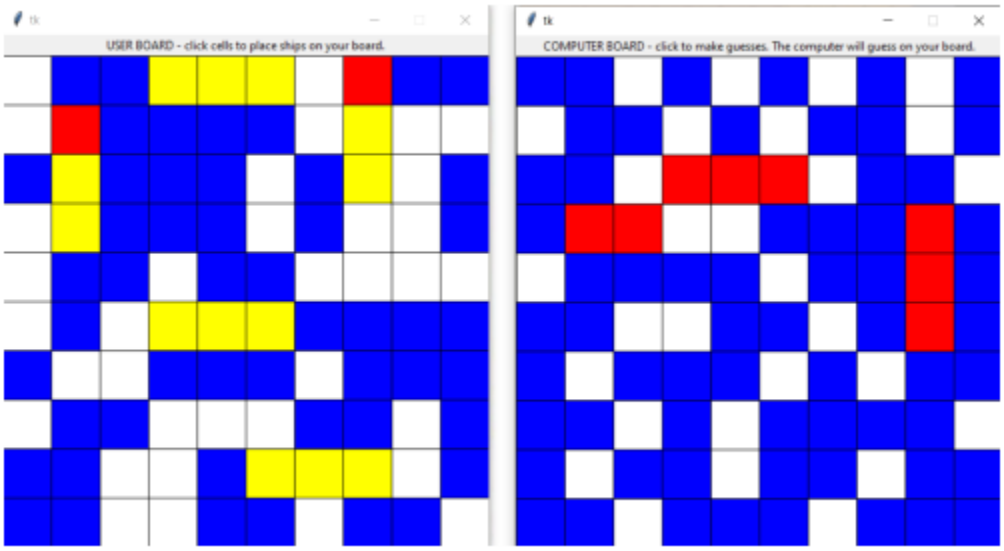
Project: Battleship

CS Focus: Simulation

Requirements: 2D lists, tkinter, and event-based simulation

Description:



In this project, you will use the class simulation framework to program an interactive game of Battleship. If you've never played Battleship before, try an example game here: https://www.mathsisfun.com/games/battleship.html

The game implementation is divided into 3 stages and each stage again divided into days.

In stage - 1 (day 1 and day 2), you will automatically set up ships on the enemy's board. In stage - 2 (day 3 to day 5), you will implement setup for the user by letting them click on squares to set ships. In stage - 3 (day 6 to day 9), you will implement the core gameplay, where the user clicks on the enemy's board to guess ships, and the enemy (the computer) automatically guesses where ships are on the user's board.

Stage - 1
Stage 1 is divided into 2 days day - 1 and day - 2

Stage - 1 (Day: 1)

Today you will begin to create the framework behind the game by having the enemy player set ships in random positions on their board. To do this, you'll write functions that create an empty grid, create random ships, and add random ships to the grid

Step 1: Generate an Empty Grid

Write a function emptyGrid(rows, cols) which creates a new 2D list (called a grid) with rows number of rows and cols number of columns. The value of each grid[row][col] should be 1, which stands for an empty spot that has not been clicked. Return the new 2D list.

Note that we'll use a number system to represent all cells that can show up in the grid. This number system has been provided as global variables at the top of the file. When referring to these values in your code, use the variable names, not the numbers; this will add clarity to your code.

```
EMPTY_UNCLICKED = 1
SHIP_UNCLICKED = 2
EMPTY_CLICKED = 3
SHIP_CLICKED = 4
```

To test this function, run testEmptyGrid().

Note: To keep the starter file from becoming too crowded, all the tests have been moved to the file battleship_tests.py, which you can open and read while debugging. The functions in this file are called at the bottom of battleship.py. If you change the filename of battleship.py, you will need to change the filename imported by battleship_tests.py in the first line of the file too.

Step 2: Create Ships
Write a function createShip() which creates a ship and returns a 2D list of its positions on the grid. All ships are three grid cells long (where each cell is a two-element list) and can be placed either vertically or horizontally on the grid.

First choose a random row in the range [1, 8] and a random column in the same range to be the center point of a ship. We choose 1-8 so that the center of a ship is never placed on the first or last row or column, where it potentially wouldn't fit on the board.

After choosing a row and column as the center, your code should choose a random number 0 or 1 to decide if the ship will be vertical or horizontal. If it chose vertical, create a ship within the same column where the ends are one above and one below the chosen center - e.g., row-1, row, row+1. Similarly, if it chose horizontal, create a ship in the same row where the ends are one column left and right of center - col-1, col, col+1.

To test this function, run testCreateShip().

Step 3: Validate Ships
We'll need to check if randomly-generated ships can actually be added to the grid. Write a function checkShip(grid, ship) that iterates through the given ship and checks if each coordinate in the ship is 'clear'. A coordinate is clear if the corresponding location on the given grid is empty (EMPTY_UNCLICKED). It should return True if all ships are clear and False otherwise.

To test this function, run testCheckShip().

Step 4: Add Ships to Board
Write a function addShips(grid, numShips) which returns the grid updated with numShips added to it. You can do this by looping until the program has added numShips ships to the grid.

Each time through the loop, create a ship using createShip(), then checkShip for that ship on the given grid. If the ship is clear, it can be placed. To place the ship, iterate through each coordinate of the ship, and set the grid at that coordinate to SHIP_UNCLICKED (2), then add 1 to the current count of ships.

To test this function, run testAddShips(). Note that this function involves randomness, so it may pass on one run and fail on another. If it fails, then something is wrong with the function that needs to be fixed.

Stage - 1 (Day: 2)
Today you will begin to create the data required to run the game and draw the initial board. To do this we will write the function that will draw the grid.

Step 5: Store Initial Data
Now that we can make the grid, we need to set up the simulation itself!

You'll need to update makeModel(data) to set up several variables. These variables will be a part of data, so make sure to define them as data["name"] = value.

First, store data about the board dimensions. You should store the number of rows, number of cols, board size, and cell size. You can start with 10 rows, 10 cols, and a 500px board size. You can then compute the cell size based on the board size and number of rows/cols.

Next, store data about the board. You'll need to keep track of two boards- one for the computer, and one for the user. You should store the number of ships on each board (start with 5), then set both the computer board and the user board to be a new empty grid by calling your emptyGrid() function. Finally, update your computer board by calling your addShips() function. You'll add ships to the user board later.

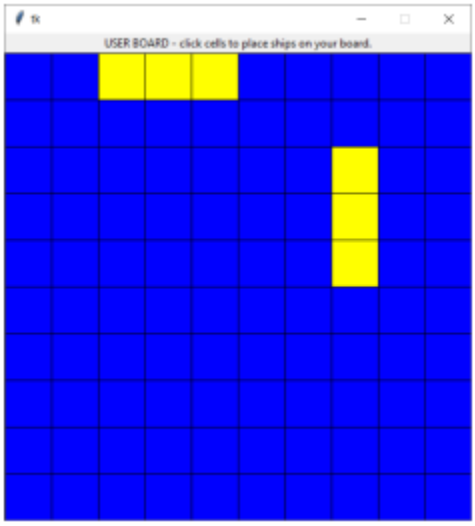
Step 6: Draw the Grid

Now we just need to add graphics. Write drawGrid(data, canvas, grid, showShips), which draws a grid of rows x cols squares on the given canvas. Each square should have the cell size you determined in the previous step. If the cell in the given grid at a coordinate is SHIP_UNCLICKED, the square should be filled yellow; otherwise, it should be filled blue. Ignore the showShips parameter for now; it will be used in a future step.

Hint: recall that we solved a very similar problem in the For Loops lecture…

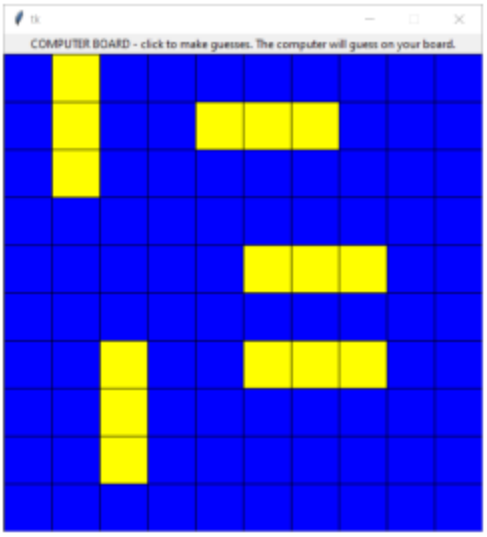
Now update makeView(data, userCanvas, compCanvas) so that it calls drawGrid() two times - once for the computer board and canvas, once for the user board and canvas. Set the showShips variable to True for now (we'll come back to this in a later step). Once this is done, when you run the simulation, you should see the computer's starter board in the computer window, as well as an empty grid in the user's window.

To test your drawGrid() function, temporarily set your user grid equal to test.testGrid() instead of an empty grid. This will set up an example grid from the test functions. Your user board should then look like this:



Once you've verified that the function works, set the user grid back to an empty grid.

At the end, you should pass all of the Day-1 and Day - 2 test cases, and you should have an all-blue user board and a computer board that looks something like this (with ships placed randomly):



Stage - 2
Stage 2 is divided into 3 days day - 3, day - 4, and day - 5

In the second stage of the project, you will write code that lets the user select where to place ships on their grid. To do this, you will write functions that detect which cell has been clicked, create temporary ships, draw temporary ships, check ship validity, and place temporary ships on the user's board.

Before you start this stage, go to the bottom of the starter file and uncomment the two test lines associated with Stage 2.

Stage - 2 (Day: 3)
Step 1: Check Direction

First, write two functions to check whether a set of three coordinates are laid in a specific direction (vertical or horizontal). Write isVertical(ship), which takes in a ship and returns True if the ship is placed vertically, or False otherwise. Recall from last week that a ship is a 2D list of coordinates. A ship is vertical if its coordinates all share the same column, and are each 1 row away from the next part.

Then write isHorizontal(ship), which takes in a ship and returns True if the ship is placed horizontally, and False otherwise. This should work in a similar manner to isVertical, except that the dimensions are flipped.

To test your functions, run testIsVertical() and testIsHorizontal().

Step 2: Detect Clicked Cells

Next, we need to handle mouse events, to detect where a user has clicked on the board. Write the function getClickedCell(data, event) which takes the simulation's data dictionary and a mouse event, and returns a two-element list holding the row and col of the cell that was clicked.

Recall that the event value holds event.x and event.y; you need to convert these to the row and col. How can you do this? You have two choices- either derive the row and col mathematically, or iterate over every possible row and col in the board, calculate each (row, col) cell's left, top, right, and bottom bounds, and check if the (x,y) coordinate falls within those.

To test your function, run testGetClickedCell(). You'll be able to test this manually as well after a few more steps.

Step 3: Update the Graphics

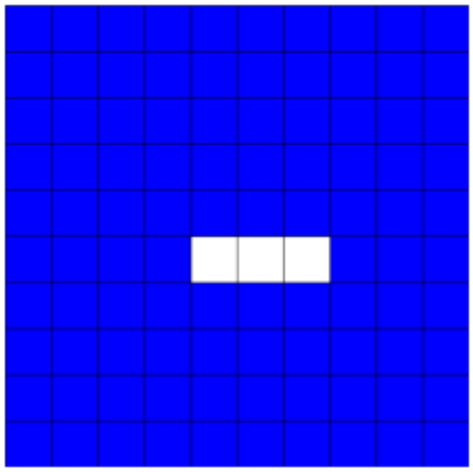
Now we'll start to write code that lets the user add ships to the board by showing 'temporary' ships as the user clicks on cells. These temporary ships will be 'placed' on the board once the user has clicked three cells.

First, we need to represent the temporary ship in the model. It will take the same ship format we've used before- a 2D list. Add a variable for the temporary ship to the data dictionary in makeModel, starting the ship as empty.

We'll need to display the temporary ship on the board. Write drawShip(data, canvas, ship), which takes the data model, a canvas, and a ship 2D list, and draws white cells for each component of the given ship. This can use very similar logic to your code for drawGrid(), except that you'll only draw cells that exist in the ship value.

While you're working on graphics, let's update the makeView function at the top of the file in preparation for the next step. You've already told it to draw the computer and user boards; now tell it to draw the temporary ship. Call drawShip() on the temporary ship in data. Note that the temporary ship should be drawn on the user's board.

To test your drawShip() function, temporarily set your temporary ship equal to test.testShip() instead of an empty ship. This will set up an example ship from the test functions. Your user board should then look like this (see next page)



Once you've verified that the drawShip() function works, change the temporary ship back to an empty ship again.

Stage - 2 (Day: 4)
Step 4: Handle User Clicks

Now we can write a function that will actually handle user clicks and let the user add ships to the board. This will be complicated, so we'll break the process down into three functions: shipIsValid(grid, ship), placeShip(data), and clickUserBoard(data, row, col).

First, implement the function shipIsValid(grid, ship), which takes a grid and a ship and determines whether it is legal to place the ship on that grid, returning a Boolean. Ships should only be added if they A) contain exactly three cells, B) do not overlap any already-placed ships, and C) cover three connected cells (vertically or horizontally). Check this by calling checkShip(), isVertical(), and isHorizontal().

To test this function, run testShipIsValid().

Next, implement the function placeShip(data). This takes the data model and checks if the current temporary ship is valid (based on the function you wrote above). If the ship is valid, 'place' it on the user's board by updating the board at each cell to hold the value SHIP_UNCLICKED. If it is not valid, print an error message to the interpreter. Either way, you should reset the temporary ship to be an empty ship (so the user can try again).

Finally, implement the function clickUserBoard(data, row, col), which handles a click event on a specific cell.

First, check if the clicked location is already in the temporary ship list; if it is, exit the function early by returning. This will keep the user from adding multiple cells in the same location.

Assuming the clicked cell is not in the temporary ship, add it to the temporary ship in the Model.

If the temporary ship contains three cells, call placeShip(data) to attempt to add it to the board. Otherwise, do nothing.

As a last step, we need to keep track of how many ships the user has added so far. Add one more variable to data in makeModel to track the number of user ships; it should start as 0. Then, in placeShip(), add one to that variable if a ship is added.

At the end of the function, check if the user has added 5 ships, and tell them to start playing the game if so. And at the beginning of the function, exit immediately if 5 ships have already been added, to keep the user from adding too many ships.

We can't test placeShip or clickUserBoard automatically, but you'll be able to test them interactively after completing the next step.

Stage - 2 (Day: 5)

Step 5: Manage Mouse Events

Now we can put everything together by capturing and handling mouse events. In the mousePressed(data, event, board) function at the top of the file, use your getClickedCell() function to determine the row and col of the cell that was clicked. Next, note that we've added an additional parameter to the mousePressed() function. This parameter, board, is "user" if the click happened on the user's board, or "comp" if the click happened on the computer's board. If it is "user", call clickUserBoard() with the row and col to update the temporary ship and board.

Once this step is complete, you can test your code by interacting with your simulation. Don't just rely on the test cases- make sure it all works yourself! Try clicking on different cells and making temporary ships. Make sure that the validity checking works, that ships are added to the board properly and that adding an illegal ship doesn't break the game. If you run into errors, try printing out what each helper function returns to determine where the error is occurring across all your functions.

At the end of Stage 2, you should pass all of the stage 2 test cases, and you should be able to place user ships on the board appropriately.

Stage - 3
Stage 3 is divided into 4 days day - 6 to day - 9

In the final stage, you will implement the actual gameplay of Battleship, so that the user can guess which cells on the enemy board contain ships and the enemy can make random guesses on the user board.

To do this, you will need to update many of the functions you have already written, to add new functionality. You will also write functions that update the boards, choose random guesses for the computer, and detect when the game is over.

Before you start this last stage, go to the bottom of the starter file and uncomment the two test lines associated with Stage 3.

Stage - 3 (Day: 6)

Step 1: Handle User Guesses
First, we need to update the simulation code to let the user guess where ships are on the computer's board. This will involve checking the spot that was clicked, updating it as appropriate, and drawing clicked cells on the grid

First, write the function updateBoard(data, board, row, col, player), which updates the given board at (row, col) based on a player's click. If the user clicks on a cell with value SHIP_UNCLICKED (2), the board should update that cell to instead be SHIP_CLICKED (4). Otherwise, if the user clicks on a cell with value EMPTY_UNCLICKED (1), it should update to be EMPTY_CLICKED (3).

To test this function, run testUpdateBoard().

Next, write the function runGameTurn(data, row, col), which manages a single turn of the game after a user clicks on (row, col). First, check whether (row, col) has already been clicked on the computer's board (ie, if it is SHIP_CLICKED or EMPTY_CLICKED); if it has, return early so that the user can click again. If the cell has not been clicked before, call updateBoard() with the appropriate parameters (including "user" as the player) to update the board at that spot.

Now we need to update the simulation code. In mousePressed, if the computer's board has been clicked and all the user's ships have been placed (ie, gameplay has started), call runGameTurn on the clicked cell.

Finally, update drawGrid to account for our two new types of cells. SHIP_CLICKED cells should be drawn as red, and EMPTY_CLICKED cells should be drawn as white. We'll also finally use the parameter showShips to make the game more interesting. Battleship is too easy if you can

see your opponent's ships, so if showShips is False, draw SHIP_UNCLICKED cells as blue (to hide them). Set the drawGrid calls in makeView so that showShips has different values in the two calls; False for the computer canvas, and True for the user canvas.

To test your code, run the simulation, place the user's ships, then try clicking on cells in the computer's board. They should be changed to red or white.

Stage - 3 (Day: 7)

Step 2: Handle Computer Guesses
For every guess the user makes, we want the computer to make a guess as well. Write the function getComputerGuess(board). This function takes a grid (the user's board) and should return a cell that the computer will 'click' on that board. We'll have the computer select cells completely randomly; use the random.randint() function to pick the row and col.

To make sure that the computer doesn't click the same cell twice, use a while loop to keep picking new (row, col) pairs until you find one that hasn't been clicked in the user board yet. You'll need to check the current value in the user's board to tell if this is true.

To test this function, run testGetComputerGuess().

Now update the runGameTurn function. After the user's guess is processed and added to the computer's board, you should have the computer make a guess by calling getComputerGuess(). Then run updateBoard() to update the user board at that location (with "comp" as the player).

To test your code, try running the simulation again. Set up the user's board, then click on a cell on the computer board. A cell should automatically be picked on the user's board as soon as you make your selection. Make sure to also test what happens if you click on a cell you've clicked before- the computer should not make a move in that case.

Stage - 3 (Day: 8)
Step 3: Detecting a Winner

Finally, we want to determine when the game ends, and who wins. We'll need to add a new variable to data in makeModel for this- something to keep track of the winner. It can start as None.

Write the function isGameOver(board), which checks whether the game is over for the given board. The game is done if there are no SHIP_UNCLICKED cells left in the boardin other words, when every ship has been clicked. Return True if the game is over for that board, and False otherwise.

To test this function, run testIsGameOver().

The best place to check whether the game is over is right after the board is updated. In updateBoard(), call isGameOver() on the board parameter. If the result is True, set the winner variable in data to the player parameter.

Now write the function drawGameOver(data, canvas). This should draw a special message on the given canvas if a winner has been chosen. If the winner is "user", draw a congratulations message. If the winner is "comp", tell the user that they lost. Make sure to call drawGameOver() in makeView(), after drawing the boards and temporary ship.

To test this function, set your data winner variable to "user", then see if the appropriate message is drawn on the grid. Also test if the appropriate message is drawn when the variable is set to "comp". Make sure to set the variable back to None when you're done Testing.

Finally, in mousePressed(), only let the user click on cells when a winner hasn't been chosen yet (when the data variable is None).

To test your code, try to win the game! You might have to lose on purpose to test the computer-winning scenario.

Stage - 3 (Day: 9)

Step 4: Detecting a Draw

It isn't very hard to beat a computer that makes guesses entirely randomly, so we'll add one extra feature to the game- declaring a draw. We'll say that if half the board (50 cells) is clicked with no winner, the result is a draw instead.

Add two data variables in makeModel- one that holds the max number of turns (50), and one that holds the current number of turns (which starts at 0). Then, in runGameTurn, once both the user and the computer have made their moves, add one to the number of turns made so far. In that same function, check whether the number of turns is equal to the max number of turns. If it is, set the winner variable in data to "draw".

Then add another message in drawGameOver()- if the winner is "draw", tell the user they're out of moves and have reached a draw. (You can test these graphics by temporarily setting the winner variable to "draw", as was done in the previous step).

Make sure to test the simulation to see what it does if you reach the draw state intentionally.

Step 5: Restarting the Game

We'll add one last feature- letting the user play again. This will be done by detecting if the user presses the Enter key after the game is over.

Add a message in drawGameOver() after each of the possible end-game messages that tells the user to press Enter if they want to play again. Then, in keyPressed, check if the user pressed enter by using the event parameter. If they did, reset the game.

The easiest way to reset the game is to reset all the data variables. You can do this very simply by calling makeModel(data) again.

Make sure to test this last feature by pressing Enter after you finish a game, to see if you can play a new game. Once that's working, congratulations- you're done!

At the end of project, you should be able to play a game of Battleship and win, lose, or tie appropriately.