Lab 4 Report

Understand the Problem:

The overarching goal is to create the “Lights Out” game. It involves a 5 x 5 grid. For this lab, we have new tools at our disposal--lists, tuples, and dictionaries. These tools will simplify our programming a bit compared to our last lab. The 5x5 grid will need to be represented visually using black and white squares--black for lights out, white for lights on. When all of the lights have been turned off, the player is notified of victory and told how many moves that it took them to solve the puzzle. Rather than the player clicking the squares, a prompt will ask first for the row number, then the column number and this will act as the click.

Plan a Solution:

We will need the Unicode characters for black squares and white squares. These characters are U+25A0 for a black square and U+25A1 for a white square.

We will need several different functions for determining which lights to turn on or off based on the row and column chosen by the player.

We will need an appropriate way to keep track of which lights are on or off. A Boolean variable seems like a great choice for this.

Several conditionals or test functions will be needed in our function that turns lights on and off to account for boundary conditions. For example, if the player chooses row 0, then the light above it does not exist and cannot be turned on or off.

All of our code must be encompassed in a while loop that tests whether all lights are off. If all lights are off, the player has won, the loop is ended, and a message is displayed telling the player that they have won. We should also keep track of the number of moves so we can notify the player of how many moves it took them to solve the puzzle. These could also be displayed after each turn as well.

One question that needs answered is: how many functions do we need for testing the boundary values of the board? Now that we can return lists, we have a convenient way of returning a Boolean value for all possible conditions and then implementing the appropriate code based on those values. This could simplify our code significantly.

There are three basic patterns of turning lights on and off--one if you choose a corner, another if you choose any spot on a side which is not a corner and a third if you choose any other spot on the board. I don’t know if these three patterns have to be accounted for in the programming through specific functions or if they will be accounted for through our light-switching function.

Implementation and Testing:

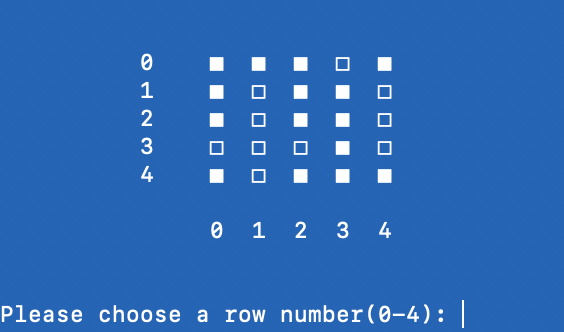
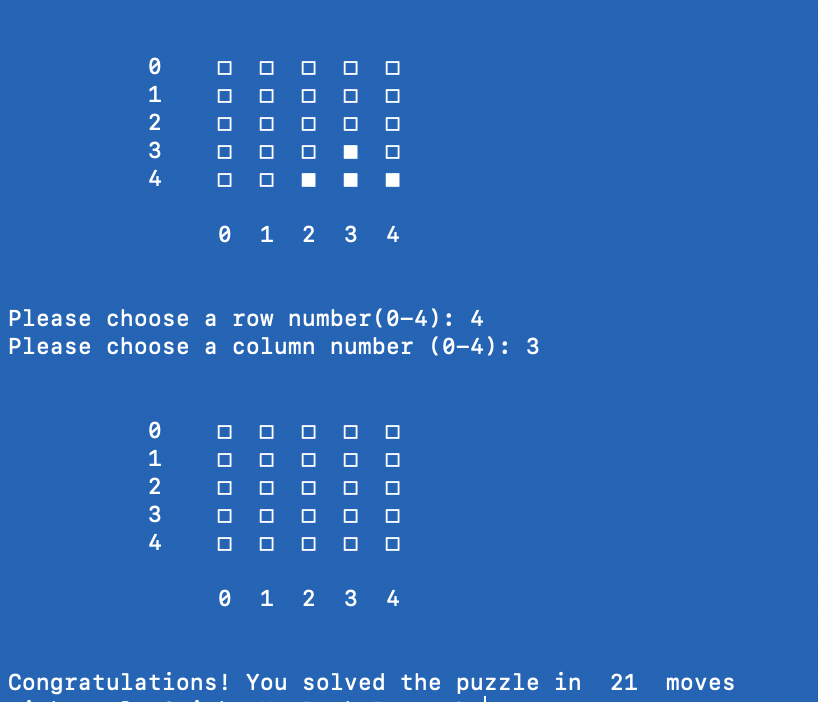
One of the first problems we encountered was that we misunderstood the use of “\N{}” We thought we were supposed to use the Unicode characters and not the name itself. Once we reread the lab instructions, we were able to fix this. After this was solved, we moved onto the randomization of the board.

At first method for randomizing the board involved using the method random.randint(0,1) to randomly return a 0 or 1 twenty-five times. Then a True or False value was assigned to each spot on the board—0’s each became a True value, 1’s became a False value. This method worked fine until the very end, when it became necessary to make the program solvable. Then we had to conceive of a way to make every game solvable.

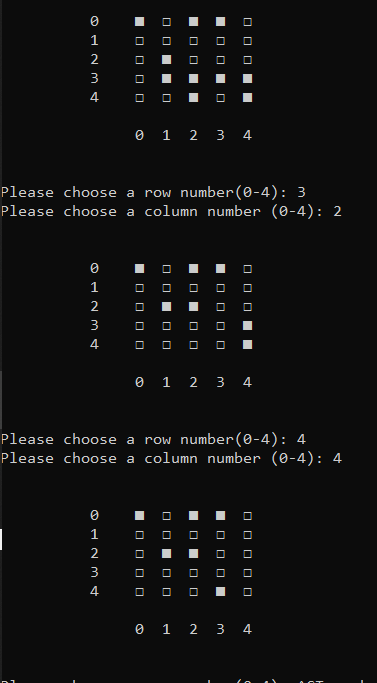
Our idea was based on the idea of a Rubik’s cube. If you start with a solved puzzle and randomly click the board a certain number of times, you must necessarily end up with a solvable puzzle because every click can be reversed. What we did was probably overkill because we created four boards that required only one more click to be solved and then randomly chose one of these boards to “mess up” using random clicks. We realized after the fact that we could have just started with an empty board and randomly clicked it 20 times or so.

Testing was accomplished by using the print function at crucial points in each newly created function. We checked the printed output against our expected output and adjusted the code as necessary.

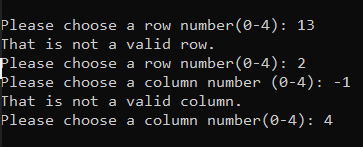
The biggest issue we ran into was tracing a None-type back through several different functions to find its source. It seemed like the None-type was being returned by a certain function when, in reality, this function was calling a different function which was calling another function which was returning a None-type. After this was resolved, we played the game multiple times under multiple starting conditions to make sure that it was solvable every time, and it was.



Final move Initial Board

This screenshot shows two “clicks” during the middle of a game

We also implemented some code which ensures the player inputs a number between 0 and 4, but we did not implement code which could handle letters or other characters.



Reflect and Refactor:

One of the most important lessons we learned was to not assume that a certain part of the code is causing an error simply because that’s where the compiler says the error occurred. Sometimes an error can be causing very indirect symptoms that must be carefully traced through the program.

After the difficulties encountered in the last lab, we knew that planning and organization would be vital to our success in this lab. We spent more time on initial planning and organization and this paid off in the end, as this lab took only four hours to complete.

We followed the “separation of concerns” idea and designed each function to accomplish one task. This made our code clean and organized and relatively easy to follow when debugging.

In order to display the board in a neat, visually pleasing way, we had to learn a few new escape characters and new inputs for the print function The most useful input for the print function was sep = ‘ ‘ at the end of the input; this got rid of the commas and quotations in a list and allowed us to choose what separated each item (in this case a space).

Lastly, we took our time to try to optimize each portion of the code before moving on to the next. If there was any way to make the code cleaner, more compact, or more efficient, we tried to do it. This made our programming experience more difficult initially, but more pleasant in the long run, especially during the testing and debugging phase.