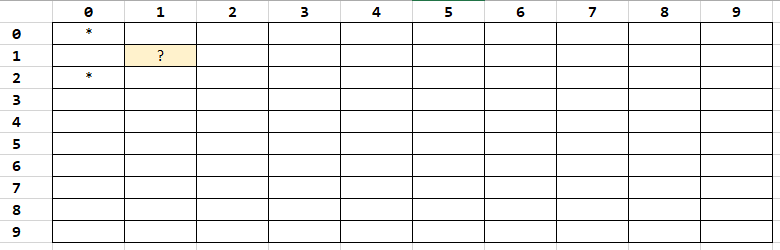
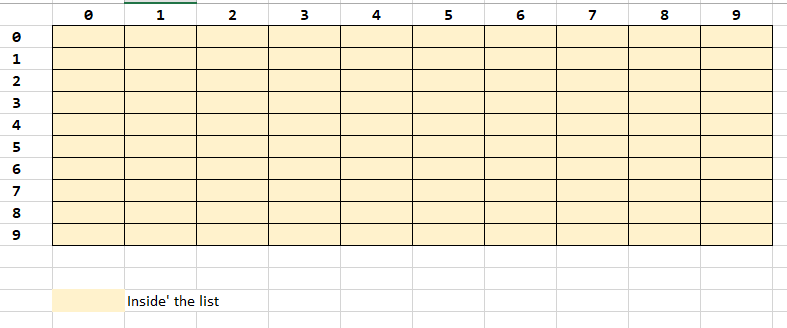
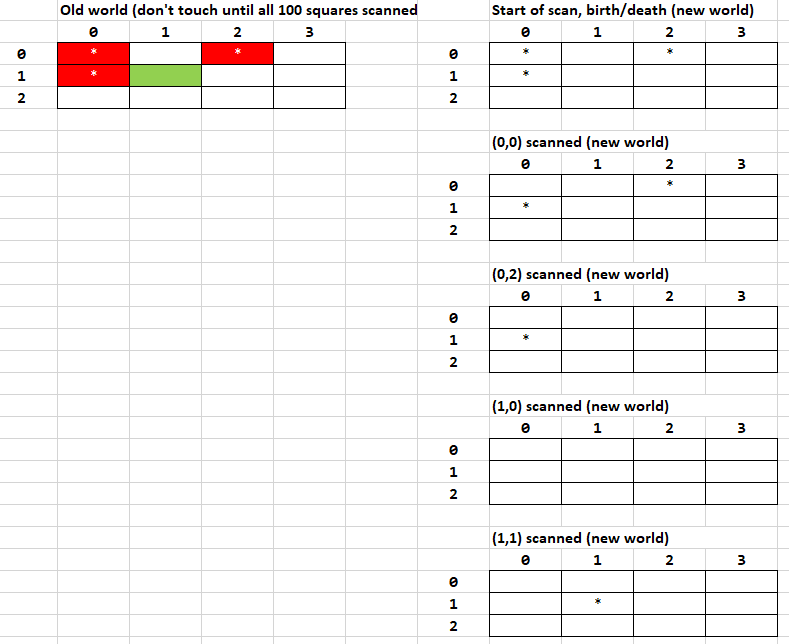
Tip for helping students with out providing a solution

* One approach: help them visualize the problem
  + **Common problem**: how to determine the (row, column) of the neighbors when checking the rules (or even more rudamentary: how do I ‘check’ the neighbors).
    - Use specific examples to help them find the general solution
      * E.g. making change by returning: quarters, dimes, pennies and minimizing the amount of change returned.
        + What change is given back when $0.50 is owed?
        + What change is given back when $0.53 is owed?
        + What change is given back when $0.61 is owed?
        + Etc. (until the general pattern is recognized)
      * “Draw a diagram” – but JT’s tip is to draw a diagram which labels everything (i.e. row, columns for a list) and represents a specific example.
      * Applying both techniques you can use as a reference the spreadsheet ‘biosphere’
      * Example problem: determining the (row, column) coordinates of the neighbors for any of the 100 squares.
      * 
        + Q1: How many neighbor exist for the square marked '?'
        + Q2: How did you determine what are the neighbors?
        + Q3: What are the (row,column) coordinates of the neighbors of '?'
  + **Common problem**: what if the neighbor is ‘outside’ the bounds of the biosphere list?
    - To determine what’s ‘outside’ first define what’s ‘inside’
    - (If they have trouble then show them the diagram of the list and ask again what’s defined as ‘inside’).
    - 
    - Once they are able to distinguish inside from outside (range of the row,column references) then remind them of the Boolean function you went over.
  + **Student question**: are separate functions needed to handle the neighbor count for locations that border on edges or the corners.
    - Ideally the program should be properly handle the count with one function for all cases.
    - But if that’s too much for student’s to visualize then separate functions for the special cases is acceptable.
  + **Student question**: How are the birth and death rules applied at the same time using the ‘oldWorld’ and ‘newWorld’.
    - * + Illustrate with a diagram but not code
        + 
    - Cases in the right hand column (data in old world doesn’t change while we run the code for the 4 cases)
      * Determine birth/death at (0,0), critter at (0,0) dies of loneliness in the new world.
        + In oldWorld critter at (0,0) is untouched.
      * Determine birth/death at (0,2), critter at (0,2) dies of loneliness in the new world.
        + In oldWorld critter at (0,2) is untouched.
      * Determine birth/death at (1,0), critter at (1,0) dies of loneliness in the new world.
        + In oldWorld critter at (1,0) is untouched.
      * Now when the program tries to determine birth/death at (1,1), program examines the pattern in the oldWorld not the newWorld
        + In the image in the left column examining location (1,1) in green
        + This location has 3 neighbors in the oldWorld so the critter at (1,1) remains alive in the newWorld (last image in the column on the right)
        + Summary of the scan for (0,0) to (1,2)
        + Although 3 deaths occurred at locations that bordered (1,1) the 3 deaths appear to occur at the same time that the count for this location is made thus the rules appear to be applied at the same time.
        + After all 100 squares are scanned then the contents of the oldWorld are copied to the newWorld.
        + The next turn the pattern is scanned in the oldWorld while new births/deaths are made in the newWorld.
  + **Student question:** 
    - How does user input affect the debugging mode.
    - It does not directly cause debugging messages to appear
      * Why? Because if the mode is already on then selecting ‘d’ or ‘D’ will turn off the display i.e. debugOn is now set to False.
    - User input merely ‘toggles’ debugging mode
      * Pressing ‘d’ or ‘D’ when the flag is off will turn it on
      * Pressing ‘d’ or ‘D’ when the flag is on will turn it off
    - Example of a function that changes negative numbers to positive and vice versa (zero unaffected).
    - The branch isn’t really needed except to display output of course but it’s illustrating how state is checked and instructions execute

num = 7

def flip():

global num

num = int(“enter new value for num: “))

If (num < 0):

Print(“Negative becomes positive”)

Num = num \* -1

If (num > 0):

Print(“Positive becomes negative”)

Num = num \* -1

def useNum():

if (num < 0):

print(“You are negative today”)

if (num > 0):

print(“Sunny days”)

* + **Student question:** 
    - How can we implement the debugging mode.
    - Here’s a couple of acceptable approaches.
    - To be awarded credit the program has to:
      1. Allow debugging mode to be toggled (off becomes on, on becomes off)

If (debugOn == True):

debugOn = False

Else:

debugOn = True

1. Display a message, any message when the debugging mode is set to true.
   * + - * **Example 1 for type of debugging message**: if the flag is set to true show the sequence of function calls (“stack trace” but don’t use this term with students).

SIZE = 10

debugOn = False

def oneEmpty():

if (debugOn == True):

print("<<< oneEmpty() >>> ")

...

def display(turn,oldWorld,newWorld):

if (debugOn == True):

print("<<< display() >>> ")

...

* + - * + **Example 2**: display information about the state of the game e.g. the (row/column) on which a neighbor count is performed showing the relative coordinates for the neighboring squares as well as the total critter count for the neighbors.

Hit enter to continue ('q' to quit): d

<<< DEBUG messages ON! >>>

Counting the after case

**(ROW,COL): 0/0**

(row,col): -1/-1

(row,col): -1/0

(row,col): -1/1

(row,col): 0/-1

(row,col): 0/0

(row,col): 0/1

(row,col): 1/-1

(row,col): 1/0

(row,col): 1/1

ROW/COL 0/0 count=0

###

**(ROW,COL): 0/1**

(row,col): -1/0

(row,col): -1/1

(row,col): -1/2

(row,col): 0/0

(row,col): 0/1

(row,col): 0/2

(row,col): 1/0

(row,col): 1/1

(row,col): 1/2

ROW/COL 0/1 count=0