```
import turtle as t
import tkinter as tk
import random
import math

##########################################
###    I.  VARIABLE AND TURTLE SETUP     ###
##########################################
''' Variable and Turtle Setup - TABLE OF CONTENTS:
A. Variables
-These are variables that need to be defined globally, but not dynamically.

B. Turtle, Screen Object Setup
-These are the turtles/screen objects that need to be defined globally and used throughout the program.
-Some turtles, such as block_i_j for each cell of the game, need to be created dynamically later in the code and
cannot be created here.
    -This is because the blocks are created based on the variable turt_in_row, or how many turtles are in the row.
So, we cannot make
    the blocks in advance (here), because we don't know how many we'll need.
    -Also, the blocks need to be created and destroyed each level, but the turtles here (drawer, set_block, set_x,
and circler) are used
    throughout every level of the program.
'''

######################
# A. Variables
######################



### Variables for visual formatting, basic functionality ###
width, height = 450, 450
default_delay = 5
font_setup = ("Verdana", 15, "normal")

board_shift = 50       #amount by which the all objects will be shifted left and down to center them on board
num_shift = 5          # amount to shift numbers down to be centered on each row/col
turtle_gap = 28        #the gap between turtles on the gameboard
switch_y = -125        #y value of switch at bottom of screen

block_color_tuple = ((0.8941176470588236, 0.9254901960784314, 0.9294117647058824), (0.8941176470588236,
0.9254901960784314, 0.9294117647058824))

t_switch_resize = 1.2   #amount to resize the turtle switches when switching from blocks to X's
t_switch_rotate = 90    #amount to rotate the turtle switches when switching from blocks to X's
block_state = True      #This variable will control the switch at the bottom of the screen (T = blocks drawn, F =
X's drawn)

shift_for_hearts = 30   #amount to shift the switch, messages due to the hearts' placement.
heart_horiz_shift = 1.2 #amount to shift subsequent hearts horizontally
heart_img_empty = 'nonograms_images/empty-heart.gif' #found from https://www.shutterstock.com/image-vector/set-
pixelated-heart-icons-digital-260nw-2320764891.jpg
heart_img_full = 'nonograms_images/full-heart.gif' #found from https://www.shutterstock.com/image-vector/set-
pixelated-heart-icons-digital-260nw-2320764891.jpg

screen_setup_needed = True   #This variable is so that 1st time through level 1, we call screen_setup(). If we
replay level 1 b/c of game over, we DO NOT call screen_seup again.

bar_len = 25                    #Used for rectangle is polyRectangle (part of X cursor)
bar_width = 2                   #Used for rectangle is polyRectangle (part of X cursor)
X_horiz = bar_len *math.sqrt(2) #Used for XCursors to place 2nd rectangle at correct spot

### Variables that have an effect on gameplay, changed within code ###
turt_in_row = 5     #the number of block turtles in 1 row/column of the board
current_level = 1   #the game starts with level 1.
max_levels = 2
level_list = []     #This is supposed to be empty and populated with functions. Hard coded for debugging purposes.
select_diffs_per_level = False  #for difficulty mode selection
lost = False        #Boolean to keep track of loss. When we lose (and we are selecting diffs per level), we should
not be reprompted for level difficulty.


### Variables that have an effect on gameplay, NOT changed within code ###
max_lives = 3
current_lives = max_lives       #Max_lives is a constant that does not change. But current_lives will be the "life
meter," changing per level
lives_reduction = 1             #used within lose_a_life, determines how many lives are reduced when we replay a
level


######################
```

```
# B. Turtle, Screen Object Setup
######################

#Create screen object
wn = t.Screen()
# t.title('Nonograms Game!')
wn.delay(default_delay)

#Add heart images (used for lives functionality)
wn.addshape(heart_img_empty)
wn.addshape(heart_img_full)

'''#Creating the turtles that are never destroyed/ reassigned. Since they all need to be created, penup, and
hidden, I did this in a loop.
1. drawer--draws the lines of the gameboard, numbers, and switch border
2. circler--used for circling the correct value on the switch at bottom of screen
3. set_block--turtle that represents the "block" part of the switch
4. set_x--turtle that represents the "x" part of the switch'''
#creating a list to iterate over
turt_names = ['drawer', 'circler', 'set_block', 'set_x']

for name in turt_names:
    globals()[name] = t.Turtle()
    globals()[name].hideturtle()
    globals()[name].penup()

#delete these variables, since they're no longer needed.
del name, turt_names


##############################################
###             II.  FUNCTIONS             ###
##############################################

''' ***FUNCTIONS - TABLE OF CONTENTS***

A. LEVEL SETUP FUNCTIONS
#These functions are called once per level. They format/ draw/ create the board.

    screen_setup(current_lev, t_in_row)
        switch_setup()
        draw_lines()
        create_random_answer()
        create_custom_5_answer_testing()      ##TESTING ONLY
        write_numbers()
            get_row_numbers(board)
            get_col_numbers()
                col_to_row_matrix(board)
            get_row_and_col_sums(row_n, col_n)
        turn_on_clicks_and_switch(num_in_row)
        place lives()
        write_level_num()


B. RESET FUNCTIONS
#These functions are called to restart the level/ reset something so it can be played again.

    reset_current_level()
        reset_switch()
        reset_lives(reduction)
    clear_screen_and_restart(cur_lev, t_in_r)


C. SETUP ONLY ONCE PER CODE RUNNING
#These functions are only called once per running of the code, to create the polygon X image to be used for the X's
in the game

    XCursors() **
        polyRectangle(turt, x, y, slant, length1, length2)

** = While technically a part of the game setup, the XCursors function needs to be called only once per code
running. Therefore, this was moved
out of the screen_setup function and into its own category.


D. SWITCH FUNCTIONS
#These functions are called multiple times per level. They make the functionality of the switch at the bottom of
the screen. Block_or_x is called
via onkeypress with the "x" key.

    block_or_x()
```

```
        circler_fun(x_or_b)


E. CLICKING AND CHECK FUNCTIONS
# These functions are all called whenever a block is clicked. Some of the things they handle: determining which
block we've clicked,
# changing its appearance, checking that the cell is correct with the puzzle, checking that the row/column/ puzzle
is correct,
# handling what happens when a cell is incorrect or a row/col is correct, etc.

    clicked(x,y)
        check_cell(current_row_num,current_col_num)
        new_check_row(current_row_num)
        new_check_col(current_col_num)
        check_win()
        turt_dance(turt)
        lose_a_life()


F. DEBUGGING FUNCTIONS
# This section is for functions that will not be used in the "final" version of the game but are for use during
debugging.
# There is just one function in this section, print_board(). It is used to print both the answer_board and
game_board. We use this to
# check that the "behind the scenes" changes match the on-screen changes.
 print_game_board(game_board)

G. WIN/ LOSE/ GAMEPLAY FUNCTIONS
# These functions are necessary for playing multiple levels, winning, losing functionality

    play_one_level()
    turn_off_turtle_events()

    win_level()
    win_whole_game()
    game_over()


H. BEFORE OR BETWEEN GAMEPLAY
# Called within play_one_level(), has effect only when picking difficulty per level
    select_diffs_this_level()

# Beginning of game, starting message/ pick settings
    welcome_and_settings()
        random_diffs(m_levels)
        increasing_diffs(m_levels)
        diffs_per_level()
'''

######################
# A. Game Level Setup functions
######################

'''screen_setup is the 'brains' of this operation. It does many things:
    1) formats the screen based on the number of turtles in the row,
    2) creates the turtles for the board
    3) draws the lines for the board
    4) sets up/places the turtles to act as switches at the bottom of the screen
    5) draws the border (and initial circle) around the switches at the bottom of the screen
    6) Turns on the onclick() for each turtle on game board (also enables typing "x" to activate switch)

    This function needs to use the globals() dictionary:
    a) creates blank gameboard, based on turt_in_row
    b) creates block_{i}_{j}, all the turtles for the board, based on turt_in_row
    '''
def screen_setup(current_lev, t_in_row):
    ####
    # 1. Setup for "behind the scenes" variables and screen
    ###
    global turt_in_row, width, height, switch_y, board_shift, turtle_gap, font_setup, shift_for_hearts

    #Screen_setup will be used to override the default value of turt_in_row. This will be necessary for when we
pick a different diffulty per level.
    turt_in_row = t_in_row

    #conditional setup based on 5, 10, 15, or 20-board size
    if turt_in_row ==5:
        width, height = 450, 450
        board_shift = 50  #amount by which the board will be shifted left and down to center it
        switch_y = -125
    elif turt_in_row==10:
```

```
        width, height = 600, 600
        board_shift = 100
        switch_y = -175
    elif turt_in_row == 15:
        width, height = 750,750
        board_shift = 200
        switch_y = -275
    elif turt_in_row == 20:
        width, height = 950, 950
        board_shift = 270
        switch_y = -350
    else:
        print("That is an invalid entry. Here is a 5-by-5")
        screen_setup(current_level, 5)

    #creating/redefining a blank board to begin
    globals()[f'game_board'] = [['-' for column in range(turt_in_row)] for row in range(turt_in_row)]

    #creating a random answer
    create_random_answer()

    #set up screen, add images
    wn.setup(width, height)
    wn.tracer(False)

    ###
    #2.  Setup for the visual aspects of the screen.
    ###

    '''#dynamically create the turtles for the screen
    block0_0 through 0_4 should be in column 0 (first column-- x val of 0)
    block0_0 through 4_0 should be in row 0 (first row-- y val of 0)

    Original idea for dynamically creating variables came from this website:
    https://www.quora.com/How-do-I-create-a-loop-that-creates-variables-in-
Python#:~:text=To%20create%20a%20loop%20that%20creates%20variables%20in%20Python%2C%20you,in%20range(1%2C%204)%3A'''

    for j in range(turt_in_row):
        for i in range(turt_in_row):
            globals()[f'block{i}_{j}'] = t.Turtle(shape="square")
            globals()[f'block{i}_{j}'].color('#E4ECED')
            globals()[f'block{i}_{j}'].penup()
            globals()[f'block{i}_{j}'].goto(i*turtle_gap-board_shift, j*turtle_gap-board_shift)

    #In case there is any writing on the screen already made by the turtle drawer, clear it.
    drawer.clear()

    #draw lines for gameboard
    draw_lines()

    #draw the numbers along the top and left of the board, based on answer (function in progress)
    write_numbers()

    #Drawing the switch and setting up the turtles for the "switch" at the bottom of the screen.
    #The switch is used to change the clicks from X's to blocks.
    switch_setup()

    #turn on click functionality for the already-created turtles on the board; turn on typing "x" to use the switch
    turn_on_clicks_and_switch(turt_in_row)

    #place hearts on the screen for the lives, equal to the global variable max_lives
    place_lives()

    write_level_num()

    wn.tracer(True)

#Formats the "switch" at the bottom of the screen and its turtles
def switch_setup():
    global switch_y, drawer, set_block, circler, set_x, font_setup
    #Drawing the border around the "switch" to Switch from Solid blocks to X's
    drawer.goto(-25,switch_y - 15-shift_for_hearts)
    drawer.showturtle()
    drawer.pendown()
    drawer.pensize(1)
    drawer.fillcolor('#E4ECED')
    drawer.begin_fill()
    drawer.forward(50)
    drawer.circle(15,180)
    drawer.forward(50)
    drawer.circle(15,180)
```

```
    drawer.end_fill()
    drawer.hideturtle()
    drawer.penup()

    #Setting up the turtles to act as the switches at the bottom of the screen
        #block switch
    set_block.goto(-15, switch_y-shift_for_hearts)
    set_block.showturtle()
    set_block.shape("square")
    set_block.shapesize(1.2)
    set_block.right(360)
    #circling the Block switch, because block_state is true to start program
    circler.goto(-15,switch_y-18-shift_for_hearts)
    circler.pendown()
    circler.circle(18)
    circler.penup()

        #X switch
    set_x.goto(15,switch_y-shift_for_hearts)
    set_x.showturtle()
    set_x.shape("turtle")               #changing to arbitrary shape and back to X will move the X to the "top"
above oval
    set_x.shape("X_turtle")

        #A message to indicate how to change from X to block
    drawer.penup()
    drawer.goto(0,switch_y-40-shift_for_hearts)
    drawer.write("Use the 'x' key to change your click from blocks to X's.", align = 'center', font=font_setup)

#Draws the horizontal and vertical lines
def draw_lines():
    global drawer, turtle_gap, board_shift, turt_in_row
    #drawing vertical lines
    drawer.setheading(90)
    for i in range(turt_in_row + 1):
        drawer.goto(i*turtle_gap-.5*turtle_gap-board_shift,-.5*turtle_gap-board_shift)
        drawer.pendown()
        if i %5 == 0:
            drawer.pensize(4)
        else:
            drawer.pensize(1)
        drawer.forward(turt_in_row*turtle_gap)
        drawer.penup()

    #drawing horizontal lines
    drawer.setheading(0)
    for i in range(turt_in_row+1):
        drawer.goto(-.5*turtle_gap-board_shift, i*turtle_gap-.5*turtle_gap-board_shift)
        drawer.pendown()
        if i %5 == 0:
            drawer.pensize(4)
        else:
            drawer.pensize(1)
        drawer.forward(turt_in_row*turtle_gap)
        drawer.penup()

#Creates a random grid answer, based on turt_in_row
#Uses globals() dictionary to create answer matrix, based on turt_in_row
def create_random_answer():
    global turt_in_row
    globals()['answer'] = []

    #create each row of the answer
    for j in range(turt_in_row):        #making each uniquely named row: answer_row_0, answer_row_1, etc.
        globals()[f'answer_row_{j}'] = []
        for i in range(turt_in_row):     #for each row, appending 0's and 1's to fill out the row
            globals()[f'answer_row_{j}'].append(str(random.randint(0,1)))

    #putting the rows together in one answer board
    for j in range(turt_in_row):
        globals()[f'answer'].append(globals()[f'answer_row_{j}'])

    #deleting the answer_row_{j} variables
    for j in range(turt_in_row):
        del globals()[f'answer_row_{j}']

    return globals()[f'answer']

#Creates a custom 5-by-5 game board
def create_custom_5_answer_testing():
    globals()['answer'] = []
```

```
    #create each row of the answer
    for j in range(5):        #making each uniquely named row: answer_row_0, answer_row_1, etc.
        globals()[f'answer_row_{j}'] = []

    globals()['answer_row_0'] = ['0', '1', '0', '1', '0']
    globals()['answer_row_1'] = ['0', '0', '0', '0', '1']
    globals()['answer_row_2'] = ['0', '1', '1', '1', '1']
    globals()['answer_row_3'] = ['0', '0', '0', '0', '0']
    globals()['answer_row_4'] = ['0', '0', '1', '0', '1']

    #putting the rows together in one answer board
    for j in range(5):
        globals()[f'answer'].append(globals()[f'answer_row_{j}'])

    #deleting the answer_row_{j} variables
    for j in range(5):
        del globals()[f'answer_row_{j}']

    #print this ANSWER board
    print('printing CUSTOM ANSWER board...')
    print_game_board(globals()[f'answer'])
    return globals()[f'answer']

#Writes the numbers (from get_col_numbers and get_row_numbers) that go on the left and top of the screen.
def write_numbers():
    row_nums = get_row_numbers(globals()['answer'])
    col_nums = get_col_numbers()

    '''The following function is not needed for writing the numbers, but this was a good place to call it.
    It is used to create two globals lists, row_sums and col_sums, that are necessary for new_check_row
    new_check_col. I decided to call this function here because it is only needed once per level, even
    though new_check_row and new_check_col will be called multiple times per level.
    '''
    get_row_and_col_sums(row_nums, col_nums)

    #actually drawing the row numbers on the left
    for j in range(len(row_nums)):
        drawer.goto(-.75*turtle_gap-board_shift, j*turtle_gap-board_shift-num_shift)
        drawer.write(" ".join(row_nums[j]), align = 'right', font = font_setup)

        #top column numbers
    for i in range(len(col_nums)):
        #When I didn't reverse each list in col_nums, it printed the numbers for a given column upside-down
        reversed_nums = [col_nums[i][val] for val in range(len(col_nums[i])-1,-1,-1)]
        drawer.goto(i*turtle_gap-board_shift-num_shift, turt_in_row*turtle_gap-num_shift-board_shift)
        drawer.write('\n'.join(reversed_nums), align = 'left', font = font_setup)

#Using answer, returns the numbers that will be displayed on the left of the screen
#Called within write_numbers
def get_row_numbers(board):
    global turt_in_row

    row_nums = []
    #creating one number list per row of answer. This will start at row 0, the bottom of the image.
    for j in range(len(board)):
        globals()[f'row_nums_list_{j}'] = []

    for j in range(len(board)):        #loop over every row
        temp_sum = 0
        for i in range(len(board)):        #loop over every index in the row

            if board[j][i]=='1' and i != (turt_in_row - 1):     #when there is a 1 not at the end of row/ col
                temp_sum += 1
            elif board[j][i]=='1' and i == (turt_in_row -1):    #when there is a 1 at the end of row/ col
                temp_sum += 1
                globals()[f'row_nums_list_{j}'].append(str(temp_sum))
                temp_sum = 0

            elif board[j][i] == '0' and temp_sum != 0:         #when there is a 0 throughout a row/col
                globals()[f'row_nums_list_{j}'].append(str(temp_sum))
                temp_sum = 0

            elif i == (turt_in_row-1) and temp_sum == 0 and globals()[f'row_nums_list_{j}']==[]:    #when there is
a 0 at the end of row/ col of all 0's
                globals()[f'row_nums_list_{j}'].append(str(temp_sum))

        row_nums.append(globals()[f'row_nums_list_{j}'])
        del globals()[f'row_nums_list_{j}']
    return row_nums
```

```
#returns a matrix where the row and columns are switched.
#Called within get_col_numbers
def col_to_row_matrix(board):
    col_to_row_list = []
    #create one list for each column of the original board
    for col_num in range(len(board)):
        globals()[f'col_to_row_{col_num}'] = []

    #iterate through the rows and append each value to the correct column list
    for j in range(len(board)):        #j is row number of original board (y value)
        for i in range(len(board[j])): #i is col number of original board (x value)
            globals()[f'col_to_row_{i}'].append(board[j][i]) #append this value to correct col list

    #printing each column, appending to col_to_row_list
    for col_num in range(len(board)):
        col_to_row_list.append(globals()[f'col_to_row_{col_num}'])
        del globals()[f'col_to_row_{col_num}']
    return col_to_row_list

#Using answer, returns the column numbers that will be displayed on the top of the screen
#Called within write_numbers
def get_col_numbers():

    #First, switch the x and y of the matrix.
    transposed_answer_matrix = col_to_row_matrix(globals()['answer'])

    #Then, use get_row_numbers to get the row numbers of the transposed matrix.
    #Thus, getting the column numbers of the actual matrix.

    col_nums = get_row_numbers(transposed_answer_matrix)
    return col_nums

#Used to get the numerical sums of the number of blocks in each row. Does this by changing row_nums and col_nums,
#which are lists of lists of stringed integers, into a list of integers.
def get_row_and_col_sums(row_n, col_n):
    #For this algorithm, I got input from https://www.geeksforgeeks.org/python-get-summation-of-numbers-in-string-
list/
    #I sought help because I needed help with changing a list of lists of stringed integers into a list of integer
sums
    global row_sums, col_sums


    #getting the sum of the 1's for each row of row_nums
    row_sums = [sum(int(ele) for ele in sub) for sub in row_n] #borrowed from the above website
    col_sums = [sum(int(ele) for ele in sub) for sub in col_n] #borrowed from the above website

#Enable the on_click events for each block turtle.
#Also, enable typing "x" to switch the gameboard from clicking X's to clicking blocks.
def turn_on_clicks_and_switch(num_in_row):
    for j in range(num_in_row):
        for i in range(num_in_row):
            globals()[f'block{i}_{j}'].onclick(clicked)
    wn.onkeypress(block_or_x,'x')

#This function places the lives images on the screen. Based on the global current_lives, it places that many full
hearts on the screen.
def place_lives():
    '''this function is used to initially create the lives turtles (based on global variable max_lives). It places
them at the bottom
    of the screen. It gives them the "full heart" image. '''
    global current_lives
    current_lives = max_lives  #for the new level, reset the global current_lives to be equal to max_lives
    for i in range(max_lives):
        globals()[f'heart_{i}'] = t.Turtle()
        globals()[f'heart_{i}'].shape(heart_img_full)
        globals()[f'heart_{i}'].penup()
        globals()[f'heart_{i}'].goto(-35+i*heart_horiz_shift*turtle_gap, switch_y+10)

#This function places the current level number on the screen.
def write_level_num():
    global current_level
    drawer.goto(-35, switch_y+30)
    drawer.write(f"Level {current_level}", align = 'left', font=("Arial", 18, 'normal', 'bold', 'underline'))


######################
# B. Reset functions
######################

#This function is used to leave the gameboard as is, but allow for the user to replay the curernt level. This is to
be one of the options after game over.
```

```
def reset_current_level():
    '''used to reset current level after a game_over. Called within the game_over function '''
    wn.tracer(False)
    #reset the colors of the blocks to be a square in the neutral color again.
    for j in range(turt_in_row):
        for i in range(turt_in_row):
            globals()[f'block{i}_{j}'].clear()
            globals()[f'block{i}_{j}'].shape("square")
            globals()[f'block{i}_{j}'].color('#E4ECED')

    #recreating a blank board
    globals()[f'game_board'] = [['-' for column in range(turt_in_row)] for row in range(turt_in_row)]

    #should reset the switch to "square"
    reset_switch()
    #rest the lives, decreased by life reduction.
    reset_lives(lives_reduction)
    wn.tracer(True)


    turn_on_clicks_and_switch(turt_in_row)   #turn on clicks of the gameboard turtles (and typing 'x'), to fix bug
where some turtles couldn't be clicked after this function.

#This function is to be called within reset_current_level. Note, it does NOT move the switches to a new location
#It resets the switch to being set to "block".
def reset_switch():
    global switch_y, set_block, circler, set_x, block_state

    #Resetting global block_state to True, because we want to reset the "click block" functionality
    block_state =  True


    #Recircling, resizing block switch (it needs to be large again)
    set_block.shape("square")
    set_block.shapesize(1.2)
    set_block.showturtle()
    #circling the Block switch, because block_state is true again
    circler.clear()     #clear any old writings of circler
    circler.goto(-15,switch_y-18-shift_for_hearts)
    circler.pendown()
    circler.circle(18)
    circler.penup()

    #resizing X switch back to regular size
    set_x.shape("X_turtle")
    set_x.shapesize(1)
    set_x.showturtle()

#This function is used to reset the number of lives, based on a reduction. This is to be used to reset the lives
after a game over.
#If reduction = 0, then the same number of lives are replaced.
def reset_lives(reduction):
    """this function is used to reset the lives when the level is reset. It does NOT create turtles, but it loops
over the existing turtles
    and resets current_lives.
    reduction is an integer <= max lives. Otherwise, the function passes.
    This function is to be used for resetting after a game over (functionality not fully created yet.)"""
    global current_lives

    if reduction >= max_lives:
        print('life reduction must be less than maximum life number of:', max_lives)
        pass
    else:
        current_lives = max_lives - reduction
        for i in range(current_lives):
            globals()[f'heart_{i}'].shape(heart_img_full)
        for i in range(current_lives, max_lives):
            globals()[f'heart_{i}'].shape(heart_img_empty)

#This function is to be used to reset the screen and variables. It then restarts the game. Goal: use it to reset
after a level change, or game over.
def clear_screen_and_restart(cur_lev, t_in_r):
    global turt_in_row, game_board, drawer, circler, set_x, set_block, wn, answer

    #delete the turtles in the gameboard
    wn.tracer(False)
    for j in range(turt_in_row):
        for i in range(turt_in_row):
            globals()[f'block{i}_{j}'].clear()
            globals()[f'block{i}_{j}'].hideturtle()
            del globals()[f'block{i}_{j}']
```

```
    #delete the writings of any other turtles
    drawer.clear()
    circler.clear()

    #hide these turtles
    set_x.hideturtle()
    set_block.hideturtle()

    #make the hearts disappear
    for i in range(max_lives):
        globals()[f'heart_{i}'].hideturtle()

    #reassign the gameboard and answer to be blank, turn on tracer
    game_board = []
    answer = []
    wn.tracer(True)

    #restart the game
    screen_setup(cur_lev, t_in_r)
    block_or_x()
    reset_switch()


######################
# C. Setup only once per coding
######################

#Function for drawing the bar of the X's
#Adapted from a post at https://stackoverflow.com/questions/35834691/change-appearance-of-turtle/35837569#35837569
#Called within XCursors
def polyRectangle(turt, x, y, slant, length1, length2):
    turt.goto(x, y)
    turt.setheading(slant)

    turt.begin_poly()

    for count in range(2):
        turt.forward(length1)
        turt.left(90)
        turt.forward(length2)
        turt.left(90)

    turt.end_poly()

    return turt.get_poly()

#Function to create the red and black X turtle shapes, for use on the gameboard and on the bottom of the screen
#Adapted from a post at https://stackoverflow.com/questions/35834691/change-appearance-of-turtle/35837569#35837569
#Called presently at the top level, before screen_setup()
def XCursors():
    temporary = t.Turtle()
    screen = t.getscreen()

    delay = screen.delay()
    screen.delay(0)

    temporary.hideturtle()
    temporary.penup()


    X_turtle = t.Shape("compound")
    X_turtle_red = t.Shape("compound")

    left_side = polyRectangle(temporary, 0.35*bar_len, -0.375*bar_len, 45, bar_width, bar_len)  # left_side of X
    X_turtle.addcomponent(left_side, "black", "black")
    X_turtle_red.addcomponent(left_side, "red", "red")

    right_side = polyRectangle(temporary, 0.35*bar_len, 0.325*bar_len, 135, bar_width, bar_len)  #right side of X
    X_turtle.addcomponent(right_side, "black", "black")
    X_turtle_red.addcomponent(right_side, "red", "red")

    t.register_shape("X_turtle", X_turtle)
    t.register_shape("X_turtle_red", X_turtle_red)

    screen.delay(default_delay)
    del temporary

    for turt in t.turtles():
        turt.hideturtle()
```

```
######################
# D. Switch functions
######################

#Based on the parameter x_or_b, this function either circles the Block Button or the X button at the bottom of the
screen
#Called within block_or_x()
def circler_fun(x_or_b):
    if x_or_b == "block":
        wn.tracer(False)
        circler.clear()
        circler.penup()
        circler.goto(-15,switch_y-18-shift_for_hearts)
        circler.pendown()
        circler.circle(18)
        circler.penup()
        wn.tracer(True)
    if x_or_b == "x":
        wn.tracer(False)
        circler.clear()
        circler.penup()
        circler.goto(15,switch_y-18-shift_for_hearts)
        circler.pendown()
        circler.circle(18)
        circler.penup()
        wn.tracer(True)

#this function controls the value of the Boolean block_state and changes the look of the turtles set_x and
set_block at the bottom of the screen.
#called multiple times per level, via onkeypress ('x')
def block_or_x():
    global block_state
    if block_state==True:
        block_state=False
        set_block.shapesize(1)
        set_x.shapesize(t_switch_resize)
        circler_fun("x")

    else:
        block_state=True
        set_block.shapesize(t_switch_resize)
        set_x.shapesize(1)
        circler_fun("block")


######################
# E. Clicking and Check Functions
######################

'''clicking function
The brain of the game! This function controls everything that happens each time we click on one of the turtles on
the board

This function does several things:
        1) loops over all blank blocks on the board, only checking ones that have not yet been clicked (using
original color tuple)
        2) Finds which block the click is closest to by finding the distance from x and y to all blank block
coordinates
        3) once we've found the correct block, check if block_state is true. If so, turn the block blue! If not,
turn the block to x
        4) If blank block --> full block, set  game_board at that cell = 1; if blank block --> x, set game_board
at that cell = 0
        5) print the game_board
'''
def clicked(x,y):
    global block_state, turt_in_row, block_color_tuple, answer
    for j in range(turt_in_row):        #go through all rows
        for i in range(turt_in_row):    #go through all turtles within a row
            if globals()[f'block{i}_{j}'].color()==block_color_tuple:                       #only want to alter
unclicked (whitish-blue blocks)
                if abs(x-globals()[f'block{i}_{j}'].xcor()) < 15 and abs(y-globals()[f'block{i}_{j}'].ycor()) <
15:     #finding the block we've clicked near
                    if block_state==True:
                        globals()[f'block{i}_{j}'].color('blue')
                        globals()[f'game_board'][j][i]='1'              #change the gameboard at that spot to a
1, for filled

                    else:
                        globals()[f'block{i}_{j}'].shape("X_turtle")
                        globals()[f'game_board'][j][i]='0'              #change the gameboard at that spot to a
```

```
0, for x

                    #The following should happen whether block_state is true or not.
                    #It needs to happen every time we have clicked on a cell and changed it to an X or block.
                    check_cell(j,i)                                     #this should check that the cell has been
entered correctly
                    new_check_row(j)                                        #this should check that the whole row has
been entered correctly
                    new_check_col(i)                                        #this should check that the whole column
has been entered correctly
                    check_win()

def check_cell(current_row_num, current_col_num):
    if globals()['game_board'][current_row_num][current_col_num]== globals()['answer'][current_row_num]
[current_col_num]:
        pass
    else:
        if globals()['answer'][current_row_num][current_col_num] == '0':
            wn.delay(200)
            globals()[f'game_board'][current_row_num][current_col_num]='0'
            globals()[f'block{current_col_num}_{current_row_num}'].shapesize(1.5)
            globals()[f'block{current_col_num}_{current_row_num}'].shape("X_turtle_red")
        else:
            wn.delay(200)
            globals()[f'game_board'][current_row_num][current_col_num]='1'
            globals()[f'block{current_col_num}_{current_row_num}'].shapesize(1.5)
            globals()[f'block{current_col_num}_{current_row_num}'].color("red")
            globals()[f'block{current_col_num}_{current_row_num}'].shape("square")

        wn.delay(200)
        globals()[f'block{current_col_num}_{current_row_num}'].shapesize(1)
        wn.delay(default_delay)
        lose_a_life()

def new_check_row(current_row_num):
    global turt_in_row

    '''This variable will be used to determine how many 1's (blocks) are currently filled in on the gameboard.
    If this variable matches the row_sum, then the row is correct.'''
    count_1s_in_gameboard = 0

    #This loop is getting the value of count_1s_in_gameboard. Do count when the 1's match.
    #Don't count when there is a 1 in answer, but a - in gameboard.
    for i in range(turt_in_row):
        if globals()[f'game_board'][current_row_num][i] == '1' and globals()['answer'][current_row_num][i] ==
'1':
            count_1s_in_gameboard += 1

        # if our gameboard is a - and our answer board is a 1 in a cell, do nothing.
        # This means that not the user has not yet filled in the correct number of blocks.
        elif globals()[f'game_board'][current_row_num][i] == '-' and globals()['answer'][current_row_num][i] ==
'1':
            break

    '''if we don't yet have all the 1's in our game_board matching the 1's in answer board, do nothing.
    If they do match (and you're not dead)
    a) change every - in gameboard to 0
    b) change every blank turtle to the x turtle
    c) make the turtles dance '''
    if count_1s_in_gameboard != row_sums[current_row_num]:
        pass
    else:
        #only fix the rest of the row and make the turtles dance if you're not dead.
        if current_lives != 0:
            for i in range(turt_in_row):
                turt_dance(globals()[f'block{i}_{current_row_num}'])
                if globals()[f'game_board'][current_row_num][i] == '-': #change the blanks to x's
                    globals()[f'block{i}_{current_row_num}'].shape("X_turtle")
                    globals()[f'game_board'][current_row_num][i]='0'              #change the gameboard at that
spot to a 0, for x

def new_check_col(current_col_num): #I needed a new function, now that the cells are checked for correctness
    global turt_in_row

    '''This variable will be used to determine how many 1's (blocks) are currently filled in on the gameboard.
    If this variable matches the col_sum, then the col is correct.'''
    count_1s_in_gameboard = 0

    #This loop is getting the value of count_1s_in_gameboard. Do count when the 1's match.
    #Don't count when there is a 1 in answer, but a - in gameboard.
    for j in range(turt_in_row):
```

```
        if globals()[f'game_board'][j][current_col_num] == '1' and globals()['answer'][j][current_col_num] ==
'1':
            count_1s_in_gameboard += 1

        #if our gameboard is a - and our answer board is a 1, do nothing. not yet answered.
        elif globals()[f'game_board'][j][current_col_num] == '-' and globals()['answer'][j][current_col_num] ==
'1':
            break

    '''if we don't yet have all the 1's in our game_board matching the 1's in answer board, do nothing.
    If they do match (and you're not dead)
    a) change every - in gameboard to 0
    b) change every blank turtle to the x turtle
    c) make the turtles dance '''
    if count_1s_in_gameboard != col_sums[current_col_num]:
        pass
    else:
        #only fix the rest of the row and make the turtles dance if you're not dead.
        if current_lives != 0:
            for j in range(turt_in_row):
                turt_dance(globals()[f'block{current_col_num}_{j}'])
                if globals()[f'game_board'][j][current_col_num] == '-': #change the blanks to x's
                    globals()[f'block{current_col_num}_{j}'].shape("X_turtle")
                    globals()[f'game_board'][j][current_col_num]='0'              #change the gameboard at that
spot to a 0, for x

def check_win():
    global win, current_level
    if game_board == answer:
        win_level()

#this function will make a given turtle dance.
#This will be called when the row/ column has been correctly solved.
def turt_dance(turt):
    turt.speed(6)
    turt.setheading(90)
    turt.forward(30)
    turt.back(30)

def lose_a_life():
    global current_lives

    globals()[f'heart_{current_lives - 1}'].shape(heart_img_empty) #if we have 3 lives, heart_2 must be changed
    current_lives -= 1

    if current_lives == 0:
        game_over()


######################
# F. Debugging Functions
######################

#prints whatever game_board is input into this function (used to print both the current game_board and the
answer_board)
#Function created so that the output would not be a list of lists. Also, so the top of the output matched the top
of the board.
def print_game_board(game_board):
    for row in range(len(game_board)-1,-1,-1):
        print (' '.join(game_board[row]) )

######################
# G. WIN/ LOSE/ GAMEPLAY FUNCTIONS
######################

# Should turn off wn.onkeypress and turtle.onclick() functionality. (Note, it will not disable wn.listen()
# it will merely reassign the click and key 'x' to the function None).
# See https://stackoverflow.com/questions/36924609/python-turtle-stop-listening and
# https://docs.python.org/3/library/turtle.html#turtle.onclick under "events"
def turn_off_turtle_events():
    global turt_in_row
    wn.onkeypress(None, 'x')
    for j in range(turt_in_row):
        for i in range(turt_in_row):
            globals()[f'block{i}_{j}'].onclick(None)

#Some of the functionality, like incrementing current_level, should be handled by tkinter screen, eventually
def win_level():
    global current_level, max_levels
    turn_off_turtle_events()
    print('YOU WON!')
```

```
    if current_level != max_levels:
        current_level += 1 #but the incrementing needs to happen later (via the button on win_screen)
        choice = input('do you want to play the next level? "y" or "n"\n')
        if choice == "y":
            play_one_level() #should happen via a button, eventually
    else:
        #Still need to finish this function
        win_whole_game()

def win_whole_game():
    global max_levels
    print(f'you beat all {max_levels} levels! Yay for you!\n\n')

def game_over():
    global lost
    lost = True
    turn_off_turtle_events()
    print('game over.')
    print(f"You have two options:\n\n1. Replay the exact level you just played, but with {lives_reduction} less
lives. \
\n2. Play a brand new level #{current_level}, with all your lives back.")
    choice = int(input(f'Type 1 for the first option, 2 for the second option.   '))
    if choice==1:
        reset_current_level()
    elif choice==2:
        play_one_level()

def play_one_level():
    global current_level, level_list, screen_setup_needed, lost

    select_diff_this_level()
    if lost ==True:
        lost = False

    if current_level==1 and screen_setup_needed:
        screen_setup(current_level, level_list[current_level-1])
        screen_setup_needed = False
    else:
        clear_screen_and_restart(current_level, level_list[current_level-1])

    turn_on_clicks_and_switch(level_list[current_level-1])


######################
#H. Before gameplay, difficulty/ mode selections
######################

#Called at the beginning of every level, but only does something if the user opted at the beginning of the game to
select difficulty before each level.
def select_diff_this_level():
    global select_diffs_per_level, current_level, level_list
    diff_per_level_message = f'''
You have four options of difficulty for level {current_level}:
When prompted, please type the letter of your choice. \n\n
Easy\t\t(5 x 5 puzzle)\t--type "e"
Medium\t\t(10 x 10 puzzle)--type "m"
Hard\t\t(15 x 15 puzzle)--type "h"
Very Hard\t(20 x 20 puzzle)--type "v"\n'''

    if not(select_diffs_per_level):
        pass
    elif lost:
        pass
    else:
        print(diff_per_level_message)
        choice = input(f'\tPlease type your difficulty selection for level {current_level}:\n\t')
        if choice == "e":
            level_list.append(5)
        elif choice == "m":
            level_list.append(10)
        elif choice == "h":
            level_list.append(15)
        elif choice == "v":
            level_list.append(20)
        else:
            print("You have not made a valid selection")
            select_diff_this_level()

def welcome_and_settings():
    global max_levels
```

```
    #part of the welcome message was from this website: https://swadge.com/super2023/picross/
    welcome_message = '''
Welcome to Nonograms!!

Nonograms, commonly known as "Picross", is a puzzle game in the family of Sudoku. The objective is to correctly
fill in the grid according to
the clues. Spaces will either be empty or filled, and when all of the squares are correctly set, you will have won,
revealing the picture.

In a typical picross game, you are trying to reveal an image. In this version, however, the "answers" are all
randomly generated. Also,
this version has 4 difficulties:

Easy (5 x 5 grid)
Medium (10 x 10 grid)
Hard (15 x 15 grid)
Very Hard (20 x 20 grid)

Once the game window pops up, use the numbers along the top and left of the screen to tell you how many spaces to
fill. You will click on each
of the squares to set them as blocks or X's. You will also be able to type 'x' on the keyboard to switch between
laying blocks and X's on the
nonograms board.
But before we begin, we need some information from YOU...
'''
    difficulty_message = '''
\nNow that you've selected the number of levels to play, the next thing to do is select your difficulty mode.

1. Random difficulties--the game will randomly select the difficulty of each level for you.
2. Increasing difficulties--the game will make the levels get harder as you go.
(Remember, there are only 4 difficulty levels: Easy, Medium, Hard, or Very Hard)
3. Pick difficulty per level--this puts YOU in the driver seat! You will be prompted to select the difficulty of
each level before it begins.

'''

    print(welcome_message)
    #Where I found this while True loop to validate user input: https://stackoverflow.com/questions/70733583/how-
would-i-type-a-string-in-an-int-input-without-getting-an-error-in-python
    while True:
        try:
            max_levels = int(input('\tTo begin, please enter an integer (at least 1) to represent the number of
levels you\'d like to play\n\t'))
        except ValueError:
            print("You did not input a valid integer")
        else:
            if max_levels <= 0:
                print("Try again, your number must be greater than or equal to 1.")
            else:
                break
    print("You've chosen to play", max_levels, "levels today.")
    print(difficulty_message)
    while True:
        try:
            choice_diff_mode = int(input('\tPlease enter the number of your selection for difficulty mode.\n\t'))
        except ValueError:
            print("You did not input a valid integer")
        else:
            if choice_diff_mode not in [1,2,3]:
                print("Try again, you must select 1, 2, or 3.")
            else:
                break
    if choice_diff_mode == 1:
        random_diffs(max_levels)
    elif choice_diff_mode == 2:
        increasing_diffs(max_levels)
    elif choice_diff_mode == 3:
        diffs_per_level()
    print("Now, we begin with level 1!")
    play_one_level()

def random_diffs(m_levels):
    global level_list
    level_list = [random.choice([5,10,15,20]) for level in range(m_levels)]

def increasing_diffs(m_levels):
    global level_list

    if m_levels == 3:                   #When testing, m_levels = 3 made the list [10,15,20]. I wanted there to
always be at least 1 easy level.
        level_list = [5,10,15]
```

```
    elif m_levels == 6:
        level_list = [5,10,10,15,15,20] #When testing, m_levels = 6 made the list [10,10,15,15,20,20]. I wanted
there to always be at least 1 easy level.
    else:
        num_levels_per_diff = round(m_levels/ 4) #we want to basically divide the number of total levels by 4, to
approximately get an equal number of levels for each diff
        num_easy_levels = m_levels - 3*num_levels_per_diff  #we want all the "rest" of the levels to be easy
levels. (if we select 1 or 2 levels, they will be easy.)
        for i in range(num_easy_levels):
            level_list.append(5)


        for j in [10,15,20]:
            for i in range(num_levels_per_diff):
                level_list.append(j)

def diffs_per_level():
    global select_diffs_per_level
    select_diffs_per_level = True


################################################
###          III. Events--Main code          ###
################################################
XCursors()

welcome_and_settings()
wn.listen()

wn.mainloop()
```