**Kehron Chambers**

Programming Fundamentals Assignment 1

**Introduction**

This report details my experience and journey in the process of learning python to develop a game of Sudoku. Due to the circumstances of the sudden coronavirus pandemic, it proved to be a bit more difficult to produce this program but in this report my aim is to detail the trails I went through to complete the assignment despite this, as well as the time constraints and knowledge I had, considering I was recently self-taught on the language.

I used Python 3.6 IDLE to code core of the program and used pygame to create the GUI. I chose to go with pygame as I felt it was easy to make a simple GUI within the time constraints that I had.

The overall task I had to undergo was creating a simple game in python of my choice and I decided to go with Sudoku. This is because I enjoy Sudoku a lot and felt that in turn I would find it enjoyable to code.

**Assignment**

**Sudoku.py**

def findEmptyBox(box):#empty square in the boardd

    for a in range(len(box)):#loops through board

        for b in range(len(box[a])):#length of each row##

            if box[a][b] == 0:#checks if position is 0

                return (a, b)  # row, col

Within my program I have two separate python files connected together with programming logic. The first one Sudoku.py main purpose is to code the core framework of the program. The first subroutine above showcases how the computer will search through and find empty boxes. This is done by looping through the board and checks to see if the value of square is equal to 0. If it is it returns the value.

def solver(box):#uses recurrsion

    search = findEmptyBox(box)

    if not search:

        return True#solution is found and board is full

    else:

        row, col = search#else user can keep inputting values into board

    for a in range(1,10):#loops through values that are being used for solution

        if valid(box, a, (row, col)):#if valid

            box[row][col] = a#input is placed in board

            if solver(box):

                return True#sub is constantly called until finished

            box[row][col] = 0#if valid is incorrect we backtrack and reset the value and repeat process

    return False#if input is invalid

The solver subroutine uses recursion and backtracking to solve the Sudoku game to create a model so the computer has knowledge of what values go where and whether or not the grid is solvable. Backtracking goes through and picks an empty box, tries all numbers until they find one that works and repeats. If a number doesn’t work, then they will backtrack and try again. This approach made the most sense as it would load the fastest and would be most efficient, in comparison to trying every combination on the board until you find one that works.

The search variable calls the findEmptyBox subroutine so it can go through the grid and find an empty square and once it does it loops through the values of 1-9 until they find a value that is correct and puts it in and keeps doing that until finished. False is returned if value is incorrect.

def valid(box, num, pos):#checks to see if the current board is valid

    for a in range(len(box[0])):#loops through each column in the row

        if box[pos[0]][a] == num and pos[1] != a:

            return False#if entered number is repeated return false

    for a in range(len(box)):#loops through each row in column

        if box[a][pos[1]] == num and pos[0] != a:

            return False#if entered number is repeated return false

    # Checking box using integer position

    row\_boxes = pos[1] // 3#integer divide row

    column\_boxes = pos[0] // 3#integer divide column

    for a in range(column\_boxes\*3, column\_boxes\*3 + 3):#loop through all nine elements in box

        for b in range(row\_boxes \* 3, row\_boxes\*3 + 3):#look through the values in each grid

            if box[a][b] == num and (a,b) != pos:#check if repeated numbers are in box

                return False#returns false if duplicate is found

    return True#if checks are passed then true is returned

    return None#no blank squares means we're done

This subroutine checks to see if the current board being used is valid and solvable. If any number looped through each row and column is repeated, then False is returned. Using integer division, addition and multiplication, we loop through each element to check to see if any numbers are repeated. If they aren’t True is returned but if they are repeated, then False is returned. Return none at the end lets python know that check is finished.

**GUI.py**

import pygame

from Sudoku import solver, valid#importing code file into gui file and other values

import time

pygame.font.init()

In this GUI program, pygame is imported, and solver and valid subroutines are imported from the Sudoku file. Time is also imported so that the user can track how long he has been playing for.

class Grid:#grid class

    #grid of values that start on board

    #to change starting board use this grid

    board = [

        [4, 2, 0, 1, 7, 0, 3, 9, 0],

        [0, 7, 8, 0, 0, 0, 0, 5, 0],

        [0, 0, 0, 8, 0, 0, 2, 0, 7],

        [0, 5, 0, 0, 0, 7, 0, 3, 0],

        [3, 0, 0, 2, 9, 6, 0, 0, 5],

        [0, 1, 0, 5, 0, 0, 0, 7, 0],

        [7, 0, 1, 0, 0, 8, 0, 0, 0],

        [0, 6, 0, 0, 0, 0, 9, 4, 0],

        [0, 9, 3, 0, 2, 4, 0, 8, 1]

    ]

So starting off, we make a class to design the grid and a board for the Sudoku game. In the board, you can hard code the grid to create a wireframe to keep it challenging and entertaining.

    def \_\_init\_\_(self, rows, cols, width, height):#attributes

        self.rows = rows#rows

        self.cols = cols#columns

        self.cubes = [[Cube(self.board[i][j], i, j, width, height) for j in range(cols)] for i in range(rows)]#cube

        self.width = width#width

        self.height = height#height

        self.model = None#model

        self.selected = None#selected

    def update\_model(self):#board sent to solver to see if board can be solvable and has final values

        self.model = [[self.cubes[i][j].value for j in range(self.cols)] for i in range(self.rows)]

    def place(self, val):#sets permanent value

        row, col = self.selected

        if self.cubes[row][col].value == 0:

            self.cubes[row][col].set(val)

            self.update\_model()#if value of box is 0 then update to inputted value

            if valid(self.model, val, (row,col)) and solver(self.model):

                return True#if inputted value is same as model value then return true

            else:

                self.cubes[row][col].set(0)

                self.cubes[row][col].set\_temp(0)

                self.update\_model()

                return False#if else then return false and answer is wrong

    def sketch(self, val):#set temp value

        row, col = self.selected

        self.cubes[row][col].set\_temp(val)

Then after, we create the attributes for the elements on the grid, including the cubes, rows and columns. The board is sent to the other file for it to validate model and check to see if its solvable by filling out the grid. Then after we set conditions for permanent value inputs and create a subroutine for temporary value inputs.

    def draw(self, win):

        #draws grid lines

        gap = self.width / 9

        for i in range(self.rows+1):

            if i % 3 == 0 and i != 0:

                thick = 4

            else:

                thick = 1

            pygame.draw.line(win, (0,0,0), (0, i\*gap), (self.width, i\*gap), thick)

            pygame.draw.line(win, (0, 0, 0), (i \* gap, 0), (i \* gap, self.height), thick)

        #draws cubes

        for i in range(self.rows):

            for j in range(self.cols):

                self.cubes[i][j].draw(win)

    def select(self, row, col):#select desired square

        #reset all other

        for i in range(self.rows):

            for j in range(self.cols):

                self.cubes[i][j].selected = False

        self.cubes[row][col].selected = True

        self.selected = (row, col)

    def clear(self):#lets you remove a temp entered number

        row, col = self.selected

        if self.cubes[row][col].value == 0:

            self.cubes[row][col].set\_temp(0)

    def click(self, pos):#return position on cube clicked on

        """

        :param: pos

        :return: (row, col)

        """

        if pos[0] < self.width and pos[1] < self.height:

            gap = self.width / 9

            x = pos[0] // gap

            y = pos[1] // gap

            return (int(y),int(x))

        else:

            return None

    def is\_finished(self):#checks if no empty squares on board

        for i in range(self.rows):

            for j in range(self.cols):

                if self.cubes[i][j].value == 0:

                    return False

        return True

Clear subroutine allows you to remove temporarily entered numbers, and click returns position of cube. The ‘is finished’ subroutine checks if there are any empty squares on the board. If there are still spaces, then false is returned, but if there isn’t then true is returned.

class Cube:#cube class

    rows = 9

    cols = 9

    def \_\_init\_\_(self, value, row, col, width ,height):

        self.value = value#set value inputted

        self.temp = 0#tempoary value

        self.row = row#row

        self.col = col#column

        self.width = width#width

        self.height = height#height

        self.selected = False#selected set as false

    def draw(self, win):#drawing values

        fnt = pygame.font.SysFont("comicsans", 40)#font of values

        gap = self.width / 9

        x = self.col \* gap

        y = self.row \* gap#sets the gap between the values

        if self.temp != 0 and self.value == 0:

            text = fnt.render(str(self.temp), 1, (128,128,128))

            win.blit(text, (x+5, y+5))#will show temp value in box

        elif not(self.value == 0):

            text = fnt.render(str(self.value), 1, (0, 0, 0))#if value already there nothing will come up

            win.blit(text, (x + (gap/2 - text.get\_width()/2), y + (gap/2 - text.get\_height()/2)))

        if self.selected:

            pygame.draw.rect(win, (255,0,0), (x,y, gap ,gap), 3)#draws value in

    def set(self, val):

        self.value = val#sets value

    def set\_temp(self, val):

        self.temp = val#sets temp value

def redraw\_window(win, board, time, strikes):

    win.fill((255,255,255))

    # Draw time

    fnt = pygame.font.SysFont("comicsans", 40)

    text = fnt.render("Time: " + format\_time(time), 1, (0,0,0))

    win.blit(text, (540 - 160, 560))

    # Draw Strikes

    text = fnt.render("X " \* strikes, 1, (255, 0, 0))

    win.blit(text, (20, 560))

    # Draw grid and board

    board.draw(win)

def format\_time(secs):#allows user to see how long he has spent

    sec = secs%60

    minute = secs//60

    hour = minute//60

    mat = " " + str(minute) + ":" + str(sec)

    return mat

This class ‘Cube’ designs the look of each grid within the board. It uses attributes to design it and pygame functions to help draw the values and time. Conditionals are used to show temp values in box and if there’s already a value within a box then nothing will come up. Strikes, grid and board are also drawn. The format time function allows the user to view how long he’s been attempting the game for and lay it out well.

def main():#main

    win = pygame.display.set\_mode((540,600))#window size

    pygame.display.set\_caption("Sudoku")#name of window

    board = Grid(9, 9, 540, 540)#board dimensions

    key = None#key starts with no value

    run = True#when program starts run is set to true

    start = time.time()#timer begins

    strikes = 0#incorrect answers are counted

    while run:

        play\_time = round(time.time() - start)#shows playtime

        for event in pygame.event.get():

            if event.type == pygame.QUIT:

                run = False#if user quits program is closed and run is set to false

            if event.type == pygame.KEYDOWN:#depending on what key is pressed is what will be displayed in program

                if event.key == pygame.K\_1:

                    key = 1

                if event.key == pygame.K\_2:

                    key = 2

                if event.key == pygame.K\_3:

                    key = 3

                if event.key == pygame.K\_4:

                    key = 4

                if event.key == pygame.K\_5:

                    key = 5

                if event.key == pygame.K\_6:

                    key = 6

                if event.key == pygame.K\_7:

                    key = 7

                if event.key == pygame.K\_8:

                    key = 8

                if event.key == pygame.K\_9:

                    key = 9

                if event.key == pygame.K\_DELETE:

                    board.clear()#delete will clear entry

                    key = None

                if event.key == pygame.K\_RETURN:

                    i, j = board.selected#confirm entered value

                    if board.cubes[i][j].temp != 0:

                        if board.place(board.cubes[i][j].temp):

                            print("Success")#if temp value is correct it will be permanent and this is displayed

                        else:

                            print("Wrong")

                            strikes += 1#if temp value is incorrect wrong is printed and a strike value is incrimented

                        key = None#after a key value is entered and submitted it returns back to 0 waiting for new value

                        if board.is\_finished():

                            print("Game over")

                            print(time.time())

                            run = False#when game is finished game over is printed and program closes

            if event.type == pygame.MOUSEBUTTONDOWN:

                pos = pygame.mouse.get\_pos()

                clicked = board.click(pos)

                if clicked:

                    board.select(clicked[0], clicked[1])

                    key = None#if u click a box its highlighted

        if board.selected and key != None:

            board.sketch(key)#if u click a box with a value already inside of it

        redraw\_window(win, board, play\_time, strikes)

        pygame.display.update()

main()

pygame.quit()

The main subroutine is where majority of the program takes place. Using pygame, we create the windows, display and board. They key value is set at none as we start with no value, run is set to true as an indication of the program running. Start sets up the timer and strikes begin to get counted.

While run records play time and creates conditionals for each key to output a value in the game. For example, if 5 is pressed, then it will be placed as a value. If it is a correct value, then ‘success’ is printed and if its incorrect then ‘wrong’ is printed and the strike counter is incremented. Also if the game is over then ‘game over’ is displayed, run is set to false and the time is displayed.

**Testing**

I followed a tutorial and worked on sections one by one and testing to see what works with what, and changed things to fit to my program more accordingly. This went from using different methods of loops, removing unnecessary subroutines and renaming variables so they were easier to identify to help make amendments to the code whenever I chose to come back to it.

After running a few tests and making amendments there were times where I encountered opportunities where I wanted to add more features to the game but was unable to due to time constraints, the ongoing situation, and my limited knowledge. This included, multiple difficulties, the ability for the computer to randomise the Sudoku board each game, and displaying a congrats screen when the game is completed with accurate final time, rather than displaying it in the console.

**Referencing**

<https://www.pygame.org/wiki/tutorials> - Was used to help understand the fundamentals of pygame.

<https://www.youtube.com/watch?v=eqUwSA0xI-s&t=901s> – was used as my main source and guideline behind the assignment but still made sure to modify and incorporate my own knowledge and other resources where possible.

<https://www.youtube.com/watch?v=r_cmJBgrq5k> – was also used as an insight in how the coding behind a Sudoku game would function.