Documentation

Snake Game

Ankur Bohra (11-F)

28th September, 2020

Certificate of Originality

This is to certify that this project documentation paper submitted by me, **Ankur Bohra**, is an outcome of my independent and original work. I have duly acknowledged all the sources from which the ideas and extracts have been taken. The project is free from any plagiarism and has not been submitted elsewhere for publication.

Name of author: Ankur Bohra
Date: 28th November, 2020
Signature:

Name of teacher: Gulroop Kaur
Date: 28th November, 2020
Signature:

System Overview

Hardware and Software Requirements

(I) Hardware

S. No.	Item	Purpose
1	English keyboard	Game movement
2	Pointing device	Interface navigation

(II) Software

S. No.	Item	Version	Installation	Purpose
1	Python	3.8.2	www.python.org /downloads/	Code interpreter
2	Tkinter	8.6	Installed with python	Interface construction

Project Synopsis

(I) Theme:

A classic snake game written in python implemented via tkinter.

(II) Scope

The project is written in a single player scope, it implements collision with various objects and retains scores.

(III) Assumptions

1. The player does not enter multiple inputs in one “frame”/step i.e. input in a single step is final and never overwritten.
2. The 4 programs main.py, core.py, interface.py, settings.py and one JSON file scores.json are placed in a folder “src” residing inside another folder. The entry program main.py is assumed to be run through the command line from directory src/..; e.g. Snake Game > src > *.py*.json is executed from Snake Game: `python src/main.py`.

(IV) Enhancements

1. Clean program environment/globals.
2. Allow users to modify settings.
3. Develop more independent functions.

Packages and Functions

Packages

S. No.	Package	Version	Installation	Purpose
1	Tkinter	8.6	Installed with python	Interface construction

Functions

(I) core.py

S. No.	Function	Purpose
1	void makegrid()	Makes the 2D matrix the game operates by. Each cell in this grid represents a slot, holding one frame and the object associated with the slot.
2	void reset()	Resets all game-dependent variables for the next run.
3	dict cell makecell(str cell_type , dict data)	Makes the cell object, makes

		object creation cleaner.
4	void occupy(dict cell)	Given a cell object, occupies a grid slot and reflects the changes.
5	dict cell neighbours(dict cell)	Finds the non-diagonal neighbours of a cell, adjusted for edge/corner cases.
6	void die()	Changes game state and initiates flow back to the menu.
7	void makefood()	Makes a weighted random food type in a random position.
8	void makesnake()	Creates the snake and sets up the cell data.
9	void makepowerup()	Creates a random powerup in a random position.
10	dict cell collision(dict snake , dict obj)	Checks collision cells and acts accordingly.
11	tuple randomdir defaultdir(dict snake)	Makes the cell object, makes object creation cleaner.
12	void movesnake(dict snake)	Fills and empties cells based on the direction the snake is moving in.
13	void movebind(keysym key)	Updates the movement direction based on player input.
14	void startlife(frame window)	Makes the cell object, makes object creation cleaner.
15	void playgame(frame window, int mode_step, function passed_ender)	Starts one play session.

(II) interface.py

S. No.	Function	Purpose
1	Tk window makegrid()	Makes the main game window, created only once.
2	void clearscreens()	Destroys all screens, called before making a new one. Screen frames are not stored indefinitely, they are created on demand.
3	void endgame(int mode_step , int score)	Shows the death interface. Updates the score after awaiting for activity response.
4	void displaycredits(frame screen)	Shows creator credits on any given screen (omitted in game).
5	void displaymodes()	Shows game mode/difficulty screen. Game is started with the respective step from here.
6	void displayscore()	Shows stored scores sorted by mode and rank. Implements player filtered scores.
7	void displaymenu()	Shows main menu, links all screens together.

(III) main.py

No functions

(IV) settings.py

No functions

(V) scores.json

Not a program

Code

core.py

```
'''
Core game and only required interface
'''
import random
from tkinter import *
from settings import *


# Globals
game = None
end_game = None # Pass down score update function from interface
score = {"frame": None, "value": 0}
step = 300

# Snake state
state = "PLAYING"
movedir = (0, 0)
powerups = {}


# Grid formation
grid = []
def makegrid():
    '''
    Makes the 2D matrix the game operates by.
    Each cell in this grid represents a slot, holding one frame and the object associated with the slot.
    '''
    global game
    for y in range(GRID_SIZE):
        row = []
        for x in range(GRID_SIZE):
            frame = Frame(game, bg="black", height=CELL_SIZE,
width=CELL_SIZE)
            frame.place(relx=x/GRID_SIZE, rely=y/GRID_SIZE)
            slot = {"frame":frame, "object":None}
            row.append(slot)
        grid.append(row)


# Utility
def reset():
    '''
    Resets all game-dependent variables for the next run.
    '''
    global grid, game, end_game, score, state, movedir
    grid = []
    movedir = (0, 0) # So the next life find movedir again
    game, end_game = None, None
    score["value"] = 0

def makecell(cell_type, pos, data):
    '''
    Makes the cell object, makes object creation cleaner
    '''
    cell = {"type": cell_type, "pos":pos}
    cell.update(data)
    return cell

color_map = {
        "snakehead": HEAD_COLOR,
        "snakebody": BODY_COLOR,
        "free": GRID_COLOR
    }
```

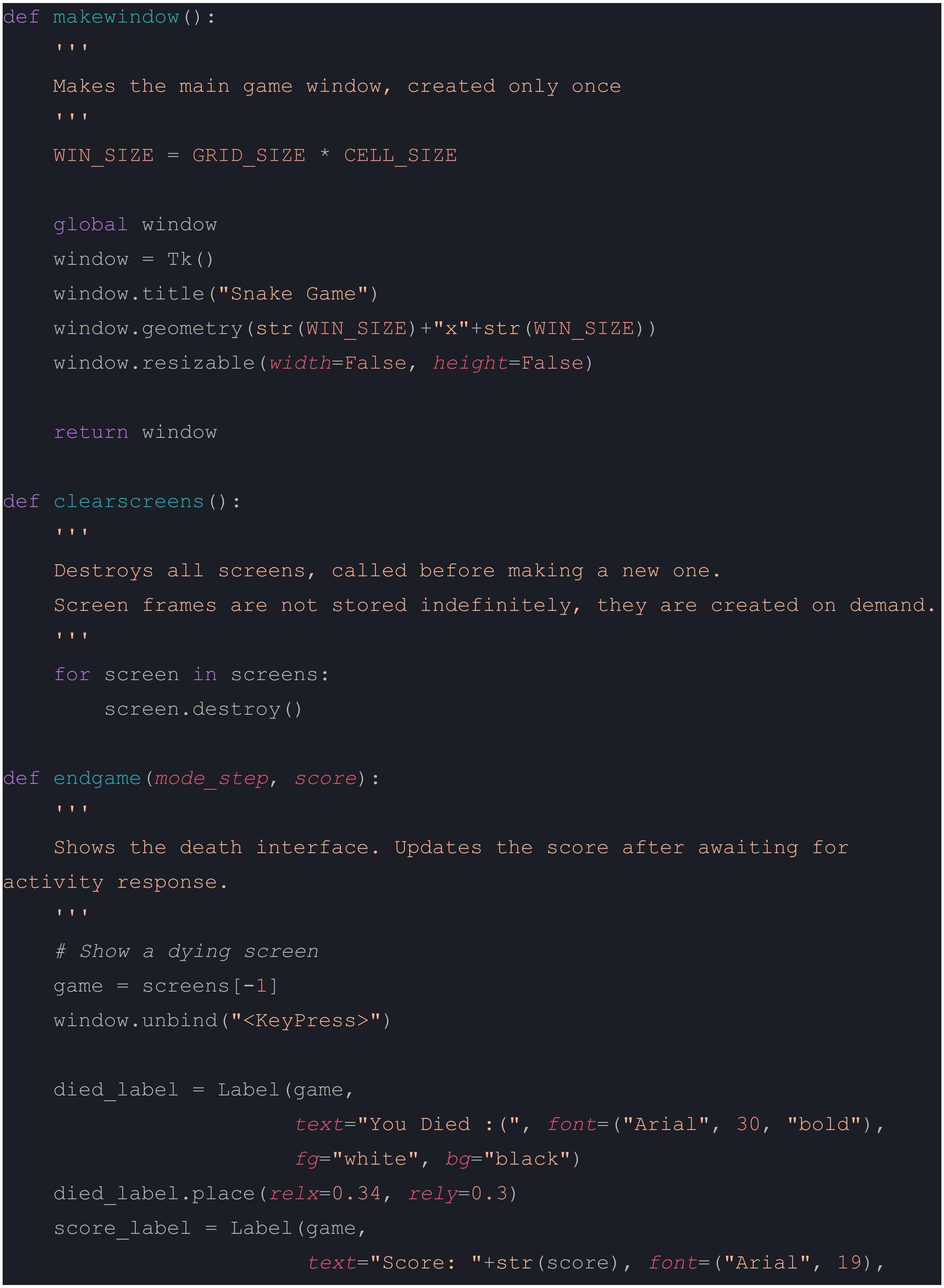
```
for index in range(len(FOOD_POINTS)):
    points = FOOD_POINTS[index]
    color = FOOD_COLORS[index]
    name = "food"+str(points)
    color_map.update({name:color})
for index in range(len(POWERUP_TYPES)):
    powerup_type = POWERUP_TYPES[index]
    color = POWERUP_COLORS[index]
    name = "powerup"+str(powerup_type)
    color_map.update({name:color})
def occupy(cell):
    '''
    Given a cell object, occupies a grid slot and reflects the changes.
    '''
    if grid != [] and state == "PLAYING":
        cellx, celly = cell["pos"][0], cell["pos"][1]

        bg = color_map[cell["type"]]

        slot = grid[celly][cellx]
        slot["frame"].config(bg=bg)
        if slot["object"] and not
(slot["object"]["type"].startswith("snake") or slot["object"]["type"] ==
"free"):
            slot["object"]["label"].destroy()
        slot["object"] = cell

        if cell["type"].startswith("food"):
            pts = cell["points"]
            pts_label = Label(slot["frame"], text=pts, bg=bg,
font=("Arial", 10), fg="black")
            pts_label.place(relx=0.5, rely=0.5, anchor=CENTER)
            slot["object"]["label"] = pts_label
        elif cell["type"].startswith("powerup"):
            powerup_type = cell["type"][7:]
            if powerup_type == "boost":
                text = "+⚡"
            elif powerup_type == "multiplier":
                text = "+"+str(cell["value"])+"x"
            powerup_label = Label(slot["frame"], text=text, bg=bg, font=("Arial", 10), fg="black")
            powerup_label.place(relx=0.5, rely=0.5, anchor=CENTER)
            slot["object"]["label"] = powerup_label

def neighbours(cell):
    '''
    Finds the non-diagonal neighbours of a cell, adjusted for edge/corner cases.
    '''
    pos = cell["pos"]
    cellx = pos[0]
    celly = pos[1]
    cell_neighbours = []
    if cellx in [0, GRID_SIZE - 1] and celly in [0, GRID_SIZE - 1]:
        # Some corner
        cell_neighbours.append((1 if cellx == 0 else GRID_SIZE - 2, celly))
        cell_neighbours.append((cellx, 1 if celly == 0 else GRID_SIZE - 2))
    elif cellx in [0, GRID_SIZE - 1]:
        cell_neighbours.extend([(cellx, celly - 1), (cellx, celly + 1)]) # Above, below
        cell_neighbours.append((cellx + 1 if cellx == 0 else cellx - 1, celly)) # Left, right
    elif celly in [0, GRID_SIZE - 1]:
        cell_neighbours.extend([(cellx - 1, celly), (cellx + 1, celly)]) # Left, right
        cell_neighbours.append((cellx, celly + 1 if celly == 0 else celly - 1)) # Above, below
    else:
        # Not on edge
        cell_neighbours.extend([(cellx, celly - 1), (cellx, celly + 1)]) # Above, below
        cell_neighbours.extend([(cellx - 1, celly), (cellx + 1, celly)]) # Left, right
    return cell_neighbours

def die():
    '''
    Changes game state and initiates flow back to menu
    '''
    global state, step, score
    state = "DEAD"
    end_game(step, score["value"])
    reset()

# Generators
def makefood():
    '''
    Makes a weighted random food type in a random position
    '''
    global state
    if state != "PLAYING":
        return

    foodx, foody = (random.randint(0, GRID_SIZE - 1), random.randint(0, GRID_SIZE - 1))
    while grid[foody][foodx]["object"]:
        foodx, foody = (random.randint(0, GRID_SIZE - 1), random.randint(0, GRID_SIZE - 1))
    food_pos = (foodx, foody)

    weighted_points = []
    for index in range(len(FOOD_POINTS)):
        points = FOOD_POINTS[index]
        probability = FOOD_RARITY[index]

        weighted_points.extend([points] * int(probability * 10)) # total weight = 10
    points = random.choice(weighted_points)
    food = {"type":"food"+str(points), "pos":food_pos, "points":points,
"label": None}

    occupy(food)

def makesnake():
    '''
    Creates the snake and sets up the cell data.
    '''
    snake = {
        "head":[],
        "body":[],
        "tail":[],
        "positions":[],
        "cells_occupied": 0
    }
    head_pos = (random.randint(0, GRID_SIZE - 1), random.randint(0,
GRID_SIZE - 1))
    head_cell = makecell("snakehead", head_pos, {"next": None,
"previous":None})
    occupy(head_cell)
    snake.update({"head":head_cell, "cells_occupied":1,
"positions":[head_pos]})

    while snake["cells_occupied"] < MIN_LENGTH:
        if snake["cells_occupied"] == 1:
            latest = snake["head"]
        else:
            latest = snake["body"][-1]

        pos = random.choice(neighbours(latest))
        while pos in snake["positions"]:
            pos = random.choice(neighbours(latest))
        snake["positions"].append(pos)
        cell = makecell("snakebody", pos, {"next": latest,
"previous":None})
        occupy(cell)

        latest.update({"previous":cell})
        snake["cells_occupied"] += 1

        if snake["cells_occupied"] == MIN_LENGTH:
            snake["tail"] = cell
        else:
            snake["body"].append(cell)
            if snake["cells_occupied"] == 2:
                snake["head"]["previous"] = cell
    return snake

def makepowerup():
    '''
    Creates a random powerup in a random position.
    '''
    global state
    if state != "PLAYING":
        return

    powerupx, powerupy = (random.randint(0, GRID_SIZE - 1),
random.randint(0, GRID_SIZE - 1))
    while grid[powerupy][powerupx]["object"]:
        powerupx, powerupy = (random.randint(0, GRID_SIZE - 1),
random.randint(0, GRID_SIZE - 1))
    powerup_pos = (powerupx, powerupy)

    powerup_type = random.choice(POWERUP_TYPES)
    powerup_value = POWERUP_DATA[POWERUP_TYPES.index(powerup_type)]

    powerup = {"type":"powerup"+powerup_type, "pos":powerup_pos,
"value":powerup_value, "label": None}
    occupy(powerup)
```

```
# Movement
def collision(snake, obj):
    '''
    Checks collision cells   and acts accordingly.
    '''
    global state, step, game, grid, score, powerups
    if obj["type"].startswith("snake"):
        die()
    elif obj["type"].startswith("food"):
        score["value"] += obj["points"] * powerups["multiplier"]
        score["label"].config(text="Score: "+str(score["value"]))

        # Extend snake from TAIL
        tail = snake["tail"]
        pos = tail["pos"]
        pos1 = tail["next"]["pos"]
        new_pos = (-(pos[0] - pos1[0]), -(pos[1] - pos1[1]))
        new_tail = makecell("snakebody", new_pos, {"next": tail,
"previous":None})

        snake["tail"] = new_tail
        tail["previous"] = new_tail
        snake["body"].append(tail)

        # Make new food but after a delay
        gen_step = random.randint(FOOD_GEN_STEP_MIN, FOOD_GEN_STEP_MAX)
        game.after(gen_step, makefood)
    elif obj["type"].startswith("powerup"):
        powerup_type = obj["type"][7:]
        powerups[powerup_type] += obj["value"]

        index = POWERUP_TYPES.index(powerup_type)
        duration = POWERUP_DURATIONS[index]
        powerups["active"] += 1

        count = 0
        for position in range(len(powerups["occupants"].keys())):
            if position in powerups["occupants"].keys():
                occupant = powerups["occupants"][position]
                if occupant["type"] == powerup_type:
                    count = position
            else:
                count = position

        if count not in powerups["occupants"].keys():
            powerups["occupants"][count] = {"n":0, "type": powerup_type}
        powerups["occupants"][count]["n"] += 1

        frame = Frame(game, bg=POWERUP_COLORS[index], height=7,
width=GRID_SIZE*CELL_SIZE + 3)
        frame.place(x=0, y=GRID_SIZE*CELL_SIZE - count * 7, anchor=SW)

        game.update()
        width = GRID_SIZE*CELL_SIZE + 3
        change_step = int(duration/width)
        def changesize():
            if frame.winfo_exists() and state == "PLAYING":
                width = frame.winfo_width()
                width -= 1
                frame.config(width=width)
                game.after(change_step, changesize)
        changesize()

        def revert():
            if state == "PLAYING":
                # Revert values, show it
                powerups[powerup_type] -= obj["value"]
                powerups["active"] -= 1
                powerups["occupants"][count]["n"] -= 1
                frame.destroy()

                # Make new powerup but after a delay
                gen_step = random.randint(POWERUP_GEN_STEP_MIN,
POWERUP_GEN_STEP_MAX)
```

```
                game.after(gen_step, makepowerup)

        game.after(duration, revert)

def defaultdir(snake):
    '''
    Gets the starting direction to move the snake in so it doesn't immediately die.
    '''
    neighbour_pos = neighbours(snake["head"])
    for pos in neighbour_pos:
        if pos in snake["positions"]:
            neighbour_pos.remove(pos)

    next_pos = random.choice(neighbour_pos)
    pos = snake["head"]["pos"]
    randomdir = (next_pos[0] - pos[0], - (next_pos[1] - pos[1])) # Y axis is flipped
    return randomdir

def movesnake(snake):
    '''
    Fills and empties cells based on the direction the snake is moving in.
    '''
    global state
    global movedir
    global game
    global powerups
    if movedir == (0, 0): # Set starting direction
        movedir = defaultdir(snake)
    if state == "PLAYING":
        cell = snake["tail"]
        while cell:
            if cell == snake["tail"]:
                occupy({"type": "free", "pos":cell["pos"]}) # Clear tail cell
            if cell == snake["head"]:
                oldx, oldy = cell["pos"][0], cell["pos"][1]
                newx, newy = oldx + movedir[0], oldy + movedir[1]

                # Handle edge movement
                if newx == GRID_SIZE:
                    newx = 0
                elif newx == -1:
                    newx = GRID_SIZE - 1
                elif newy == GRID_SIZE:
                    newy = 0
                elif newy == -1:
                    newy = GRID_SIZE - 1

                cell["pos"] = (newx, newy)

                slot = grid[newy][newx]
                if slot["object"]:
                    collision(snake, slot["object"])
                occupy(cell)
            else:
                cell["pos"] = cell["next"]["pos"]
                occupy(cell)

            cell = cell["next"]

        if state == "PLAYING": # When the loop breaks check the state again, state may have changed
            def move_selfcall():
                global state
                movesnake(snake)
            game.after(step - powerups["boost"], move_selfcall)

def movebind(key):
    '''
    Updates the movement direction based on player input.
    '''
    global movedir, state
    key_map = { # The Y signs are flipped since the Y axis is flipped
        "w": (0, -1),
        "s": (0, 1),
        "a": (-1, 0),
        "d": (1, 0)
    }
    key = key.char.lower()
    # Validate key
    validkey = key in key_map.keys() and key_map[key][movedir.index(0)] !=
0 # in wasd, must be across other axis
    if validkey and state == "PLAYING":
        movedir = key_map[key]


# Combine game
def startlife(window):
    '''
    Set up snake and start movement and object generation after input.
    '''
    global state, powerups
    state = "PLAYING"
    # Start core logic
    makegrid()
    snake = makesnake()

    # Listen for some input before starting movement
    label = Label(game,
                  text="Press [ENTER] to start", font=("Arial", 15,
"bold"),
                  fg="white", bg="black", width=30)
    label.place(relx=0.2, rely=0.45)

    def startmechanics(key):
        if key.keysym == "Return":
            global state, score, powerups
            state = "PLAYING"
            # Initialize
            score_label = Label(game,
                                text="Score: 0", font=("Arial", 10,
"bold"),
                                fg="white", bg="black", width=10)
            score_label.place(x=10, y=10)
            score["label"] = score_label

            for powerup in POWERUP_TYPES:
                powerups[powerup] =
POWERUP_DEFAULTS[POWERUP_TYPES.index(powerup)]
            powerups["active"] = 0
            powerups["occupants"] = {}

            movesnake(snake)

            # Start generating
            for _ in range(FOOD_CELLS_PRESENT):
                makefood()
            for _ in range(POWERUPS_PRESENT):
                gen_step = random.randint(POWERUP_GEN_STEP_MIN,
POWERUP_GEN_STEP_MAX)
                window.after(gen_step, makepowerup)
            if state == "PLAYING":
                window.unbind("<Return>")
            label.destroy()

    state = "WAITING"
    window.bind("<Return>", startmechanics)

def playgame(window, mode_step, passed_ender):
    '''
    Starts one play session
    '''
    global game, step, score, end_game
    # Initialize globals
    step = mode_step

    game = Frame(window, bg="black")
    game.pack(fill=BOTH, expand=True)

    end_game = passed_ender

    # Start a single life
    startlife(window)

    def completesetup(): # Don't overwrite bind before input
        global state, game
        if state == "PLAYING":
            window.bind("<KeyPress>", movebind) # Don't need to bind right before mainloop
        elif game:
            game.after(500, completesetup)
    completesetup()

    return game # Pass up the window so interface can clean it
```

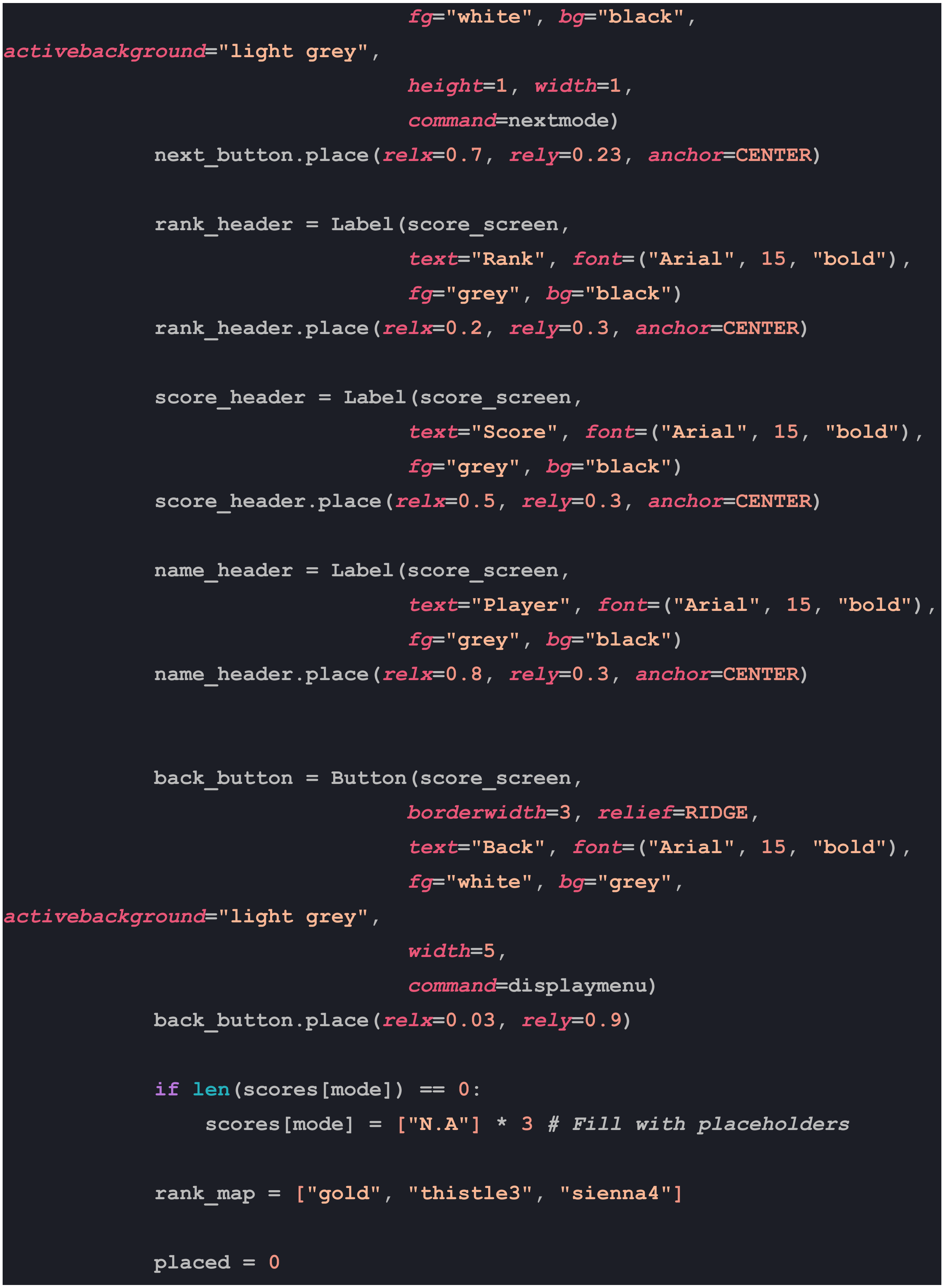
interface.py

```
'''
Menu and other interface of game
'''
import json
from tkinter import *
from core import playgame
from settings import *

window = None
screens = []
```

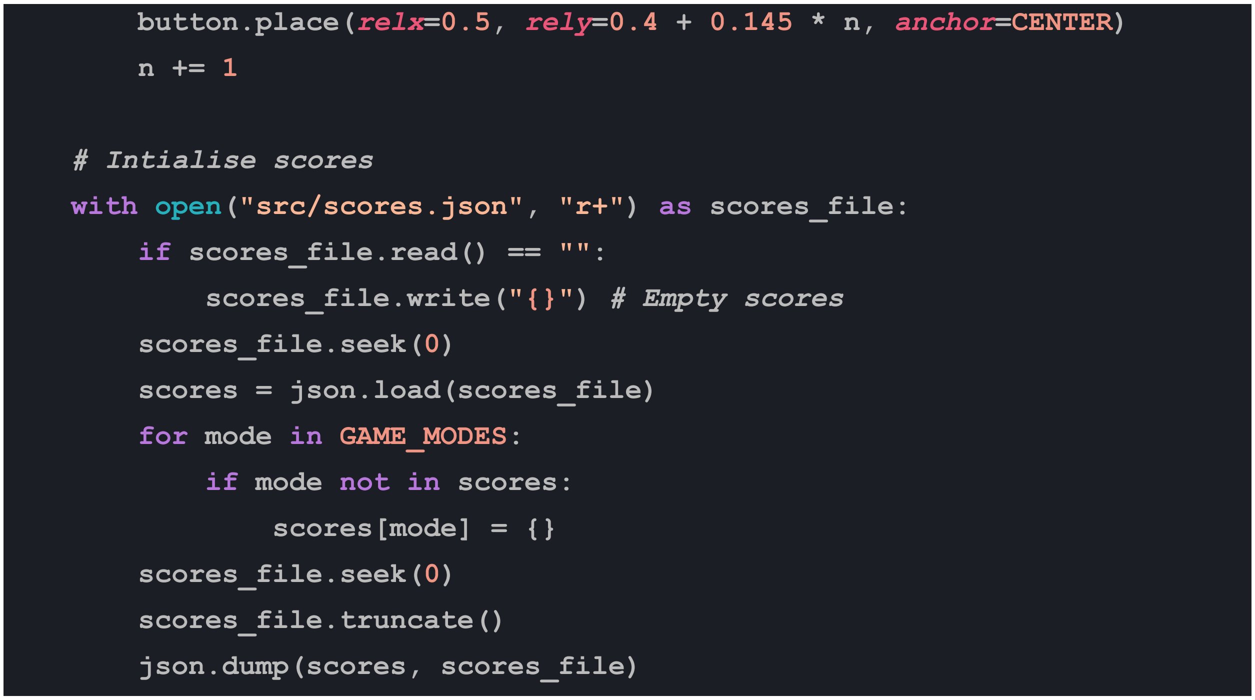


```
def makewindow():
    '''
    Makes the main game window, created only once
    '''
    WIN_SIZE = GRID_SIZE * CELL_SIZE

    global window
    window = Tk()
    window.title("Snake Game")
    window.geometry(str(WIN_SIZE)+"x"+str(WIN_SIZE))
    window.resizable(width=False, height=False)

    return window

def clearscreens():
    '''
    Destroys all screens, called before making a new one.
    Screen frames are not stored indefinitely, they are created on demand.
    '''
    for screen in screens:
        screen.destroy()

def endgame(mode_step, score):
    '''
    Shows the death interface. Updates the score after awaiting for
activity response.
    '''
    # Show a dying screen
    game = screens[-1]
    window.unbind("<KeyPress>")

    died_label = Label(game,
                       text="You Died :(", font=("Arial", 30, "bold"),
                       fg="white", bg="black")
    died_label.place(relx=0.34, rely=0.3)
    score_label = Label(game,
                        text="Score: "+str(score), font=("Arial", 19),
                        fg="white", bg="black")
    score_label.place(relx=0.5, rely=0.41, anchor=CENTER)
    cont_label = Label(game,
                       text="Press [ENTER] to continue", font=("Arial",
15, "bold"),
                       fg="grey", bg="black")
    cont_label.place(relx=0.3, rely=0.85)

    player_label = Label(game,
                         text="Player:", font=("Arial", 15),
                         fg="white", bg="black")
    player_label.place(relx=0.45, rely=0.6, anchor=E)

    player_entry = Entry(game,
                         font=("Arial", 15),
                         width=15)
    player_entry.place(relx=0.48, rely=0.6, anchor=W)
    player_entry.insert(0, "Unknown")

    def resume(key):
        if key.keysym == "Return":
            if score > 0:
                with open("src/scores.json", "r+") as scores_file:
                    mode = GAME_MODES[GAME_STEPS.index(mode_step)]

                    scores = json.load(scores_file)
                    scores[mode].update({score:
player_entry.get().title()})

                    scores_file.seek(0)
                    scores_file.truncate()
                    json.dump(scores, scores_file)

            clearscreens()
            displaymenu()
    window.bind("<Return>", resume)
```

```
def displaycredits(screen):
    '''
    Shows creator credits on any given screen (omitted in game)
    '''
    credits_label = Label(screen,
                          fg="grey", bg="black",
                          text="Made by Ankur Bohra in tkinter",
font=("Arial", 10))
    credits_label.place(relx=0.5, rely=0.935, anchor=CENTER)

def displaymodes():
    '''
    Shows game mode/difficulty screen. Game is started with respective
step from here.
    '''
    global window
    clearscreens()

    modes_screen = Frame(window, bg="black")
    screens.append(modes_screen)
    displaycredits(modes_screen)
    modes_screen.pack(fill=BOTH, expand=True)

    mode_title = Label(modes_screen,
                       text="Game Mode", font=("Arial", 30, "bold"),
                       fg="green", bg="black")
    mode_title.place(relx=0.5, rely=0.2, anchor=CENTER)

    def playmode(mode_step):
        '''
        Returns a callback but "injects" the mode_step variable into the
environment
        '''
        def command():
            modes_screen.pack_forget()
            game = playgame(window, mode_step, endgame)
            screens.append(game)
        return command

    for index in range(len(GAME_MODES)):
        mode = GAME_MODES[index]
        step = GAME_STEPS[index]
        button = Button(modes_screen,
                        borderwidth=3, relief=RIDGE,
                        text=mode, font=("Arial", 20, "bold"),
                        fg="black", bg="green", activebackground="light
green",
                        width=9,
                        command=playmode(step))
        button.place(relx=0.5, rely=0.4 + index*0.12, anchor=CENTER)

    back_button = Button(modes_screen,
                         borderwidth=3, relief=RIDGE,
                         text="Back", font=("Arial", 15, "bold"),
                         fg="white", bg="grey", activebackground="light
grey",
                         width=5,
                         command=displaymenu)
    back_button.place(relx=0.03, rely=0.9)

def displayscore():
    '''
    Shows stored scores sorted by mode and rank. Implements player
filtered scores.
    '''
    def showmode(mode_no, filter_player):
        global window
        if mode_no > len(GAME_MODES) - 1:
            mode_no = 0
        mode = GAME_MODES[mode_no]
        with open("src/scores.json", "r+") as scores_file:
            scores = json.load(scores_file)
            sorted_scores = list(scores[mode])
            sorted_scores.sort(reverse=True)
            sorted_scores = sorted_scores[:7] # upto 7 highscores shown

            clearscreens()

            score_screen = Frame(window, bg="black")
            screens.append(score_screen)
            displaycredits(score_screen)
            score_screen.pack(fill=BOTH, expand=True)

            screen_title = Label(score_screen,
                                text="Highscores", font=("Arial", 30,
"bold"),
                                fg="green", bg="black")
            screen_title.place(relx=0.5, rely=0.15, anchor=CENTER)

            mode_label = Label(score_screen,
                            text=mode, font=("Arial", 15, "bold"),
                            fg="light green", bg="black")
            mode_label.place(relx=0.5, rely=0.23, anchor=CENTER)

            def prevmode():
                showmode(mode_no - 1, filter_player)
            def nextmode():
                showmode(mode_no + 1, filter_player)
            previous_button = Button(score_screen,
                                    borderwidth=0,
                                    text="<", font=("Arial", 10, "bold"),
                                    fg="white", bg="black",
activebackground="light grey",
                                    height=1, width=1,
                                    command=prevmode)
            previous_button.place(relx=0.3, rely=0.23, anchor=CENTER)

            next_button = Button(score_screen,
                                borderwidth=0,
                                text=">", font=("Arial", 10, "bold"),
```

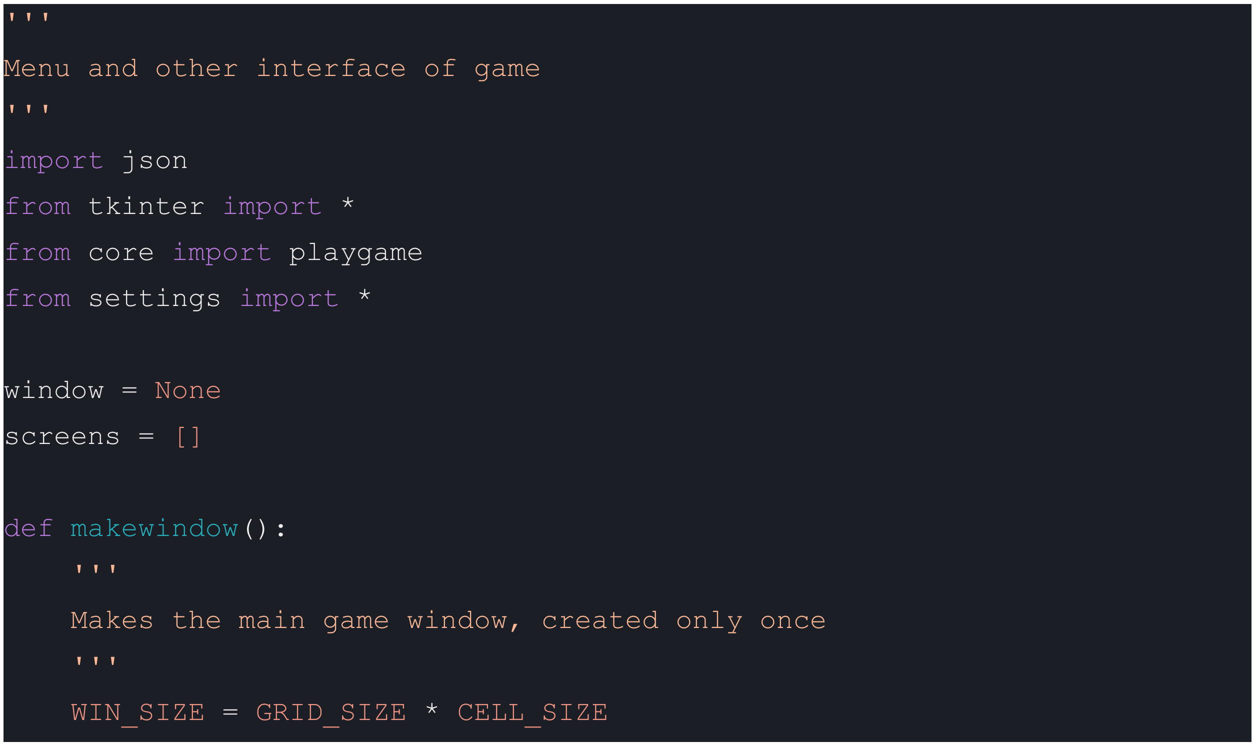


```
                                fg="white", bg="black",
activebackground="light grey",
                                height=1, width=1,
                                command=nextmode)
            next_button.place(relx=0.7, rely=0.23, anchor=CENTER)

            rank_header = Label(score_screen,
                                text="Rank", font=("Arial", 15, "bold"),
                                fg="grey", bg="black")
            rank_header.place(relx=0.2, rely=0.3, anchor=CENTER)

            score_header = Label(score_screen,
                                text="Score", font=("Arial", 15, "bold"),
                                fg="grey", bg="black")
            score_header.place(relx=0.5, rely=0.3, anchor=CENTER)

            name_header = Label(score_screen,
                                text="Player", font=("Arial", 15, "bold"),
                                fg="grey", bg="black")
            name_header.place(relx=0.8, rely=0.3, anchor=CENTER)


            back_button = Button(score_screen,
                                borderwidth=3, relief=RIDGE,
                                text="Back", font=("Arial", 15, "bold"),
                                fg="white", bg="grey",
activebackground="light grey",
                                width=5,
                                command=displaymenu)
            back_button.place(relx=0.03, rely=0.9)

            if len(scores[mode]) == 0:
                scores[mode] = ["N.A"] * 3 # Fill with placeholders

            rank_map = ["gold", "thistle3", "sienna4"]

            placed = 0
            for index in range(len(sorted_scores)):
                score = sorted_scores[index]
                player = scores[mode][score]
                if filter_player and player != filter_player:
                    continue
                rank = index + 1
                offset = (placed + 1) * 0.075
                if rank <= len(rank_map):
                    color = rank_map[rank - 1]
                else:
                    color = "white"
                rank_no = Label(score_screen,
                                    text=rank, font=("Arial", 15, "bold"),
                                    fg=color, bg="black")
                rank_no.place(relx=0.2, rely=0.3 + offset, anchor=CENTER)

                score = Label(score_screen,
                                    text=score, font=("Arial", 15,
"bold"),
                                    fg=color, bg="black")
                score.place(relx=0.5, rely=0.3 + offset, anchor=CENTER)

                def filtergen(player):
                    def filtermode():
                        if filter_player == player:
                            showmode(mode_no, None)
                        else:
                            showmode(mode_no, player)
                    return filtermode

                name = Button(score_screen,
                                text=player, font=("Arial", 15, "bold"),
                                fg=color, bg="black",
                                command=filtergen(player))
                name.place(relx=0.8, rely=0.3 + offset, anchor=CENTER)

                index = index + 1
                placed = placed + 1

            if scores[mode] == ["N.A"] * 3:
                scores[mode] = [] # Placeholders only temporary
    showmode(0, None)

def displaymenu():
    '''
    Shows main menu, links all screens together.
    '''
    global window
    clearscreens()

    menu = Frame(window, bg="black")
    screens.append(menu)
    displaycredits(menu)
    menu.pack(fill=BOTH, expand=True)

    buttons = {
        "Play":displaymodes,
        "Scores": displayscore
    }
    game_title = Label(menu,
                       text="Snake Game", font=("Arial", 30, "bold"),
                       fg="green", bg="black")
    game_title.place(relx=0.5, rely=0.2, anchor=CENTER)

    n = 0
    for buttonName in buttons:
        command = buttons[buttonName]
        button = Button(menu,
                        borderwidth=3, relief=RIDGE,
                        text=buttonName, font=("Arial", 20, "bold"),
                        fg="black", bg="green", activebackground="light
green",
                        width=7,
                        command=command)
```

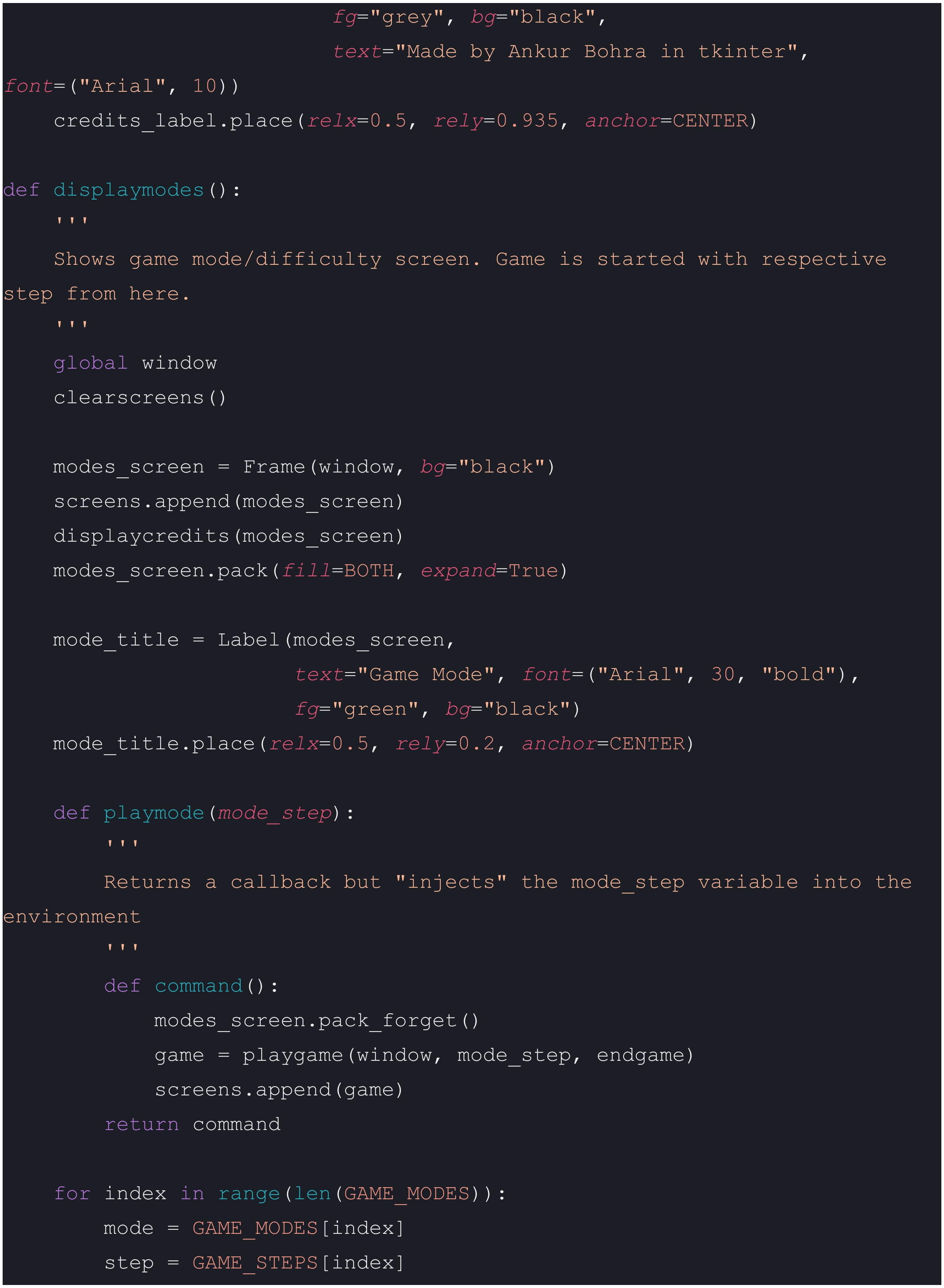


```
        button.place(relx=0.5, rely=0.4 + 0.145 * n, anchor=CENTER)
        n += 1

    # Intialise scores
    with open("src/scores.json", "r+") as scores_file:
        if scores_file.read() == "":
            scores_file.write("{}") # Empty scores
        scores_file.seek(0)
        scores = json.load(scores_file)
        for mode in GAME_MODES:
            if mode not in scores:
                scores[mode] = {}
        scores_file.seek(0)
        scores_file.truncate()
        json.dump(scores, scores_file)
```

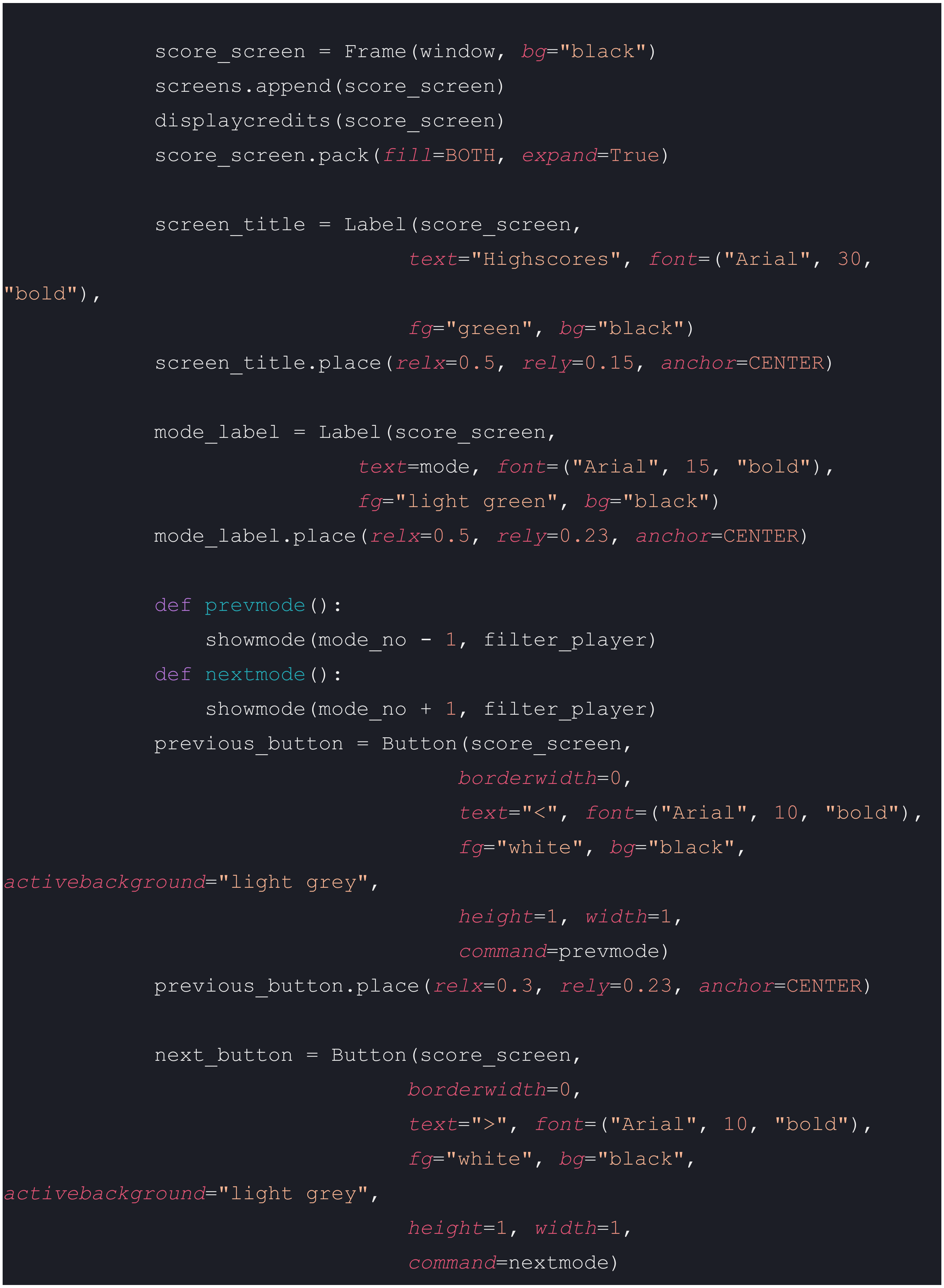
interface.py



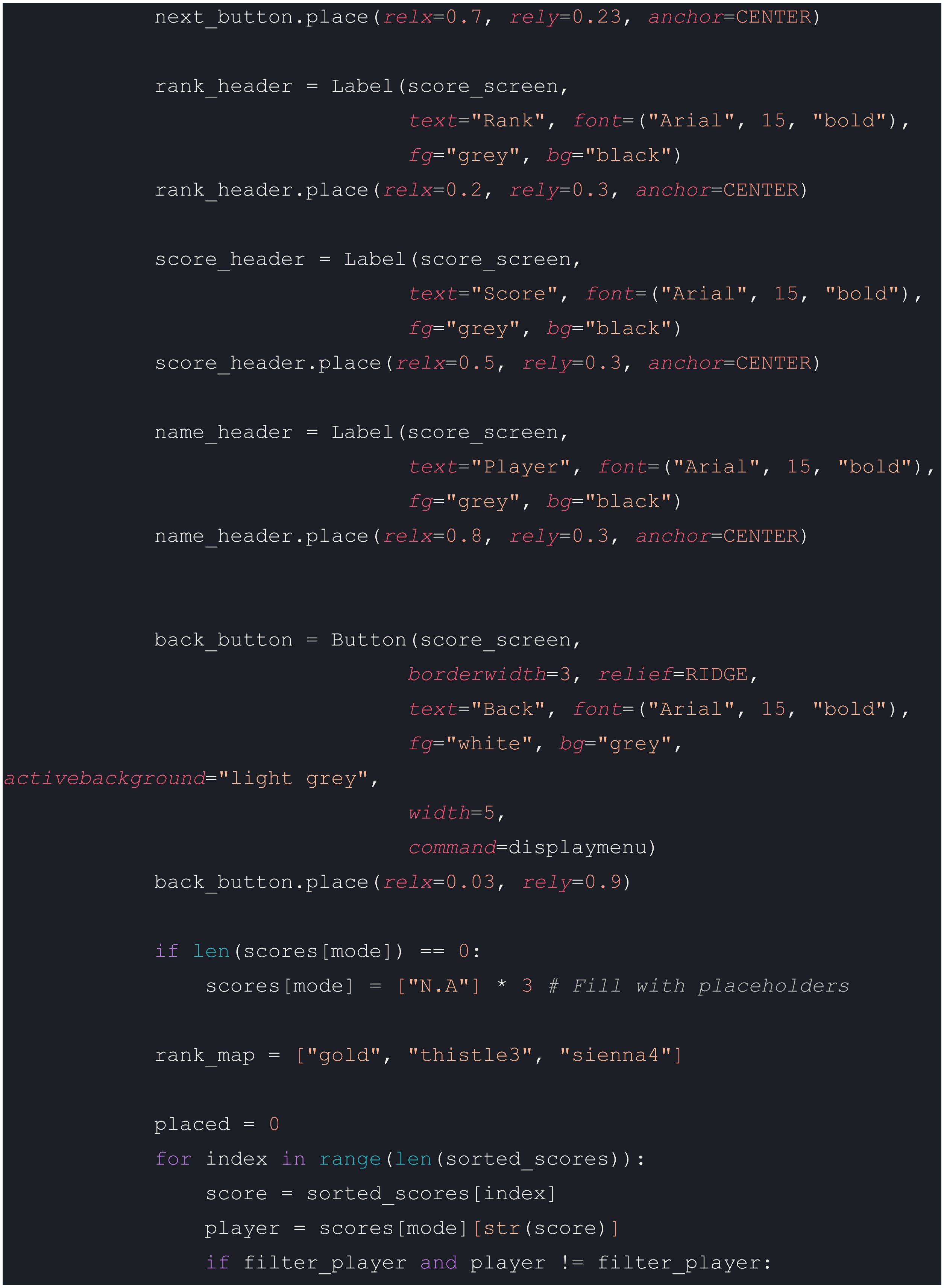
```
'''
Menu and other interface of game
'''
import json
from tkinter import *
from core import playgame
from settings import *

window = None
screens = []

def makewindow():
    '''
    Makes the main game window, created only once
    '''
    WIN_SIZE = GRID_SIZE * CELL_SIZE
    global window
    window = Tk()
    window.title("Snake Game")
    window.geometry(str(WIN_SIZE)+"x"+str(WIN_SIZE))
    window.resizable(width=False, height=False)

    return window

def clearscreens():
    '''
    Destroys all screens, called before making a new one.
    Screen frames are not stored indefinitely, they are created on demand.
    '''
    for screen in screens:
        screen.destroy()

def endgame(mode_step, score):
    '''
    Shows the death interface. Updates the score after awaiting for
activity response.
    '''
    # Show a dying screen
    game = screens[-1]
    window.unbind("<KeyPress>")

    died_label = Label(game,
                       text="You Died :(", font=("Arial", 30, "bold"),
                       fg="white", bg="black")
    died_label.place(relx=0.34, rely=0.3)
    score_label = Label(game,
                        text="Score: "+str(score), font=("Arial", 19),
                        fg="white", bg="black")
    score_label.place(relx=0.5, rely=0.41, anchor=CENTER)
    cont_label = Label(game,
                       text="Press [ENTER] to continue", font=("Arial",
15, "bold"),
                       fg="grey", bg="black")
    cont_label.place(relx=0.3, rely=0.85)

    player_label = Label(game,
                         text="Player:", font=("Arial", 15),
                         fg="white", bg="black")
    player_label.place(relx=0.45, rely=0.6, anchor=E)

    player_entry = Entry(game,
                         font=("Arial", 15),
                         width=15)
    player_entry.place(relx=0.48, rely=0.6, anchor=W)
    player_entry.insert(0, "Unknown")

    def resume(key):
        if key.keysym == "Return":
            if score > 0:
                with open("src/scores.json", "r+") as scores_file:
                    mode = GAME_MODES[GAME_STEPS.index(mode_step)]

                    scores = json.load(scores_file)
                    scores[mode].update({score:
player_entry.get().title()})

                    scores_file.seek(0)
                    scores_file.truncate()
                    json.dump(scores, scores_file)

            clearscreens()
            displaymenu()
    window.bind("<Return>", resume)

def displaycredits(screen):
    '''
    Shows creator credits on any given screen (omitted in game)
    '''
    credits_label = Label(screen,
```



```
                          fg="grey", bg="black",
                          text="Made by Ankur Bohra in tkinter",
font=("Arial", 10))
    credits_label.place(relx=0.5, rely=0.935, anchor=CENTER)

def displaymodes():
    '''
    Shows game mode/difficulty screen. Game is started with respective
step from here.
    '''
    global window
    clearscreens()

    modes_screen = Frame(window, bg="black")
    screens.append(modes_screen)
    displaycredits(modes_screen)
    modes_screen.pack(fill=BOTH, expand=True)

    mode_title = Label(modes_screen,
                       text="Game Mode", font=("Arial", 30, "bold"),
                       fg="green", bg="black")
    mode_title.place(relx=0.5, rely=0.2, anchor=CENTER)

    def playmode(mode_step):
        '''
        Returns a callback but "injects" the mode_step variable into the
environment
        '''
        def command():
            modes_screen.pack_forget()
            game = playgame(window, mode_step, endgame)
            screens.append(game)
        return command

    for index in range(len(GAME_MODES)):
        mode = GAME_MODES[index]
        step = GAME_STEPS[index]
        button = Button(modes_screen,
                        borderwidth=3, relief=RIDGE,
                        text=mode, font=("Arial", 20, "bold"),
                        fg="black", bg="green", activebackground="light
green",
                        width=9,
                        command=playmode(step))
        button.place(relx=0.5, rely=0.4 + index*0.12, anchor=CENTER)

    back_button = Button(modes_screen,
                         borderwidth=3, relief=RIDGE,
                         text="Back", font=("Arial", 15, "bold"),
                         fg="white", bg="grey", activebackground="light
grey",
                         width=5,
                         command=displaymenu)
    back_button.place(relx=0.03, rely=0.9)

def displayscore():
    '''
    Shows stored scores sorted by mode and rank. Implements player
filtered scores.
    '''
    def showmode(mode_no, filter_player):
        global window
        if mode_no > len(GAME_MODES) - 1:
            mode_no = 0
        mode = GAME_MODES[mode_no]
        with open("src/scores.json", "r+") as scores_file:
            scores = json.load(scores_file)
            sorted_scores = list(scores[mode])
            for index in range(len(sorted_scores)):
                score = sorted_scores[index]
                sorted_scores[index] = int(score)
            sorted_scores.sort(reverse=True)
            sorted_scores = sorted_scores[:7] # upto 7 highscores shown
            clearscreens()
```



```
            score_screen = Frame(window, bg="black")
            screens.append(score_screen)
            displaycredits(score_screen)
            score_screen.pack(fill=BOTH, expand=True)

            screen_title = Label(score_screen,
                                text="Highscores", font=("Arial", 30,
"bold"),
                                fg="green", bg="black")
            screen_title.place(relx=0.5, rely=0.15, anchor=CENTER)

            mode_label = Label(score_screen,
                            text=mode, font=("Arial", 15, "bold"),
                            fg="light green", bg="black")
            mode_label.place(relx=0.5, rely=0.23, anchor=CENTER)

            def prevmode():
                showmode(mode_no - 1, filter_player)
            def nextmode():
                showmode(mode_no + 1, filter_player)
            previous_button = Button(score_screen,
                                    borderwidth=0,
                                    text="<", font=("Arial", 10, "bold"),
                                    fg="white", bg="black",
activebackground="light grey",
                                    height=1, width=1,
                                    command=prevmode)
            previous_button.place(relx=0.3, rely=0.23, anchor=CENTER)

            next_button = Button(score_screen,
                                borderwidth=0,
                                text=">", font=("Arial", 10, "bold"),
                                fg="white", bg="black",
activebackground="light grey",
                                height=1, width=1,
                                command=nextmode)
```



```
            next_button.place(relx=0.7, rely=0.23, anchor=CENTER)

            rank_header = Label(score_screen,
                                text="Rank", font=("Arial", 15, "bold"),
                                fg="grey", bg="black")
            rank_header.place(relx=0.2, rely=0.3, anchor=CENTER)

            score_header = Label(score_screen,
                                text="Score", font=("Arial", 15, "bold"),
                                fg="grey", bg="black")
            score_header.place(relx=0.5, rely=0.3, anchor=CENTER)

            name_header = Label(score_screen,
                                text="Player", font=("Arial", 15, "bold"),
                                fg="grey", bg="black")
            name_header.place(relx=0.8, rely=0.3, anchor=CENTER)


            back_button = Button(score_screen,
                                borderwidth=3, relief=RIDGE,
                                text="Back", font=("Arial", 15, "bold"),
                                fg="white", bg="grey",
activebackground="light grey",
                                width=5,
                                command=displaymenu)
            back_button.place(relx=0.03, rely=0.9)

            if len(scores[mode]) == 0:
                scores[mode] = ["N.A"] * 3 # Fill with placeholders

            rank_map = ["gold", "thistle3", "sienna4"]

            placed = 0
            for index in range(len(sorted_scores)):
                score = sorted_scores[index]
                player = scores[mode][str(score)]
                if filter_player and player != filter_player:
                    continue
                rank = index + 1
                offset = (placed + 1) * 0.075
                if rank <= len(rank_map):
                    color = rank_map[rank - 1]
                else:
                    color = "white"
                rank_no = Label(score_screen,
                                    text=rank, font=("Arial", 15, "bold"),
                                    fg=color, bg="black")
                rank_no.place(relx=0.2, rely=0.3 + offset, anchor=CENTER)

                score = Label(score_screen,
                                    text=score, font=("Arial", 15,
"bold"),
                                    fg=color, bg="black")
                score.place(relx=0.5, rely=0.3 + offset, anchor=CENTER)

                def filtergen(player):
                    def filtermode():
                        if filter_player == player:
                            showmode(mode_no, None)
                        else:
                            showmode(mode_no, player)
                    return filtermode

                name = Button(score_screen,
                                text=player, font=("Arial", 15, "bold"),
                                fg=color, bg="black",
                                command=filtergen(player))
                name.place(relx=0.8, rely=0.3 + offset, anchor=CENTER)

                index = index + 1
                placed = placed + 1

            if scores[mode] == ["N.A"] * 3:
                scores[mode] = [] # Placeholders only temporary
```

```
    showmode(0, None)

def displaymenu():
    '''
    Shows main menu, links all screens together.
    '''
    global window
    clearscreens()

    menu = Frame(window, bg="black")
    screens.append(menu)
    displaycredits(menu)
    menu.pack(fill=BOTH, expand=True)

    buttons = {
        "Play":displaymodes,
        "Scores": displayscore
    }
    game_title = Label(menu,
                       text="Snake Game", font=("Arial", 30, "bold"),
                       fg="green", bg="black")
    game_title.place(relx=0.5, rely=0.2, anchor=CENTER)

    n = 0
    for buttonName in buttons:
        command = buttons[buttonName]
        button = Button(menu,
                        borderwidth=3, relief=RIDGE,
                        text=buttonName, font=("Arial", 20, "bold"),
                        fg="black", bg="green", activebackground="light
green",
                        width=7,
                        command=command)
        button.place(relx=0.5, rely=0.4 + 0.145 * n, anchor=CENTER)
        n += 1

    # Intialise scores
    with open("src/scores.json", "r+") as scores_file:
        if scores_file.read() == "":
            scores_file.write("{}") # Empty scores
        scores_file.seek(0)
        scores = json.load(scores_file)
        for mode in GAME_MODES:
            if mode not in scores:
                scores[mode] = {}
        scores_file.seek(0)
        scores_file.truncate()
        json.dump(scores, scores_file)
```

main.py

```
'''
Entry program
'''
from interface import makewindow, displaymenu

window = makewindow()
displaymenu()

window.mainloop()
```

settings.py

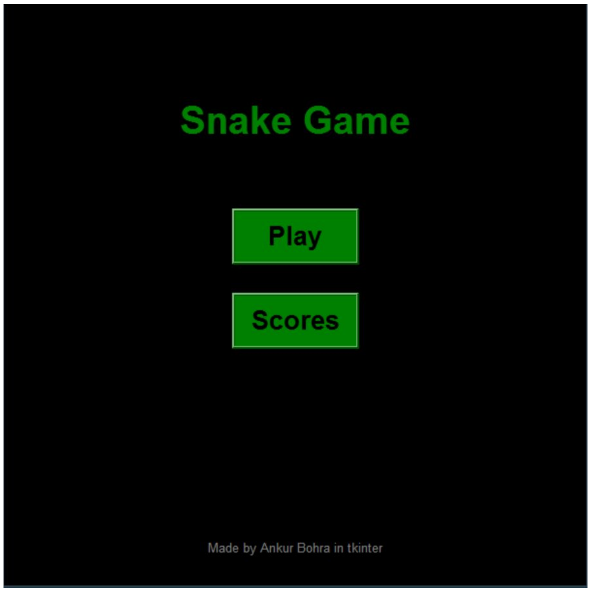
```
'''
All adjustable game settings
'''
GRID_SIZE = 20 # cells
CELL_SIZE = 30 # px
```

```
GRID_COLOR = "black"

MIN_LENGTH = 3 # cells
BODY_COLOR = "green" # body + tail
HEAD_COLOR = "dark green"

GAME_MODES = ["Easy", "Moderate", "Difficult", "Hard"]
GAME_STEPS = [250, 200, 150, 100] # corresponding to GAME_MODES

FOOD_CELLS_PRESENT = 3 # at a time
FOOD_GEN_STEP_MIN = 2000 # between eating and replacement
FOOD_GEN_STEP_MAX = 4000
FOOD_POINTS = [10, 15, 20]
FOOD_RARITY = [0.5, 0.3, 0.2] # probability, corresponding to
FOOD_STEP_LOSSES, multiples of 0.1
FOOD_COLORS = ["red", "orange", "yellow"] # corresponding to
FOOD_STEP_LOSSES

POWERUP_GEN_STEP_MIN = 3000 #10000 # between eating and replacement
POWERUP_GEN_STEP_MAX = 4000 #15000
POWERUPS_PRESENT = 2
POWERUP_TYPES = ["boost", "multiplier"]
POWERUP_DATA = [15, 1] # data concerned with respective powerup
POWERUP_DEFAULTS = [0, 1] # data concerned with respective powerup
POWERUP_COLORS = ["DarkOrchid1", "cyan2"] # corresponding to POWERUP_TYPES
POWERUP_DURATIONS = [5000, 8000] # ms
```

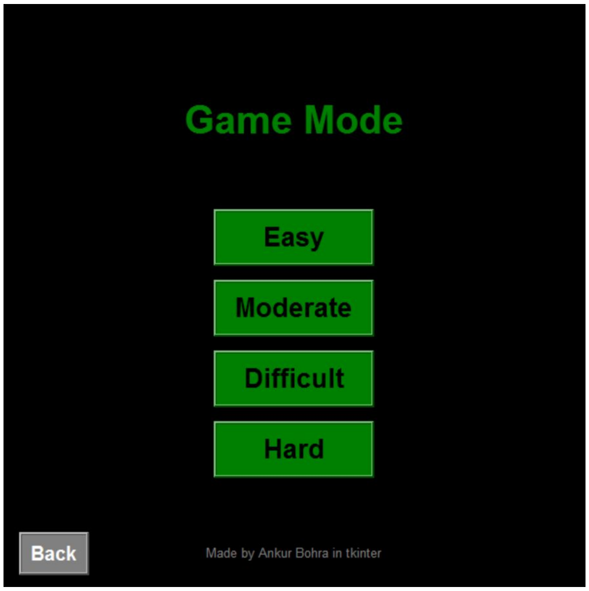
scores.json

```
{}
```

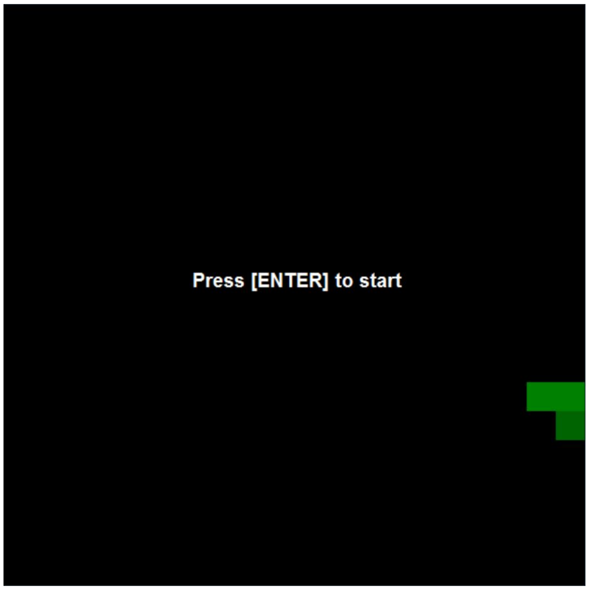
Output Screens

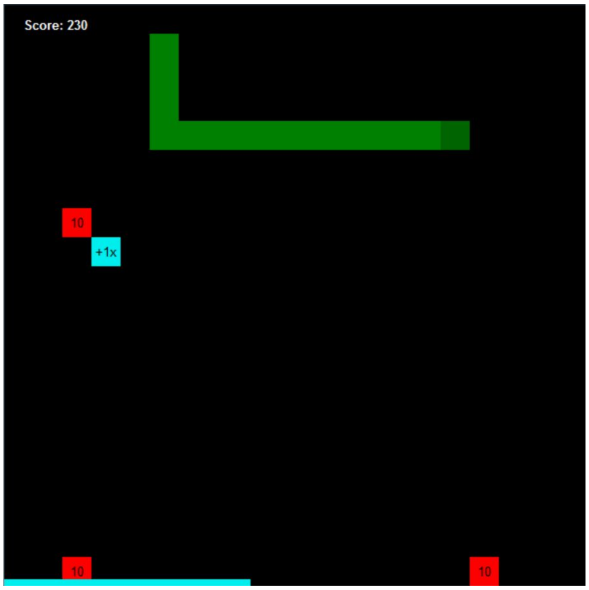

Snake Game

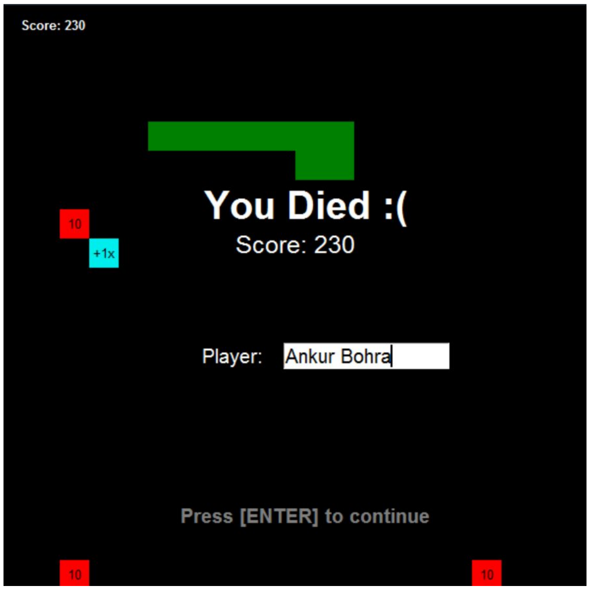
Made by Ankur Bohra in tkinter


Game Mode

Made by Ankur Bohra in tkinter


Press [ENTER] to start


Score: 230



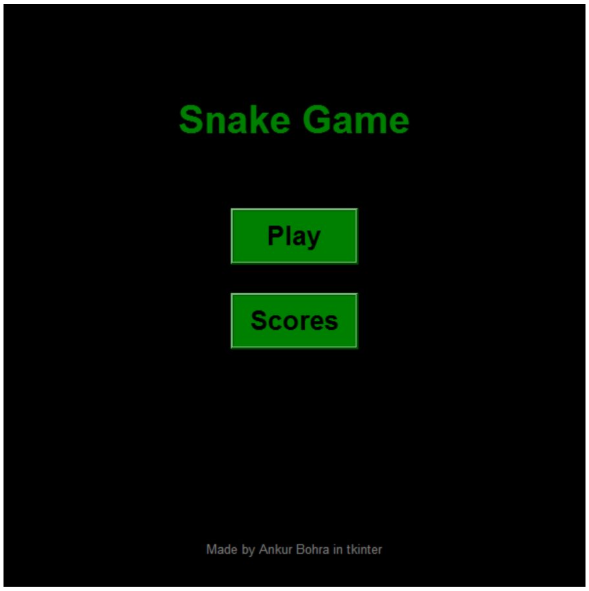
Score: 230

You Died :(

Score: 230

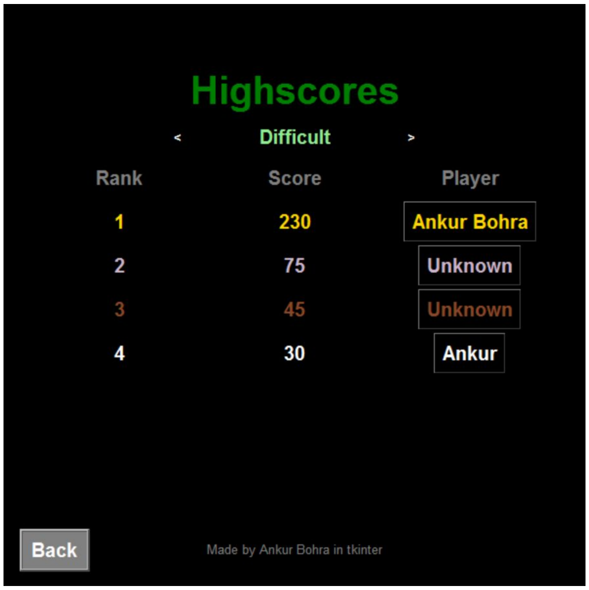
Player: Ankur Bohra

Press [ENTER] to continue



Snake Game

Made by Ankur Bohra in tkinter

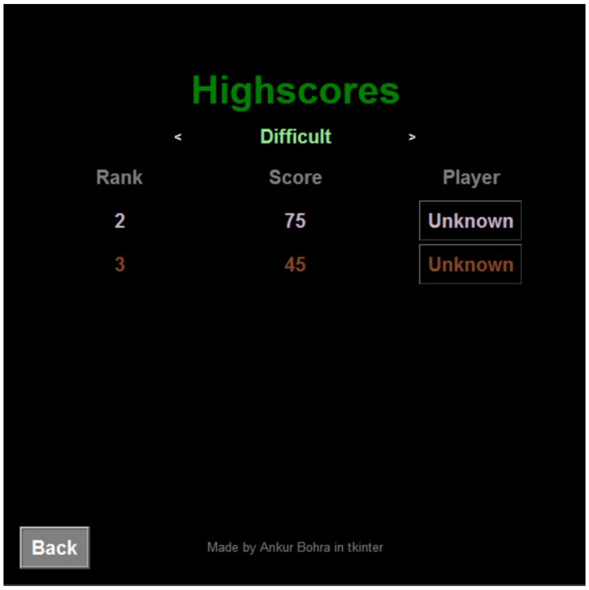


Highscores

< Difficult >

Rank	Score	Player
1	230	Ankur Bohra
2	75	Unknown
3	45	Unknown
4	30	Ankur

Made by Ankur Bohra in tkinter



Highscores

< Difficult >

Rank	Score	Player
2	75	Unknown
3	45	Unknown

Made by Ankur Bohra in tkinter

Bibliography

There is no code taken from external sources, however some websites served as references or as solutions to bugs:

1. https://docs.python.org/3/library/tkinter.html
2. http://www.science.smith.edu/dftwiki/index.php/Color_Charts_for_TKinter
3. https://stackoverflow.com
4. https://www.geeksforgeeks.org