```
#!/usr/bin/python

from Tkinter import *
from PIL import Image
from PIL import ImageTk
import tkFileDialog
import tkMessageBox
import re
import time
import Racecar.Tree
import Racecar.Compiler
import random
import pdb
import math

random.seed()

#current_program ised used to store the current file open in order to save back
#to that file
current_program = None

#Variable that serves as an interrupt to stop the program
should_stop = False
collision_occurred = False

#List of obstacles on the course at any given time
obstacles = []

#list of walls on the course at any given time
walls = []

#grid ticks
grid_ticks = []


class Obstacle:
    def __init__(self, x, y, width, height):
        self.obstacle_object = canvas.create_oval(
            x-width/2,
            y-height/2,
            x+width/2,
            y+height/2,
            fill="#000")
        self.width = width
        self.height = height
        self.center = (x, y)
        self.radius = width/2


#Wall class: can only be vertical or horizontal
class Wall:
    def __init__(self, start_x, start_y, length, is_horizontal):
        if is_horizontal:
            self.wall_object = canvas.create_line(
                start_x,
                start_y,
                start_x+length,
                start_y)
            self.start = start_x
            self.end = start_x+length
            self.constant_coord = start_y
        else:
            self.wall_object = canvas.create_line(
                start_x,
```

```
                start_y,
                start_x,
                start_y+length)
            self.start = start_y
            self.end = start_y+length
            self.constant_coord = start_x
        self.is_horizontal = is_horizontal


class Program:
    def __init__(self):
        self.name = ''
        self.file_obj = None


#Static variables for turning the car
class WheelDirection:
    LEFT = 1
    RIGHT = -1


#Car direction object
#X and Y can be 1,0,-1 respectively. The only invalid combination is when x = 0
#and y = 0. Positive axes point right and up respectively
class CarDirection:
    FORWARDS = 1
    BACKWARDS = -1

    def __init__(self):
        self.direction = 0

    DIRECTIONS = [(1, 0),
                  (1, -1),
                  (0, -1),
                  (-1, -1),
                  (-1, 0),
                  (-1, 1),
                  (0, 1),
                  (1, 1)]

    def get_direction(self):
        return CarDirection.DIRECTIONS[self.direction]

    def turn_right(self):
        self.direction = (self.direction - 1) % len(CarDirection.DIRECTIONS)

    def turn_left(self):
        self.direction = (self.direction + 1) % len(CarDirection.DIRECTIONS)

    def opposite_direction(self):
        return DIRECTIONS[
            (self.direction + len(CarDirection.DIRECTIONS)/2) %
            len(CarDirection.DIRECTIONS)]


class Car:
    def __init__(self):
        self.position_x = 0
        self.position_y = 0
        #Car direction starts facing right
        self.car_direction = CarDirection()
        self.image = None
        self.image_tk = None
        self.car_object = None
        self.width = 27
```

```
        self.height = 27
        self.radius = 25

    #Drive method that updates the car's position (in the model, not on the UI)
    #UI animation will need to be done moving x and y simultaneously
    def update_position(self, steps, movement_direction):
        self.position_x += (
            self.car_direction.get_direction()[0]
            * steps
            * movement_direction)

        self.position_y += (
            self.car_direction.get_direction()[1]
            * steps
            * movement_direction)


#Function to get a unique position of object, in order to detect for collisions
def get_position(x, y):
    return 1000 * int(x) + int(y)


def getCurrentPosition():
    global car
    return get_position(car.position_x, car.position_y)


#Checks if there is going to be a collision on the upcoming path
#drive_direction has to be CarDirection.FORWARDS or CarDirection.BACKWARDS
def can_move(num_steps, drive_direction):
    global car
    curr_x = int(car.position_x)
    curr_y = int(car.position_y)
    direction = car.car_direction.get_direction()
    path = []

    #If the direction is backwards, just reverse the direction
    if drive_direction == CarDirection.BACKWARDS:
        direction = car.car_direction.opposite_direction()

    #Create path coordinates
    for i in range(0, steps_to_pixels(num_steps)):
        pos = (curr_x + i * direction[0], curr_y + i * direction[1])
        path.append(pos)

    #Check each point in the path to see if it collides with any of the
    #obstacles
    for pos in path:
        if is_collision(pos[0], pos[1]):
            return False

    return True


#Number of steps on screen is proportional to screen size
def steps_to_pixels(steps):
    return canvas_frame.winfo_reqwidth()/110*steps


#Function to find the distance between two points
def distance_between_points(x_1, y_1, x_2, y_2):
    return math.sqrt(math.pow((x_2-x_1), 2) + math.pow((y_2-y_1), 2))


#API Functions
```

```
#direction must be either CarDirection.FORWARDS or CarDirection.BACKWARDS
def translate_car(steps, direction):
    global car
    global should_stop
    global collision_occurred

    steps = int(steps)
    direction = int(direction)

    curr_x = car.position_x
    curr_y = car.position_y

    one_step = steps_to_pixels(1)

    for i in range(0, steps_to_pixels(int(steps))):
        #Check interrupt variable
        if should_stop and i % one_step == 0:
            return

        time.sleep(0.01)
        #car_direction is FORWARDS or BACKWARDS (1 and -1 respectively)

        if is_collision(curr_x, curr_y):
            print_to_console("COLLISION")
            #Stop execution of program
            #TODO Deal with delay on collision
            should_stop = True
            collision_occurred = True
            return
        else:
            canvas.move(
                car.car_object,
                direction * car.car_direction.get_direction()[0],
                direction * car.car_direction.get_direction()[1])

            curr_x = curr_x + direction * car.car_direction.get_direction()[0]
            curr_y = curr_y + direction * car.car_direction.get_direction()[1]
            canvas.update()

        car.update_position(1, direction)


#direction must be WheelDirection.LEFT or WheelDirection.RIGHT
#Note: only check interrupt variable at the beginning, because
#we shouldn't allow partial rotations
def rotate_car(direction):
    global car
    global should_stop

    #Check interrupt variable
    if should_stop:
        return

    #This is current index in DIRECTIONS array
    current_direction_deg = car.car_direction.direction*45

    if direction == WheelDirection.LEFT:
        car.car_direction.turn_left()
    elif direction == WheelDirection.RIGHT:
        car.car_direction.turn_right()
    else:
        return

    for i in range(0, 45):
        time.sleep(0.01)
```

```
        canvas.delete(car.car_object)

        if direction == WheelDirection.LEFT:
            car.image_tk = ImageTk.PhotoImage(
                car.image.rotate(current_direction_deg + i))
        elif direction == WheelDirection.RIGHT:
            car.image_tk = ImageTk.PhotoImage(
                car.image.rotate(current_direction_deg - i))
        else:
            return

        car.car_object = canvas.create_image(
            car.position_x,
            car.position_y,
            image=car.image_tk)
        canvas.update()


#Check for collision with walls of the maze and a finish line
def collision_with_internal_walls(pos_x, pos_y):
    for wall in walls:
        #horizontal wall
        if wall.is_horizontal:
            #in rance of wall
            if wall.start <= pos_x <= wall.end:
                #Current direction of the car
                direction = car.car_direction.get_direction()
                #distance from wall
                dist_to_wall = math.fabs(pos_y-wall.constant_coord)
                #Car is horizontally oriented
                if direction == (1, 0) or direction == (-1, 0):
                    if dist_to_wall < car.radius/2:
                        return True
                    #Check for collision with car
                #Car is not horizontally oriented
                else:
                    if dist_to_wall < car.radius:
                        return True
        #vertical wall
        else:
            #in range of wall
            if wall.start <= pos_y <= wall.end:
                direction = car.car_direction.get_direction()
                #distance from wall
                dist_to_wall = math.fabs(pos_x-wall.constant_coord)
                #Car is vertically oriented
                if direction == (0, 1) or direction == (0, -1):
                    if dist_to_wall < car.radius/2:
                        return True
                #Car is not vertically oriented
                else:
                    if dist_to_wall < car.radius:
                        return True

    return False


def is_collision(curr_x, curr_y):
    #Check for collisions with obstacles and walls

    #Check obstacles
    for obstacle in obstacles:
        distance = distance_between_points(
            curr_x,
            curr_y,
```

```
            obstacle.center[0],
            obstacle.center[1])
        if distance < (car.radius + obstacle.radius):
            return True
    #check internal walls for collision
    if collision_with_internal_walls(curr_x, curr_y):
        return True
    #Check boundary walls
    elif not (origin[0] <= curr_x <= anti_origin[0]):
        return True
    elif not (origin[1] <= curr_y <= anti_origin[1]):
        return True
    else:
        return False


def print_to_console(message):
#Should console be cleared each time the program is restart?
#Or should there be a button?
    console.config(state=NORMAL)
    console.insert(END, str(message) + '\n')
    console.config(state=DISABLED)


#Course generation functions

#Course one is a slalom of blocks
def course_one():
    clear_course()
    obstacle_coord_x = 123
    obstacle_coord_y = int(canvas.winfo_reqheight())/2
    while obstacle_coord_x < anti_origin[0]:
        obstacle = Obstacle(obstacle_coord_x, obstacle_coord_y, 30, 30)
        obstacles.append(obstacle)
        obstacle_coord_x = obstacle_coord_x + 150


#TODO -- Fill in the rest of the courses
#Course two is a simple maze
finish_line = None


def course_two():
    global finish_line

    clear_console()

    message = "Try to navigate through the maze and cross the finish line!"
    print_to_console(message)

    clear_course()
    wall_coord_x = 123
    wall_length = 4*int(canvas.winfo_reqheight())/5

    #used to toggle position of line
    put_wall_on_top = True

    #walls
    while wall_coord_x < anti_origin[0]:
        if put_wall_on_top:
            wall = Wall(
                wall_coord_x,
                0,
                wall_length,
                False)
```

```
            walls.append(wall)
        else:
            wall = Wall(
                wall_coord_x,
                int(canvas.winfo_reqheight())/5+23,
                wall_length,
                False)
            walls.append(wall)
        put_wall_on_top = not put_wall_on_top
        wall_coord_x = wall_coord_x+100

    #finish line
    wall_coord_x = wall_coord_x-100
    finish_line = canvas.create_line(
        wall_coord_x,
        wall_length,
        wall_coord_x,
        canvas.winfo_reqheight()+23,
        fill="black",
        dash=(4, 4))


def course_three():
    clear_course()

    max_x = canvas.winfo_reqwidth()
    max_y = canvas.winfo_reqheight()

    while len(obstacles) < 30:
        pos_x = random.randrange(0, max_x, 1)
        pos_y = random.randrange(0, max_y, 1)
        radius = random.randrange(10, 50, 1)

        if is_collision(pos_x, pos_y):
            continue
        #Check for collision with car
        elif distance_between_points(
                pos_x,
                pos_y,
                car.position_x,
                car.position_y) < (car.radius + radius):
            continue
        else:
            obstacle = Obstacle(pos_x, pos_y, radius, radius)
            obstacles.append(obstacle)


def course_four():
    clear_course()
    obstacle_coord_x = 123
    obstacle_coord_y = 60
    while obstacle_coord_y < anti_origin[1]:
        while obstacle_coord_x < anti_origin[0]:
            obstacle = Obstacle(obstacle_coord_x, obstacle_coord_y, 30, 30)
            obstacles.append(obstacle)
            obstacle_coord_x = obstacle_coord_x + 150
        obstacle_coord_y = obstacle_coord_y + 80
        obstacle_coord_x = 123

    for obstacle in obstacles:
        print obstacle.center


def clear_course():
    global obstacles
```

```
    global walls
    global finish_line
    #remove obstacles from the course
    for obstacle in obstacles:
        canvas.delete(obstacle.obstacle_object)

    #Needs to take care of finish line too, which isn't a wall object
    for wall in walls:
        canvas.delete(wall.wall_object)

    if finish_line is not None:
        canvas.delete(finish_line)
    #clear the obstacles and walls array
    obstacles = []
    walls = []
    finish_line = None


#Menu functions
def open_file():
    global current_program

    #Keep returning to the file dialog if they didn't select a .race file
    while True:
        file_name = tkFileDialog.askopenfilename(defaultextension=".race")
        if file_name == '':
            return

        #Check validity of file being opened
        file_regex = re.compile("\w*\.race$")
        if len(file_regex.findall(file_name)) == 0:
            tkMessageBox.showwarning(
                "Open File Error",
                "You must open a .race file")
        else:
            break

    file_object = open(file_name, 'r')
    current_program = Program()
    current_program.name = file_name
    current_program.file_obj = file_object
    code.delete(1.0, END)
    code.insert(1.0, file_object.read())
    current_program.file_obj.close()


def save():
    global current_program
    if current_program is None:
        save_file_as()
    else:
        save_file()


def save_file():
    global current_program
    if not current_program.file_obj.closed:
        current_program.file_obj.close()
    #Open file for writing (will clear it)
    current_program.file_obj = open(current_program.name, 'w')
    current_program.file_obj.truncate()
    current_program.file_obj.write(code.get(1.0, END))
    current_program.file_obj.close()
```

```
def save_file_as():
    global current_program
    file_name = tkFileDialog.asksaveasfilename(defaultextension=".race")

    #Defaults to saving on the desktop
    if file_name == '':
        file_name = '~/Desktop/racecar_program.race'

    current_program = Program()
    current_program.name = file_name
    current_program.file_obj = open(file_name, 'w')
    current_program.file_obj.write(code.get(1.0, END))
    current_program.file_obj.close()


def clear():
    if code.get(1.0, END) == '':
        return

    if tkMessageBox.askyesno(
            "Clear code",
            "Are you sure you want to delete all of your code?"):
        code.delete(1.0, END)


def clear_console():
    console.config(state=NORMAL)
    console.delete(1.0, END)
    console.config(state=DISABLED)


#Triggers interrupt
def stop_program():
    global should_stop
    should_stop = True


#Code generation and compilation
#Runs code
def generate_program(code):
    global should_stop
    global collision_occurred

    #Set the interrupt variable whenever a program is run
    should_stop = False
    collision_occurred = False
    if len(code) > 1:
        #print code[:-1]
        #demo(code)
        python_code, errors, correct = verify_program(code)
        if(correct):
            #Print message to console saying program is executing
            print_to_console("Program executing")
            console.tag_add("Correct", "1.0", "1.end")
            console.tag_config("Correct", foreground="Green")

            #Toggle the buttons on the bottom and run program
            toggle_buttons(True)
            tempGlobal = globals().copy()
            exec(python_code, tempGlobal)
            toggle_buttons(False)

            #If collision occurred
            if should_stop:
                if collision_occurred:
```

```
                    if tkMessageBox.showwarning(
                            "Oops!", "You crashed! Try again"):
                        reset_car_position()
            else:
                #Print message to console saying program is finished executing
                print_to_console("Done running program")
                console.tag_add("End", "end -2 l", END)
                console.tag_config("End", foreground="Green")
                print "OUTPUT: " + console.index("end -1 l")
        else:
            #Print message to console saying program has errors
            print_to_console(
                "You have " +
                str(len(errors)) +
                " error(s) in your program")
            console.tag_add("Error", "1.0", "1.end")
            console.tag_config("Error", foreground="Red")

            for error in errors:
                print_to_console(error)
    else:
        print "Blank"


#Checks if program is a valid Racecar program and returns corresponding python
#code if necessary
def verify_program(code):
    clear_console()
    if len(code) < 2:
        return ("BLANK", False)
    code, errors = Racecar.Compiler.getPythonCode(code)
    if errors:
        return (code, errors, False)
    else:
        return (code, errors, True)


#Called when verify program is called
def verify_program_callback(code):
    verification = verify_program(code)
    if verification[2]:
        print_to_console("Program syntax correct")
        console.tag_add("Correct", "1.0", "1.end")
        console.tag_config("Correct", foreground="Green")
    else:
        errors = verification[1]
        print_to_console(
            "You have " +
            str(len(errors)) +
            " error(s) in your program")
        console.tag_add("Error", "1.0", "1.end")
        console.tag_config("Error", foreground="Red")

        for error in errors:
            print_to_console(error)


#Resets car's position and orientation to original
def reset_car_position():
        global car
        canvas.delete(car.car_object)
        car.image_tk = ImageTk.PhotoImage(car.image)
        car_height = int(canvas.winfo_reqheight())/2
        car.car_object = canvas.create_image(
            23,
```

```
            car_height,
            image=car.image_tk)
        car.position_x = 23
        car.position_y = car_height
        car.car_direction = CarDirection()

#car object
car = Car()


#User interface
#Toggle enabled and disabled buttons when program is run and stopped
def toggle_buttons(stop_button_should_be_enabled):
    if stop_button_should_be_enabled:
        run_button.config(state=DISABLED)
        stop_button.config(state=NORMAL)
        reset_car_position_button.config(state=DISABLED)
        clear_button.config(state=DISABLED)
    else:
        run_button.config(state=NORMAL)
        stop_button.config(state=DISABLED)
        reset_car_position_button.config(state=NORMAL)
        clear_button.config(state=NORMAL)

root = Tk()
root.title('Racecar')
#Height is always three fourths the width of the window
window_width = root.winfo_screenwidth() - 100
window_height = 9*window_width/16
root.geometry("%dx%d" % (window_width, window_height))
root.resizable(width=FALSE, height=FALSE)

menu_bar = Menu(root)

menu = Menu(menu_bar, tearoff=0)
menu.add_command(label="Open", command=open_file)
menu.add_command(label="Save", command=save)
menu.add_command(label="Save As", command=save_file_as)
menu.add_separator()
menu.add_command(label="Quit", command=exit)
menu_bar.add_cascade(label="File", menu=menu)

menu = Menu(menu_bar, tearoff=0)

command = lambda: verify_program_callback(code.get(1.0, END))
menu.add_command(label="Verify Code", command=command)

command = lambda: generate_program(code.get(1.0, END))
menu.add_command(label="Run Code", command=command)

menu.add_command(label="Clear Code", command=clear)
menu.add_command(label="Clear Console", command=clear_console)
menu_bar.add_cascade(label="Code", menu=menu)

menu = Menu(menu_bar, tearoff=0)
menu.add_command(label="Course 1", command=course_one)
menu.add_command(label="Course 2", command=course_two)
menu.add_command(label="Course 3", command=course_three)
menu.add_command(label="Course 4", command=course_four)
menu.add_command(label="Course 5", command=course_five)
menu.add_separator()
menu.add_command(label="Clear course", command=clear_course)
menu_bar.add_cascade(label="Courses", menu=menu)

root.config(menu=menu_bar)
```

```
715.
716. #frame for left side of window
717. left_frame = Frame(root)
718.
719. #label for code window
720. code_label = Label(left_frame, text="Enter code here", anchor=W, pady=5)
721.
722. #frame for code window to hold textbox and scrollbar
723. code_frame = Frame(
724.     left_frame,
725.     width=int(0.3*window_width),
726.     height=9*window_height/10)
727. code_frame.grid_propagate(False)
728.
729. #scrollbar for code window
730. code_scrollbar = Scrollbar(code_frame)
731. code_scrollbar.pack(side=RIGHT, fill=Y)
732.
733. #code is the window in which the code is written
734. code = Text(
735.     code_frame,
736.     width=50,
737.     #height=window_height/16-8,
738.     wrap=WORD,
739.     yscrollcommand=code_scrollbar.set)
740.
741. #Frame for buttons
742. button_frame = Frame(left_frame)
743.
744. #run_button passes code into a run program method
745. command = lambda: generate_program(code.get(1.0, END))
746. run_button = Button(
747.     button_frame,
748.     text="Run Code",
749.     pady=5,
750.     padx=5,
751.     command=command)
752.
753. #Stop execution of running program
754. stop_button = Button(
755.     button_frame,
756.     text="Stop Program",
757.     padx=5,
758.     pady=5,
759.     command=stop_program)
760. stop_button.config(state=DISABLED)
761.
762. #reset car position button puts the car back in its original position and
763. #orientation
764. reset_car_position_button = Button(
765.     button_frame,
766.     text="Reset Car Position",
767.     pady=5,
768.     padx=5,
769.     command=reset_car_position)
770.
771. #clear_button clears the code in the text box
772. clear_button = Button(
773.     button_frame,
774.     text="Clear Code",
775.     command=clear)
776.
777. #canvas is where the car will go
778. canvas_frame = Frame(
779.     root,
```

```
    width=window_width/1.5,
    height=window_height/1.5,
    padx=2,
    pady=2)

canvas_frame.configure(borderwidth=1.5, background='black')
canvas = Canvas(
    canvas_frame,
    width=window_width/1.5,
    height=window_height/1.5)

car.image = Image.open('Racecar/RacecarGUI/images/racecar.png')
car.image_tk = ImageTk.PhotoImage(car.image)

car.car_object = canvas.create_image(
    23,
    int(canvas.winfo_reqheight())/2,
    image=car.image_tk)

car.position_x = 23
car.position_y = int(canvas.winfo_reqheight())/2

#label above the console
console_label = Label(root, text="Console", anchor=W, pady=5)

#frame for the console to hold the textbox and the scrollbar
console_frame = Frame(root)

#scrollbar for the console
console_scrollbar = Scrollbar(console_frame)
console_scrollbar.pack(side=RIGHT, fill=Y)

#console to print to
console = Text(
    console_frame,
    width=int(window_width/1.5),
    height=8,
    padx=2,
    pady=2,
    wrap=WORD,
    yscrollcommand=console_scrollbar.set)

console.config(state=DISABLED)

#add them to GUI Window
#These are grouped logically in order to better see what's going on
left_frame.pack(side=LEFT, fill=BOTH)

code_label.pack()

code_frame.pack(expand=1, fill=BOTH)
code.pack(expand=1, fill=BOTH)

button_frame.pack(fill=BOTH)
run_button.grid(row=1, column=1)
stop_button.grid(row=1, column=2)
reset_car_position_button.grid(row=1, column=3)
clear_button.grid(row=1, column=4)

canvas_frame.pack(expand=1, fill=BOTH)
canvas.pack(expand=1, fill=BOTH)

console_label.pack()

console_frame.pack(expand=1, fill=BOTH, pady=(0, 10))
```

```
console.pack(expand=1, fill=BOTH)

code_scrollbar.config(command=code.yview)
console_scrollbar.config(command=console.yview)

root.update_idletasks()

#Origin and antiorigin are limits on the canvas where the car moves
origin = (23, 26)
anti_origin = (
    23+106*canvas_frame.winfo_width()/110,
    26+56*canvas_frame.winfo_width()/110)

#horizontal grid lines
position = 0
while position < anti_origin[0]:
    tick = canvas.create_line(
        position,
        anti_origin[1]-5+35,
        position,
        anti_origin[1]+35,
        fill="#000",
        width=2)
    grid_ticks.append(tick)
    position += steps_to_pixels(5)

#vertical grid lines
position = 0
while position < anti_origin[1]:
    tick = canvas.create_line(
        0,
        position,
        5,
        position,
        fill="#000",
        width=2)
    grid_ticks.append(tick)
    position += steps_to_pixels(5)

print anti_origin
#Run the GUI
root.mainloop()
```