**PyRacing documentation**

*Date: 15.11.2022*

PyRacing is a 2D physics-based game, where the player is in control of a car with a driver inside. The goal of the game is to reach the end of the map and the player loses when the driver’s head hits the ground. The player is able to change aspects of the simulation & the terrain.

**Contents**

Съдържание

[Installation 2](#_Toc119763426)

[User interface 4](#_Toc119763427)

[Main menu 4](#_Toc119763428)

[Options menu 4](#_Toc119763429)

[Credits menu 4](#_Toc119763430)

[Game 5](#_Toc119763431)

[GAME SETTINGS 6](#_Toc119763432)

[CODE OVERVIEW 7](#_Toc119763433)

[CLASS APP 7](#_Toc119763434)

[The app.run() method 8](#_Toc119763435)

[CLASS MENU 9](#_Toc119763436)

[CLASS MAINMENU 10](#_Toc119763437)

[CLASS OPTIONSMENU 11](#_Toc119763438)

[CLASS CREDITSMENU 12](#_Toc119763439)

[CLASS GAME 13](#_Toc119763440)

[Class game’s attributes 13](#_Toc119763441)

[The game loop 14](#_Toc119763442)

[Part 1: Game set-up 14](#_Toc119763443)

[Part 2: The Game Loop 15](#_Toc119763444)

[SPRITES – CLASS ENTITY 16](#_Toc119763445)

[About the simulation 16](#_Toc119763446)

[CLASS PLAYER 16](#_Toc119763447)

[Class Car 16](#_Toc119763448)

[Class Person 19](#_Toc119763449)

[Class Player 20](#_Toc119763450)

[Game.objects\_group objects 21](#_Toc119763451)

[CLASS CarMovementHandler 21](#_Toc119763452)

[CLASS HeadCollisionHandlerPygame 22](#_Toc119763453)

[CLASS VictoryHandler 23](#_Toc119763454)

[CLASS MessageBox 24](#_Toc119763455)

[TERRAIN GENERATION – Classes Boundaries and Terrain 25](#_Toc119763456)

[CLASS Boundaries 25](#_Toc119763457)

[CLASS Terrain 25](#_Toc119763458)

[generate\_terrain() 26](#_Toc119763459)

[The terrain generation functions 26](#_Toc119763460)

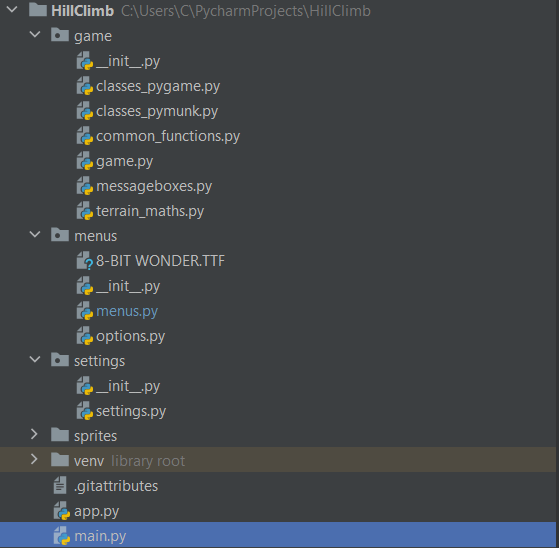
# Installation

Download the project from the GitHub directory:

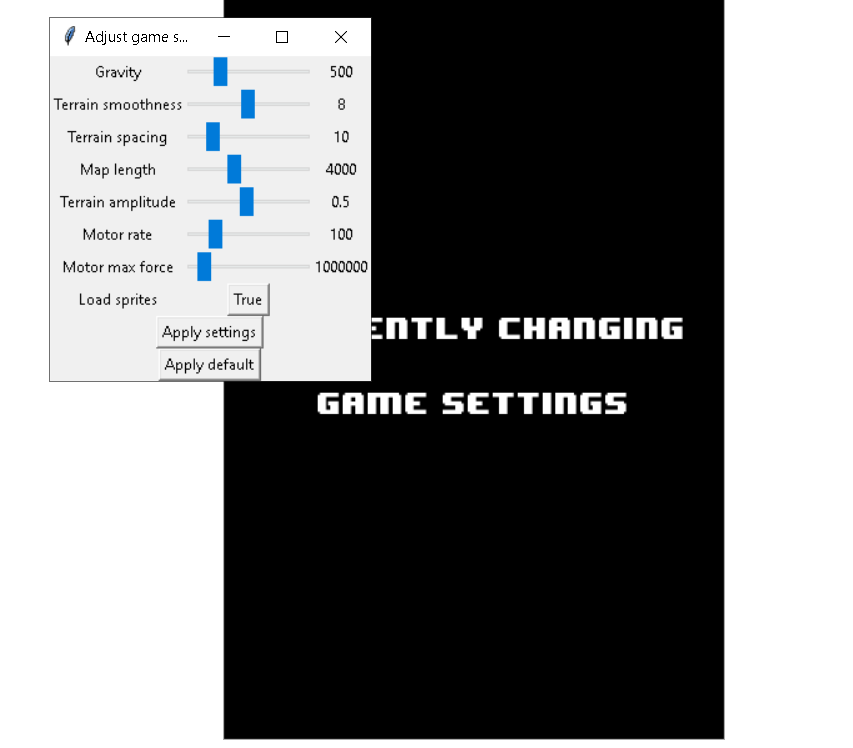
https://github.com/VelikTzar/HillClimb

Required libraries: Pygame, Pymunk, Tkinter

Project files and folders shown below:

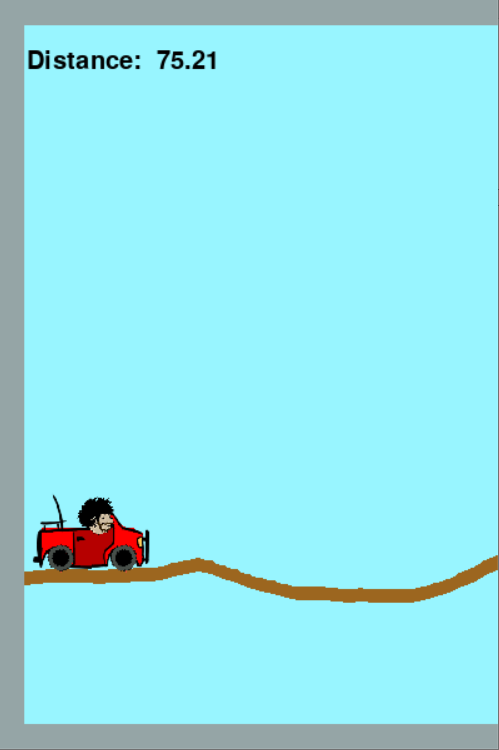
Run the main.py file to start the application.

# User interface

Main menu(left):  
Use up/down arrows to move the cursor. Press Enter to click the relevant option.

Options menu(right):  
Change the settings in the tkinter window. Unless you click apply settings, any changes in the game settings won’t be recorded. Click apply default to go back to default settings. The current settings will be applied to this instance of the app only (if you close the app, the settings will be reset to the default). To go back to the main menu, close the tkinter window. You will be returned to the main menu directly. If you want to close the app whilst in the options menu, close the tkinter window first, then the pygame window, otherwise the app won’t close.

Credits menu: Use the Esc, Backspace or the Enter button to go return to the main menu.



Game:  
In the Main Menu, press Enter whilst your cursor is on the START GAME option to start the game.

At any point during the game, if the player presses Escape, a message box will appear asking them whether they want to return to the main menu. Selecting No returns them to the game (thus this can be used as a pause button).   
A screenshot from the game is shown to the left.

The player is in control of a car with a driver inside, moving forward in a 2D map with randomly generated modifiable terrain (in brown)

The view moves automatically with the car. The distance (x – coordinate of the car chassis’ center) is displayed in the top left corner. The map’s size is visible in and can be changed in the options. The map is surrounded by a boundary (colored in grey to differentiate it from the brown terrain). of fixed thickness.

There are two controls – gas (applied with the right arrow key) and breaks (applied with the left arrow key). The controls modify the force applied by the motor, by a set value every frame the

relevant key is pressed. If no controls are applied, the car will slow down over time at a rate equal to the rate with which gas is applied.

WIN CONDITION: Reach the end of the map (more specifically, be within 2 \* boundary thickness + the Car Chassis’ width from the end of the map).

LOSS CONDITION: The driver’s head touches the ground. (This means that hitting the boundaries won’t result in a loss.)

WIN(right) and LOSS(left) messages are displayed on the next page.



GAME SETTINGS:

There are several settings about the game which can be modified.   
Non-default values, especially in the extreme, may cause strange behavior.

**Gravity** – defines the value of the simulation’s gravity. Very low gravity will lead to weightlessness.

**Terrain smoothness** – affects the smoothness of the terrain. View the terrain generation section for more information. Unless the spacing is large, values below 7 are generally recommended.

**Terrain spacing** – the terrain is constructed from points with randomly generated y-coordinates, but at equal spacing. This value defines how far apart the terrain points are.

**Map length** – defines how long the map is. For comparison, the car’s size is 80. The game window’s size is 400, which is the minimal value for the map length.

**Terrain amplitude** – defines how high the terrain can be in relation to height. A value of one would mean that the terrain *could* have a point that’s on the top edge of the game display which will obviously impede gameplay.

**Motor rate** – the motors on the two wheels try and maintain a constant angular velocity between the wheels and the chassis. This changes that rate (which is constant throughout the game).

**Motor max force** – the maximum force that the motor can apply on the wheels and chassis. In practice a higher value means the car can go faster. In game, the value is increased and decreased by using the gas and break buttons, but only up to a constant maximum value/0.1.

It is this “max rate” that this value changes. High values this may cause the car to behave erratically.

# CODE OVERVIEW

## CLASS APP

Class app’s code (with the exception of app.run())

A representation of the app loop

Class app is responsible for running the whole program, and contains the current overall state of the program, the game settings and generates instances of the Menu and Game classes, which then start the menu or game loops respectively.

Class app attributes:

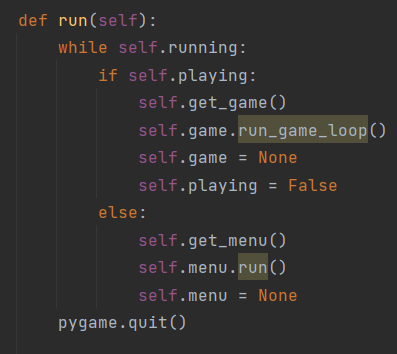
* self.menu (initially set to None) - an instance of the menu currently running. The instance is created by the get\_menu() method. After the menu loop ends, the value is set to None again.
* self.game (initially set to None) - an instance of the Game class. The instance is created by the get\_game() method. After the game loop ends, the value is set to None again.
* self.curr\_settings - an instance of the Settings class. This class contains the game settings which the player can change. When starting the program, an instance of the class is created with no arguments, which sets all its attributes to their default values. This attribute is replaced with a new instance of the class in the Options menu.
* self.curr\_menu(initially set to ‘MAIN’) – a string attribute. It defines which of the three menu classes will be created when the get\_menu() method is called. ‘MAIN’ for an instance of the MainMenu class, ‘OPTIONS’ for an instance of the OptionsMenu class, ‘CREDITS’ for an instance of the CreditsMenu class.
* self.running – a bool attribute. While true, the app loop continues. If false, the program is closed.
* self.playing – a bool attribute. If true, the get\_game() method is called, and then the game loop(Game.run\_game\_loop()) is started. If false, the get\_menu() method is called, and the menu loop is started(Menu.run()).

Class app methods:

* get\_game()

Creates an instance of the Game class, which receives the current instance of the App class as an argument.

* get\_menu()  
  After checking the value of the curr\_menu attribute, it creates an instance of the relevant Menu class, which receives the current instance of the App class as an argument.

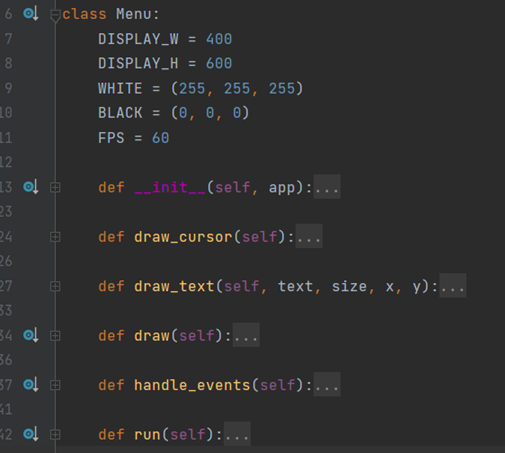


### The app.run() method

The method, which runs the whole program. A while loop which continues while self.running is True.

If self.playing is True, an instance of the Game class is created, and the game loop is started from the run\_game\_loop() method of the Game class. After that, self.game is again set to None, and self.playing is set to False(if the self.running is still True, this means we will return to the Main menu – as we cannot start the game from any other menu therefore self.curr\_menu will be ‘MAIN’). If self.playing is False, we get\_menu(), and run the relevant menu’s loop.

## CLASS MENU



A representation of the menu loop

Class Menu is the parent class to all three Menu classes, and defines all their common methods and attributes. No instances of this parent class are ever generated.   
Class Menu and all its children receive an instance of the App class which generated them. This allows them to access App’s attributes – self.running, self.playing, self.curr\_menu, self.curr\_settings

Class Menu attributes:

* app – the instance of the app which created the menu.
* display – a Pygame display.
* mid\_h, mid\_w – the coordinates of the middle of the display
* run\_display – a bool attribute. If false, the menu loop ends and we return to the app loop
* cursor\_rect – a Pygame rectangle, which we use for our cursor
* font\_name – set to './menus/8-BIT WONDER.TTF'. Defines the file path to the font which is used for the text in the menus.
* offset – set to - 100. Defines the offset for the text drawn by the different menus
* clock – a pygame clock. Used to maintain a constant FPS (60).

Class Menu methods:

* draw\_text(text, size, x, y)

Takes the text to be drawn, the size (aka the font size), and the x and y coordinates as arguments.

Blits the text to the Menu display, in a rect centered on x, y.

* draw\_cursor()

Calls draw\_text with ‘\*’ as the text, 15 as the size, and self.cursor\_rect.x and self.cursor\_rect.y as coordinates.

* draw()

Does not have a body (except pass). Initialized in the children classes, responsible for drawing the screen.

* handle\_events()

Does not have a body (except pass). Initialized in the children classes, responsible for handling the pygame events.

* run()

Runs the menu loop (displayed in an illustration on the previous page).

### CLASS MAINMENU

Inherits from the Menu class. Responsible for the main menu.

The controls, as mentioned above, are the up and down arrows for moving the cursor, and enter for selecting the option the cursor is on.

Class Main Menu unique (not inherited) attributes:

* start\_x, start\_y, options\_x, options\_y, credits\_x, credits\_y – store the coordinates for where the text will be drawn – start x, y for Start game, options x, y for Options, credits x, y for Credits
* state - a string attribute. Default value – ‘Start’ Holds which ‘button’ the cursor is on – ‘Start’, ‘Options’, ‘Credits’. Moving the cursor changes the state. When the enter button is pressed, the check\_input() method is called, which checks the value of the state attribute

Class Main Menu methods:

* draw()

Draws the menu text and cursor on the screen, using the parent class draw\_text and draw\_cursor methods.

* handle\_events()

Handles the events. If the player closes the app window, self.app.running is set to False which causes the app loop to end. If the player a key, this calls the move\_cursor() method, giving the key as an argument(it does anything only if the up or down arrow key are pressed). If the player presses the Enter key, this calls the check\_input() method.

* move\_cursor(button)

Accepts the key pressed as an argument. If the key is the up arrow key, the cursor moves up. If the key is the down arrow key, the cursor moves down. The cursor loops (pressing up when at the uppermost option will bring the cursor to the bottommost option). This is done by checking the current state – which corresponds to the location of the cursor. Depending on the state attribute and the direction, the cursor\_rect attribute’s coordinates are changed, which causes the cursor to be drawn at a different place during the next iteration of the menu loop. The state attribute is changed to reflect the current location of the cursor.

* check\_input()

Called when the Enter button is pressed. The method checks the state attribute. If it is ‘Start’ – this changes the run\_display to False, causing the menu loop to end. It also sets app.playing to True, so during the next iteration of the app loop the game loop will start. If it is ‘Options’ – this changes the run\_display to False, again causing the current menu loop to end. It then changes app.curr\_menu to ‘OPTIONS’ – so that during the next iteration of the app loop an OptionsMenu will be created, and its menu loop will be run. The same happens if the state is Credits, but with app.curr\_menu being set to CREDITS instead, which causes a CreditsMenu to be created.

### CLASS OPTIONSMENU

Inherits from the Menu class. Responsible for the options Menu.

The controls are sliders and button in the tkinter window, which can be pressed with the mouse.

Class OptionsMenu shares all of its attributes with the parent class.

Class OptionsMenu methods:

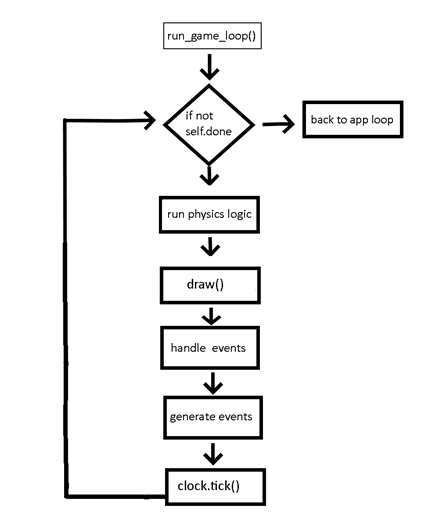
* draw() – draws the menu text on the pygame display
* handle\_events() – Handles the events. If the player closes the app window, self.app.running is set to False which causes the app loop to end.
* run()

Unlike with the other Menus, the run method does not start a loop. Instead, it draws the relevant text on the screen, and creates a tkinter window, which contains all the sliders and buttons for changing the settings. Once the tkinter mainloop is over, the run method sets the app.curr\_menu attribute to “Main” and the player is thus returned to the Main Menu. The tkinter window is actually given as an argument to create an instance of the Options Window class, which in its run method creates all the sliders and buttons, and associated with them functions. Upon clicking one of the apply settings/apply default buttons, an instance of the Settings class is created. If apply settings is clicked, then the value from the sliders and loadsprites button are taken, and given as arguments to the Settings class. If apply default is pressed, an instance of the Setting class with no arguments supplied is created (the default). The values that can be adjusted from the Options menu are all listed in the Interface overview, and will be shown again further below when the Settings class is discussed. After that, a messagebox will appear telling the user that the settings have been applied.   
Closing the tkinter window, or pressing the apply settings/apply default buttons will return the player to the main menu.

### CLASS CREDITSMENU

The simplest menu. Inherits from Menu. Inherits all attributes from Menu. The methods draw and handle\_events() are changed, with draw drawing the relevant text on the screen, and handle\_events – closing the app window results in self.app.running is set to False which causes the app loop to end. Pressing Enter, Escape or Backspace results in self.app.curr\_menu being set to ‘MAIN’ and self.run\_display to False – resulting in the app loop continuing, and displaying the main menu in its next iteration.

## CLASS GAME



A representation of the game loop

****Class game is responsible for creating, running and displaying the actual game. It accepts an instance of the app class as an argument when initialized.

### Class game’s attributes

There are two main groups of attributes   
The first are ones that are about the game settings – which are then supplied to other classes when creating the game. They are pretty self – explanatory, and their role will be shown when examining other classes. They are  
self.car\_rate, self.max\_force, self.height\_amplitude, self.smoothness, self.gravity, self.terrain\_spacing, self.load\_sprites, self.width  
They are all given a value of None initially, then another attribute – self.settings is set to app.curr\_settings, and then the method load\_settings is called with self.settings as an argument. The method just sets all of these attributes to the value given in self.settings.

The second group are the following

* self.app – gives the game access to the App class’ attributes – for example allows us to change app.running to False and close the whole program.
* self.done – a bool attribute, which controls our game loop.
* self.clock – a Pygame clock, which allows us to maintain a constant FPS
* self.display\_window – a Pygame window which is our actual display
* self.window – a Pygame surface, with height equal to the one of self.display\_window, and width which can be changed from the Options menu(this is the self.width attribute). This contains the map for the game. We blit a part of it to self.display­\_window every frame
* self.space – a Pymunk space, which accepts self.window as an argument. This is a pymunk object which will hold all other pymunk objects, and this is where the simulation actually happens. The gravity of the space is set to self.gravity
* self.sprite\_group - a Group, containing all of the sprites. In this project, I’ve created a child class to Sprite, Entity, and it is used solely in the draw() phase of the game loop. Entity instances do not interact with the simulation below.
* self.objects\_group – a list, containing instances of other classes, which pertain to the game(e.g. victory handler)
* self.camera – holds an instance of the camera class. The camera class gives us the offset of the part of self.window that we will blit onto the display(self.display\_window)

## The game loop

In the app loop, after creating an instance of the Game class with all of these attributes, we call the Game.run\_game\_loop() method. We will describe it below, as it contains everything necessary to run the game, and involves all other methods and classes, defined in the files in the game folder.

The run\_game\_loop() method first does everything necessary to set-up the game, and then calls the run() method which actually executes the game loop. We will look at these two parts in turn.

### Part 1: Game set-up

First, we call pygame.init().   
After that, we generate the game terrain and map boundaries (they are pymunk objects, existing in the simulation occurring in self.space). How that happens will be explained in detail later. We do it by creating instances of the Terrain and Boundaries class, and calling the Terrain.generate\_terrain() and Boundaries.generate\_boundaries() respectively. The end result is creating the boundaries and terrain in the simulation.

Then we call the Terrain.return\_spawn() method, which gives us the spawn point for our player. We do this, because, since the terrain is randomly generated, and the player can modify it to a large extent, using a fixed value might result in the car getting stuck into or under the terrain. With this spawn point and self.space as arguments, we create an instance of the Player class(it then creates instances of the Car and Person classes. This will be shown in more detail on the sections about these classes). It creates all the Pymunk bodies in the simulation, and contains references to them.   
We then create instances of 4 different classes - CarMovementHandler(which accepts the reference to the car class from player – player.car), HeadCollisionHandlePygame(which accepts self as argument), VictoryHandler(which accepts self, and a reference to the player) and Messagebox(which accepts self). By accepting the reference to this instance of the Game class, these classes can modify and interact with its attributes. All of these objects are appended to the objects\_group attribute of the Game class.   
Then, if self.load\_sprites is True, we load the various sprites for the game, and add them to the sprites\_group attribute.   
Then we set our camera objects(self.camera) to follow the position of the car by using its set\_obj method and supplying an instance of the CarMovementHandler class.

### Part 2: The Game Loop

The game loop is executed by calling the self.run() method, which starts a while loop which continues as long as the bool attribute self.done is False. The game loop consists of five steps, represented by five methods which are called

#### Step 1: Update the simulation

Done by calling the self.run\_logic() method. It updates the pymunk .Space by calling its step method, and supplying the time interval passed(1/FPS = 1/60) as an argument. Then the sprite group is updated.

#### Step 2: Draw

Done by calling the self.draw() method. We first draw everything on the window, and then blit a portion of it on our display. We draw the simulation on our window by using the space’s method debug\_draw(which accepts as its argument an instance of the DrawOptions class. We can also set flags for what will be draw. In our case, we only draw the pymunk shapes and their collision points on the screen). Then we draw our (now updated thanks to the last step) sprites at their new locations and angles. We then get the offset from the camera – by calling its follow\_coordinates() method. We blit the relevant part of the window on the display. The last step is calling the self.show\_distance() method, which draws the current distance passed by the player in the top left of the display.

#### Step 3: Handle events

Done by calling the self.handle\_events() method. We loop over the current pygame events. If the player has closed the game window, we set self.done to True and self.app.running to False – thus terminating the game and app loops.   
If the player did not close the game, we call the handle\_event() method of the objects inside the objects\_group attribute, supplying one event at a time as an argument.

After handling the events, the method self.handle\_pressed\_keys() is called. It also loops over the objects in the objects\_group, calling their handle\_keys() method.

#### Step 4: Handle event generation

Done by calling the self.handle\_event\_generation() method. Loops over objects in the objects\_group attribute, calling their generate\_event() method

#### Step 5: Constant frame rate

We maintain constant frame rate by using a pygame.Clock() object, and ticking 1/FPS every iteration of the game loop.

## SPRITES – CLASS ENTITY

Class entity inherits from the pygame sprite class. It is almost identical, save for the fact it is attached to a pymunk body, whose coordinates and angle it follows through its update method. The entity class is used solely for drawing, and plays no part in the simulation.

## About the simulation

The simulation is conducted in a pymunk space object. The space holds all other pymunk objects. For more information about the pymunk library, refer to the pymunk documentation. <https://www.pymunk.org/en/latest/>

## CLASS PLAYER

Class Player is the class which generates all of our pymunk bodies and shapes. It does so by calling the classes Car and Person, and joining them together with several pymunk joints.

### Class Car

The actual car itself, with the wheels. It receives a spawnpoint and a pymunk space as arguments. The class has the following attributes:

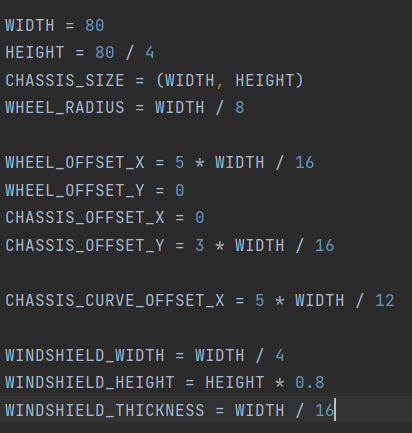
space – the pymunk space where all of the bodies and shapes are added to  
chassis\_body – the body of the car chassis  
chassis\_shape - shape of the main part of the chassis. A polygon. Attached to the chassis\_body  
windshield\_shape – shape of the windshield. A polygon. Attached to the chassis\_body  
wheel1\_body  
wheel1\_shape   
wheel2\_body  
wheel2\_shape

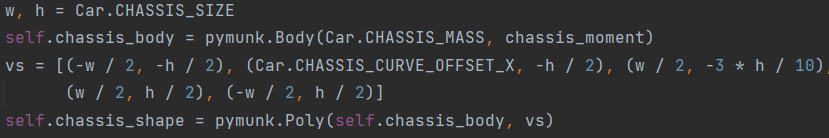
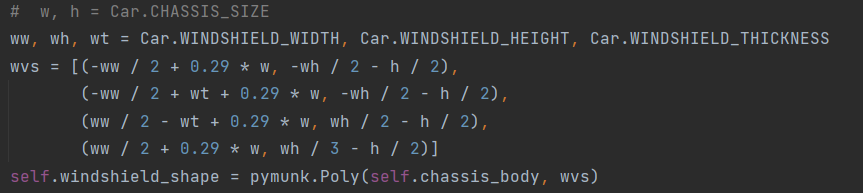
The upper four attributes correspond to the shapes and corresponding bodies for the 2 wheels.

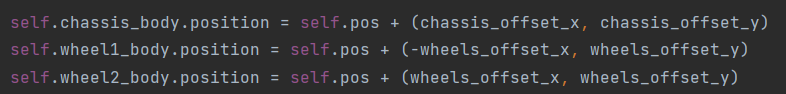
The other six attributes correspond to the six constraints, holding the car together.

spring1

spring2  
groove1  
groove2  
motor1  
motor2

These are the pymunk DampenedSpring, GrooveJoint and SimpleMotor.  
The sizes and positions of all of these objects are relative to the spawn point supplied to Car, and to the WIDTH of the Car.   


A lot of the values regarding size are defined as class attributes. The windshield is offset compared relative to the car width as well. Its vertices and the one of the chassis are shown below

The coordinates of these vertices are relative to the center point of the bodies. Then, before adding the objects to the space, we set the absolute coordinates of the center points. 

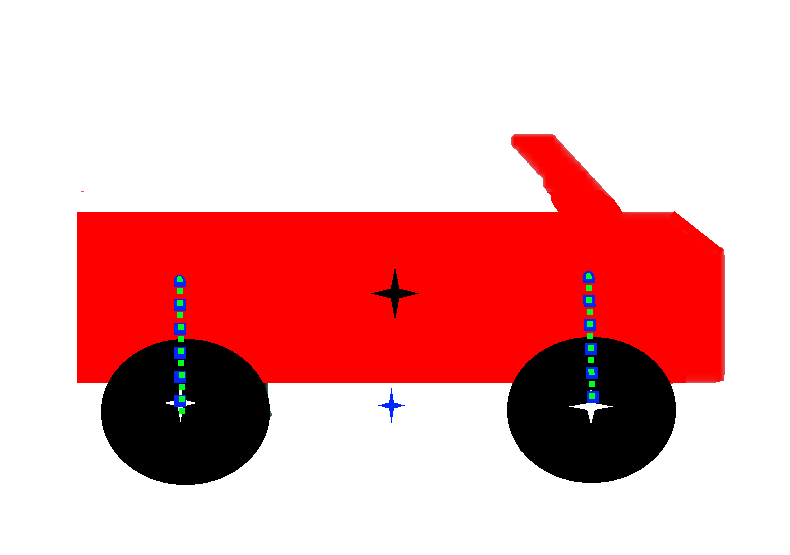
The Car class has 4 methods. One – get\_velocity() just returns the velocity of the chassis. The others are:

gas() - increases the maximum force of the motor. In practical terms this accelerates the car. The cap to how high the value can go, as well as how much it is increased by, are the MAX\_FORCE and ACCELARATION class attributes, respectively.

break() - decreases the maximum force of the motor. In practical terms this deccelarates the car. How much it is increased by corresponds to the DECCELARATION class attribute.

slow\_down() – like break, but with a smaller amount(it uses the acceleration value here, as it is 10 times smaller than the break one).

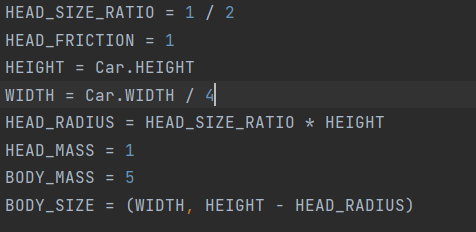
For comparison,

The starting max\_force value for the motors is a 1000.   
The default cap (MAX\_FORCE) is 100000. It can be adjusted from the setting.  
ACCELERATION is 1000. DECCELERATION is 10000.

Representation of the car. The chassis is in red, the wheels in black. The blue star is the supplied spawnpoint, the black star is the center of the chassis, and the white stars are the centers of the wheels.   
The blue and green lines represent the groove and spring joints, starting and ending at their respective anchor points

Class Person:

Responsible for creating the driver, represented by 2 pymunk bodies – one for the head, with an attached Circle shape, and one for the body, with an attached Box shape (a rectangle).  
As with the Car class, there are class attributes used for the size etc. Again, the size is based on ratios compared to the Car.WIDTH attribute. Accepts the spawnpoint and the space as arguments.

 The class attributes regarding size shown to the left.

The instance attributes are:

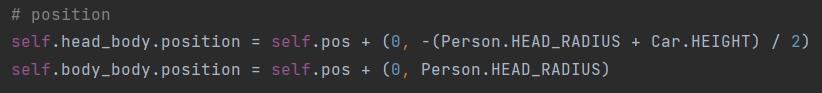
space – a pymunk space where to add the objects

body\_body – the pymunk body for the body of the driver  
body\_shape – the pymunk shape for the body of the driver  
head\_body – the pymunk body for the head  
head\_shape – the pymunk shape for the head.

**The head shape has a special collision type, as does the terrain. This special collision type is represented by an integer. When two objects with special collision types meet in pymunk, we can create a custom collision handler that describes how the collision should proceed. This is how we detect and process loss – aka the head hitting the ground.**

pivot1 – a pivot joint between the head and body  
spring1- a spring connecting the center of the head and of the body

The class has no methods other than \_\_init\_\_()



### Class Player

Class Player accepts a spawnpoint (x, y coordinates in the pymunk space) and a pymunk space as its arguments. It then uses these to create instances of the Car and Person class, mentioned above.

These instances are attributes of the Player class, and therefore all of their methods and attributes are accessible through the player class.

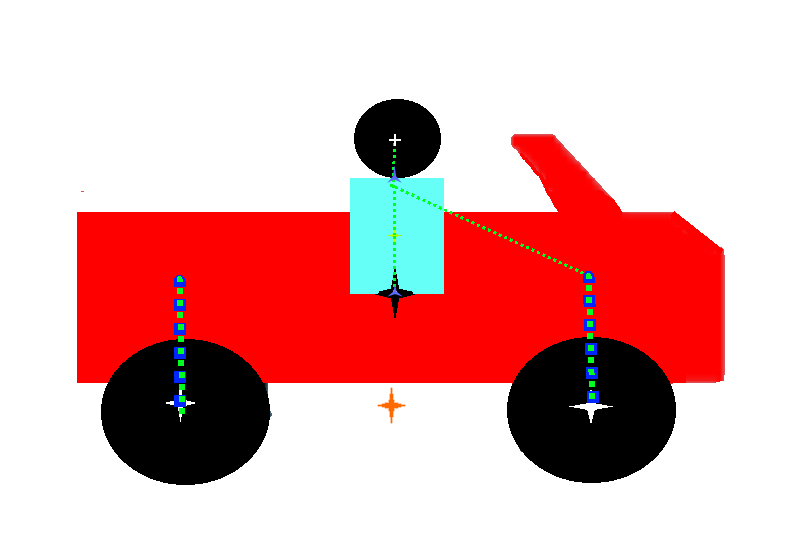
The player class’ attributes are:

car – refers to an instance of the Car class

person – refers to an instance of the Person class

pivot1 – a pivot joint connecting the center of the base of the body to the chassis center

spring1 – a spring joint connecting the center of the body to the chassis center

spring2 – a spring joint connecting the top of the body to the anchor point of the joints connecting of the front wheel

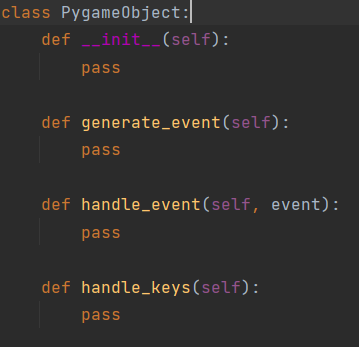
A representation of all the bodies, shapes and constraints that are created via the Person class.

4 –pointed stars – centers of the respective shapes.   
The orange star represents the spawn point supplied to the person class.   
The blue three pointed start represent pivot joints.

The blue dashed lines represent groove joints, from one anchor point to the other. The green dashed lines represent the dampened springs, again from one anchor point to the other

## Game.objects\_group objects

All of the objects in the game class’ instance attribute objects\_group inherit from the same parent class – PygameObject, and share the same methods.   
  
CLASS PygameObject

The PygameObject class has three methods in addition to \_\_init\_\_, all of them have no bodies.  
The class is never directly instantiated, only its children are.

These methods are called during the game loop.   
handle\_event and handle\_keys are called for each object in the object group by Game.handle\_events() ( which also calls Game.handle\_pressed\_keys(), which calls handle\_keys() for each object in the object group).  
generate\_event is called by the Game.handle\_event\_generation () for every object in the object group.   
This inheritance is done so that there need not be specific attributes for every single object that concerns the game. Instead, we can iterate over them all, as they all share all of these methods, even if some are without a body for most of the classes.

### CLASS CarMovementHandler

Inherits from PygameObject. Accepts an instance of the Car class described above. Responsible for handling the movement and current position of the car (and thus, by extension, the player).   
Instance attributes:  
car - an instance of the Car class

Methods:

handle\_event

Handles events. Accepts one pygame event as an argument. If the right arrow key is pressed, the gas() method of the Car instance is called. If the left arrow key is pressed, the break() method is called.

handle\_keys(

Checks pressed keys. If the right arrow key is pressed, the gas() method is called. If the left arrow key is pressed, the break() method is called. Otherwise, the slow\_down() method is called.

Because of this, the accelerates and decelerates for every frame in which the right/left arrow keys are held, and automatically slows down if no controls are applied – this was done because otherwise the player could get the car to a certain constant speed and watch it passively complete the game. Now the gameplay is more unpredictable, as if acceleration is not applied the car will eventually stop, but at the same time it is very easy for the acceleration to get out of control causing the car to flip.

There is one more method – get\_coordinates, which returns the coordinates of the chassis\_body. It is used to allow the camera to track the player(the camera has an obj attribute, which is later set to an instance of the CarMovementHandler. Through that the current distance is displayed on the screen, and the offset is constantly updated to keep the player in the middle of the screen).

### CLASS HeadCollisionHandlerPygame

Inherits from PygameObject. Accepts an instance of the Game class. Is responsible for responding to the head of the player colliding with the terrain, alongside another class – HeadCollisionHandler

Class attributes – HEADCOLLISIONEVENT – a custom pygame event

Instance attributes –

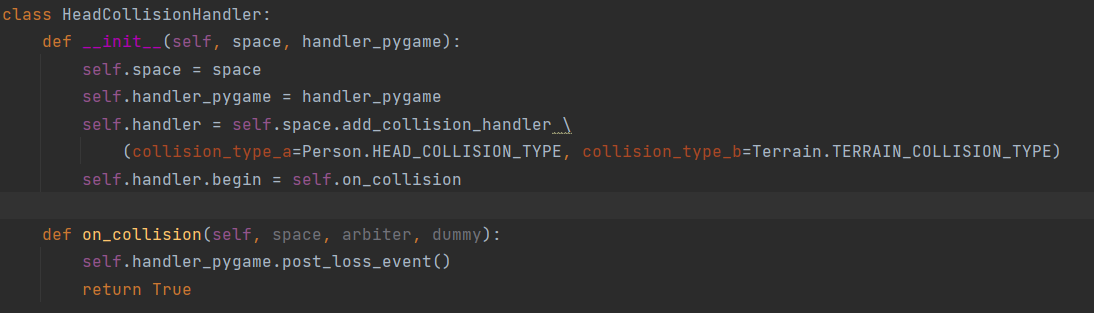
collision\_event – equal to the HEADCOLLISIONEVENT class attribute

game – an instance of the game class

head\_collider – creates an instance of the HeadCollisionHandler class, which accepts a pymunk space(taken from game) and the current instance of HeadCollisionHandlerPygame.

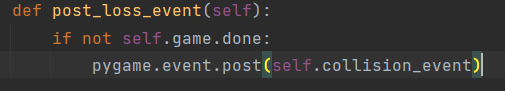
#### HeadCollisionHandler

Adds a special collision handler to the pymunk space. As mentioned above, the head and terrain have special collision types, represented by integers.

  
On collision, the self.handler\_pygame’s(aka the instance of HeadCollisionHandlerPygame supplied when creating the class) method post\_loss\_event() is called.

HeadCollisionHandlerPygame methods –

post\_loss\_event



Posts the custom pygame loss event defined as a class attribute. Originally this was done by checking whether there was a collision every iteration of the game loop through a generate\_event() method, but this method, utilizing pymunk’s existing functionality, is far easier and one doesn’t need to check manually every iteration.

### CLASS VictoryHandler

Inherits from the PygameObject class. Responsible for checking whether or not the player has met the victory condition and posting an event if he does. Accepts an instance of the Game and Player classes.

Class attributes – VICTORY\_EVENT – a custom pygame event

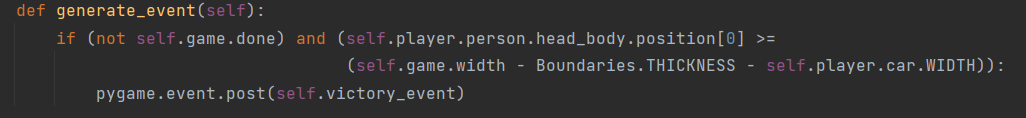
Instance attributes –

victory\_event – equal to the class attribute

game – an instance of the Game class.

player - an instance of the Player class.

Methods:  
generate\_event

If the game is still ongoing, and the player’s head’s center’s x coordinate is within a certain distance (Boundary thickness plus the Car width) the victory event is posted.   
This method is called in the handle\_event\_generation() part of the game loop every iteration.

### CLASS MessageBox

Inherits from the PygameObject class. Responsible for reacting to certain events that require a change in the game state. Accepts an instance of the Game class (which it uses to access its instance attributes).

Instance attributes:   
fired – a bool attribute, that exists to prevent pop-ups from appearing more than once.

game – an instance of the Game class

Methods:

handle\_event

Accepts one pygame event as an argument. If the event is the custom loss event, the custom win event, or the escape key has been pressed, a relevant tkinter messagebox is displayed. This is done by instantiating one of three “messagebox” classes, and then their method run() is called.

If the message was for win/loss, self.fired is set to True

The messageboxes are defined in the file messageboxes.py.  
They all inherit from a base class MessageBox.  
They accept an instance of the Game class as an argument. They use it to access the game and app instance attributes, such as app.running and game.done

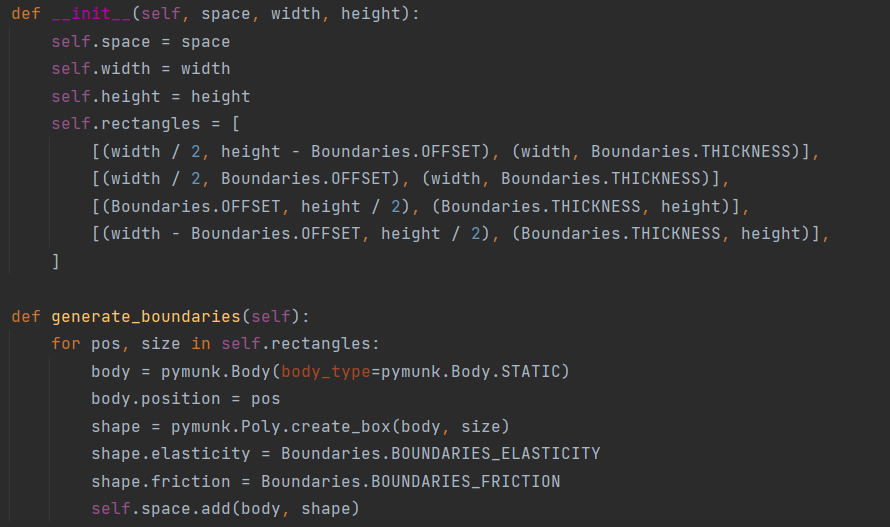
All of the messageboxes are in the form of a yes/no question.   
For a EscapeMessageBox, pressing yes (by setting game.done to True), will return the player to the Main Menu, and pressing no will return the player to the game(by not changing anything). In that way, the Escape button can act as a pause button – as the game loop is put on hold whilst the tkinter loop created by the MessageBox handle\_event method is closed.

For a Loss/VictoryMessageBox(which both inherit from a class OutcomeMessageBox, which itself inherits from MessageBox), pressing yes will return the player to the Main Menu(by setting game.done to True), and pressing no will close the game(by setting app.running to False)

## TERRAIN GENERATION – Classes Boundaries and Terrain

### CLASS Boundaries

Creates static pymunk bodies, with attached shapes – rectangles of fixed thickness, on the edges of the map. Accepts the pymunk space, and the width and height as arguments.   
  
It has one method – generate\_boundaries(), which actually creates the four boundaries.



### CLASS Terrain

The terrain class also accepts the space, width and height as arguments upon initialization. It is responsible for creating our semi-random terrain every game.

It has three methods:

create\_segment – a static method, which saves work when creating a pymunk Segment. Returns the body of the segment, and adds the segment to the space. Accepts the pymunk Space, a radius (how thick the segment is) and coordinates for the 2 anchor points (a and b)

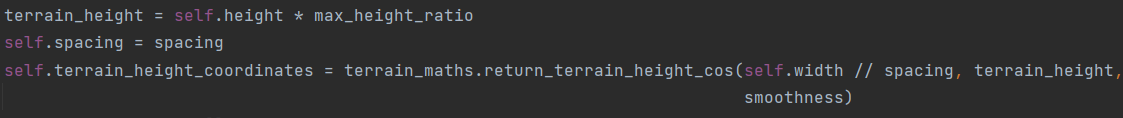
return\_spawn – after the terrain has been generated, returns a spawn point slightly above the terrain. It checks the highest terrain point that’s after the boundary in range as big as the car width, so that the car doesn’t end up stuck partly in the terrain.

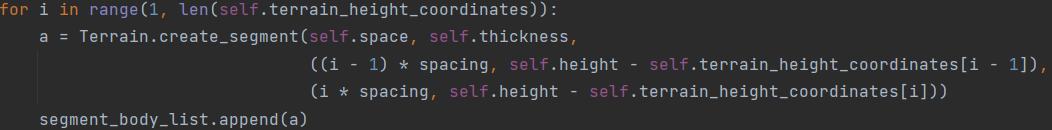
And finally, the method that actually generates the Terrain –

### generate\_terrain()

Accepts three arguments – spacing, max\_height\_ratio, and smoothness (default = 8)

All these values can be modified from the Options Menu (which then creates a Settings object, from which the Game class takes the values and uses them when calling the generate\_terrain method when starting up the game)

The max\_height\_ratio(corresponding to height amplitude from the options menu) says how high our terrain points could be. The smoothness is also supplied, which we will look at below.

The return\_terrain\_height\_cos() function returns a list of y coordinates. The x coordinates for all of the points which generate the terrain are equally spaced, by a value corresponding to spacing.  
We iterate over this list, and create segments between every 2 points. The exact code is shown below.

The thickness is a Class attribute of the Terrain class, and is thus constant.

Now let’s look at how we actually create this list to get somewhat realistic looking terrain.

### The terrain generation functions

#### return\_terrain\_height\_cos()



Accepts the number of points, the height, and the smoothness as arguments.

In turn it returns the value of another function(terrain\_superpos\_cosp) and supplies it with arguments in the form of the result of a third function (terrain\_naive, which accepts the count and height) and the smoothness. Let’s now look at these two functions.

The following code has not been written by me. I had the idea of using randomly generated points, but did not know how to modify them. Whilst I was searching material on the topic, I found a blogpost, outlining how and what to do to generate 1D terrain, and had detailed code showing the process. The code given suited my purposes very well. You can see the original article here - https://arpitbhayani.me/blogs/1d-terrain.

#### terrain\_naive()

Accepts a number of points, alongside a maximum value(height). It returns a list of randomly generated points mapped onto the new range (0, height).

#### terrain\_superpos\_cosp()

The code, with comments explaining every line, is shown below.

def terrain\_superpos\_cosp(naive\_terrain, iterations=8):  
 *"""Using naive terrain `naive\_terrain` the function generates  
 Cosine Interpolated Superpositioned terrain that looks real world like.  
 """* terrains = []  
  
 # holds the sum of weights for normalization  
 weight\_sum = 0  
  
 # for every iteration  
 for z in range(iterations, 0, -1):  
 terrain = []  
  
 # compute the scaling factor (weight)  
 weight = 1 / (2 \*\* (z - 1))  
  
 # compute sampling frequency suggesting every `sample`to  
 # point to be picked from the naive terrain.  
 sample = 1 << (iterations - z)  
  
 # get the sample points  
 sample\_points = naive\_terrain[::sample]  
  
 weight\_sum += weight  
  
 for i in range(len(sample\_points)):  
  
 # append the current sample point (scaled) to the terrain  
 terrain.append(weight \* sample\_points[i])  
  
 # perform interpolation and add all interpolated values to  
 # to the terrain.  
 for j in range(sample - 1):  
 # compute relative distance from the left point  
 mu = (j + 1) / sample  
  
 # compute interpolated point at relative distance of mu  
 a = sample\_points[i]  
 b = sample\_points[(i + 1) % len(sample\_points)]  
 v = cosp(a, b, mu)  
  
 # add interpolated point (scaled) to the terrain  
 terrain.append(weight \* v)  
  
 # append this terrain to list of terrains preparing  
 # it to be superpositioned.  
 terrains.append(terrain)  
  
 # perform super position and normalization of terrains to  
 # get the final terrain  
 return [sum(x)/weight\_sum for x in zip(\*terrains)]

The smoothness value corresponds to the number of iterations.