Salmon RPG

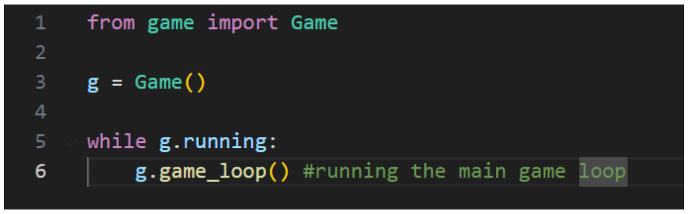
Overview
Salmon RPG is a role-playing game built using Python and Pygame. The game features creating a player character that can move around a map, engage in battles with enemies, and level up by gaining experience points. The game is structured into several modules and classes, each responsible for different aspects of the game.

Contents

- **Modules and Classes:**
    - **Main Modules**
    - **Game States**
    - **Supporting Classes**
- **Configuration**
- **Assets**
- **Proof of Working Code**

Modules and Classes

Main Modules
1. **main.py**

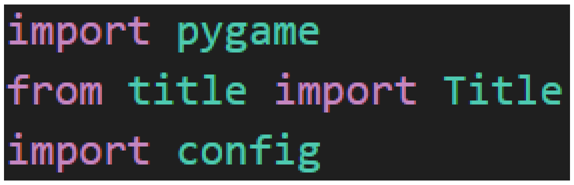
- Entry point of the game.
- Initializes the game and starts the main game loop.
- 

```python
from game import Game

g = Game()

while g.running:
    g.game_loop() #running the main game loop
```

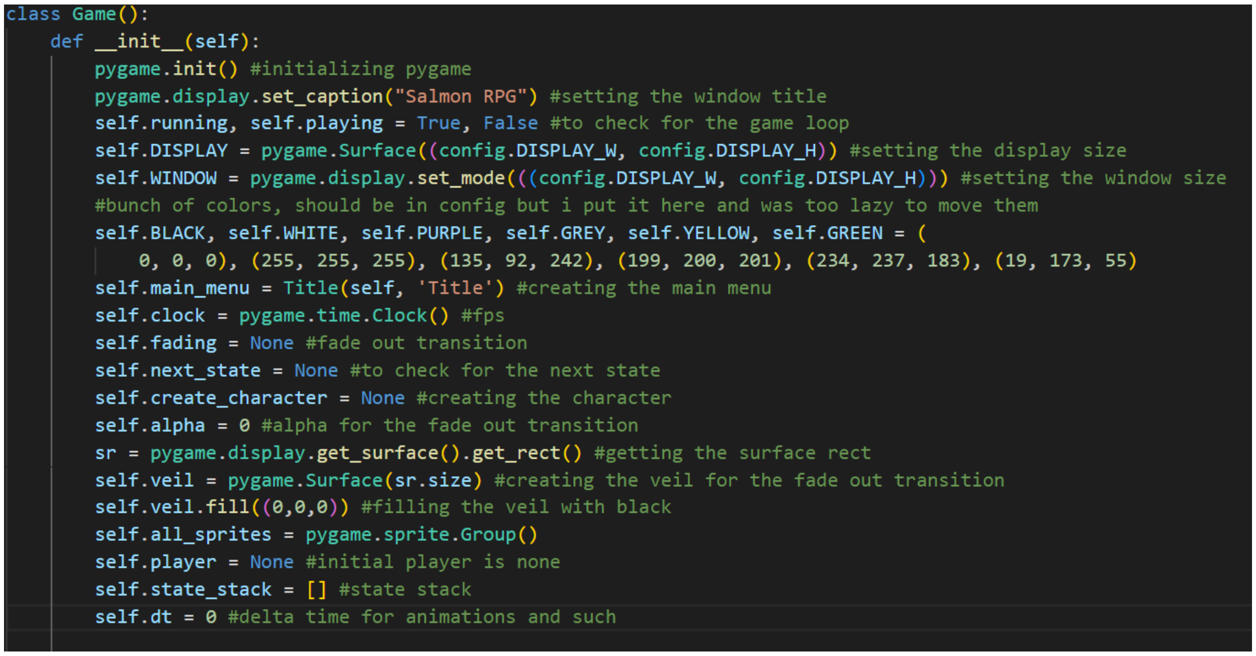
- I essentially just imported the Game class from the game.py file and initialized the loop.

2. **game.py**

- Manages the overall game state and transitions between different states (e.g., title screen, character creation, game world, battle, game over).
- Handles input, rendering, and updating game objects.

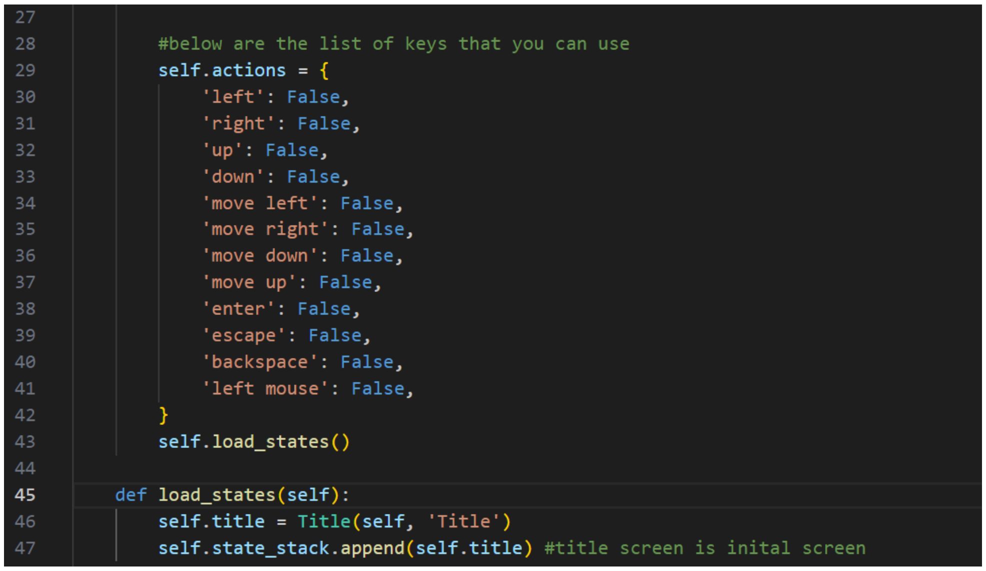
- 

```python
import pygame
from title import Title
import config
```

- Initial imports to make sure the game loop runs smoothly (e.g. initial screen being the title screen from Title class, display width and height from config)
- 

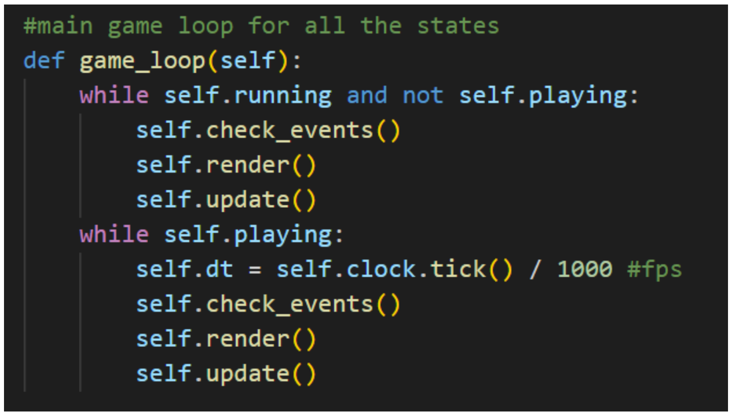
```python
class Game():
    def __init__(self):
        pygame.init() #initializing pygame
        pygame.display.set_caption("Salmon RPG") #setting the window title
        self.running, self.playing = True, False #to check for the game loop
        self.DISPLAY = pygame.Surface((config.DISPLAY_W, config.DISPLAY_H)) #setting the display size
        self.WINDOW = pygame.display.set_mode(((config.DISPLAY_W, config.DISPLAY_H))) #setting the window size
        #bunch of colors, should be in config but i put it here and was too lazy to move them
        self.BLACK, self.WHITE, self.PURPLE, self.GREY, self.YELLOW, self.GREEN = (
            0, 0, 0), (255, 255, 255), (135, 92, 242), (199, 200, 201), (234, 237, 183), (19, 173, 55)
        self.main_menu = Title(self, 'Title') #creating the main menu
        self.clock = pygame.time.Clock() #fps
        self.fading = None #fade out transition
        self.next_state = None #to check for the next state
        self.create_character = None #creating the character
        self.alpha = 0 #alpha for the fade out transition
        sr = pygame.display.get_surface().get_rect() #getting the surface rect
        self.veil = pygame.Surface(sr.size) #creating the veil for the fade out transition
        self.veil.fill((0,0,0)) #filling the veil with black
        self.all_sprites = pygame.sprite.Group()
        self.player = None #initial player is none
        self.state_stack = [] #state stack
        self.dt = 0 #delta time for animations and such
```

- The comments should be enough to briefly explain what these are for, but some specific lines such as **self.running and self.playing, self.main_menu, self.all_sprites, self.veil, self.state_stack, and self.dt** will be explained below.
- **self.running and self.playing:** It's used to control whether the game loop runs or not. If self.running is true, then the title and character creation screen will load, and the loop will run. If self.playing is true, then the world will load, and all the loops within the game world will run.
- **self.main_menu:** It dictates which screen will come up first in the game loop, which is the title screen, hence the Title class import to the game.py file.
- **self.all_sprites:** This creates a group for all the sprites, so that they can be manipulated easily using the pygame module.
- **self.veil:** It creates an overlapping black screen that slowly fades in and out, serving as a transition for moving from screen to screen.
- **self.state_stack:** It basically stores states that will be used as the main state, it will be explained later in the main game loop and state.py file
- **self.dt:** It ensures that animation or movement speed stays the same regardless of the device frame rate.

- 

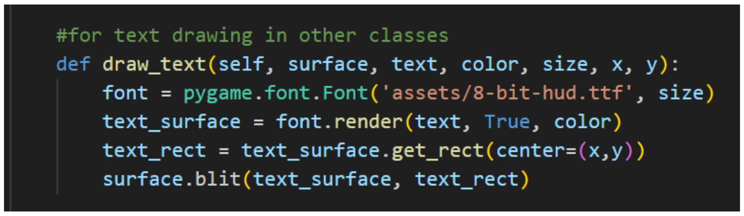
```python

        #below are the list of keys that you can use
        self.actions = {
            'left': False,
            'right': False,
            'up': False,
            'down': False,
            'move left': False,
            'move right': False,
            'move down': False,
            'move up': False,
            'enter': False,
            'escape': False,
            'backspace': False,
            'left mouse': False,
        }
        self.load_states()

    def load_states(self):
        self.title = Title(self, 'Title')
        self.state_stack.append(self.title) #title screen is inital screen
```

- **self.actions:** For controlling key presses, will be mentioned again further in the file.
- Here, you can see that load_states is called, so that the first screen that appears is the main menu screen.

- 

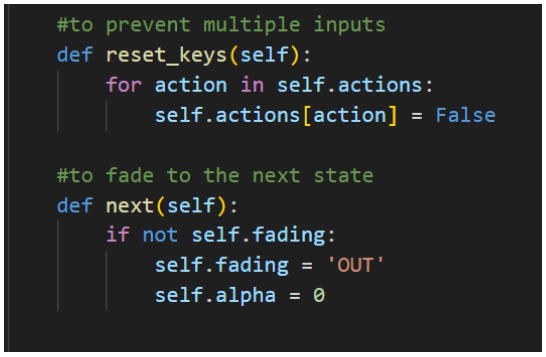
```python
#main game loop for all the states
def game_loop(self):
    while self.running and not self.playing:
        self.check_events()
        self.render()
        self.update()
    while self.playing:
        self.dt = self.clock.tick() / 1000 #fps
        self.check_events()
        self.render()
        self.update()
```

- The main game loop includes a bunch of functions that will be explained later.

- 

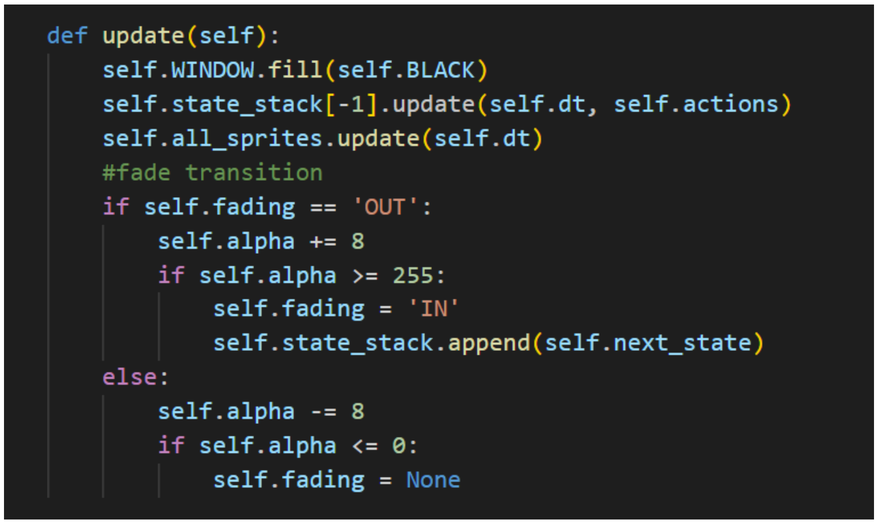
```python
    #for text drawing in other classes
    def draw_text(self, surface, text, color, size, x, y):
        font = pygame.font.Font('assets/8-bit-hud.ttf', size)
        text_surface = font.render(text, True, color)
        text_rect = text_surface.get_rect(center=(x,y))
        surface.blit(text_surface, text_rect)
```

- Just to make life a bit easier, I put down a function for text drawing so I don't need to manually input the text one by one.



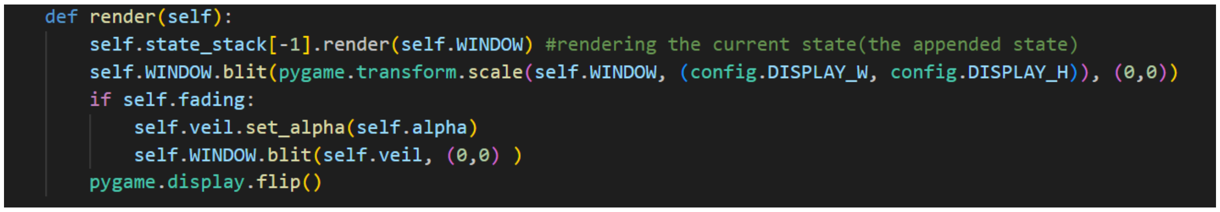
```python
    #to prevent multiple inputs
    def reset_keys(self):
        for action in self.actions:
            self.actions[action] = False

    #to fade to the next state
    def next(self):
        if not self.fading:
            self.fading = 'OUT'
            self.alpha = 0
```

-
- Reset keys ensure that the keys are properly reset, and not held down all the time.
- Next is to initialize the fading transition.



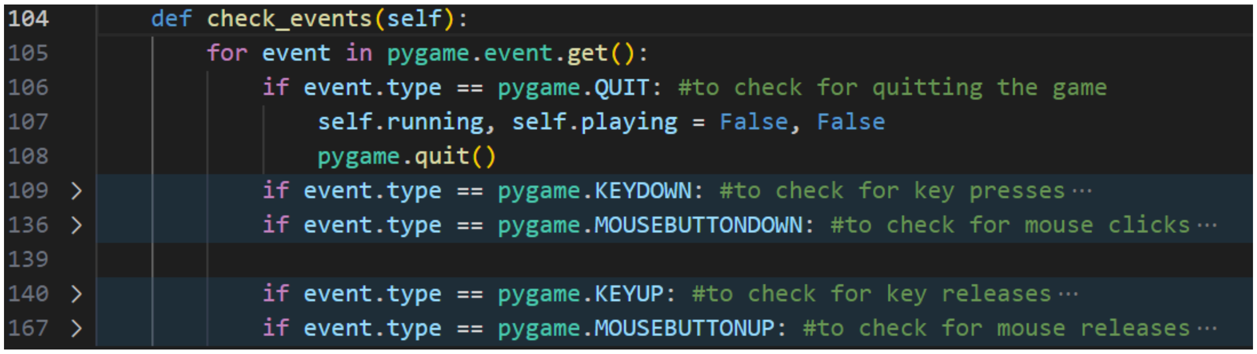
```python
    def update(self):
        self.WINDOW.fill(self.BLACK)
        self.state_stack[-1].update(self.dt, self.actions)
        self.all_sprites.update(self.dt)
        #fade transition
        if self.fading == 'OUT':
            self.alpha += 8
            if self.alpha >= 255:
                self.fading = 'IN'
                self.state_stack.append(self.next_state)
        else:
            self.alpha -= 8
            if self.alpha <= 0:
                self.fading = None
```

-
- In the update function, you can see that the state stack updates using the last item in the state stack list, meaning that the main state will be the last in the state stack list.
- The fading transition is also controlled here.



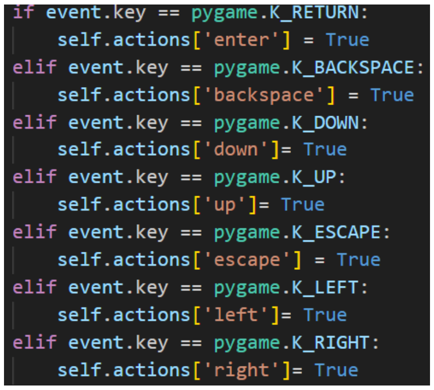
```python
    def render(self):
        self.state_stack[-1].render(self.WINDOW) #rendering the current state(the appended state)
        self.WINDOW.blit(pygame.transform.scale(self.WINDOW, (config.DISPLAY_W, config.DISPLAY_H)), (0,0))
        if self.fading:
            self.veil.set_alpha(self.alpha)
            self.WINDOW.blit(self.veil, (0,0) )
        pygame.display.flip()
```

-
- In the render function, the state stack controls which state will be rendered. (for example: if the last state in the state stack list is the world state, then the world will be rendered)
- The veil transparency is also controlled here, giving it that fade in fade out effect, based on the status of self.fading.

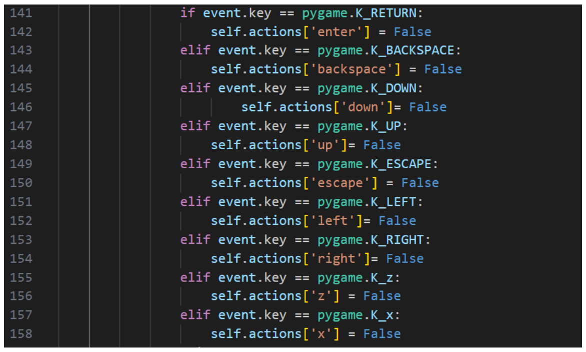
- Flip just puts the new surface onto the display, functions similarly to blit but blit just copies from the current or old surface, while flip creates a new surface.
- 

```
    def check_events(self):
        for event in pygame.event.get():
            if event.type == pygame.QUIT: #to check for quitting the game
                self.running, self.playing = False, False
                pygame.quit()
            if event.type == pygame.KEYDOWN: #to check for key presses⋯
            if event.type == pygame.MOUSEBUTTONDOWN: #to check for mouse clicks⋯

            if event.type == pygame.KEYUP: #to check for key releases⋯
            if event.type == pygame.MOUSEBUTTONUP: #to check for mouse releases⋯
```

- This controls key inputs and what they affect.

- Here are some examples of the key down event:
- 

```
if event.key == pygame.K_RETURN:
    self.actions['enter'] = True
elif event.key == pygame.K_BACKSPACE:
    self.actions['backspace'] = True
elif event.key == pygame.K_DOWN:
    self.actions['down']= True
elif event.key == pygame.K_UP:
    self.actions['up']= True
elif event.key == pygame.K_ESCAPE:
    self.actions['escape'] = True
elif event.key == pygame.K_LEFT:
    self.actions['left']= True
elif event.key == pygame.K_RIGHT:
    self.actions['right']= True
```

- Here are some examples of the key up event:
- 

```
                if event.key == pygame.K_RETURN:
                    self.actions['enter'] = False
                elif event.key == pygame.K_BACKSPACE:
                    self.actions['backspace'] = False
                elif event.key == pygame.K_DOWN:
                        self.actions['down']= False
                elif event.key == pygame.K_UP:
                    self.actions['up']= False
                elif event.key == pygame.K_ESCAPE:
                    self.actions['escape'] = False
                elif event.key == pygame.K_LEFT:
                    self.actions['left']= False
                elif event.key == pygame.K_RIGHT:
                    self.actions['right']= False
                elif event.key == pygame.K_z:
                    self.actions['z'] = False
                elif event.key == pygame.K_x:
                    self.actions['x'] = False
```

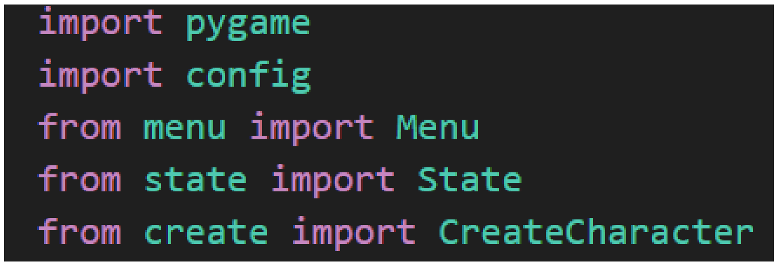
- By setting the self.actions values to true, they're basically screaming, "I'm being pressed down!", and it will register. Same goes for setting the self.actions values to false, but instead they scream, "I'm no longer being pressed!"

Game States

1. **Title Screen (title.py)**

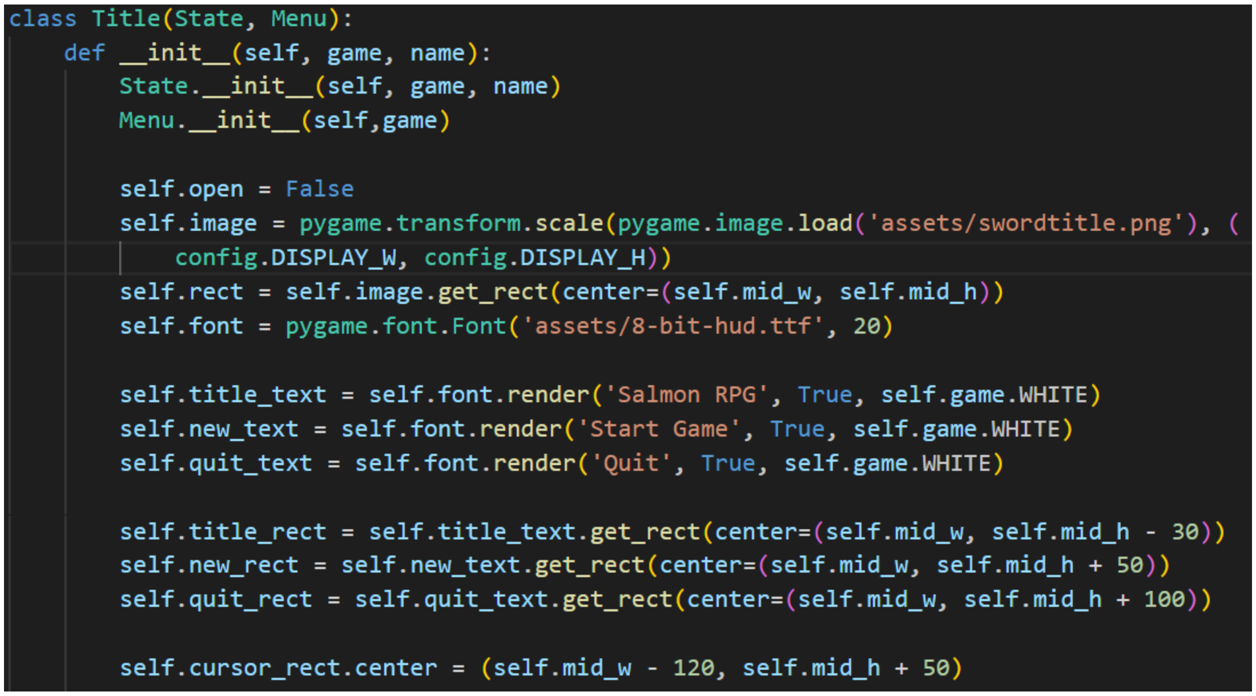
- Displays the title screen and main menu.

- Allows the player to start a new game or quit.



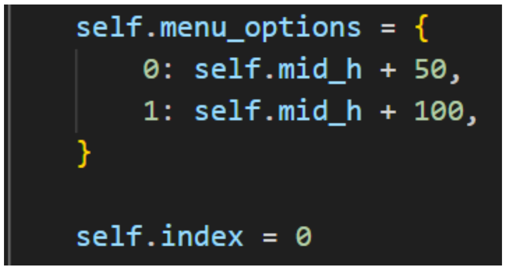
```
import pygame
import config
from menu import Menu
from state import State
from create import CreateCharacter
```

-
- Initial imports to setup the menu and to transition to the character creation screen



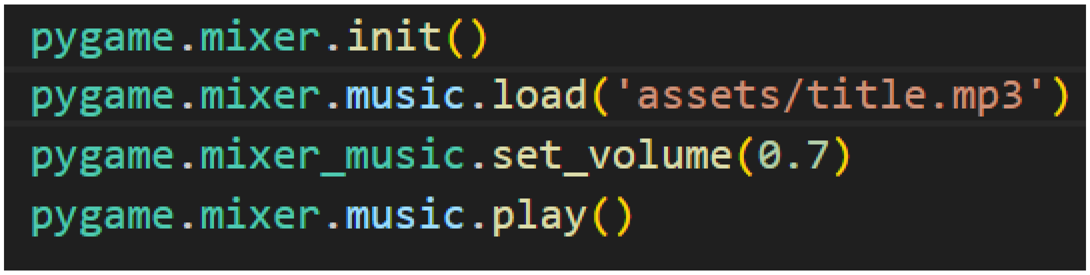
```
class Title(State, Menu):
    def __init__(self, game, name):
        State.__init__(self, game, name)
        Menu.__init__(self,game)

        self.open = False
        self.image = pygame.transform.scale(pygame.image.load('assets/swordtitle.png'), (
            config.DISPLAY_W, config.DISPLAY_H))
        self.rect = self.image.get_rect(center=(self.mid_w, self.mid_h))
        self.font = pygame.font.Font('assets/8-bit-hud.ttf', 20)

        self.title_text = self.font.render('Salmon RPG', True, self.game.WHITE)
        self.new_text = self.font.render('Start Game', True, self.game.WHITE)
        self.quit_text = self.font.render('Quit', True, self.game.WHITE)

        self.title_rect = self.title_text.get_rect(center=(self.mid_w, self.mid_h - 30))
        self.new_rect = self.new_text.get_rect(center=(self.mid_w, self.mid_h + 50))
        self.quit_rect = self.quit_text.get_rect(center=(self.mid_w, self.mid_h + 100))

        self.cursor_rect.center = (self.mid_w - 120, self.mid_h + 50)
```

-
- This controls the text and images that are shown on the screen in the title screen.



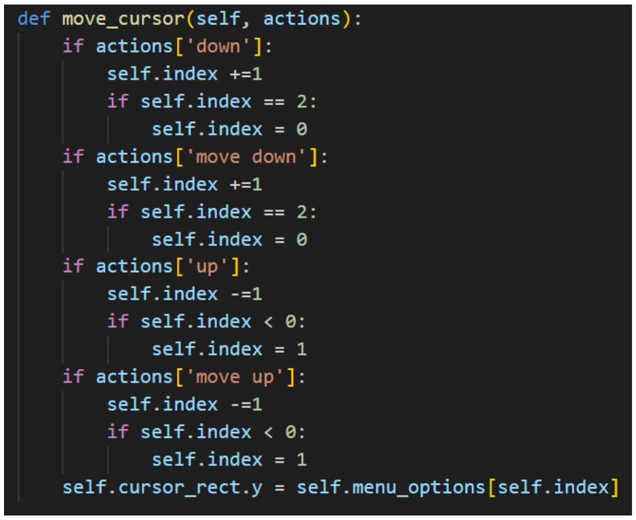
```
        self.menu_options = {
            0: self.mid_h + 50,
            1: self.mid_h + 100,
        }

        self.index = 0
```

-
- This controls the initial position of the cursor (the “>” symbol), when the self.index is 0, it puts the cursor beside the “Start Game” text.



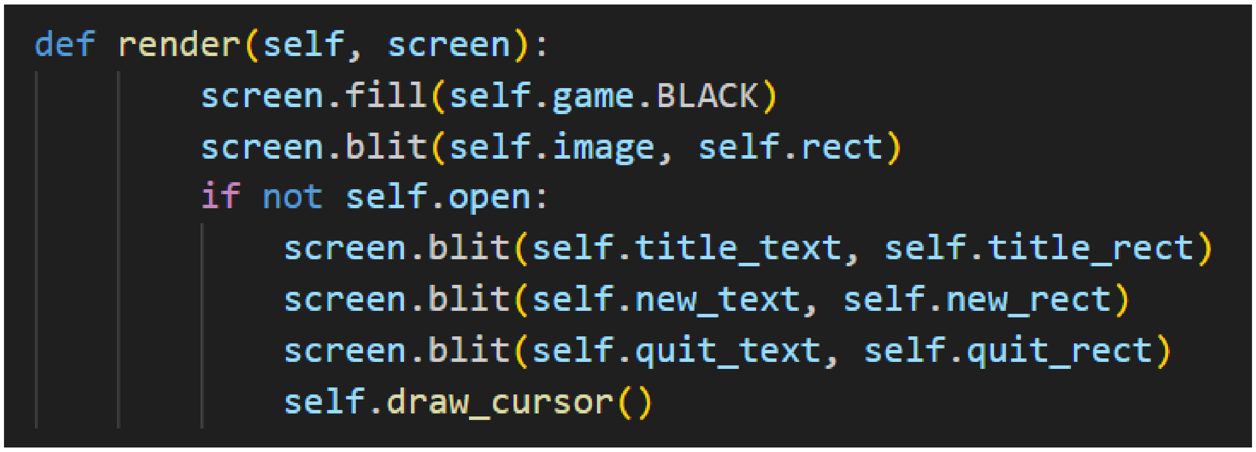
```python
pygame.mixer.init()
pygame.mixer.music.load('assets/title.mp3')
pygame.mixer_music.set_volume(0.7)
pygame.mixer.music.play()
```

-
- This initializes the pygame mixer and plays the music that is located in the assets file.



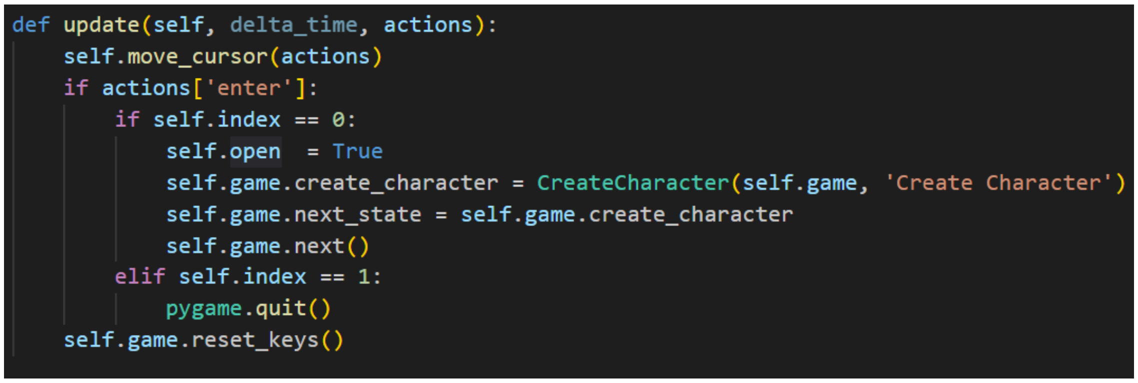
```python
def move_cursor(self, actions):
    if actions['down']:
        self.index +=1
        if self.index == 2:
            self.index = 0
    if actions['move down']:
        self.index +=1
        if self.index == 2:
            self.index = 0
    if actions['up']:
        self.index -=1
        if self.index < 0:
            self.index = 1
    if actions['move up']:
        self.index -=1
        if self.index < 0:
            self.index = 1
    self.cursor_rect.y = self.menu_options[self.index]
```

-
- Based on the actions created in game.py, and the index created above, this function controls where the cursor will be



```python
def render(self, screen):
        screen.fill(self.game.BLACK)
        screen.blit(self.image, self.rect)
        if not self.open:
            screen.blit(self.title_text, self.title_rect)
            screen.blit(self.new_text, self.new_rect)
            screen.blit(self.quit_text, self.quit_rect)
            self.draw_cursor()
```

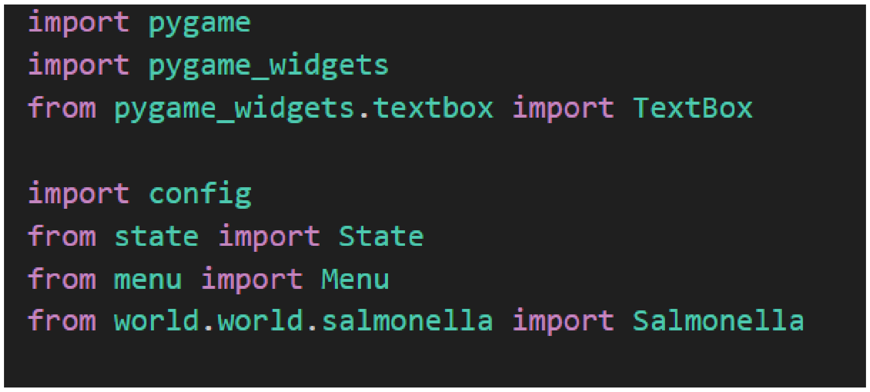
-
- This loads everything onto the screen, self.open is to control whether the title screen will transition to the character creation screen or not.

- 

```
def update(self, delta_time, actions):
    self.move_cursor(actions)
    if actions['enter']:
        if self.index == 0:
            self.open  = True
            self.game.create_character = CreateCharacter(self.game, 'Create Character')
            self.game.next_state = self.game.create_character
            self.game.next()
        elif self.index == 1:
            pygame.quit()
    self.game.reset_keys()
```

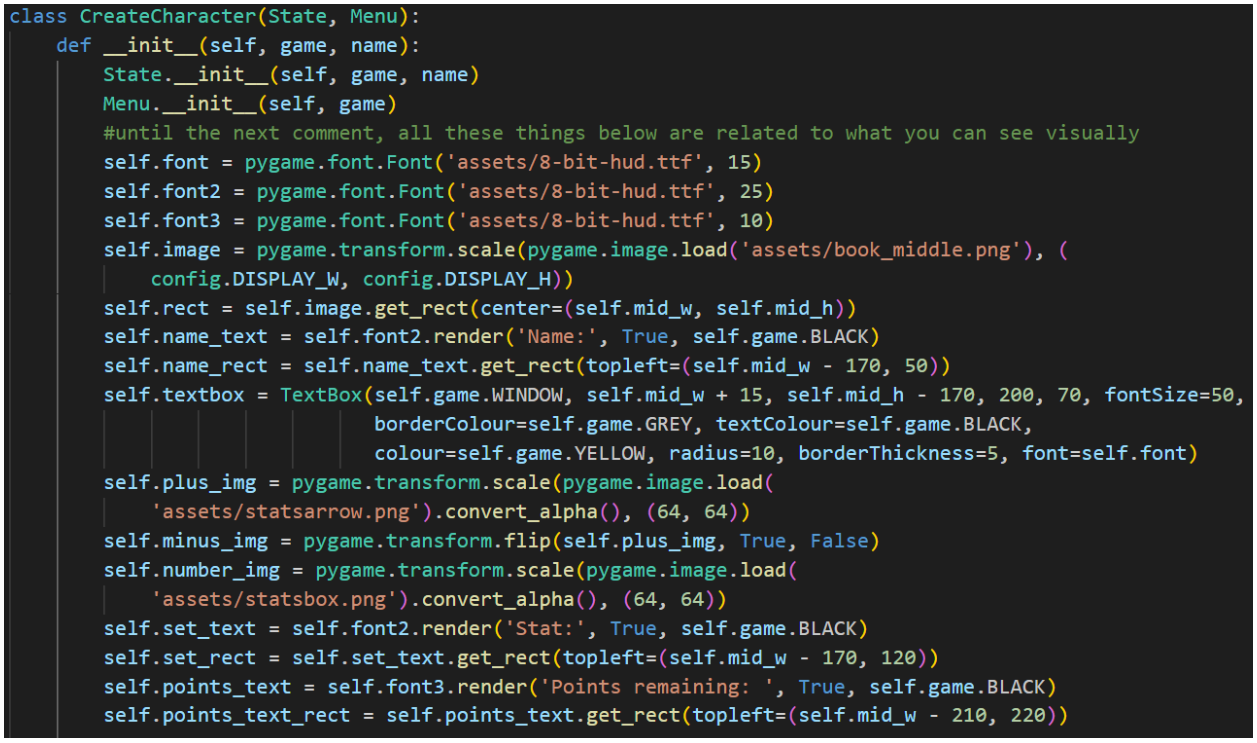
- The update function here makes sure that the cursor will move according to the index. When the enter key is pressed and the index is 0 (cursor is positioned on "Start Game"), the text on the title screen will disappear, then it will transition to the character creation screen. When the index is 1 however, then the game will simply quit. The reset keys function here is called so that the keys don't get repeatedly pressed.

2. Character Creation (create.py)

- Allows the player to create a character by entering a name and distributing stat points.
- 

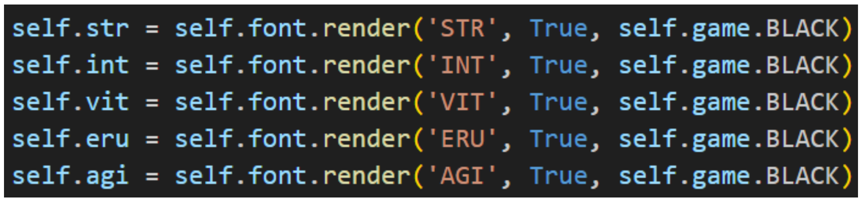
```
import pygame
import pygame_widgets
from pygame_widgets.textbox import TextBox

import config
from state import State
from menu import Menu
from world.world.salmonella import Salmonella
```

- Pygame widgets textbox is imported so that a textbox can be displayed on the screen.
- Initial imports for the menu and salmonella for transitioning to the game world.



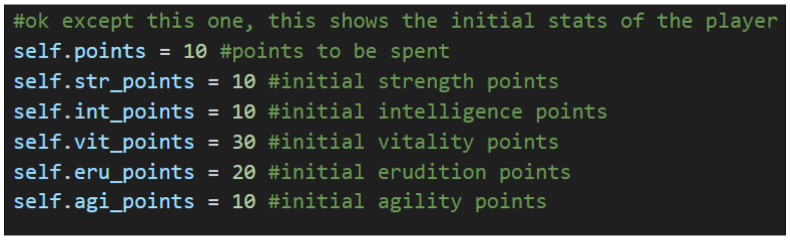
```python
class CreateCharacter(State, Menu):
    def __init__(self, game, name):
        State.__init__(self, game, name)
        Menu.__init__(self, game)
        #until the next comment, all these things below are related to what you can see visually
        self.font = pygame.font.Font('assets/8-bit-hud.ttf', 15)
        self.font2 = pygame.font.Font('assets/8-bit-hud.ttf', 25)
        self.font3 = pygame.font.Font('assets/8-bit-hud.ttf', 10)
        self.image = pygame.transform.scale(pygame.image.load('assets/book_middle.png'), (
            config.DISPLAY_W, config.DISPLAY_H))
        self.rect = self.image.get_rect(center=(self.mid_w, self.mid_h))
        self.name_text = self.font2.render('Name:', True, self.game.BLACK)
        self.name_rect = self.name_text.get_rect(topleft=(self.mid_w - 170, 50))
        self.textbox = TextBox(self.game.WINDOW, self.mid_w + 15, self.mid_h - 170, 200, 70, fontSize=50,
                               borderColour=self.game.GREY, textColour=self.game.BLACK,
                               colour=self.game.YELLOW, radius=10, borderThickness=5, font=self.font)
        self.plus_img = pygame.transform.scale(pygame.image.load(
            'assets/statsarrow.png').convert_alpha(), (64, 64))
        self.minus_img = pygame.transform.flip(self.plus_img, True, False)
        self.number_img = pygame.transform.scale(pygame.image.load(
            'assets/statsbox.png').convert_alpha(), (64, 64))
        self.set_text = self.font2.render('Stat:', True, self.game.BLACK)
        self.set_rect = self.set_text.get_rect(topleft=(self.mid_w - 170, 120))
        self.points_text = self.font3.render('Points remaining: ', True, self.game.BLACK)
        self.points_text_rect = self.points_text.get_rect(topleft=(self.mid_w - 210, 220))
```

-



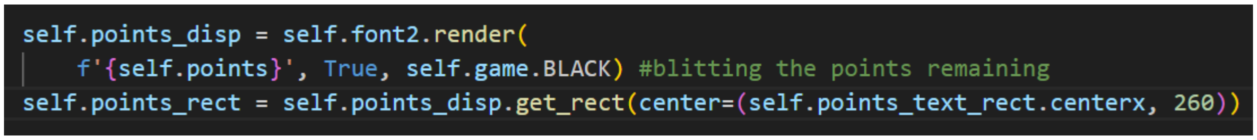
```python
self.str = self.font.render('STR', True, self.game.BLACK)
self.int = self.font.render('INT', True, self.game.BLACK)
self.vit = self.font.render('VIT', True, self.game.BLACK)
self.eru = self.font.render('ERU', True, self.game.BLACK)
self.agi = self.font.render('AGI', True, self.game.BLACK)
```

-
- Visual stuff, this controls all the text within the character creation screen, and controls the image for the background.



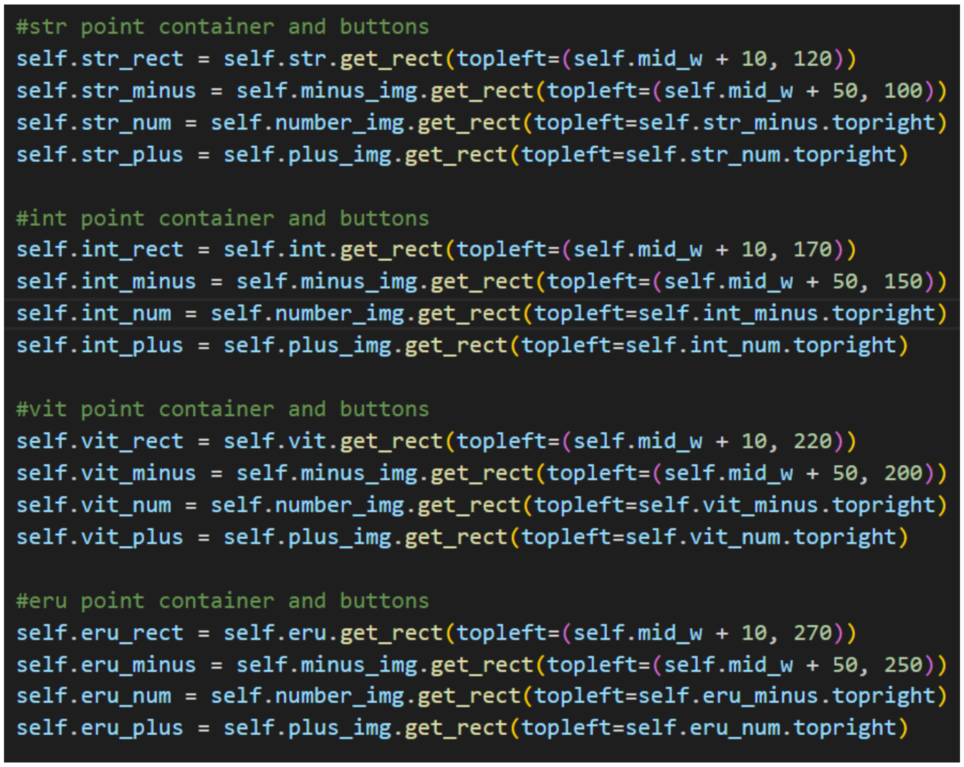
```python
#ok except this one, this shows the initial stats of the player
self.points = 10 #points to be spent
self.str_points = 10 #initial strength points
self.int_points = 10 #initial intelligence points
self.vit_points = 30 #initial vitality points
self.eru_points = 20 #initial erudition points
self.agi_points = 10 #initial agility points
```

-
- This controls the initial points that the character has on each stat.



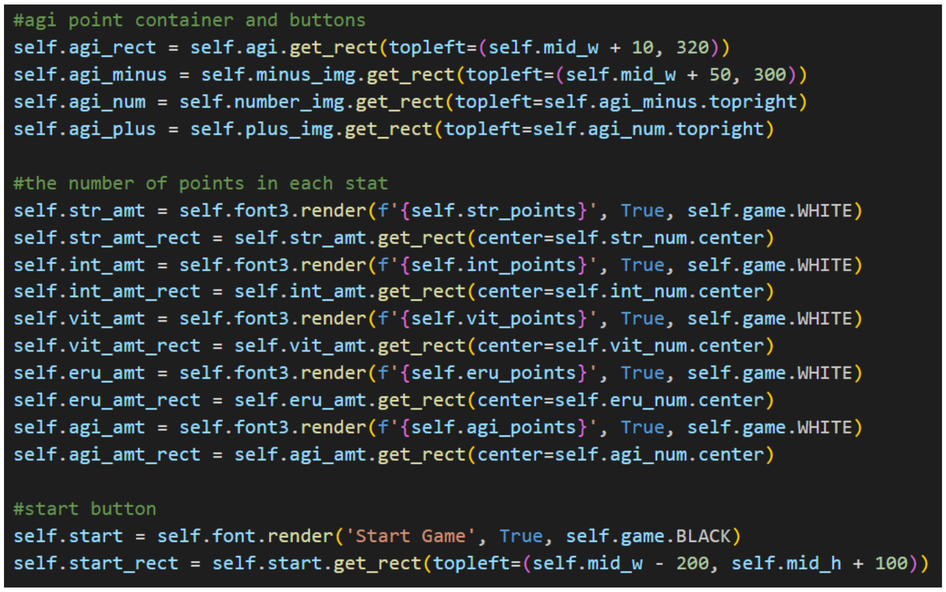
```python
self.points_disp = self.font2.render(
    f'{self.points}', True, self.game.BLACK) #blitting the points remaining
self.points_rect = self.points_disp.get_rect(center=(self.points_text_rect.centerx, 260))
```

-
- This displays the remaining points to be spent on a stat.



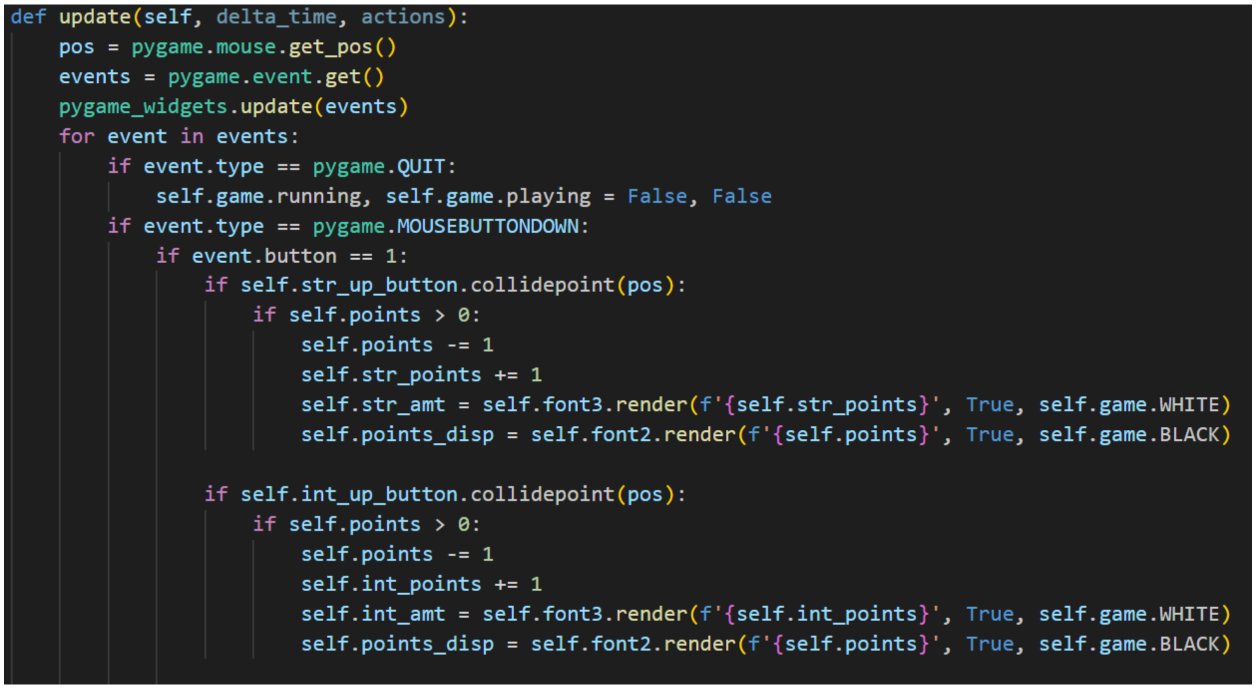
```python
#str point container and buttons
self.str_rect = self.str.get_rect(topleft=(self.mid_w + 10, 120))
self.str_minus = self.minus_img.get_rect(topleft=(self.mid_w + 50, 100))
self.str_num = self.number_img.get_rect(topleft=self.str_minus.topright)
self.str_plus = self.plus_img.get_rect(topleft=self.str_num.topright)

#int point container and buttons
self.int_rect = self.int.get_rect(topleft=(self.mid_w + 10, 170))
self.int_minus = self.minus_img.get_rect(topleft=(self.mid_w + 50, 150))
self.int_num = self.number_img.get_rect(topleft=self.int_minus.topright)
self.int_plus = self.plus_img.get_rect(topleft=self.int_num.topright)

#vit point container and buttons
self.vit_rect = self.vit.get_rect(topleft=(self.mid_w + 10, 220))
self.vit_minus = self.minus_img.get_rect(topleft=(self.mid_w + 50, 200))
self.vit_num = self.number_img.get_rect(topleft=self.vit_minus.topright)
self.vit_plus = self.plus_img.get_rect(topleft=self.vit_num.topright)

#eru point container and buttons
self.eru_rect = self.eru.get_rect(topleft=(self.mid_w + 10, 270))
self.eru_minus = self.minus_img.get_rect(topleft=(self.mid_w + 50, 250))
self.eru_num = self.number_img.get_rect(topleft=self.eru_minus.topright)
self.eru_plus = self.plus_img.get_rect(topleft=self.eru_num.topright)
```

-



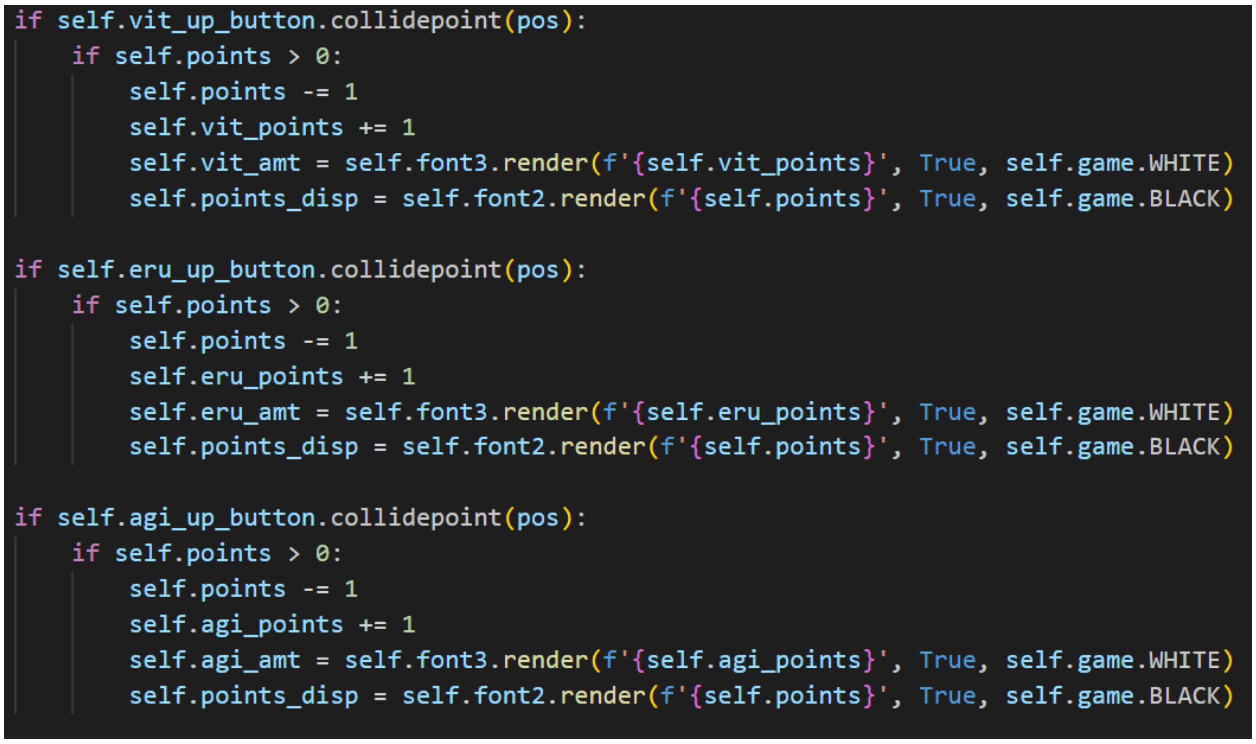
```python
#agi point container and buttons
self.agi_rect = self.agi.get_rect(topleft=(self.mid_w + 10, 320))
self.agi_minus = self.minus_img.get_rect(topleft=(self.mid_w + 50, 300))
self.agi_num = self.number_img.get_rect(topleft=self.agi_minus.topright)
self.agi_plus = self.plus_img.get_rect(topleft=self.agi_num.topright)

#the number of points in each stat
self.str_amt = self.font3.render(f'{self.str_points}', True, self.game.WHITE)
self.str_amt_rect = self.str_amt.get_rect(center=self.str_num.center)
self.int_amt = self.font3.render(f'{self.int_points}', True, self.game.WHITE)
self.int_amt_rect = self.int_amt.get_rect(center=self.int_num.center)
self.vit_amt = self.font3.render(f'{self.vit_points}', True, self.game.WHITE)
self.vit_amt_rect = self.vit_amt.get_rect(center=self.vit_num.center)
self.eru_amt = self.font3.render(f'{self.eru_points}', True, self.game.WHITE)
self.eru_amt_rect = self.eru_amt.get_rect(center=self.eru_num.center)
self.agi_amt = self.font3.render(f'{self.agi_points}', True, self.game.WHITE)
self.agi_amt_rect = self.agi_amt.get_rect(center=self.agi_num.center)

#start button
self.start = self.font.render('Start Game', True, self.game.BLACK)
self.start_rect = self.start.get_rect(topleft=(self.mid_w - 200, self.mid_h + 100))
```

-
- This whole thing above controls and displays the text and buttons for adding and reducing stat points, and displays the start button.
- There's also a render function that is for rendering and displaying all of the things above, (the background image, the text for stats, the buttons) but I'm not gonna put the image here because it's too long.



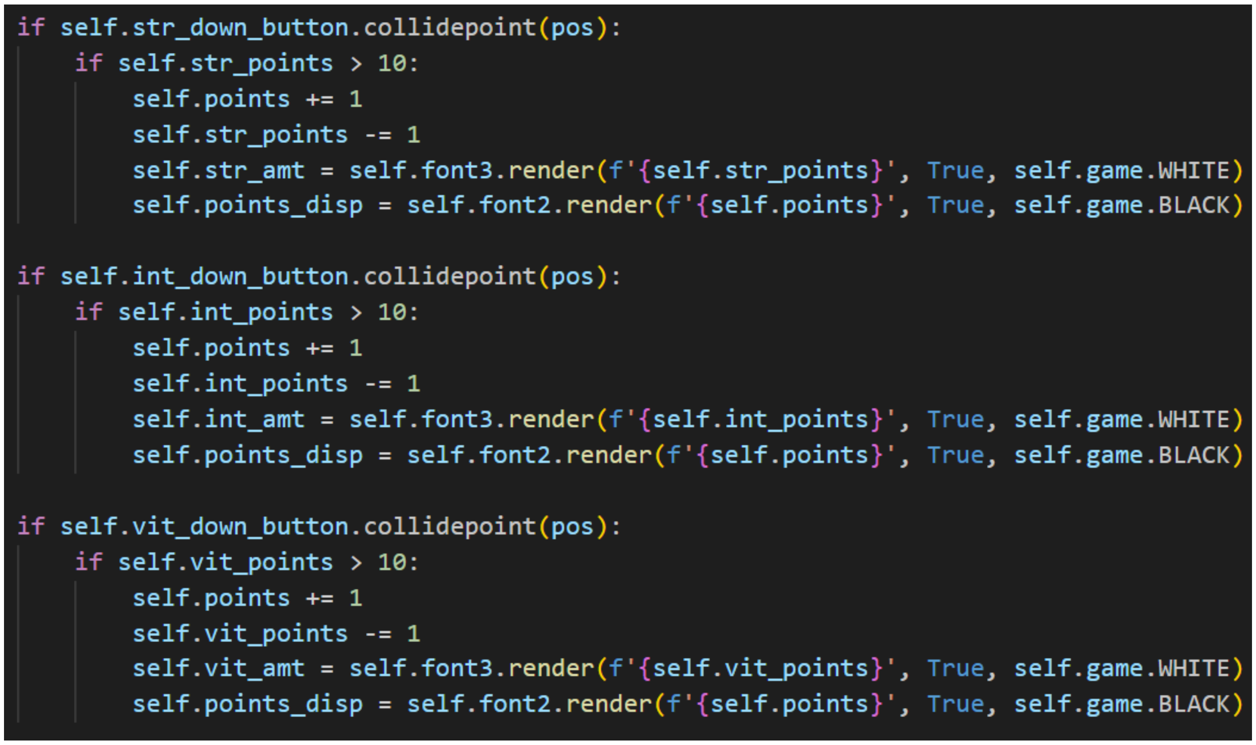
```python
def update(self, delta_time, actions):
    pos = pygame.mouse.get_pos()
    events = pygame.event.get()
    pygame_widgets.update(events)
    for event in events:
        if event.type == pygame.QUIT:
            self.game.running, self.game.playing = False, False
        if event.type == pygame.MOUSEBUTTONDOWN:
            if event.button == 1:
                if self.str_up_button.collidepoint(pos):
                    if self.points > 0:
                        self.points -= 1
                        self.str_points += 1
                        self.str_amt = self.font3.render(f'{self.str_points}', True, self.game.WHITE)
                        self.points_disp = self.font2.render(f'{self.points}', True, self.game.BLACK)

                if self.int_up_button.collidepoint(pos):
                    if self.points > 0:
                        self.points -= 1
                        self.int_points += 1
                        self.int_amt = self.font3.render(f'{self.int_points}', True, self.game.WHITE)
                        self.points_disp = self.font2.render(f'{self.points}', True, self.game.BLACK)
```

-



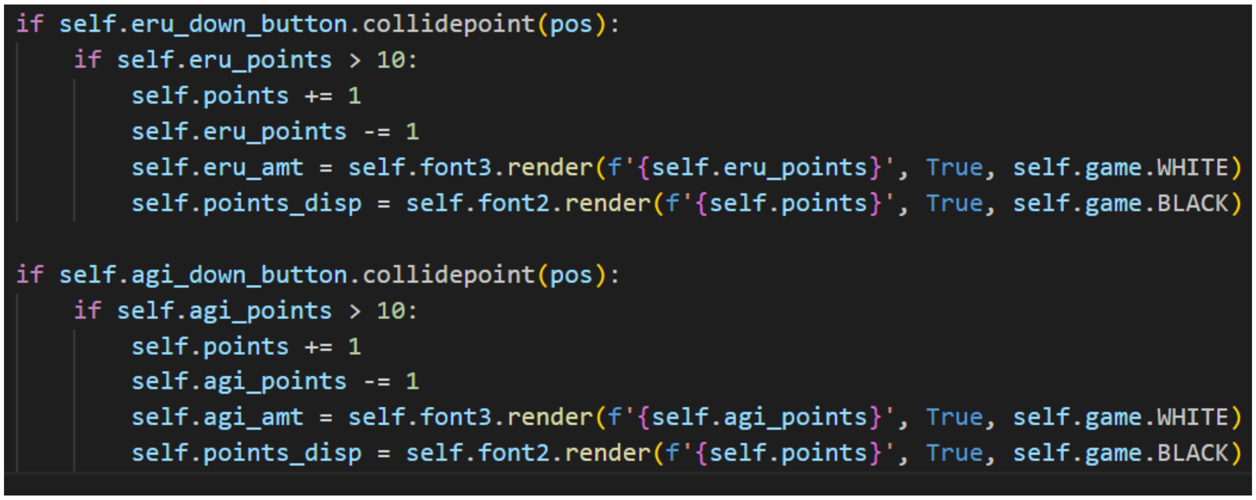
```python
if self.vit_up_button.collidepoint(pos):
    if self.points > 0:
        self.points -= 1
        self.vit_points += 1
        self.vit_amt = self.font3.render(f'{self.vit_points}', True, self.game.WHITE)
        self.points_disp = self.font2.render(f'{self.points}', True, self.game.BLACK)

if self.eru_up_button.collidepoint(pos):
    if self.points > 0:
        self.points -= 1
        self.eru_points += 1
        self.eru_amt = self.font3.render(f'{self.eru_points}', True, self.game.WHITE)
        self.points_disp = self.font2.render(f'{self.points}', True, self.game.BLACK)

if self.agi_up_button.collidepoint(pos):
    if self.points > 0:
        self.points -= 1
        self.agi_points += 1
        self.agi_amt = self.font3.render(f'{self.agi_points}', True, self.game.WHITE)
        self.points_disp = self.font2.render(f'{self.points}', True, self.game.BLACK)
```

-



```
if self.str_down_button.collidepoint(pos):
    if self.str_points > 10:
        self.points += 1
        self.str_points -= 1
        self.str_amt = self.font3.render(f'{self.str_points}', True, self.game.WHITE)
        self.points_disp = self.font2.render(f'{self.points}', True, self.game.BLACK)

if self.int_down_button.collidepoint(pos):
    if self.int_points > 10:
        self.points += 1
        self.int_points -= 1
        self.int_amt = self.font3.render(f'{self.int_points}', True, self.game.WHITE)
        self.points_disp = self.font2.render(f'{self.points}', True, self.game.BLACK)

if self.vit_down_button.collidepoint(pos):
    if self.vit_points > 10:
        self.points += 1
        self.vit_points -= 1
        self.vit_amt = self.font3.render(f'{self.vit_points}', True, self.game.WHITE)
        self.points_disp = self.font2.render(f'{self.points}', True, self.game.BLACK)
```

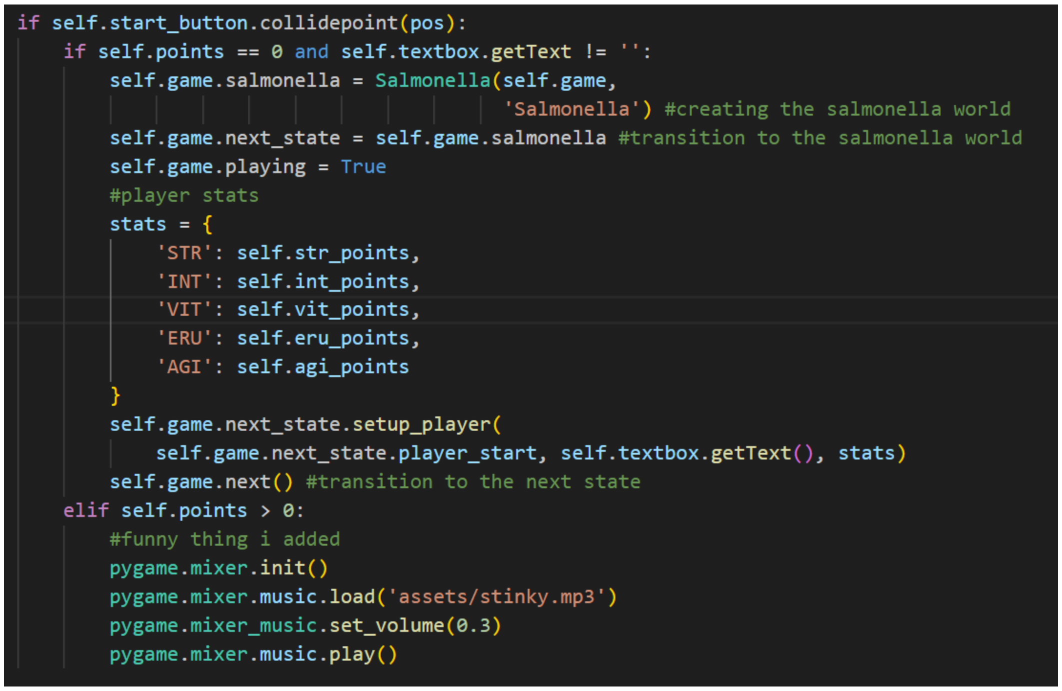
-



```
if self.eru_down_button.collidepoint(pos):
    if self.eru_points > 10:
        self.points += 1
        self.eru_points -= 1
        self.eru_amt = self.font3.render(f'{self.eru_points}', True, self.game.WHITE)
        self.points_disp = self.font2.render(f'{self.points}', True, self.game.BLACK)

if self.agi_down_button.collidepoint(pos):
    if self.agi_points > 10:
        self.points += 1
        self.agi_points -= 1
        self.agi_amt = self.font3.render(f'{self.agi_points}', True, self.game.WHITE)
        self.points_disp = self.font2.render(f'{self.points}', True, self.game.BLACK)
```

-

- This controls the point up and point down buttons, to change the amount of points that are added or subtracted from the stats.

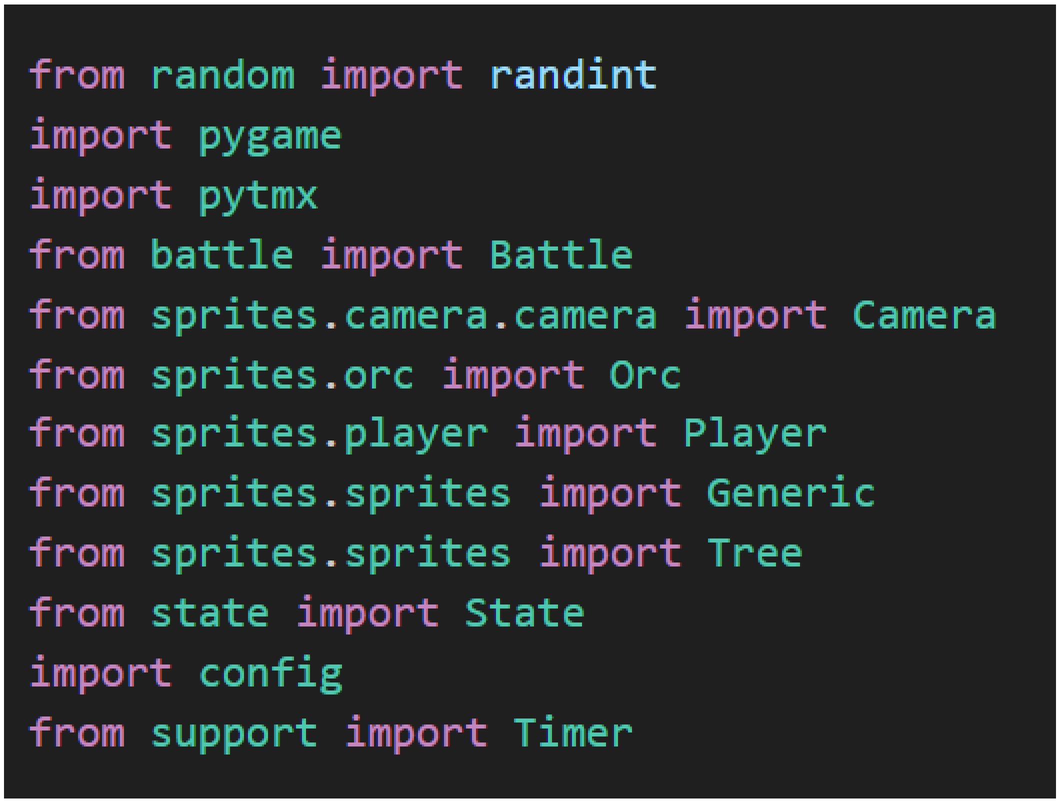


```
if self.start_button.collidepoint(pos):
    if self.points == 0 and self.textbox.getText != '':
        self.game.salmonella = Salmonella(self.game,
                                          'Salmonella') #creating the salmonella world
        self.game.next_state = self.game.salmonella #transition to the salmonella world
        self.game.playing = True
        #player stats
        stats = {
            'STR': self.str_points,
            'INT': self.int_points,
            'VIT': self.vit_points,
            'ERU': self.eru_points,
            'AGI': self.agi_points
        }
        self.game.next_state.setup_player(
            self.game.next_state.player_start, self.textbox.getText(), stats)
        self.game.next() #transition to the next state
    elif self.points > 0:
        #funny thing i added
        pygame.mixer.init()
        pygame.mixer.music.load('assets/stinky.mp3')
        pygame.mixer_music.set_volume(0.3)
        pygame.mixer.music.play()
```

-
- This is for when the start button is pressed, it first makes sure that there are no points left to spend, and that there is a name written in the textbox. The stats are initialized and then put into the setup_player function that is within the salmonella.py file. Then it transitions to the next state using self.game.next()

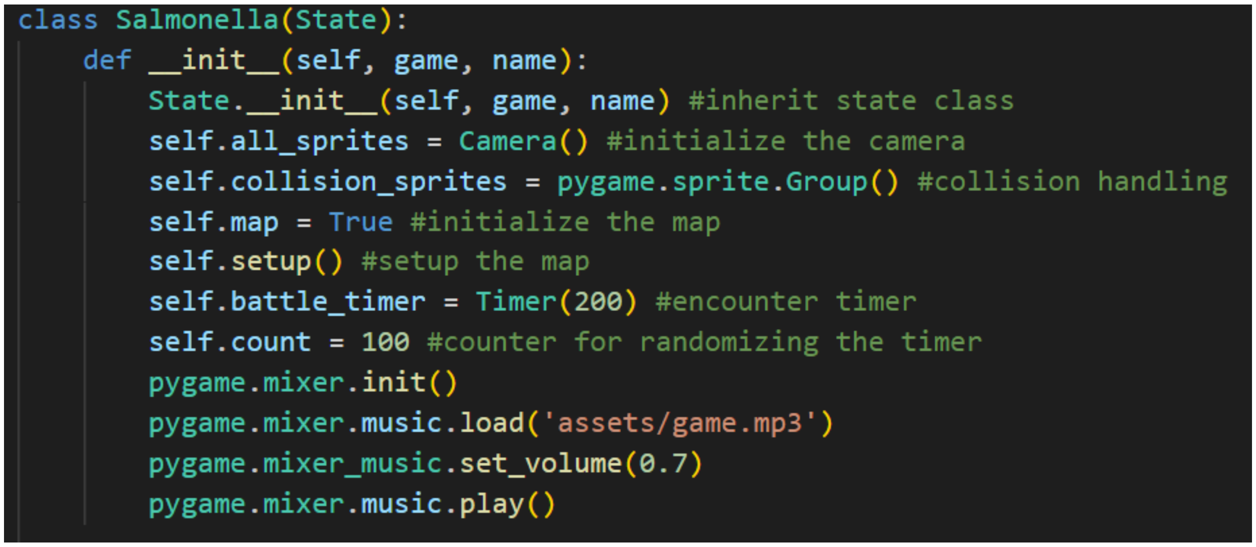
3. Game World (salmonella.py)

- Manages the main game world where the player can move around the map.
- Handles player movement, collision detection, and random enemy encounters.



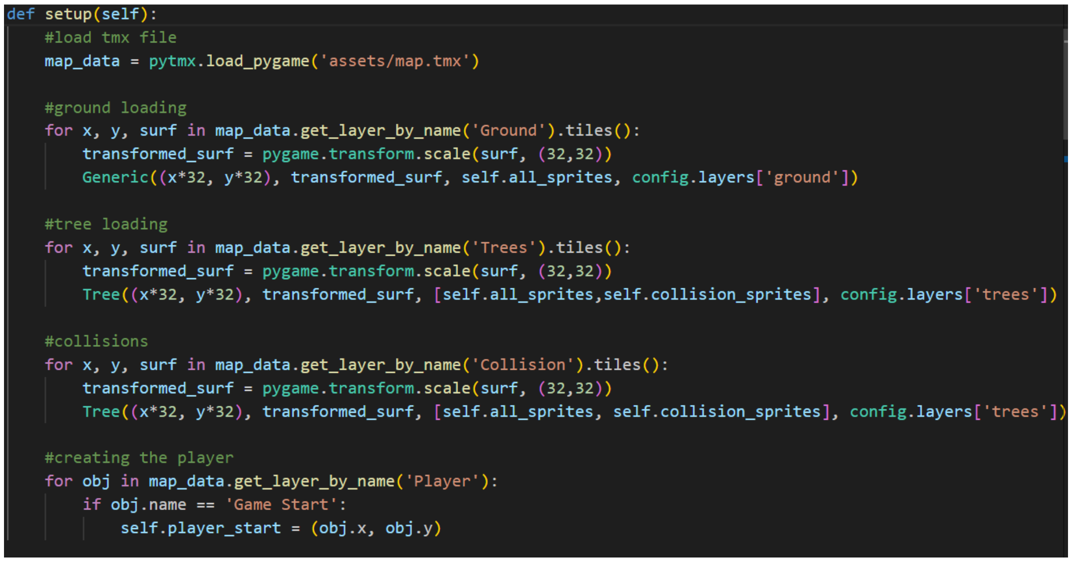
```python
from random import randint
import pygame
import pytmx
from battle import Battle
from sprites.camera.camera import Camera
from sprites.orc import Orc
from sprites.player import Player
from sprites.sprites import Generic
from sprites.sprites import Tree
from state import State
import config
from support import Timer
```

-
- Initial imports, randint for determining enemy level, pytmx for loading the map, battle to transition to the battle screen, camera for the camera movement.
- Orc class to determine the size and level of the orc in the determine_enemy function that I will explain below.
- Player class to control the player position and load player stats.
- Generic and Tree class for hitboxes.
- Config is used here for enemy levels and sizes, as well as the world layers.
- Timer is used for the enemy timer, how long it takes before the player encounters the enemy.

- 

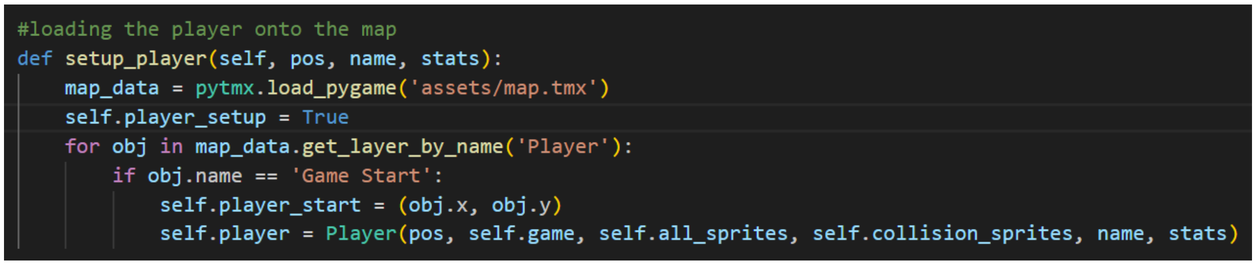
```python
class Salmonella(State):
    def __init__(self, game, name):
        State.__init__(self, game, name) #inherit state class
        self.all_sprites = Camera() #initialize the camera
        self.collision_sprites = pygame.sprite.Group() #collision handling
        self.map = True #initialize the map
        self.setup() #setup the map
        self.battle_timer = Timer(200) #encounter timer
        self.count = 100 #counter for randomizing the timer
        pygame.mixer.init()
        pygame.mixer.music.load('assets/game.mp3')
        pygame.mixer_music.set_volume(0.7)
        pygame.mixer.music.play()
```

- **self.all_sprites** initializes the camera, how the camera works will be explained later.
- **self.collision_sprites** creates a group for the sprites that will be put into self.collision_sprites, so that the sprites can easily be controlled.
- **self.map** initializes the map, and **self.setup** loads all the layers of the map.
- **Battle timer and count** will be used later to randomize the time needed to encounter an enemy.
- Music is also loaded here.

- 

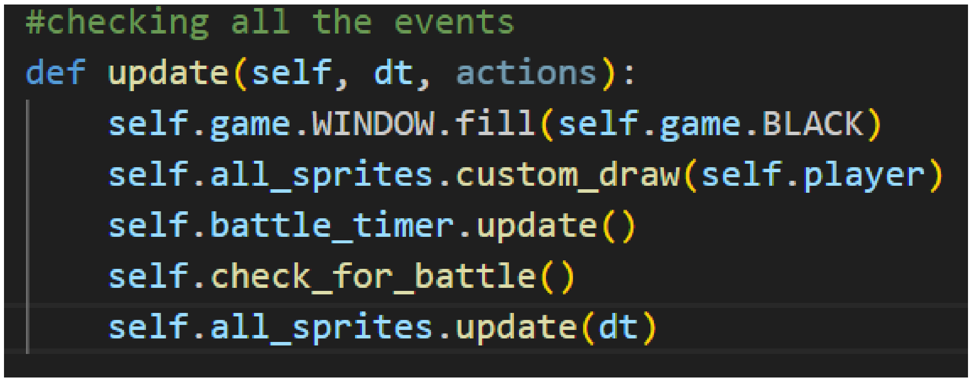
```python
def setup(self):
    #load tmx file
    map_data = pytmx.load_pygame('assets/map.tmx')

    #ground loading
    for x, y, surf in map_data.get_layer_by_name('Ground').tiles():
        transformed_surf = pygame.transform.scale(surf, (32,32))
        Generic((x*32, y*32), transformed_surf, self.all_sprites, config.layers['ground'])

    #tree loading
    for x, y, surf in map_data.get_layer_by_name('Trees').tiles():
        transformed_surf = pygame.transform.scale(surf, (32,32))
        Tree((x*32, y*32), transformed_surf, [self.all_sprites,self.collision_sprites], config.layers['trees'])

    #collisions
    for x, y, surf in map_data.get_layer_by_name('Collision').tiles():
        transformed_surf = pygame.transform.scale(surf, (32,32))
        Tree((x*32, y*32), transformed_surf, [self.all_sprites, self.collision_sprites], config.layers['trees'])

    #creating the player
    for obj in map_data.get_layer_by_name('Player'):
        if obj.name == 'Game Start':
            self.player_start = (obj.x, obj.y)
```

- The setup function first loads the map, then it loads each tile layer, starting from the ground, the trees and fences, then to the collision logic, before it finally loads the player onto the map, based on the position of the player object within the tmx file.

- 

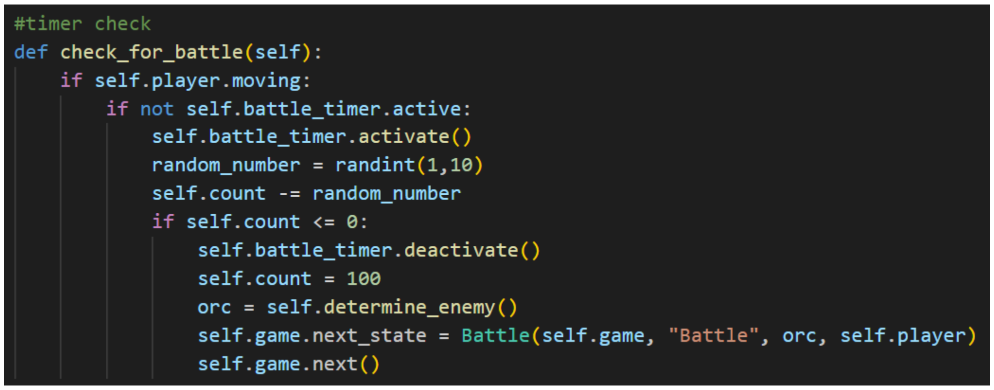
```python
#loading the player onto the map
def setup_player(self, pos, name, stats):
    map_data = pytmx.load_pygame('assets/map.tmx')
    self.player_setup = True
    for obj in map_data.get_layer_by_name('Player'):
        if obj.name == 'Game Start':
            self.player_start = (obj.x, obj.y)
            self.player = Player(pos, self.game, self.all_sprites, self.collision_sprites, name, stats)
```

- Afterwards, the player is then loaded again, this time not only the player sprite being loaded, but the stats with it.

- 

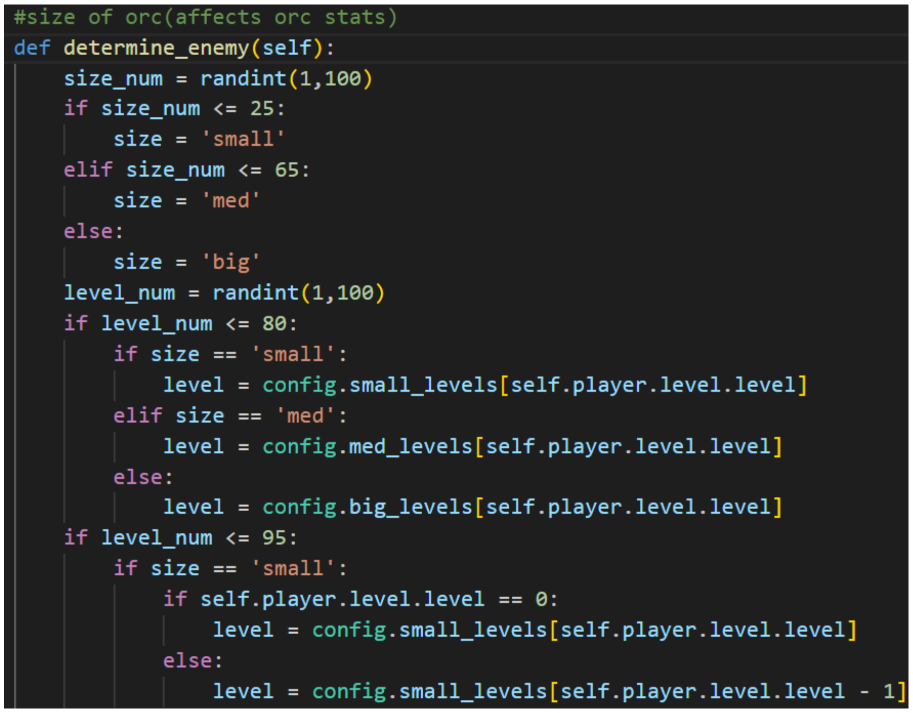
```python
#checking all the events
def update(self, dt, actions):
    self.game.WINDOW.fill(self.game.BLACK)
    self.all_sprites.custom_draw(self.player)
    self.battle_timer.update()
    self.check_for_battle()
    self.all_sprites.update(dt)
```

- This initializes the player hitbox, updates the battle timer so it can check whether it should run or not, then it actively checks if it should run the timer or not (will be explained later), then it updates the player animations based on the dt, so it will run at the same rate regardless of the device framerate.

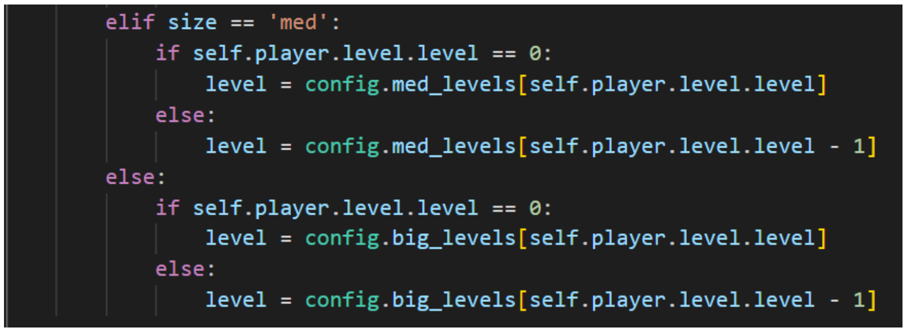
- 

```python
#timer check
def check_for_battle(self):
    if self.player.moving:
        if not self.battle_timer.active:
            self.battle_timer.activate()
            random_number = randint(1,10)
            self.count -= random_number
            if self.count <= 0:
                self.battle_timer.deactivate()
                self.count = 100
                orc = self.determine_enemy()
                self.game.next_state = Battle(self.game, "Battle", orc, self.player)
                self.game.next()
```

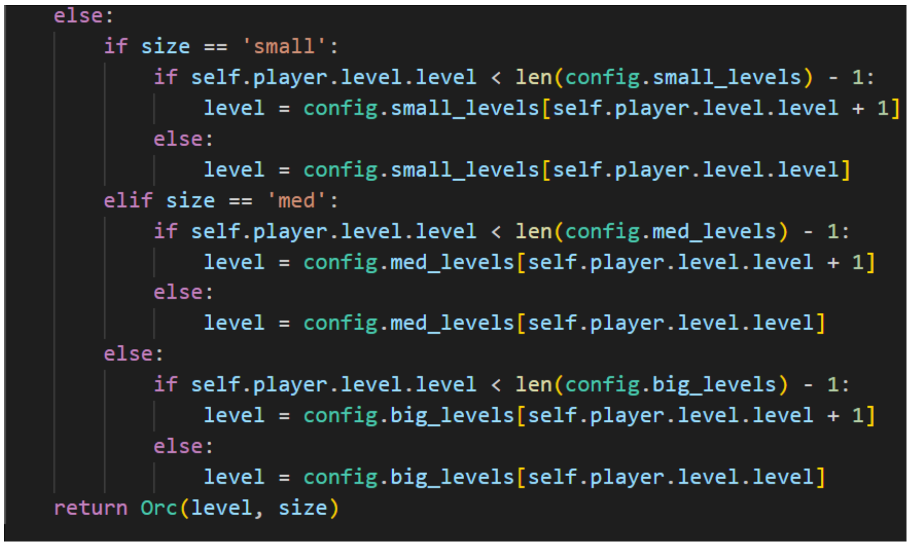
- This function checks whether the player is moving or not, then it activates the battle timer, which will tick down if the player is moving, and will deactivate the timer if the player is not moving. If the timer runs out, then the game quickly tries to determine the enemy size and level, before resetting the timer and moving on to the battle screen.

- 

```python
#size of orc(affects orc stats)
def determine_enemy(self):
    size_num = randint(1,100)
    if size_num <= 25:
        size = 'small'
    elif size_num <= 65:
        size = 'med'
    else:
        size = 'big'
    level_num = randint(1,100)
    if level_num <= 80:
        if size == 'small':
            level = config.small_levels[self.player.level.level]
        elif size == 'med':
            level = config.med_levels[self.player.level.level]
        else:
            level = config.big_levels[self.player.level.level]
    if level_num <= 95:
        if size == 'small':
            if self.player.level.level == 0:
                level = config.small_levels[self.player.level.level]
            else:
                level = config.small_levels[self.player.level.level - 1]
```

- 

```python
        elif size == 'med':
            if self.player.level.level == 0:
                level = config.med_levels[self.player.level.level]
            else:
                level = config.med_levels[self.player.level.level - 1]
        else:
            if self.player.level.level == 0:
                level = config.big_levels[self.player.level.level]
            else:
                level = config.big_levels[self.player.level.level - 1]
```

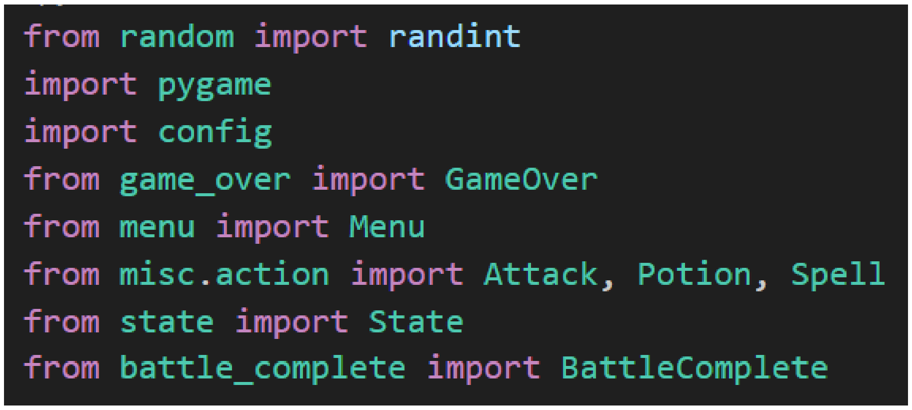
- 

```python
    else:
        if size == 'small':
            if self.player.level.level < len(config.small_levels) - 1:
                level = config.small_levels[self.player.level.level + 1]
            else:
                level = config.small_levels[self.player.level.level]
        elif size == 'med':
            if self.player.level.level < len(config.med_levels) - 1:
                level = config.med_levels[self.player.level.level + 1]
            else:
                level = config.med_levels[self.player.level.level]
        else:
            if self.player.level.level < len(config.big_levels) - 1:
                level = config.big_levels[self.player.level.level + 1]
            else:
                level = config.big_levels[self.player.level.level]
    return Orc(level, size)
```

- The determine enemy function controls the size and level of the enemy that the player will encounter in the battle screen

4. **Battle (battle.py)**

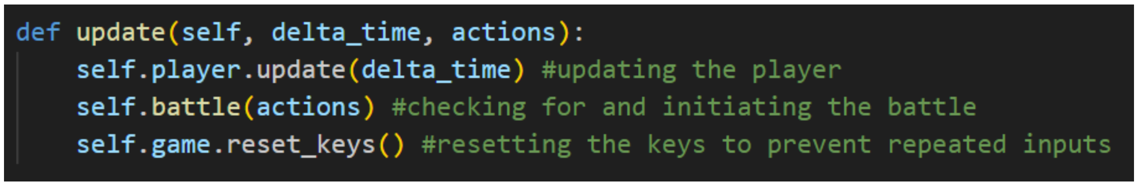
- Manages the battle state where the player engages in combat with enemies.
- Handles player and enemy actions, rendering battle UI, and updating health and mana.



```python
from random import randint
import pygame
import config
from game_over import GameOver
from menu import Menu
from misc.action import Attack, Potion, Spell
from state import State
from battle_complete import BattleComplete
```

-
- Initial import, GameOver class for when you lose, BattleComplete for when you win, Attack, Potion, and Spell to control the amount of damage dealt or hp healed.

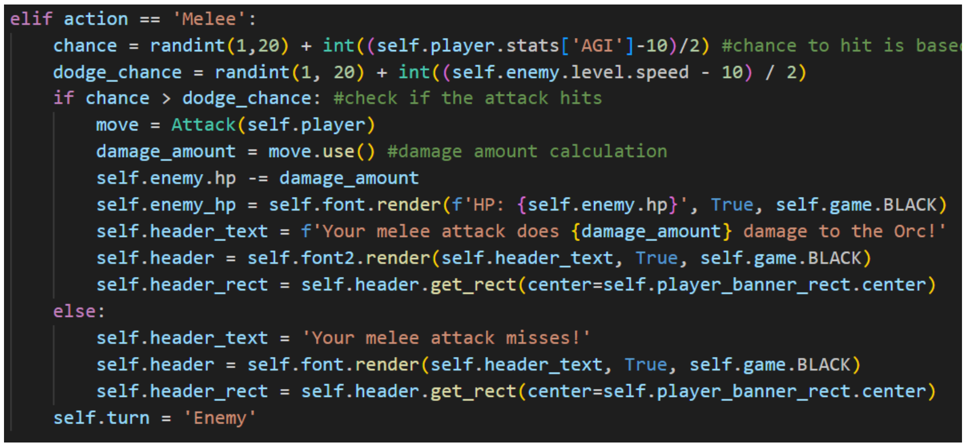
- The init function basically handles the visual aspect of the battle screen (text, buttons, cursor, loading the player and enemy sprite)



```python
def update(self, delta_time, actions):
    self.player.update(delta_time) #updating the player
    self.battle(actions) #checking for and initiating the battle
    self.game.reset_keys() #resetting the keys to prevent repeated inputs
```

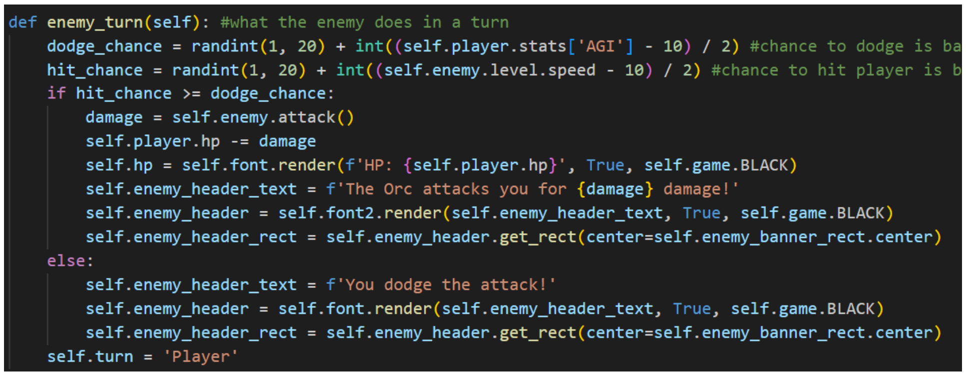
-
- This is the update function, self.battle(actions) checks for the turns, whether the enemy is dead or whether the player is dead, and checks if the player leveled up or not. The keys are then reset so multiple inputs don't happen.

- A render function is there to put everything in the init function onto the screen. It also draws the cursor onto the screen.
- The move cursor function is also back here in the battle screen so that the player can freely move the cursor to whatever button they want to use. Logic stays the same, using indexes for the position of the cursor.
- There is also a player turn function that I will explain briefly in the document, but will not screenshot and put in the document, as it will clutter the document because it's pretty long.
- For the player turn, it basically checks where the cursor is at, and when the enter key is pressed, the action is executed. It then checks whether the selected action works or not. For example, take this action below:

- 

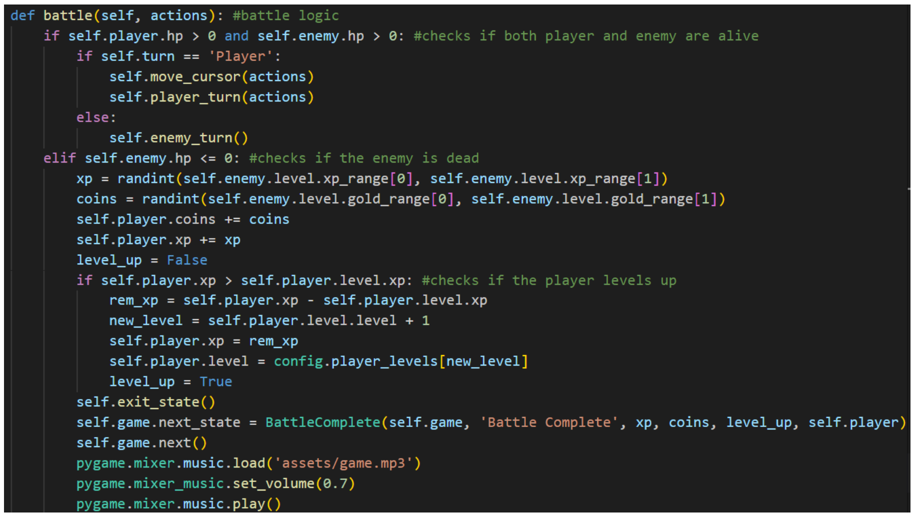
```
elif action == 'Melee':
    chance = randint(1,20) + int((self.player.stats['AGI']-10)/2) #chance to hit is base
    dodge_chance = randint(1, 20) + int((self.enemy.level.speed - 10) / 2)
    if chance > dodge_chance: #check if the attack hits
        move = Attack(self.player)
        damage_amount = move.use() #damage amount calculation
        self.enemy.hp -= damage_amount
        self.enemy_hp = self.font.render(f'HP: {self.enemy.hp}', True, self.game.BLACK)
        self.header_text = f'Your melee attack does {damage_amount} damage to the Orc!'
        self.header = self.font2.render(self.header_text, True, self.game.BLACK)
        self.header_rect = self.header.get_rect(center=self.player_banner_rect.center)
    else:
        self.header_text = 'Your melee attack misses!'
        self.header = self.font.render(self.header_text, True, self.game.BLACK)
        self.header_rect = self.header.get_rect(center=self.player_banner_rect.center)
    self.turn = 'Enemy'
```

- The chance to hit a melee attack is based on the agility of the player, and the chance for the enemy to dodge is based on the enemy's speed. The damage is then calculated (will be explained later), and it will display a text saying how much damage it dealt or whether or not it hit. The turn then ends and is given to the enemy.

- 

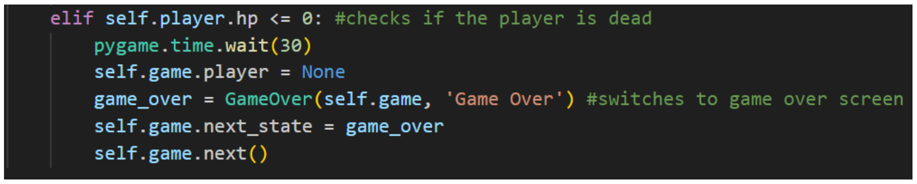
```
def enemy_turn(self): #what the enemy does in a turn
    dodge_chance = randint(1, 20) + int((self.player.stats['AGI'] - 10) / 2) #chance to dodge is ba
    hit_chance = randint(1, 20) + int((self.enemy.level.speed - 10) / 2) #chance to hit player is b
    if hit_chance >= dodge_chance:
        damage = self.enemy.attack()
        self.player.hp -= damage
        self.hp = self.font.render(f'HP: {self.player.hp}', True, self.game.BLACK)
        self.enemy_header_text = f'The Orc attacks you for {damage} damage!'
        self.enemy_header = self.font2.render(self.enemy_header_text, True, self.game.BLACK)
        self.enemy_header_rect = self.enemy_header.get_rect(center=self.enemy_banner_rect.center)
    else:
        self.enemy_header_text = f'You dodge the attack!'
        self.enemy_header = self.font.render(self.enemy_header_text, True, self.game.BLACK)
        self.enemy_header_rect = self.enemy_header.get_rect(center=self.enemy_banner_rect.center)
    self.turn = 'Player'
```

- The enemy turn has similar logic to the player turn (the chance for the player to dodge is based on the player's agility, the chance for the enemy to hit the player is based on the enemy's speed, and the player will dodge if the chance to dodge is bigger than the chance to hit). Then it displays a text of how much damage the attack dealt or whether it hit the player or not. It then transitions back to the player's turn. This will continue looping until the player or the enemy is dead, or if the player flees the battle.

- 

```python
def battle(self, actions): #battle logic
    if self.player.hp > 0 and self.enemy.hp > 0: #checks if both player and enemy are alive
        if self.turn == 'Player':
            self.move_cursor(actions)
            self.player_turn(actions)
        else:
            self.enemy_turn()
    elif self.enemy.hp <= 0: #checks if the enemy is dead
        xp = randint(self.enemy.level.xp_range[0], self.enemy.level.xp_range[1])
        coins = randint(self.enemy.level.gold_range[0], self.enemy.level.gold_range[1])
        self.player.coins += coins
        self.player.xp += xp
        level_up = False
        if self.player.xp > self.player.level.xp: #checks if the player levels up
            rem_xp = self.player.xp - self.player.level.xp
            new_level = self.player.level.level + 1
            self.player.xp = rem_xp
            self.player.level = config.player_levels[new_level]
            level_up = True
        self.exit_state()
        self.game.next_state = BattleComplete(self.game, 'Battle Complete', xp, coins, level_up, self.player)
        self.game.next()
        pygame.mixer.music.load('assets/game.mp3')
        pygame.mixer_music.set_volume(0.7)
        pygame.mixer.music.play()
```

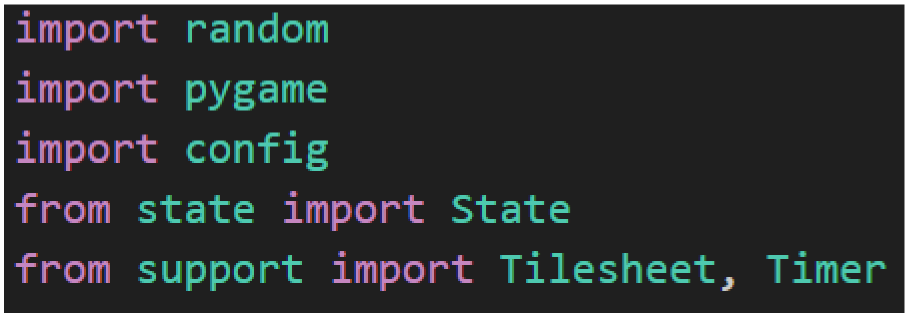
- If both player and enemy are alive, the battle continues and the turn goes to either the player or the enemy. If the enemy is dead then the game will check whether the player leveled up or not. If the player leveled up then it will return the values of the new level (current level, remaining xp before next level). It will then go to the battle complete screen.

- 

```python
    elif self.player.hp <= 0: #checks if the player is dead
        pygame.time.wait(30)
        self.game.player = None
        game_over = GameOver(self.game, 'Game Over') #switches to game over screen
        self.game.next_state = game_over
        self.game.next()
```

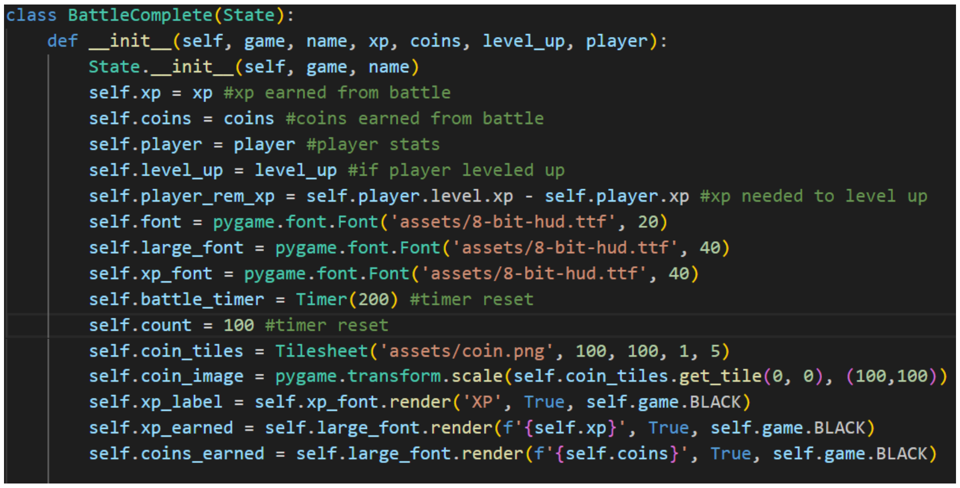
- If the player is dead however, the game ends, the player is removed, then it transitions to the game over screen.

5. **Battle Complete (battle_complete.py)**

- Displays the results of a battle, including experience points and gold earned.
- Handles level-up logic and stat increases.
- 

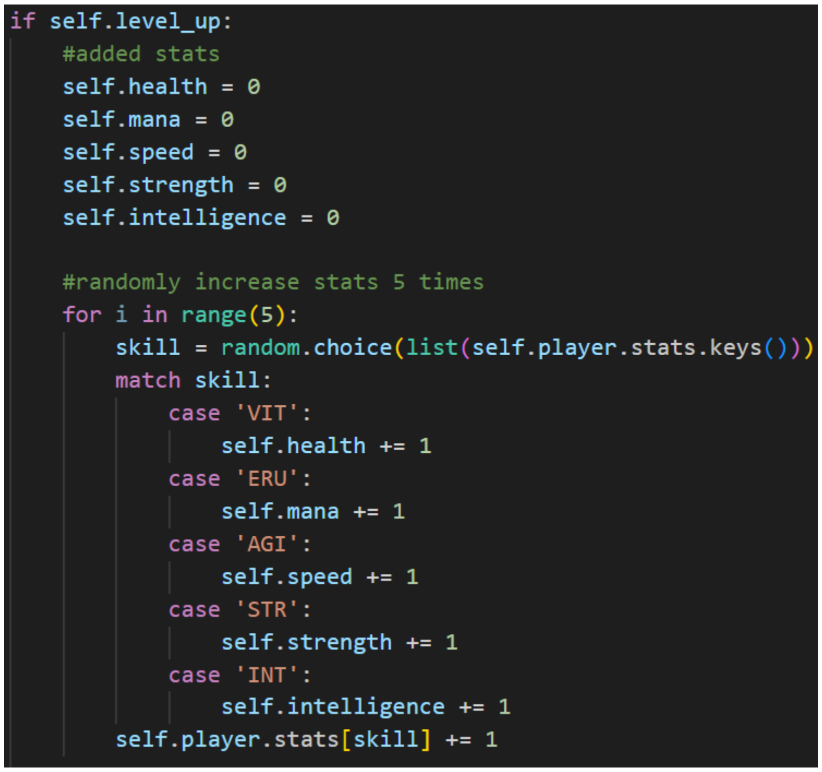
```python
import random
import pygame
import config
from state import State
from support import Tilesheet, Timer
```

- Initial imports, tilesheet for the coin image, timer to reset the timer.

- 

```python
class BattleComplete(State):
    def __init__(self, game, name, xp, coins, level_up, player):
        State.__init__(self, game, name)
        self.xp = xp #xp earned from battle
        self.coins = coins #coins earned from battle
        self.player = player #player stats
        self.level_up = level_up #if player leveled up
        self.player_rem_xp = self.player.level.xp - self.player.xp #xp needed to level up
        self.font = pygame.font.Font('assets/8-bit-hud.ttf', 20)
        self.large_font = pygame.font.Font('assets/8-bit-hud.ttf', 40)
        self.xp_font = pygame.font.Font('assets/8-bit-hud.ttf', 40)
        self.battle_timer = Timer(200) #timer reset
        self.count = 100 #timer reset
        self.coin_tiles = Tilesheet('assets/coin.png', 100, 100, 1, 5)
        self.coin_image = pygame.transform.scale(self.coin_tiles.get_tile(0, 0), (100,100))
        self.xp_label = self.xp_font.render('XP', True, self.game.BLACK)
        self.xp_earned = self.large_font.render(f'{self.xp}', True, self.game.BLACK)
        self.coins_earned = self.large_font.render(f'{self.coins}', True, self.game.BLACK)
```

- This section of the class controls the xp, coins, and stats of the players, as well as the level up status, and the remaining xp needed to level up. Fonts are also being set up here, along with the timer, images, and more text to be displayed on the screen.

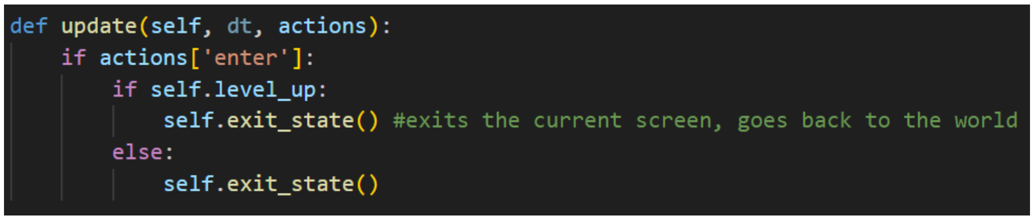
- 

```python
if self.level_up:
    #added stats
    self.health = 0
    self.mana = 0
    self.speed = 0
    self.strength = 0
    self.intelligence = 0

    #randomly increase stats 5 times
    for i in range(5):
        skill = random.choice(list(self.player.stats.keys()))
        match skill:
            case 'VIT':
                self.health += 1
            case 'ERU':
                self.mana += 1
            case 'AGI':
                self.speed += 1
            case 'STR':
                self.strength += 1
            case 'INT':
                self.intelligence += 1
        self.player.stats[skill] += 1
```

- If the player levels up, the game just randomly assigns 5 stat points to the attributes. It then displays which attribute was increased and the amount it was increased by.
- After this line of code, there's a long part where it's essentially just putting everything onto the screen, starting from updating the stats to match the new

amount, displaying the amount of points increased in each stat/attribute, to displaying how much xp is needed to level up again.

- There's also a render function in this file, used to actually put all of the things above onto the screen.

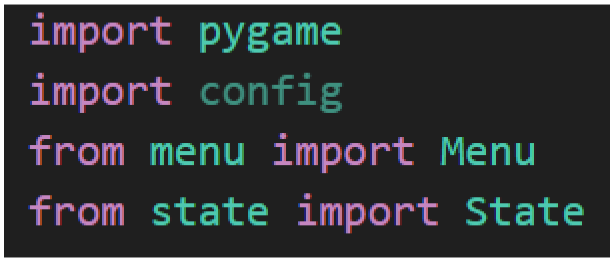
- 

```python
def update(self, dt, actions):
    if actions['enter']:
        if self.level_up:
            self.exit_state() #exits the current screen, goes back to the world
        else:
            self.exit_state()
```

- This makes sure that returning to the game world after you've completed a battle whether you leveled up or not, runs smoothly.

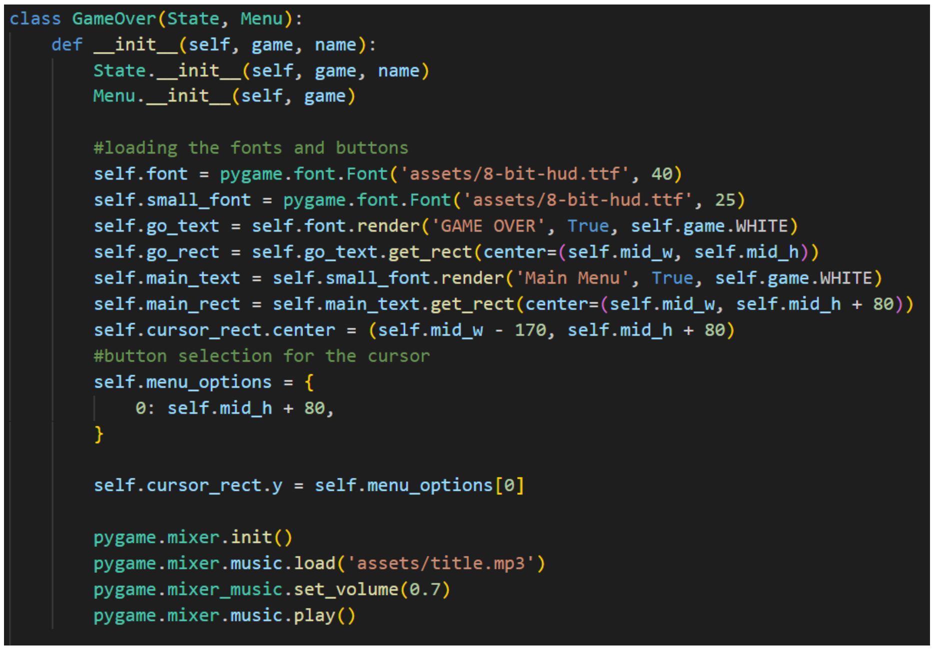
6. **Game Over (game_over.py)**

- Displays the game over screen.
- Allows the player to return to the main menu.

- 

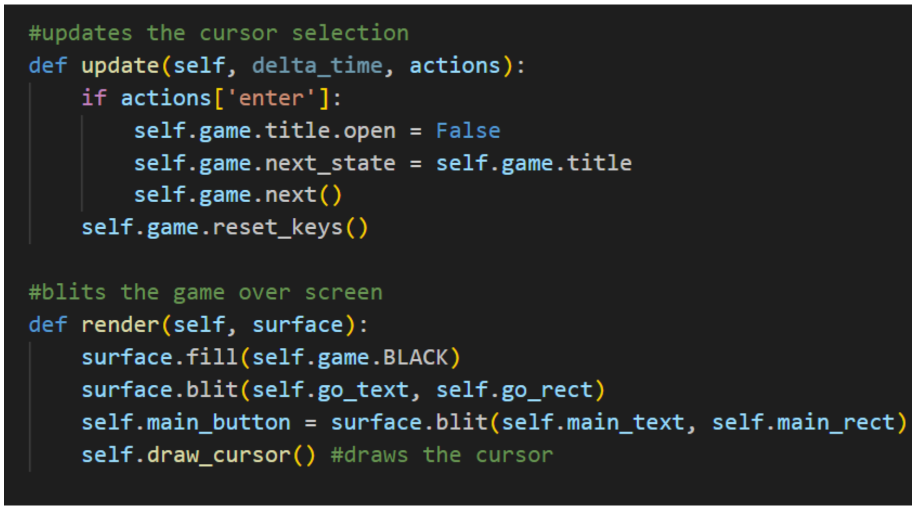
```python
import pygame
import config
from menu import Menu
from state import State
```

- Initial import

- 

```python
class GameOver(State, Menu):
    def __init__(self, game, name):
        State.__init__(self, game, name)
        Menu.__init__(self, game)

        #loading the fonts and buttons
        self.font = pygame.font.Font('assets/8-bit-hud.ttf', 40)
        self.small_font = pygame.font.Font('assets/8-bit-hud.ttf', 25)
        self.go_text = self.font.render('GAME OVER', True, self.game.WHITE)
        self.go_rect = self.go_text.get_rect(center=(self.mid_w, self.mid_h))
        self.main_text = self.small_font.render('Main Menu', True, self.game.WHITE)
        self.main_rect = self.main_text.get_rect(center=(self.mid_w, self.mid_h + 80))
        self.cursor_rect.center = (self.mid_w - 170, self.mid_h + 80)
        #button selection for the cursor
        self.menu_options = {
            0: self.mid_h + 80,
        }

        self.cursor_rect.y = self.menu_options[0]

        pygame.mixer.init()
        pygame.mixer.music.load('assets/title.mp3')
        pygame.mixer_music.set_volume(0.7)
        pygame.mixer.music.play()
```

- This controls the text that will be displayed on the game over screen, and controls the cursor so you can return to the main menu. Title screen music is played again here.

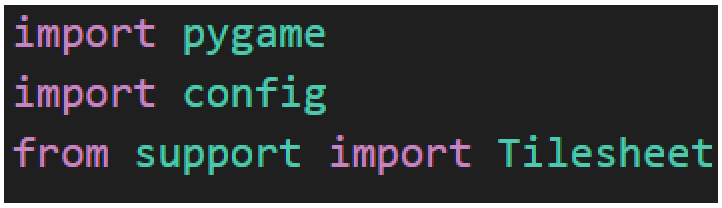
-


```python
#updates the cursor selection
def update(self, delta_time, actions):
    if actions['enter']:
        self.game.title.open = False
        self.game.next_state = self.game.title
        self.game.next()
    self.game.reset_keys()

#blits the game over screen
def render(self, surface):
    surface.fill(self.game.BLACK)
    surface.blit(self.go_text, self.go_rect)
    self.main_button = surface.blit(self.main_text, self.main_rect)
    self.draw_cursor() #draws the cursor
```

- Update function makes sure that you return to the title screen once you press enter, and render just makes everything in the init function visible, such as the game over text.

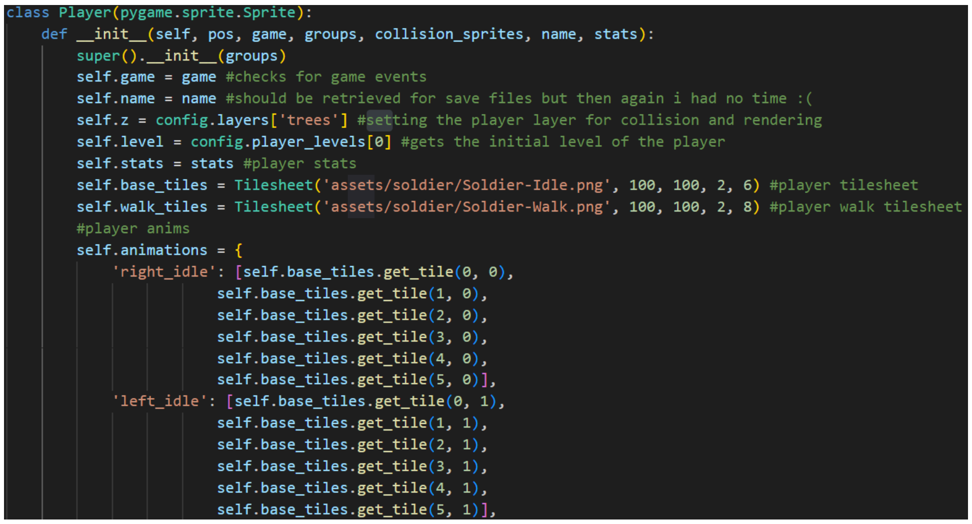
Supporting Classes

1. **Player (player.py)**

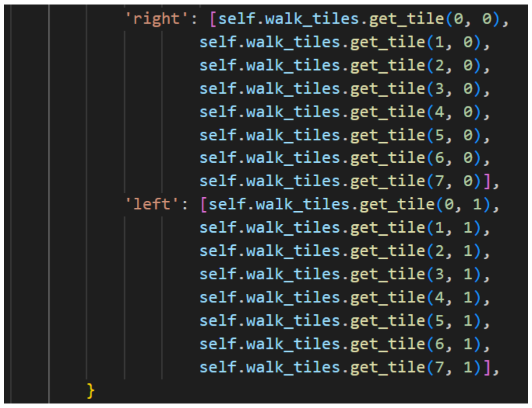
- Represents the player character.
- Manages player stats, inventory, movement, and animations.
-


```python
import pygame
import config
from support import Tilesheet
```

- Initial imports, config for determining layers, tilesheet to control the animations.

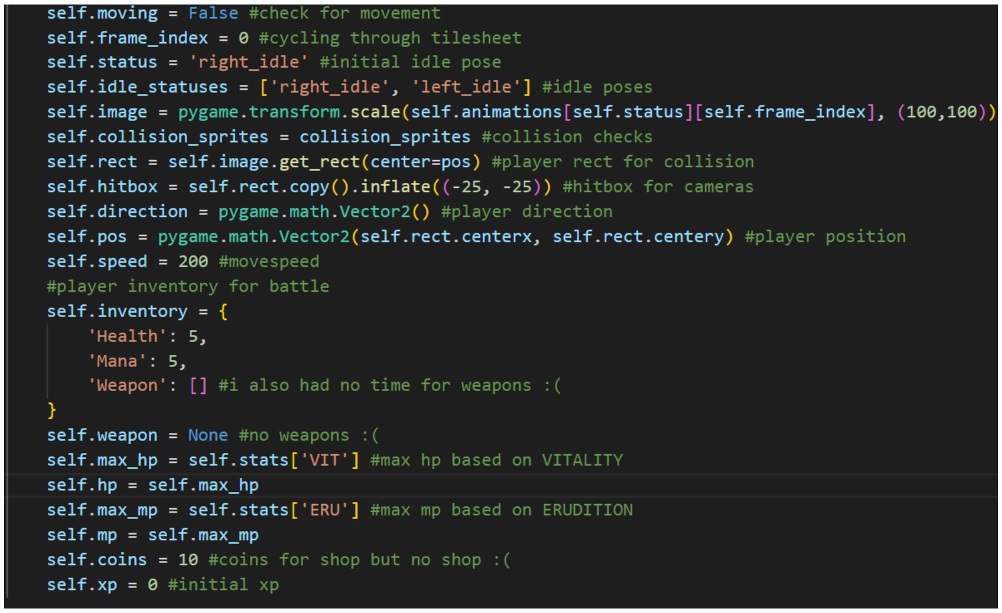
- 

```
class Player(pygame.sprite.Sprite):
    def __init__(self, pos, game, groups, collision_sprites, name, stats):
        super().__init__(groups)
        self.game = game #checks for game events
        self.name = name #should be retrieved for save files but then again i had no time :(
        self.z = config.layers['trees'] #setting the player layer for collision and rendering
        self.level = config.player_levels[0] #gets the initial level of the player
        self.stats = stats #player stats
        self.base_tiles = Tilesheet('assets/soldier/Soldier-Idle.png', 100, 100, 2, 6) #player tilesheet
        self.walk_tiles = Tilesheet('assets/soldier/Soldier-Walk.png', 100, 100, 2, 8) #player walk tilesheet
        #player anims
        self.animations = {
            'right_idle': [self.base_tiles.get_tile(0, 0),
                        self.base_tiles.get_tile(1, 0),
                        self.base_tiles.get_tile(2, 0),
                        self.base_tiles.get_tile(3, 0),
                        self.base_tiles.get_tile(4, 0),
                        self.base_tiles.get_tile(5, 0)],
            'left_idle': [self.base_tiles.get_tile(0, 1),
                        self.base_tiles.get_tile(1, 1),
                        self.base_tiles.get_tile(2, 1),
                        self.base_tiles.get_tile(3, 1),
                        self.base_tiles.get_tile(4, 1),
                        self.base_tiles.get_tile(5, 1)],
```

- 

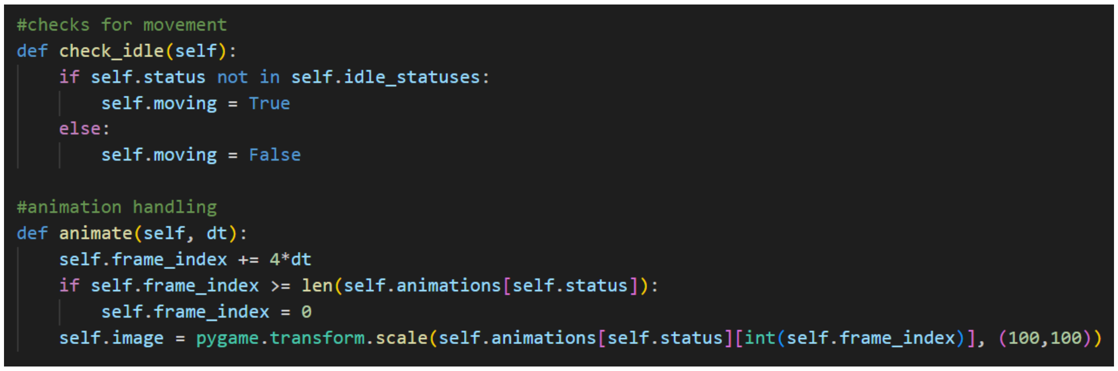
```
            'right': [self.walk_tiles.get_tile(0, 0),
                    self.walk_tiles.get_tile(1, 0),
                    self.walk_tiles.get_tile(2, 0),
                    self.walk_tiles.get_tile(3, 0),
                    self.walk_tiles.get_tile(4, 0),
                    self.walk_tiles.get_tile(5, 0),
                    self.walk_tiles.get_tile(6, 0),
                    self.walk_tiles.get_tile(7, 0)],
            'left': [self.walk_tiles.get_tile(0, 1),
                    self.walk_tiles.get_tile(1, 1),
                    self.walk_tiles.get_tile(2, 1),
                    self.walk_tiles.get_tile(3, 1),
                    self.walk_tiles.get_tile(4, 1),
                    self.walk_tiles.get_tile(5, 1),
                    self.walk_tiles.get_tile(6, 1),
                    self.walk_tiles.get_tile(7, 1)],
        }
```

- get_tiles is a function in tilesheet that takes every image within the tilesheet and breaks them up into sections, hence the individual tiles being put into a list within a dictionary. These lists serve as a collection of images to cycle through, so that the player character will be animated.

- 

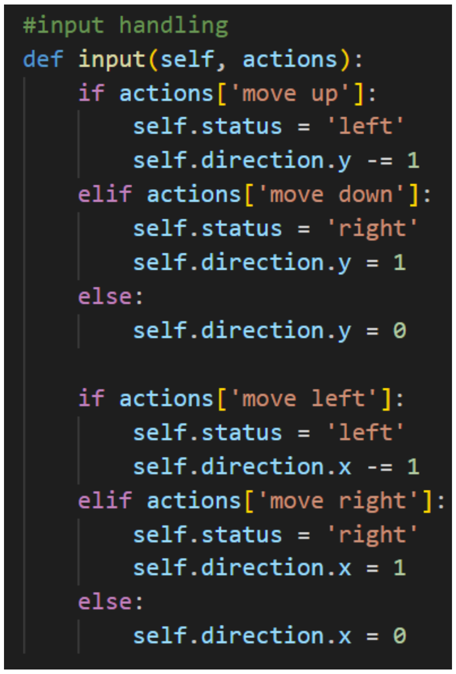
```
self.moving = False #check for movement
self.frame_index = 0 #cycling through tilesheet
self.status = 'right_idle' #initial idle pose
self.idle_statuses = ['right_idle', 'left_idle'] #idle poses
self.image = pygame.transform.scale(self.animations[self.status][self.frame_index], (100,100))
self.collision_sprites = collision_sprites #collision checks
self.rect = self.image.get_rect(center=pos) #player rect for collision
self.hitbox = self.rect.copy().inflate((-25, -25)) #hitbox for cameras
self.direction = pygame.math.Vector2() #player direction
self.pos = pygame.math.Vector2(self.rect.centerx, self.rect.centery) #player position
self.speed = 200 #movespeed
#player inventory for battle
self.inventory = {
    'Health': 5,
    'Mana': 5,
    'Weapon': [] #i also had no time for weapons :(
}
self.weapon = None #no weapons :(
self.max_hp = self.stats['VIT'] #max hp based on VITALITY
self.hp = self.max_hp
self.max_mp = self.stats['ERU'] #max mp based on ERUDITION
self.mp = self.max_mp
self.coins = 10 #coins for shop but no shop :(
self.xp = 0 #initial xp
```

- Moving is false as default, because the player starts from an idle position.
- Frame index cycles through the list of tiles or frames so the player can be animated.
- Status is set to right_idle as default, so when the player is idling for the first time, the animation that will play is the right facing idle animation.
- I then put the idle statuses into a list, so that when the player is moving, the status will not be in either of the items in the list, meaning that the moving variable will be set to true (will be shown below).
- The player image is then loaded with the size of 100x100 px
- Then collision_sprites, rect, and hitbox is to control the hitbox of the player.
- The direction and position is determined using the vector2 function that is built into pygame.
- Inventory is used as a place to store potions and supposedly weapons (i had no time).
- Initial weapon is none, and the player is supposed to get coins then buy a weapon at the weapon shop. This was unused.
- The hp, mp, coins, and initial xp amount is then set.

- 

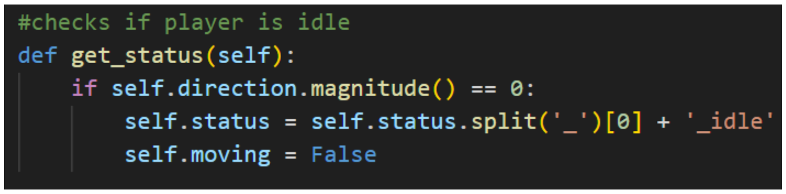
```python
#checks for movement
def check_idle(self):
    if self.status not in self.idle_statuses:
        self.moving = True
    else:
        self.moving = False

#animation handling
def animate(self, dt):
    self.frame_index += 4*dt
    if self.frame_index >= len(self.animations[self.status]):
        self.frame_index = 0
    self.image = pygame.transform.scale(self.animations[self.status][int(self.frame_index)], (100,100))
```

- check_idle checks whether the player is idle or not, and if the player is not idle, then the animation moves to the moving animation.
- animate makes sure that no matter how slow or fast your device frame rate may be, the animation of the player will not be affected.

- 

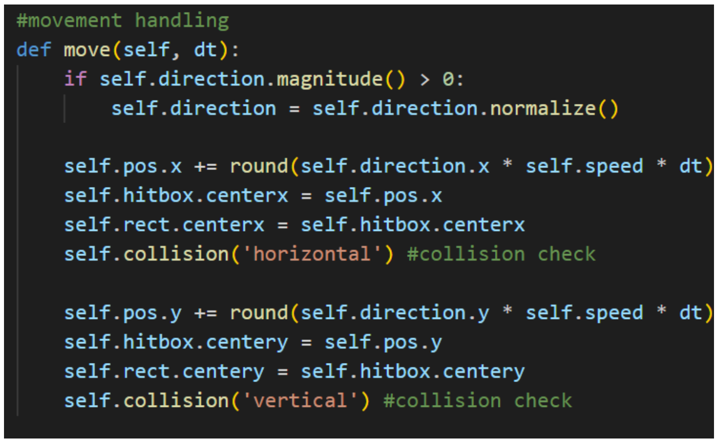
```python
#input handling
def input(self, actions):
    if actions['move up']:
        self.status = 'left'
        self.direction.y -= 1
    elif actions['move down']:
        self.status = 'right'
        self.direction.y = 1
    else:
        self.direction.y = 0

    if actions['move left']:
        self.status = 'left'
        self.direction.x -= 1
    elif actions['move right']:
        self.status = 'right'
        self.direction.x = 1
    else:
        self.direction.x = 0
```

- input handles the player movement and moving animation.

- 

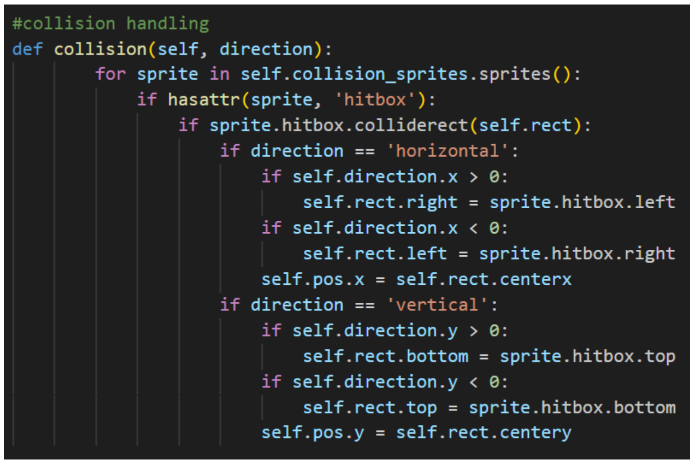
```python
#checks if player is idle
def get_status(self):
    if self.direction.magnitude() == 0:
        self.status = self.status.split('_')[0] + '_idle'
        self.moving = False
```

- get_status is used to make sure that the animation matches the current movement status of the player. It specifically checks whether the player is moving or not.



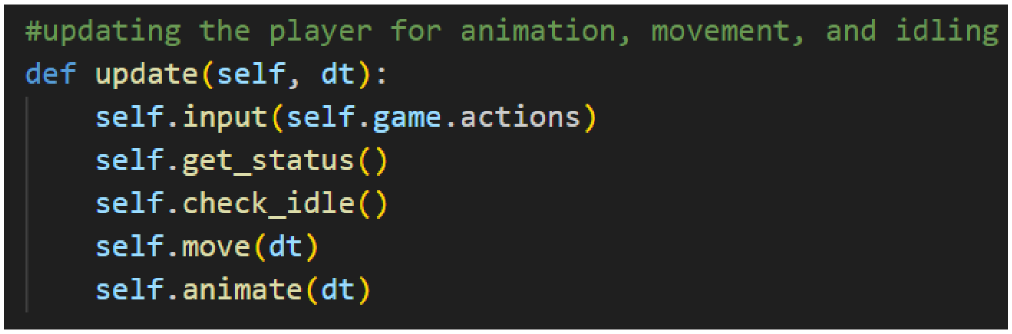
```
#movement handling
def move(self, dt):
    if self.direction.magnitude() > 0:
        self.direction = self.direction.normalize()

    self.pos.x += round(self.direction.x * self.speed * dt)
    self.hitbox.centerx = self.pos.x
    self.rect.centerx = self.hitbox.centerx
    self.collision('horizontal') #collision check

    self.pos.y += round(self.direction.y * self.speed * dt)
    self.hitbox.centery = self.pos.y
    self.rect.centery = self.hitbox.centery
    self.collision('vertical') #collision check
```

-
- move controls player movement by multiplying the direction amount (either 1 or -1) with the speed and dt to make sure that frame rate won't affect the speed. It also checks for collisions.



```
#collision handling
def collision(self, direction):
        for sprite in self.collision_sprites.sprites():
            if hasattr(sprite, 'hitbox'):
                if sprite.hitbox.colliderect(self.rect):
                    if direction == 'horizontal':
                        if self.direction.x > 0:
                            self.rect.right = sprite.hitbox.left
                        if self.direction.x < 0:
                            self.rect.left = sprite.hitbox.right
                        self.pos.x = self.rect.centerx
                    if direction == 'vertical':
                        if self.direction.y > 0:
                            self.rect.bottom = sprite.hitbox.top
                        if self.direction.y < 0:
                            self.rect.top = sprite.hitbox.bottom
                        self.pos.y = self.rect.centery
```

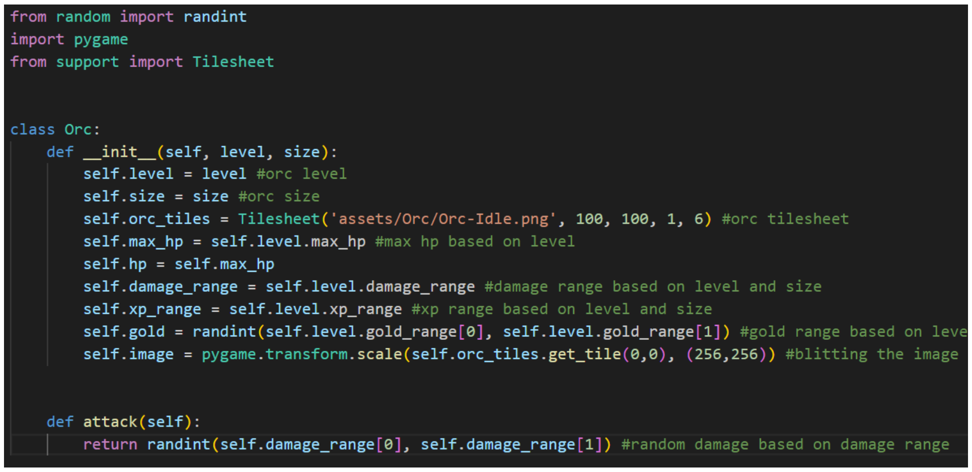
-
- The collision works by checking whether the player sprite hitbox has hit an object hitbox or not. When it hits, it resets the player position so it won't phase through the object, and so that it stays in place.



```
#updating the player for animation, movement, and idling
def update(self, dt):
    self.input(self.game.actions)
    self.get_status()
    self.check_idle()
    self.move(dt)
    self.animate(dt)
```

-
- update then updates the player accordingly, based on the status of the player.

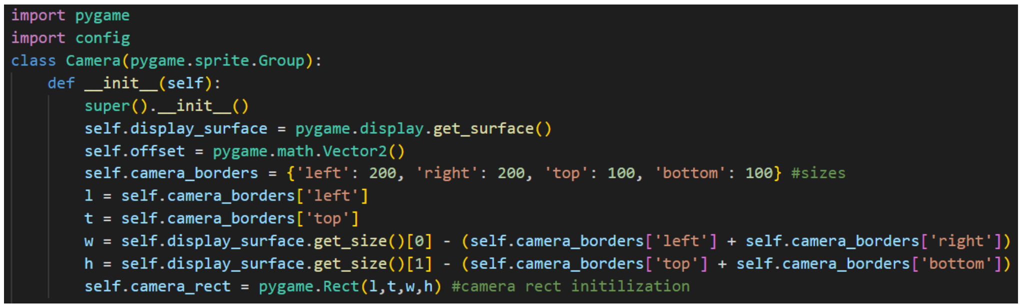
2. Orc (orc.py)

- Represents enemy orcs.
- Manages orc stats, animations, and attack logic.
- 

```python
from random import randint
import pygame
from support import Tilesheet


class Orc:
    def __init__(self, level, size):
        self.level = level #orc level
        self.size = size #orc size
        self.orc_tiles = Tilesheet('assets/Orc/Orc-Idle.png', 100, 100, 1, 6) #orc tilesheet
        self.max_hp = self.level.max_hp #max hp based on level
        self.hp = self.max_hp
        self.damage_range = self.level.damage_range #damage range based on level and size
        self.xp_range = self.level.xp_range #xp range based on level and size
        self.gold = randint(self.level.gold_range[0], self.level.gold_range[1]) #gold range based on leve
        self.image = pygame.transform.scale(self.orc_tiles.get_tile(0,0), (256,256)) #blitting the image


    def attack(self):
        return randint(self.damage_range[0], self.damage_range[1]) #random damage based on damage range
```

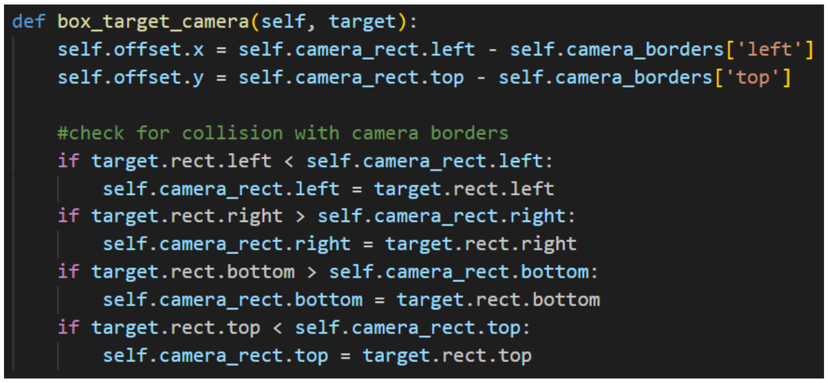
- Level, size, hp, damage, xp, and gold gain is determined by the level and size of the orc.
- attack controls how much damage the orc will deal based on its level, but it's still randomized within a certain range.

3. Camera (camera.py)

- Manages the camera view and follows the player around the map.
- Handles rendering of game objects based on their layers.
- 

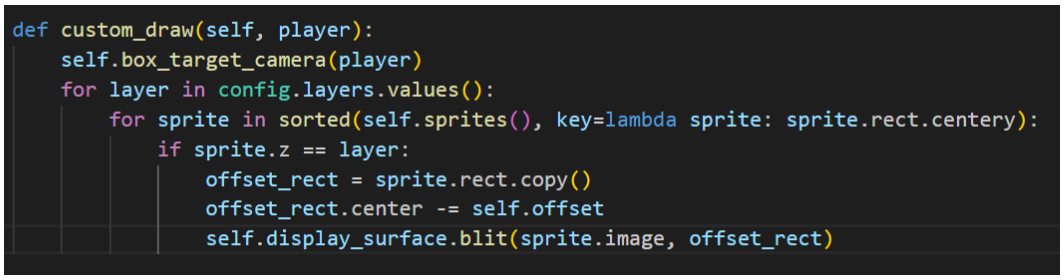
```python
import pygame
import config
class Camera(pygame.sprite.Group):
    def __init__(self):
        super().__init__()
        self.display_surface = pygame.display.get_surface()
        self.offset = pygame.math.Vector2()
        self.camera_borders = {'left': 200, 'right': 200, 'top': 100, 'bottom': 100} #sizes
        l = self.camera_borders['left']
        t = self.camera_borders['top']
        w = self.display_surface.get_size()[0] - (self.camera_borders['left'] + self.camera_borders['right'])
        h = self.display_surface.get_size()[1] - (self.camera_borders['top'] + self.camera_borders['bottom'])
        self.camera_rect = pygame.Rect(l,t,w,h) #camera rect initilization
```

- The init function creates a big rectangle that wraps around the player.

- 

```python
def box_target_camera(self, target):
    self.offset.x = self.camera_rect.left - self.camera_borders['left']
    self.offset.y = self.camera_rect.top - self.camera_borders['top']

    #check for collision with camera borders
    if target.rect.left < self.camera_rect.left:
        self.camera_rect.left = target.rect.left
    if target.rect.right > self.camera_rect.right:
        self.camera_rect.right = target.rect.right
    if target.rect.bottom > self.camera_rect.bottom:
        self.camera_rect.bottom = target.rect.bottom
    if target.rect.top < self.camera_rect.top:
        self.camera_rect.top = target.rect.top
```

- The big rectangle then checks whether it has collided with the edge of the map or not, then it determines when the camera should move.

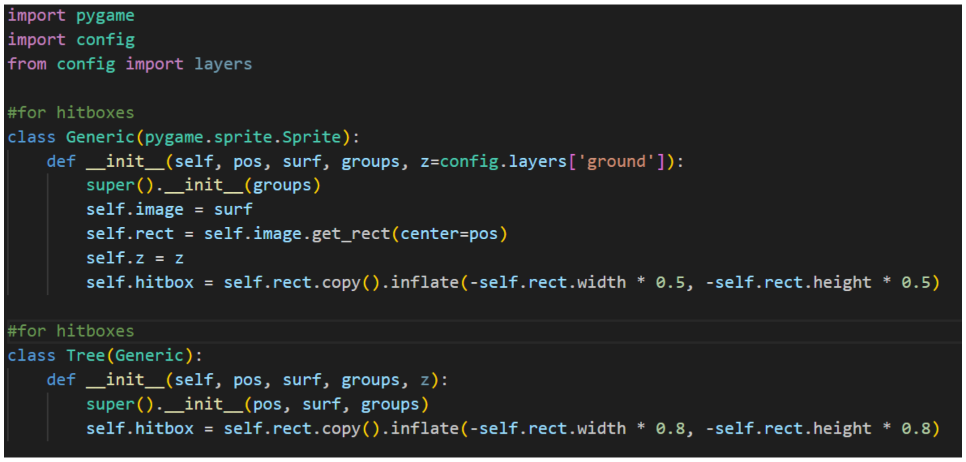
- 

```python
def custom_draw(self, player):
    self.box_target_camera(player)
    for layer in config.layers.values():
        for sprite in sorted(self.sprites(), key=lambda sprite: sprite.rect.centery):
            if sprite.z == layer:
                offset_rect = sprite.rect.copy()
                offset_rect.center -= self.offset
                self.display_surface.blit(sprite.image, offset_rect)
```

- custom_draw then makes sure that the camera rectangle is center to the player, then it makes sure that the layers load properly.

4. **Sprites (sprites.py)**

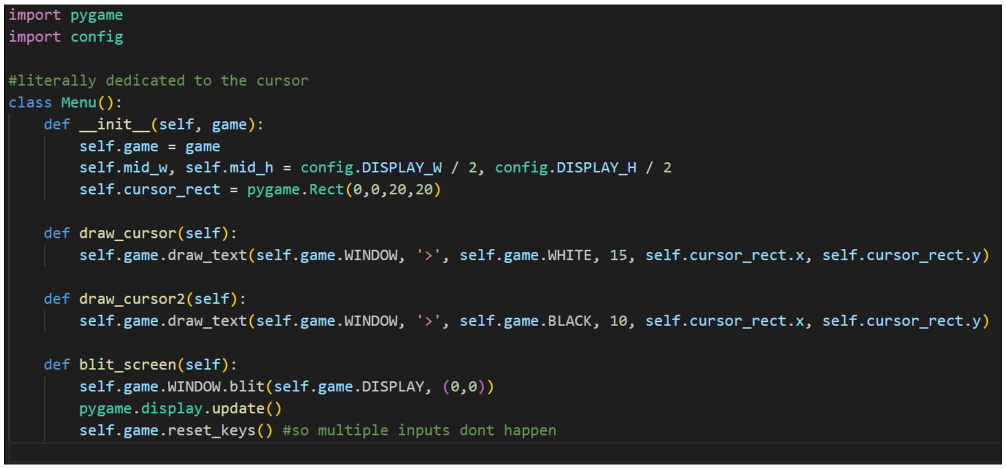
- Contains generic sprite classes for game objects like trees and hitboxes.

- 

```python
import pygame
import config
from config import layers

#for hitboxes
class Generic(pygame.sprite.Sprite):
    def __init__(self, pos, surf, groups, z=config.layers['ground']):
        super().__init__(groups)
        self.image = surf
        self.rect = self.image.get_rect(center=pos)
        self.z = z
        self.hitbox = self.rect.copy().inflate(-self.rect.width * 0.5, -self.rect.height * 0.5)

#for hitboxes
class Tree(Generic):
    def __init__(self, pos, surf, groups, z):
        super().__init__(pos, surf, groups)
        self.hitbox = self.rect.copy().inflate(-self.rect.width * 0.8, -self.rect.height * 0.8)
```

5. **Menu (menu.py)**

- Manages the cursor and menu interactions.

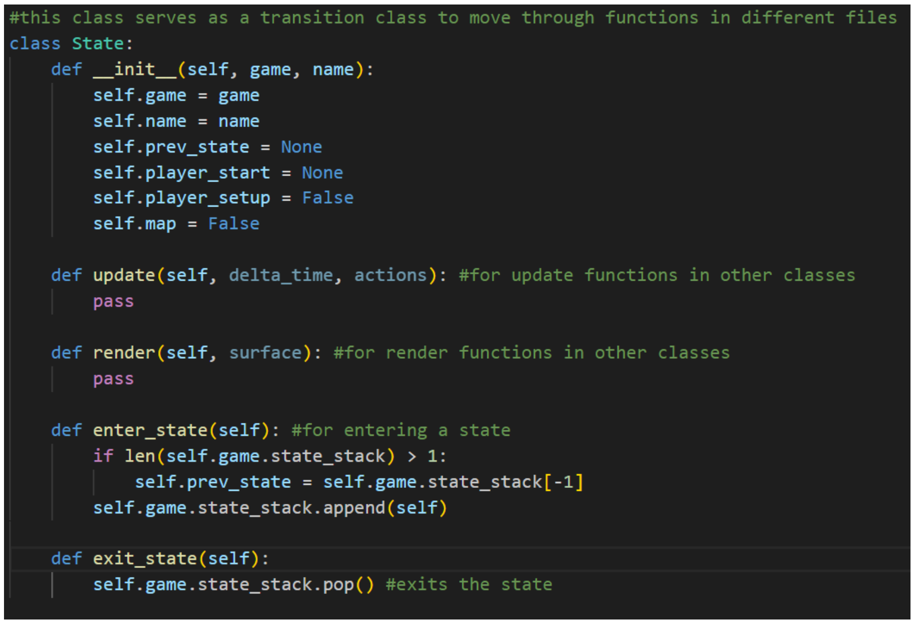


```
import pygame
import config

#literally dedicated to the cursor
class Menu():
    def __init__(self, game):
        self.game = game
        self.mid_w, self.mid_h = config.DISPLAY_W / 2, config.DISPLAY_H / 2
        self.cursor_rect = pygame.Rect(0,0,20,20)

    def draw_cursor(self):
        self.game.draw_text(self.game.WINDOW, '>', self.game.WHITE, 15, self.cursor_rect.x, self.cursor_rect.y)

    def draw_cursor2(self):
        self.game.draw_text(self.game.WINDOW, '>', self.game.BLACK, 10, self.cursor_rect.x, self.cursor_rect.y)

    def blit_screen(self):
        self.game.WINDOW.blit(self.game.DISPLAY, (0,0))
        pygame.display.update()
        self.game.reset_keys() #so multiple inputs dont happen
```

-

6. State (state.py)

- Base class for game states.
- Manages state transitions and updates.

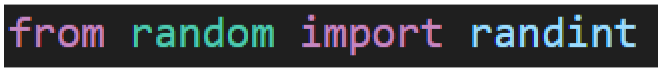


```
#this class serves as a transition class to move through functions in different files
class State:
    def __init__(self, game, name):
        self.game = game
        self.name = name
        self.prev_state = None
        self.player_start = None
        self.player_setup = False
        self.map = False

    def update(self, delta_time, actions): #for update functions in other classes
        pass

    def render(self, surface): #for render functions in other classes
        pass

    def enter_state(self): #for entering a state
        if len(self.game.state_stack) > 1:
            self.prev_state = self.game.state_stack[-1]
        self.game.state_stack.append(self)

    def exit_state(self):
        self.game.state_stack.pop() #exits the state
```

-

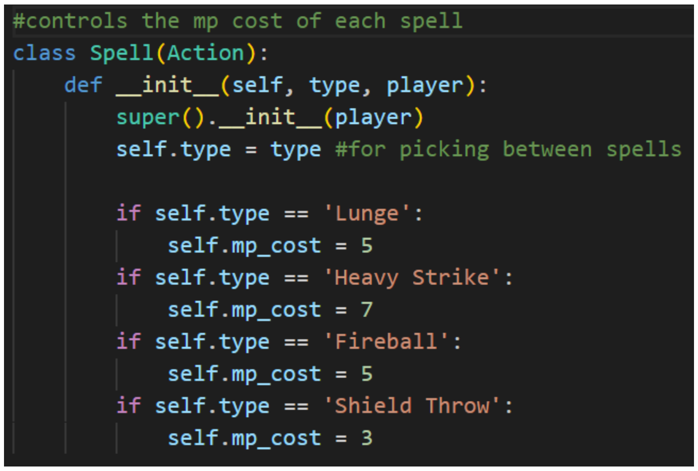
7. Action (action.py)

- Represents actions that can be performed in battle, such as attacks and spells.



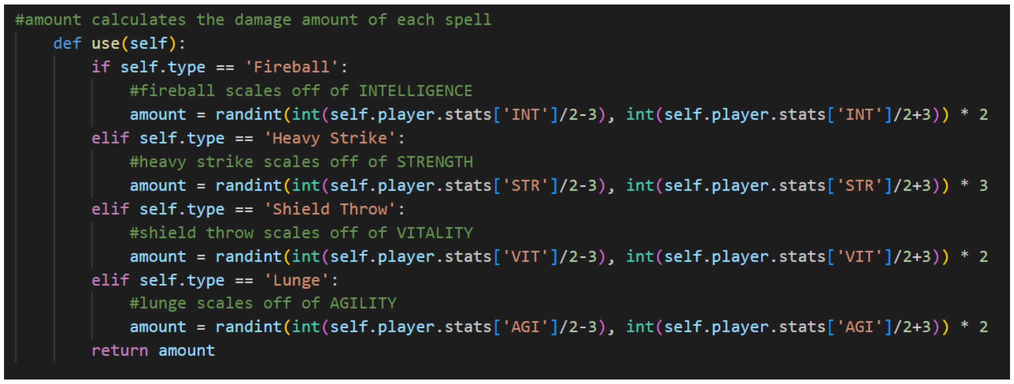
```
from random import randint
```

-
- Randint used to randomize the damage.

- 

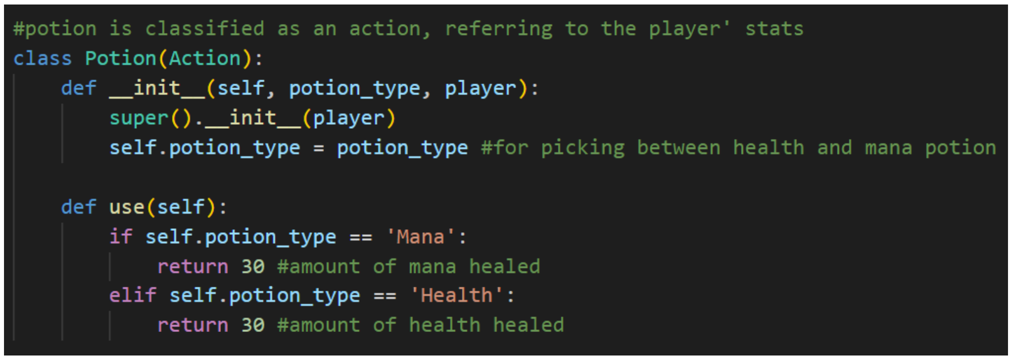
```python
#controls the mp cost of each spell
class Spell(Action):
    def __init__(self, type, player):
        super().__init__(player)
        self.type = type #for picking between spells

        if self.type == 'Lunge':
            self.mp_cost = 5
        if self.type == 'Heavy Strike':
            self.mp_cost = 7
        if self.type == 'Fireball':
            self.mp_cost = 5
        if self.type == 'Shield Throw':
            self.mp_cost = 3
```

- This controls the mp cost of each spell

- 

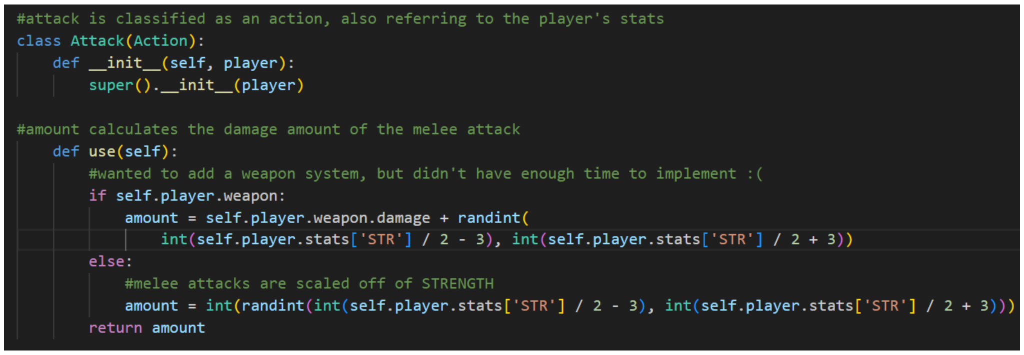
```python
#amount calculates the damage amount of each spell
    def use(self):
        if self.type == 'Fireball':
            #fireball scales off of INTELLIGENCE
            amount = randint(int(self.player.stats['INT']/2-3), int(self.player.stats['INT']/2+3)) * 2
        elif self.type == 'Heavy Strike':
            #heavy strike scales off of STRENGTH
            amount = randint(int(self.player.stats['STR']/2-3), int(self.player.stats['STR']/2+3)) * 3
        elif self.type == 'Shield Throw':
            #shield throw scales off of VITALITY
            amount = randint(int(self.player.stats['VIT']/2-3), int(self.player.stats['VIT']/2+3)) * 2
        elif self.type == 'Lunge':
            #lunge scales off of AGILITY
            amount = randint(int(self.player.stats['AGI']/2-3), int(self.player.stats['AGI']/2+3)) * 2
        return amount
```

- Then this controls the damage amount of each spell

- 

```python
#potion is classified as an action, referring to the player' stats
class Potion(Action):
    def __init__(self, potion_type, player):
        super().__init__(player)
        self.potion_type = potion_type #for picking between health and mana potion

    def use(self):
        if self.potion_type == 'Mana':
            return 30 #amount of mana healed
        elif self.potion_type == 'Health':
            return 30 #amount of health healed
```

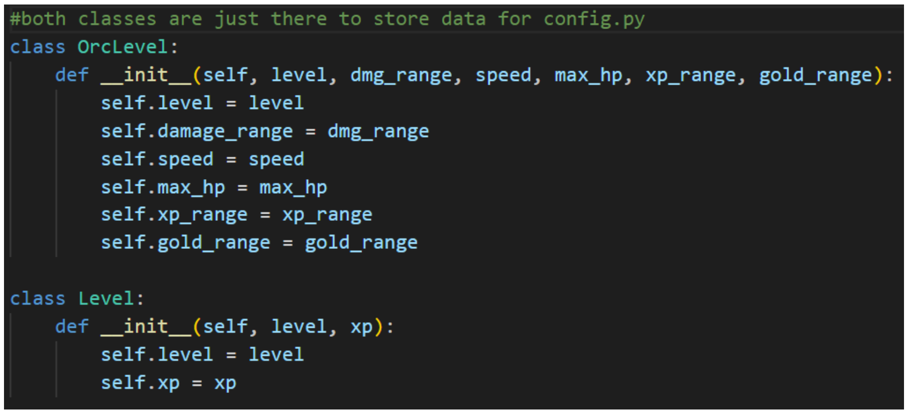
- This controls the amount healed by the potions.

- 

```python
#attack is classified as an action, also referring to the player's stats
class Attack(Action):
    def __init__(self, player):
        super().__init__(player)

#amount calculates the damage amount of the melee attack
    def use(self):
        #wanted to add a weapon system, but didn't have enough time to implement :(
        if self.player.weapon:
            amount = self.player.weapon.damage + randint(
                int(self.player.stats['STR'] / 2 - 3), int(self.player.stats['STR'] / 2 + 3))
        else:
            #melee attacks are scaled off of STRENGTH
            amount = int(randint(int(self.player.stats['STR'] / 2 - 3), int(self.player.stats['STR'] / 2 + 3)))
        return amount
```

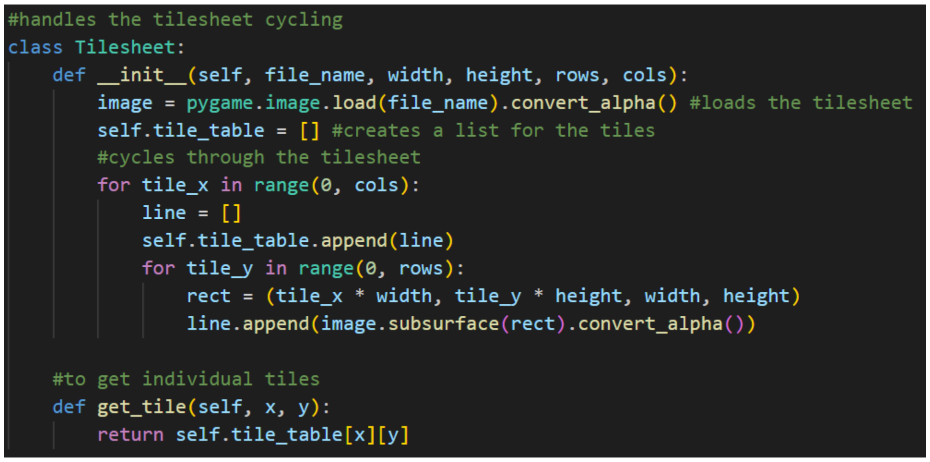
- This controls the amount of damage a melee attack does.
- Spell casting and melee attacks scale off of stats, and the damage is determined randomly within a certain range.

8. Level (level.py)

- Represents player and orc levels and their associated stats.
- 

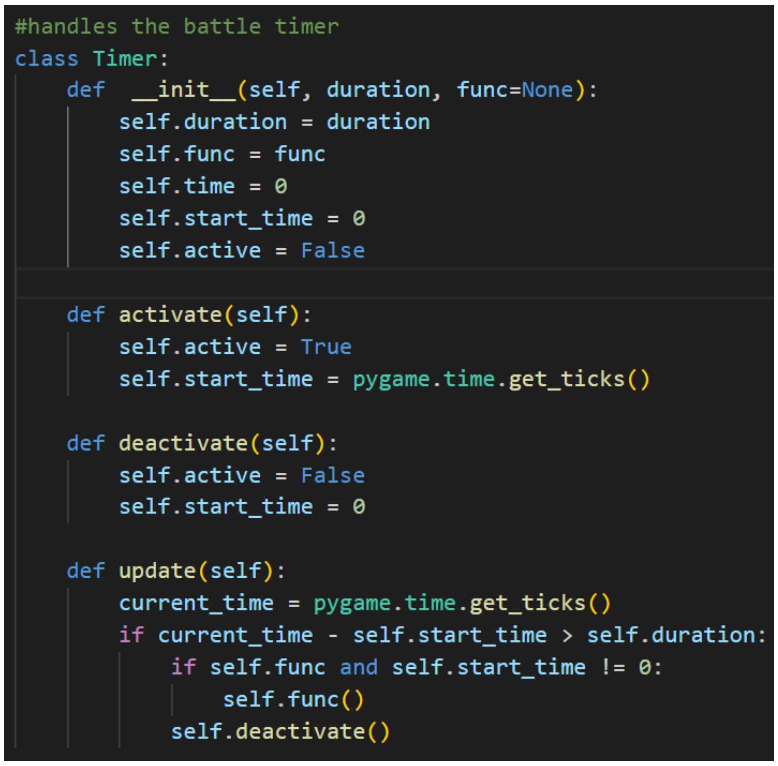
```python
#both classes are just there to store data for config.py
class OrcLevel:
    def __init__(self, level, dmg_range, speed, max_hp, xp_range, gold_range):
        self.level = level
        self.damage_range = dmg_range
        self.speed = speed
        self.max_hp = max_hp
        self.xp_range = xp_range
        self.gold_range = gold_range

class Level:
    def __init__(self, level, xp):
        self.level = level
        self.xp = xp
```

9. Support (support.py)

- Contains utility classes for handling tilesheets and timers.
- 

```python
#handles the tilesheet cycling
class Tilesheet:
    def __init__(self, file_name, width, height, rows, cols):
        image = pygame.image.load(file_name).convert_alpha() #loads the tilesheet
        self.tile_table = [] #creates a list for the tiles
        #cycles through the tilesheet
        for tile_x in range(0, cols):
            line = []
            self.tile_table.append(line)
            for tile_y in range(0, rows):
                rect = (tile_x * width, tile_y * height, width, height)
                line.append(image.subsurface(rect).convert_alpha())

    #to get individual tiles
    def get_tile(self, x, y):
        return self.tile_table[x][y]
```

- The init function loads the tilesheet and creates a list for it, then appends each and every tile that is separated from the tilesheet into the list. The get_tile function is there to call on a specific tile that is within the tile list.

- 

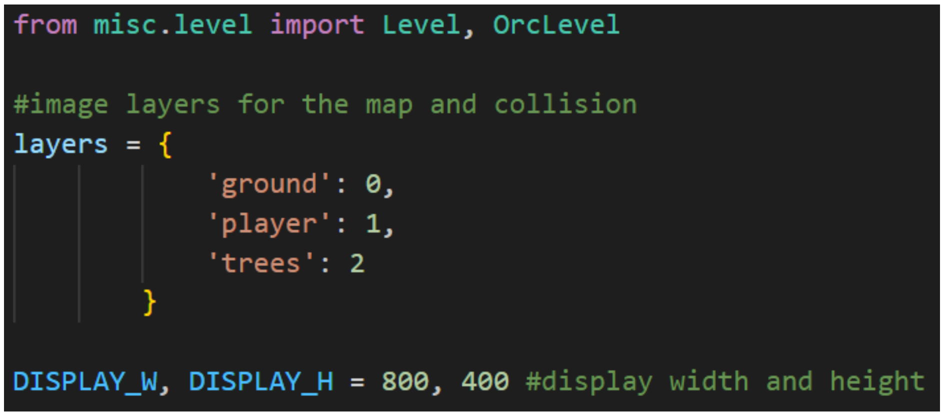
```python
#handles the battle timer
class Timer:
    def  __init__(self, duration, func=None):
        self.duration = duration
        self.func = func
        self.time = 0
        self.start_time = 0
        self.active = False

    def activate(self):
        self.active = True
        self.start_time = pygame.time.get_ticks()

    def deactivate(self):
        self.active = False
        self.start_time = 0

    def update(self):
        current_time = pygame.time.get_ticks()
        if current_time - self.start_time > self.duration:
            if self.func and self.start_time != 0:
                self.func()
            self.deactivate()
```

- This controls the timer for battles, and it dictates how long it takes before the timer deactivates and the battle starts.

Configuration

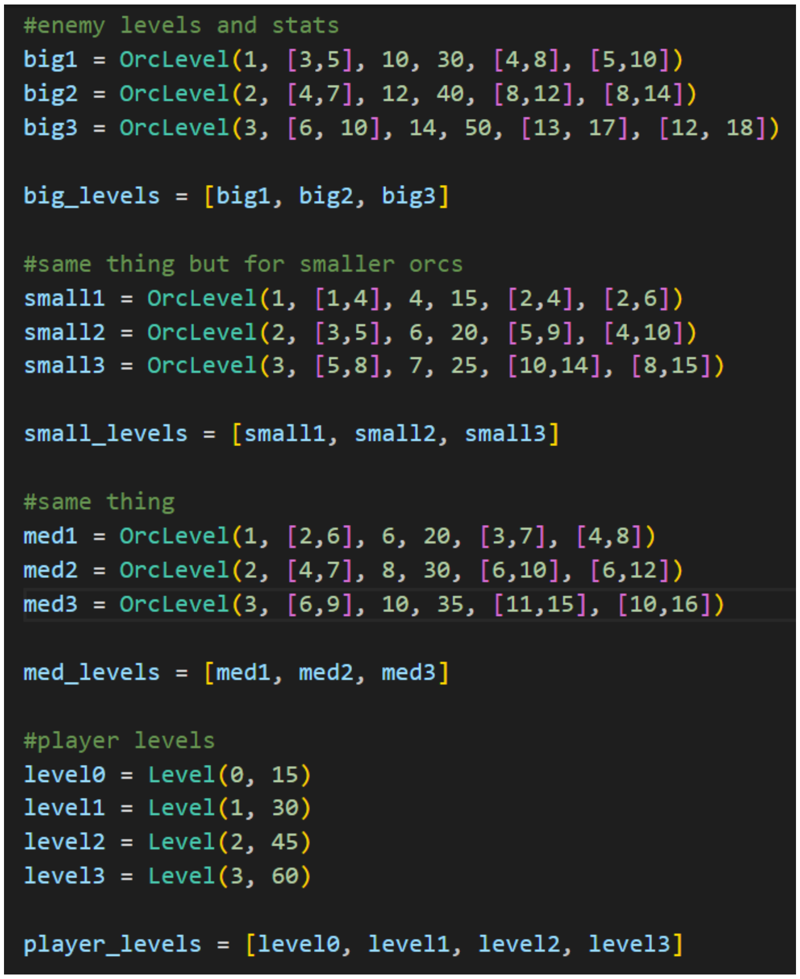
1. **config.py**

- Contains game settings, such as display dimensions and layer configurations.

- 

```python
from misc.level import Level, OrcLevel

#image layers for the map and collision
layers = {
            'ground': 0,
            'player': 1,
            'trees': 2
        }

DISPLAY_W, DISPLAY_H = 800, 400 #display width and height
```

- Defines levels and stats for player and orcs.



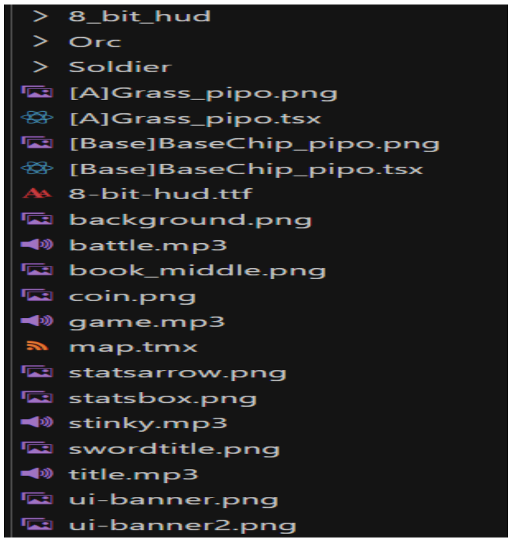
```
#enemy levels and stats
big1 = OrcLevel(1, [3,5], 10, 30, [4,8], [5,10])
big2 = OrcLevel(2, [4,7], 12, 40, [8,12], [8,14])
big3 = OrcLevel(3, [6, 10], 14, 50, [13, 17], [12, 18])

big_levels = [big1, big2, big3]

#same thing but for smaller orcs
small1 = OrcLevel(1, [1,4], 4, 15, [2,4], [2,6])
small2 = OrcLevel(2, [3,5], 6, 20, [5,9], [4,10])
small3 = OrcLevel(3, [5,8], 7, 25, [10,14], [8,15])

small_levels = [small1, small2, small3]

#same thing
med1 = OrcLevel(1, [2,6], 6, 20, [3,7], [4,8])
med2 = OrcLevel(2, [4,7], 8, 30, [6,10], [6,12])
med3 = OrcLevel(3, [6,9], 10, 35, [11,15], [10,16])

med_levels = [med1, med2, med3]

#player levels
level0 = Level(0, 15)
level1 = Level(1, 30)
level2 = Level(2, 45)
level3 = Level(3, 60)

player_levels = [level0, level1, level2, level3]
```

-

Assets

1. **Assets folder**

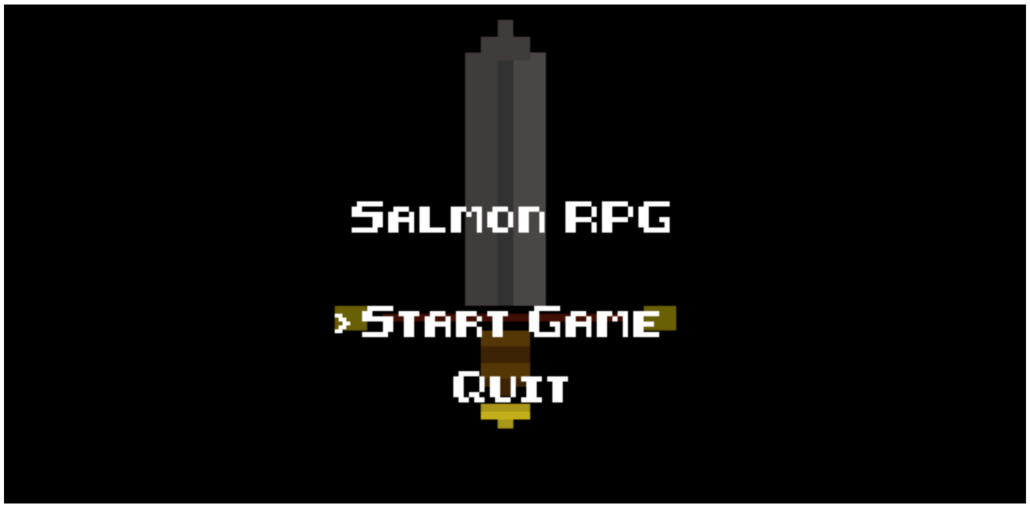
- Directory containing game assets such as images, fonts, and sounds.



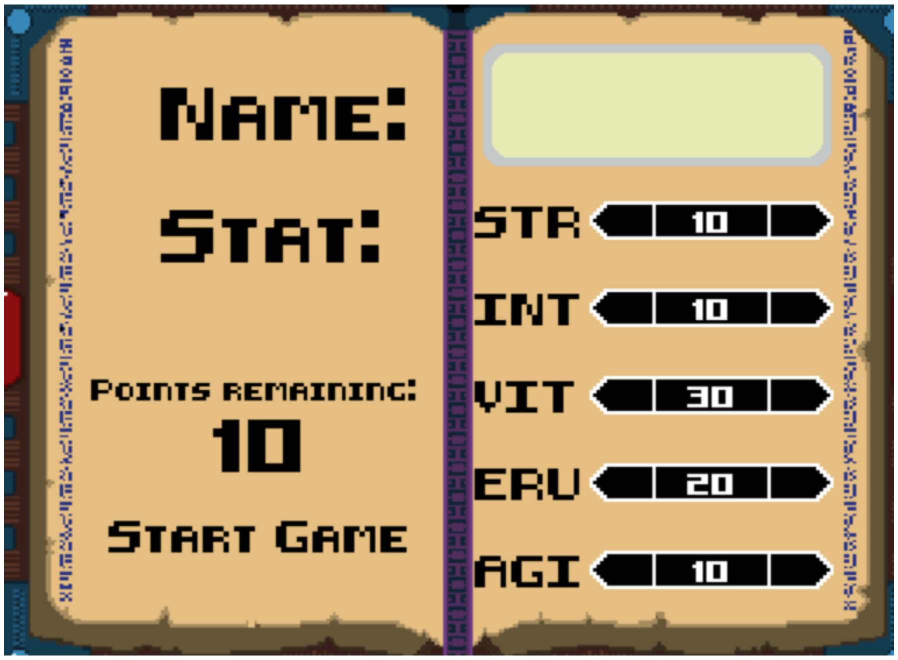
```
> 8_bit_hud
> Orc
> Soldier
[A]Grass_pipo.png
[A]Grass_pipo.tsx
[Base]BaseChip_pipo.png
[Base]BaseChip_pipo.tsx
8-bit-hud.ttf
background.png
battle.mp3
book_middle.png
coin.png
game.mp3
map.tmx
statsarrow.png
statsbox.png
stinky.mp3
swordtitle.png
title.mp3
ui-banner.png
ui-banner2.png
```

-

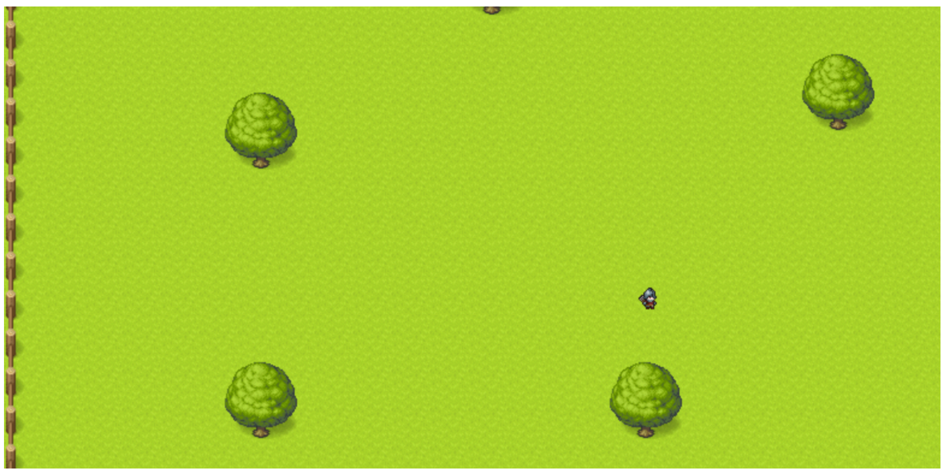
Proof of Working Code

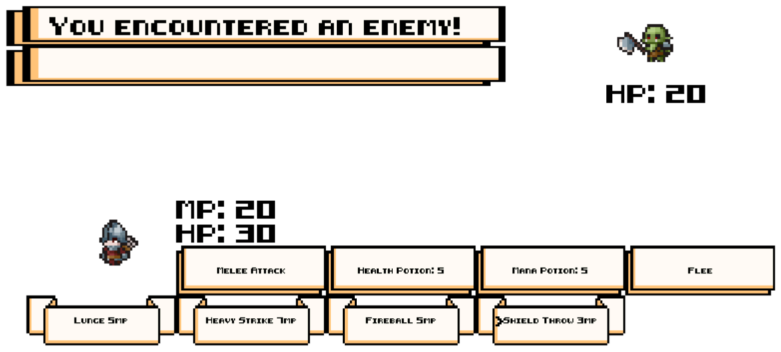


Title screen

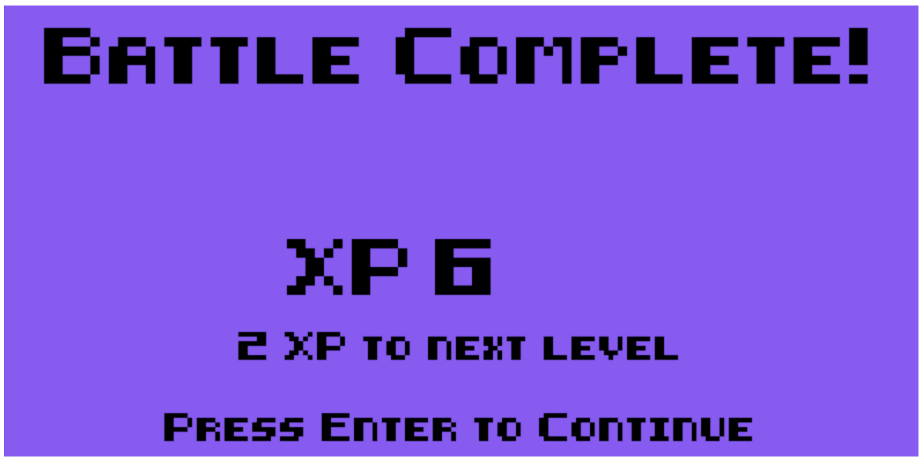


Character creation screen

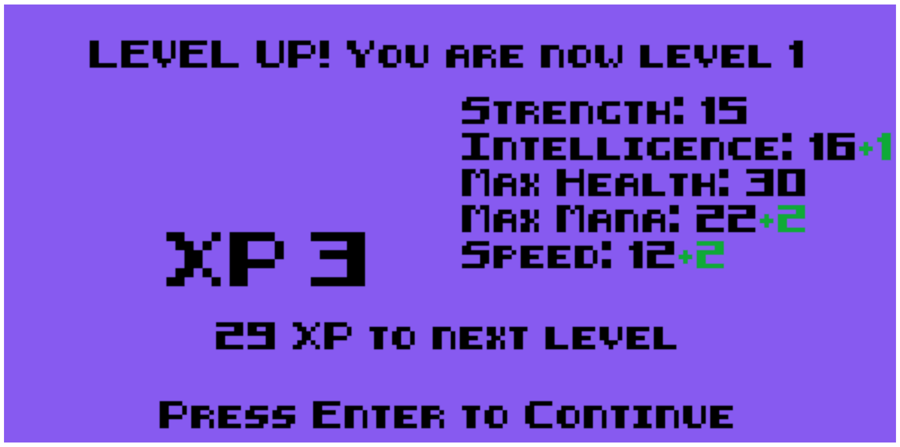

Game world



Battle screen



Battle complete screen



Level up screen

Game over screen