Binus university

Binus international

Algorithm and programming final project

RPG Game

Student Information:

Surname: Marlent **Given Name:** Steven gerald **Student ID:**2702398283

Course Code : COMP6047001 **Course Name :** Algorithm and Programming

Class : L1AC **Lecturer :** Jude Joseph Lamug Martinez, MCS

Type of Assignment : Final Project Report

Submission Pattern

Due Date : 12 January 2024

Submission Date : 12 January 2024

The assignment should meet the below requirements.

1. Assignment (hard copy) is required to be submitted on clean paper, and (soft copy) as per lecturer's instructions.
2. Soft copy assignment also requires the signed (hardcopy) submission of this form, which automatically validates the softcopy submission.
3. The above information is complete and legible.
4. Compiled pages are firmly stapled.
5. Assignment has been copied (soft copy and hard copy) for each student ahead of the submission.

Plagiarism/Cheating

BiNus International seriously regards all forms of plagiarism, cheating, and collusion as academic offences which may result in severe penalties, including loss/drop of marks, course/class discontinuity, and other possible penalties executed by the university. Please refer to the related course syllabus for further information.

Declaration of Originality

By signing this assignment, I understand, accept, and consent to Binus International terms and policy on plagiarism. Herewith I declare that the work contained in this assignment is my own work and has not been submitted for the use of assessment in another course or class, except where this has been notified and accepted in advance. Signature of Student:
Steven

A. Background

As part of this algorithm and programming course, we are expected to create a comprehensive application that uses all of the Python knowledge and skills we have acquired during the semester to solve a certain problem. We are also expected to apply concepts that are beyond our lesson in class. As a result, I decided to make an RPG game called game-game.

B. Project specification

1. Introduction

In many respects, role-playing game (RPG) development is an urgent task in today's society. Technological advancements have given developers great tools for creating visually beautiful and engrossing game experiences. Easily accessible game production platforms such as Unity and Unreal Engine have democratized game development, enabling an environment in which creators of all skill levels may realize their RPG ambitions. Aspiring developers benefit from a friendly development community, various tools, and easily accessible lessons. Creating RPGs offers for not just creative narrative, world-building, and character development, but also cross-platform publication, possible profitability, and overcoming technological obstacles. In today's setting, role-playing game creation fundamentally blends technological skill, artistic expression, and business potential, making it a profitable and promising venture for game creators.

2. Modules/Library/Framework

- **Pygame - library**
  Pygame is a free and open-source cross-platform library for the development of multimedia applications like video games using Python.

- **Sys - module**
  Sys is a part of the Python Standard Library and doesn't require installation separately. In this case, I used to exit the program.
- **Math - module**

This module can be used for a variety of purposes, such as calculating character statistics (like health points, damage), managing movement trajectories, handling collision detection, or implementing game mechanics that involve complex mathematical calculations.

- **Random - module**
  Random plays a significant role in various aspects such as generating random encounters, loot drops, enemy behavior patterns, procedural map generation, or determining critical hits and misses during combat.

3. Overview

This program is an implementation of an RPG-style game using the Pygame library in Python. It consists of multiple classes that define various game elements such as the player, enemies, blocks, ground, attacks, buttons, and the main game loop.

Program input:

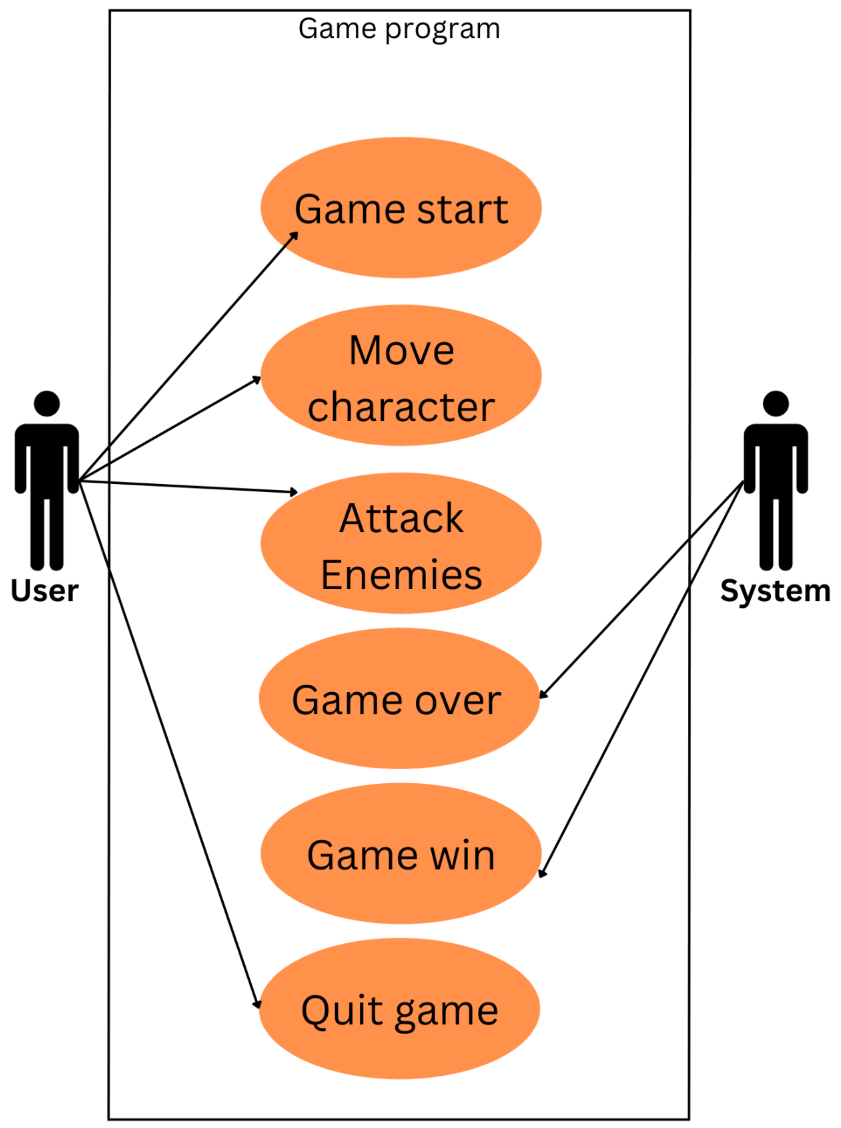
1. Mouse interaction with the button during the intro screen, game over screen, and win screen. The button used to restart or quit the game.
2. Keyboard inputs to control the player character's movement and actions. In this case, arrow keys for movement and the spacebar for attacking.

Program output:

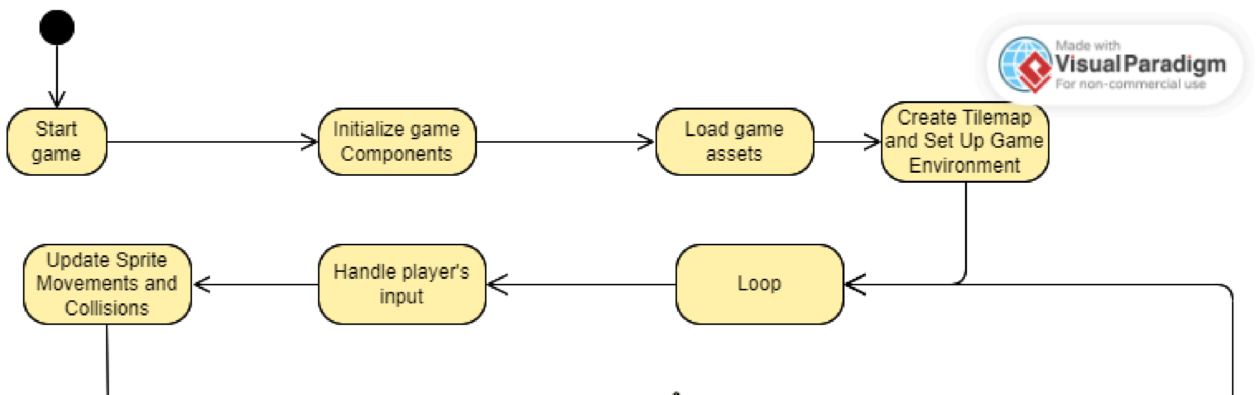
1. The game window itself, which shows the player, opponents, blocks, ground, and other features, is the primary output.
2. The Pygame font module is used by the game to create and show text components on the screen.
3. The game may switch between several states, such as the opening screen, game over screen, win screen, and main game loop.

C. Solution design

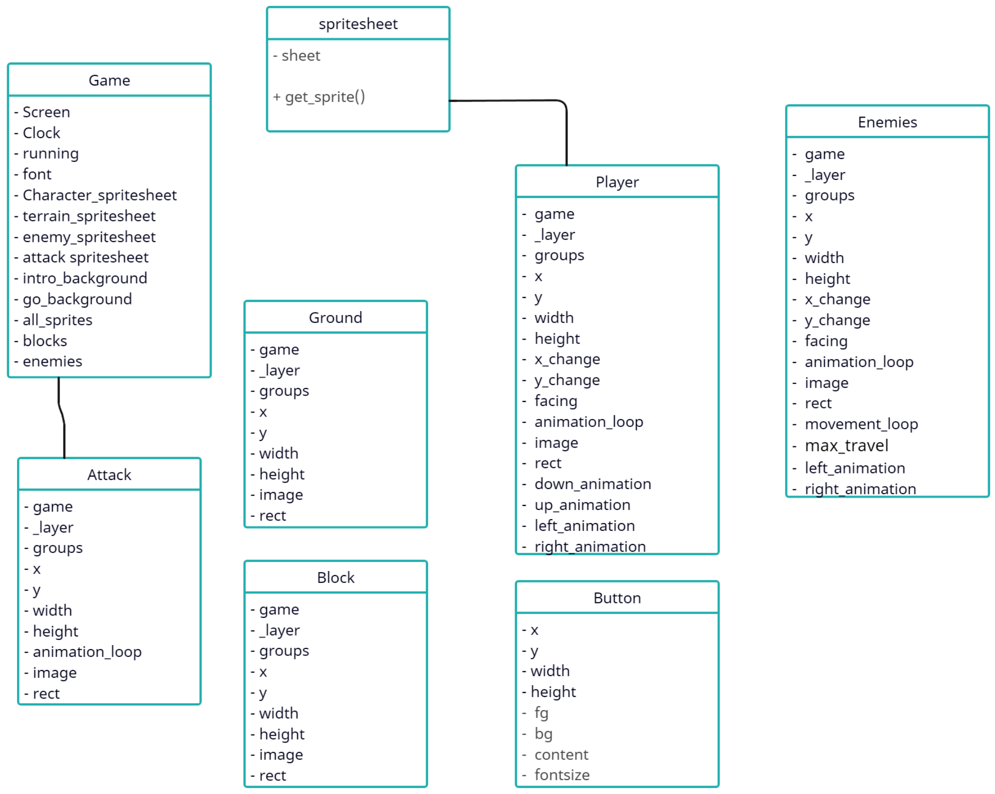
1) case diagram



2) activity diagram


Start game
Initialize game Components
Load game assets
Create Tilemap and Set Up Game Environment
Loop
Handle player's input
Update Sprite Movements and Collisions

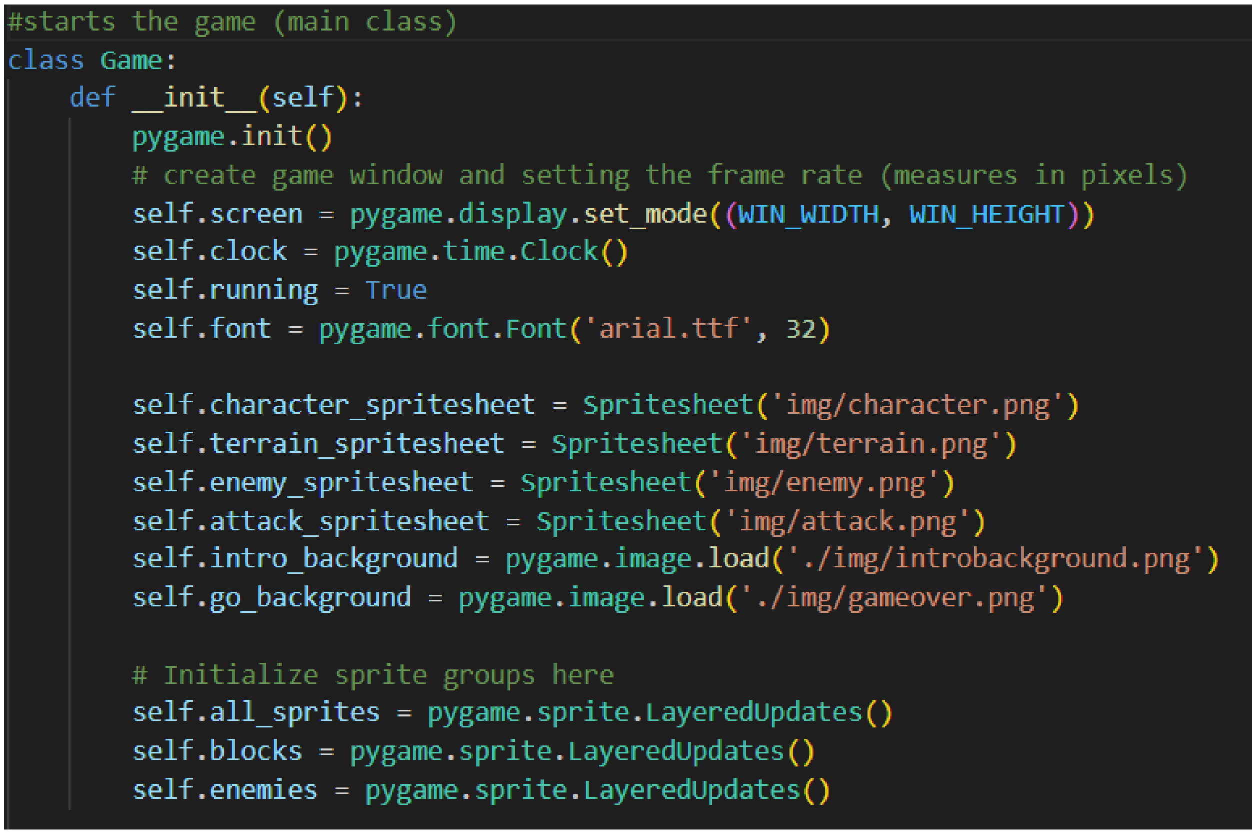
3) class diagram



D. Essential Algorithms

main.py

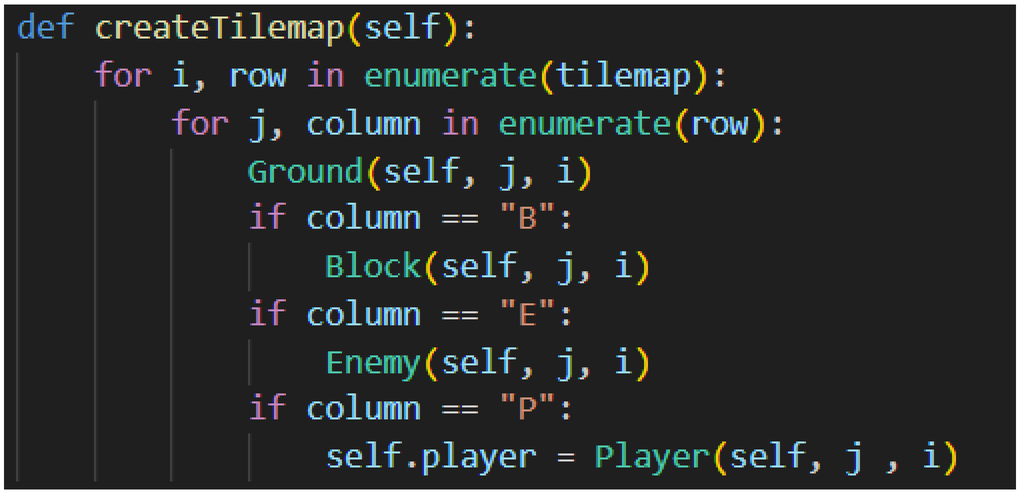
- **Game class**



```python
#starts the game (main class)
class Game:
    def __init__(self):
        pygame.init()
        # create game window and setting the frame rate (measures in pixels)
        self.screen = pygame.display.set_mode((WIN_WIDTH, WIN_HEIGHT))
        self.clock = pygame.time.Clock()
        self.running = True
        self.font = pygame.font.Font('arial.ttf', 32)

        self.character_spritesheet = Spritesheet('img/character.png')
        self.terrain_spritesheet = Spritesheet('img/terrain.png')
        self.enemy_spritesheet = Spritesheet('img/enemy.png')
        self.attack_spritesheet = Spritesheet('img/attack.png')
        self.intro_background = pygame.image.load('./img/introbackground.png')
        self.go_background = pygame.image.load('./img/gameover.png')

        # Initialize sprite groups here
        self.all_sprites = pygame.sprite.LayeredUpdates()
        self.blocks = pygame.sprite.LayeredUpdates()
        self.enemies = pygame.sprite.LayeredUpdates()
```

This game class is where the main code that starts everything that the game offers. The _init_ is the constructor for the class. pygame.init() is a function that is part of the pygame that initializes all the pygame modules needed for proper function of pygame. Then, it sets up the game window with specific size and sets up a clock for controlling the frame rate. After that it loads spritesheets and background images and finally Initializes the sprite groups.

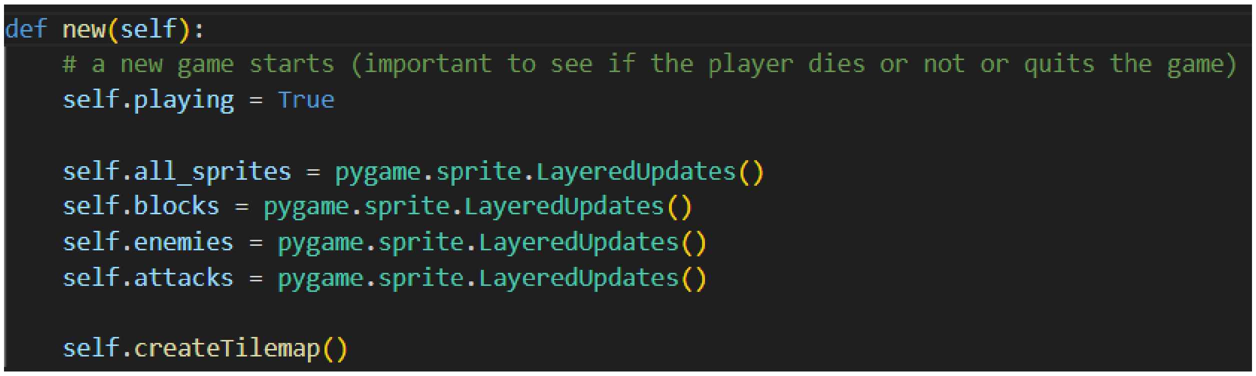
- **Game.createTilemap**



```python
def createTilemap(self):
    for i, row in enumerate(tilemap):
        for j, column in enumerate(row):
            Ground(self, j, i)
            if column == "B":
                Block(self, j, i)
            if column == "E":
                Enemy(self, j, i)
            if column == "P":
                self.player = Player(self, j , i)
```

This code defines the createTilemap method where it is responsible for initializing the game world by creating different types of game objects based on the tilemap. In this case it will iterate over the rows of the ‘tilemap’ then it will iterate over the columns within each row. Afterward it will create a Ground object at the current position (j, i) in the game world. If the character of the current position is ‘B’, it creates a Block object at the same position. It goes the same as the enemy as ‘E’ and the player as ‘P’. Additionally it assigns the ‘Player’ object to ‘self.player’ attribute to game class, so the user can control the player.

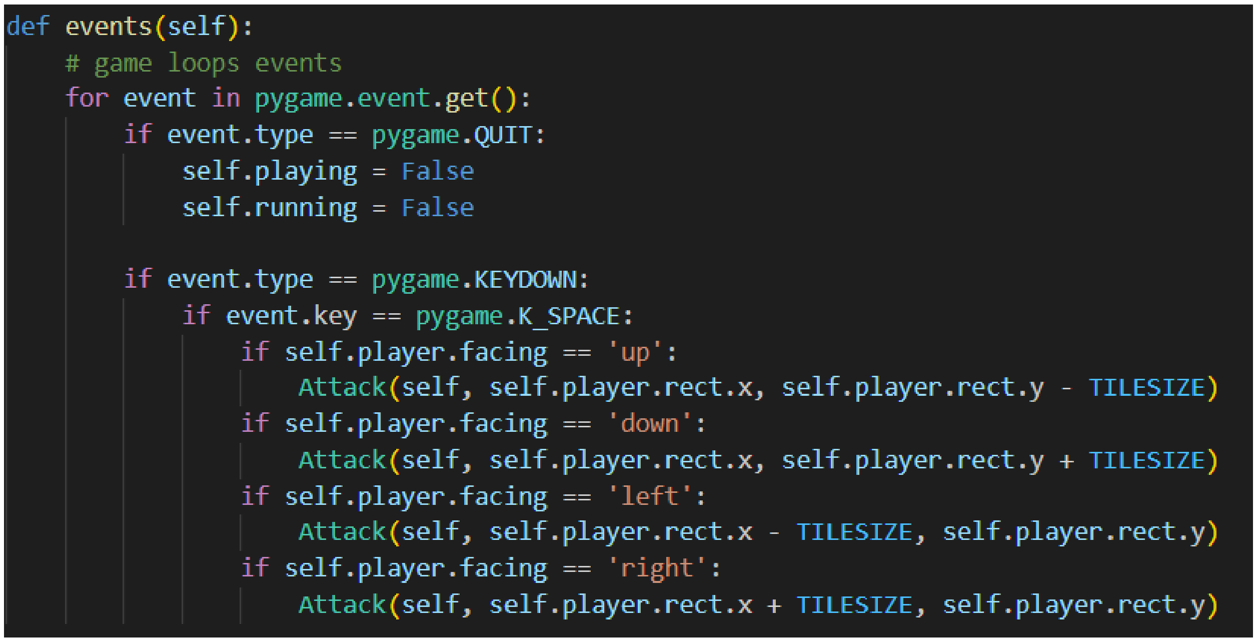
- **Game.new**



```python
def new(self):
    # a new game starts (important to see if the player dies or not or quits the game)
    self.playing = True

    self.all_sprites = pygame.sprite.LayeredUpdates()
    self.blocks = pygame.sprite.LayeredUpdates()
    self.enemies = pygame.sprite.LayeredUpdates()
    self.attacks = pygame.sprite.LayeredUpdates()

    self.createTilemap()
```

This code defines a method new where it will be called whenever the user runs the game. First it indicates that the game is in an active state, next it initializes the game world by creating different game objects, separate sprite group for blocks, sprite group for enemies, sprite group for attacks.

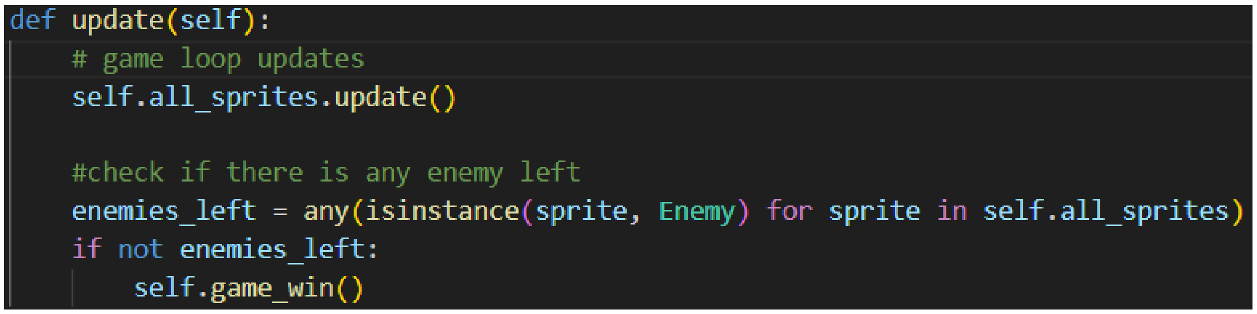
- **Game.events**



```python
def events(self):
    # game loops events
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            self.playing = False
            self.running = False

        if event.type == pygame.KEYDOWN:
            if event.key == pygame.K_SPACE:
                if self.player.facing == 'up':
                    Attack(self, self.player.rect.x, self.player.rect.y - TILESIZE)
                if self.player.facing == 'down':
                    Attack(self, self.player.rect.x, self.player.rect.y + TILESIZE)
                if self.player.facing == 'left':
                    Attack(self, self.player.rect.x - TILESIZE, self.player.rect.y)
                if self.player.facing == 'right':
                    Attack(self, self.player.rect.x + TILESIZE, self.player.rect.y)
```

This code defines a method of events where It is important to manage user input events, such as ending a game or starting an attack when a particular key is hit.
First it iterates over all the events that have occurred since the last frame. Then it checks whenever the user has quit the game or not and if that is the case it will lead to an exit from the game loop and shutting down the game. Next is the condition where the user has pressed any keys, in this case if the space keys is being pressed then it creates an Attack object at specific position, if facing up then it will create attack above the player, if down then below the player, if left then left of the player and if right then right of the player.

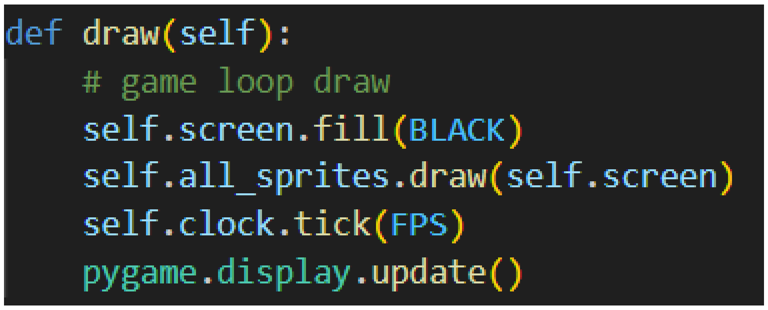
- **Game.update**



```python
def update(self):
    # game loop updates
    self.all_sprites.update()

    #check if there is any enemy left
    enemies_left = any(isinstance(sprite, Enemy) for sprite in self.all_sprites)
    if not enemies_left:
        self.game_win()
```

This code defines a method of update where it's responsible for updating the game during iteration of each. First is the method where a method of each sprite is called to perform any necessary updates. Next it will check if there are any instances of the Enemy class in the all_sprites group. Afterwards if no enemy is left in the game then it calls game_win method.

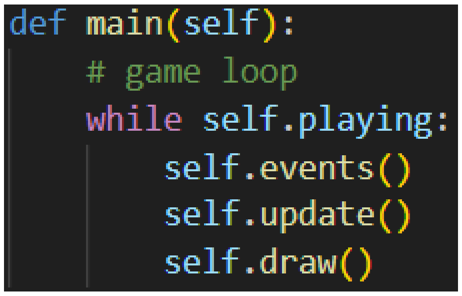
- **Game.draw**



```python
def draw(self):
    # game loop draw
    self.screen.fill(BLACK)
    self.all_sprites.draw(self.screen)
    self.clock.tick(FPS)
    pygame.display.update()
```

This code defines a method draw where it's rendering the game state on the screen during each iteration of the game loop. First it will fill the game window screen with black then by calling draw it will render all the sprites in all_sprites group in its position. Next it will limit the frame rate to specific values which is 60 frame rate per second, it ensures that the game doesn't render too quickly. Lastly it will update the display, making the changes made in the current frame visible on the screen

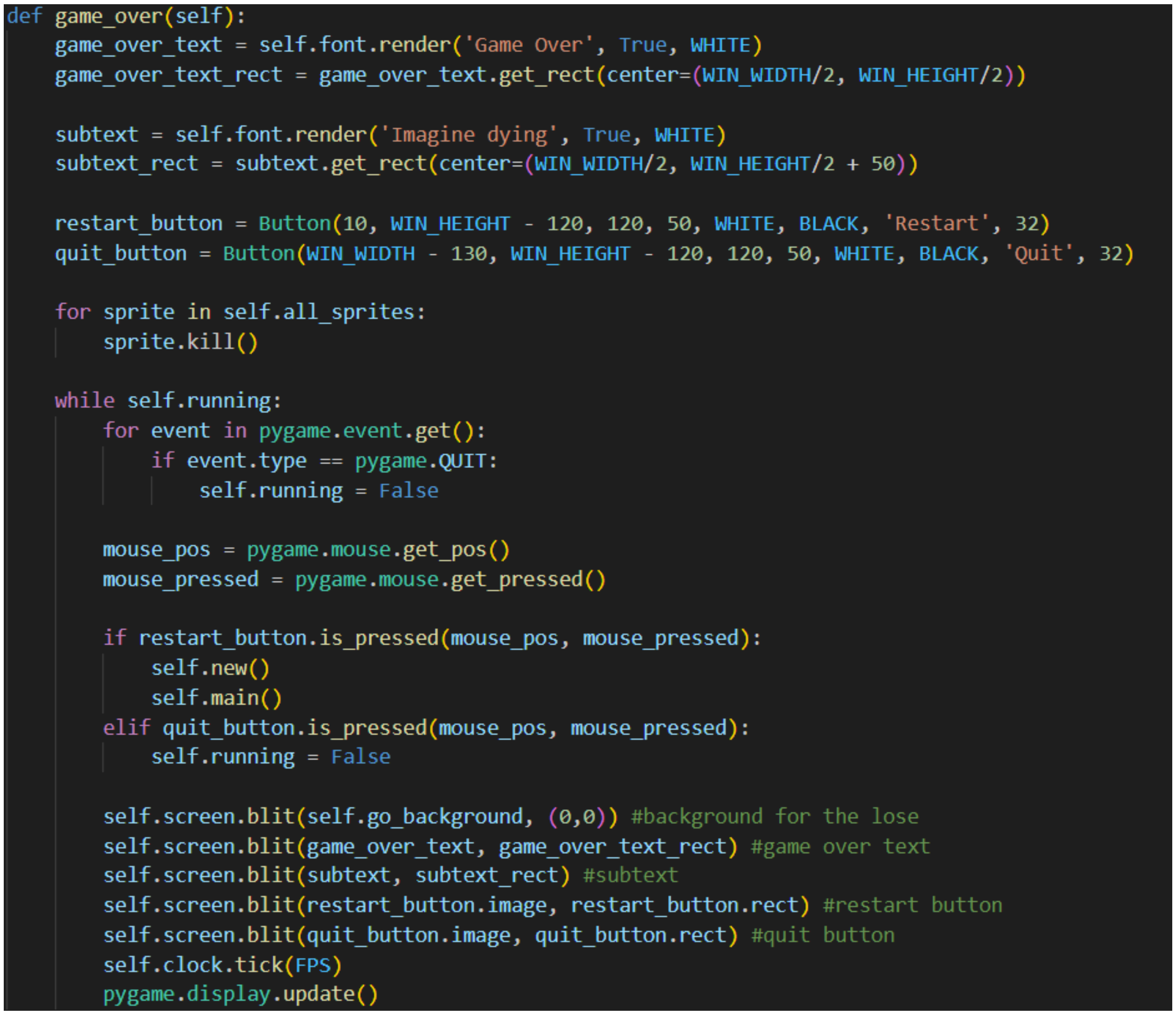
- **Game.main**



```python
def main(self):
    # game loop
    while self.playing:
        self.events()
        self.update()
        self.draw()
```

This code defines a method main which is the core logic for running the game. Firstly this will run as long as self.playing is true 'events', 'update' and 'draw' will be called. 'events' where it is expected to handle user input events during each iteration of the game loop. 'update' where it is expected to handle updating the game like moving game entities, handling collisions, etc. 'draw' is for rendering the current state of the game on the screen.

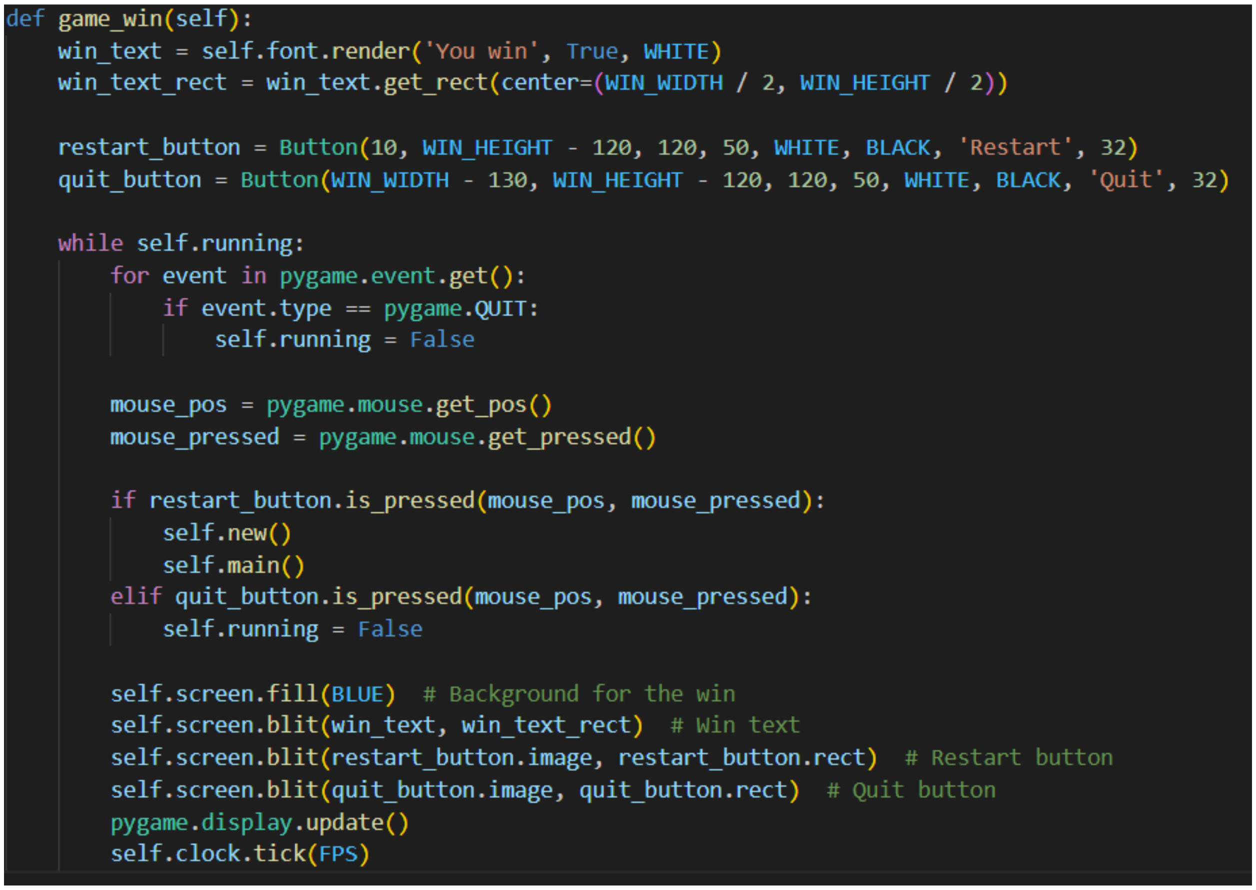
- **Game.game_over**



```python
def game_over(self):
    game_over_text = self.font.render('Game Over', True, WHITE)
    game_over_text_rect = game_over_text.get_rect(center=(WIN_WIDTH/2, WIN_HEIGHT/2))

    subtext = self.font.render('Imagine dying', True, WHITE)
    subtext_rect = subtext.get_rect(center=(WIN_WIDTH/2, WIN_HEIGHT/2 + 50))

    restart_button = Button(10, WIN_HEIGHT - 120, 120, 50, WHITE, BLACK, 'Restart', 32)
    quit_button = Button(WIN_WIDTH - 130, WIN_HEIGHT - 120, 120, 50, WHITE, BLACK, 'Quit', 32)

    for sprite in self.all_sprites:
        sprite.kill()

    while self.running:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                self.running = False

        mouse_pos = pygame.mouse.get_pos()
        mouse_pressed = pygame.mouse.get_pressed()

        if restart_button.is_pressed(mouse_pos, mouse_pressed):
            self.new()
            self.main()
        elif quit_button.is_pressed(mouse_pos, mouse_pressed):
            self.running = False

        self.screen.blit(self.go_background, (0,0)) #background for the lose
        self.screen.blit(game_over_text, game_over_text_rect) #game over text
        self.screen.blit(subtext, subtext_rect) #subtext
        self.screen.blit(restart_button.image, restart_button.rect) #restart button
        self.screen.blit(quit_button.image, quit_button.rect) #quit button
        self.clock.tick(FPS)
        pygame.display.update()
```

This code defines method game_over where this will show when the player loses. First it will render the game over text and subtext which is rendered in white color. Then it will create the quit and restart button and position it on the screen. Next it will loop iterates over all sprites in the game and kill them, effectively clearing the game state. Then loop as long as the game is running Inside the loop it checks for events, and if the player clicks the close button, it stops the game loop. Afterwards It checks if the restart or quit buttons are pressed based on the mouse position and button clicks. If the restart button is pressed, it resets the game and starts the main game loop. If the quit button is pressed, it effectively exits the game. Then it renders the background, game over text, subtext, restart button and quit button. Before updating them, it will be controlled so it doesn't render too quickly. After that it showed the updates making them visible.

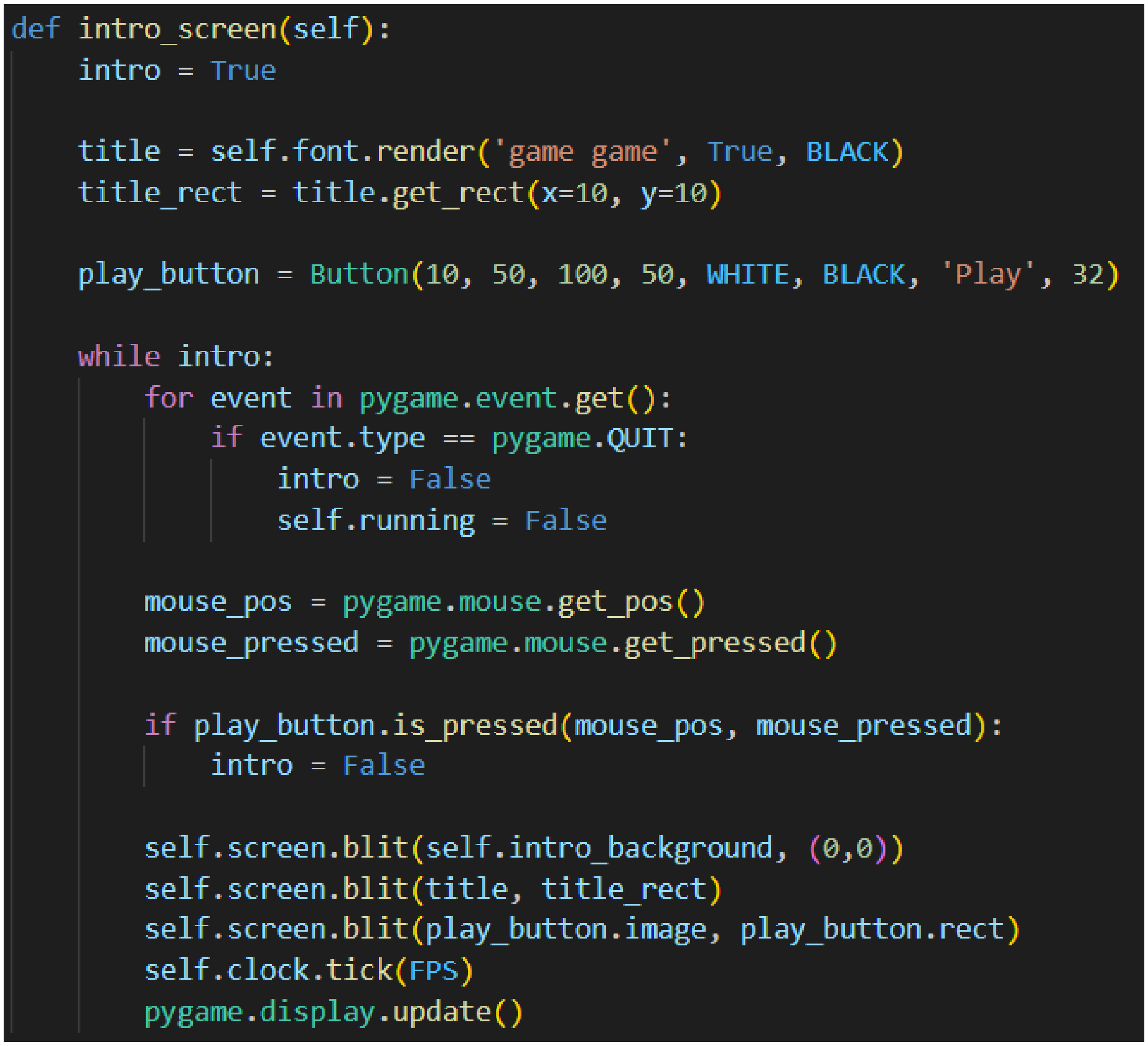
- **Game.game_win**



```python
def game_win(self):
    win_text = self.font.render('You win', True, WHITE)
    win_text_rect = win_text.get_rect(center=(WIN_WIDTH / 2, WIN_HEIGHT / 2))

    restart_button = Button(10, WIN_HEIGHT - 120, 120, 50, WHITE, BLACK, 'Restart', 32)
    quit_button = Button(WIN_WIDTH - 130, WIN_HEIGHT - 120, 120, 50, WHITE, BLACK, 'Quit', 32)

    while self.running:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                self.running = False

        mouse_pos = pygame.mouse.get_pos()
        mouse_pressed = pygame.mouse.get_pressed()

        if restart_button.is_pressed(mouse_pos, mouse_pressed):
            self.new()
            self.main()
        elif quit_button.is_pressed(mouse_pos, mouse_pressed):
            self.running = False

        self.screen.fill(BLUE)  # Background for the win
        self.screen.blit(win_text, win_text_rect)  # Win text
        self.screen.blit(restart_button.image, restart_button.rect)  # Restart button
        self.screen.blit(quit_button.image, quit_button.rect)  # Quit button
        pygame.display.update()
        self.clock.tick(FPS)
```

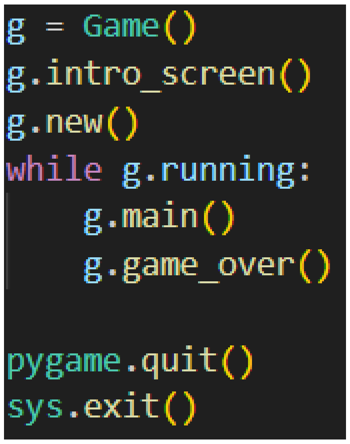
This code defines the method game_win where this will show when the player wins. This code is almost the same as game_over but the difference is that it will render you win instead of Game over, the background is blue and there is no subtext in this case.

- **Game intro_screen**



```python
def intro_screen(self):
    intro = True

    title = self.font.render('game game', True, BLACK)
    title_rect = title.get_rect(x=10, y=10)

    play_button = Button(10, 50, 100, 50, WHITE, BLACK, 'Play', 32)

    while intro:
        for event in pygame.event.get():
            if event.type == pygame.QUIT:
                intro = False
                self.running = False

        mouse_pos = pygame.mouse.get_pos()
        mouse_pressed = pygame.mouse.get_pressed()

        if play_button.is_pressed(mouse_pos, mouse_pressed):
            intro = False

        self.screen.blit(self.intro_background, (0,0))
        self.screen.blit(title, title_rect)
        self.screen.blit(play_button.image, play_button.rect)
        self.clock.tick(FPS)
        pygame.display.update()
```

This code defines the method intro_screen where it will be the welcoming screen for the players before continuing the game. First , the variable intro is set true, then it renders the title text and position according to the coordinate (10,10), then creates a button to start the game. It is positioned at (10,50) with a width of 100, height of 50, white foreground, black background, the text will be as ‘play’ and the font is 32. Afterwards, a loop continues running as long as the intro is true, then It checks for events, particularly looking for the quit event. If the user closes the window, it exits the introductory screen loop and effectively ends the game. Next It checks if the play button is pressed based on the mouse position and button clicks. If the play button is pressed, the intro screen will be closed. Then it renders the background, title text, play button. Before updating them, it will be controlled so it doesn't render too quickly. After that it showed the updates making them visible.

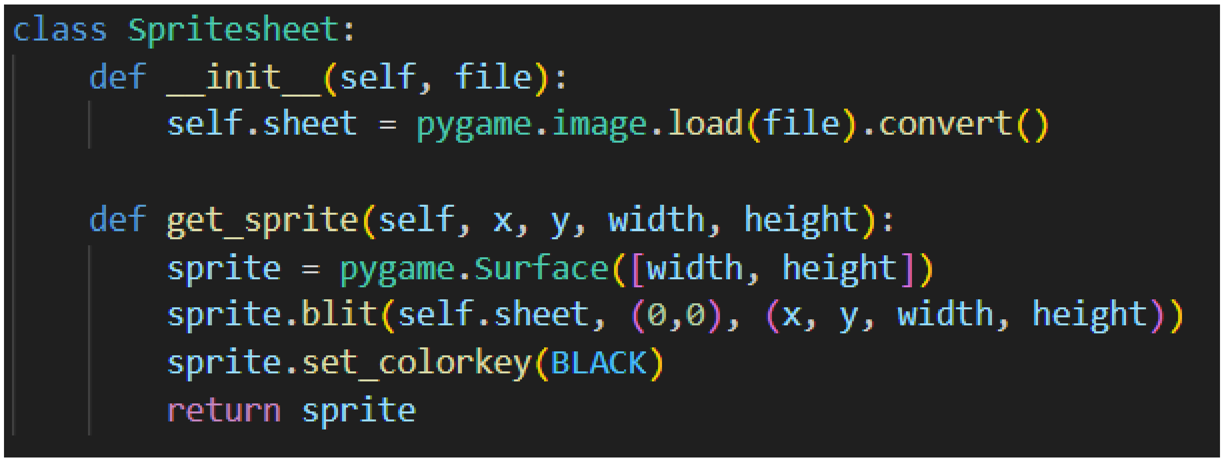


```python
g = Game()
g.intro_screen()
g.new()
while g.running:
    g.main()
    g.game_over()

pygame.quit()
sys.exit()
```

The g variable is used for calling game class, then it will display the introduction and set up a game when the user opens the game. Then a loop which is kept as a long game is running. Inside the loop there is a method that handles game events, updates the game state, and draws the screen, and another method is called when the game ends. Afterwards the line quits the pygame module and terminates the Python scripts.

sprites.py

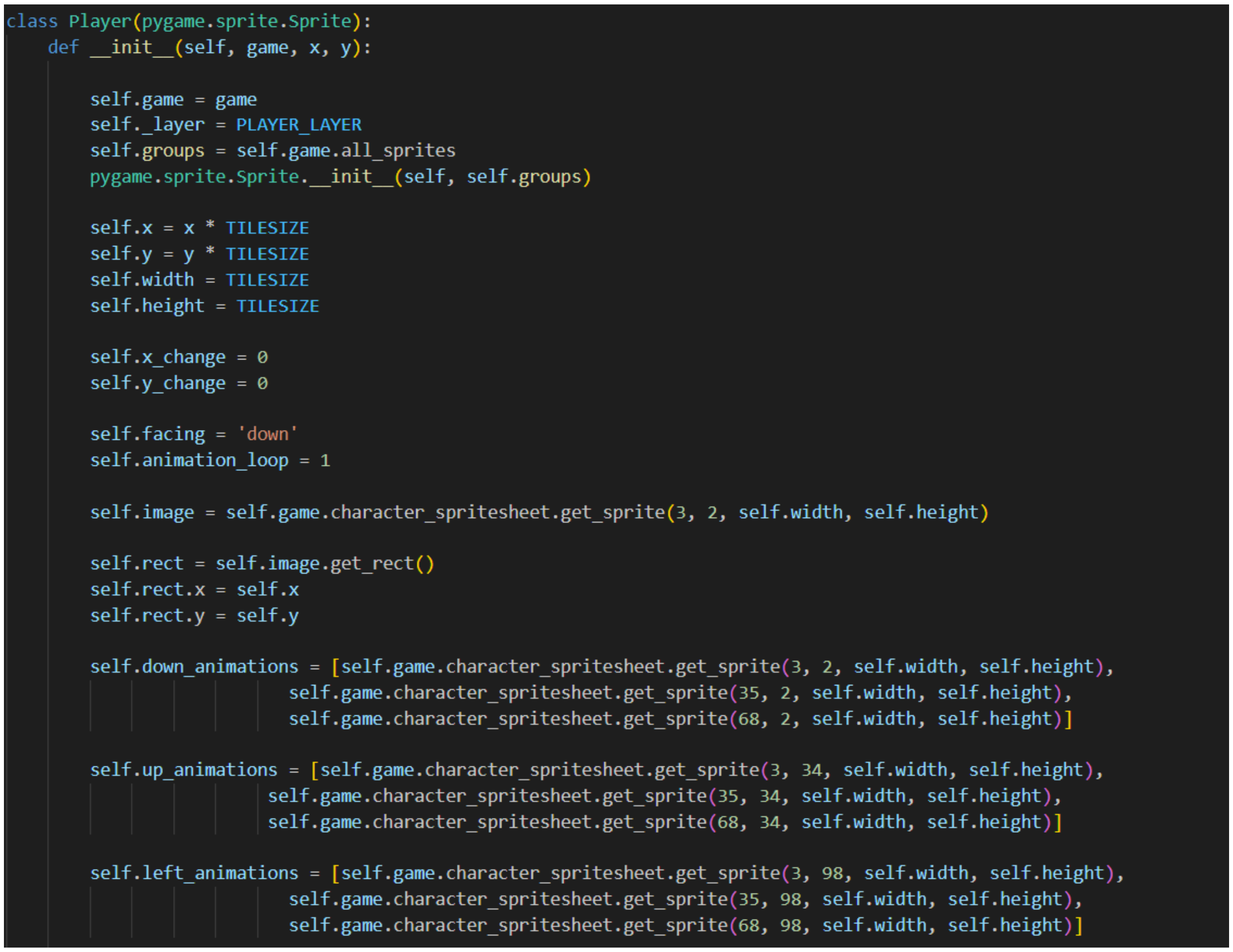
- **Spritesheet Class**



```python
class Spritesheet:
    def __init__(self, file):
        self.sheet = pygame.image.load(file).convert()

    def get_sprite(self, x, y, width, height):
        sprite = pygame.Surface([width, height])
        sprite.blit(self.sheet, (0,0), (x, y, width, height))
        sprite.set_colorkey(BLACK)
        return sprite
```

This code defines class Spritesheet which is purposed to load an image file that contains multiple sprites arranged. First It takes a single argument which is the path to the image file containing the sprites.The image is loaded and is called to ensure the image has the same pixel format as the display surface. The class has one method get_sprite extracts a sprite from the spritesheet based on the provided coordinate and dimension. Here, a new object is created with the specified dimensions. This surface is essentially a blank canvas that can be used to draw images or manipulate pixel data. The 'blit' method is used to copy a portion of the original spritesheet onto the newly created surface. Finally it set the color key of black to make it transparent. The method returns the extracted sprite.

- **Player class**



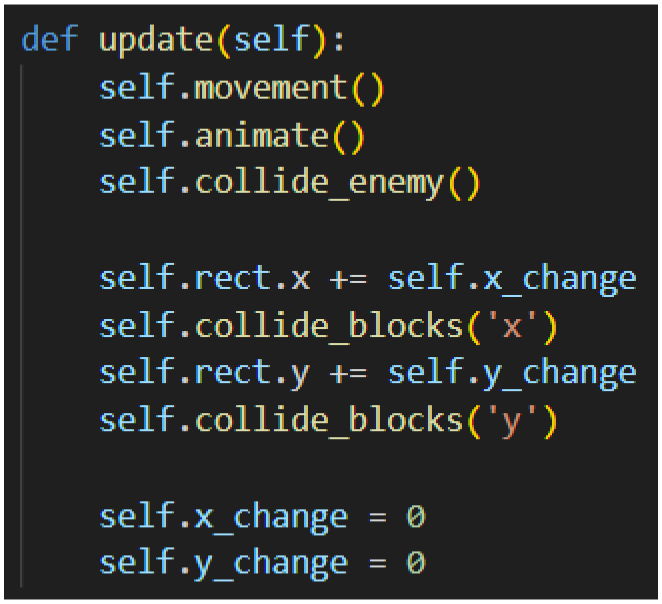
```python
class Player(pygame.sprite.Sprite):
    def __init__(self, game, x, y):

        self.game = game
        self._layer = PLAYER_LAYER
        self.groups = self.game.all_sprites
        pygame.sprite.Sprite.__init__(self, self.groups)

        self.x = x * TILESIZE
        self.y = y * TILESIZE
        self.width = TILESIZE
        self.height = TILESIZE

        self.x_change = 0
        self.y_change = 0

        self.facing = 'down'
        self.animation_loop = 1

        self.image = self.game.character_spritesheet.get_sprite(3, 2, self.width, self.height)

        self.rect = self.image.get_rect()
        self.rect.x = self.x
        self.rect.y = self.y

        self.down_animations = [self.game.character_spritesheet.get_sprite(3, 2, self.width, self.height),
                                self.game.character_spritesheet.get_sprite(35, 2, self.width, self.height),
                                self.game.character_spritesheet.get_sprite(68, 2, self.width, self.height)]

        self.up_animations = [self.game.character_spritesheet.get_sprite(3, 34, self.width, self.height),
                              self.game.character_spritesheet.get_sprite(35, 34, self.width, self.height),
                              self.game.character_spritesheet.get_sprite(68, 34, self.width, self.height)]

        self.left_animations = [self.game.character_spritesheet.get_sprite(3, 98, self.width, self.height),
                                self.game.character_spritesheet.get_sprite(35, 98, self.width, self.height),
                                self.game.character_spritesheet.get_sprite(68, 98, self.width, self.height)]
```

This code defines the class Player which represents a player character in the game. First it establishes a connection between the player and the game instance by storing the game reference in the 'self.game' attribute. This association enables the player to access game-specific functionalities and resources. Next The '_layer' attribute is utilized to determine the drawing order of sprites. Then the player is added to the 'all_sprites' group, which typically contains all the sprites present in the game. Afterwards it Calls the constructor of the superclass to set up the sprite-related functionality.

- Self.x and self.y represent the player's current position
- Self.width and self.height represent dimension of player sprite
- self.x_change, self.y_change: Used to track changes in the player's position.
- self.facing: Represents the direction the player is facing.
- self.animation_loop: Used to control the player's animation loop.

Next it loads the initial player sprite using a specific region from the character sprite. Afterward it creates a rectangular collision object based on the player's current position. Lastly a bunch of Lists of sprites for different animation directions (up, down, left, right).

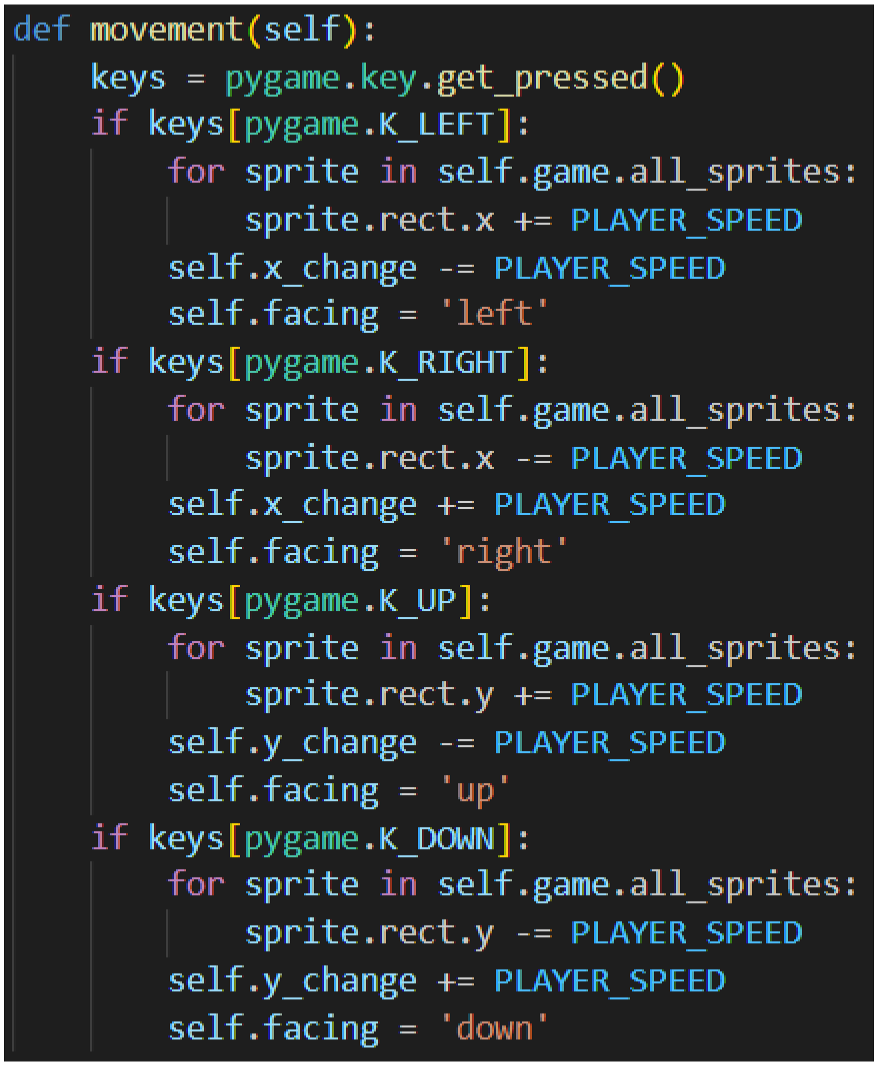
- **Player Update**



```
def update(self):
    self.movement()
    self.animate()
    self.collide_enemy()

    self.rect.x += self.x_change
    self.collide_blocks('x')
    self.rect.y += self.y_change
    self.collide_blocks('y')

    self.x_change = 0
    self.y_change = 0
```

This code defines the method update which is responsible for updating the state of the player sprite in game. Firstly the movement method is where it handles the player's movement logic. Then the animate method is where to manage the animation state of the player. The collide_enemy method is where it deals with the collision between enemy and player. Afterward The player's horizontal position is adjusted by the current horizontal change . After this adjustment, the collide_blocks method is called with the argument x. This helps prevent players from walking to solid blocks. Similar to the horizontal movement, the player's vertical position is updated by the current vertical change. The method is then called with the argument 'y', indicating a vertical collision check with blocks. Lastly x_change and y_change are set zero, indicating the player won't continue moving unless new input is received during the next update cycle.

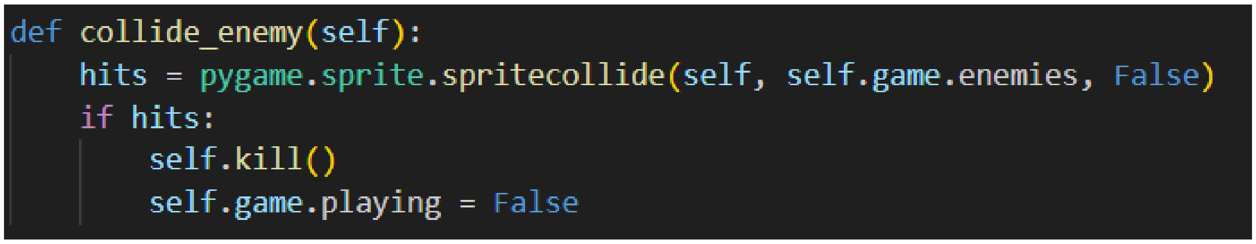
- **Player movement**



```python
def movement(self):
    keys = pygame.key.get_pressed()
    if keys[pygame.K_LEFT]:
        for sprite in self.game.all_sprites:
            sprite.rect.x += PLAYER_SPEED
        self.x_change -= PLAYER_SPEED
        self.facing = 'left'
    if keys[pygame.K_RIGHT]:
        for sprite in self.game.all_sprites:
            sprite.rect.x -= PLAYER_SPEED
        self.x_change += PLAYER_SPEED
        self.facing = 'right'
    if keys[pygame.K_UP]:
        for sprite in self.game.all_sprites:
            sprite.rect.y += PLAYER_SPEED
        self.y_change -= PLAYER_SPEED
        self.facing = 'up'
    if keys[pygame.K_DOWN]:
        for sprite in self.game.all_sprites:
            sprite.rect.y -= PLAYER_SPEED
        self.y_change += PLAYER_SPEED
        self.facing = 'down'
```

This code defines method movement which is responsible for handling player movement based on keyboard input. First it obtains the current state of all keys on the keyboard using Pygame's get_pressed method. If the left arrow key is pressed, all the player characters and all other sprites on the screen are shifted to the right, creating leftward movement. Then setting the player facing to the left. If the right arrow key is pressed, all the player characters and all other sprites on the screen are shifted to the left, creating rightward movement. Then setting the player facing to the right. If the down arrow key is pressed, all the player characters and all other sprites on the screen are shifted to upward, creating a downward movement. Then setting the player facing down. If the up arrow key is pressed, all the player characters and all other sprites on the screen are shifted to downward, creating upward movement. Then setting the player facing up.

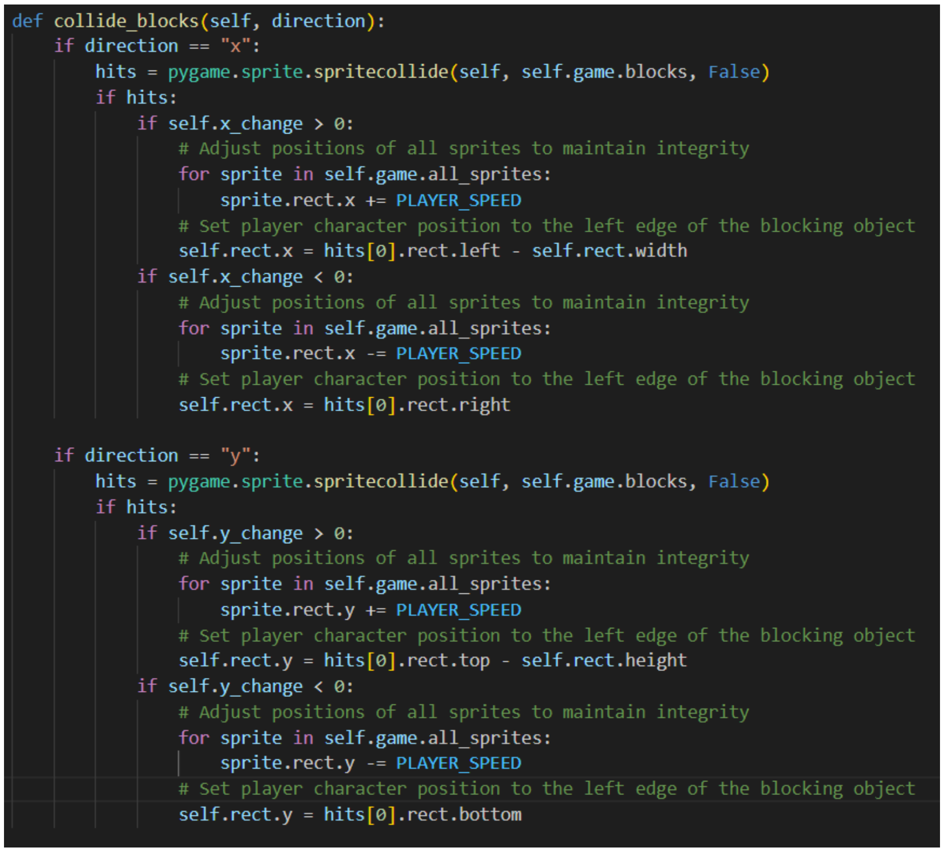
- **Player collide_enemy**



```
def collide_enemy(self):
    hits = pygame.sprite.spritecollide(self, self.game.enemies, False)
    if hits:
        self.kill()
        self.game.playing = False
```

This code defines the method collide_enemy where it is responsible for detecting collisions between the player and enemy. The spritecollide function from Pygame is employed to identify collisions between the player character and the group of enemy sprites. If it occurs then the player character is removed from all sprite groups thus the next will trigger the game over screen.

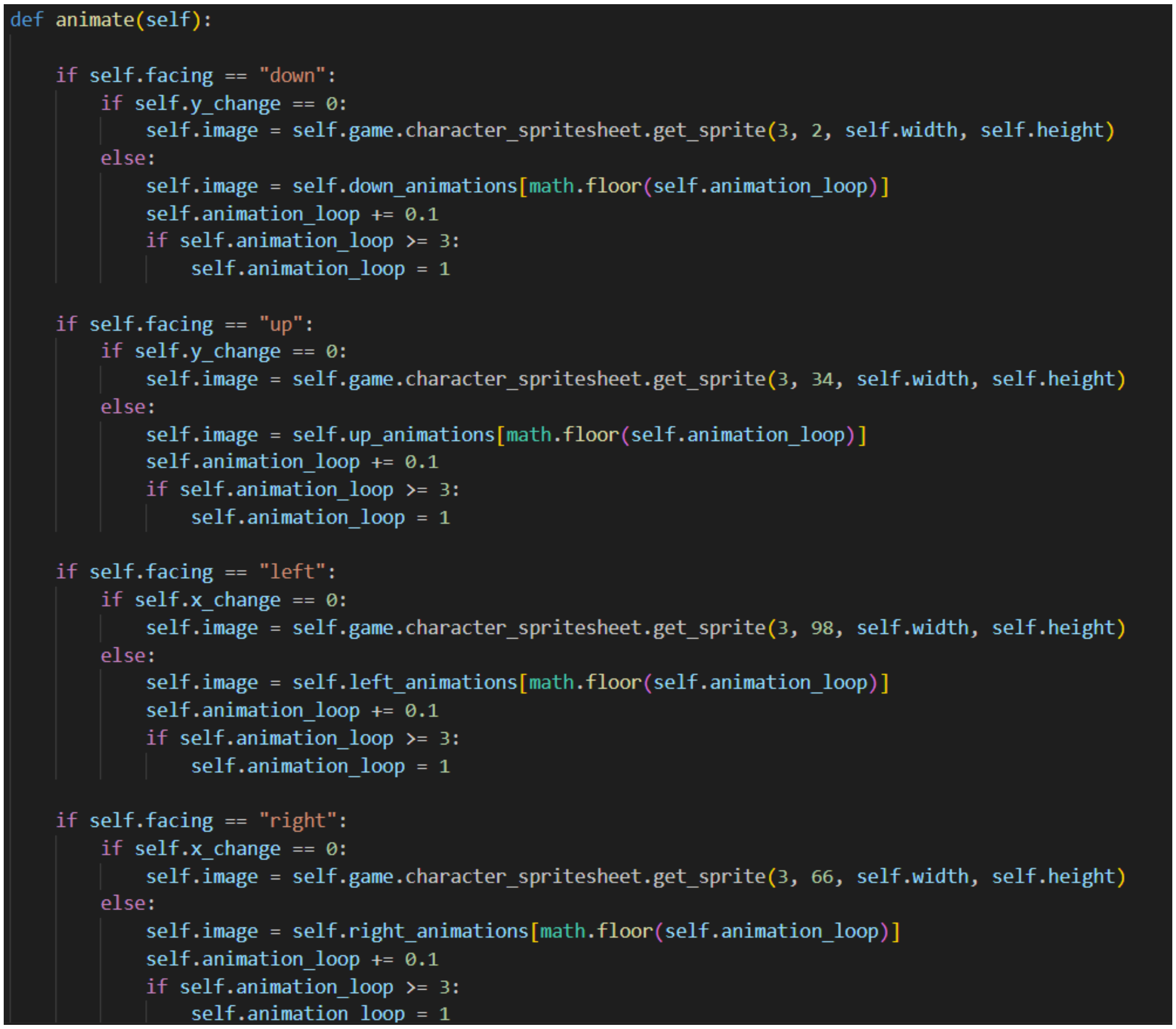
- **Player collide_blocks**



```
def collide_blocks(self, direction):
    if direction == "x":
        hits = pygame.sprite.spritecollide(self, self.game.blocks, False)
        if hits:
            if self.x_change > 0:
                # Adjust positions of all sprites to maintain integrity
                for sprite in self.game.all_sprites:
                    sprite.rect.x += PLAYER_SPEED
                # Set player character position to the left edge of the blocking object
                self.rect.x = hits[0].rect.left - self.rect.width
            if self.x_change < 0:
                # Adjust positions of all sprites to maintain integrity
                for sprite in self.game.all_sprites:
                    sprite.rect.x -= PLAYER_SPEED
                # Set player character position to the left edge of the blocking object
                self.rect.x = hits[0].rect.right

    if direction == "y":
        hits = pygame.sprite.spritecollide(self, self.game.blocks, False)
        if hits:
            if self.y_change > 0:
                # Adjust positions of all sprites to maintain integrity
                for sprite in self.game.all_sprites:
                    sprite.rect.y += PLAYER_SPEED
                # Set player character position to the left edge of the blocking object
                self.rect.y = hits[0].rect.top - self.rect.height
            if self.y_change < 0:
                # Adjust positions of all sprites to maintain integrity
                for sprite in self.game.all_sprites:
                    sprite.rect.y -= PLAYER_SPEED
                # Set player character position to the left edge of the blocking object
                self.rect.y = hits[0].rect.bottom
```

This code defines the method collide_blocks which is responsible for managing the collision between player and the block. The spritecollide function from Pygame is employed to identify collisions between the player character and the group of enemy sprites. If a collision is detected along the X-axis, the method checks the direction of movement. If moving right, adjust the positions of all sprites to prevent overlap and place the player character to the left of the blocking object. If moving left, adjust positions and place the player character to the right of the blocking object. If a collision is detected along the Y-axis, the method checks the direction of movement.If moving down, adjust positions of all sprites and place the player character above the blocking object.If moving up, adjust positions and place the player character below the blocking object.

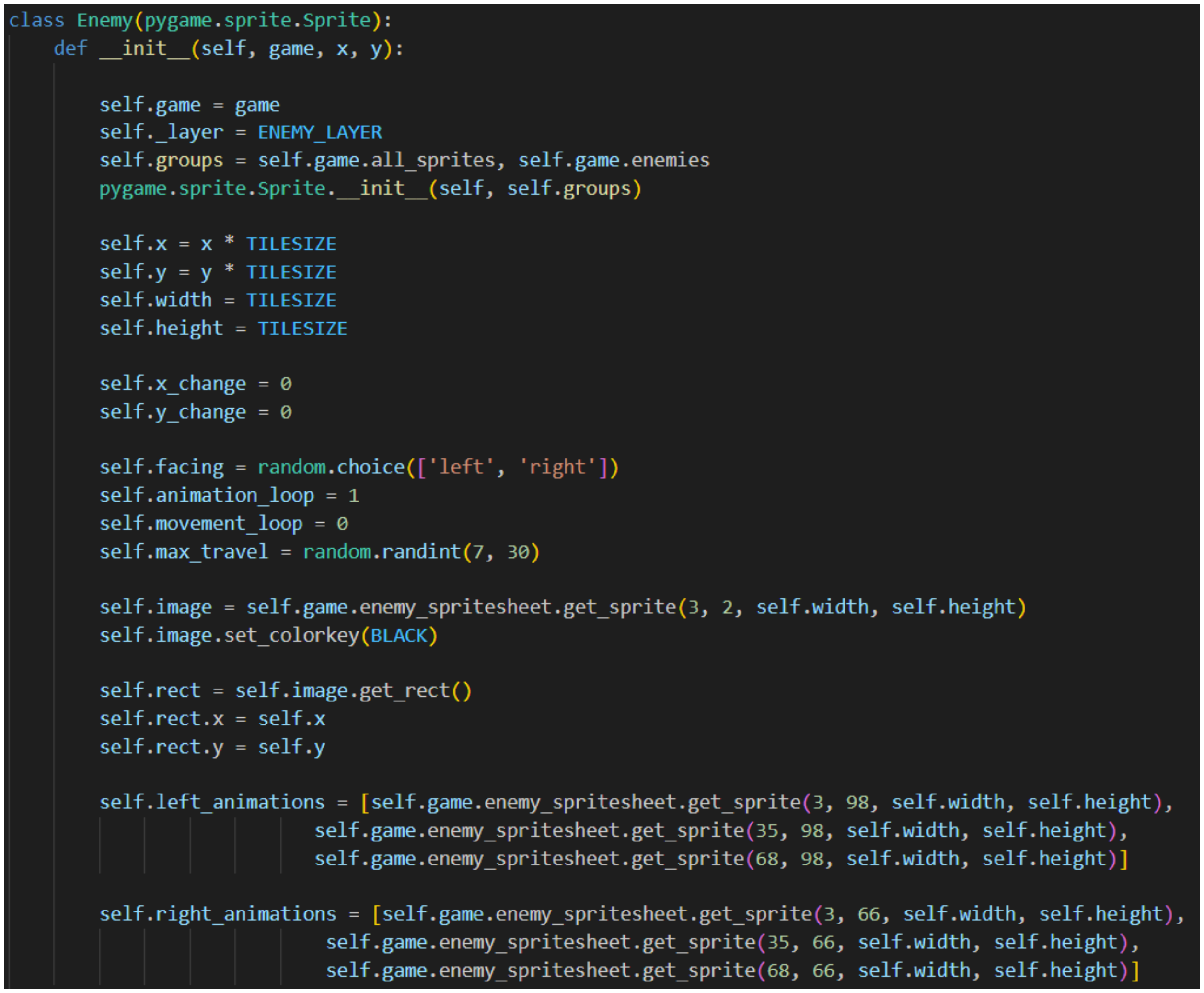
- **Player animate**



```
def animate(self):

    if self.facing == "down":
        if self.y_change == 0:
            self.image = self.game.character_spritesheet.get_sprite(3, 2, self.width, self.height)
        else:
            self.image = self.down_animations[math.floor(self.animation_loop)]
            self.animation_loop += 0.1
            if self.animation_loop >= 3:
                self.animation_loop = 1

    if self.facing == "up":
        if self.y_change == 0:
            self.image = self.game.character_spritesheet.get_sprite(3, 34, self.width, self.height)
        else:
            self.image = self.up_animations[math.floor(self.animation_loop)]
            self.animation_loop += 0.1
            if self.animation_loop >= 3:
                self.animation_loop = 1

    if self.facing == "left":
        if self.x_change == 0:
            self.image = self.game.character_spritesheet.get_sprite(3, 98, self.width, self.height)
        else:
            self.image = self.left_animations[math.floor(self.animation_loop)]
            self.animation_loop += 0.1
            if self.animation_loop >= 3:
                self.animation_loop = 1

    if self.facing == "right":
        if self.x_change == 0:
            self.image = self.game.character_spritesheet.get_sprite(3, 66, self.width, self.height)
        else:
            self.image = self.right_animations[math.floor(self.animation_loop)]
            self.animation_loop += 0.1
            if self.animation_loop >= 3:
                self.animation_loop = 1
```

This code defines an animation method that updates the player character and sprite based on its direction and movement. When facing down, if the player is not moving vertically, set the default down-facing sprite. If moving down, select an animation frame from the down-facing animation sequence based on the animation loop. The loop advances gradually, and when reaching the end, it resets to the beginning. When facing up, similar logic is applied. If not moving vertically, set the default up-facing sprite; otherwise, select the appropriate animation frame. The logic for left and right have a similar pattern, The default left-facing and right-facing sprites are set when the player is not moving horizontally.If moving left or right, the corresponding animation frame is selected based on the animation loop, and the loop is reset when reaching the end.

- **Enemy class**



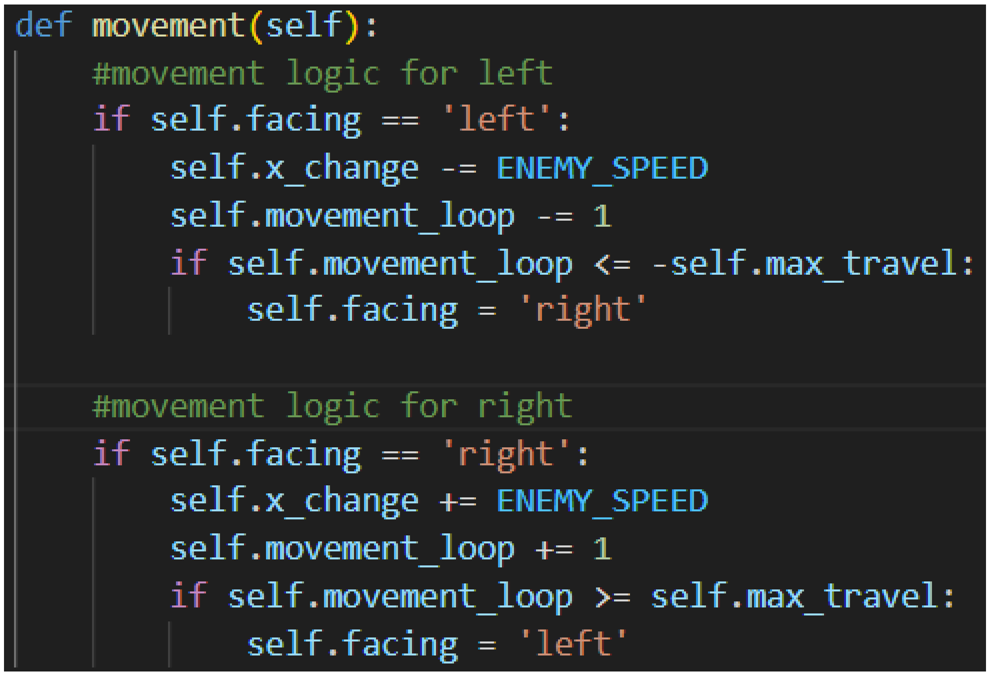
```
class Enemy(pygame.sprite.Sprite):
    def __init__(self, game, x, y):

        self.game = game
        self._layer = ENEMY_LAYER
        self.groups = self.game.all_sprites, self.game.enemies
        pygame.sprite.Sprite.__init__(self, self.groups)

        self.x = x * TILESIZE
        self.y = y * TILESIZE
        self.width = TILESIZE
        self.height = TILESIZE

        self.x_change = 0
        self.y_change = 0

        self.facing = random.choice(['left', 'right'])
        self.animation_loop = 1
        self.movement_loop = 0
        self.max_travel = random.randint(7, 30)

        self.image = self.game.enemy_spritesheet.get_sprite(3, 2, self.width, self.height)
        self.image.set_colorkey(BLACK)

        self.rect = self.image.get_rect()
        self.rect.x = self.x
        self.rect.y = self.y

        self.left_animations = [self.game.enemy_spritesheet.get_sprite(3, 98, self.width, self.height),
                                self.game.enemy_spritesheet.get_sprite(35, 98, self.width, self.height),
                                self.game.enemy_spritesheet.get_sprite(68, 98, self.width, self.height)]

        self.right_animations = [self.game.enemy_spritesheet.get_sprite(3, 66, self.width, self.height),
                                 self.game.enemy_spritesheet.get_sprite(35, 66, self.width, self.height),
                                 self.game.enemy_spritesheet.get_sprite(68, 66, self.width, self.height)]
```

This code defines the enemy method where it defines the attribute and behaviour of the enemy character. First, The Enemy class is a subclass of pygame.sprite.Sprite, allowing it to be part of sprite groups in the game and it sets the colorkey Black for transparency. The enemy randomly starts facing either left or right. The initialization of animation_loop is 1, representing the current animation frame.

- x_change and y_change represent the change in position for movement.
- movement_loop tracks the current movement phase.
- max_travel is a randomly determined maximum distance the enemy will travel in one direction before changing.

Lastly , a bunch of Lists of animation frames are created for left and right movement.These frames will be cycled through during the enemy's animation loop.

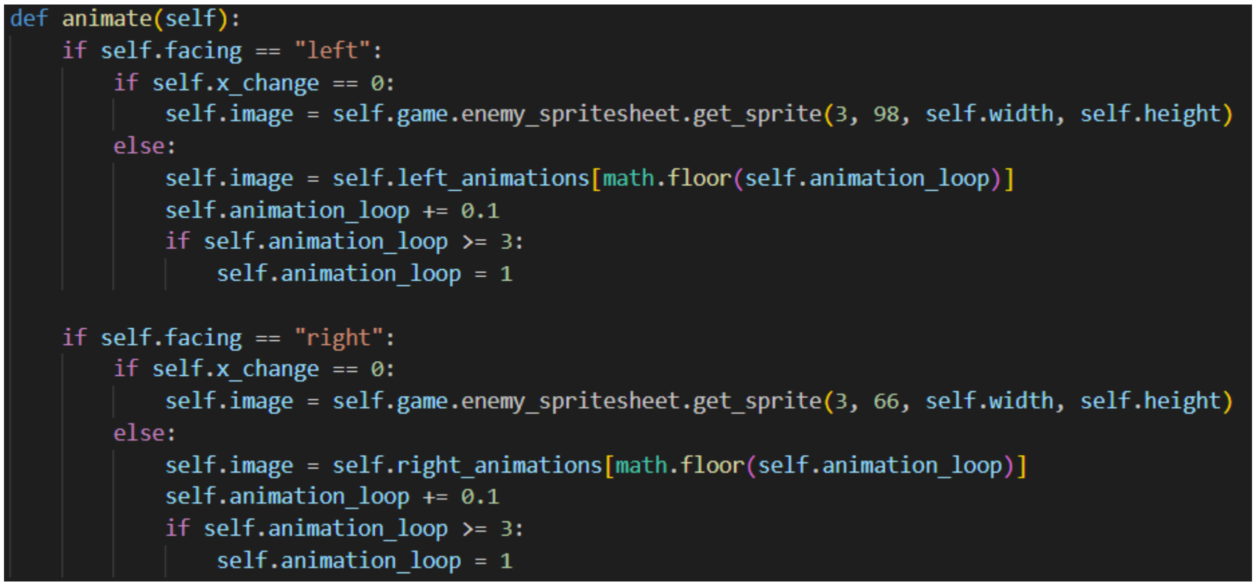
- **Enemy movement**



```python
def movement(self):
    #movement logic for left
    if self.facing == 'left':
        self.x_change -= ENEMY_SPEED
        self.movement_loop -= 1
        if self.movement_loop <= -self.max_travel:
            self.facing = 'right'

    #movement logic for right
    if self.facing == 'right':
        self.x_change += ENEMY_SPEED
        self.movement_loop += 1
        if self.movement_loop >= self.max_travel:
            self.facing = 'left'
```

This code defines a movement method which is responsible for enemy movement in the game. First the method uses a conditional statement to differentiate between left and right facing direction for the enemy. Then if the enemy facing to the right x_change will decrease leading to the left then keeping track of the distance traveled in the left direction. If the enemy reaches maximum distance it will change the direction to the right. The logic for right movement is similar to left movement, but instead of decreasing in x_change it will increase leading to the right and lastly if the maximum distance is exceeded it will change the direction to the left.

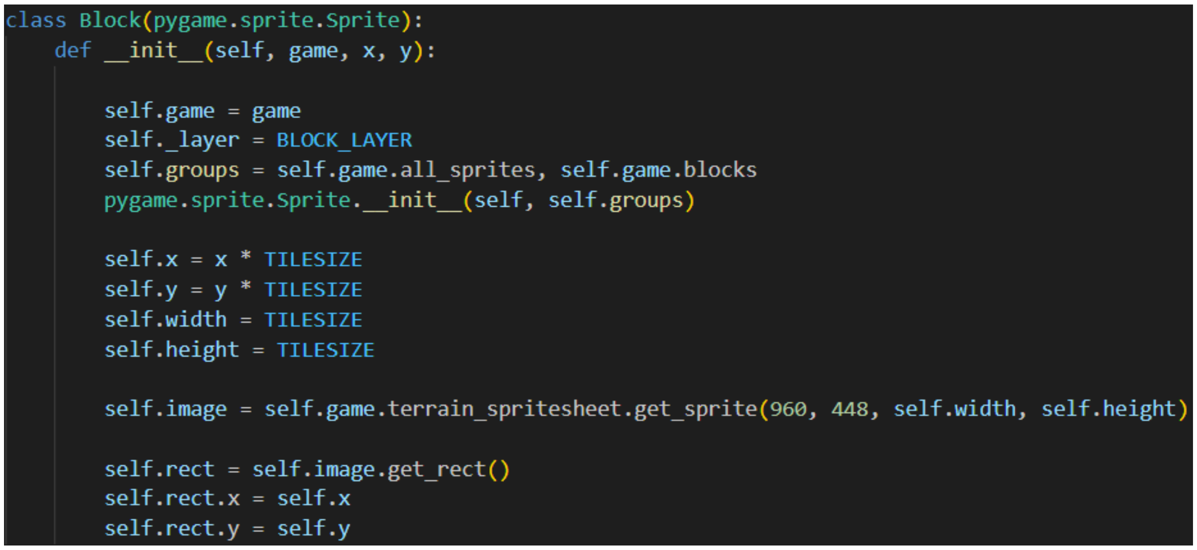
- **Enemy animate**



```python
def animate(self):
    if self.facing == "left":
        if self.x_change == 0:
            self.image = self.game.enemy_spritesheet.get_sprite(3, 98, self.width, self.height)
        else:
            self.image = self.left_animations[math.floor(self.animation_loop)]
            self.animation_loop += 0.1
            if self.animation_loop >= 3:
                self.animation_loop = 1

    if self.facing == "right":
        if self.x_change == 0:
            self.image = self.game.enemy_spritesheet.get_sprite(3, 66, self.width, self.height)
        else:
            self.image = self.right_animations[math.floor(self.animation_loop)]
            self.animation_loop += 0.1
            if self.animation_loop >= 3:
                self.animation_loop = 1
```

This code defines an animation method that updates the enemy character and sprite based on its direction and movement. When facing left, if the enemy is not moving horizontal, set the default left-facing sprite. If moving left, it selects an animated sprite from the left_animations list. Then the animation_loop is increased, creating a smooth effect and if it exceeds more than 3, it will reset to 1 making a continuous animation loop. When facing right, similar logic is applied. If the enemy is moving, it will choose an animated sprite from the right_animation list.

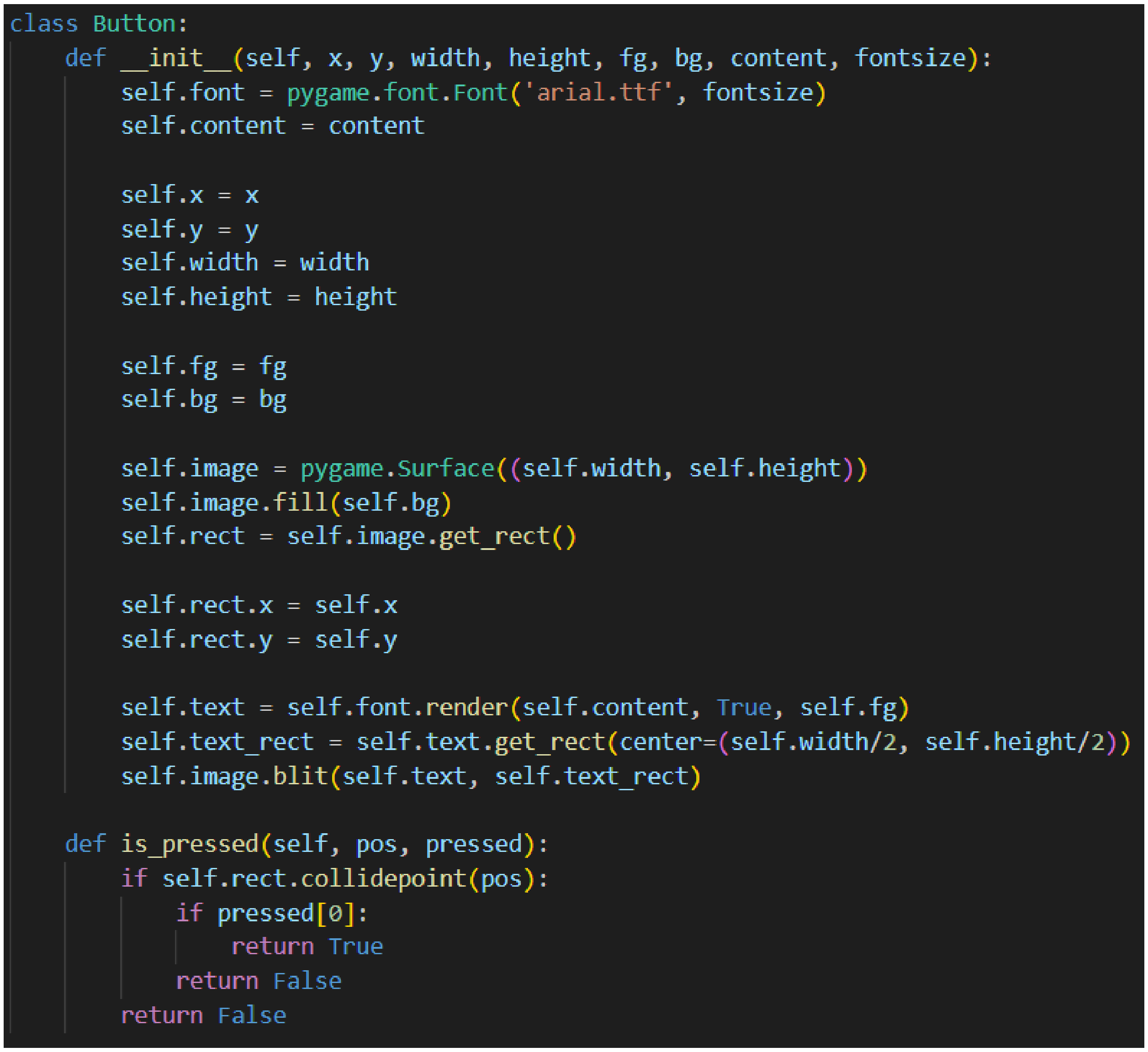
- **Block Class**



```python
class Block(pygame.sprite.Sprite):
    def __init__(self, game, x, y):

        self.game = game
        self._layer = BLOCK_LAYER
        self.groups = self.game.all_sprites, self.game.blocks
        pygame.sprite.Sprite.__init__(self, self.groups)

        self.x = x * TILESIZE
        self.y = y * TILESIZE
        self.width = TILESIZE
        self.height = TILESIZE

        self.image = self.game.terrain_spritesheet.get_sprite(960, 448, self.width, self.height)

        self.rect = self.image.get_rect()
        self.rect.x = self.x
        self.rect.y = self.y
```

This code defines Block class where it initializes the block sprite. First it determines the layering order of the sprite, then defines the sprite groups where the blocks belong, next initialized with these groups, incorporating the block into the sprite system. Then obtain a specific sprite from the spritesheet for the block, get_sprite is a method where defining the sprite's position on the spritesheet and its dimensions. Get_rect is a method where it calls on the sprite's image a rectangular collision area, then positioned with specific coordinates.

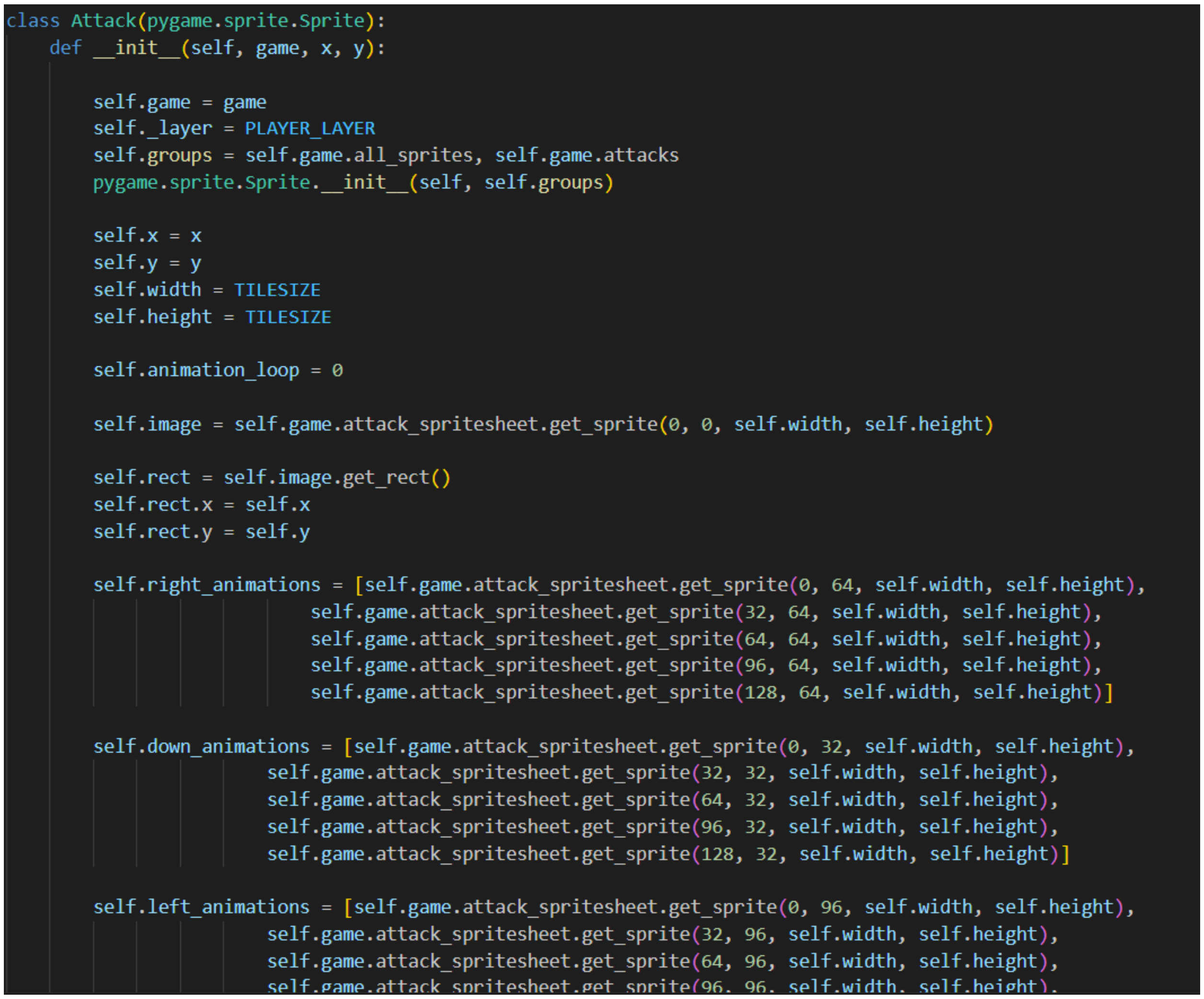
- **Button class**



```
class Button:
    def __init__(self, x, y, width, height, fg, bg, content, fontsize):
        self.font = pygame.font.Font('arial.ttf', fontsize)
        self.content = content

        self.x = x
        self.y = y
        self.width = width
        self.height = height

        self.fg = fg
        self.bg = bg

        self.image = pygame.Surface((self.width, self.height))
        self.image.fill(self.bg)
        self.rect = self.image.get_rect()

        self.rect.x = self.x
        self.rect.y = self.y

        self.text = self.font.render(self.content, True, self.fg)
        self.text_rect = self.text.get_rect(center=(self.width/2, self.height/2))
        self.image.blit(self.text, self.text_rect)

    def is_pressed(self, pos, pressed):
        if self.rect.collidepoint(pos):
            if pressed[0]:
                return True
            return False
        return False
```

This code defines button class, inside there are two methods __init__ and is_pressed method, the __init__ method is where it deals with fonts, text content on button, coordinates of the button, background of the button. The rect attribute defines a rectangular area for collision detection and text_rect attribute holds the rectangular area for the rendered text, centered within the button. Then the rendered text is blitted onto the button's image at the specified position. The is_pressed method checks if a given position is within the button's rectangular area. If the left mouse button is clicked then it returns true otherwise false.

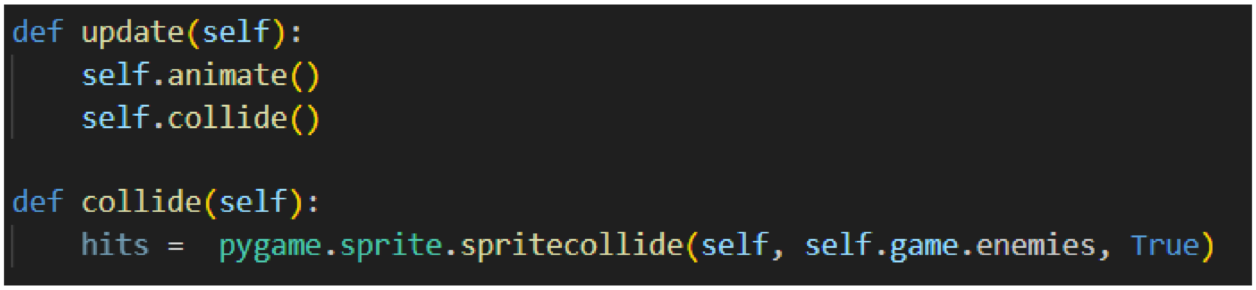
- **Attack class**



```python
class Attack(pygame.sprite.Sprite):
    def __init__(self, game, x, y):

        self.game = game
        self._layer = PLAYER_LAYER
        self.groups = self.game.all_sprites, self.game.attacks
        pygame.sprite.Sprite.__init__(self, self.groups)

        self.x = x
        self.y = y
        self.width = TILESIZE
        self.height = TILESIZE

        self.animation_loop = 0

        self.image = self.game.attack_spritesheet.get_sprite(0, 0, self.width, self.height)

        self.rect = self.image.get_rect()
        self.rect.x = self.x
        self.rect.y = self.y

        self.right_animations = [self.game.attack_spritesheet.get_sprite(0, 64, self.width, self.height),
                                 self.game.attack_spritesheet.get_sprite(32, 64, self.width, self.height),
                                 self.game.attack_spritesheet.get_sprite(64, 64, self.width, self.height),
                                 self.game.attack_spritesheet.get_sprite(96, 64, self.width, self.height),
                                 self.game.attack_spritesheet.get_sprite(128, 64, self.width, self.height)]

        self.down_animations = [self.game.attack_spritesheet.get_sprite(0, 32, self.width, self.height),
                             self.game.attack_spritesheet.get_sprite(32, 32, self.width, self.height),
                             self.game.attack_spritesheet.get_sprite(64, 32, self.width, self.height),
                             self.game.attack_spritesheet.get_sprite(96, 32, self.width, self.height),
                             self.game.attack_spritesheet.get_sprite(128, 32, self.width, self.height)]

        self.left_animations = [self.game.attack_spritesheet.get_sprite(0, 96, self.width, self.height),
                             self.game.attack_spritesheet.get_sprite(32, 96, self.width, self.height),
                             self.game.attack_spritesheet.get_sprite(64, 96, self.width, self.height),
                             self.game.attack_spritesheet.get_sprite(96, 96, self.width, self.height),
```

This code defines an attack method where it is responsible for attacks that are performed in game. In this case __init__ method is used for the constructor of the class. First it defines that it stores the instance in game, then determines the layer in which the sprite belongs, and defines the sprite groups to which the sprite belongs. X and y belong to coordinate meanwhile width and height belong to dimension, animation_loop used for cycle through different frames of the attack animation. Image attribute is the set of initial frames of the attack animation obtained from the attack spritesheet. rect attribute defines a rectangular area for collision detection and positioning. Lastly a bunch of lists containing frames of the attack animation in different directions.

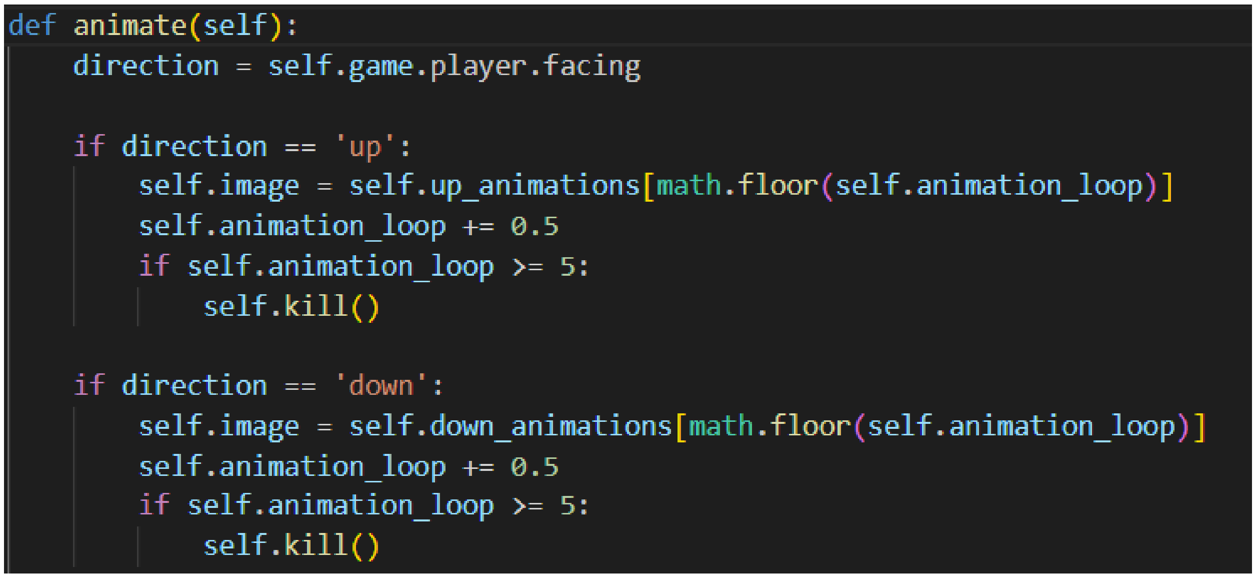
- **Attack update and collide**



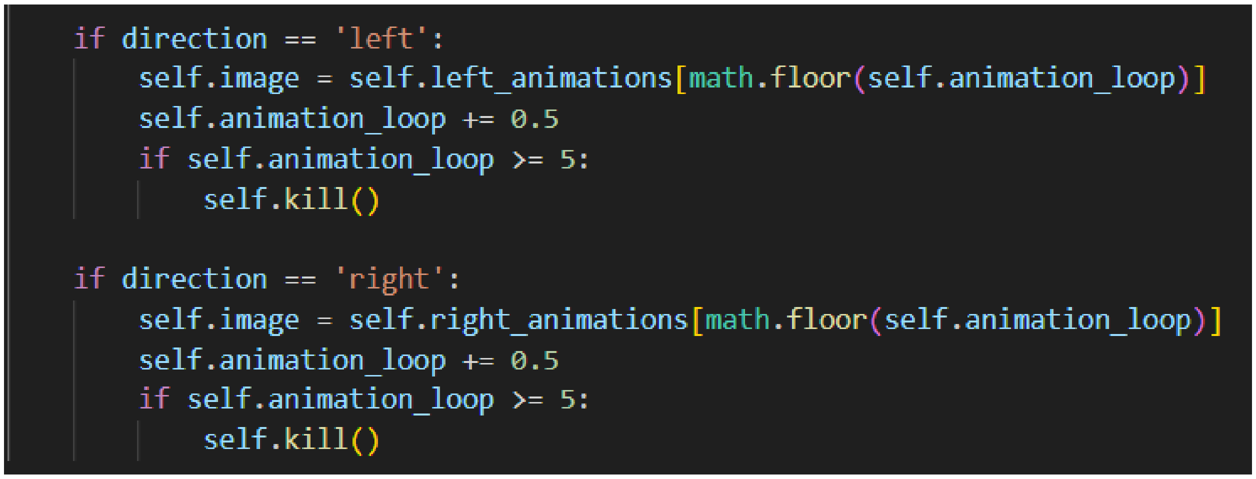
```python
def update(self):
    self.animate()
    self.collide()

def collide(self):
    hits =  pygame.sprite.spritecollide(self, self.game.enemies, True)
```

This code defines the update and collide method. Where update responsible for updating the state of entity during game loop and collide method is responsible for collision between enemy and the attack. As for update just handling animation and managing collide. Meanwhile collide, it defines spritecollide where it detects collide between a sprite and a group of sprites. The first parameter is for attack animation to check the collision, the second parameter is for enemies to check the collision, and the third parameter if there is collision then the enemies will be removed.

- **Attack animate**



```python
def animate(self):
    direction = self.game.player.facing

    if direction == 'up':
        self.image = self.up_animations[math.floor(self.animation_loop)]
        self.animation_loop += 0.5
        if self.animation_loop >= 5:
            self.kill()

    if direction == 'down':
        self.image = self.down_animations[math.floor(self.animation_loop)]
        self.animation_loop += 0.5
        if self.animation_loop >= 5:
            self.kill()
```

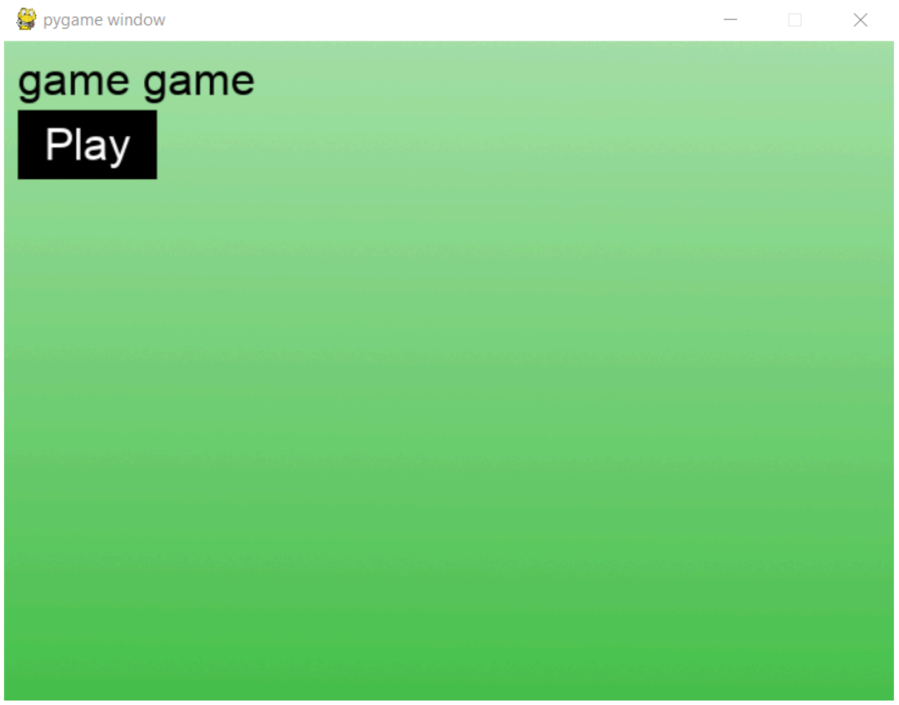


```
if direction == 'left':
    self.image = self.left_animations[math.floor(self.animation_loop)]
    self.animation_loop += 0.5
    if self.animation_loop >= 5:
        self.kill()

if direction == 'right':
    self.image = self.right_animations[math.floor(self.animation_loop)]
    self.animation_loop += 0.5
    if self.animation_loop >= 5:
        self.kill()
```

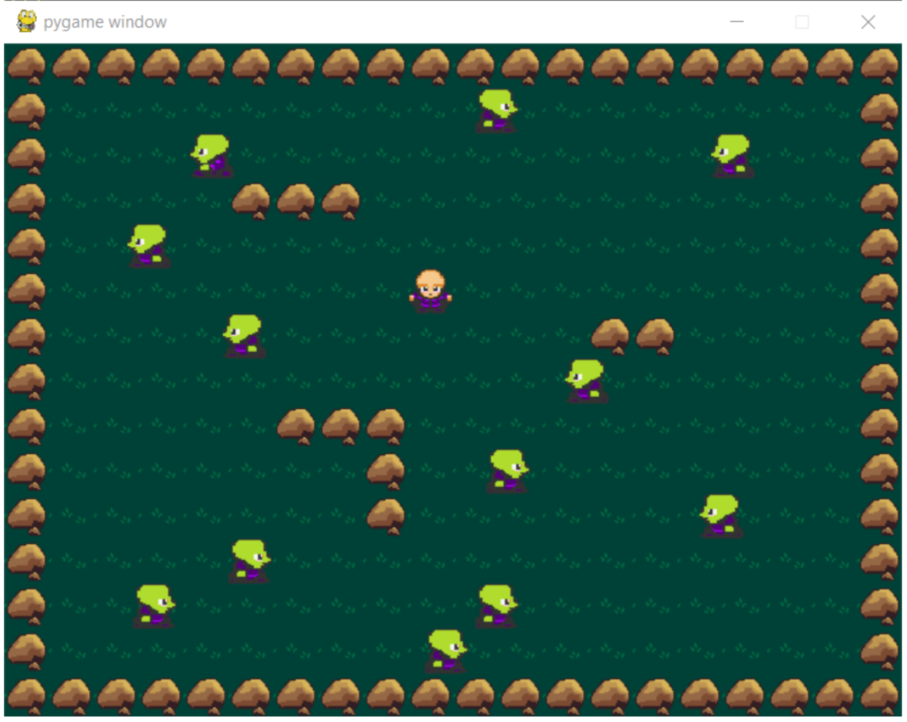
This code defines the animate method where it handles the attack animation to be the same that the player is facing. First it determines the animation frames based on the direction and updates the entity's image accordingly. The animation is performed using an increase in self.animation_loop. Next the specific animation is set up and math.floor(self.animation_loop) is used to select the appropriate frame based on the animation loop value at that moment, self.animation_loop is used to controlling the animation speed and if it reaches 5, the attack sprite is removed. Similar logic for left, right, and down,each with its corresponding animation frames and loop handling.

E. Evidence of the program (screenshots)

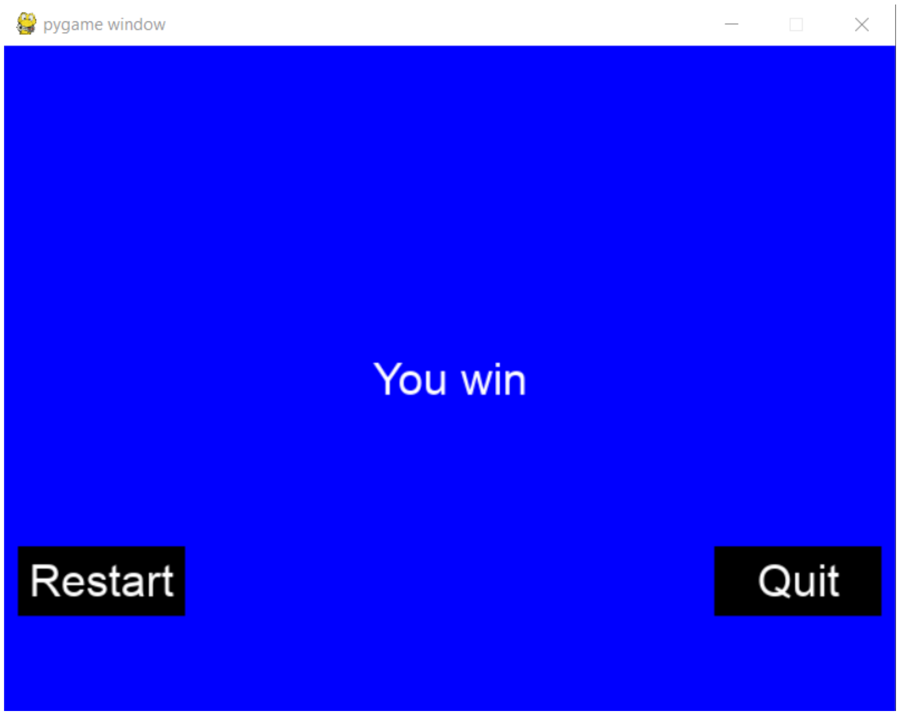
Intro screen



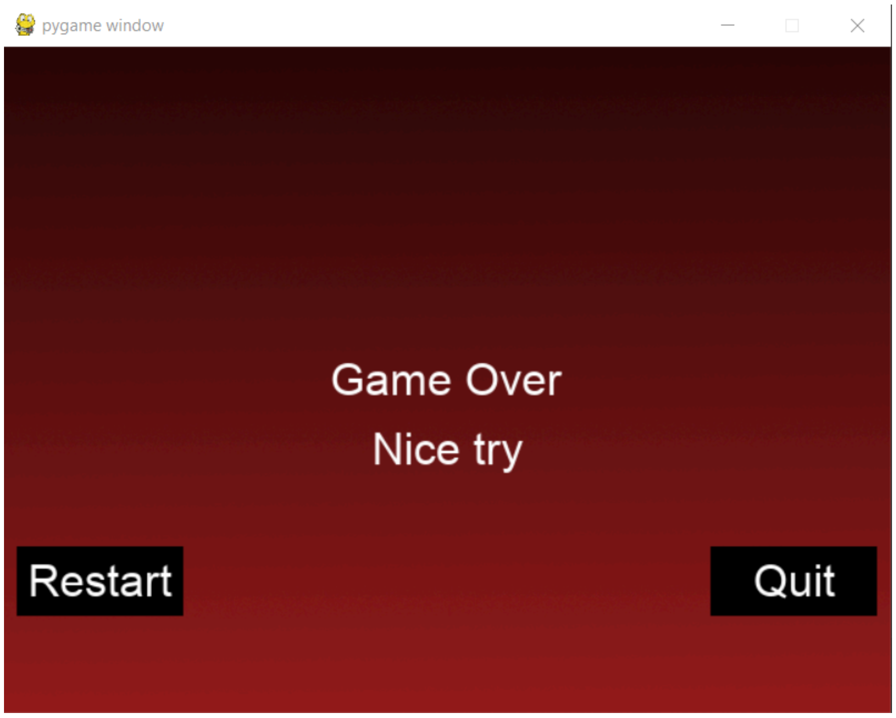
Main game



Game win



Game over


pygame window
Game Over
Nice try
Restart
Quit