Flappy Bird AI Code Explanation

This Python code implements a Flappy Bird game where a population of birds, controlled by neural networks, learns to play the game using a genetic algorithm.

Python

```
import pygame
import random
import numpy as np
from pygame.locals import *
```

These lines **import necessary libraries**:

- pygame: For game development (graphics, sounds, events).
- random: To generate random numbers, used for pipe heights and mutations.
- numpy: For numerical operations, especially for the neural network calculations.
- pygame.locals: Imports common Pygame constants like QUIT, KEYDOWN, etc.

<!-- end list -->

Python

```
# Initialize pygame
pygame.init()
```

pygame.init() **initializes all the Pygame modules** required to run the game.

Python

```
# Game constants
SCREEN_WIDTH = 400
SCREEN_HEIGHT = 600
GRAVITY = 0.25
FLAP_STRENGTH = -5
PIPE_WIDTH = 50
PIPE_GAP = 150
PIPE_FREQUENCY = 1500  # milliseconds
FONT = pygame.font.SysFont('Arial', 20)
```

These lines define **global constants** for the game:

- SCREEN_WIDTH, SCREEN_HEIGHT: Dimensions of the game window.
- GRAVITY: How fast the bird falls.
- FLAP_STRENGTH: The upward velocity applied when the bird flaps.
- PIPE_WIDTH, PIPE_GAP: Dimensions of the pipes.
- PIPE_FREQUENCY: How often new pipes appear (in milliseconds).
- FONT: The font used for displaying game statistics.

<!-- end list -->

Python

```
# Colors
WHITE = (255, 255, 255)
BLACK = (0, 0, 0)
GREEN = (0, 255, 0)
RED = (255, 0, 0)
BLUE = (0, 0, 255)
SKY_BLUE = (135, 206, 235)
YELLOW = (255, 255, 0)
```

These lines define **RGB color tuples** used for drawing various game elements.

Python

```
class Bird:
    def __init__(self, x, y, brain=None):
        self.x = x
        self.y = y
        self.velocity = 0
        self.radius = 15
        self.alive = True
        self.fitness = 0
        self.score = 0
```

```
        self.pipes_passed = 0
```

The Bird class represents each bird in the game:

- __init__: The constructor initializes a bird's properties:
    - x, y: Position on the screen.
    - velocity: Current vertical speed.
    - radius: Size of the bird for drawing and collision.
    - alive: Boolean indicating if the bird is still in play.
    - fitness: A measure of how well the bird performed (used in natural selection).
    - score: General score, usually increasing with time.
    - pipes_passed: Number of pipes successfully navigated.

<!-- end list -->

Python

```
        # Neural network (5 inputs, 8 hidden, 1 output)
        if brain is None:
            self.brain = NeuralNetwork(5, 8, 1)
        else:
            self.brain = brain.copy()
            self.brain.mutate(0.1)
```

This part of the Bird constructor handles the **neural network (brain)** for each bird:

- If brain is None (for new birds in the first generation), a new NeuralNetwork is created.
- Otherwise (for birds in subsequent generations), it copies an existing brain (from a "parent") and then mutates it slightly, introducing variation.

<!-- end list -->

Python

```
    def flap(self):
        self.velocity = FLAP_STRENGTH
```

flap(): Sets the bird's **vertical velocity** to a negative value, making it move upward.

Python

```
    def update(self):
        self.velocity += GRAVITY
        self.y += self.velocity

        if self.y >= SCREEN_HEIGHT - self.radius or self.y <= self.radius:
            self.alive = False
```

update(): **Updates the bird's position and status**:

- self.velocity += GRAVITY: Applies gravity to the bird's velocity.
- self.y += self.velocity: Updates the bird's vertical position based on its velocity.
- The if condition checks if the bird has hit the top or bottom boundaries of the screen, setting self.alive to False if it has.

<!-- end list -->

Python

```
    def think(self, pipes):
        if not self.alive:
            return

        next_pipe = None
        for pipe in pipes:
            if pipe.x + PIPE_WIDTH > self.x - 20:
                next_pipe = pipe
                break
```

think(): This is where the bird's **neural network makes decisions**:

- It first checks if the bird is alive.
- It then iterates through the pipes to **find the next_pipe** the bird is approaching. The -20 offset is to consider pipes that are just off-screen to the left.

<!-- end list -->

Python

```
        if next_pipe:
```

```
        inputs = np.array([
            self.y / SCREEN_HEIGHT,
            (self.velocity + 10) / 20,
            next_pipe.top_height / SCREEN_HEIGHT,
            (next_pipe.x - self.x) / SCREEN_WIDTH,
            (self.y - (next_pipe.top_height + PIPE_GAP/2)) / SCREEN_HEIGHT
        ])
```

If a next_pipe is found, it prepares the **inputs for the neural network**:

- self.y / SCREEN_HEIGHT: Bird's current height, normalized.
- (self.velocity + 10) / 20: Bird's vertical velocity, normalized to a 0-1 range (assuming velocity ranges from -10 to 10 after normalization constant adjustments based on typical game behavior).
- next_pipe.top_height / SCREEN_HEIGHT: Height of the top pipe, normalized.
- (next_pipe.x - self.x) / SCREEN_WIDTH: Horizontal distance to the next pipe, normalized.
- (self.y - (next_pipe.top_height + PIPE_GAP/2)) / SCREEN_HEIGHT: Vertical distance from the bird to the center of the pipe gap, normalized.

<!-- end list -->

Python

```
        output = self.brain.predict(inputs)


        if output[0] > 0.5:
            self.flap()
```

- output = self.brain.predict(inputs): The neural network processes the inputs to produce an output.
- if output[0] > 0.5: If the output (which is a value between 0 and 1 due to the sigmoid activation in the output layer) is greater than 0.5, the bird flaps(). This acts as a simple binary decision.

<!-- end list -->

Python

```
    def draw(self, screen, show_brain=False):
        if not self.alive:
            return

        pygame.draw.circle(screen, BLUE, (int(self.x), int(self.y)), self.radius)
        pygame.draw.circle(screen, BLACK, (int(self.x), int(self.y)), self.radius, 1)

        eye_x = self.x + 5 if self.velocity > 0 else self.x - 5
        pygame.draw.circle(screen, WHITE, (int(eye_x), int(self.y - 5)), 5)
        pygame.draw.circle(screen, BLACK, (int(eye_x), int(self.y - 5)), 2)

        beak_offset = 10 if self.velocity > 0 else -10
        pygame.draw.polygon(screen, (255, 165, 0), [
            (self.x + self.radius, self.y),
            (self.x + self.radius + beak_offset, self.y - 5),
            (self.x + self.radius + beak_offset, self.y + 5)
        ])
```

draw(): **Renders the bird on the screen**:

- It first checks if the bird is alive.
- Draws a blue circle for the bird's body, a black outline, white eyes with black pupils, and an orange beak. The eye and beak position change slightly based on the bird's velocity to simulate movement.

<!-- end list -->

Python

```
        if show_brain and self.alive:
            next_pipe = None
            for pipe in game.pipes:
                if pipe.x + PIPE_WIDTH > self.x - 20:
                    next_pipe = pipe
```

```
                    break

            if next_pipe:
                inputs = np.array([
                    self.y / SCREEN_HEIGHT,
                    (self.velocity + 10) / 20,
                    next_pipe.top_height / SCREEN_HEIGHT,
                    (next_pipe.x - self.x) / SCREEN_WIDTH,
                    (self.y - (next_pipe.top_height + PIPE_GAP/2)) / SCREEN_HEIGHT
                ])
                output = self.brain.predict(inputs)[0]

                decision_x = self.x + 30
                pygame.draw.rect(screen, BLACK, (decision_x - 15, self.y - 15, 30, 30), 1)
                if output > 0.5:
                    bar_height = (output - 0.5) * 30
                    pygame.draw.rect(screen, GREEN, (decision_x - 14, self.y + 14 - bar_height, 28, bar_height))
```

This block in draw() **visualizes the bird's neural network decision**:

- If show_brain is True and the bird is alive, it recalculates the same inputs as in think().
- It then draws a small black box next to the bird.
- If the neural network's output for flapping is greater than 0.5, a green bar is drawn inside the box, with its height proportional to how strongly the bird wants to flap. This provides a visual cue of the AI's internal state.

<!-- end list -->

```
Python
class Pipe:
    def __init__(self, x=None, top_height=None):
        self.x = SCREEN_WIDTH if x is None else x
```

```
        if top_height is None:
            self.top_height = random.randint(100, SCREEN_HEIGHT - PIPE_GAP - 100)
        else:
            self.top_height = top_height
        self.bottom_height = self.top_height + PIPE_GAP
        self.passed = False
        self.speed = 2
```

The Pipe class represents the obstacles:

- __init__: Constructor for a pipe:
    - x: Horizontal position. Defaults to SCREEN_WIDTH if not provided (for new pipes entering from the right).
    - top_height: Height of the top pipe segment. Randomly generated if not provided, ensuring a gap for the bird.
    - bottom_height: Calculated based on top_height and PIPE_GAP.
    - passed: Boolean indicating if a bird has successfully passed this pipe.
    - speed: How fast the pipe moves left.

<!-- end list -->

Python

```
    def update(self, speed_factor=1):
        self.x -= self.speed * speed_factor
```

update(): **Moves the pipe to the left** based on its speed and the global speed_factor.

Python

```
    def draw(self, screen):
        pygame.draw.rect(screen, GREEN, (self.x, 0, PIPE_WIDTH, self.top_height))
        pygame.draw.rect(screen, BLACK, (self.x, 0, PIPE_WIDTH, self.top_height), 2)

        pygame.draw.rect(screen, GREEN, (self.x, self.bottom_height, PIPE_WIDTH, SCREEN_HEIGHT - self.bottom_height))
```

```
        pygame.draw.rect(screen, BLACK, (self.x, self.bottom_height, PIPE_WIDTH, SCREEN_HEIGHT - self.bottom_height), 2)

        gap_center = self.top_height + PIPE_GAP/2
        pygame.draw.line(screen, YELLOW, (self.x, gap_center), (self.x + PIPE_WIDTH, gap_center), 2)
```

draw(): **Renders the pipe on the screen**:

- Draws two green rectangles (top and bottom pipes) with black outlines.
- Draws a yellow line in the middle of the gap, which can act as a visual target for the birds.

<!-- end list -->

Python

```
    def collides_with(self, bird):
        if not bird.alive:
            return False

        if (bird.x + bird.radius > self.x and bird.x - bird.radius < self.x + PIPE_WIDTH):
            if (bird.y - bird.radius < self.top_height or bird.y + bird.radius > self.bottom_height):
                return True
        return False
```

collides_with(): **Checks for collision between a bird and a pipe**:

- Returns False if the bird is not alive.
- It performs two checks:
  - Horizontal overlap: bird.x + bird.radius > self.x (bird's right edge past pipe's left edge) AND bird.x - bird.radius < self.x + PIPE_WIDTH (bird's left edge past pipe's right edge).
  - Vertical overlap: bird.y - bird.radius < self.top_height (bird's top edge past top pipe's bottom edge) OR bird.y + bird.radius > self.bottom_height (bird's bottom edge past bottom pipe's top edge).
- If both horizontal and vertical overlaps occur, it returns True, indicating a collision.

<!-- end list -->

Python

```
class NeuralNetwork:
    def __init__(self, input_size, hidden_size, output_size):
        self.weights1 = np.random.randn(input_size, hidden_size) * np.sqrt(2./input_size)
        self.weights2 = np.random.randn(hidden_size, output_size) * np.sqrt(2./hidden_size)
        self.bias1 = np.zeros((1, hidden_size))
        self.bias2 = np.zeros((1, output_size))
```

The NeuralNetwork class implements a simple **feedforward neural network**:

- __init__: Constructor to set up the network's architecture.
    - input_size, hidden_size, output_size: Number of neurons in each layer.
    - weights1, weights2: Weight matrices for connections between input-hidden and hidden-output layers, initialized using He initialization (np.sqrt(2./input_size)) for ReLU.
    - bias1, bias2: Bias vectors for each layer, initialized to zeros.

<!-- end list -->

Python

```
    def copy(self):
        new_nn = NeuralNetwork(1, 1, 1) # Placeholder sizes
        new_nn.weights1 = self.weights1.copy()
        new_nn.weights2 = self.weights2.copy()
        new_nn.bias1 = self.bias1.copy()
        new_nn.bias2 = self.bias2.copy()
        return new_nn
```

copy(): **Creates a deep copy of the neural network's weights and biases**. This is crucial for genetic algorithms when creating new generations from "parent" networks. The (1, 1, 1) are just placeholder sizes; they are immediately overwritten by the actual copied weights/biases.

Python

```
    def predict(self, inputs):
        hidden = np.dot(inputs, self.weights1) + self.bias1
        hidden = np.maximum(0, hidden) # ReLU activation
        output = np.dot(hidden, self.weights2) + self.bias2
        output = 1 / (1 + np.exp(-output)) # Sigmoid activation
        return output
```

predict(): **Performs a forward pass through the network**:

- hidden = np.dot(inputs, self.weights1) + self.bias1: Calculates the weighted sum of inputs and adds bias for the hidden layer.
- hidden = np.maximum(0, hidden): Applies the **ReLU (Rectified Linear Unit)** activation function to the hidden layer.
- output = np.dot(hidden, self.weights2) + self.bias2: Calculates the weighted sum from the hidden layer to the output layer and adds bias.
- output = 1 / (1 + np.exp(-output)): Applies the **sigmoid activation function** to the output layer, squishing the output to a range between 0 and 1. This is suitable for binary classification (flap or don't flap).

<!-- end list -->

Python

```
    def mutate(self, rate):
        def mutate_array(arr):
            mask = np.random.random(arr.shape) < rate
            random_values = np.random.randn(*arr.shape) * 0.5
            arr[mask] += random_values[mask]
            arr = np.clip(arr, -5, 5)
            return arr

        self.weights1 = mutate_array(self.weights1)
        self.weights2 = mutate_array(self.weights2)
        self.bias1 = mutate_array(self.bias1)
```

```
        self.bias2 = mutate_array(self.bias2)
```

mutate(): **Introduces random changes to the network's weights and biases**:

- rate: The probability of a weight/bias being mutated.
- mutate_array: A helper function that:
  - Creates a mask where True indicates elements to be mutated (based on rate).
  - Generates random_values (Gaussian noise) to add to the mutated elements.
  - Adds these random_values to the arr where the mask is True.
  - np.clip(arr, -5, 5): **Clips the values** to keep them within a reasonable range, preventing weights from becoming too large or too small, which can destabilize training.
- This function is applied to all weight and bias matrices.

<!-- end list -->

```
Python
class Game:
    def __init__(self, population_size=50):
        self.screen = pygame.display.set_mode((SCREEN_WIDTH, SCREEN_HEIGHT))
        pygame.display.set_caption("Flappy Bird AI - Consistent Obstacles")
        self.clock = pygame.time.Clock()
        self.speed_factor = 1
        self.show_brain = False
        self.population_size = population_size
        self.pipe_sequence = []
        self.reset()
```

The Game class manages the overall game simulation and the genetic algorithm:

- __init__: Constructor for the game:
  - Sets up the Pygame screen and window title.
  - self.clock: Used to control the frame rate.
  - self.speed_factor: Controls the game speed (1x to 10x).
  - self.show_brain: Toggles the visualization of the AI's decision.

- self.population_size: Number of birds in each generation.
- self.pipe_sequence: An empty list that will store a predefined sequence of pipe positions for consistent obstacle generation across generations.
- self.reset(): Calls the reset method to initialize game state.

<!-- end list -->

Python

```python
    def generate_pipe_sequence(self):
        self.pipe_sequence = []
        x = SCREEN_WIDTH
        for _ in range(50):
            top_height = random.randint(100, SCREEN_HEIGHT - PIPE_GAP - 100)
            self.pipe_sequence.append((x, top_height))
            x += PIPE_FREQUENCY // 2
```

generate_pipe_sequence(): **Creates a fixed sequence of pipe positions and gaps**. This is crucial for evaluating different generations of birds on the *same set of challenges*, making fitness comparison fair and consistent.

- It generates 50 pipes.
- x starts at SCREEN_WIDTH and increments by PIPE_FREQUENCY // 2 for each subsequent pipe.
- top_height is randomly chosen within a safe range.

<!-- end list -->

Python

```python
    def reset(self):
        if not self.pipe_sequence:
            self.generate_pipe_sequence()

        self.birds = [Bird(100, SCREEN_HEIGHT // 2) for _ in range(self.population_size)]
        self.pipes = []
        self.last_pipe_time = pygame.time.get_ticks()
```

```
        self.generation = 1
        self.best_score = 0
        self.best_pipes_passed = 0
        self.running = True
        self.next_pipe_index = 0
        self.add_pipe()
```

reset(): **Resets the game state for a new generation or a new game run**:

- If pipe_sequence is empty, it calls generate_pipe_sequence() to create it.
- self.birds: Creates a new population of Bird objects.
- Resets pipes, last_pipe_time, generation, best_score, best_pipes_passed, and running status.
- self.next_pipe_index: Tracks which pipe from the pipe_sequence to add next.
- self.add_pipe(): Adds the first pipe from the sequence.

<!-- end list -->

Python

```
    def add_pipe(self):
        if self.next_pipe_index < len(self.pipe_sequence):
            x, top_height = self.pipe_sequence[self.next_pipe_index]
            self.pipes.append(Pipe(x, top_height))
            self.next_pipe_index += 1
            self.last_pipe_time = pygame.time.get_ticks()
```

add_pipe(): **Adds a new pipe to the game** from the predefined pipe_sequence.

- It checks if there are still pipes left in the sequence to add.

<!-- end list -->

Python

```
    def natural_selection(self):
        for bird in self.birds:
            bird.fitness = bird.score + bird.pipes_passed * 500
```

```
    self.birds.sort(key=lambda x: x.fitness, reverse=True)

    top_birds = self.birds[:max(2, len(self.birds)//5)]

    new_birds = []

    for bird in top_birds[:2]:
        new_bird = Bird(100, SCREEN_HEIGHT // 2, bird.brain)
        new_birds.append(new_bird)

    for _ in range(self.population_size - 2):
        candidates = random.sample(top_birds, min(3, len(top_birds)))
        parent = max(candidates, key=lambda x: x.fitness)
        child_brain = parent.brain.copy()
        child_brain.mutate(0.15)
        new_birds.append(Bird(100, SCREEN_HEIGHT // 2, child_brain))

    self.birds = new_birds
    self.generation += 1
    self.pipes = []
    self.next_pipe_index = 0
    self.add_pipe()

    if self.generation % 20 == 0:
        for bird in self.birds:
            bird.brain.mutate(0.02)
```

natural_selection(): This is the **core of the genetic algorithm**:

- **Calculate Fitness**: For each bird, fitness is calculated as score + pipes_passed * 500. Passing pipes gives a significant bonus, encouraging birds to survive longer.
- **Sort Birds**: The birds are sorted by their fitness in descending order.
- **Select Top Birds**: top_birds are the top performing birds (the best 2, or 20% of the population, whichever is greater).
- **Create New Generation (new_birds)**:
    - The **top 2 birds (elites)** are copied directly to the new generation without mutation, preserving the best traits.
    - The remaining population_size - 2 birds are created by **selecting a parent using tournament selection** (picking 3 random candidates from top_birds and choosing the best among them), copying their brain, and then mutating it.
- **Replace Population**: self.birds is updated with the new_birds.
- **Reset Game State**: The generation count is incremented, pipes are cleared, next_pipe_index is reset, and the first pipe for the new generation is added.
- **Periodic Mutation Boost**: Every 20 generations, a small mutation (0.02 rate) is applied to *all* birds in the new generation. This helps introduce more diversity and prevent the population from getting stuck in a local optimum.

<!-- end list -->

Python

```
    def handle_events(self):
        for event in pygame.event.get():
            if event.type == QUIT:
                self.running = False
            elif event.type == KEYDOWN:
                if event.key == K_ESCAPE:
                    self.running = False
                elif event.key == K_r:
                    self.generate_pipe_sequence()
                    self.reset()
                elif event.key == K_b:
```

```
                    self.show_brain = not self.show_brain
                elif event.key == K_0:
                    self.speed_factor = 10
                elif K_1 <= event.key <= K_9:
                    self.speed_factor = event.key - K_0
```

handle_events(): **Processes user input**:

- QUIT or K_ESCAPE: Exits the game.
- K_r: Regenerates a *new, random* pipe sequence and resets the game for a completely new challenge.
- K_b: Toggles the show_brain visualization.
- K_0: Sets speed to 10x.
- K_1 to K_9: Sets the game speed_factor to the corresponding number.

<!-- end list -->

Python

```
    def update(self):
        current_time = pygame.time.get_ticks()
        if current_time - self.last_pipe_time > PIPE_FREQUENCY / self.speed_factor:
            self.add_pipe()

        for pipe in self.pipes[:]:
            pipe.update(self.speed_factor)
            if pipe.x < -PIPE_WIDTH:
                self.pipes.remove(pipe)

        alive_birds = [bird for bird in self.birds if bird.alive]

        if not alive_birds:
            self.natural_selection()
```

```
        return
```

update(): **Updates the game state for each frame**:

- **Pipe Generation**: Checks if it's time to add_pipe based on PIPE_FREQUENCY and speed_factor.
- **Pipe Movement and Removal**: Iterates through existing pipes, updates their position, and removes them if they go off-screen to the left.
- **Check for All Birds Dead**: If alive_birds is empty (all birds have died), it triggers natural_selection() to evolve the population and returns, ending the current game cycle for that generation.

<!-- end list -->

Python

```
    for _ in range(min(3, self.speed_factor)): # Simulate multiple steps at higher speeds
        for bird in alive_birds[:]:
            if not bird.alive:
                alive_birds.remove(bird)
                continue

            bird.think(self.pipes)
            bird.update()
            bird.score += 1

            for pipe in self.pipes:
                if pipe.collides_with(bird):
                    bird.alive = False

            for pipe in self.pipes:
                if not pipe.passed and pipe.x + PIPE_WIDTH < bird.x:
                    pipe.passed = True
                    bird.pipes_passed += 1
```

```
                    bird.score += 1000
```

This loop within update() **updates each alive bird's state**:

- The outer loop for _ in range(min(3, self.speed_factor)) allows the game to effectively simulate multiple frames of bird and pipe movement per actual display frame when speed_factor is high. This speeds up the simulation without dropping visual frames, making the AI training faster to observe.
- It iterates through alive_birds:
    - If a bird is no longer alive (e.g., from hitting a boundary), it's removed from alive_birds.
    - bird.think(self.pipes): The bird's neural network decides whether to flap.
    - bird.update(): The bird's position and velocity are updated.
    - bird.score += 1: The bird's score increases simply by surviving.
    - **Collision Detection**: It checks for collisions with all pipes. If a collision occurs, the bird's alive status is set to False.
    - **Pipe Passing**: It checks if a bird has successfully passed a pipe (pipe.x + PIPE_WIDTH < bird.x). If so, pipe.passed is set to True (so it's only counted once), the bird's pipes_passed count increases, and its score gets a significant bonus (1000 points).

<!-- end list -->

Python

```
        current_max_pipes = max(bird.pipes_passed for bird in self.birds)
        if current_max_pipes > self.best_pipes_passed:
            self.best_pipes_passed = current_max_pipes

        current_max_score = max(bird.score for bird in self.birds)
        if current_max_score > self.best_score:
            self.best_score = current_max_score
```

This block **tracks and updates the best scores and pipes passed** across all generations.

Python

```
    def draw(self):
```

```
        self.screen.fill(SKY_BLUE)

        for pipe in self.pipes:
            pipe.draw(self.screen)

        # Only draw alive birds
        for bird in self.birds:
            if bird.alive:
                bird.draw(self.screen, self.show_brain)

        alive_count = sum(1 for bird in self.birds if bird.alive)
        stats_text = [
            f"Generation: {self.generation}",
            f"Alive: {alive_count}/{len(self.birds)}",
            f"Best Pipes Passed: {self.best_pipes_passed}",
            f"Current Max Pipes: {max(bird.pipes_passed for bird in self.birds) if self.birds else 0}",
            f"Speed: {self.speed_factor}x (1-9,0)",
            f"Show Brain: {'ON' if self.show_brain else 'OFF'} (B)",
            f"Press R to generate new obstacles"
        ]

        for i, text in enumerate(stats_text):
            text_surface = FONT.render(text, True, BLACK)
            self.screen.blit(text_surface, (10, 10 + i * 25))

        pygame.display.flip()
```

draw(): **Renders all game elements and statistics**:

- Fills the screen with SKY_BLUE.

- Draws all active pipes.
- Draws only the alive birds, optionally showing their brain visualization.
- Calculates and displays various statistics using FONT.render and screen.blit, including:
    - Current generation.
    - Number of alive birds.
    - Best pipes passed ever by any bird.
    - Current generation's max pipes passed.
    - Current speed factor.
    - Brain visualization status.
    - Hint for regenerating obstacles.
- pygame.display.flip(): **Updates the entire screen** to show what has been drawn.

<!-- end list -->

Python

```
    def run(self):
        while self.running:
            self.handle_events()
            self.update()
            self.draw()
            self.clock.tick(60)

        pygame.quit()
```

run(): The **main game loop**:

- while self.running: Continues as long as the running flag is True.
- Calls handle_events(), update(), and draw() in sequence.
- self.clock.tick(60): Caps the frame rate at 60 frames per second.
- pygame.quit(): Uninitializes Pygame modules when the loop ends.

<!-- end list -->

Python

```python
if __name__ == "__main__":
    game = Game(population_size=50)
    game.run()
```

This is the **entry point of the script**:

- if __name__ == "__main__":: Ensures the code inside this block only runs when the script is executed directly (not when imported as a module).
- game = Game(population_size=50): Creates an instance of the Game class with a population of 50 birds.
- game.run(): Starts the game loop.
