***Prompt:***

*Help me create a snake game in Python. Then, make it play itself; the snake should automatically play the game, eat fruit, and stay alive.*

**Reasoning/planning:**

**Crafting the snake**

I’m figuring out how to build a Python-based snake game using Pygame, incorporating an AI for automatic gameplay, including fruit consumption and survival.

**Crafting the snake AI**

I’m writing a Python script with Pygame for a self-playing snake game, using a simple A\* search algorithm to navigate the snake towards fruit, ensuring its survival.

**Charting safe paths**

I’m piecing together a streamlined method to navigate the snake, ensuring it can eat the fruit and safely return to its tail, avoiding dead-ends through a simplified algorithmic approach.

**Mapping a secure route**

I’m setting up the snake's grid coordinates and fruit's position, then leveraging BFS to find the shortest path to the fruit, ensuring the snake can return to its tail.

**Calculating paths**

I'm mapping the snake's route, using BFS/DFS to ensure the snake can eat fruit safely, and if not, find a way to return to its tail without collisions.

**Piecing together paths**

I’m using BFS to find the shortest route to the fruit, ensuring the snake can safely return to its tail, or in case of no safe path, making a secure move to avoid collision.

**Crafting the setup**

I’m laying out the plan to use Pygame for visualization, initializing the game window, and setting up the grid size in preparation for the snake game simulation.

**Setting up game logistics**

I’m preparing to import libraries, initialize the game, and configure the grid. Then, I’ll define constants, classes for the snake and fruit, and map the shortest path using BFS.

**Laying out the game elements**

I’m thinking through Pygame for graphics. After setting up the grid, defining the Snake and Fruit classes, and constants, I’ll use BFS to find the shortest path.

**Tracking progress**

I’m tracking the snake's movements, checking for collisions, and updating the game state each frame. Logical steps ensure accurate game progression and outcomes.

**Charting the course**

I’m working on defining a find\_path() function to trace the snake's journey towards the fruit while avoiding obstacles. Adding logic for safe moves and implementing AI decision-making.

**Tracking paths**

I’m thinking through the snake's journey to the fruit, ensuring its tail remains safe by checking alternate paths if the immediate route is obstructed.

**Breaking it down**

OK, let me see. The snake moves by extending its head and retracting its tail. I’m generating a safe direction and updating the grid coordinates, ensuring the new fruit is strategically placed.

**Mapping the route**

I’m coding the BFS function to find a path avoiding obstacles. The function will return a list of nodes if a path exists, or an empty list if no path is found.

**Refining the approach**

I’m mapping out a BFS function to find a path, considering the snake body as obstacles. The function will return a list of nodes if a path exists, or an empty list if not.

**Considering path-adjustments**

I’m working on a BFS function. It uses obstacles as snake body parts but treats the tail as free for pathfinding, allowing a flexible and efficient search.

**Mapping strategies**

I'm thinking through two BFS functions: one treats the snake's body as obstacles, while the other uses the tail as free space for safe moves.

**Laying out the options**

I’m piecing together the BFS function, defining neighbors, checking boundaries, and setting grid sizes. This approach should streamline pathfinding nicely.

**Setting up the framework**

I’m configuring Pygame and constants for the snake game. The BFS function will use grid size, directions, and free-tail logic.

**Strategizing moves**

I'm generating a BFS function to plot the snake's route. It considers obstacles, tail, and safe paths. The AI's choices balance fruit gathering and avoiding collisions.

**Mapping movements**

AI is making strategic moves to help the snake collect fruit while dodging collisions. This involves calculating the path to the fruit and considering the snake’s growth post-consumption.

**Simulating fruit consumption**

I’m working through the simulation code, noting that updating the fruit position isn't crucial. The key is ensuring the snake can reach its tail for safety.

**Verifying growth**

Ensuring the snake's growth is correctly simulated: the tail is only removed if the new head isn't at the fruit. This approach maintains accuracy in the simulation.

**Ensuring BFS alignment**

Verifying BFS boundaries and obstacle removal. The edge case of a one-segment snake tail equals the head, which is manageable. The simulation code is accurate.

**Balancing computational efficiency**

Handling edge cases like a 1-unit snake where the tail matches the head. Ensuring BFS longevity and gameplay continuity. Ready to provide optimized code with thorough explanations.

Below is one complete example that uses Pygame to both display a Snake game and have an “AI” control the snake automatically. In this example the AI uses a simple strategy:

1. **Find a path to the fruit:** It uses breadth‐first search (BFS) on the grid (with the snake’s body as obstacles) to compute a path from the snake’s head to the fruit.
2. **Check “safety”:** Before following that path, the AI “simulates” following it and then checks (again via BFS) that from the new head the snake’s tail remains reachable. (Since the tail moves, being able to “chase” it is a simple way to help avoid dead‐ends.)
3. **Fallback moves:** If no safe path to the fruit exists, it will try to “follow its tail” or—if needed—choose any safe move.

You can tweak or improve the strategy further, but the following code is a good starting point.

**Note:** You must have [Pygame](https://www.pygame.org/news) installed. (You can install it with pip install pygame.)

Below is the full code:

import pygame

import random

import sys

from collections import deque

# === Configuration constants ===

CELL\_SIZE = 20

GRID\_WIDTH = 20

GRID\_HEIGHT = 20

WINDOW\_WIDTH = CELL\_SIZE \* GRID\_WIDTH

WINDOW\_HEIGHT = CELL\_SIZE \* GRID\_HEIGHT

FPS = 10

# Colors (R, G, B)

WHITE = (255, 255, 255)

BLACK = (0, 0, 0)

GREEN = (0, 255, 0)

RED = (255, 0, 0)

# Directions (dx, dy)

UP = (0, -1)

DOWN = (0, 1)

LEFT = (-1, 0)

RIGHT = (1, 0)

# === Helper functions ===

def get\_random\_position(snake):

"""Return a random grid cell that is not occupied by the snake."""

while True:

pos = (random.randint(0, GRID\_WIDTH - 1), random.randint(0, GRID\_HEIGHT - 1))

if pos not in snake:

return pos

def bfs(start, target, snake, allow\_tail\_as\_free=False):

"""

Use breadth-first search (BFS) to find a path from start to target.

The snake’s body cells are treated as obstacles.

If allow\_tail\_as\_free is True then the tail cell is considered free (since it will move).

Returns:

A list of grid positions (cells) that is the shortest path from start to target,

not including the start cell. Returns None if no path is found.

"""

obstacles = set(snake)

if allow\_tail\_as\_free and snake:

# The tail will move; so temporarily remove it as an obstacle.

obstacles.remove(snake[-1])

queue = deque()

queue.append(start)

came\_from = {start: None}

while queue:

current = queue.popleft()

if current == target:

# Reconstruct the path by walking backwards from target to start.

path = []

while current != start:

path.append(current)

current = came\_from[current]

path.reverse() # so that the first step is first

return path

# Check all four neighbors

for direction in [UP, DOWN, LEFT, RIGHT]:

next\_cell = (current[0] + direction[0], current[1] + direction[1])

# Make sure next\_cell is within bounds and not an obstacle.

if (0 <= next\_cell[0] < GRID\_WIDTH and

0 <= next\_cell[1] < GRID\_HEIGHT and

next\_cell not in came\_from and

next\_cell not in obstacles):

came\_from[next\_cell] = current

queue.append(next\_cell)

return None

def get\_direction(from\_cell, to\_cell):

"""Given two adjacent cells, return the direction as a (dx,dy) tuple."""

return (to\_cell[0] - from\_cell[0], to\_cell[1] - from\_cell[1])

def draw\_grid(surface):

"""Draw a grid on the given surface (for visual effect)."""

for x in range(0, WINDOW\_WIDTH, CELL\_SIZE):

pygame.draw.line(surface, WHITE, (x, 0), (x, WINDOW\_HEIGHT))

for y in range(0, WINDOW\_HEIGHT, CELL\_SIZE):

pygame.draw.line(surface, WHITE, (0, y), (WINDOW\_WIDTH, y))

# === Main game loop ===

def main():

pygame.init()

screen = pygame.display.set\_mode((WINDOW\_WIDTH, WINDOW\_HEIGHT))

pygame.display.set\_caption("AI Snake Game")

clock = pygame.time.Clock()

# Initialize snake: start with a length of 3 segments.

snake = [

(GRID\_WIDTH // 2, GRID\_HEIGHT // 2),

(GRID\_WIDTH // 2 - 1, GRID\_HEIGHT // 2),

(GRID\_WIDTH // 2 - 2, GRID\_HEIGHT // 2)

]

direction = RIGHT

# Place the first fruit

fruit = get\_random\_position(snake)

running = True

while running:

clock.tick(FPS)

# Handle quit events.

for event in pygame.event.get():

if event.type == pygame.QUIT:

running = False

# ===== AI DECISION MAKING =====

# First, try to find a path from the snake's head to the fruit.

path\_to\_fruit = bfs(snake[0], fruit, snake, allow\_tail\_as\_free=True)

if path\_to\_fruit is not None:

# Simulate following the path to check safety:

simulated\_snake = snake.copy()

for cell in path\_to\_fruit:

# Add the new head.

simulated\_snake.insert(0, cell)

if cell == fruit:

# If fruit is eaten, the snake grows (do not remove tail).

pass

else:

# Otherwise, remove the tail.

simulated\_snake.pop()

# Now check: can the simulated snake reach its own tail?

safe = bfs(simulated\_snake[0], simulated\_snake[-1], simulated\_snake, allow\_tail\_as\_free=True) is not None

if safe:

# Follow the first step along the path to fruit.

next\_cell = path\_to\_fruit[0]

direction = get\_direction(snake[0], next\_cell)

else:

# If following the fruit path is not “safe,” try to follow the tail.

path\_to\_tail = bfs(snake[0], snake[-1], snake, allow\_tail\_as\_free=True)

if path\_to\_tail is not None:

next\_cell = path\_to\_tail[0]

direction = get\_direction(snake[0], next\_cell)

else:

# If even that fails, pick any valid move.

for d in [UP, DOWN, LEFT, RIGHT]:

next\_cell = (snake[0][0] + d[0], snake[0][1] + d[1])

if (0 <= next\_cell[0] < GRID\_WIDTH and

0 <= next\_cell[1] < GRID\_HEIGHT and

next\_cell not in snake):

direction = d

break

else:

# No direct path to the fruit.

# Try moving toward the tail (which is always safe if available).

path\_to\_tail = bfs(snake[0], snake[-1], snake, allow\_tail\_as\_free=True)

if path\_to\_tail is not None:

next\_cell = path\_to\_tail[0]

direction = get\_direction(snake[0], next\_cell)

else:

# If no safe option is found, pick any valid direction.

for d in [UP, DOWN, LEFT, RIGHT]:

next\_cell = (snake[0][0] + d[0], snake[0][1] + d[1])

if (0 <= next\_cell[0] < GRID\_WIDTH and

0 <= next\_cell[1] < GRID\_HEIGHT and

next\_cell not in snake):

direction = d

break

# ===== MOVE THE SNAKE =====

new\_head = (snake[0][0] + direction[0], snake[0][1] + direction[1])

# Check for collisions with walls or itself.

if (new\_head[0] < 0 or new\_head[0] >= GRID\_WIDTH or

new\_head[1] < 0 or new\_head[1] >= GRID\_HEIGHT or

new\_head in snake):

print("Game over! Final score:", len(snake))

running = False

continue

# If the snake has eaten the fruit:

if new\_head == fruit:

snake.insert(0, new\_head)

fruit = get\_random\_position(snake)

else:

# Normal move: add new head and remove tail.

snake.insert(0, new\_head)

snake.pop()

# ===== DRAWING =====

screen.fill(BLACK)

# Draw the fruit.

pygame.draw.rect(screen, RED, (fruit[0] \* CELL\_SIZE, fruit[1] \* CELL\_SIZE, CELL\_SIZE, CELL\_SIZE))

# Draw the snake.

for segment in snake:

pygame.draw.rect(screen, GREEN, (segment[0] \* CELL\_SIZE, segment[1] \* CELL\_SIZE, CELL\_SIZE, CELL\_SIZE))

# Optionally, draw the grid lines.

draw\_grid(screen)

pygame.display.update()

pygame.quit()

sys.exit()

if \_\_name\_\_ == "\_\_main\_\_":

main()

**How It Works**

1. **Initialization:**  
   The game sets up a 20×20 grid where each cell is 20×20 pixels. The snake starts in the middle of the grid (with 3 segments), and a fruit is randomly placed on an unoccupied cell.
2. **The AI Loop:**  
   Each frame (controlled by a clock ticking at FPS frames per second) the AI does the following:
   * **Path to Fruit:** It attempts to compute a path to the fruit using BFS. The tail is temporarily allowed as “free” because it will move when the snake advances.
   * **Safety Check:** It simulates moving along that path (growing if fruit is eaten) and then uses BFS again to check whether from the new head the snake’s tail is reachable. This “chase the tail” idea is a common heuristic to help avoid trapping yourself.
   * **Fallback:** If the safe path to fruit is not available, it looks for a path to its tail or any safe move.
3. **Movement and Collision:**  
   The snake’s head is updated in the chosen direction. If the new head would hit the wall or the snake’s body, the game ends. If the fruit is eaten, the snake grows (by not removing the tail on that move) and a new fruit appears.
4. **Rendering:**  
   The game window is updated to display the snake (in green), the fruit (in red), and a grid in white over a black background.

You can run this script directly. Enjoy watching the snake “play itself” and feel free to experiment with the AI’s logic or the game settings!