FUNDAMENTOS DE INTELIGÊNCIA ARTIFICIAL

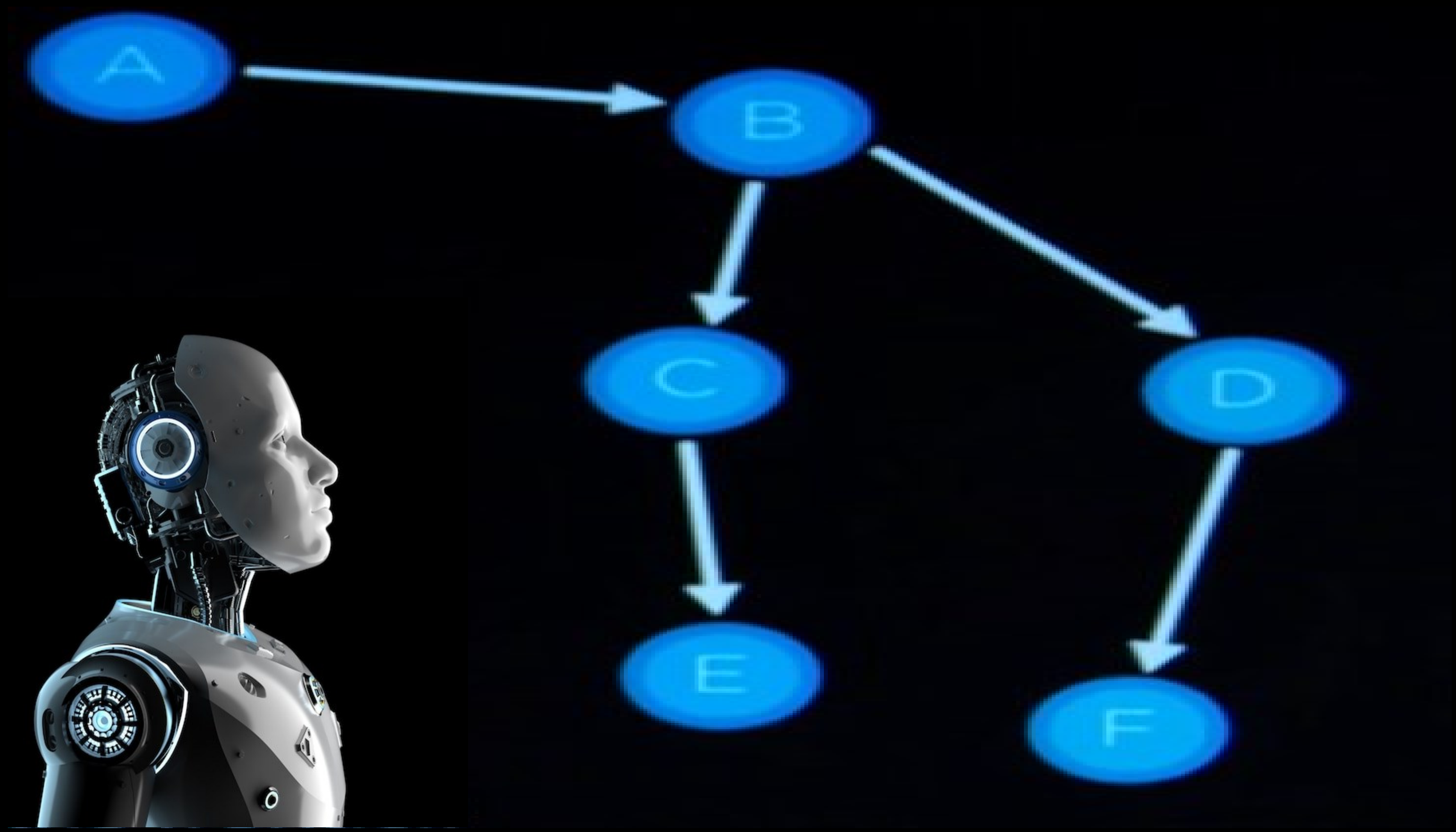
Aula 4:

FRONTIER

Prof. Dr. Rodrigo Xavier de Almeida Leão

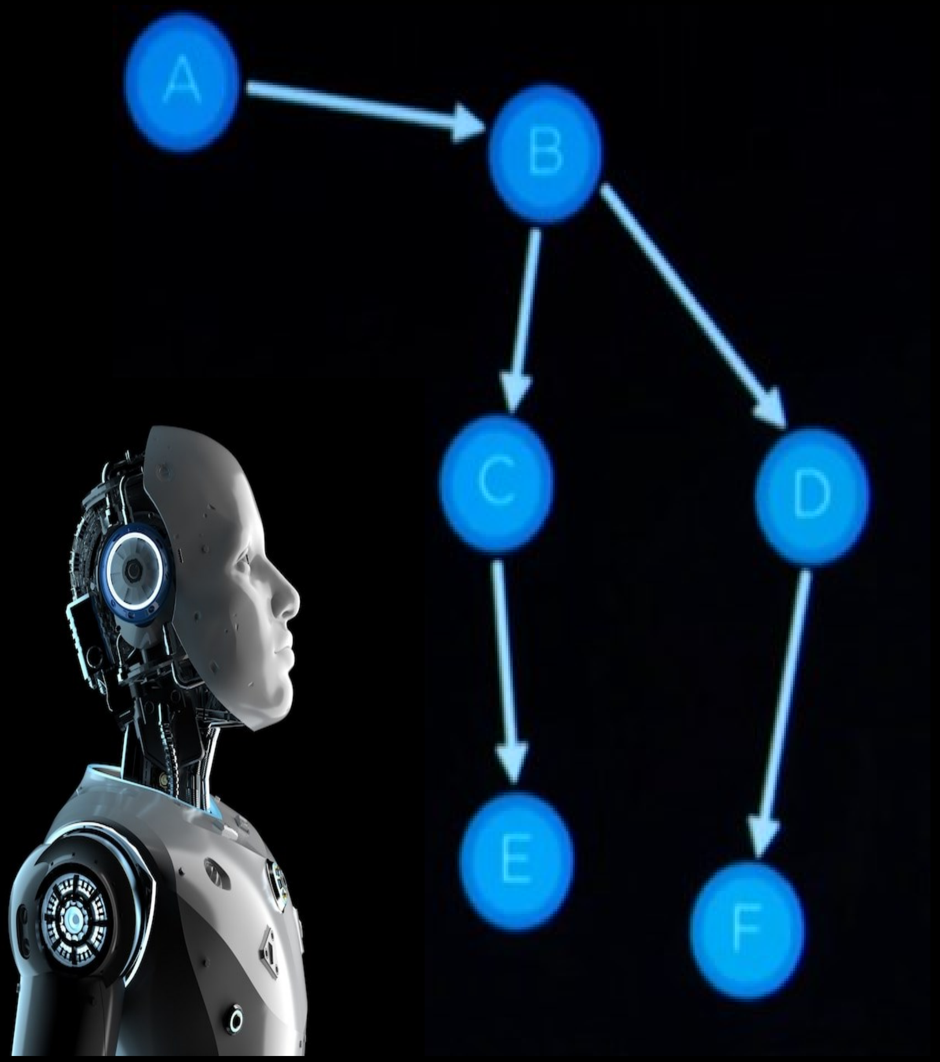
Cientista de Dados e Big Data

A -> E



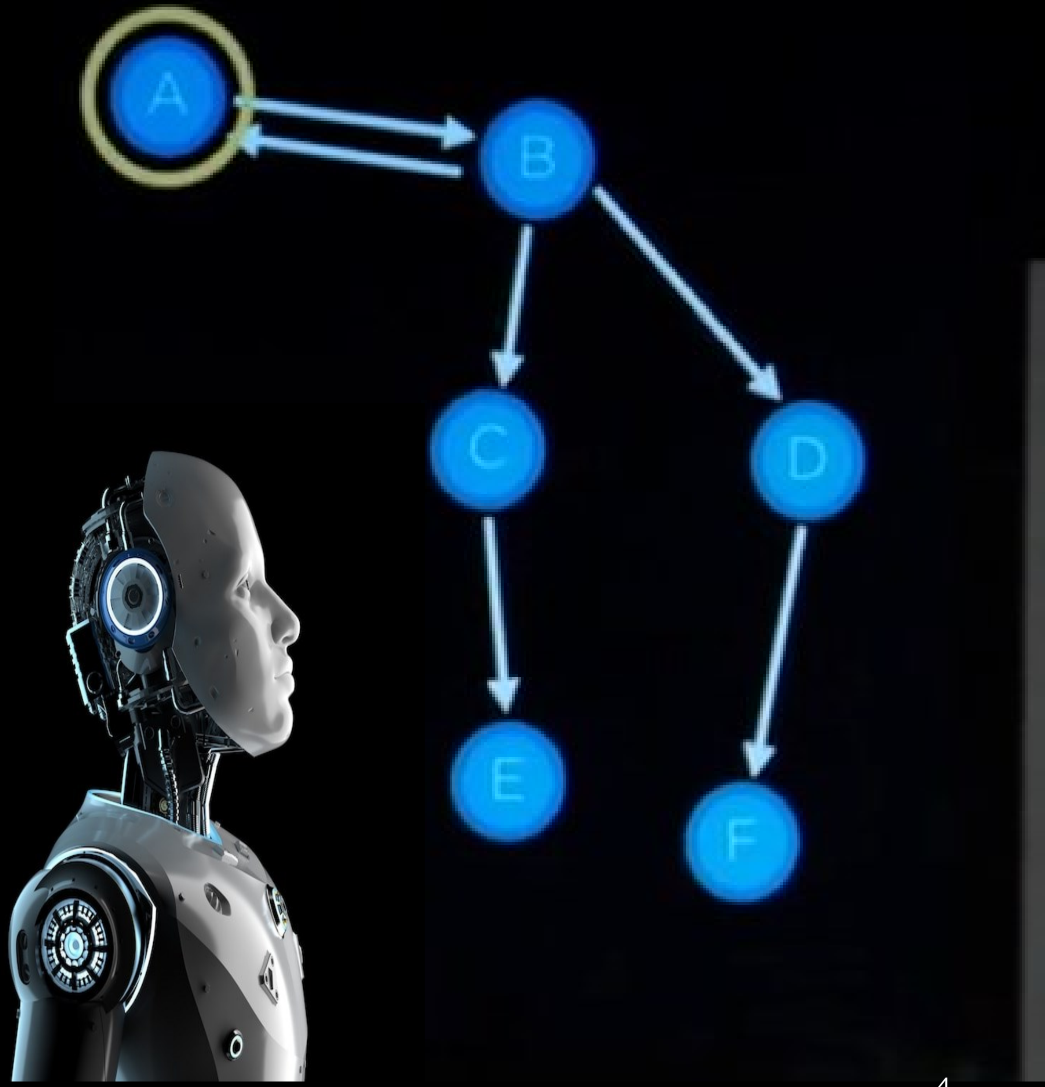
FRONTIER

1. FRONTIER = ESTADO INICIAL
2. REPETE:
3. SE O FRONTIER ESTÁ VAZIO ENTÃO NÃO HÁ SOLUÇÃO
4. REMOVA UM NÓ DO FRONTIER
5. SE O NÓ CONTÉM A SOLUÇÃO ENTÃO RETORNE SOLUÇÃO
6. SENÃO, EXPANDIR O NÓ E ADICIONAR OS NÓS RESULTANTES AO FRONTIER



FRONTIER

1. FRONTIER = ESTADO INICIAL
2. REPETE:
3. SE O FRONTIER ESTÁ VAZIO ENTÃO NÃO HÁ SOLUÇÃO
4. REMOVA UM NÓ DO FRONTIER
5. SE O NÓ CONTÉM A SOLUÇÃO ENTÃO RETORNE SOLUÇÃO
6. SENÃO, EXPANDIR O NÓ E ADICIONAR OS NÓS RESULTANTES AO FRONTIER



FRONTIER

1. FRONTIER = ESTADO INICIAL

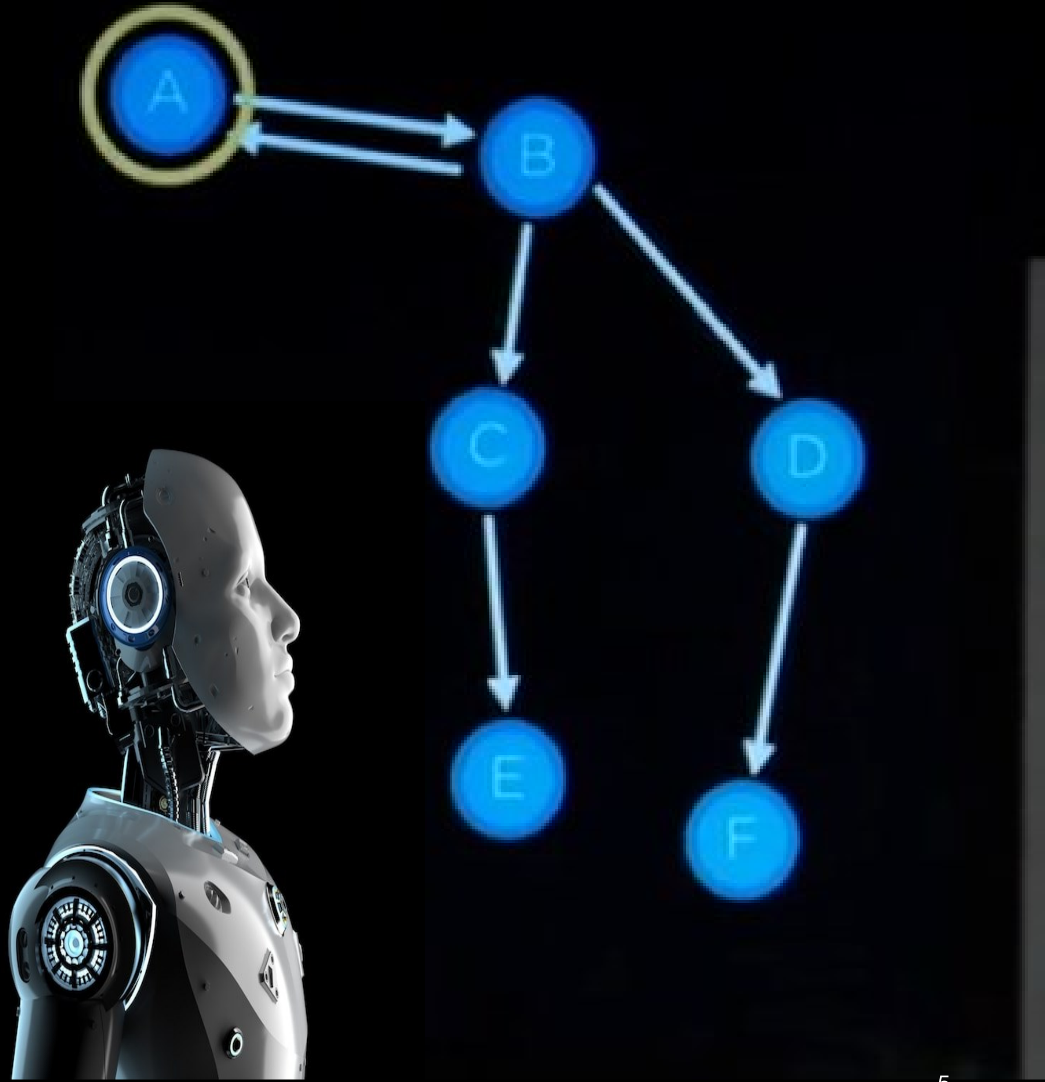
2. REPETE:

3. SE O FRONTIER ESTÁ VAZIO ENTÃO NÃO HÁ SOLUÇÃO

4. REMOVA UM NÓ DO FRONTIER

5. SE O NÓ CONTÉM A SOLUÇÃO ENTÃO RETORNE SOLUÇÃO

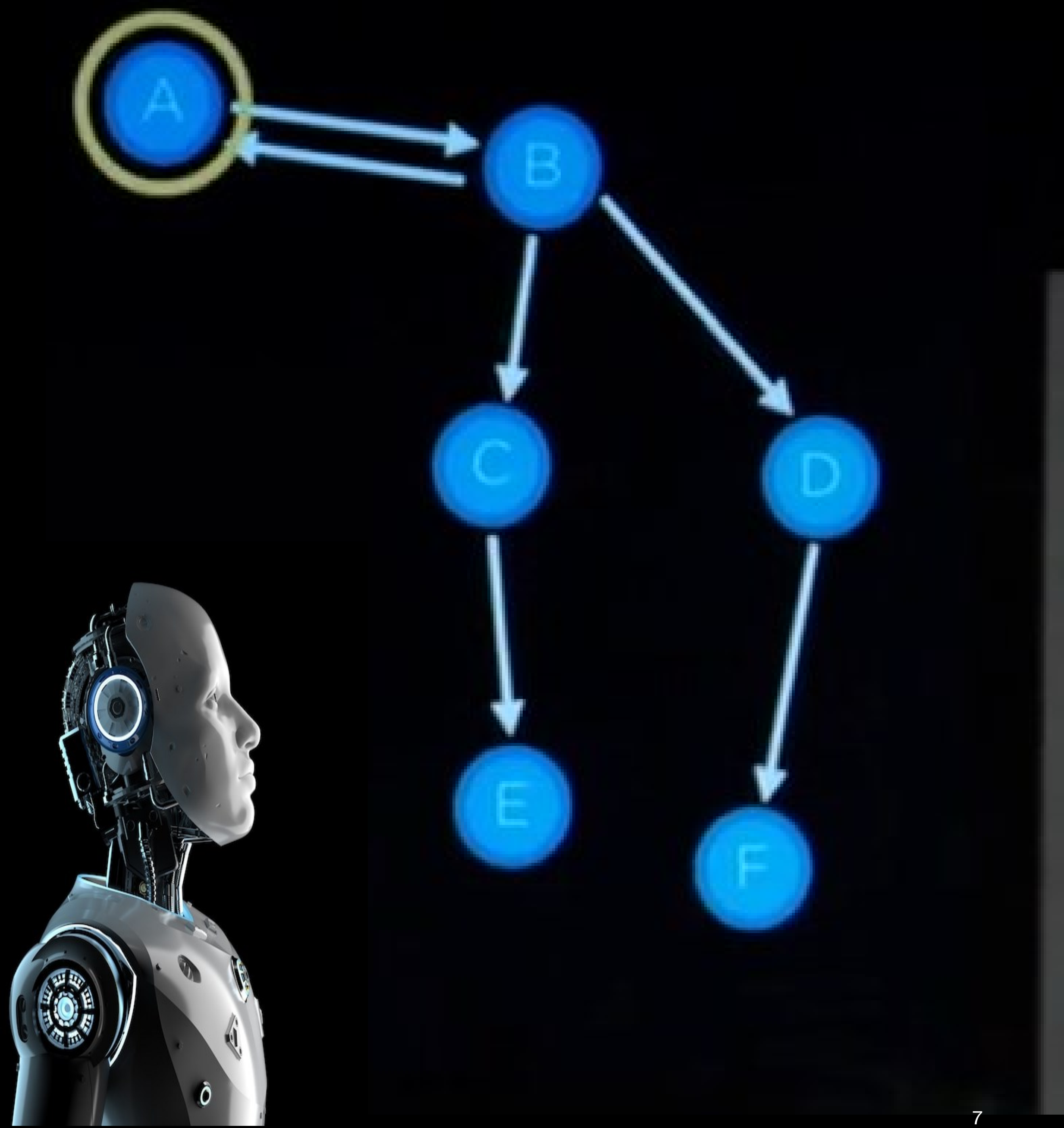
6. SENÃO, EXPANDIR O NÓ E ADICIONAR OS NÓS RESULTANTES AO FRONTIER



```
FRONTIER = ESTADO INICIAL
ESTADOS VISITADOS = VAZIO

REPETE:
  SE O FRONTIER ESTÁ VAZIO ENTÃO NÃO HÁ SOLUÇÃO
  REMOVA UM NÓ DO FRONTIER
  SE O NÓ CONTÉM A SOLUÇÃO ENTÃO RETORNE SOLUÇÃO
  ADICIONAR O NÓ AOS ESTADOS VISITADOS
  EXPANDIR O NÓ E ADICIONAR OS RESULTADOS AO FRONTIER SE ELES NÃO ESTÃO NO FRONTIER NEM NOS ESTADOS VISITADOS.
```

FRONTIER

EXPLORED



DEPTH-FIRST SEARCH

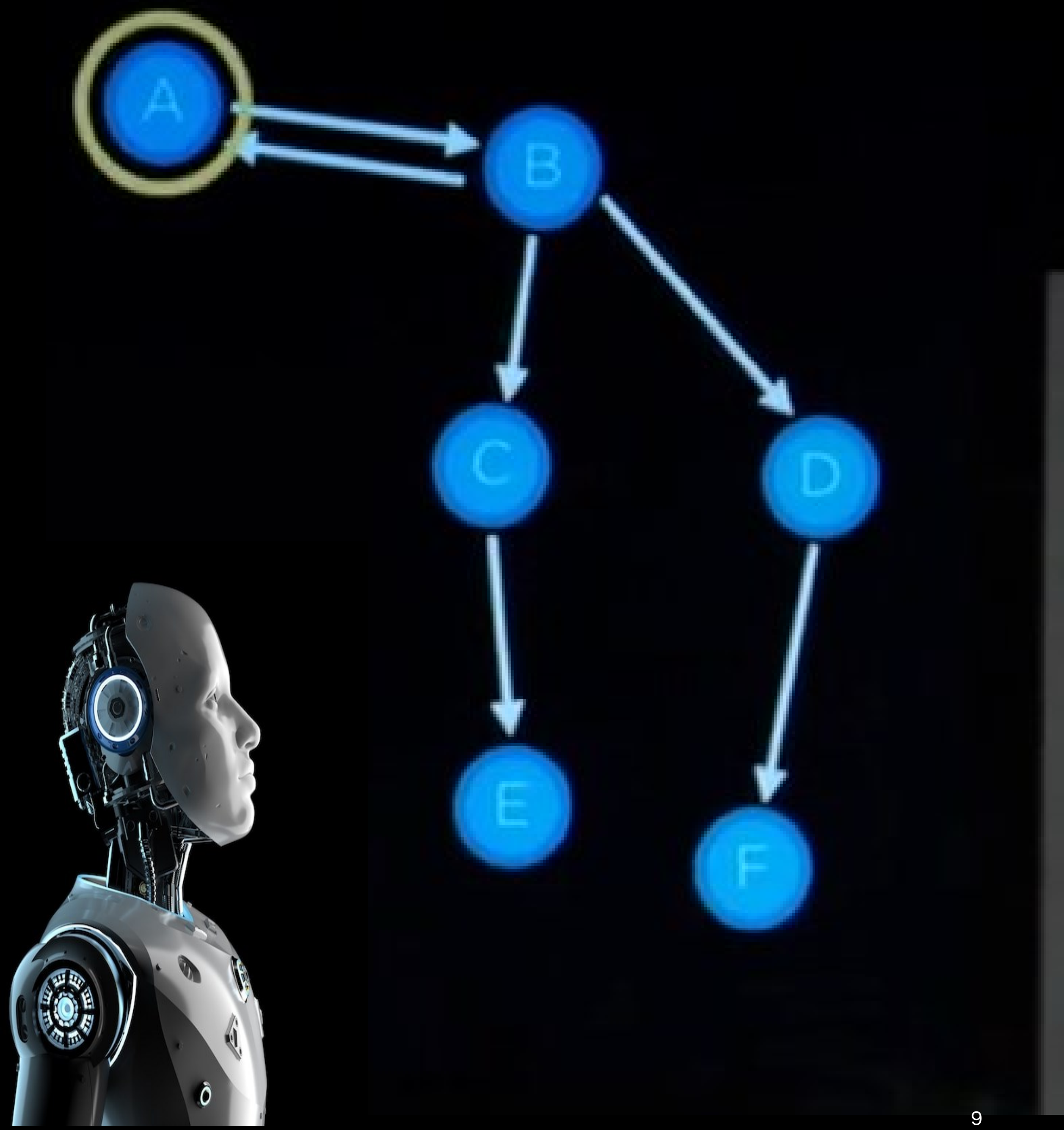
FRONTIER TIPO STACK

LAST IN FIRST OUT DATA TYPE

ALGORITMO QUE EXPANDE ATÉ O NÓ MAIS PROFUNDO DO FRONTIER

FRONTIER

EXPLORED



BREADTH-FIRST SEARCH

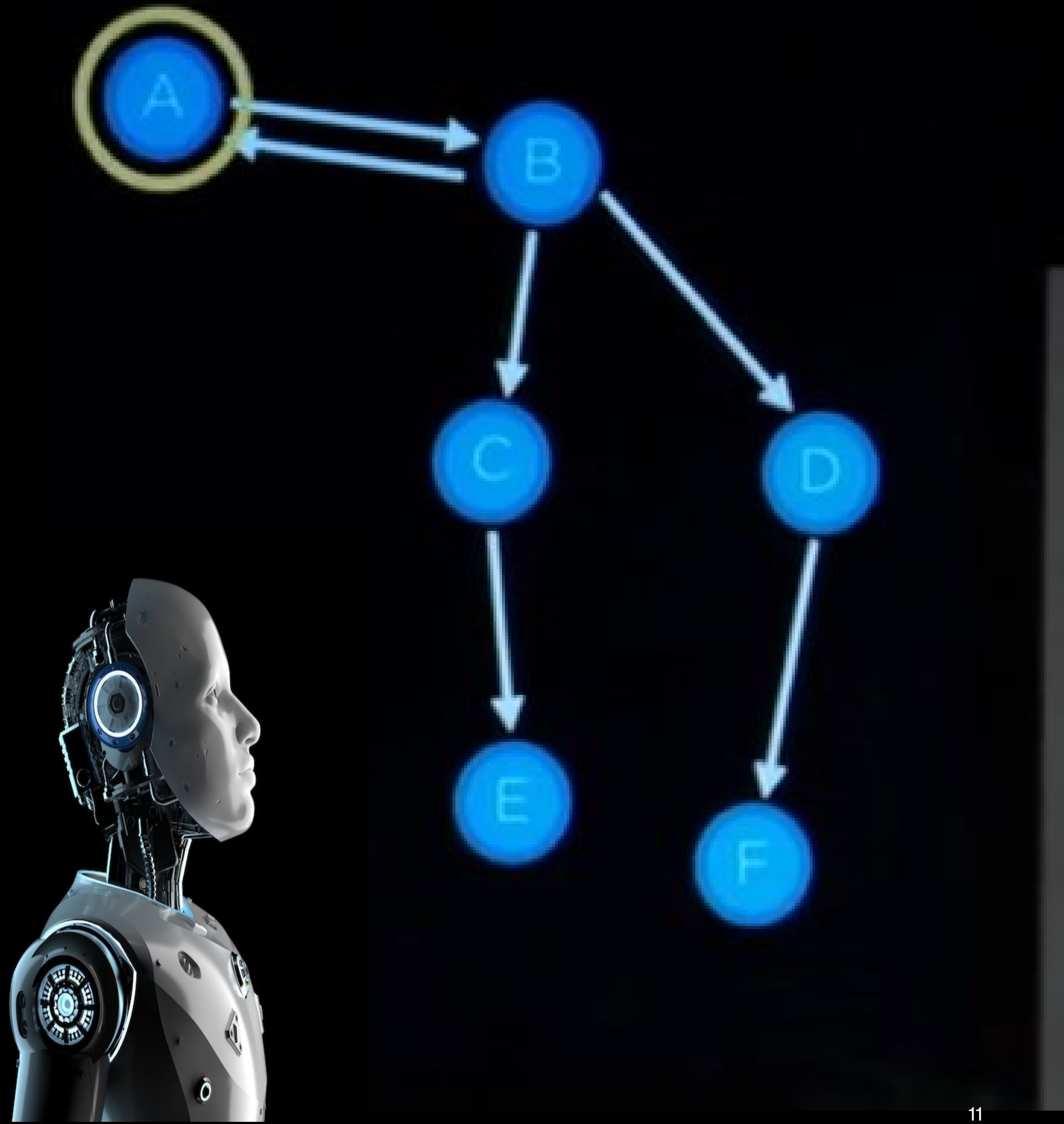
FRONTIER TIPO QUEUE

FIRST IN FIRST OUT DATA TYPE

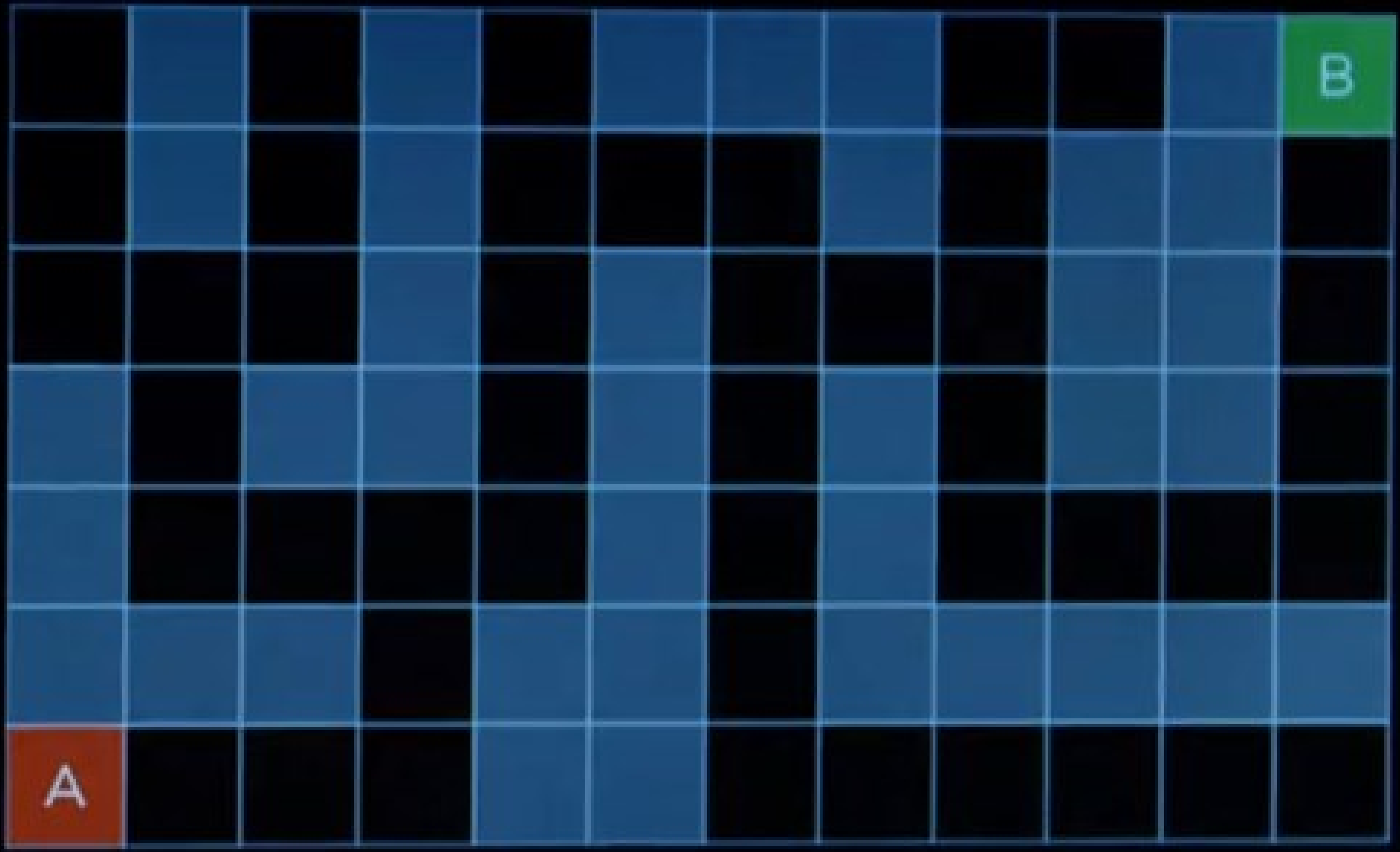
ALGORITMO QUE EXPANDE O NÓ MAIS RASO DO FRONTIER

FRONTIER

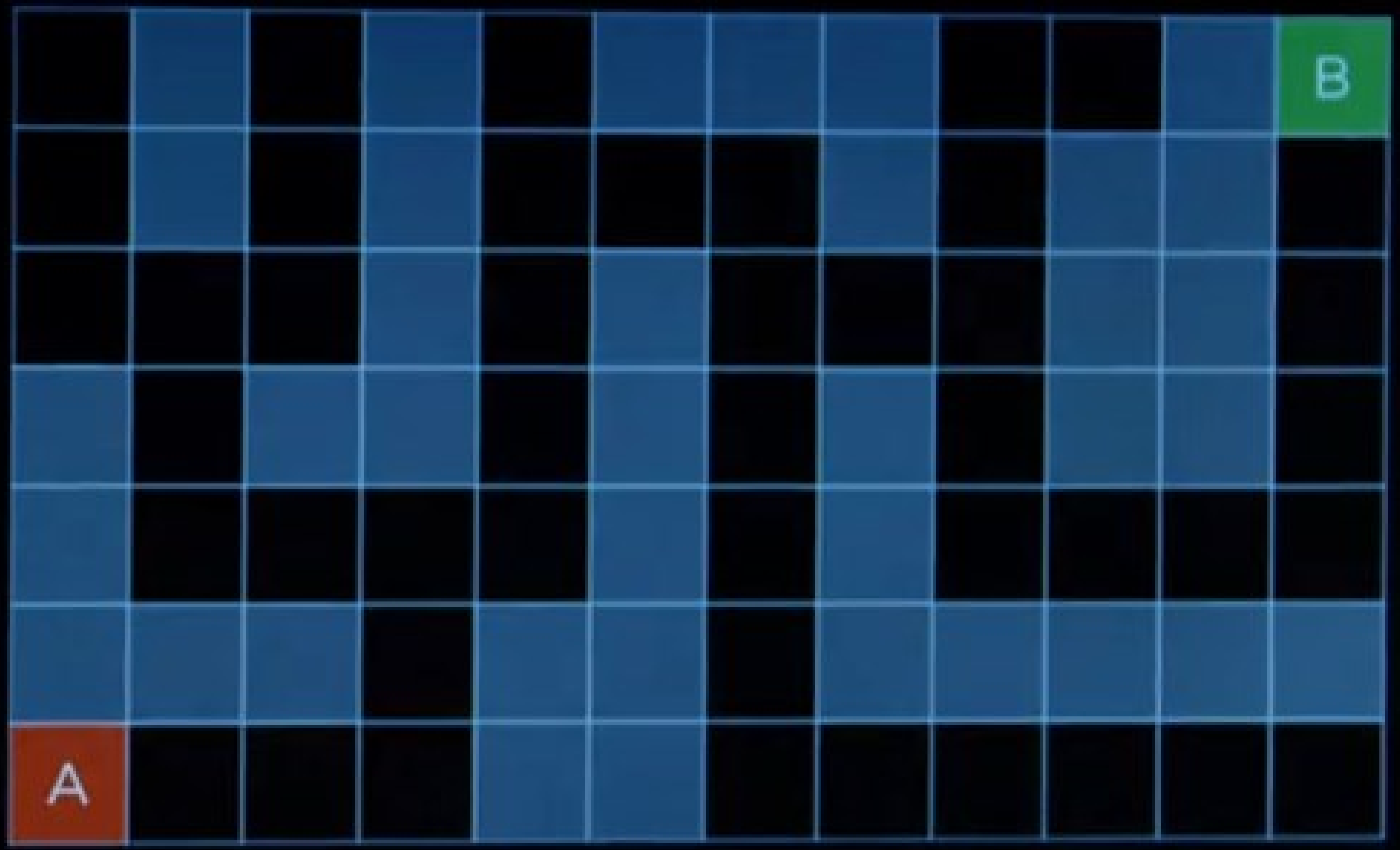
EXPLORED



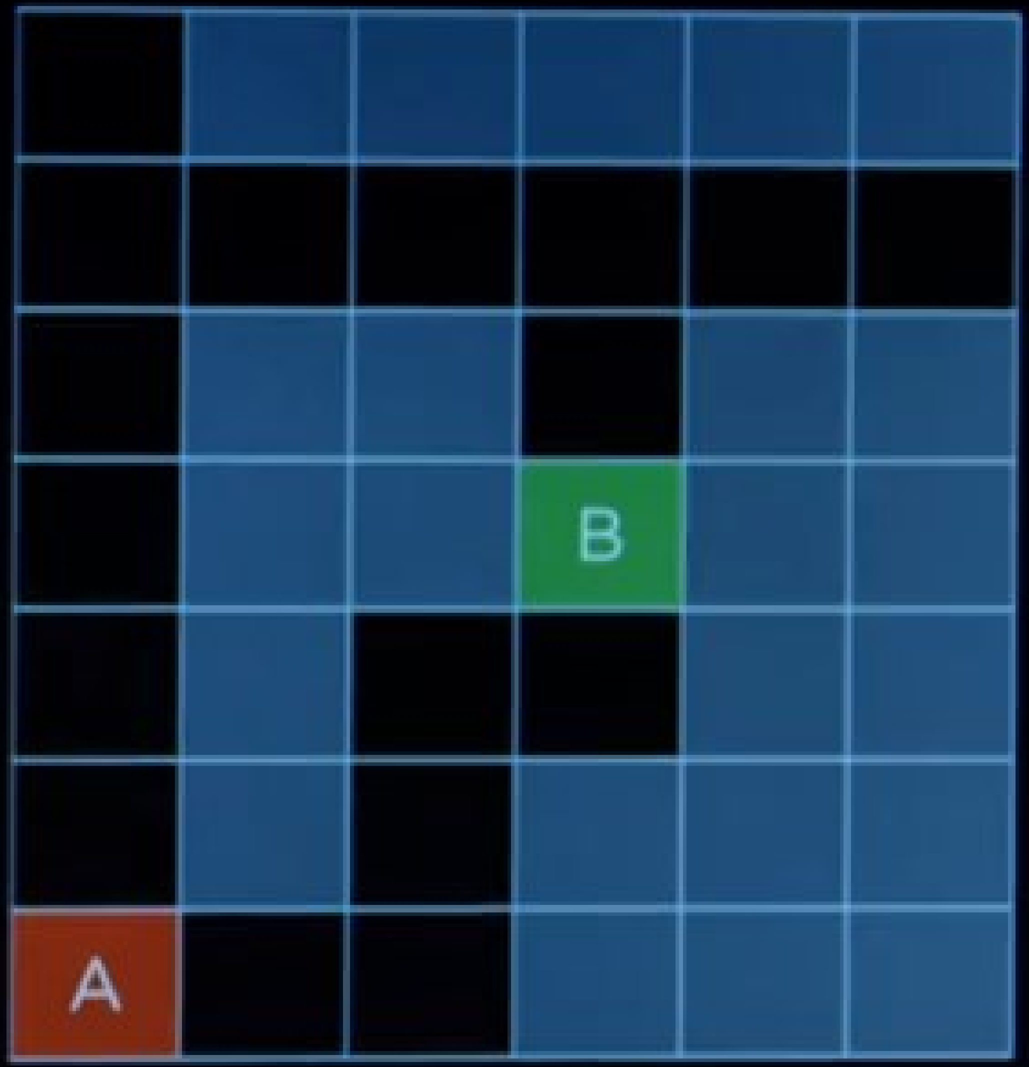
DEPTH-FIRST SEARCH



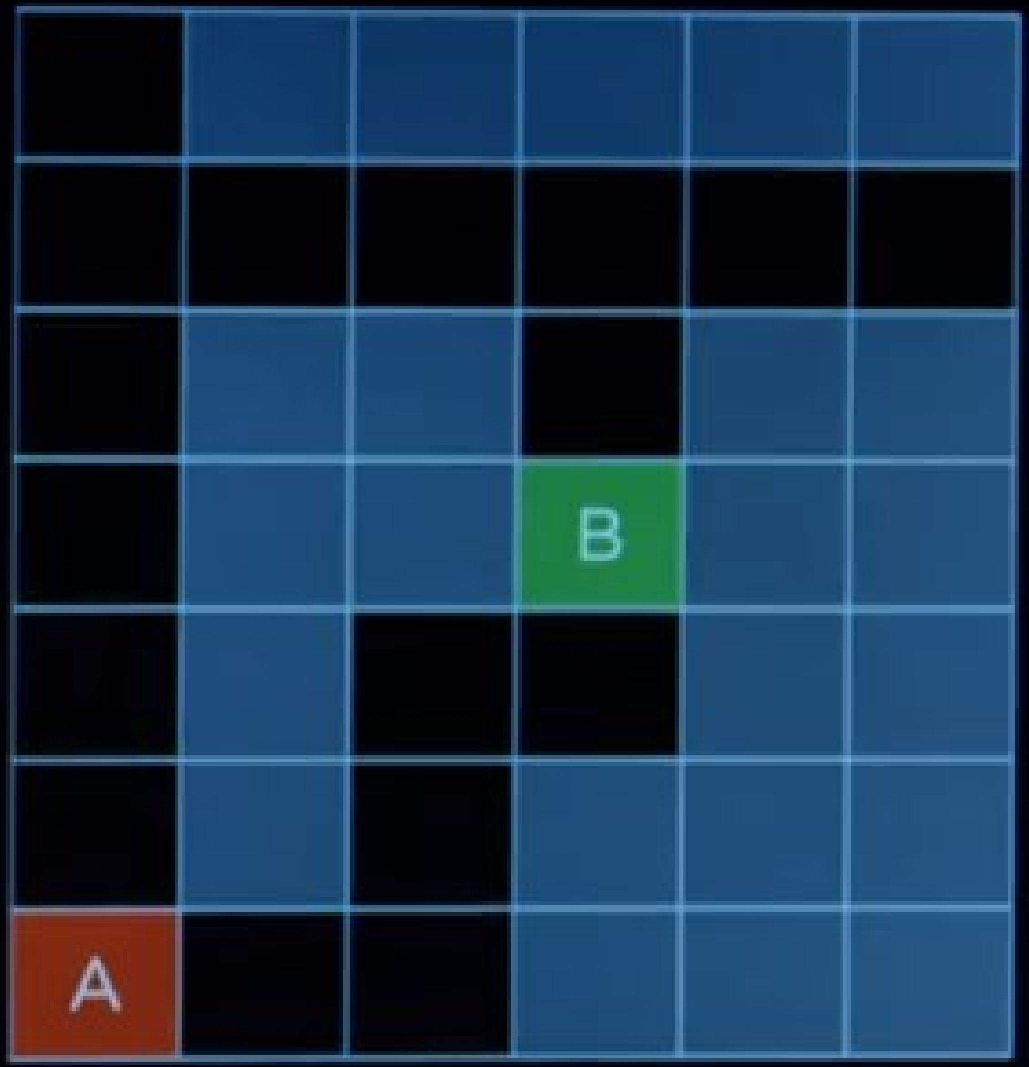
BREADTH-FIRST SEARCH

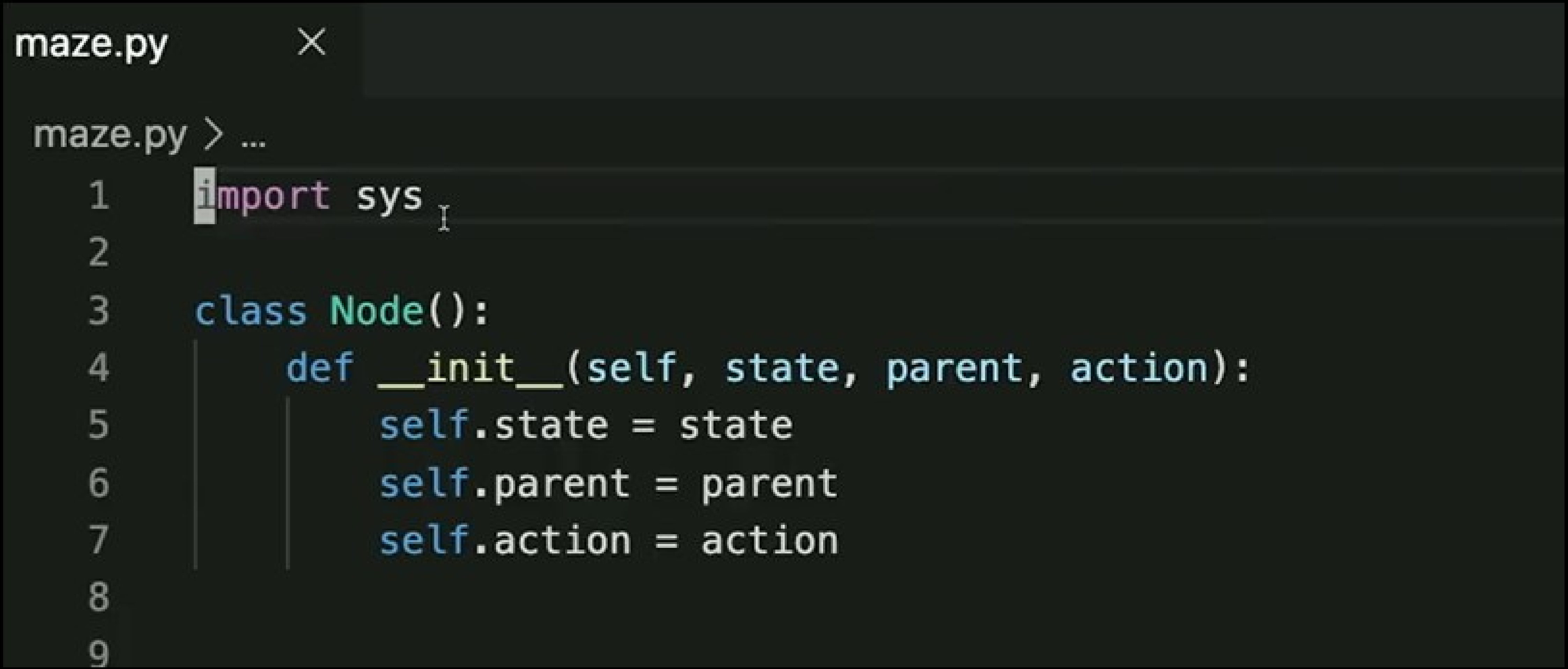


DEPTH-FIRST SEARCH

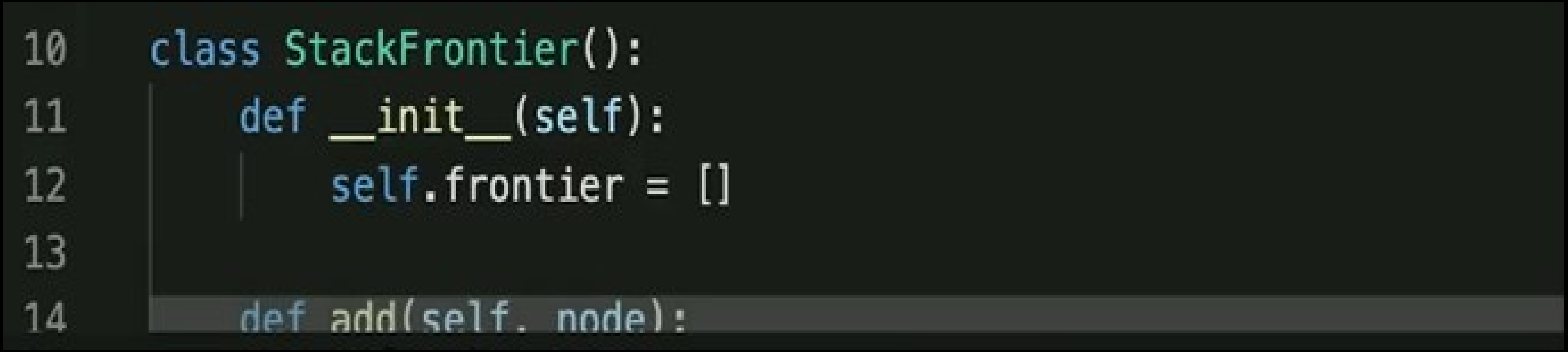


BREADTH-FIRST SEARCH

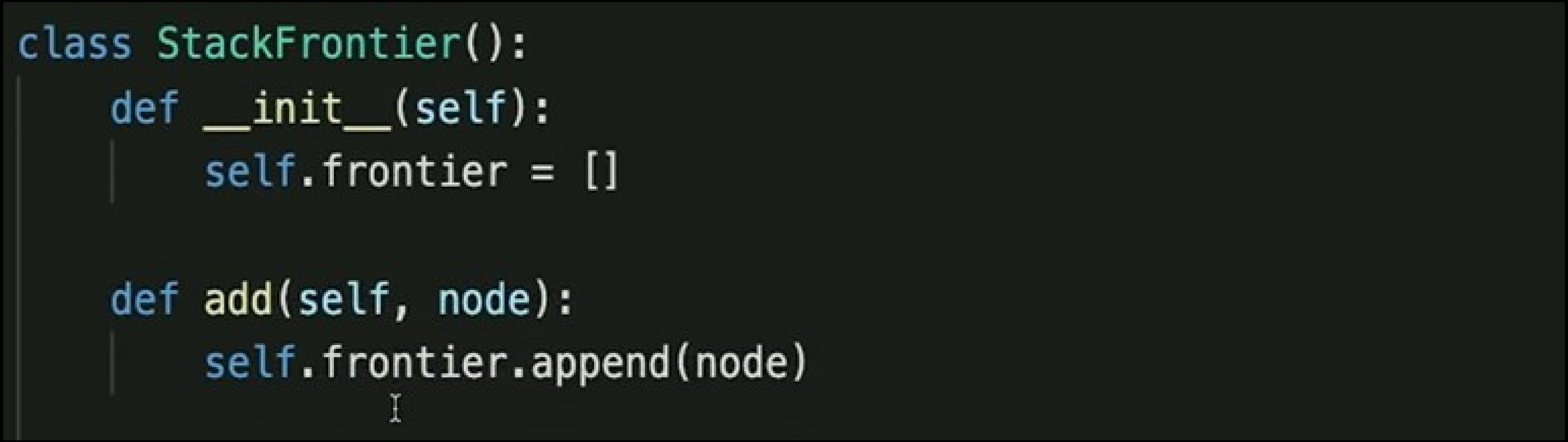




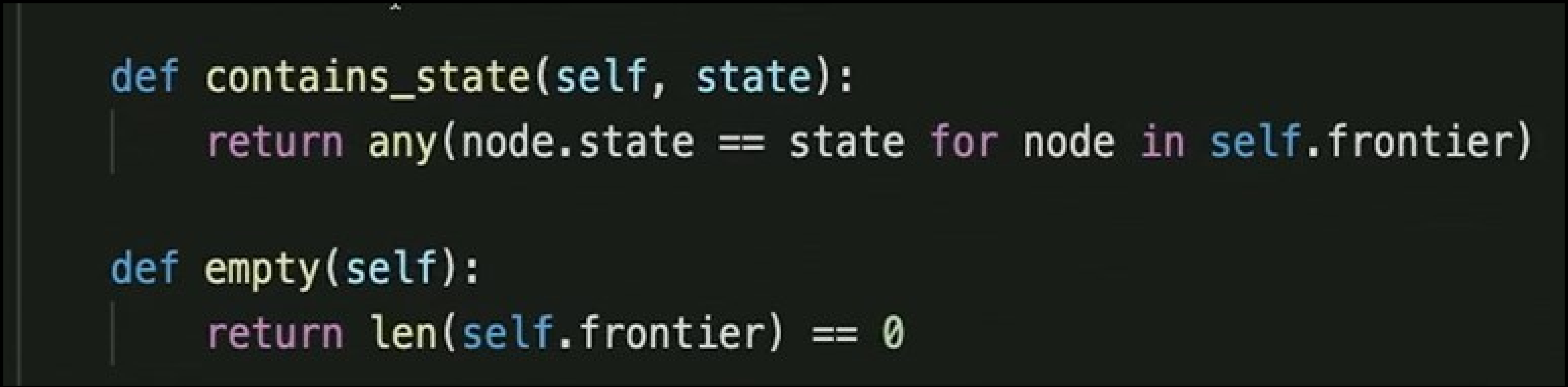
```python
import sys

class Node():
    def __init__(self, state, parent, action):
        self.state = state
        self.parent = parent
        self.action = action


```



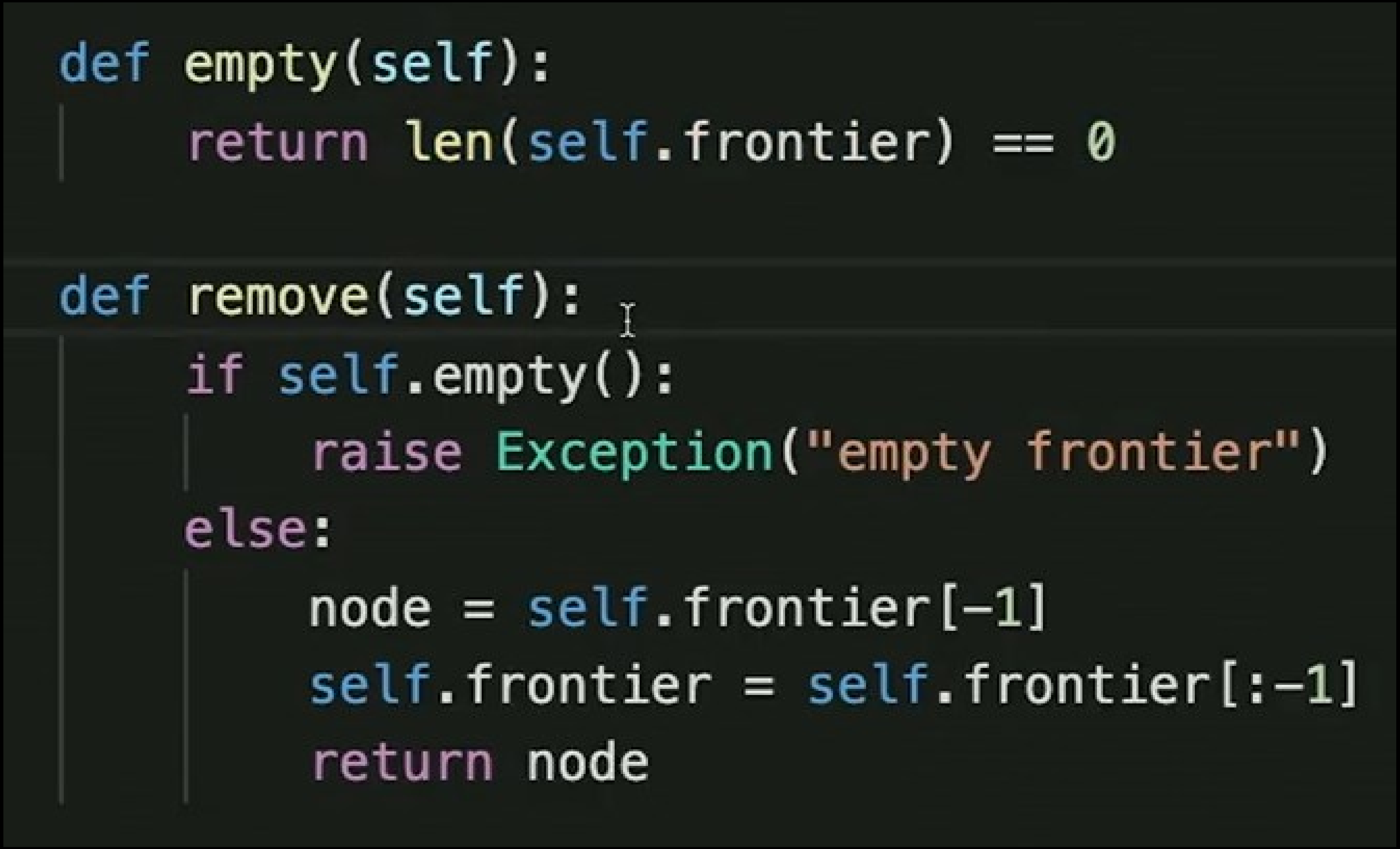
```python
class StackFrontier():
    def __init__(self):
        self.frontier = []

    def add(self, node):
```



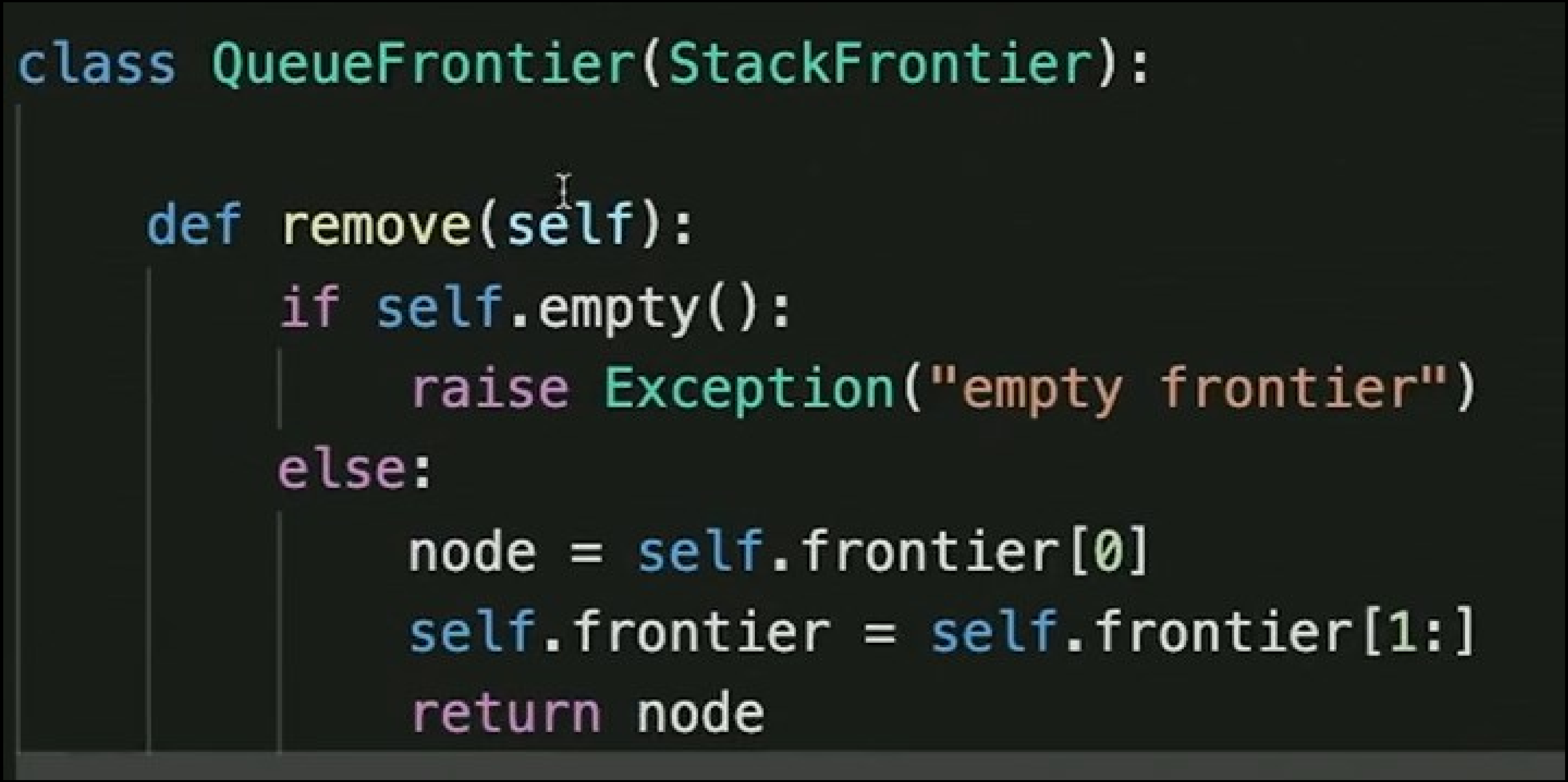
```
class StackFrontier():
    def __init__(self):
        self.frontier = []

    def add(self, node):
        self.frontier.append(node)
```



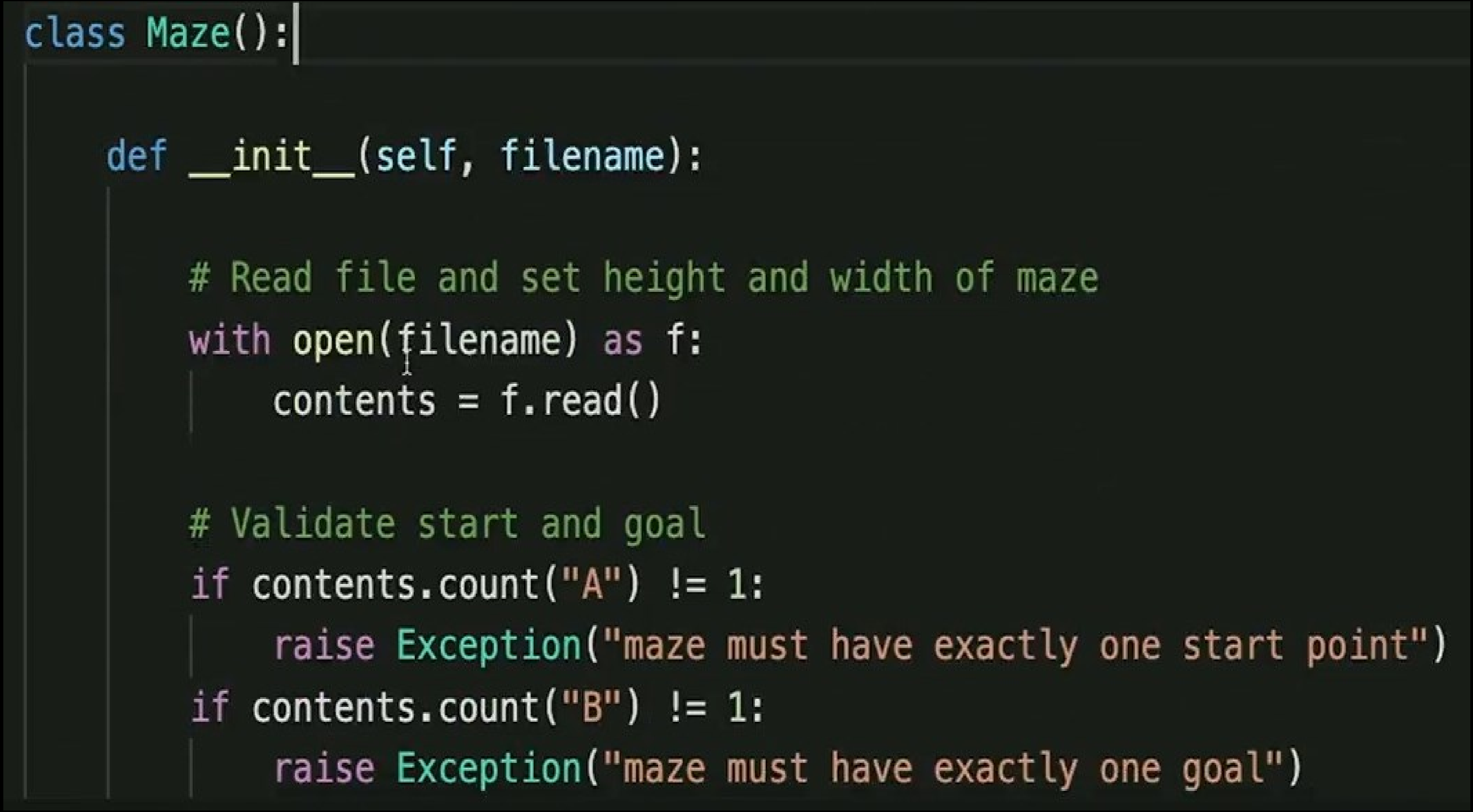
```
    def contains_state(self, state):
        return any(node.state == state for node in self.frontier)

    def empty(self):
        return len(self.frontier) == 0
```



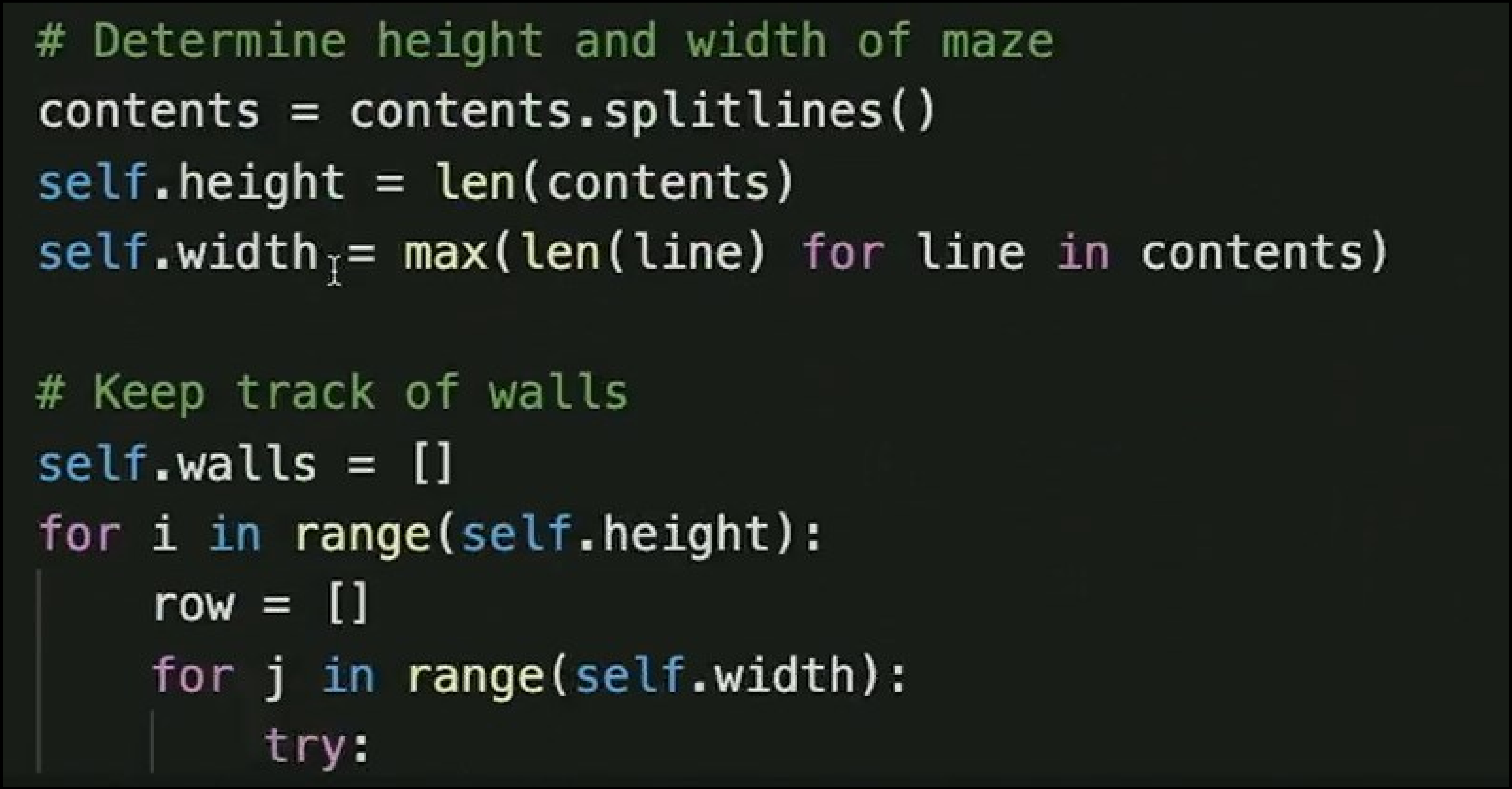
```python
def empty(self):
    return len(self.frontier) == 0

def remove(self):
    if self.empty():
        raise Exception("empty frontier")
    else:
        node = self.frontier[-1]
        self.frontier = self.frontier[:-1]
        return node
```



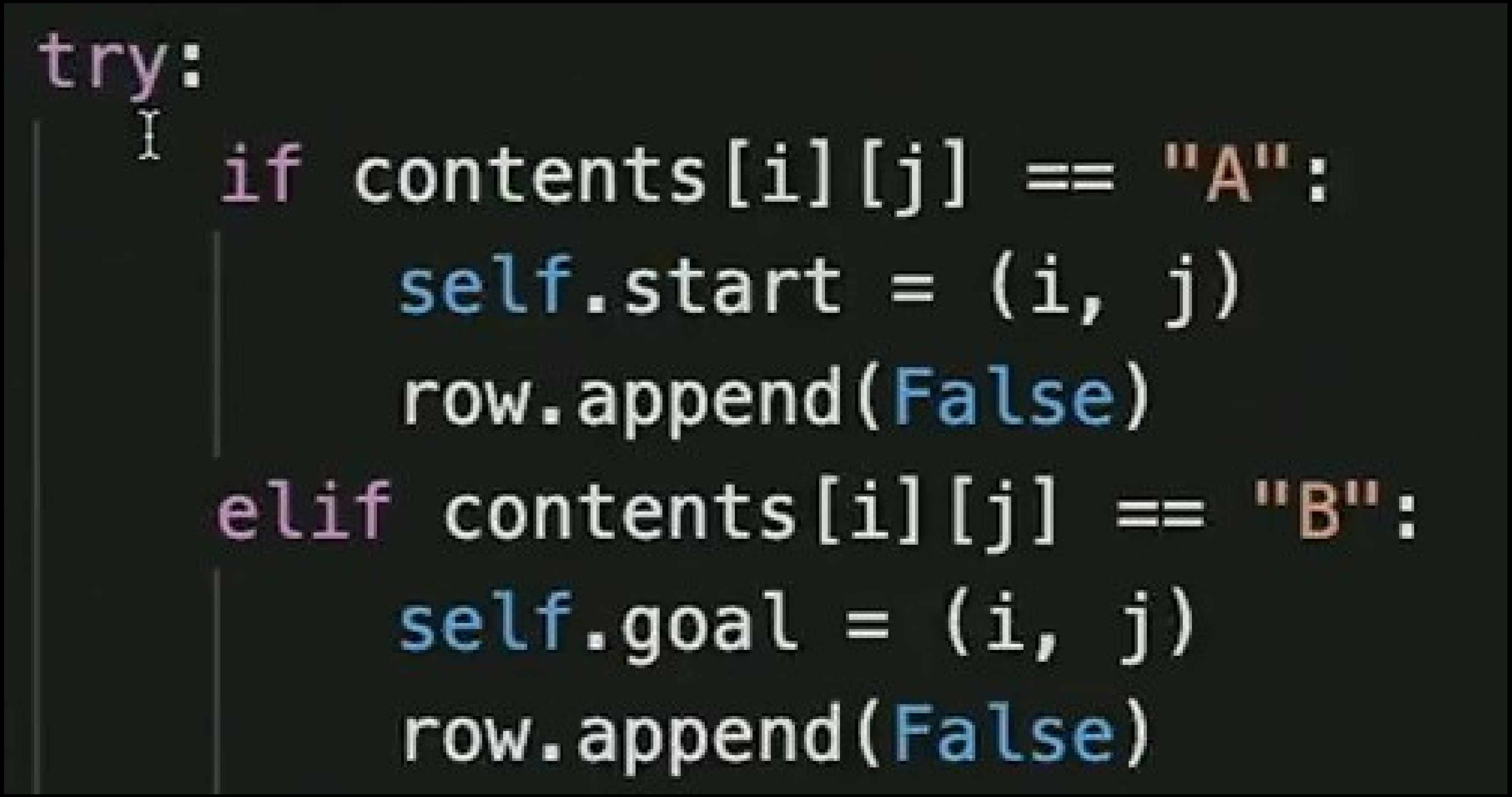
```python
class QueueFrontier(StackFrontier):

    def remove(self):
        if self.empty():
            raise Exception("empty frontier")
        else:
            node = self.frontier[0]
            self.frontier = self.frontier[1:]
            return node
```



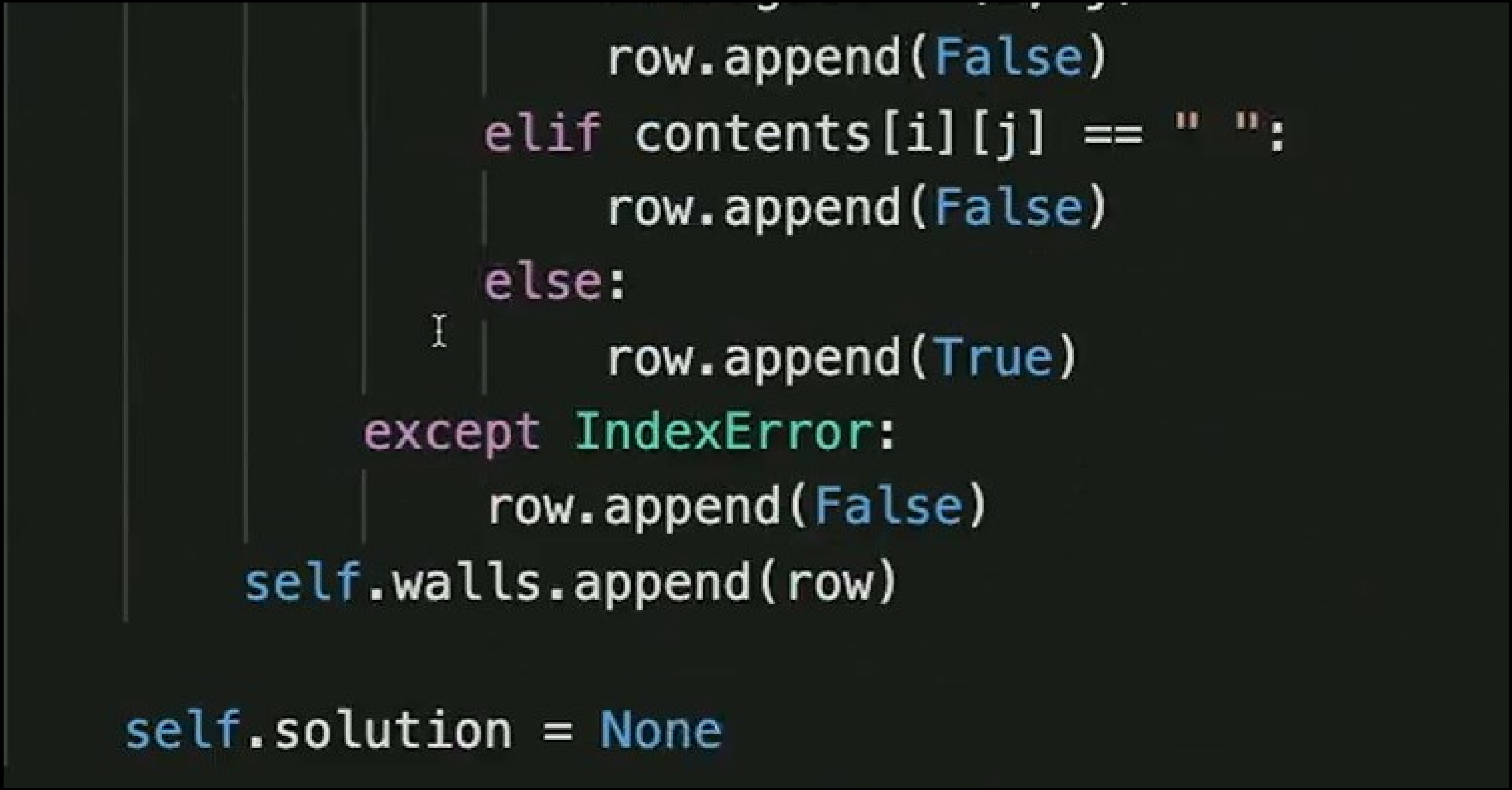
```python
class Maze():

    def __init__(self, filename):

        # Read file and set height and width of maze
        with open(filename) as f:
            contents = f.read()

        # Validate start and goal
        if contents.count("A") != 1:
            raise Exception("maze must have exactly one start point")
        if contents.count("B") != 1:
            raise Exception("maze must have exactly one goal")
```



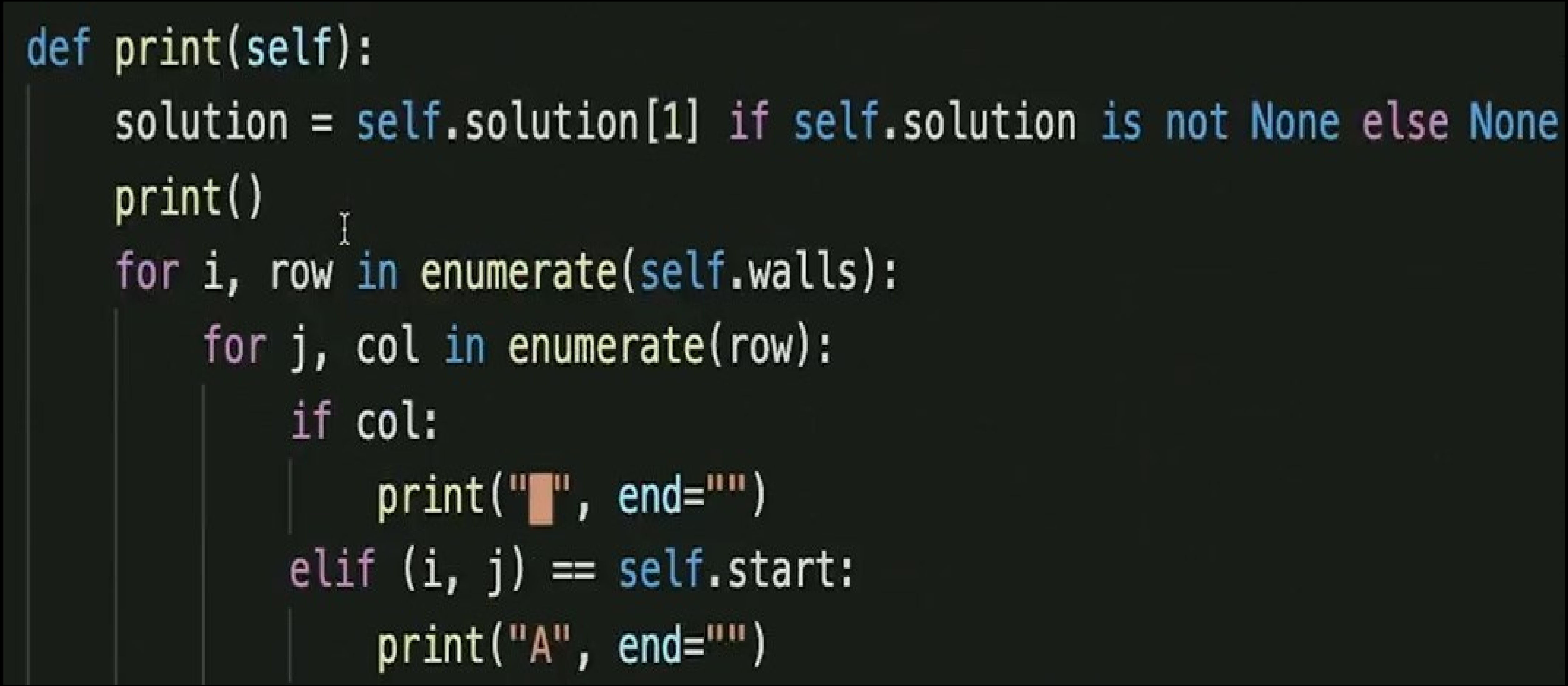
```python
# Determine height and width of maze
contents = contents.splitlines()
self.height = len(contents)
self.width = max(len(line) for line in contents)

# Keep track of walls
self.walls = []
for i in range(self.height):
    row = []
    for j in range(self.width):
        try:
```



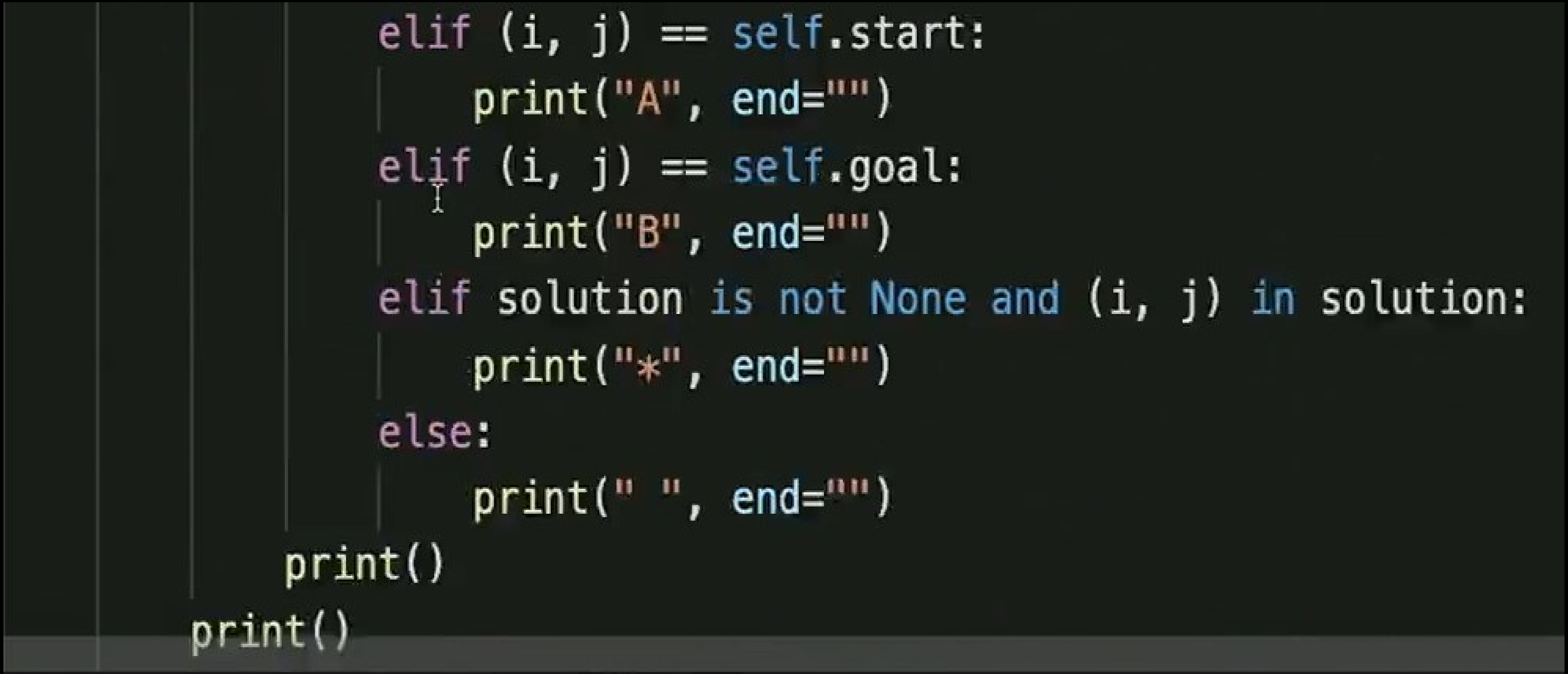
```
try:
    if contents[i][j] == "A":
        self.start = (i, j)
        row.append(False)
    elif contents[i][j] == "B":
        self.goal = (i, j)
        row.append(False)
```



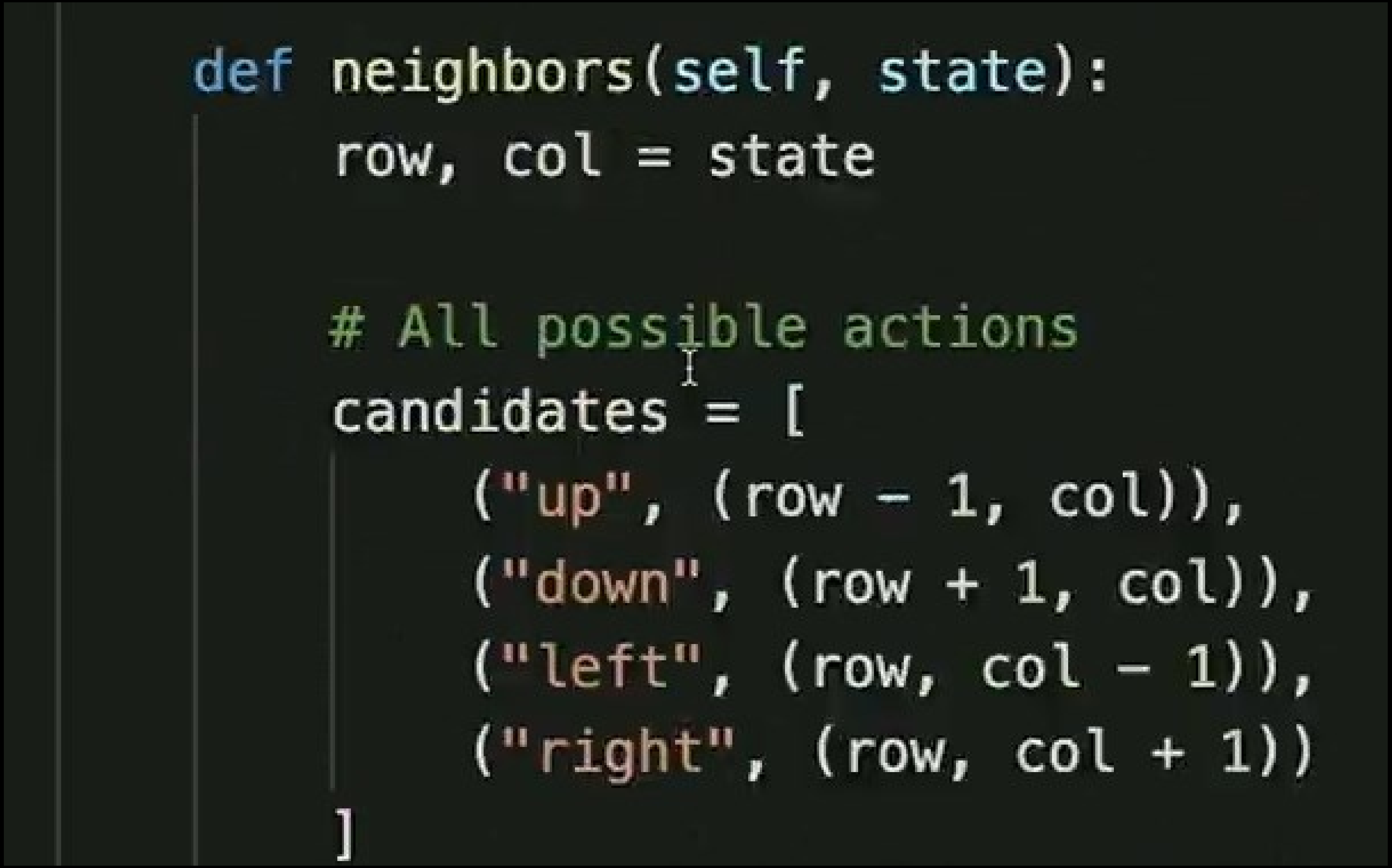
```
                row.append(False)
            elif contents[i][j] == " ":
                row.append(False)
            else:
                row.append(True)
        except IndexError:
            row.append(False)
    self.walls.append(row)

self.solution = None
```



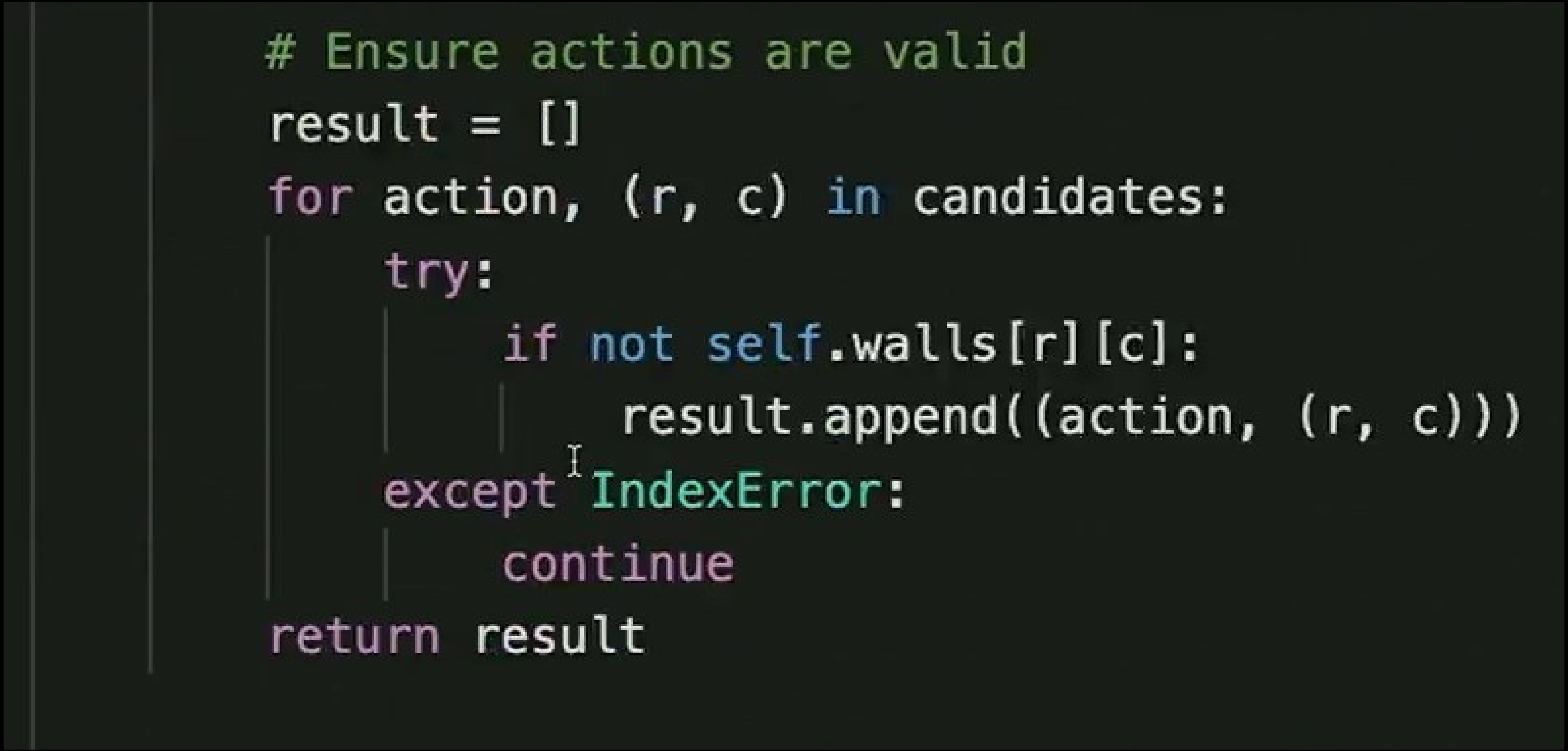
```
def print(self):
    solution = self.solution[1] if self.solution is not None else None
    print()
    for i, row in enumerate(self.walls):
        for j, col in enumerate(row):
            if col:
                print("█", end="")
            elif (i, j) == self.start:
                print("A", end="")
```



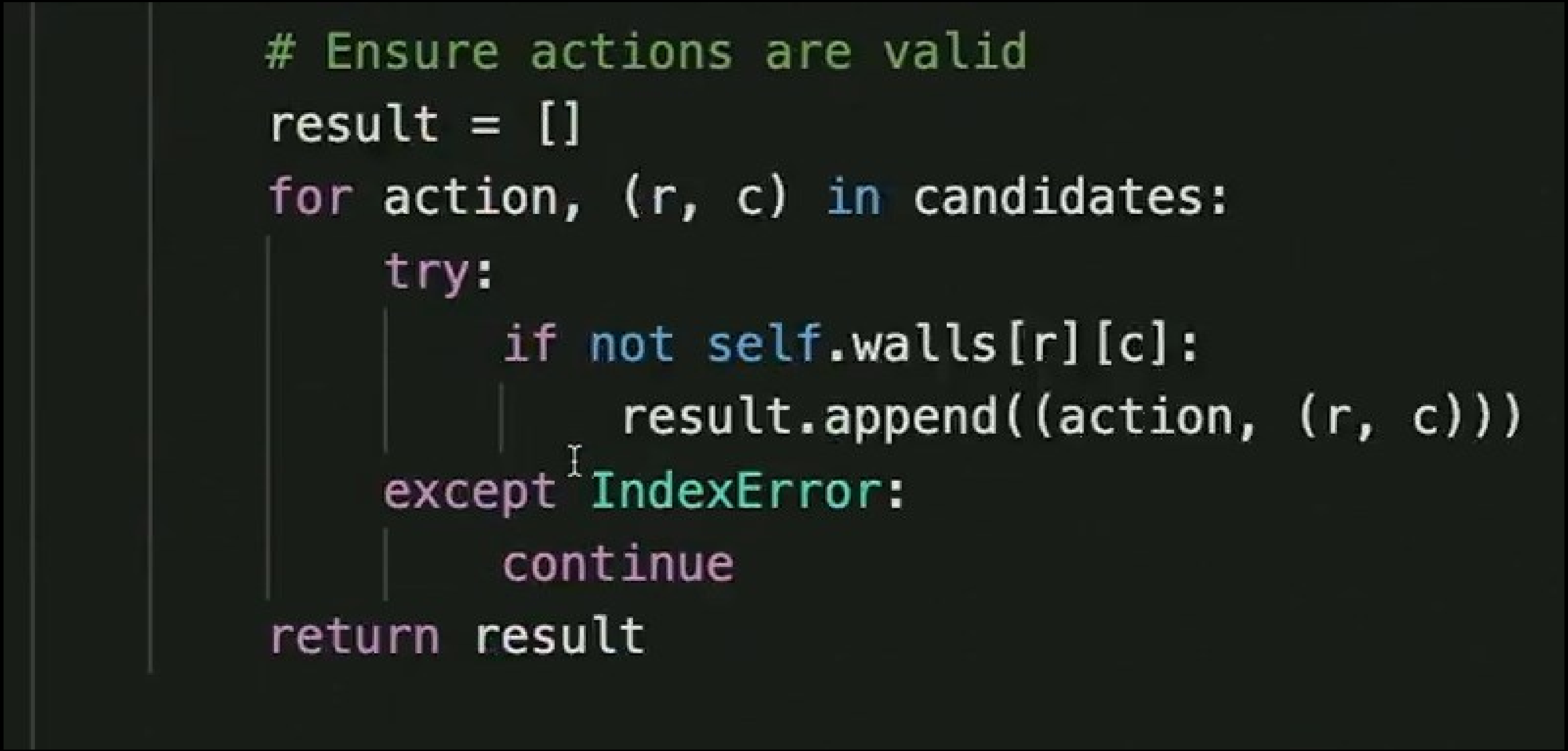
```
                elif (i, j) == self.start:
                    print("A", end="")
                elif (i, j) == self.goal:
                    print("B", end="")
                elif solution is not None and (i, j) in solution:
                    print("*", end="")
                else:
                    print(" ", end="")
            print()
        print()
```



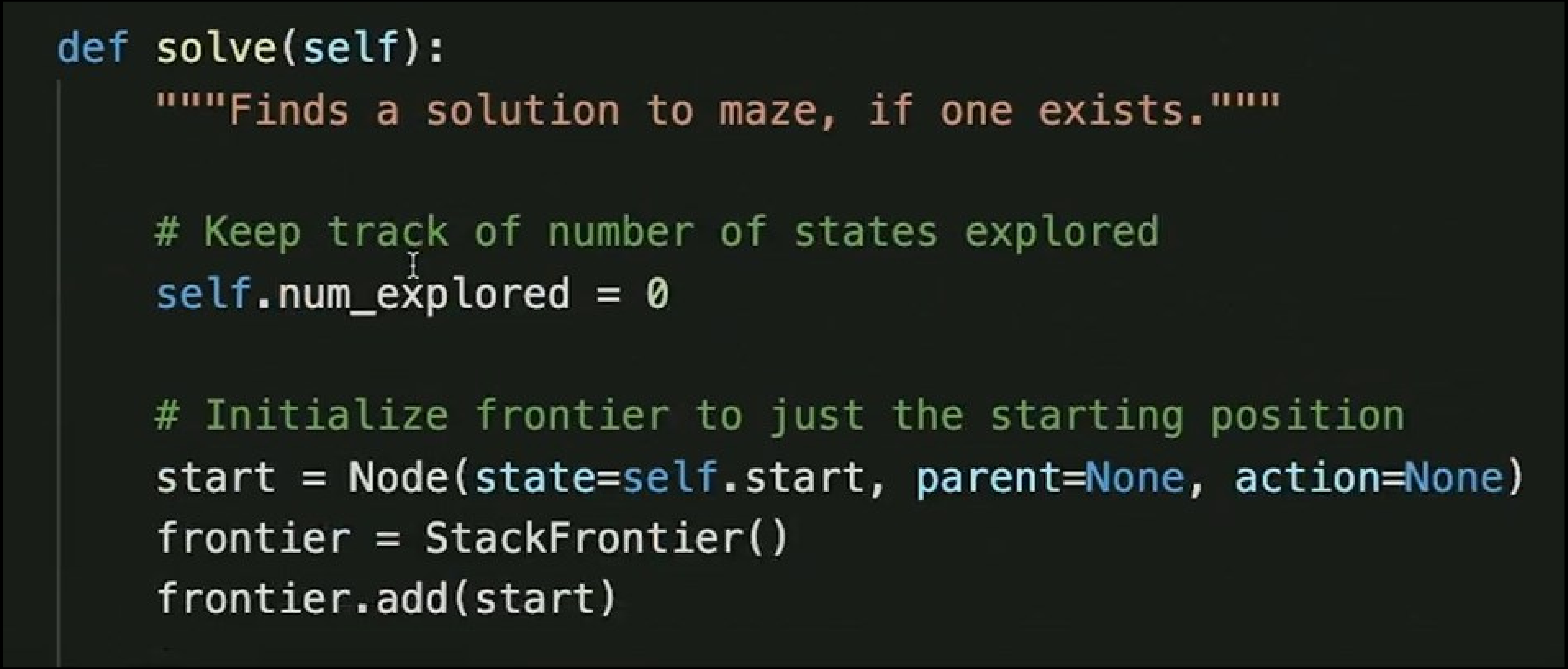
```python
def neighbors(self, state):
    row, col = state

    # All possible actions
    candidates = [
        ("up", (row - 1, col)),
        ("down", (row + 1, col)),
        ("left", (row, col - 1)),
        ("right", (row, col + 1))
    ]
```



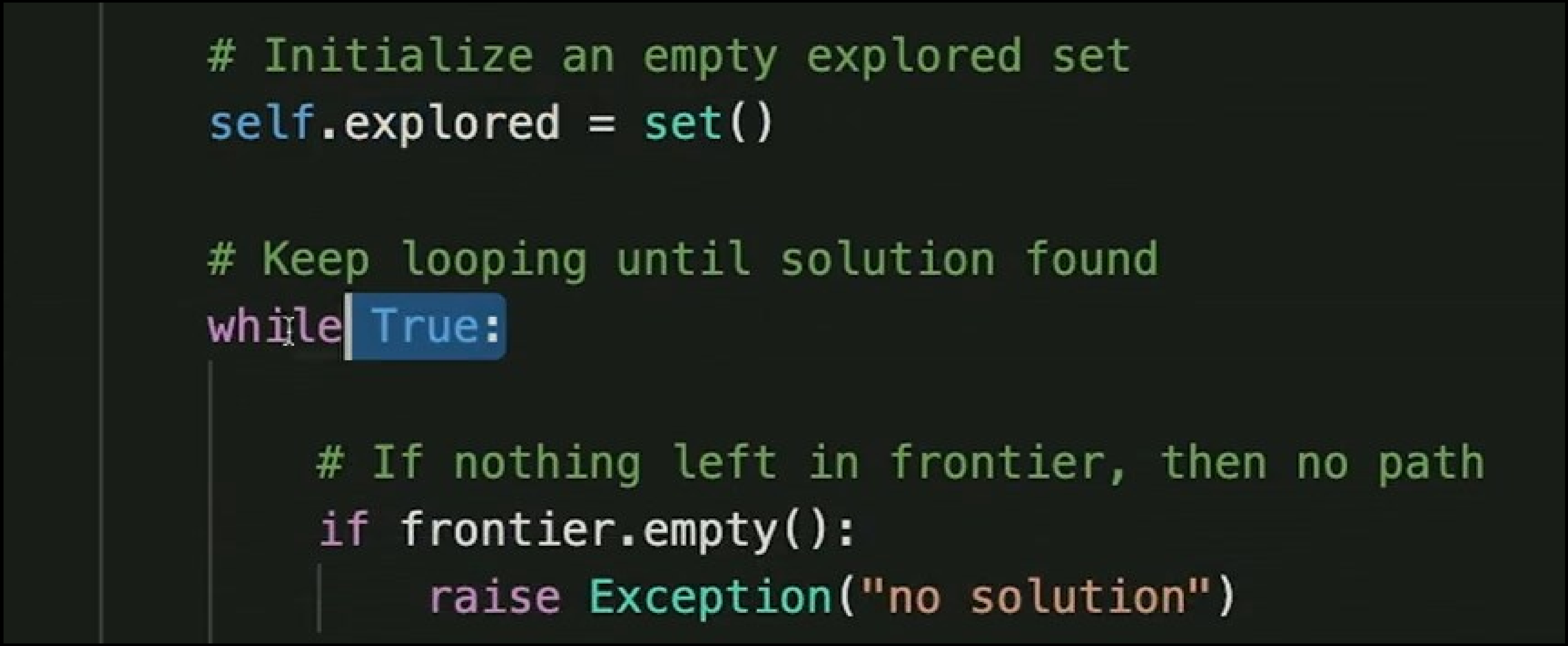
```python
# Ensure actions are valid
result = []
for action, (r, c) in candidates:
    try:
        if not self.walls[r][c]:
            result.append((action, (r, c)))
    except IndexError:
        continue
return result
```



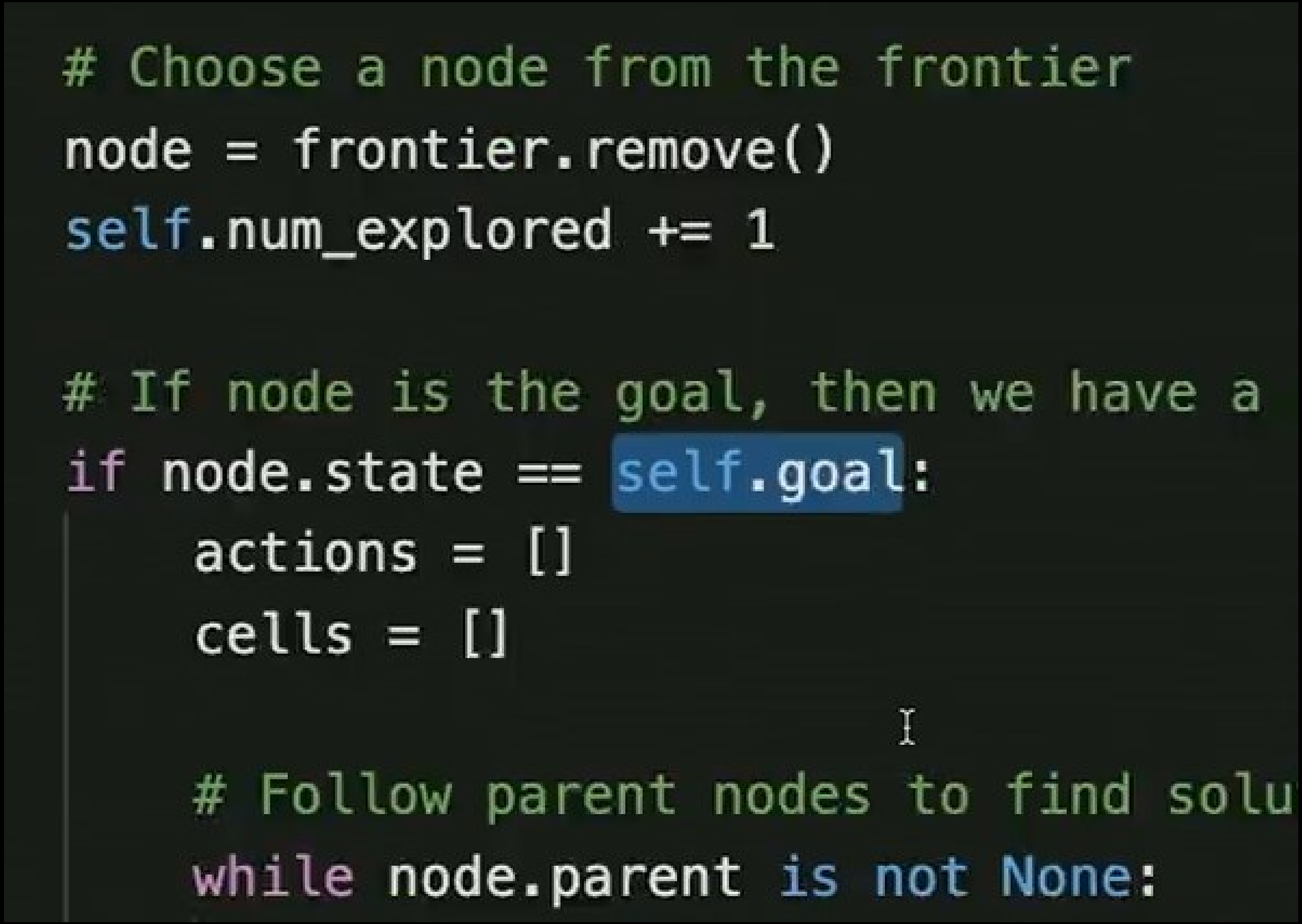
```python
# Ensure actions are valid
result = []
for action, (r, c) in candidates:
    try:
        if not self.walls[r][c]:
            result.append((action, (r, c)))
    except IndexError:
        continue
return result
```



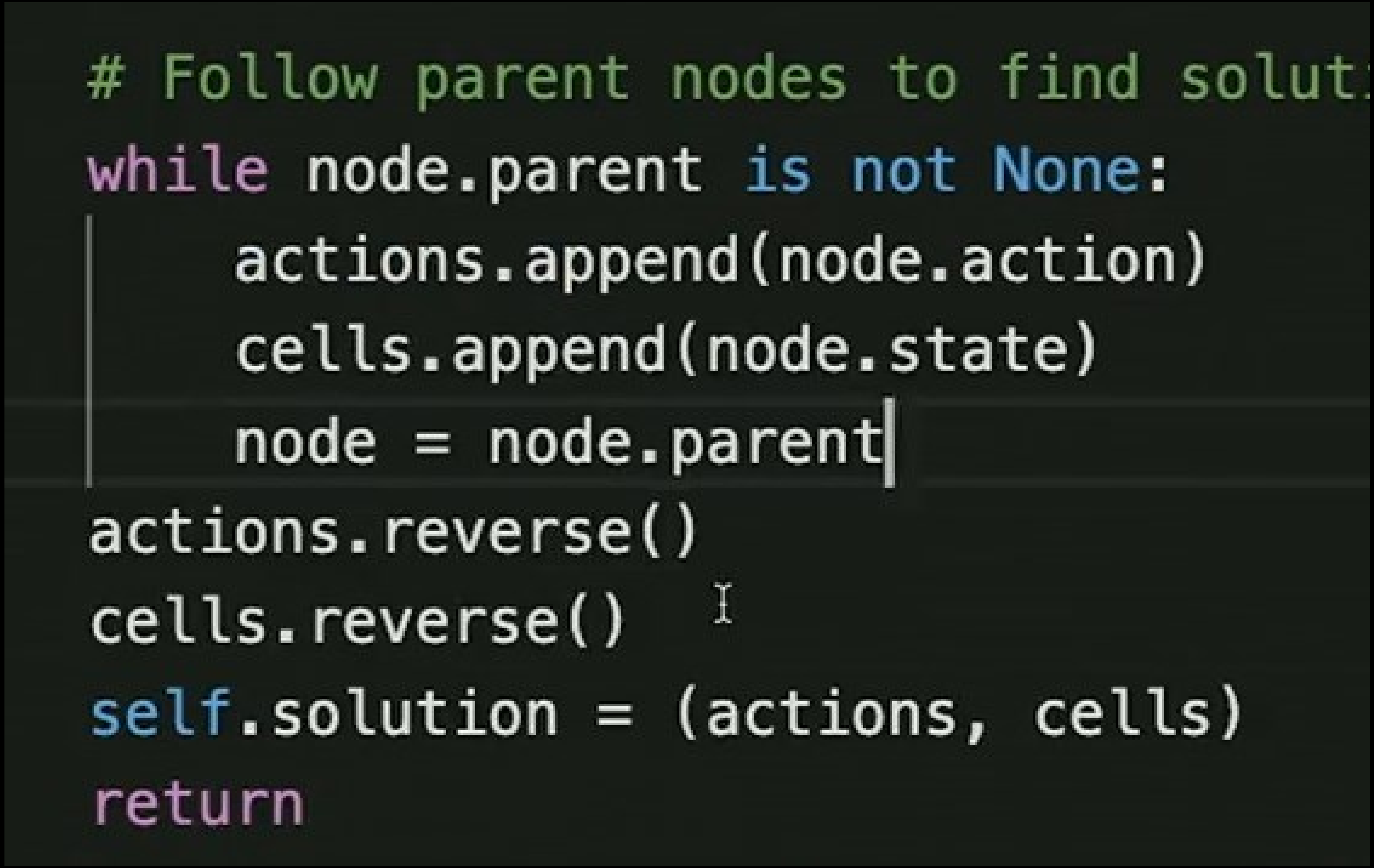
```python
def solve(self):
    """Finds a solution to maze, if one exists."""

    # Keep track of number of states explored
    self.num_explored = 0

    # Initialize frontier to just the starting position
    start = Node(state=self.start, parent=None, action=None)
    frontier = StackFrontier()
    frontier.add(start)
```



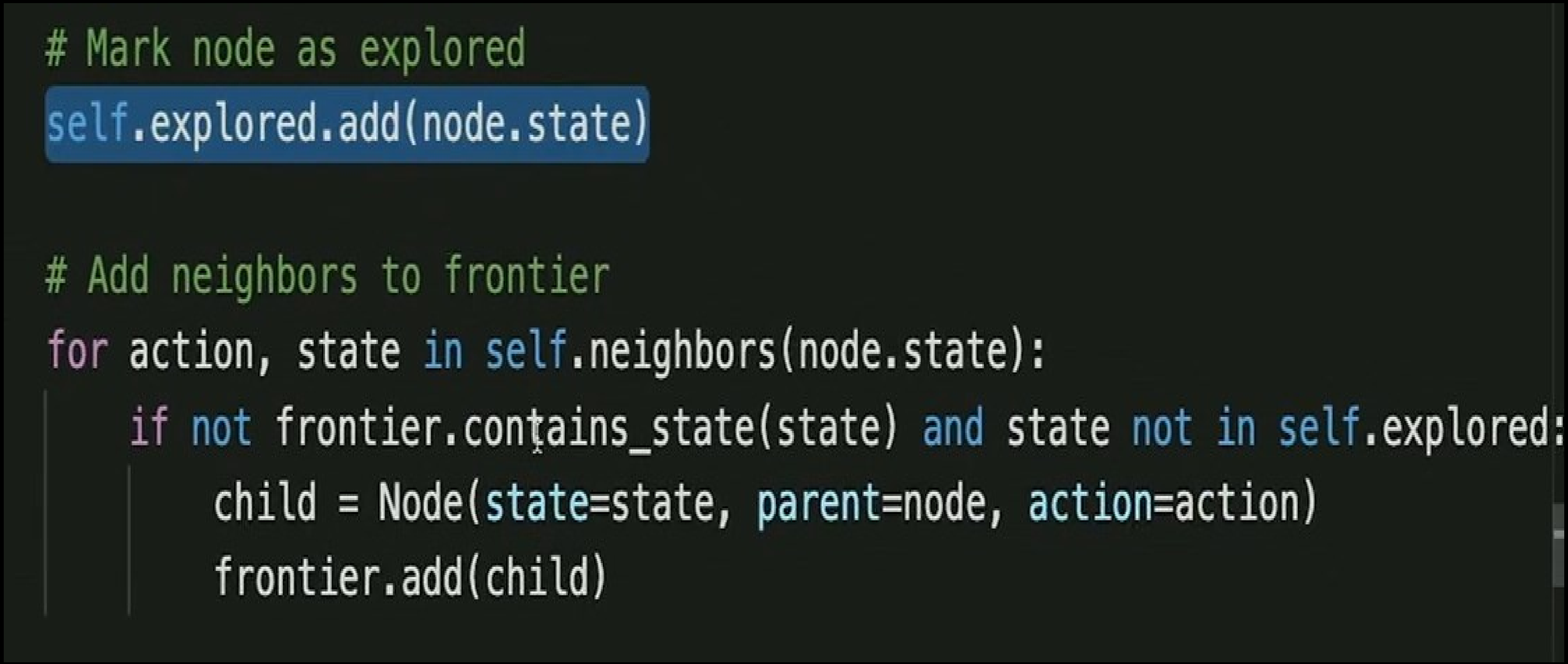
```python
# Initialize an empty explored set
self.explored = set()

# Keep looping until solution found
while True:

    # If nothing left in frontier, then no path
    if frontier.empty():
        raise Exception("no solution")
```



```
# Choose a node from the frontier
node = frontier.remove()
self.num_explored += 1

# If node is the goal, then we have a
if node.state == self.goal:
    actions = []
    cells = []

    # Follow parent nodes to find solu
    while node.parent is not None:
```



```
# Follow parent nodes to find solut
while node.parent is not None:
    actions.append(node.action)
    cells.append(node.state)
    node = node.parent
actions.reverse()
cells.reverse()
self.solution = (actions, cells)
return
```



```
# Mark node as explored
self.explored.add(node.state)

# Add neighbors to frontier
for action, state in self.neighbors(node.state):
    if not frontier.contains_state(state) and state not in self.explored:
        child = Node(state=state, parent=node, action=action)
        frontier.add(child)
```