Why Python?

Python is highly suitable for rapid GUI prototyping, data analysis, GIS (geographic information systems), and machine learning tasks. It has robust libraries for working with spatial data and creating web applications.

Technology	Why	Alternatives	Why not Alternatives
Language	**Python** - Widely used for scientific, simulation, and GIS tasks. - Large ecosystem for mapping and data science. - Easily integrates with visualization and AI tools. - Rapid prototyping, readable syntax. - Used in commercial and academic evacuation systems	Java, C++, MATLAB, R	- Higher complexity for rapid GUIs or data handling. - Less intuitive for beginners. - Slower prototyping. - Smaller number of specialized evacuation packages.
GUI (Graphic User Interface)	**Streamlit** - Very fast to create interactive web apps with Python. - No front-end coding needed. - Excellent for dashboards and real-time feedback. - Easily integrates with mapping libraries.	Flask, Django, Dash, Tkinter	- Flask/Django/Dash need more manual setup and web knowledge. - Dash less intuitive for mapping. - Tkinter not web-based.
Search Algorithm	**A* (A-star)** - Finds optimal routes, adapts fast to changes (blockages, risk). - Well-studied and implemented in evacuation studies and GIS engines . - Uses both travel cost and heuristic for efficiency. - Proven critical in dynamic hazardous scenarios	Dijkstra, Greedy, Custom Heuristics	- Dijkstra is slower for large networks without heuristic. - Greedy misses safest/shortest routes if hazards change. - Custom algorithms less tested, may not guarantee optimal paths in emergencies.

CODE (Overview)

1. Import Statements

import streamlit as st - Loads Streamlit, which is used to create the interactive web user interface.

import folium - A mapping library for rendering geographical maps and paths.

from streamlit_folium import st_folium - Allows Folium maps to be embedded in Streamlit apps for user interaction.

import xml.etree.ElementTree as ET - Handles XML parsing—specifically for extracting route data from KML files.

import numpy as np - Used for numeric operations, especially for fuzzy logic calculations.

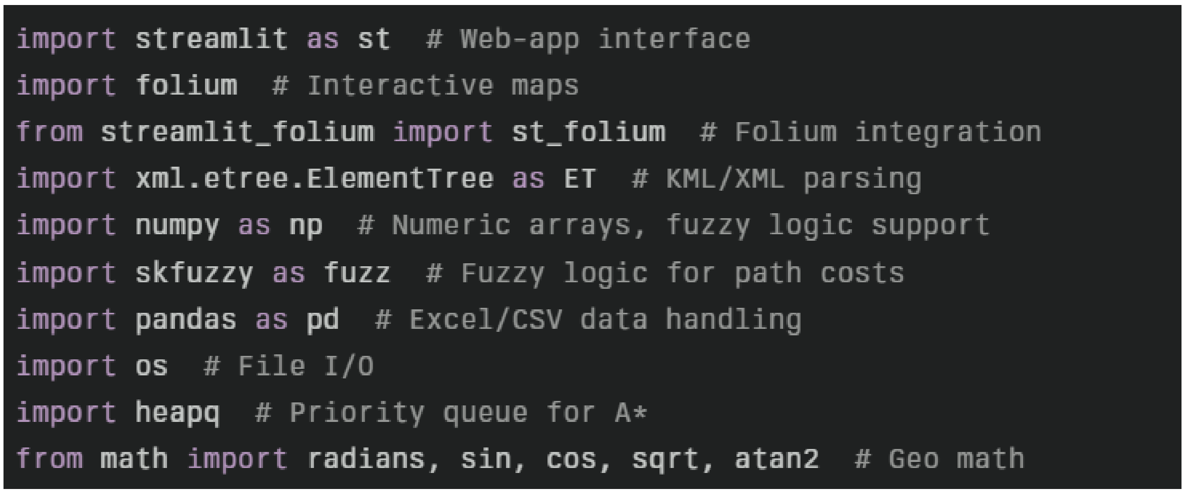
import skfuzzy as fuzz - Provides fuzzy logic functions to rate path difficulty based on slope, travel time, and curvature.

import pandas as pd - Reads Excel files containing travel segment data.

import os - Used for file and directory handling.

import heapq - Implements a priority queue needed by A* pathfinding.

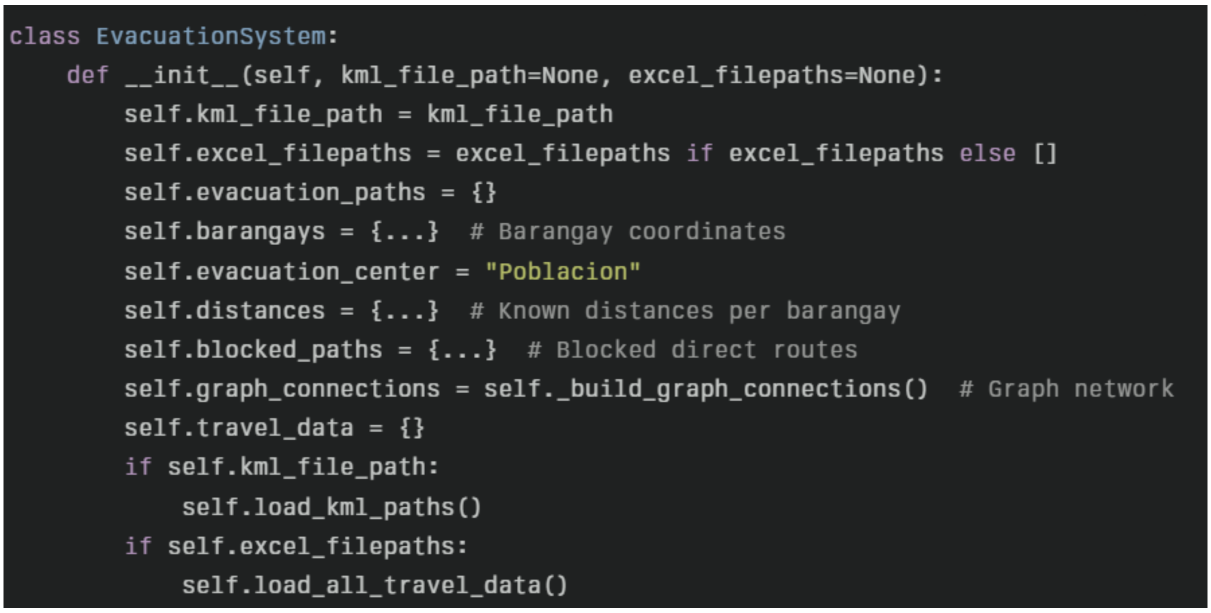
from math import radians, sin, cos, sqrt, atan2 - Math functions for accurate geodesic distance calculations.



```
import streamlit as st  # Web-app interface
import folium  # Interactive maps
from streamlit_folium import st_folium  # Folium integration
import xml.etree.ElementTree as ET  # KML/XML parsing
import numpy as np  # Numeric arrays, fuzzy logic support
import skfuzzy as fuzz  # Fuzzy logic for path costs
import pandas as pd  # Excel/CSV data handling
import os  # File I/O
import heapq  # Priority queue for A*
from math import radians, sin, cos, sqrt, atan2  # Geo math
```

2. EvacuationSystem Class Initialization

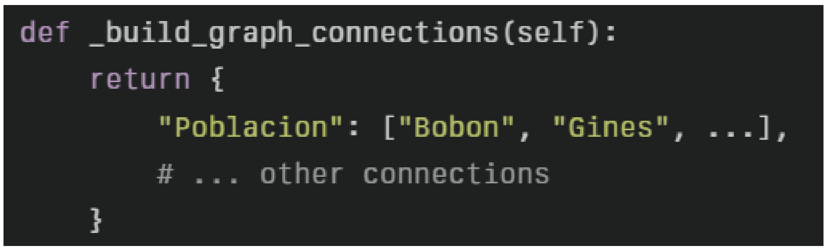
- Stores file paths, prepares data structures, loads barangay coordinates, edge distances, blocked paths, and sets up the routing graph.
- On startup, calls functions to load KML paths and Excel travel data if the sources are available.



```python
class EvacuationSystem:
    def __init__(self, kml_file_path=None, excel_filepaths=None):
        self.kml_file_path = kml_file_path
        self.excel_filepaths = excel_filepaths if excel_filepaths else []
        self.evacuation_paths = {}
        self.barangays = {...}  # Barangay coordinates
        self.evacuation_center = "Poblacion"
        self.distances = {...}  # Known distances per barangay
        self.blocked_paths = {...}  # Blocked direct routes
        self.graph_connections = self._build_graph_connections()  # Graph network
        self.travel_data = {}
        if self.kml_file_path:
            self.load_kml_paths()
        if self.excel_filepaths:
            self.load_all_travel_data()
```

3. Graph Structure Method

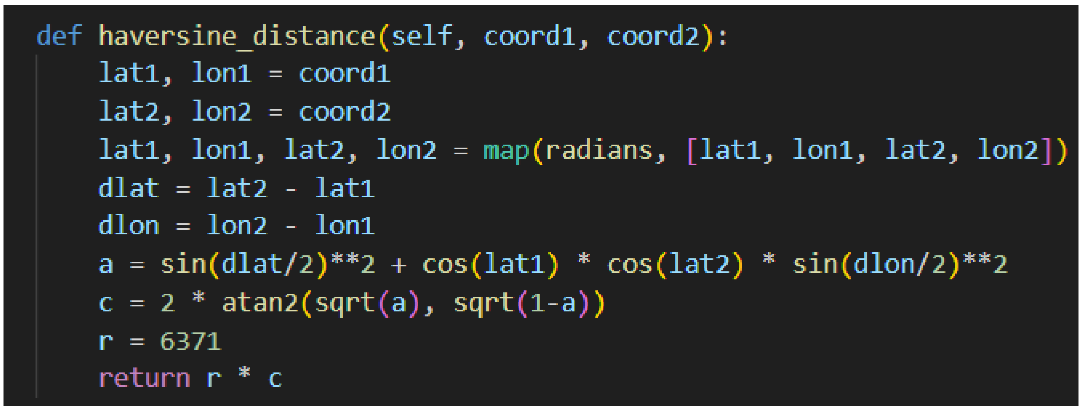
- _build_graph_connections: Defines how each barangay is connected to its neighbors. This adjacency dictionary forms the graph the pathfinding algorithm will traverse.



```python
def _build_graph_connections(self):
    return {
        "Poblacion": ["Bobon", "Gines", ...],
        # ... other connections
    }
```

4. Geo Distance Calculation

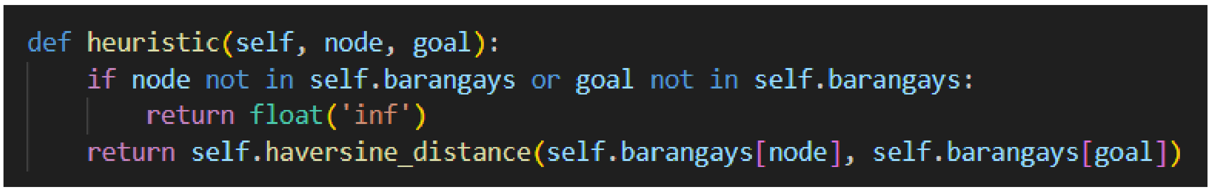
- haversine_distance: Computes distance between two sets of latitude/longitude using the haversine formula (returns value in kilometers). Used for both path cost and as a heuristic for A*.



```python
def haversine_distance(self, coord1, coord2):
    lat1, lon1 = coord1
    lat2, lon2 = coord2
    lat1, lon1, lat2, lon2 = map(radians, [lat1, lon1, lat2, lon2])
    dlat = lat2 - lat1
    dlon = lon2 - lon1
    a = sin(dlat/2)**2 + cos(lat1) * cos(lat2) * sin(dlon/2)**2
    c = 2 * atan2(sqrt(a), sqrt(1-a))
    r = 6371
    return r * c
```

5. A Heuristic Function*

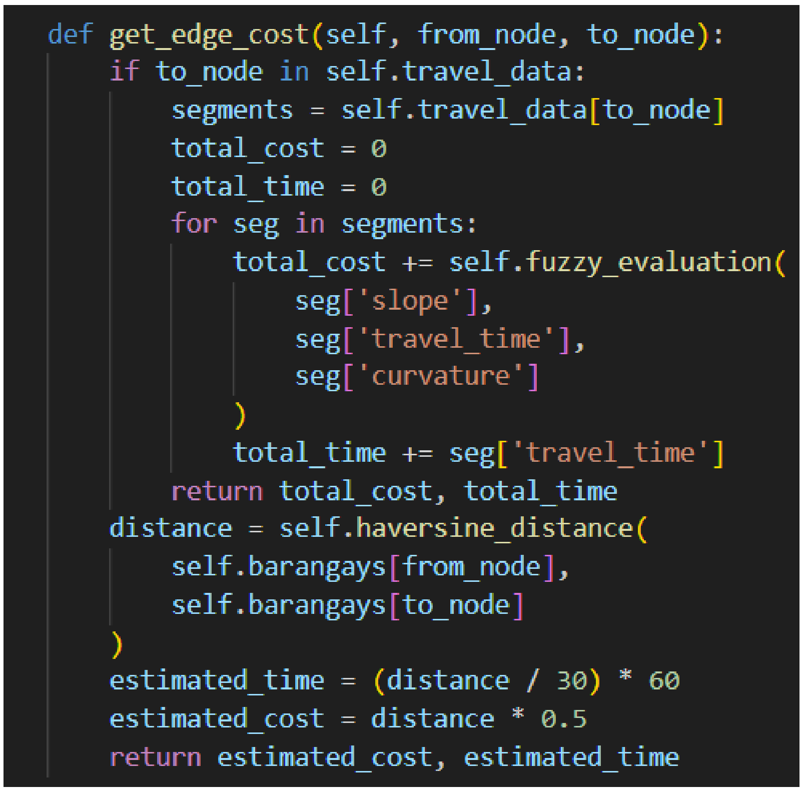
- heuristic: Uses the haversine distance to estimate the cost to reach the goal from a given node in the graph. Helps A* pick promising nodes first.



```python
def heuristic(self, node, goal):
    if node not in self.barangays or goal not in self.barangays:
        return float('inf')
    return self.haversine_distance(self.barangays[node], self.barangays[goal])
```

6. Edge Cost Function

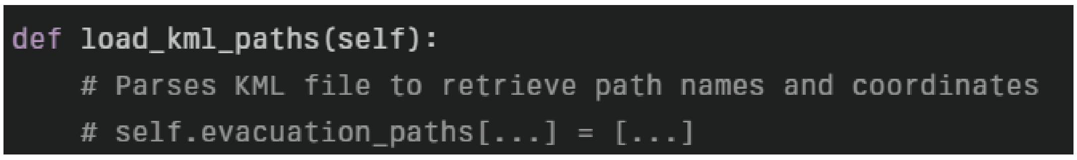
- get_edge_cost: Calculates the cost and travel time between two barangays either from real data (Excel, using fuzzy logic) or by estimation based on straight-line distance. These costs help rate which paths are safest/fastest.



```python
def get_edge_cost(self, from_node, to_node):
    if to_node in self.travel_data:
        segments = self.travel_data[to_node]
        total_cost = 0
        total_time = 0
        for seg in segments:
            total_cost += self.fuzzy_evaluation(
                seg['slope'],
                seg['travel_time'],
                seg['curvature']
            )
            total_time += seg['travel_time']
        return total_cost, total_time
    distance = self.haversine_distance(
        self.barangays[from_node],
        self.barangays[to_node]
    )
    estimated_time = (distance / 30) * 60
    estimated_cost = distance * 0.5
    return estimated_cost, estimated_time
```

7. KML Loading

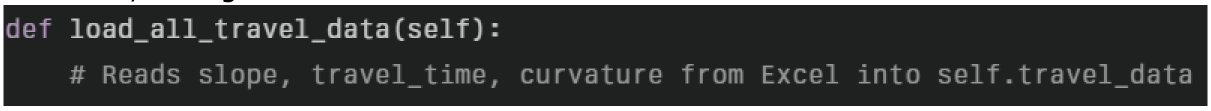
- load_kml_paths: Parses the evacuation routes from a KML file, extracting each path's name and associated coordinates. This information is used for both map visualization and path analysis.



```python
def load_kml_paths(self):
    # Parses KML file to retrieve path names and coordinates
    # self.evacuation_paths[...] = [...]
```

8. Excel Travel Data Loading

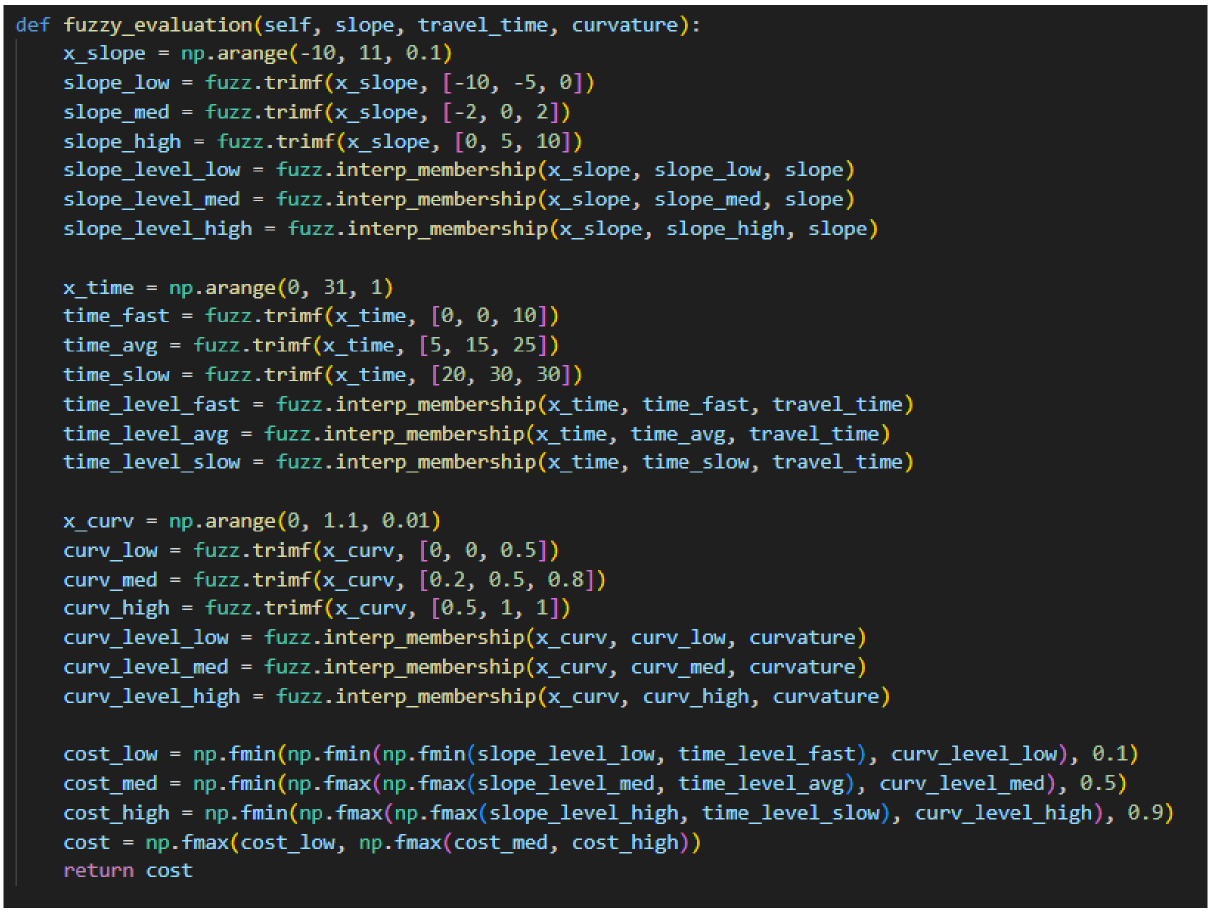
- load_all_travel_data: Reads each segment's slope, travel time, and curvature from Excel files, storing the detailed route data for cost calculation.



```python
def load_all_travel_data(self):
    # Reads slope, travel_time, curvature from Excel into self.travel_data
```

9. Fuzzy Logic Evaluation

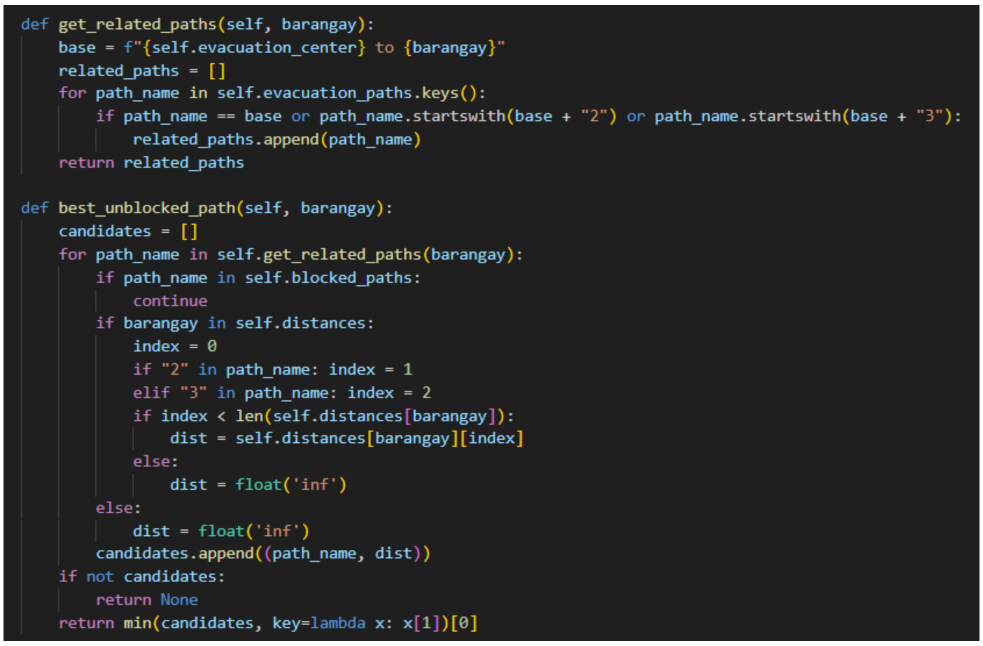
- fuzzy_evaluation: Rates how difficult each segment is based on slope (hilliness), travel time, and curvature using fuzzy logic rules.



```python
def fuzzy_evaluation(self, slope, travel_time, curvature):
    x_slope = np.arange(-10, 11, 0.1)
    slope_low = fuzz.trimf(x_slope, [-10, -5, 0])
    slope_med = fuzz.trimf(x_slope, [-2, 0, 2])
    slope_high = fuzz.trimf(x_slope, [0, 5, 10])
    slope_level_low = fuzz.interp_membership(x_slope, slope_low, slope)
    slope_level_med = fuzz.interp_membership(x_slope, slope_med, slope)
    slope_level_high = fuzz.interp_membership(x_slope, slope_high, slope)

    x_time = np.arange(0, 31, 1)
    time_fast = fuzz.trimf(x_time, [0, 0, 10])
    time_avg = fuzz.trimf(x_time, [5, 15, 25])
    time_slow = fuzz.trimf(x_time, [20, 30, 30])
    time_level_fast = fuzz.interp_membership(x_time, time_fast, travel_time)
    time_level_avg = fuzz.interp_membership(x_time, time_avg, travel_time)
    time_level_slow = fuzz.interp_membership(x_time, time_slow, travel_time)

    x_curv = np.arange(0, 1.1, 0.01)
    curv_low = fuzz.trimf(x_curv, [0, 0, 0.5])
    curv_med = fuzz.trimf(x_curv, [0.2, 0.5, 0.8])
    curv_high = fuzz.trimf(x_curv, [0.5, 1, 1])
    curv_level_low = fuzz.interp_membership(x_curv, curv_low, curvature)
    curv_level_med = fuzz.interp_membership(x_curv, curv_med, curvature)
    curv_level_high = fuzz.interp_membership(x_curv, curv_high, curvature)

    cost_low = np.fmin(np.fmin(np.fmin(slope_level_low, time_level_fast), curv_level_low), 0.1)
    cost_med = np.fmin(np.fmax(np.fmax(slope_level_med, time_level_avg), curv_level_med), 0.5)
    cost_high = np.fmin(np.fmax(np.fmax(slope_level_high, time_level_slow), curv_level_high), 0.9)
    cost = np.fmax(cost_low, np.fmax(cost_med, cost_high))
    return cost
```

10. Related Routes and Greedy Path Selection

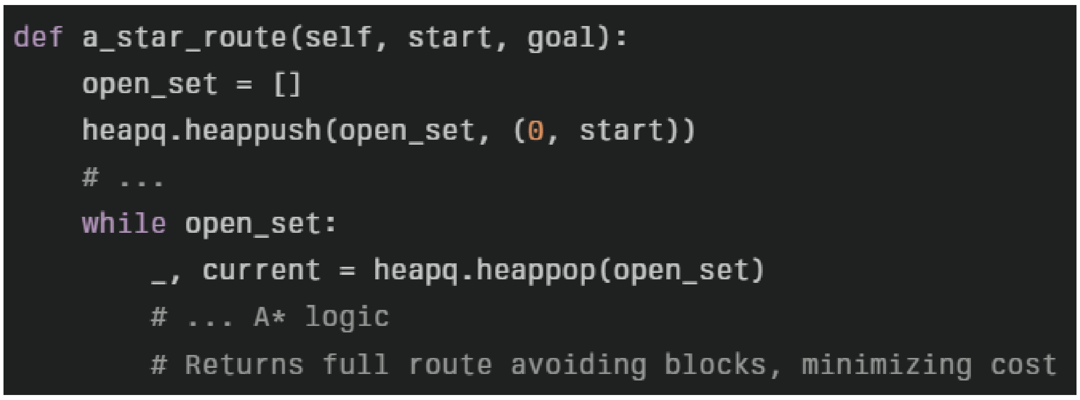
- get_related_paths: Finds all KML path options between the evacuation center and a target barangay.
- best_unblocked_path: Scans related paths for those not currently blocked and picks the one with minimum distance (greedy, not optimal for multi-hop or complex scenarios).



```python
def get_related_paths(self, barangay):
    base = f"{self.evacuation_center} to {barangay}"
    related_paths = []
    for path_name in self.evacuation_paths.keys():
        if path_name == base or path_name.startswith(base + "2") or path_name.startswith(base + "3"):
            related_paths.append(path_name)
    return related_paths

def best_unblocked_path(self, barangay):
    candidates = []
    for path_name in self.get_related_paths(barangay):
        if path_name in self.blocked_paths:
            continue
        if barangay in self.distances:
            index = 0
            if "2" in path_name: index = 1
            elif "3" in path_name: index = 2
            if index < len(self.distances[barangay]):
                dist = self.distances[barangay][index]
            else:
                dist = float('inf')
        else:
            dist = float('inf')
        candidates.append((path_name, dist))
    if not candidates:
        return None
    return min(candidates, key=lambda x: x[1])[0]
```

11. *A Pathfinding Algorithm Integration**

- a_star_route: This method implements the classic A* search, which:
    - Uses open/closed sets and heapq to efficiently pick the lowest-cost next step,
    - Considers blocked paths,
    - Calculates actual edge cost from real or estimated travel data,
    - Traces back from the goal to reconstruct the optimal route (series of barangay names).
- A* ensures the found route is truly optimal (minimum risk/cost) even if it has to follow a multi-hop path avoiding blocked segments.

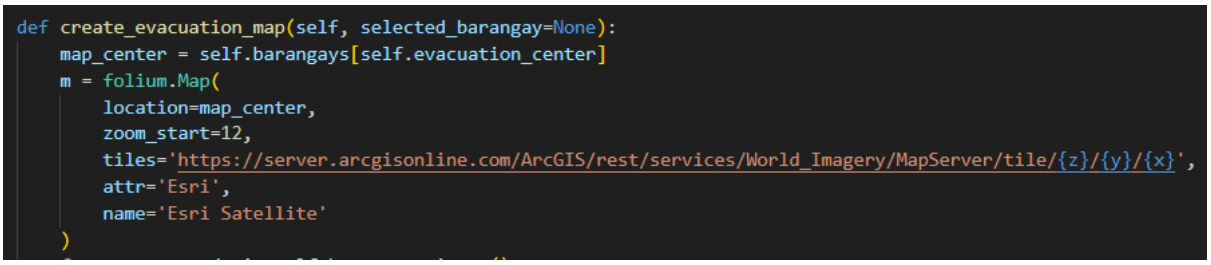


```python
def a_star_route(self, start, goal):
    open_set = []
    heapq.heappush(open_set, (0, start))
    # ...
    while open_set:
        _, current = heapq.heappop(open_set)
        # ... A* logic
        # Returns full route avoiding blocks, minimizing cost
```

12. Map Visualization

- create_evacuation_map: Builds an interactive Folium map centered on the evacuation center. Colors barangays, highlights the selected path (using the route from A*), and overlays all direct/alternate routes with correct color codes for blocked (red), best (blue), and others (gray).
- If supplied a full route (from A*), traces it step-by-step with polylines.

- Map from Esri satellite map



```
def create_evacuation_map(self, selected_barangay=None):
    map_center = self.barangays[self.evacuation_center]
    m = folium.Map(
        location=map_center,
        zoom_start=12,
        tiles='https://server.arcgisonline.com/ArcGIS/rest/services/World_Imagery/MapServer/tile/{z}/{y}/{x}',
        attr='Esri',
        name='Esri Satellite'
    )
```

13. Data File Discovery Function

- load_embedded_files: Looks in the data directory for KML and Excel files, returning their file paths for loading.

```
def load_embedded_files():
    # Returns KML/Excel filenames in ./data
```

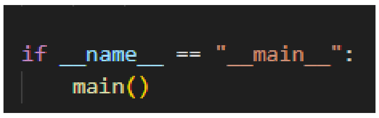
14. Main Function and User Interface

- main: Sets up the Streamlit application with a sidebar featuring usage instructions, system status, and a legend.
- Lets the user select a barangay, runs the A* algorithm to find the optimal route, computes and displays metrics (total distance and time), and shows the route on the map.
- If no route is available due to blocked paths, reports to the user.

```
def main():
    # Streamlit UI setup, file loading, map controls
    # User selects barangay, optimal route is calculated and displayed
```

15. Application Entrypoint

- if __name__ == "__main__": main(): Ensures the Streamlit app runs only when the script is executed directly (good programming practice).



```
if __name__ == "__main__":
    main()
```