zipfile: A Python library used for working with ZIP archives(read and extract ZIP file)
os: A library for interacting with the operating system.
numpy: numerical processing library used for handling arrays and matrices efficiently.
matplotlib.pyplot: creates static, interactive, and animated visualizations.

```
uploaded_file = 'problems-1.zip'
with zipfile.ZipFile(uploaded_file, 'r') as zip_ref:
    zip_ref.extractall('problems-1')
```

uploaded_file: A string variable storing the name of the ZIP file to be extracted
`problems-1.zip`
zipfile.ZipFile(uploaded_file, 'r'): Opens the ZIP file in read mode
Zipfile.zipfile is and object created to access the file
zip_ref.extractall('problems-1'): Extracts all the files and directories inside the ZIP file into a folder named 'problems-1'

```
extracted_dir = 'data/problems-1'
for root, dirs, files in os.walk(extracted_dir):
    for filename in files:
        print(os.path.join(root, filename))
```

extracted_dir: This variable holds the path to the directory where the files have been extracted.
os.walk(extracted_dir): Walks through the directory tree, returning a tuple for each directory it visits.

- **root**: The current directory path.
- **dirs**: A list of directories in the current directory.
- **files**: A list of files in the current directory.

os.path.join(root, filename): Combines the directory path and the filename to create the full path of the file, which is then printed.

```
def plot(paths_XYs):
    fig, ax = plt.subplots(tight_layout=True, figsize=(8, 8))
    colours = ['r', 'g', 'b', 'c', 'm', 'y', 'k']
    for i, XYs in enumerate(paths_XYs):
        c = colours[i % len(colours)]
        for XY in XYs:
            ax.plot(XY[:, 0], XY[:, 1], c=c, linewidth=2)
    ax.set_aspect('equal')
    plt.show()
```

plot(paths_XYs): This function takes a list of paths, `paths_XYs`, and plots them using matplotlib.

- **fig, ax = plt.subplots(...)**: Creates a figure and axes for plotting. `tight_layout=True` ensures the layout fits well within the figure size `(8, 8)`.
- **colours**: A list of color codes used to differentiate the paths.
- **for i, XYs in enumerate(paths_XYs)**: Iterates over each path in `paths_XYs`.
    - **c = colours[i % len(colours)]**: Selects a color for the current path.
    - **for XY in XYs**: Iterates over each set of coordinates in a path.
        - **ax.plot(XY[:, 0], XY[:, 1], c=c, linewidth=2)**: Plots the X and Y coordinates using the selected color and line width.
- **ax.set_aspect('equal')**: Sets the aspect ratio to equal, ensuring that units are the same on both axes.
- **plt.show()**: Displays the plot.

```
[33]
    def regularize_shapes(paths_XYs):
        regularized_paths = paths_XYs
        return regularized_paths
```

`def regularize_shapes(paths_XYs):`:

Defines a function called `regularize_shapes` meant to handle shape detection and regularization . Here `paths_XYs` is the input, assumed to be a list of coordinate paths.

`regularized_paths = paths_XYs`:returns the input without modification. You should replace this line with actual code that processes and regularizes shapes.

`return regularized_paths`:Returns the regularized paths. The goal is to make geometric shapes conform to expected mathematical properties, such as straightening lines or smoothing curves.

```
[34] def detect_symmetry(paths_XYs):
         symmetrical_paths = paths_XYs
         return symmetrical_paths
```

```
def detect_symmetry(paths_XYs):
    symmetrical_paths = paths_XYs
    return symmetrical_paths
```

```
def complete_curves(paths_XYs):

    completed_paths = paths_XYs
    return completed_paths
```

```
print("Listing all files in the extracted directory:")
for root, dirs, files in os.walk(extracted_dir):
    for filename in files:
        print(os.path.join(root, filename))

def read_csv(csv_path):
    np_path_XYs = np.genfromtxt(csv_path, delimiter=',')
    path_XYs = []
    for i in np.unique(np_path_XYs[:, 0]):
        npXYs = np_path_XYs[np_path_XYs[:, 0] == i][:, 1:]
        XYs = []
        for j in np.unique(npXYs[:, 0]):
            XY = npXYs[npXYs[:, 0] == j][:, 1:]
            XYs.append(XY)
        path_XYs.append(XYs)
    return path_XYs


complex_path = os.path.join(extracted_dir, 'problems/frag0.csv')


if os.path.exists(complex_path):
    paths_XYs = read_csv(complex_path)

    def plot(paths_XYs):
        fig, ax = plt.subplots(tight_layout=True, figsize=(8, 8))
        colours = ['r', 'g', 'b', 'c', 'm', 'y', 'k']
        for i, XYs in enumerate(paths_XYs):
            c = colours[i % len(colours)]
            for XY in XYs:
                ax.plot(XY[:, 0], XY[:, 1], c=c, linewidth=2)
        ax.set_aspect('equal')
        plt.show()


    plot(paths_XYs)
else:
    print(f"File {complex_path} not found.")
```

FILLING SATURN:

The data in the CSV file consists of several rows, each containing information about a point in a path. The first column typically indicates which path a point belongs to, and the second column indicates a specific segment or sub-path within the main path. The remaining columns contain the coordinates of the points (usually x and y).

`import os`: This imports the `os` module, which provides functions for interacting with the operating system, such as file handling.
`import numpy as np`: This imports the NumPy library and names it as `np`. NumPy is used for numerical operations and working with arrays.
`import matplotlib.pyplot as plt`: This imports the `pyplot` module from Matplotlib and aliases it as `plt`. It's used for creating plots and visualizations.
`from matplotlib.patches import Polygon`: This imports the `Polygon` class, which represents a polygonal shape and can be added to plots.
`from matplotlib.collections import PatchCollection`: This imports `PatchCollection`, a container for efficiently drawing multiple patches (such as polygons)

```
import os
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.patches import Polygon
from matplotlib.collections import PatchCollection
```

`def read_csv(csv_path):`: This defines a function named `read_csv` that takes a file path (`csv_path`) as an argument.
`np_path_XYs = np.genfromtxt(csv_path, delimiter=',')`: This reads the CSV file into a NumPy array (`np_path_XYs`) using `np.genfromtxt`, with `,` as the delimiter.
`path_XYs = []`: Initializes an empty list `path_XYs` to store the processed polylines.
`for i in np.unique(np_path_XYs[:, 0]):`: Iterates over unique values in the first column of `np_path_XYs`, which represent different paths.
`npXYs = np_path_XYs[np_path_XYs[:, 0] == i][:, 1:]`: Filters rows corresponding to the current path `i` and excludes the first column (identifier).
`XYs = []`: Initializes an empty list `XYs` to store polylines for the current path.

`for j in np.unique(npXYs[:, 0]):`: Iterates over unique values in the first column of `npXYs`, which represent different segments or polylines.
`XY = npXYs[npXYs[:, 0] == j][:, 1:]`: Filters rows corresponding to the current segment `j` and excludes the first column (identifier).
`XYs.append(XY)`: Appends the coordinates of the current segment to the `XYs` list.
`path_XYs.append(XYs)`: Appends the list of polylines for the current path to `path_XYs`.
`return path_XYs`: Returns the list of all paths and their polylines.

```
def read_csv(csv_path):
    np_path_XYs = np.genfromtxt(csv_path, delimiter=',')
    path_XYs = []
    for i in np.unique(np_path_XYs[:, 0]):
        npXYs = np_path_XYs[np_path_XYs[:, 0] == i][:, 1:]
        XYs = []
        for j in np.unique(npXYs[:, 0]):
            XY = npXYs[npXYs[:, 0] == j][:, 1:]
            XYs.append(XY)
        path_XYs.append(XYs)
    return path_XYs
```

Extracting Coordinates: The function takes coordinates (`XY`) and separates them into `x` (horizontal) and `y` (vertical) components.
Shoelace Formula: This is a method to calculate the area of a polygon by using the coordinates of its vertices. It involves a specific formula that involves summing products of x and y coordinates in a certain way.

```
def calculate_polygon_area(XY):
    x = XY[:, 0]
    y = XY[:, 1]
    return 0.5 * np.abs(np.dot(x, np.roll(y, 1)) - np.dot(y, np.roll(x, 1)))
```

#Shoelace Formula

The Shoelace formula calculates the area of a polygon using the coordinates of its vertices:

$$\text{Area} = 0.5 \times \left| \sum_{i=1}^{n} (x_i \cdot y_{i+1} - y_i \cdot x_{i+1}) \right|$$

where n is the number of vertices, and (x_{n+1}, y_{n+1}) is understood to be (x_1, y_1), i.e., the first vertex repeated to close the polygon.

- **`np.dot(x, np.roll(y, 1))`**: This computes the sum of the products of x-coordinates with the y-coordinates of the next vertex (wrapping around at the end). **`np.roll(y, 1)`** shifts the array **`y`** one position to the right, with the last element wrapping around to the beginning.
- **`np.dot(y, np.roll(x, 1))`**: This computes the sum of the products of y-coordinates with the x-coordinates of the next vertex (wrapping around at the end). **`np.roll(x, 1)`** shifts the array **`x`** one position to the right, similar to **`y`**.

The difference between these two dot products gives twice the area of the polygon. The absolute value ensures that the area is non-negative, and the factor of 0.5 accounts for the Shoelace formula's specific calculation.

ntuitive Explanation

- **Shoelace Pattern**: Imagine lacing up a shoe where you draw lines between consecutive vertices and across the polygon. The formula is named for the crisscross pattern formed when you connect these vertices.
- **Dot Products**: The dot products in the formula essentially sum the areas of the trapezoids formed between the polygon's edges and the coordinate axes. The difference between these sums gives the total signed area, which we then halve and take the absolute value of to get the polygon's area.

Setting Up the Plot: `plt.subplots()` creates a figure and axis to draw on.
Color and Patch Setup: We decide on colors (`pink` for rings and `lightblue` for the rest) and create lists (`patches_ring` and `patches_rest`) to hold our polygons.
Drawing the Shapes: For each shape:

- We plot the outline (`ax.plot(...)`).
- We check if the shape is closed (starts and ends at the same point) with `np.allclose(XY[0], XY[-1])`.
- We calculate the area of the shape to decide its category (ring or rest).
- We create a `Polygon` object for each shape and add it to the appropriate list based on the area.

Adding Patches to the Plot: We create collections (`PatchCollection`) of polygons for the ring and rest shapes, applying the specified colors and transparency (`alpha`).
Displaying the Plot: `plt.show()` renders the plot.

```
def plot_with_custom_fill(paths_XYs):
    fig, ax = plt.subplots(tight_layout=True, figsize=(8, 8))
    patches_ring = []
    patches_rest = []
    ring_color = 'pink'
    rest_color = 'lightblue'
```