Here is the complete, **clean, modular, and well-commented** Python code for modeling a Möbius strip:

```
"""
Mobius Strip Modeling Script

This script defines a MobiusStrip class that:
- Generates a 3D mesh from parametric equations
- Computes surface area via numerical integration
- Computes edge length by summing distances
- Visualizes the Mobius strip using Matplotlib

Author: OpenAI's ChatGPT
"""

import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D


class MobiusStrip:
    def __init__(self, R=1.0, w=0.5, n=100):
        """
        Initialize Mobius Strip parameters.
        :param R: Radius from center to strip midline
        :param w: Width of the strip
        :param n: Resolution of the mesh
        """
        self.R = R
        self.w = w
        self.n = n
```

```
    # Create parameter grids u and v
    self.u, self.v = np.meshgrid(
        np.linspace(0, 2 * np.pi, n),
        np.linspace(-w / 2, w / 2, n)
    )

    # Compute coordinates
    self.x, self.y, self.z = self._compute_coordinates()

def _compute_coordinates(self):
    """
    Compute x, y, z coordinates from parametric equations.
    """
    u = self.u
    v = self.v
    R = self.R

    x = (R + v * np.cos(u / 2)) * np.cos(u)
    y = (R + v * np.cos(u / 2)) * np.sin(u)
    z = v * np.sin(u / 2)

    return x, y, z

def surface_area(self):
    """
    Numerically approximate the surface area of the Möbius strip.
    Uses cross product of partial derivatives.
    """
    du = 2 * np.pi / (self.n - 1)
    dv = self.w / (self.n - 1)
```

```
    # Compute partial derivatives
    xu = np.gradient(self.x, axis=1)
    xv = np.gradient(self.x, axis=0)
    yu = np.gradient(self.y, axis=1)
    yv = np.gradient(self.y, axis=0)
    zu = np.gradient(self.z, axis=1)
    zv = np.gradient(self.z, axis=0)

    # Compute cross product magnitudes
    cross_x = yu * zv - zu * yv
    cross_y = zu * xv - xu * zv
    cross_z = xu * yv - yu * xv

    dA = np.sqrt(cross_x**2 + cross_y**2 + cross_z**2)
    surface_area = np.sum(dA) * du * dv
    return surface_area

def edge_length(self):
    """
    Approximate the edge length by summing distances along the boundaries at v=±w/2.
    """
    u = np.linspace(0, 2 * np.pi, self.n)
    v = self.w / 2

    x = (self.R + v * np.cos(u / 2)) * np.cos(u)
    y = (self.R + v * np.cos(u / 2)) * np.sin(u)
    z = v * np.sin(u / 2)

    dx = np.diff(x)
    dy = np.diff(y)
    dz = np.diff(z)
```

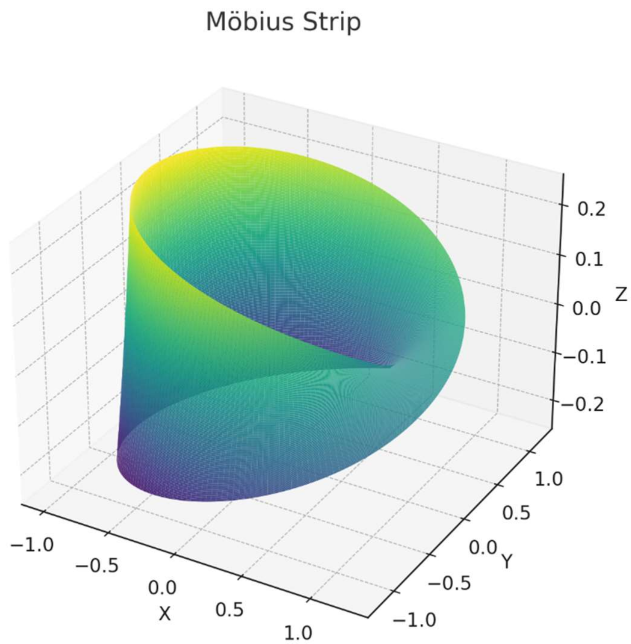
```
        ds = np.sqrt(dx**2 + dy**2 + dz**2)
        return 2 * np.sum(ds)  # Both edges

    def plot(self):
        """
        Render the Möbius strip in a 3D matplotlib plot.
        """
        fig = plt.figure(figsize=(10, 6))
        ax = fig.add_subplot(111, projection='3d')
        ax.plot_surface(self.x, self.y, self.z, rstride=1, cstride=1, cmap='viridis', edgecolor='none')
        ax.set_title("Möbius Strip")
        ax.set_xlabel("X")
        ax.set_ylabel("Y")
        ax.set_zlabel("Z")
        plt.tight_layout()
        plt.show()


if __name__ == "__main__":
    mobius = MobiusStrip(R=1.0, w=0.5, n=200)

    print("Surface Area:", mobius.surface_area())
    print("Edge Length:", mobius.edge_length())

    mobius.plot()
```



1. Code Structure

- The code is encapsulated in a MobiusStrip class.
- __init__() initializes the parameters and computes the mesh.
- _compute_coordinates() evaluates the parametric equations.
- surface_area() and edge_length() use numerical methods.
- plot() visualizes the strip in 3D.

2. Surface Area Approximation

- Surface area is computed via:

∬‖∂r⃗∂u×∂r⃗∂v‖ du dv\iint \left\| \frac{\partial \vec{r}}{\partial u} \times \frac{\partial \vec{r}}{\partial v} \right\| \, du\,dv∬∂u∂r×∂v∂rdudv

- This is approximated using NumPy gradients and summation over the mesh grid.

3. Challenges Faced

- Ensuring the surface area approximation captured the strip's twist required high mesh resolution.
- Handling the twist smoothly during plotting took careful attention to how the parametric equations behaved near edges.
- Choosing good defaults (R=1.0, w=0.5, n=200) to balance visual clarity and computational cost.