

Before you turn this problem in, make sure everything runs as expected. First, **restart the kernel** (in the menubar, select Kernel→Restart) and then **run all cells** (in the menubar, select Cell→Run All).

Make sure you fill in any place that says `YOUR CODE HERE` or "YOUR ANSWER HERE", as well as your name and collaborators below:

```
In [1]: NAME = ""  
COLLABORATORS = ""
```

CSE 30 Spring 2022 - Homework 2

Instructions

Please disregard the YOUR NAME and COLLABORATORS above. They are put there automatically by the grading tool. You can find instructions on how to work on a homework on Canvas. Here is a short summary:

Submitting your work

To submit your work:

- First, click on "Runtime > Restart and run all", and check that you get no errors. This enables you to catch any error you might have introduced, and not noticed, due to your running cells out of order.
- Second, download the notebook in .ipynb format (File > Download .ipynb) and upload the .ipynb file to [this form](#).
- This homework is due at **11:59 pm on Thursday 07 April 2022**.

You can submit multiple times; the last submission before the deadline is the one that counts.

Homework format

For each question in this notebook, there is:

- A text description of the problem.
- One or more places where you have to insert your solution. You need to complete every place marked:

```
# YOUR CODE HERE
```

and you should not modify any other place.

- One or more test cells. Each cell is worth some number of points, marked at the top. You should not modify these tests cells. The tests pass if no error is printed out: when there is a statement that says, for instance:

```
assert x == 2
```

then the test passes if `x` has value 2, and fails otherwise. You can insert a `print(x)` (for this case!) somewhere if you want to debug your work; it is up to you.

Notes:

- Your code will be tested both according to the tests you can see (the `assert` statements you can see), *and* additional tests. This prevents you from hard-coding the answer to the particular questions posed. Your code should solve the *general* intended case, not hard-code the particular answer for the values used in the tests.
- **Please do not delete or add cells!** The test is autograded, and if you modify the test by adding or deleting cells, even if you re-add cells you delete, you may not receive credit.
- **Please do not import modules that are not part of the [standard library](#).** You do not need any, and they will likely not be available in the grading environment, leading your code to fail.
- **If you are inactive too long, your notebook might get disconnected from the back-end.** Your work is never lost, but you have to re-run all the cells before you continue.
- You can write out print statements in your code, to help you test/debug it. But remember: the code is graded on the basis of what it outputs or returns, not on the basis of what it prints.
- **TAs and tutors have access to this notebook**, so if you let them know you need their help, they can look at your work and give you advice.

Grading

Each cell where there are tests is worth a certain number of points. You get the points allocated to a cell only if you pass *all* the tests in the cell.

The tests in a cell include both the tests you can see, and other, similar, tests that are used for grading only. Therefore, you cannot hard-code the solutions: you really have to solve the essence of the problem, to receive the points in a cell.

Code of Conduct

- Work on the test yourself, alone.

- You can search documentation on the web, on sites such as the Python documentation sites, Stackoverflow, and similar, and you can use the results.
- You cannot share your work with others or solicit their help.

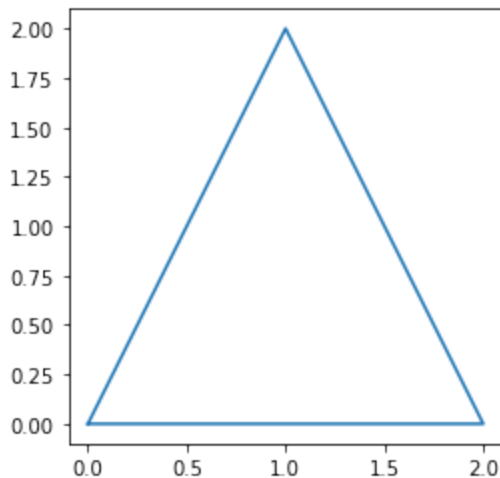
Problem: Fractals

We want to have a bit of fun generating fractals. Let's first write a function that, given a sequence of (x,y) points, draws the corresponding polygon:

```
In [2]: import matplotlib.pyplot as plt

def draw_polygon(*coords, closed=True):
    co = list(coords)
    fig, ax = plt.subplots(1, 1)
    if closed and co[0] != co[-1]:
        co.append(co[0]) # Repeat first point
    xs, ys = zip(*co) # Splits coordinates
    ax.set_aspect('equal', 'box')
    ax.plot(xs, ys)
    plt.show()
```

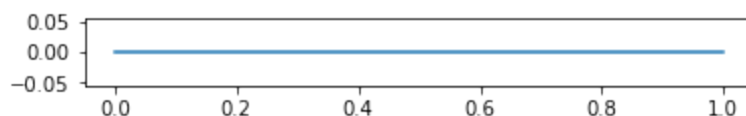
```
In [3]: draw_polygon((0, 0), (1, 2), (2, 0))
```



Kinking lines

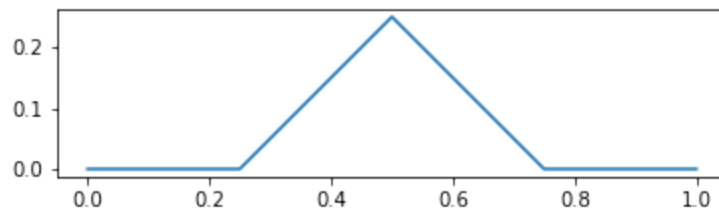
Whenever you have a line like this, from (x_0, y_0) to (x_1, y_1) :

```
In [4]: draw_polygon((0, 0), (1, 0), closed=False)
```



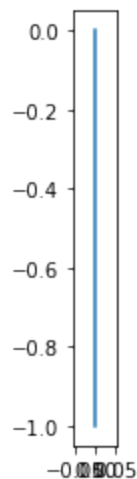
You need to add a "kink" to it, as follows:

```
In [5]: draw_polygon([(0, 0), (0.25, 0), (0.5, 0.25), (0.75, 0), (1, 0)], closed=False)
```



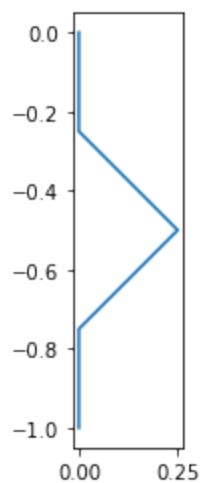
Now this kink must be invariant with respect to directions and scaling. For example, if the original line was from up to down:

```
In [6]: draw_polygon([(0, 0), (0, -1)], closed=False)
```



Then adding the kink will result in:

```
In [7]: draw_polygon([(0, 0), (0, -0.25), (0.25, -0.5), (0, -0.75), (0, -1)], closed=False)
```



We call the above operation "adding a kink". To define it mathematically, we need to use *vector addition*, so here is a quick reminder. If you have two vectors $v = (v_x, v_y)$ and $w = (w_x, w_y)$, their sum $z = v + w$ is computed in *componentwise fashion*, which you can do in code, like so:

```

vx, vy = v
wx, wy = w
z = (vx + wx, vy + wy)

```

With that notation, we can define adding a kink as follows. If you have a segment from point $p_0 = (x_0, y_0)$ to point $p_1 = (x_1, y_1)$, its direction is $\Delta = (x_1, y_1) - (x_0, y_0) = (x_1 - x_0, y_1 - y_0) = (\Delta_x, \Delta_y)$, and its perpendicular direction is $\nu = (-\Delta_y, \Delta_x)$. The points along the kink are then:

- $q_0 = p_0$ (initial point);
- $q_1 = p_0 + \frac{1}{4}\Delta$ (points where it "kinks up");
- $q_2 = p_0 + \frac{1}{2}\Delta + \frac{1}{4}\nu$ (tip of the kink);
- $q_3 = p_0 + \frac{3}{4}\Delta$ (points where it kinks back to straight);
- p_1 (final point).

The first task for you in implementing fractals is to implement a function `kink` that, given a segment from p_0 to p_1 , returns the points q_0, q_1, q_2, q_3 . Every point here is represented as a tuple. Note that the last point p_1 is *not* returned (this will make it easier to kink a polygon).

```

In [8]: def kink(p0, p1):
        """Returns the tuple of the four points q0, q1, q2, q3, resulting from kinking
        Note that each q is returned as a two-element tuple."""
        (p0x, p0y) = p0
        (p1x, p1y) = p1
        p0x = float(p0x)
        p0y = float(p0y)
        p1x = float(p1x)
        p1y = float(p1y)
        q1d = ((1/4) * (p1x-p0x), (1/4) * (p1y-p0y))
        q2d = ((1/2) * (p1x-p0x), (1/2) * (p1y-p0y))
        q3d = ((3/4) * (p1x-p0x), (3/4) * (p1y-p0y))
        v = ((1/4) * -(p1y-p0y), (1/4) * (p1x-p0x))

        q0 = p0
        q1 = (p0[0] + q1d[0], p0[1] + q1d[1])
        q2 = (p0[0] + q2d[0] + v[0], p0[1] + q2d[1] + v[1])
        q3 = (p0[0] + q3d[0], p0[1] + q3d[1])
        line = (q0, q1, q2, q3)
        return line

```

```

In [9]: ## Here you can test your code.

```

```

kink((0, 0), (1,0))

```

```

Out[9]: ((0, 0), (0.25, 0.0), (0.5, 0.25), (0.75, 0.0))

```

Let's check that this works for the two cases above.

```

In [10]: # 5 pt

```

```
assert kink((0, 0), (1, 0)) == ((0, 0), (0.25, 0.0), (0.5, 0.25), (0.75, 0.0))
assert kink((0, 0), (0, -1)) == ((0, 0), (0.0, -0.25), (0.25, -0.5), (0.0, -0.75))
```

For the general case, let us define an approximate comparison of a list of points.

```
In [11]: def approxeq(list1, list2, epsilon=0.001):
          assert len(list1) == len(list2)
          for p1, p2 in zip(list1, list2):
              x1, y1 = p1
              x2, y2 = p2
              assert abs(x2 - x1) < epsilon and abs(y2 - y1) < epsilon
```

```
In [12]: approxeq(kink((0, 0), (1, 0)), ((0, 0), (0.25, 0.0), (0.5, 0.25), (0.75, 0.0)))
```

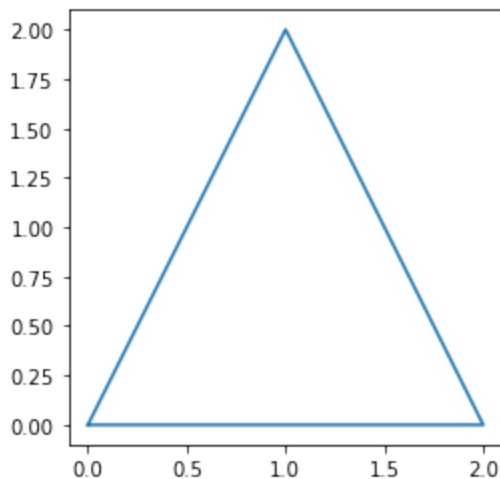
```
In [13]: # 10 points: The kink should work in general.
```

```
approxeq(kink((0, 0), (4, 5)), ((0, 0), (1.0, 1.25), (0.75, 3.5), (3.0, 3.75)))
approxeq(kink((4, 3), (-9, -1)), ((4, 3), (0.75, 2.0), (-1.5, -2.25), (-5.75, 0.0)))
```

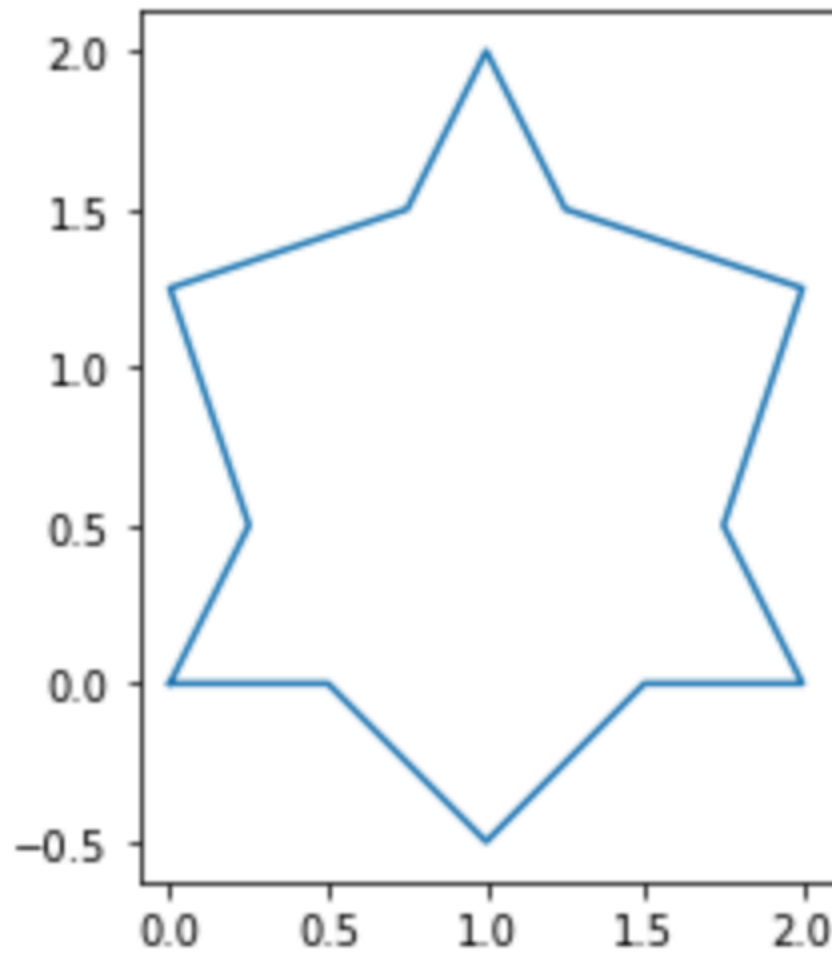
Kinking polygons

To kink a polygon, we simply kink every one of its segments. It looks like this. If the basic polygon is:

```
In [14]: poly1 = [(0, 0), (1, 2), (2, 0)]
          draw_polygon(*poly1)
```



Then its kinked version has to look like this:



Write below the kinked polygon function.

```
In [15]: def kink_polygon(poly):
    polygon = []
    for point in range(len(poly)):
        if point == len(poly) - 1:
            line = kink(poly[point], poly[0])
            for point in line:
                polygon.append(point)
        else:
            line = kink(poly[point], poly[point + 1])
            for point in line:
                polygon.append(point)
    return polygon
```

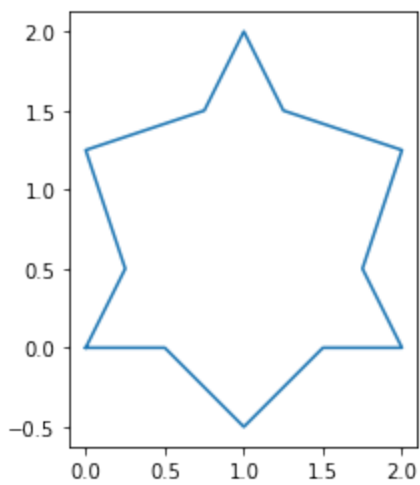
In [16]: *# You can use this space to debug your code.*

```
kink_polygon(poly1)
```

```
Out[16]: [(0, 0),
          (0.25, 0.5),
          (0.0, 1.25),
          (0.75, 1.5),
          (1, 2),
          (1.25, 1.5),
          (2.0, 1.25),
          (1.75, 0.5),
          (2, 0),
          (1.5, 0.0),
          (1.0, -0.5),
          (0.5, 0.0)]
```

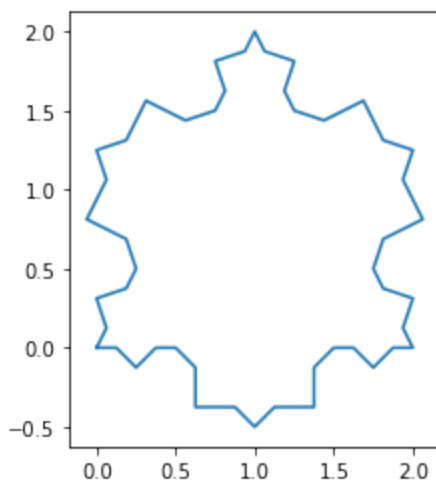
Let us check that the solution works as intended.

```
In [17]: poly1 = [(0, 0), (1, 2), (2, 0)]
poly2 = kink_polygon(poly1)
draw_polygon(*poly2)
```



We can kink it one more time!

```
In [18]: poly3 = kink_polygon(poly2)
draw_polygon(*poly3)
```



Let us check that your solution works.

```
In [19]: poly1 = [(0, 0), (1, 2), (2, 0)]
poly2 = kink_polygon(poly1)
approxeq(poly2, [(0, 0),
(0.25, 0.5),
(0.0, 1.25),
(0.75, 1.5),
(1, 2),
(1.25, 1.5),
(2.0, 1.25),
(1.75, 0.5),
(2, 0),
(1.5, 0.0),
(1.0, -0.5),
(0.5, 0.0)])
poly3 = kink_polygon(poly2)
approxeq(poly3, [(0, 0),
(0.0625, 0.125),
(0.0, 0.3125),
(0.1875, 0.375),
(0.25, 0.5),
(0.1875, 0.6875),
(-0.0625, 0.8125),
(0.0625, 1.0625),
(0.0, 1.25),
(0.1875, 1.3125),
(0.3125, 1.5625),
(0.5625, 1.4375),
(0.75, 1.5),
(0.8125, 1.625),
(0.75, 1.8125),
(0.9375, 1.875),
(1, 2),
(1.0625, 1.875),
(1.25, 1.8125),
(1.1875, 1.625),
(1.25, 1.5),
(1.4375, 1.4375),
(1.6875, 1.5625),
(1.8125, 1.3125),
(2.0, 1.25),
(1.9375, 1.0625),
(2.0625, 0.8125),
(1.8125, 0.6875),
(1.75, 0.5),
(1.8125, 0.375),
(2.0, 0.3125),
(1.9375, 0.125),
(2, 0),
(1.875, 0.0),
(1.75, -0.125),
(1.625, 0.0),
(1.5, 0.0),
(1.375, -0.125),
```

```
(1.375, -0.375),
(1.125, -0.375),
(1.0, -0.5),
(0.875, -0.375),
(0.625, -0.375),
(0.625, -0.125),
(0.5, 0.0),
(0.375, 0.0),
(0.25, -0.125),
(0.125, 0.0)])
```

Here are some hidden tests, just for the sake of suspense.

In [20]: *# 10 points. Kinking a polygon in the dark.*

Areas and perimeter

What's interesting about fractals is their area and perimeter. If you have a polygon $(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)$, you can find the area via:

$$\sum_{i=1}^n (x_{i+1} - x_i) \cdot \frac{y_i + y_{i+1}}{2},$$

and the perimeter via

$$\sum_{i=1}^n \sqrt{(x_{i+1} - x_i)^2 + (y_{i+1} - y_i)^2},$$

where we have used the convention that $x_{n+1} = x_1$ and $y_{n+1} = y_1$. Write functions to implement the above and compute area and perimeter of a polygon.

```
In [21]: def poly_area(polygon):
    """Returns the area of a polygon."""
    # YOUR CODE HERE
    area = 0.0
    for point in range(len(polygon)):
        if point == len(polygon) - 1:
            area += (polygon[0][0] - polygon[point][0]) * ((polygon[point][1] + polygon
        else:
            area += (polygon[point + 1][0] - polygon[point][0]) * ((polygon[point][1] +
    return area
```

```
In [22]: import math

def poly_perimeter(polygon):
    """Returns the perimeter of a polygon."""
    perimeter = 0.0
    for point in range(len(polygon)):
        if point == len(polygon) - 1:
            perimeter += (math.sqrt((polygon[0][0] - polygon[point][0])**2 + ((polygon[
        else:
```

```

    perimeter += (math.sqrt((polygon[point + 1][0] - polygon[point][0])**2 + ((
return perimeter

```

In [23]: *## Here you can test your code.*

```

# YOUR CODE HERE
poly1 = [(0, 0), (1, 2), (2, 0)]
print(poly_perimeter(poly1))

```

6.47213595499958

Let us test both functions.

In [24]: *# 10 points: area and perimeter of polygons.*

```

poly1 = [(0, 0), (1, 2), (2, 0)]
assert poly_area(poly1) == 2
assert abs(poly_perimeter(poly1) - 6.472) < 0.01

```

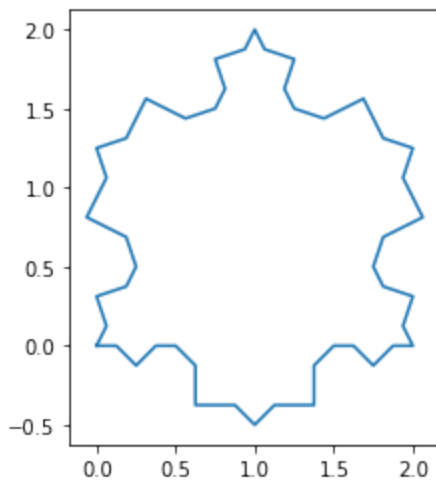
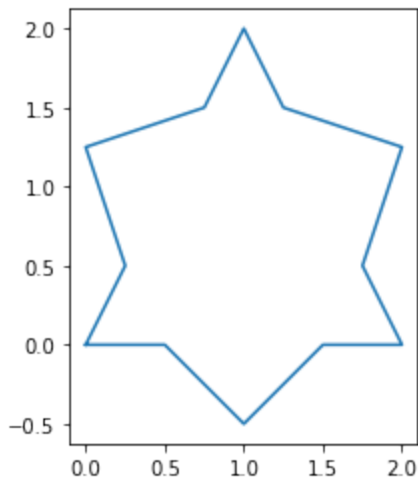
Let us visualize repeated kinking.

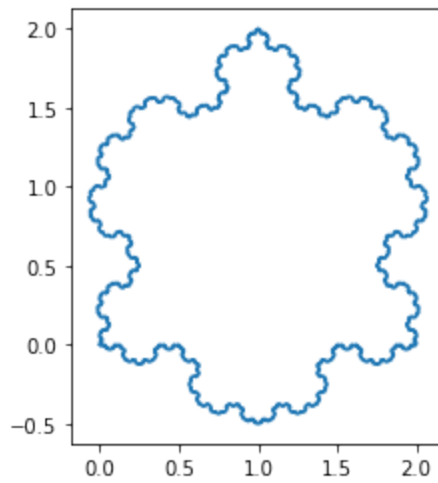
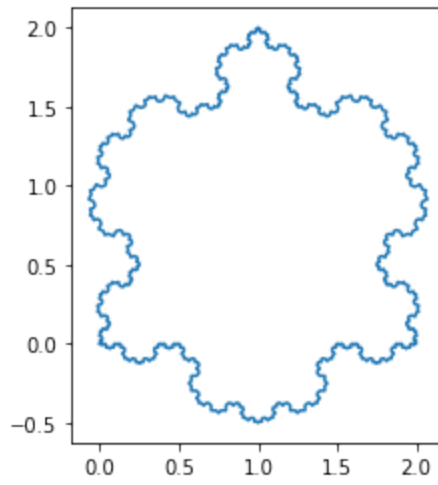
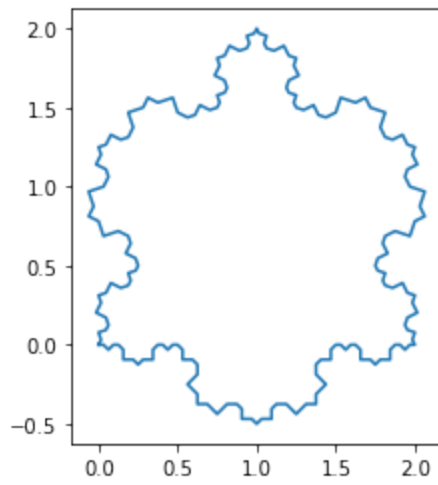
In [25]:

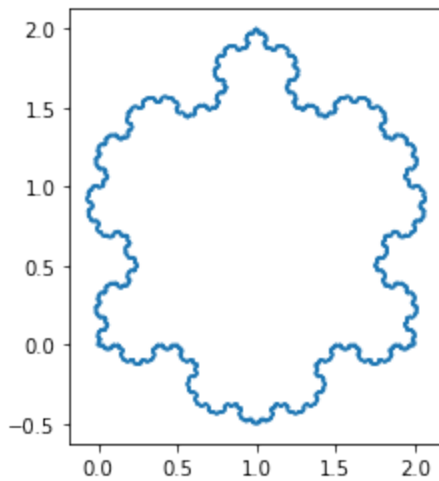
```

poly = [(0, 0), (1, 2), (2, 0)]
for n in range(6):
    poly = kink_polygon(poly)
    draw_polygon(*poly)

```







Let's see how area and perimeter change as we add kinks.

In [26]: *# tests for the behavior of perimeter, area.*

```
poly = [(0, 0), (0, 1), (1, 1), (1, 0)]
for n in range(7):
    poly = kink_polygon(poly)
    area = poly_area(poly)
    perim = poly_perimeter(poly)
    print("Area:", area, "Perimeter:", perim)
assert 14.9 < perim < 15
assert 1.399 < area < 1.4
```

```
Area: 1.25 Perimeter: 4.828427124746191
Area: 1.34375 Perimeter: 5.828427124746187
Area: 1.37890625 Perimeter: 7.035533905932754
Area: 1.39208984375 Perimeter: 8.492640687119291
Area: 1.39703369140625 Perimeter: 10.25152416360244
Area: 1.398876342773438 Perimeter: 12.374684335382558
Area: 1.399582862854004 Perimeter: 14.93756537627452
```

See what happens? The area converges, but the perimeter does not. You end up with a finite area with an infinite perimeter. That's true for every polygon you start with:

In [27]: *# more tests for the behavior of perimeter, area*

```
poly = [(0, 1), (1, 4), (6, 4), (8, 1)]
for n in range(7):
    poly = kink_polygon(poly)
    area = poly_area(poly)
    perim = poly_perimeter(poly)
    print("Area:", area, "Perimeter:", perim)
assert 30.6 < area < 31
assert 73 < perim < 74
```

Area: 26.5 Perimeter: 23.861880357537487
Area: 29.125 Perimeter: 28.803837591445575
Area: 30.109375 Perimeter: 34.769307680829996
Area: 30.478515625 Perimeter: 41.97026707869125
Area: 30.616943359375 Perimeter: 50.662593998898366
Area: 30.668853759765625 Perimeter: 61.15516076857142
Area: 30.68832015991211 Perimeter: 73.82080926826043