1

THE KOCH SNOWFLAKE

We'll start our Python adventures by figuring out how to draw an interesting shape called the *Koch snowflake*, invented by Swedish mathematician Helge von Koch in 1904. The Koch snowflake is a *fractal*—a type of figure that repeats itself as you zoom in to it.

Fractals derive their repeating nature from *recursion*, a technique where something is defined in terms of itself. In particular, you draw a fractal using a *recursive algorithm*, a repeating process where one repetition's output becomes the input of the next repetition.

As you work through this chapter, you'll learn:

- The basics of recursive algorithms and functions
- How to create graphics using the `turtle` module
- A recursive algorithm to draw the Koch snowflake
- Some linear algebra

How It Works

Figure 1-1 shows what the Koch snowflake looks like. Notice how the large branch in the middle is repeated on a smaller scale by branches on the left and right. Similarly, the large branch in the middle is itself made up of smaller branches that echo the larger shape. This is the repeating, self-similar nature of a fractal.

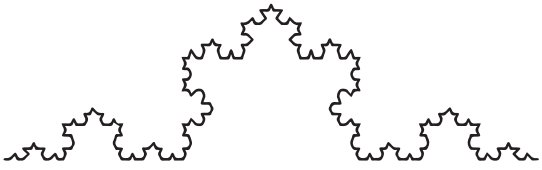

Figure 1-1: The Koch snowflake

If you know how to calculate the points that form the basic shape making up the snowflake, you can develop an algorithm to perform the same calculations recursively. This way, you'll draw smaller and smaller versions of that shape, building up the fractal. In this section, we'll look generally at how recursion works. Then we'll consider how to apply recursion, along with some linear algebra and Python's `turtle` module, to draw the Koch snowflake.

Using Recursion

To get a feel for how recursion works, let's take a look at a simple recursive algorithm: computing the factorial of a number. The factorial of a number can be defined by a function, as shown here:

$$f(N) = 1 \times 2 \times 3 \times \ldots \times (N-1) \times N$$

In other words, the factorial of N is just the product of the numbers 1 through N. You can rewrite this as:

$$f(N) = N \times (N-1) \times \ldots \times 3 \times 2 \times 1$$

which can again be rewritten as:

$$f(N) = N \times f(N-1)$$

Wait, what did you just do? You defined f in terms of itself! That's recursion. Calling $f(N)$ will end up calling $f(N-1)$, which will end up calling $f(N-2)$, and so on. But how do you know when to stop? Well, you have to define $f(1)$ as 1, and that will be the deepest step of the recursion.

Here's how to implement the recursive factorial function in Python:

```
def factorial(N):
  ❶ if N == 1:
        return 1
    else:
      ❷ return N * factorial(N-1)
```

You handle the case where *N* is equal to 1 by simply returning `1` ❶, and you implement the recursive call by calling `factorial()` again ❷, this time passing in `N-1`. The function will keep calling itself until *N* equals 1. The net effect is that when the function returns, it will have computed the product of all numbers 1 through *N*.

In general, when you're trying to implement an algorithm using recursion, follow these steps:

1. Define a base case where the recursion ends. In our factorial example, you did this by defining $f(1)$ as 1.
2. Define the recursive step. For this you need to think about how to express the algorithm as a recursive process. In some algorithms, there can be multiple recursive calls from a function—as you'll see soon.

Recursion is a helpful tool for problems that can be naturally partitioned into smaller versions of themselves. The factorial algorithm is a perfect example of this partitioning, and as you'll soon see, so is drawing the Koch snowflake. That said, recursion isn't always the most efficient way to solve a problem. In some cases, it would make sense to re-implement the recursive algorithm in terms of loops. But the fact remains that recursive algorithms are often more compact and elegant compared to their loopy counterparts.

Computing the Snowflake

Now let's look at how to construct the Koch snowflake. Figure 1-2 shows the basic pattern for drawing the snowflake. I'll call this pattern a *flake*. The basis of the figure is the line segment $\overline{AB}$ of length d. The segment is split into three equal parts, $\overline{AP_1}$, $\overline{P_1P_3}$, and $\overline{P_3B}$, each of which has a length r. Instead of directly connecting points P_1 and P_3, these points are connected through P_2, which is chosen such that P_1, P_2, and P_3 form an equilateral triangle of side length r and height h. Point C, the midpoint of P_1 and P_3 (and by extension of A and B), falls directly beneath P_2, such that $\overline{AB}$ and $\overline{CP_2}$ are perpendicular.

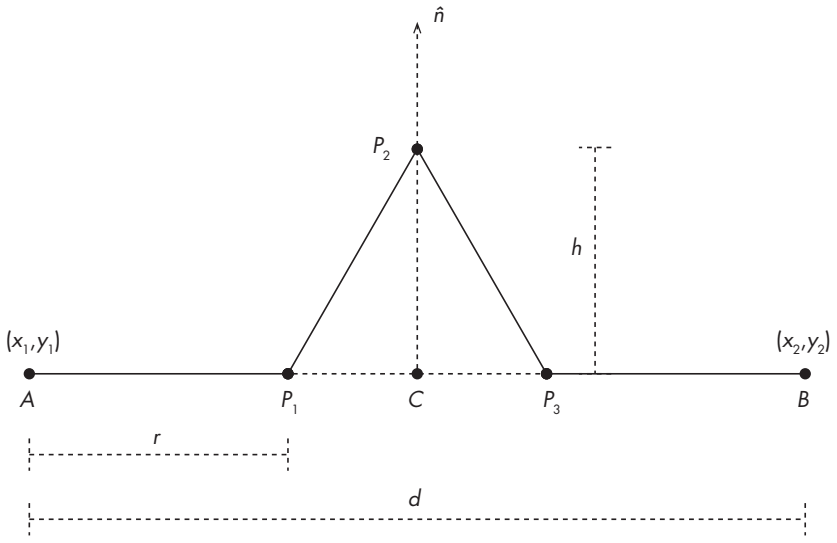

Figure 1-2: The basic pattern for drawing a Koch snowflake

Once you understand how to calculate the points shown in Figure 1-2, you'll be able to recursively draw smaller and smaller flakes to reproduce the Koch snowflake. Essentially, your goal is this: given points A and B, you want to compute the points P_1, P_2, and P_3 and join them up as shown in the figure. To calculate those points, you'll need to use some linear algebra, a mathematical discipline that lets you compute distances and figure out coordinates of points based on *vectors*, quantities that have both magnitude and direction.

Here's a simple formula from linear algebra that you'll be using. Say you have a point A in 3D space and a unit vector $\hat{n}$ (a *unit vector* is a vector with a length of 1 unit). Point B at a distance d along this unit vector is given by:

$$B = A + d \times \hat{n}$$

You can easily verify this with an example. Take the case where $A = (5, 0, 0)$ and $\hat{n} = (0, 1, 0)$. What are the coordinates for a point B that's 10 units away from A along $\hat{n}$? Using the previous formula, you get:

$$B = (5, 0, 0) + 10 \times (0, 1, 0) = (5, 10, 0)$$

In other words, to get from A to B, you move 10 units along the positive y-axis.

Here's another result you'll use—let's call it the *perpendicular vector trick*. Say you have a vector $\vec{A} = (a, b)$. If you have another vector $\vec{B}$ that's perpendicular to $\vec{A}$, it can be expressed as $\vec{B} = (-b, a)$. You can verify that this trick works by taking the dot product of $\vec{A}$ and $\vec{B}$. To take the dot product of a pair of two-dimensional vectors, multiply the first components from each vector, then multiply the second components from each vector, and finally add the results together. In this case, the dot product of $\vec{A}$ and $\vec{B}$ is:

$$\vec{A} \cdot \vec{B} = (a \times -b) + (b \times a) = -ab + ab = 0$$

The dot product of two perpendicular vectors will always be zero, so $\vec{B}$ is indeed perpendicular to $\vec{A}$.

With this in mind, let's return to the flake in Figure 1-2. How can you calculate the position of P_2, given the coordinates for points A and B? You know that P_2 falls h distance away from point C along unit vector $\hat{n}$. Your first linear algebra formula tells you:

$$P_2 = C + h \times \hat{n}$$

Now let's put those variables in terms that you know. First, C is the midpoint of line $\overline{AB}$, so $C = (A + B) / 2$. Next, h is the height of an equilateral triangle with side length r. The Pythagorean theorem tells you:

$$h = \frac{\sqrt{3}}{2} r$$

In this case, r is simply a third of the distance from A to B. If A has coordinates (x_1, y_1) and B has coordinates (x_2, y_2), you can calculate the distance between them as:

$$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

Then simply divide d by 3 to get r.

Finally, you need a way to express $\hat{n}$. You know that $\hat{n}$ is perpendicular to vector $\vec{AB}$, and you can express $\vec{AB}$ by subtracting point A's coordinates from point B's:

$$\vec{AB} = (x_2 - x_1, y_2 - y_1)$$

The magnitude of $\vec{AB}$ is given by $d = |\vec{AB}|$. You can now use the perpendicular vector trick to express $\hat{n}$ in terms of A and B:

$$\hat{n} = \frac{(-(y_2 - y_1), x_2 - x_1)}{|\vec{AB}|} = \frac{(y_1 - y_2)}{d}, \frac{(x_2 - x_1)}{d}$$

Next you need to compute P_1 and P_3. For this you're going to use another result from linear algebra. Let's say you have a line $\overline{AB}$ and a point C on the line. Let a be the distance of C from A and b be the distance of C from B. The point C is given by:

$$C = \frac{(b \times A) + (a \times B)}{a + b}$$

To understand this formula, think about what happens if C is the midpoint of A and B, meaning a and b would be the same. In this case, you can intuit that C ought to equal $(A + B) / 2$. Substitute all the bs for as in the previous equation. You'll get:

$$C = \frac{(a \times A) + (a \times B)}{a + a} = \frac{A + B}{2}$$

With this new formula in mind, you can now compute P_1 and P_3. These points divide line $\overline{AB}$ into thirds, meaning the distance from P_1 to B is twice

the distance from P_1 to A ($b = 2a$), and the distance from P_3 to A is twice the distance from P_3 to B ($a = 2b$). Feeding this into the formula, you can therefore calculate the points as:

$$P_1 = \frac{2 \times A + B}{2} \text{ and } P_3 = \frac{A + 2 \times B}{3}$$

Now you have everything you need to draw the first level of the snowflake fractal. Once you decide on A and B, you know how to compute the points P_1, P_2, and P_3. But what happens at the second level of the fractal? You take each individual line segment from the flake at the first level (Figure 1-2) and replace it with a smaller flake. The result is shown in Figure 1-3.

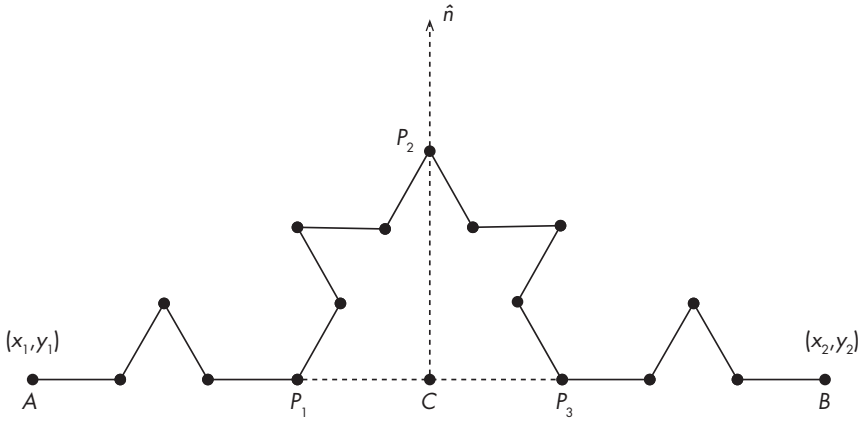

Figure 1-3: The second step of Koch snowflake construction

Notice how each of the four line segments from Figure 1-2, $\overline{AB}_1$, $\overline{P_1P_2}$, $\overline{P_2P_3}$, and $\overline{P_3B}$, has become the basis for a new flake. In the Koch snowflake program, you'll be able to use the endpoints of each line segment, for example, A and P_1, as new values for A and B and recursively perform the same calculations used to arrive at the points in Figure 1-2.

At each level of the fractal, you'll subdivide the snowflake again, drawing smaller and smaller self-similar figures. This is the *recursive step* of the algorithm, which you'll repeat until you reach a *base case*. This should happen when $\overline{AB}$ is smaller than a certain threshold—say, 10 pixels. When you hit that threshold, just draw the line segments and stop recursing.

To make the final output a bit fancy, you can draw three linked flakes as the first level of the fractal. This will give you the hexagonal symmetry of an actual snowflake. Figure 1-4 shows what the starting drawing will look like.

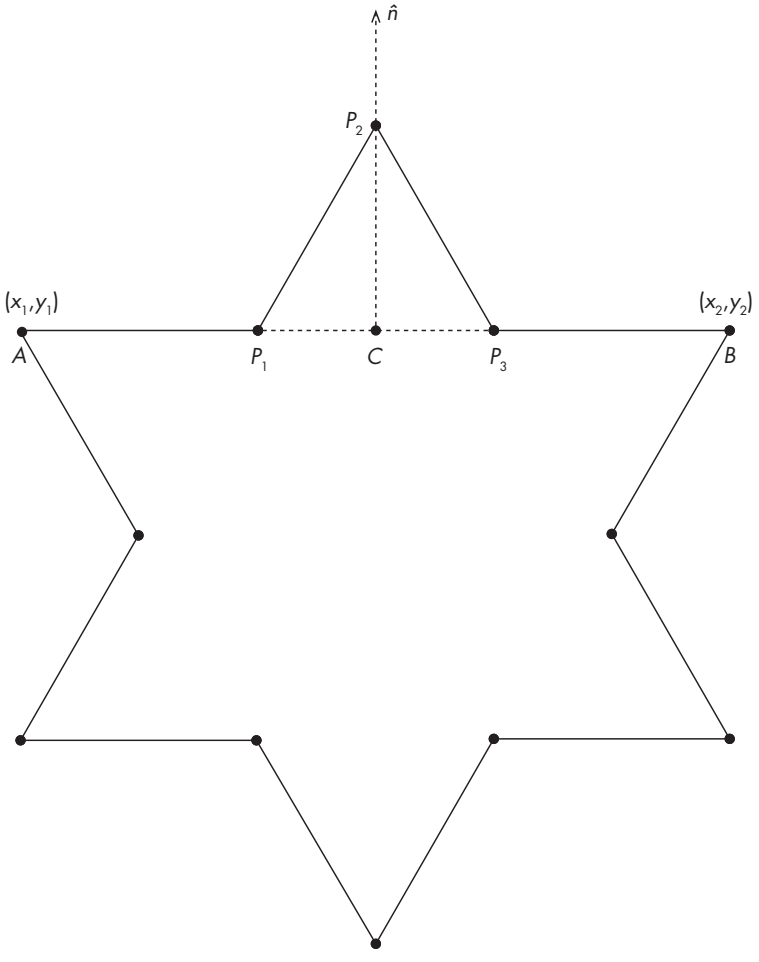

Figure 1-4: Combining three snowflakes

Now that you know how to calculate the coordinates for making the snowflake, let's see how to use those coordinates in Python to actually draw an image.

Drawing with turtle Graphics

In this chapter, you'll use Python's `turtle` module to draw the snowflake; it's a simple drawing program modeled after the idea of a turtle dragging its tail through the sand, creating patterns. The `turtle` module includes methods you can use to set the position and color of the pen (the turtle's tail) and many other useful functions for drawing.

As you'll see, all you need is a handful of graphics functions to draw the Koch snowflake. In fact, from the standpoint of `turtle`, drawing the snowflake is almost as easy as drawing a triangle. To prove it, and to give you a feel for how `turtle` works, the following program uses `turtle` to draw said triangle. Enter the code, save it as *test_turtle.py*, and run it in Python:

```
❶ import turtle

def draw_triangle(x1, y1, x2, y2, x3, y3, t):
    # go to start of triangle
  ❷ t.up()
  ❸ t.setpos(x1, y1)
  ❹ t.down()
    t.setpos(x2, y2)
    t.setpos(x3, y3)
    t.setpos(x1, y1)
    t.up()

def main():
    print('testing turtle graphics...')

  ❺ t = turtle.Turtle()
  ❻ t.hideturtle()

  ❼ draw_triangle(-100, 0, 0, -173.2, 100, 0, t)

  ❽ turtle.mainloop()

# call main
if __name__ == '__main__':
    main()
```

You start by importing the `turtle` module ❶. Next, you define the `draw_triangle()` method, whose parameters are three pairs of x-coordinates and y-coordinates (the three corners of a triangle), as well as `t`, a `turtle` object. The method starts by calling `up()` ❷. This tells Python to move the pen up; in other words, take the pen off the virtual paper so that it won't draw as you move the turtle. You want to position the turtle before you start drawing. The `setpos()` call ❸ sets the position of the turtle to the first pair of x- and y-coordinates. Calling `down()` ❹ sets the pen down, and for each of the subsequent `setpos()` calls, a line is drawn as the turtle moves to the next set of coordinates. The net result is a triangle drawing.

Next you declare a `main()` function to actually do the drawing. In it, you create the `turtle` object for drawing ❺ and hide the turtle ❻. Without this command, you'd see a small shape representing the turtle at the front of the line being drawn. You then call `draw_triangle()` to draw the triangle ❼, passing in the desired coordinates as arguments. The call to `mainloop()` ❽ keeps the `tkinter` window open after the triangle has been drawn. (`tkinter` is Python's default GUI library.)

Figure 1-5 shows the output of this simple program.

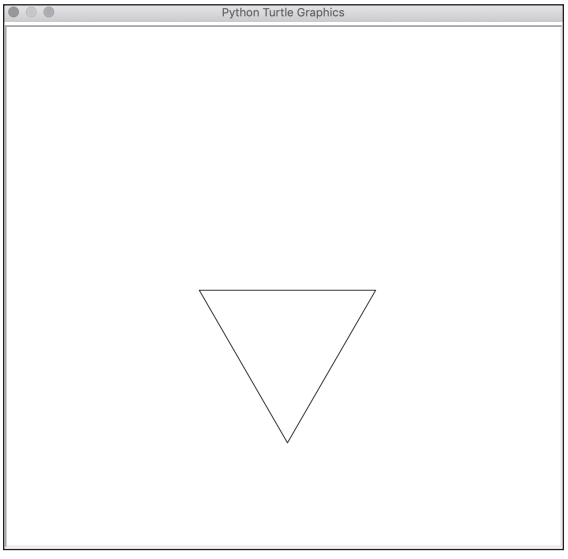

Figure 1-5: The output of a simple `turtle` program

You now have everything you need for the project. Let's draw some flakes!

Requirements

In this project, you'll use the Python `turtle` module to draw the snowflake.

The Code

To draw the Koch snowflake, define a recursive function, `drawKochSF()`. This function computes P_1, P_2, and P_3 in terms of A and B from Figure 1-2 and then recursively calls itself to perform the same calculation for smaller and smaller line segments until it reaches the smallest base case. Then it draws the flakes using `turtle`. For the full project code, skip ahead to "The Complete Code" on page 16. The code is also available in the book's GitHub repository at *https://github.com/mkvenkit/pp2e/blob/main/koch/koch.py*.

Calculating the Points

Begin the `drawKochSF()` function by calculating the coordinates for all the points needed to draw the basic flake pattern shown in Figure 1-2.

```
def drawKochSF(x1, y1, x2, y2, t):
    d = math.sqrt((x1-x2)*(x1-x2) + (y1-y2)*(y1-y2))
    r = d/3.0
    h = r*math.sqrt(3)/2.0
    p3 = ((x1 + 2*x2)/3.0, (y1 + 2*y2)/3.0)
    p1 = ((2*x1 + x2)/3.0, (2*y1 + y2)/3.0)
    c = (0.5*(x1+x2), 0.5*(y1+y2))
    n = ((y1-y2)/d, (x2-x1)/d)
    p2 = (c[0]+h*n[0], c[1]+h*n[1])
```

You define `drawKochSF()`, passing in the x- and y-coordinates for the endpoints of a line segment $\overline{AB}$, which forms the basis for one of the sides of the snowflake, as shown in Figure 1-4. You also pass in the `turtle` object `t`, which you use for the actual drawing. Then you compute all the parameters shown in Figure 1-2, as discussed in the "Computing the Snowflake" section, starting with `d`, the distance from A to B. Dividing `d` by 3 gives you `r`, the length of each of the four line segments that makes up a flake. You use `r` to find `h`, the height of the "cone" at the heart of the flake.

You calculate the rest of the parameters as tuples containing an x- and a y-coordinate. The `p3` and `p1` tuples describe the two points at the base of the cone portion of the flake. Point `c` is the midpoint of `p1` and `p3`, and `n` is the unit vector perpendicular to line $\overline{AB}$. Along with `h`, they help you calculate `p2`, the apex of the flake's cone.

Recursing

The next part of the `drawKochSF()` function uses recursion to break down the first-level flake into smaller and smaller versions of itself.

```
❶ if d > 10:
      # flake #1
    ❷ drawKochSF(x1, y1, p1[0], p1[1], t)
      # flake #2
      drawKochSF(p1[0], p1[1], p2[0], p2[1], t)
      # flake #3
      drawKochSF(p2[0], p2[1], p3[0], p3[1], t)
      # flake #4
      drawKochSF(p3[0], p3[1], x2, y2, t)
```

First you check for the recursion-stopping criteria ❶. If `d`, the length of segment $\overline{AB}$, is greater than 10 pixels, you continue the recursion. You do this by calling the `drawKochSF()` function again—four times! With each call, you pass in a different set of arguments corresponding to the coordinates for one of the four line segments that make up a flake, which you calculated at the start of the function. At ❷, for example, you call `drawKochSF()` for the segment $\overline{AB}_1$. The other function calls are for segments $\overline{P_1P_2}$, $\overline{P_2P_3}$, and $\overline{P_3B}$. Within each of these recursive calls, you'll perform a new set of calculations based on the new values for points A and B, and if `d` is still greater than 10 pixels, you'll make another four recursive calls to `drawKochSF()`, and so on.

Drawing a Flake

Now let's look at what happens if segment $\overline{AB}$ is less than 10 pixels. This is the base case for the recursive algorithm. Since you're below the threshold, you aren't going to recurse. Instead, you actually draw the four line segments that make up a single flake pattern and return from the function. You use the `up()`, `down()`, and `setpos()` methods from the `turtle` module, which you learned about in the "Drawing with `turtle` Graphics" section.

```
    else:
        # draw cone
        t.up()
      ❶ t.setpos(p1[0], p1[1])
        t.down()
        t.setpos(p2[0], p2[1])
        t.setpos(p3[0], p3[1])
        # draw sides
        t.up()
      ❷ t.setpos(x1, y1)
        t.down()
        t.setpos(p1[0], p1[1])
        t.up()
      ❸ t.setpos(p3[0], p3[1])
        t.down()
        t.setpos(x2, y2)
```

First you draw the cone formed by points `p1`, `p2`, and `p3` ❶. Then you draw lines $\overline{AB}_1$ ❷ and $\overline{P_3B}$ ❸. Since you already performed all the required calculations at the start of the function, drawing is simply a matter of passing the appropriate coordinates to the `setpos()` method.

Writing the main() Function

The `main()` function sets up a `turtle` object and calls `drawKochSF()`.

```
def main():
    print('Drawing the Koch Snowflake...')

    t = turtle.Turtle()
    t.hideturtle()

    # draw
    try:
      ❶ drawKochSF(-100, 0, 100, 0, t)
      ❷ drawKochSF(0, -173.2, -100, 0, t)
      ❸ drawKochSF(100, 0, 0, -173.2, t)
  ❹ except:
        print("Exception, exiting.")
        exit(0)

    # wait for user to click on screen to exit
  ❺ turtle.Screen().exitonclick()
```

In Figure 1-4, you saw how you were going to draw three of the snowflakes to get a hexagonally symmetric image as the final output. You do this by making three calls to `drawKochSF()`. The coordinates used for points *A* and *B* are `(-100, 0)`, `(100, 0)` for the first snowflake ❶, `(0, -173.2)`, `(-100, 0)` for the second ❷, and `(100, 0)`, `(0, -173.2)` for the third ❸. Notice that these are the same coordinates you used earlier to draw a triangle in your *test_turtle.py* program. Try to work out the coordinates for yourself. (Hint: $-173.2 \approx 100\sqrt{3}$.)

The `drawKochSF()` calls are enclosed in a Python `try` block to catch any exceptions that happen during drawing. For example, if you close the window while the drawing is still in process, an exception is thrown. You catch it in the `except` block ❹, where you print a message and exit the program. If you allow the drawing to complete, you'll get to `turtle.Screen().exitonclick()` ❺, which will wait until you close the window by clicking anywhere inside it.

Running the Snowflake Code

Run the code in a terminal as follows. Figure 1-6 shows the output.

```
$ python koch.py
```

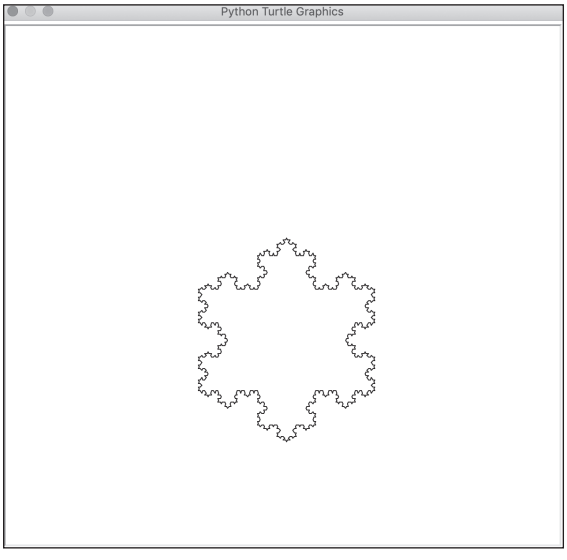

Figure 1-6: The Koch snowflake output

There's your beautiful snowflake!

Summary

In this chapter, you learned the basics of recursive functions and algorithms. You also learned how to draw simple graphics with Python's `turtle` module. You put these concepts together to create a nice drawing of an interesting fractal called the Koch snowflake.

Experiments!

Now that you have completed one fractal drawing, let's look at another interesting one called the *Sierpiński triangle*, named after the Polish mathematician Wacław Sierpiński. Figure 1-7 shows what it looks like.

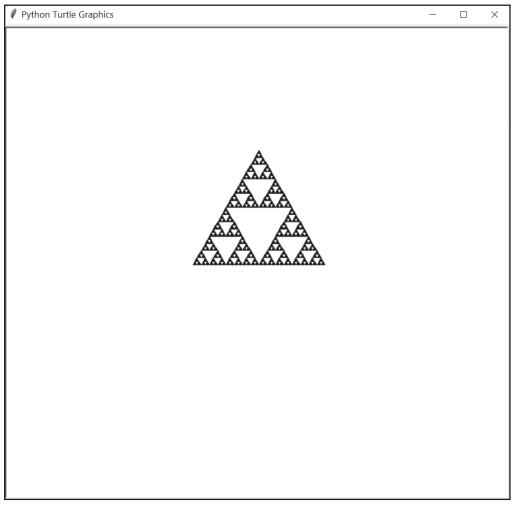

Figure 1-7: The Sierpiński triangle

Try drawing the Sierpiński triangle with `turtle` graphics. You can use a recursive algorithm like you did to draw the Koch snowflake. If you look at Figure 1-7, you'll see that the large triangle is divided into three smaller triangles, with an upside-down triangular hole in the middle. Each of the three smaller triangles is itself divided into another three triangles plus a hole in the middle, and so on. That gives you a hint on how to split up your recursion.

(The solution to this problem is in the GitHub repository for the book *https://github.com/mkvenkit/pp2e/blob/main/koch/koch.py*)

The Complete Code

Here's the complete code listing for this project:

```
"""
koch.py

A program that draws the Koch snowflake.

Author: Mahesh Venkitachalam
"""

import turtle
import math

# draw the recursive Koch snowflake
def drawKochSF(x1, y1, x2, y2, t):
    d = math.sqrt((x1-x2)*(x1-x2) + (y1-y2)*(y1-y2))
    r = d/3.0
    h = r*math.sqrt(3)/2.0
    p3 = ((x1 + 2*x2)/3.0, (y1 + 2*y2)/3.0)
    p1 = ((2*x1 + x2)/3.0, (2*y1 + y2)/3.0)
    c = (0.5*(x1+x2), 0.5*(y1+y2))
    n = ((y1-y2)/d, (x2-x1)/d)
    p2 = (c[0]+h*n[0], c[1]+h*n[1])
    if d > 10:
        # flake #1
        drawKochSF(x1, y1, p1[0], p1[1], t)
        # flake #2
        drawKochSF(p1[0], p1[1], p2[0], p2[1], t)
        # flake #3
        drawKochSF(p2[0], p2[1], p3[0], p3[1], t)
        # flake #4
        drawKochSF(p3[0], p3[1], x2, y2, t)
    else:
        # draw cone
        t.up()
        t.setpos(p1[0], p1[1])
        t.down()
        t.setpos(p2[0], p2[1])
        t.setpos(p3[0], p3[1])
        # draw sides
        t.up()
        t.setpos(x1, y1)
        t.down()
        t.setpos(p1[0], p1[1])
        t.up()
        t.setpos(p3[0], p3[1])
        t.down()
        t.setpos(x2, y2)

# main() function
def main():
    print('Drawing the Koch Snowflake...')
```

```
    t = turtle.Turtle()
    t.hideturtle()

    # draw
    try:
        drawKochSF(-100, 0, 100, 0, t)
        drawKochSF(0, -173.2, -100, 0, t)
        drawKochSF(100, 0, 0, -173.2, t)
    except:
        print("Exception, exiting.")
        exit(0)

    # wait for user to click on screen to exit
    turtle.Screen().exitonclick()

# call main
if __name__ == '__main__':
    main()
```