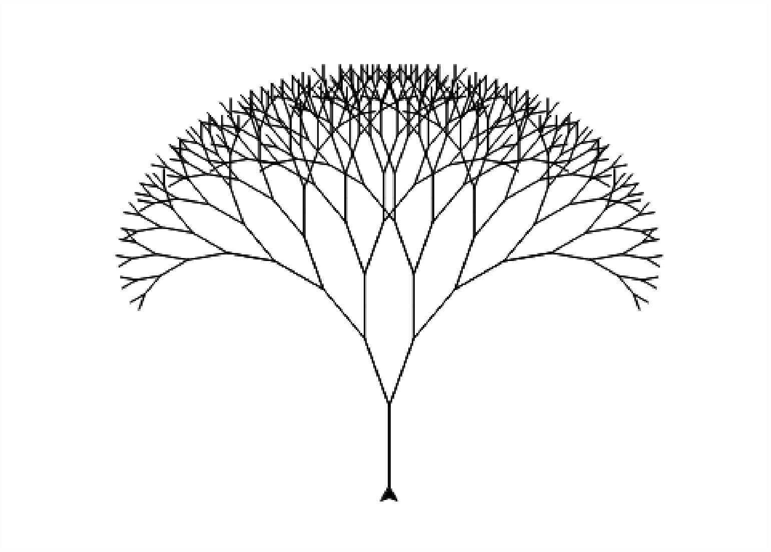
Chapter 9 - Recursion

Recursion

Lab - Recursion

Lab 1

Lab 1 - Recursive Tree



Recursive Tree

Trees can be drawn recursively. Draw a branch. At the end of the branch, draw two smaller branches with one to the left and the other to the right. Repeat until a certain condition is true. This program will walk you through drawing a tree in this way.

Start by importing the `turtle` module. Declare a turtle object, and define the function `recursive_tree`. This function should take three parameters, `branch_length`, `angle`, and `t`. Use `pass` as the function body for now. Finally, use `turtle.mainlooop()` at the end of the program.

```python
import turtle

t = turtle.Turtle()

def recursive_tree(branch_length, angle, t):
    """Draw a tree recursively"""
    pass

turtle.mainloop()
```

The base case for this function is a bit different. In previous examples, if the base case is true a value was returned. The function `recursive_tree` does not return a value, it draws on the screen. So the base case will be to keep recursing as long as `branch_length` is greater than some value. Define the base case as `branch_length` as being greater than 5. Use `pass` for the body of the conditional.

```python
def recursive_tree(branch_length, angle, t):
    """Draw a tree recursively"""
    if branch_length > 5:
        pass

turtle.mainloop()
```

Start drawing the tree by going forward and turning right. Call `recursive_tree` again, but reduce `branch_length` by 15. The code should run, but the tree will not look like a tree. It looks more like a curve made of series of line segments decreasing in size.

```python
def recursive_tree(branch_length, angle, t):
    """Draw a tree recursively"""
    if branch_length > 5:
        t.forward(branch_length)
        t.right(angle)
        recursive_tree(branch_length - 15, angle, t)

recursive_tree(45, 20, t)
turtle.mainloop()
```

The next step is to draw the branch that goes off to the left. Since the turtle turned to the right the number of degrees that the parameter `angle` represents, the turtle needs to turn to the left twice the degrees of `angle`. Turning to the left `angle` will put the turtle back at its original heading. The turtle needs to go further to the left. Then draw another branch whose length is reduced by 15.

```python
def recursive_tree(branch_length, angle, t):
    """Draw a tree recursively"""
    if branch_length > 5:
        t.forward(branch_length)
        t.right(angle)
        recursive_tree(branch_length - 15, angle, t)
        t.left(angle * 2)
        recursive_tree(branch_length - 15, angle, t)

recursive_tree(45, 20, t)
turtle.mainloop()
```

The tree is looking better, but there are two more things that need to be done. First, put the turtle back to its original heading by turning right `angle` degrees. Then go backwards the length of the branch. Call the `recursive_tree` function to draw a tree.

```
def recursive_tree(branch_length, angle, t):
    """Draw a tree recursively"""
    if branch_length > 5:
        t.forward(branch_length)
        t.right(angle)
        recursive_tree(branch_length - 15, angle, t)
        t.left(angle * 2)
        recursive_tree(branch_length - 15, angle, t)
        t.right(angle)
        t.backward(branch_length)

recursive_tree(45, 20, t)
turtle.mainloop()
```

challenge

What happens if you:

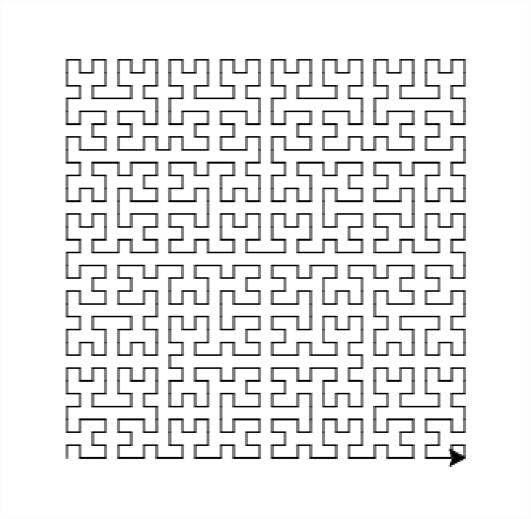
- Increase the branch length when calling `recursive_tree` for the first time?
- Increase and decrease the angle when calling `recursive_tree` for the first time?
- When decreasing `branch_length`, change 15 to something smaller (be sure to change all of the 15’s)?
- Change the base case to `if branch_length > 1:`?
- Rotate the turtle 90 degrees to the left before calling `recursive_tree` for the first time?

▼ **Solution**

```python
import turtle

t = turtle.Turtle()
t.lt(90)
t.penup()
t.backward(150)
t.pendown()
t.speed(10)

def recursive_tree(branch_length, angle, t):
    """Draw a tree recursively"""
    if branch_length > 1:
        t.forward(branch_length)
        t.right(angle)
        recursive_tree(branch_length - 7, angle, t)
        t.left(angle * 2)
        recursive_tree(branch_length - 7, angle, t)
        t.right(angle)
        t.backward(branch_length)

recursive_tree(60, 20, t)
turtle.mainloop()
```

Lab 2

Lab 2 - The Hilbert Curve



Hilbert Curve

The Hilbert Curve is a fractal, space-filling curve. Start by importing the turtle module, create a turtle object, and write the function header for the recursive function `hilbert`. The parameters for the function are the distance the turtle will travel, the rule to be used, an angle (determines how tight the fractal is), depth (how intricate the fractal is), and the turtle object. Use `pass` for the body for now.

```
import turtle

t = turtle.Turtle()

def hilbert(dist, rule, angle, depth, t):
    """Draw a Hilber Curve"""
    pass

turtle.mainloop()
```

The base case for the function is when `depth` is 0. Another way to think about the base case is that if `depth` is greater than 0, keep drawing the fractal. Use `if depth > 0:` as the base case. Also, there are two rules for the turtle. Ask if `rule` is equal to 1 or if it is equal to 2. Use `pass` for the body of these conditionals.

```python
def hilbert(dist, rule, angle, depth, t):
    """Draw a Hilber Curve"""
    if depth > 0:

        if rule == 1:
            pass

        if rule == 2:
            pass

turtle.mainloop()
```

If `rule` is equal to 1, then the turtle is going to turn left, recursively call the `hilbert` function with `rule` set to 2, go forward, turn right, recursively call the `hilbert` function with `rule` set to 1, go forward, recursively call the `hilbert` function with `rule` set to 1, turn right, and finally move forward. Because the base case is based on `depth`, it must be reduced by 1 each time the `hilbert` function is called recursively.

```python
        if rule == 1:
            t.left(angle)
            hilbert(dist, 2, angle, depth - 1, t)
            t.forward(dist)
            t.right(angle)
            hilbert(dist, 1, angle, depth - 1, t)
            t.forward(dist)
            hilbert(dist, 1, angle, depth - 1, t)
            t.right(angle)
            t.forward(dist)
            hilbert(dist, 2, angle, depth - 1, t)
            t.left(angle)
```

If `rule` is equal to 2, then the code is almost the inverse of when `rule` is equal to 1. The turtle will still go forward, but left turns become right turns, right turns become left turns, and recursive calls to `hilbert` will use 2 instead of 1 for the `rule` parameter (and vice versa).

```
        if rule == 2:
            t.right(angle)
            hilbert(dist, 1, angle, depth - 1, t)
            t.forward(dist)
            t.left(angle)
            hilbert(dist, 2, angle, depth - 1, t)
            t.forward(dist)
            hilbert(dist, 2, angle, depth - 1, t)
            t.left(angle)
            t.forward(dist)
            hilbert(dist, 1, angle, depth - 1, t)
            t.right(angle)
```

Finally, call the `hilbert` function and run the program to see the fractal.

```
hilbert(5, 1, 90, 5, t)
turtle.mainloop()
```

▼ **Speeding up the turtle**

The Hilbert Curve can be slow to draw. You can change the speed of the turtle with the following command `t.speed(10)`.

challenge

What happens if you:

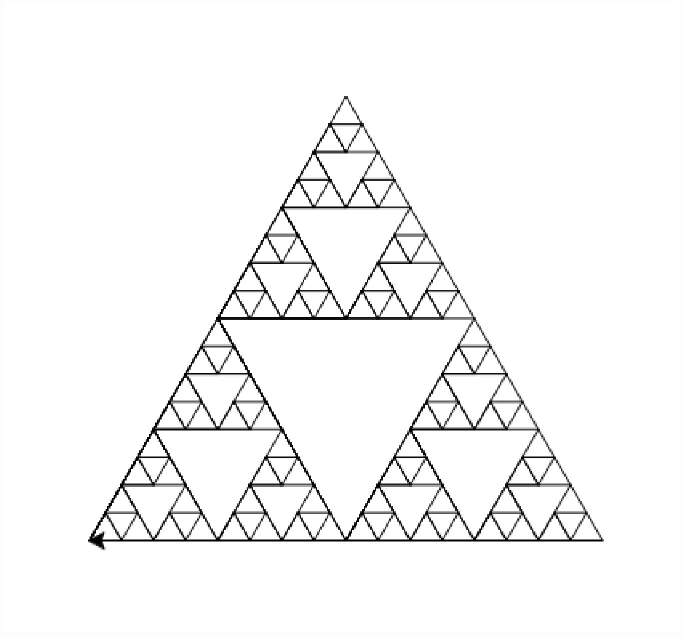
- Change the `dist` parameter?
- Start with the `rule` parameter as 2?
- Increase or decrease the `angle` parameter?
- Increase or decrease the `depth` parameter?

▼ **Solution**

```python
import turtle

t = turtle.Turtle()

def hilbert(dist, rule, angle, depth, t):
    if depth > 0:

        if rule == 1:
            t.left(angle)
            hilbert(dist, 2, angle, depth - 1, t)
            t.forward(dist)
            t.right(angle)
            hilbert(dist, 1, angle, depth - 1, t)
            t.forward(dist)
            hilbert(dist, 1, angle, depth - 1, t)
            t.right(angle)
            t.forward(dist)
            hilbert(dist, 2, angle, depth - 1, t)
            t.left(angle)

        if rule == 2:
            t.right(angle)
            hilbert(dist, 1, angle, depth - 1, t)
            t.forward(dist)
            t.left(angle)
            hilbert(dist, 2, angle, depth - 1, t)
            t.forward(dist)
            hilbert(dist, 2, angle, depth - 1, t)
            t.left(angle)
            t.forward(dist)
            hilbert(dist, 1, angle, depth - 1, t)
            t.right(angle)

hilbert(5, 1, 90, 5, t)
turtle.mainloop()
```

Lab 3

Lab 3 - Sierpinski Triangle


Sierpinski Triangle

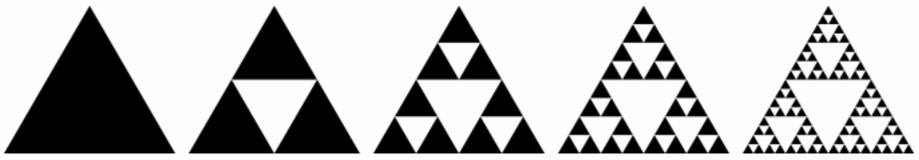
If you start to zoom in on fractals, you will see the same shapes repeat themselves. Fractals are said to be self-similar, which means they can be drawn with recursion. This lab will walk you though drawing a Sierpinski triangle. Start by preparing the program to use Python’s turtle graphics. Sierpinski triangles can become quite complex, so increase the turtle’s speed to 10 (the maximum).

```
import turtle

t = turtle.Turtle()
t.speed(10)

turtle.mainloop()
```

The building block of this fractal is the triangle. Create a function (with a parameter for length) to draw a triangle. The turtle will be walking all over the screen, so it is important to make sure that the turtle is facing a consistent position before drawing the triangle. `t.setheading(180)` ensures the turtle is facing to the left.

```
import turtle

t = turtle.Turtle()
t.speed(10)

def draw_triangle(length):
    t.setheading(180)
    for i in range(3):
        t.rt(120)
        t.fd(length)

draw_triangle(50)

turtle.mainloop()
```

Look closely at a Sierpinski triangle, and you will see clusters of three triangles that make up clusters of triangles and so forth.


Sierpinski Triangle Evolution

You are now going to create a recursive function that draws this cluster of three triangles. Define the function `sierpinski` that takes `length` and `n` as parameters. The base case is if `n` is equal to 1. If so, draw a triangle of size `length`. If `n` is not equal to 1, then you are going to call `sierpinski` again, but with `n-1`. These new triangles need to be in a different position, so move the turtle after drawing each turtle. Finally, replace the `draw_triangle` function call with `sierpinski(50, 1)`.

```
import turtle

t = turtle.Turtle()
t.speed(10)

def sierpinski(length, n):
    if n == 1:
        draw_triangle(length)
    else:
      sierpinski(length, n-1)
      t.rt(120)
      t.fd(length)
      sierpinski(length, n-1)
      t.lt(120)
      t.fd(length)
      sierpinski(length, n-1)
      t.fd(length)

def draw_triangle(length):
    t.setheading(180)
    for i in range(3):
        t.rt(120)
        t.fd(length)

sierpinski(50, 1)

turtle.mainloop()
```

challenge

What happens if you:

- Change the function call to `sierpinski(50, 2)`?
- Change the function call to `sierpinski(50, 3)`?
- Change the function call to `sierpinski(50, 4)`?

The triangles are clustered together, but the Sierpinski triangle has larger triangle-shaped voids. An adjustment needs to be made to the distance the turtle moves between calls to the `sierpinski` function. Instead of moving forward the distance of `length`, the turtle will move forward `length * (n-1)`. Change the `sierpinski` function call to `sierpinski(20, 4)`.

```
import turtle

t = turtle.Turtle()
t.speed(10)

def sierpinski(length, n):
    if n == 1:
        draw_triangle(length)
    else:
      sierpinski(length, n-1)
      t.rt(120)
      t.fd(length * (n-1))
      sierpinski(length, n-1)
      t.lt(120)
      t.fd(length * (n-1))
      sierpinski(length, n-1)
      t.fd(length * (n-1))

def draw_triangle(length):
    t.setheading(180)
    for i in range(3):
        t.rt(120)
        t.fd(length)

sierpinski(20, 4)

turtle.mainloop()
```

The fractal is getting better, but there are a few areas where the program can be improved. Change the distance the turtle goes forward from `t.fd(length * (n-1))` to `t.fd(length * 2 ** (n-2))`.

```
import turtle

t = turtle.Turtle()
t.speed(10)

def sierpinski(length, n):
    if n == 1:
        draw_triangle(length)
    else:
      sierpinski(length, n-1)
      t.rt(120)
      t.fd(length * 2**(n-2))
      sierpinski(length, n-1)
      t.lt(120)
      t.fd(length * 2**(n-2))
      sierpinski(length, n-1)
      t.fd(length * 2**(n-2))

def draw_triangle(length):
    t.setheading(180)
    for i in range(3):
        t.rt(120)
        t.fd(length)

sierpinski(20, 4)

turtle.mainloop()
```

challenge

What happens if you:

- Change the `sierpinski` function call to `sierpinski(5, 6)`?
- Change the `sierpinski` function call to `sierpinski(5, 8)`?

Lab Challenge

Lab Challenge

Problem
Write a recursive function called `recursive_power` that takes two integers as parameters. The first parameter is the base and the second parameter is the exponent. Return the base parameter to the power of the exponent.

Expected Output
* If the function call is `recursive_power(5, 3)`, then the function would return `125`
* If the function call is `recursive_power(4, 5)`, then the function would return `1024`

Code Visualizer