The term **Mandelbrot set** is used to refer both to a general class of fractal sets and to a particular instance of such a set. In general, a Mandelbrot set marks the set of points in the complex plane such that the corresponding Julia set is connected and not computable.
"The" Mandelbrot set is the set obtained from the quadratic recurrence equation

$$z_{n+1} = z_n^2 + C$$

with z0=C, where points C in the complex plane for which the orbit of zn does not tend to infinity are in the set. Setting z0 equal to any point in the set that is not a periodic point gives the same result. The Mandelbrot set was originally called a μ molecule by Mandelbrot. J. Hubbard and A. Douady proved that the Mandelbrot set is connected.

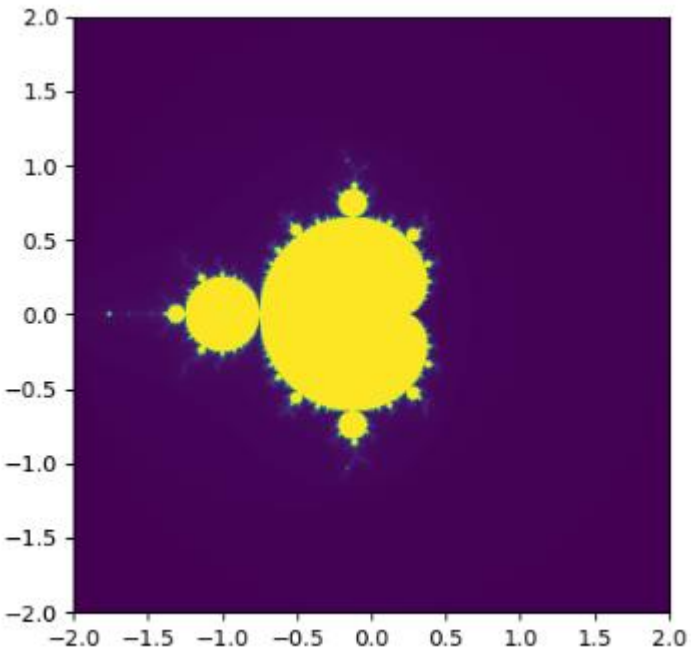
Code in python:

```
import numpy as np
import matplotlib.pyplot as plt


def mandelbrot(c, max_iter):
    z = 0
    n = 0
    while abs(z) <= 2 and n < max_iter:
        z = z**2 + c
        n += 1
    return n


def mandelbrot_set(xmin,xmax,ymin,ymax,width,height,max_iter):
    r1 = np.linspace(xmin, xmax, width)
    r2 = np.linspace(ymin, ymax, height)
    return (r1,r2,np.array([[mandelbrot(complex(r, i),max_iter) for r in r1]
for i in r2]))


def display(xmin,xmax,ymin,ymax,width,height,max_iter):
    d = mandelbrot_set(xmin,xmax,ymin,ymax,width,height,max_iter)
    plt.imshow(d[2], extent=(xmin, xmax, ymin, ymax))
    plt.show()


# Set the range and dimensions
xmin, xmax, ymin, ymax = -2.0, 2.0, -2.0, 2.0
width, height = 1000, 1000
max_iter = 256


# Display the Mandelbrot set
display(xmin, xmax, ymin, ymax, width, height, max_iter)
```

- def Mandelbrot (c, max_iter) – function that computes whether a given complex number c is in the Mandelbrot set within a maximum nr of iterations
- z = 0 - initializing variable z to 0
- n = 0 – variable keeps track of the nr of iterations performed
- while abs(z) <= 2 and n < max_iter – a while loop that continues iterating until either the absolute value of z becomes greater than 2 or the maximum nr of iterations is reached
- z = z**2 + c – updates the value of z according to Mandelbrot formula, where z is complex number and c is the input complex number
- n += 1 – increments the iteration count n by 1
- return n – returns the nr of iterations it took for the sequence to escape the threshold
- def mandelbrot_set (xmin, xmax, ymin, ymax, width, height, max_iter) – defines a function that generates the Mandelbrot set within a specified range and dimensions
  xmin and xmax are the minimum and maximum values of the real part(horizontal axis)
  ymin and ymax represent the minimum and maximum values of the imaginary part (vertical axis) of the complex nr
  width, height -values that represent the dimension of the image that will be generated
  max_iter – variable determines the maximum nr of iterations that will be performed for each point in the complex plane during the calculation of the Mandelbrot set
- r1 = np.linspace(xmin, xmax,width)- this array represent the real part of the complex nr
  array of evenly spaced nr over a specified range xmin to xmax with a specified nr of elements ‘width’
- r2 = np.linspace(ymin, ymax, height)- this array represents the imaginary part of the complex nr
  array of evenly spaced nr over a specified range ymin to ymax and specified nr of elements height
- return (r1,r2,np.array([[mandelbrot(complex(r, i),max_iter) for r in r1] for i in r2]))- returns a tuple containing three elements:
  r1: The array representing the real part of the complex numbers.
  r2: The array representing the imaginary part of the complex numbers.
  np.array([[mandelbrot(complex(r, i),max_iter) for r in r1] for i in r2]) - array representing the Mandelbrot set. Each element of the array corresponds to the number of iterations taken for the corresponding complex number to escape the threshold.

- def display (xmin,xmax,ymin,ymax,width,height,max_iter) - function generates and displays the Mandelbrot set within the specified range and dimensions.
- d = mandelbrot_set(xmin,xmax,ymin,ymax,width,height,max_iter)- generates the Mandelbrot set using the mandelbrot_set function with the specified parameters and assigns it to the variable d.
- plt.imshow(d[2], extent=(xmin, xmax, ymin, ymax) - displays the Mandelbrot set as an image using Matplotlib's imshow function. It takes the 2D NumPy array representing the Mandelbrot set (d[2]) as input and sets the extent of the image to the specified range (xmin, xmax, ymin, ymax).
- plt.show() - displays the plot of the Mandelbrot set

OUTPUT:



The Koch snowflake is a fractal curve, also known as the Koch island, which was first described by Helge von Koch in 1904. It is built by starting with an equilateral triangle, removing the inner third of each side, building another equilateral triangle at the location where the side was removed, and then repeating the process indefinitely.

The Mathematical Interpretation:

NUMBER OF SIDES (n)
One side of the figure from the previous stage becomes four sides in the subsequent step for every iteration. The equation for the number of sides in the Koch Snowflake, given that we start with three sides, is

in the a-th iteration

$$n = 3*4a.$$

The number of sides for iterations 0, 1, 2, and 3 is 3, 12, 48, and 192, in that order.

A SIDE'S LENGTH (length)
Every iteration has a side that is one-third as long as the side from the stage before. The length of a side of an equilateral triangle with side length x is, if we start with that triangle.

length = x*3-a

for iterations 0 to 3

PERIMETER (p)
Since each iteration of the Koch Snowflake has the same number of sides, the perimeter may be calculated by multiplying the total number of sides by each side's length.

$$p = n*length$$
$$p = (3*4a)*(x*3-a)$$

for the a-th iteration.

Again, the perimeter is 3a, 4a, 16a/3, and 64a/9 for the first four iterations (0 to 3).

We may rewrite the calculation as follows because, as we can see, the perimeter grows by 4/3 times each iteration.

p = (3a)*(4/3)a

Therefore, the perimeter increases without end as a-->infinity.

Furthermore, the snowflake has sharp corners with no smooth lines joining them because it is a-->infinity. As a result, although the snowflake's perimeter, which is an infinite series, is continuous due to the lack of breaks, it is not differentiable due to the absence of smooth lines.

Code in python:

```
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()
    t.speed("fastest")  # Set the turtle's speed to the fastest
    turtle.tracer(False)  # Turn off animation

    # draw
```

```
    try:
        drawKochSF(-100, 0, 100, 0, t)
        drawKochSF(0, -173.2, -100, 0, t)
        drawKochSF(100, 0, 0, -173.2, t)
    except Exception as e:
        print("Exception:", e)
        exit(0)

    turtle.Screen().update()  # Update the screen after drawing
    turtle.Screen().exitonclick()  # Wait for user to click on screen to exit

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

- def drawKochSF(x1, y1, x2, y2, t)- function which takes four pairs of coordinates (x1, y1) and (x2, y2) to draw a line segment between them using the turtle t
- d = math.sqrt((x1-x2)*(x1-x2) + (y1-y2)*(y1-y2)) -calculates the distance d between points (x1, y1) and (x2, y2) using the distance formula
- r = d/3.0 - calculates one-third of the distance d to determine the length of each segment in the Koch snowflake
- h = r*math.sqrt(3)/2.0- computes the height h of an equilateral triangle using the side length r
- p3 = ((x1 + 2*x2)/3.0, (y1 + 2*y2)/3.0)- calculates the coordinates of the point p3, which is two-thirds of the way from point (x1, y1) to point (x2, y2)
- p1 = ((2*x1 + x2)/3.0, (2*y1 + y2)/3.0) -calculates the coordinates of the point p1, which is two-thirds of the way from point (x2, y2) to point (x1, y1)
- c = (0.5*(x1+x2), 0.5*(y1+y2))-calculates the coordinates of the midpoint c between points (x1, y1) and (x2, y2)
- n = ((y1-y2)/d, (x2-x1)/d)-calculates the normal vector n to the line segment (x1, y1) - (x2, y2)
- p2 = (c[0]+h*n[0], c[1]+h*n[1])-calculates the coordinates of the point p2, which is h units away from point c in the direction of the normal vector n.
- if d > 10 -if statement checks if the distance d between points (x1, y1) and (x2, y2) is greater than 10. If it is, the line segment is further divided into smaller segments using recursion. Otherwise, the function draws the line segment and ends.
- t.up() - lifts the turtle's pen off the screen.
- t.setpos(p1[0], p1[1]) -moves the turtle to the coordinates of point p1.
- t.down()-puts the turtle's pen back on the screen.
- t.setpos(p2[0], p2[1]) -draws a line segment from the turtle's current position to point p2.

- t.setpos(p3[0], p3[1])-draws a line segment from the turtle's current position to point p3.
- t.setpos(x1, y1)- moves the turtle to the starting coordinates (x1, y1).
- def main()-defines the main() function.
- t = turtle.Turtle()-creates a turtle object named t.
- try: ... except Exception as e: ...- uses a try-except block to catch any exceptions that occur during drawing.
- turtle.Screen().update()-updates the screen after drawing to ensure all changes are visible.
- if __name__ == '__main__'-checks if the script is being run directly as the main program.
- main()-call the main() function to start the program execution.