FRACTAL GENERATOR

Project Report

Name: Naman Vohra

University: Binus University International

Course: Computer Science

Subject: Program Design Methods

ID: 2201798420

Semester: Odd Semester (1)

Submission date: 22-11-2018

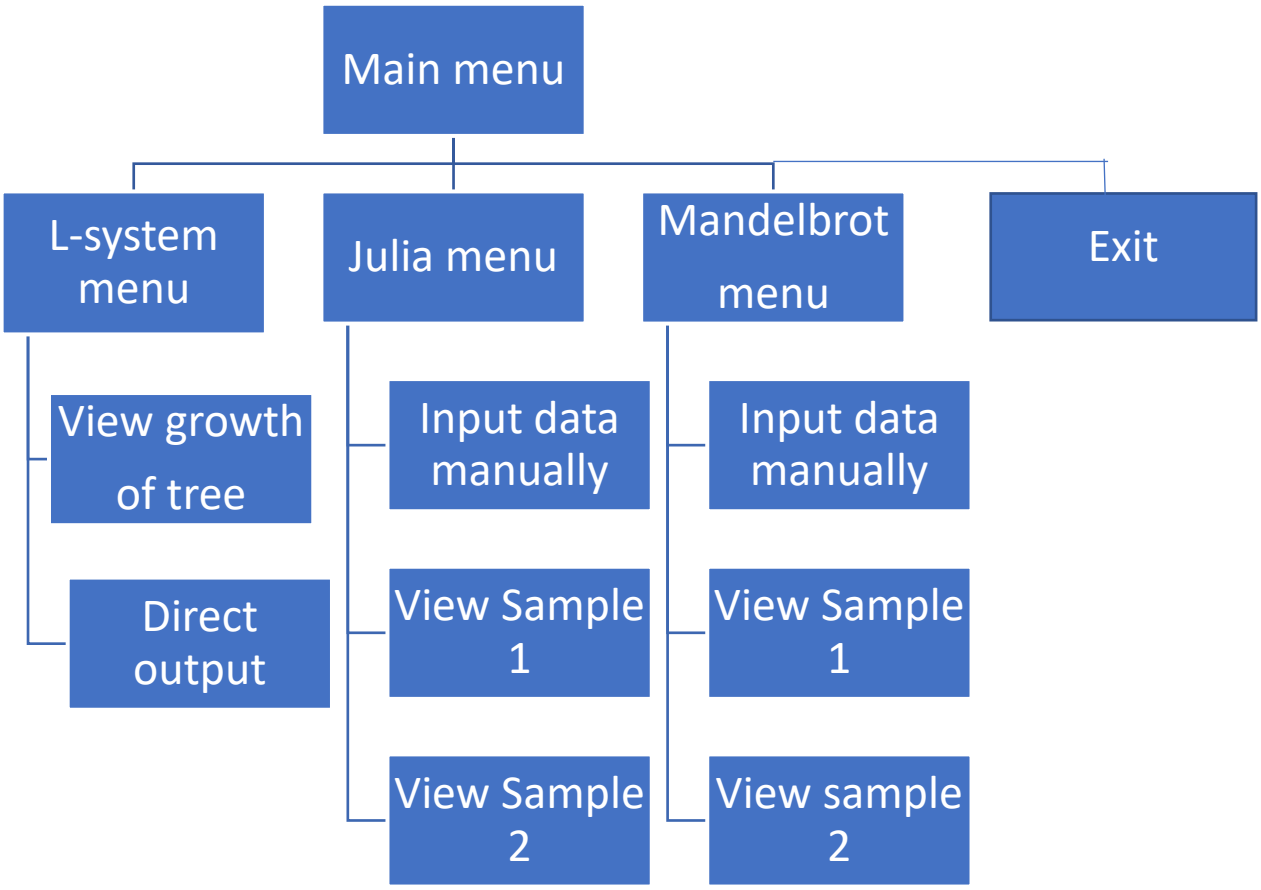
1. DESCRIPTION

The project is designed to generate 3 types of fractals. It includes the L-System, Mandelbrot, and Julia fractals. The purpose of making this project is to investigate phenomena involving complex geometry, patterns and scaling.

This program allows users to experiment with formulas(for L-system), c-values(for Julia fractals) and zoom values(for Mandelbrot set).

II.a. DESIGN/PLAN

Project's Hierarchy Chart



II.b. Explanation of Functions

Main Menu: (Main.py)

- Outside main()
  -Imports classes such as “Transformation”, “Tree”, “MandelbrotGenerator”, “JuliaGenerator”, and the module “numpy”.
- Inside main()
  --The line while True to repeat the code inside main() until the user chooses option 4, which is to exit the menu.
  --Welcome message and menu list printed.
  --If the user selects option 1, the user will be asked to input the axiom, number of rules that wanted to be implemented, letter that the user wants to transform, rule corresponding to that letter, number of times the rule(s) is to be applied and angle. At the end, he will be asked whether he wants to visualize growth of leaves of tree.
  --If the user chooses option 2, the user will be given 3 further options, the first option is to input c values manually, the second one is to view sample 1 and the third one is to view sample 2. If the user chooses sample 1, the Julia image will be displayed. Sample 2 generates different Julia image. If the user wants to experiment with different values and try colormaps, option one can be chosen.
  --If the user chooses option 3, the user will be given 3 further options, the first option is to input the zoom values and color map manually, the second one is to view sample 1 and the third one is to view sample 2. If the user opts for option 1 a Mandelbrot image will be produced and for option 2 a zoomed Mandelbrot image will be produced.

SequenceTransformer.py

- Class AxiomTransformer:
  Public member
  -sequence: string
  -transformations: dictionary
  -iterations: integer
  -angle2: float

- AxiomTransformer ():
  def iteration:
  -The function starts with a for loop to indicate how many times the rule is to be applied to the sequence(axiom).

-Each iteration will give a longer sequence(string) as an output. The sequence will be transformed into the desired value if it matches with the key(s) of the dictionary.

def get_angle:
-To return the angle value.

CoordinatesPlotter.py

- Class PlotCords:
  Public Member
  -x_cords: list
  -y_cords: list

- **def plot_graph:**
  -Plots the graph(L-system)

CoordinatesGenerator.py

- Class Tree(PlotCords):
  Public Member
  -name_of_tree: string
  -input_command: string
  -turn_amount: float
  -x: integer
  -y: integer
  -angle: float
  -x_cords: list
  -y_cords: list
  -x_y_cords: list
  -saved_x_cords: list
  -saved_y_cords: list
  -angle1: list
  -saved_angle1: list

- Inside class Tree(PlotCords)
  def coordinates_generator:
  --If the code detects any capital letter(A-Z) in the sequence (input_command), the new x and y values will be produced according to the latest angle (starts initially with 90 degrees) and each of the value will be appended to their respective list (x_cords, y_cords). The angle too will be appended to the list which stores the angle (angle1). This will leave a line/trace on the graph.
  --If the code detects any small letter(a-z) in the axiom, it will have the same effect as the previous case (if it detects any capital letter), but it will not plot any line on the graph, meaning it will only update the coordinate -s. This is the reason why float('nan') is appended to each of the list, so that nothing will be plotted on the graph.
  --If the code detects "+" sign, it will add the current angle value with the turn amount value (angle which we input) and the angle will be stored to the list angle1.

-- If the code detects "-" sign, it will deduct the current angle value with the turn amount value (angle which we input) and the angle will be stored to the list angle1.
--If "["is present in the input, it will add the latest values of x, y and angle to saved_x_cords, saved_y_cords and saved_angle1 respectively. It is like saving the current position of the "pointer".
--If "]"is present in the input, it will access the coordinates and angle of the "saved" pointer, therefore the method pop() is used as it removes and return the current coordinates in saved_x_cords, saved_y_cords and not to forget, the current angle in saved_angle1.

def return_x_cords:
--return the x coordinates.

def return_y_cords:
--return the y coordinates.

Julia_Generator.py

- Class JuliaGenerator
  Public Members
  -screen_width: integer
  -screen_height: integer
  -scale: integer
  -x_cords: numpy.ndarray
  -y_cords: numpy.ndarray
  -Z: numpy.ndarray
  -C: numpy.ndarray
  -M: numpy.ndarray
  -N: numpy.ndarray
  -cmap_type: string

- Inside class JuliaGenerator:
  def julia_calculator:
  -For loop to iterate the code 256 times.
  -Implementation of Julia formula z = z^2 + c. Before explaining further, I would like to explain what the variables Z, C, M, N stores. Z stores the pixels which now are in the form of complex numbers. C stores the c values in form of complex numbers. M stores the boolean value True and N contains zeros. Note that these variables are 2D arrays.
  - In the program we used Boolean indexing (Z[M] = Z[M] * Z[M) + C[M]). Z[M] means selecting the elements of the matrix Z for which M contains true. Similarly, it chooses the elements of the matrix C for which M contains true.
  -For the values of array Z which pass the threshold (2), M will store false. (M[abs(Z) > 2] = False) and next time the loop iterates again, it will iterate on pixels that haven't escaped yet (the 'True' pixels).
  -The next line contains the code N[M] = i, which adds colour to the Julia fractals.

def plot_julia:
-Remove the axes and plots the Julia fractal.

MandelbrotGenerator.py

- Class MandelbrotGenerator:

Public Members

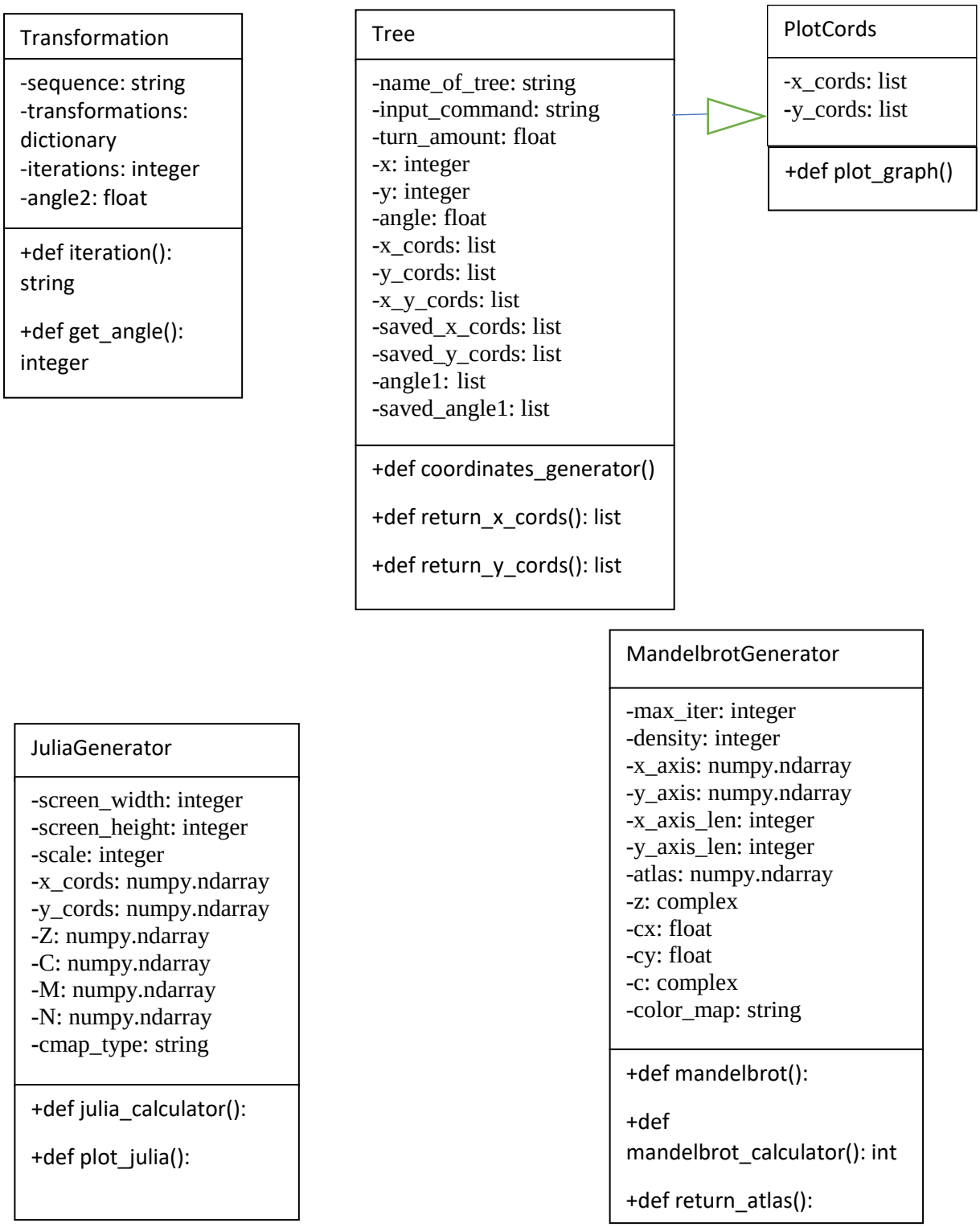
-max_iter: integer
-density: integer
-x_axis: numpy.ndarray
-y_axis: numpy.ndarray
-x_axis_len: integer
-y_axis_len: integer
-atlas: numpy.ndarray
-z: complex
-cx: float
-cy: float
-c: complex
-color_map: string

def mandelbrot
-iterates through each coordinate in the complex plane. Afterward, each pair of coordinates is converted into complex form and passed to the function mandelbrot_calculator (see below).

def mandelbrot_calculator:
--Has the same formula as Julia fractal (z = z^2 + c). The difference is that the z value starts with 0 and the c value stores the coordinate of the complex plane (unlike Julia where the z array contains the coordinates of the complex plane and the c value is fixed)
--for loop to iterate the code, and if the z value passes the threshold, it will return iteration and pass the iteration to mandelbrot function where It will be stored in the atlas.

def return_atlas:
--plots the mandelbrot fractal.

CLASS DIAGRAM



III.a. Lessons that have been learned

1. The use of pop method and float('nan')
   -I used these concepts in the code which generates the L-system fractal. Pop is used to access the latest coordinates and angle which were saved. Float('nan') is used in case the user does not want any line to be drawn on the graph, meaning if he only wish to update the position of coordinate. 'nan' stands for not a number.

```
        self.x_cords.append(float('nan'))
        self.x_cords.append(self.x)
        self.y = self.y + (sin(self.angle * pi / 180))
        self.y_cords.append(float('nan'))
```

```
self.x = self.saved_x_cords.pop()
self.y = self.saved_y_cords.pop()
self.angle = self.saved_angle1.pop()
```

2. The use of numpy methods like linspace, tile, and zeros
   -I know that I have to use array concepts in the code which plots Julia fractals. With the help of linspace method, I was able to create list of coordinates. I just have to input the margins and the number of equally spaced coordinates that I wish to be present between the margins. It also have additional method "reshape" which can be attached to linspace. With reshape I can change the list order (matrix shape), which means I can decide the number of rows and columns my matrix can have. Tile helps to repeat the elements inside array. With the help of linspace an tile, I was able to create 2D array which represents complex plane. Zeros create a 2D array of 0 values and np.full can create a 2D array consisting of elements we want (for example "True", which can be used for boolean indexing).

```
x_cords = np.linspace(-screen_width / scale, screen_width / scale, num=screen_width).reshape((1,
                                                                                              screen_width))

y_cords = np.linspace(-screen_height / scale, screen_height / scale, num=screen_height).reshape(
    (screen_height, 1))
z = np.tile(x_cords, (screen_height, 1)) + 1j * np.tile(y_cords, (1, screen_width))
m = np.full((screen_height, screen_width), True, dtype=bool)
n = np.zeros((screen_height, screen_width))
```

3. Passing values between functions in one class

   -by putting self. To the function which we want to pass the value to, I can pass values between functions.

```
    for ix in range(self.x_axis_len):
        for iy in range(self.y_axis_len):
            self.cx = self.x_axis[ix]
            self.cy = self.y_axis[iy]
            self.c = complex(self.cx, self.cy)
            # test the c values and iteration stored in atlas
            self.atlas[ix, iy] = self.mandelbrot_calculator(self.c, self.max_iter)
            pass
        pass

# In mandelbrot set the z value starts with 0
@jit
def mandelbrot_calculator(self, c, max_iter):
    # z = complex(0, 0) is the same as 0 + 0j
    self.z = complex(0, 0)
    for iteration in range(max_iter):
        self.z = (self.z * self.z) + self.c
        # checks if the magnitude of z i greater than 4
        if abs(self.z) > 4:
```

III.b. Problems that have been overcome

To complete this project was a challenging task for me. The main problem was understanding the concept of fractals, especially the Julia fractals and Mandelbrot fractals and how to implement the concepts on the code. To create the fractals, I have to grasp mathematical concepts like complex numbers, vectors and matrix. I have to learn the basic of complex numbers to understand the formula of the fractals and had to read articles and the book; Make Your Own Mandelbrot by Tariq Rashid.

For the coding part, I knew that I will be using matplotlib, but I need another library which handles mathematical operations well. This library is called numpy. It took me a while to understand the methods available in this library. This Library helped me a lot and this resulted in relatively quick computation time in generating the Julia fractals.

Apart from this, I had several careless mistakes which took a bit of time to realize.

IV. Source code

```
from SequenceTransformer import AxiomTransformer
from Coordinates_Generator import Tree
from MandelbrotGenerator import MandelbrotGenerator
from Julia_Generator import JuliaGenerator
import numpy as np


def main():
    while True:
        print("===============**WELCOME TO FRACTALS GENERATOR**===============")
        print("What type of fractals you want to generate?")
        print("--L-system(No.1)\n--Julia(No.2)\n--Mandelbrot(No.3)\n--Exit(No.4)")
        input_fractal_type = int(input("Please input the number"))
        x = 0
        y = 0
        angle = 90
        cords_x = [0]
        cords_y = [0]
        saved_x_cords = []
        saved_y_cords = []
        angle1 = []
        saved_angle1 = []
        if input_fractal_type == 1:
```

```
    input_axiom = str(input("Please input axiom"))
    rules = {}
    input_no_of_rules = int(input("The number of transformation rules you want to apply"))
    # How many rules to be applied to axiom and processed axiom
    for i in range(input_no_of_rules):
        input_rules_key = str(input("Please input letter you want to transform"))
        input_rules_value = str(input("Please input rule corresponding to that letter"))
        rules[input_rules_key] = input_rules_value
    no_of_times_rule_be_applied = int(input("How many times you want the rule to be applied"))
    angle_of_tree = float(input("Please input tree angle"))

    input_choice = int(input("Do you wish to visualize growth of leaves?\nYes(No.1)\nNo(No. 2)"))
    if input_choice == 1:
        # shows the growth of leaves as it plots not only one graph but many under different condition (diff i
        # values)
        for i in range(2, no_of_times_rule_be_applied+1):
            x = 0
            y = 0
            angle = 90
            saved_x_cords = []
            saved_y_cords = []
            angle1 = []
            saved_angle1 = []
            cords_x = [0]
            cords_y = [0]

            sym_tree = AxiomTransformer(input_axiom, rules, i, angle_of_tree)
            # stores the processed axiom
```

```
            command = str(sym_tree.iteration())
            ang = sym_tree.get_angle()
            first_tree = Tree('Banyan Tree', command, ang, x, y, angle, cords_x, cords_y,
saved_x_cords,
                              saved_y_cords, angle1, saved_angle1)
            first_tree.coordinates_generator()
            first_tree.return_x_cords()
            first_tree.return_y_cords()
            # uses inheritance so that the Tree class can access the function of PlotCords
which is to plot
            # graph
            first_tree.plot_graph()


        # plots the final graph (does not show the process of growth)
        if input_choice == 2:
            sym_tree = AxiomTransformer(input_axiom, rules, no_of_times_rule_be_applied,
angle_of_tree)
            command = str(sym_tree.iteration())
            ang = sym_tree.get_angle()


            first_tree = Tree('Banyan Tree', command, ang, x, y, angle, cords_x, cords_y,
saved_x_cords,
                              saved_y_cords, angle1, saved_angle1)
            first_tree.coordinates_generator()
            first_tree.return_x_cords()
            first_tree.return_y_cords()
            first_tree.plot_graph()


    elif input_fractal_type == 2:
        print("No.1 Input the real and imaginary parts of c")
        print("No.2 View Sample 1")
        print("No.3 View Sample 2")
```

```
    input_choice = int(input("Please input the number"))

    screen_width = 480

    screen_height = 320

    scale = 300

    # creates a list of x coordinates. The form of the list is changed (it has 1 row and 480
column consisting

    # of elements / x-coordinates) due to function reshape()

    x_cords = np.linspace(-screen_width / scale, screen_width / scale,
num=screen_width).reshape((1,

                                                    screen_width))


    # creates a list of y coordinates. The shape of the list is changed (it has 320 row and 1
column consisting

    # of elements)

    y_cords = np.linspace(-screen_height / scale, screen_height / scale,
num=screen_height).reshape(

        (screen_height, 1))

    # generates a 2D array in which each element is in complex number form. It
represents the complex plane. It

    # contains all the possible coordinates within the range specified earlier (see x_cords
and y_cords)

    # np.tile multiplies the elements inside the array. np.tile(x_cords, (screen_height, 1))
generates 320

    # duplicates of x_cords list while 1j * np.tile(y_cords, (1, screen_width)) creates
screen_width number of

    # duplicates inside each row in y_cords. (multiplied by 1j so we can distinguish
between imaginary and real

    # values). Adding these 2 matrices together results in 2D array which represents
complex plane.

    z = np.tile(x_cords, (screen_height, 1)) + 1j * np.tile(y_cords, (1, screen_width))

    # creates 2D array consisting of True values

    m = np.full((screen_height, screen_width), True, dtype=bool)

    # generates 2D array consisting of zero values

    n = np.zeros((screen_height, screen_width))
```

```
        if input_choice == 1:
            # allows user to decide real and imaginary value of c unlike in sample, where the values are already
            # fixed
            c_real = float(input("Please input real part of c"))
            c_imaginary = float(input("Please input imaginary part of c"))
            colormap_type = str(input("Please input colour type"))
            # creates a 2D array storing the same complex numbers
            c = np.full((screen_height, screen_width), complex(c_real, c_imaginary))
            general_julia = JuliaGenerator(screen_width, screen_height, scale, x_cords, y_cords, z, c, m, n,
                                           colormap_type)
            general_julia.julia_calculator()
            general_julia.plot_julia()

        # view sample 1
        elif input_choice == 2:
            c = np.full((screen_height, screen_width), complex(-0.4, 0.6))
            general_julia = JuliaGenerator(screen_width, screen_height, scale, x_cords, y_cords, z, c, m, n,
                                           "gist_earth")
            general_julia.julia_calculator()
            general_julia.plot_julia()

        # view sample 2
        elif input_choice == 3:
            c = np.full((screen_height, screen_width), complex(-0.624, 0.435))
            general_julia = JuliaGenerator(screen_width, screen_height, scale, x_cords, y_cords, z, c, m, n,
                                           "summer")
            general_julia.julia_calculator()
```

```
            general_julia.plot_julia()


        elif input_fractal_type == 3:
            # in mandelbrot set z always starts with 0
            z = complex(0, 0)
            # initial value to be passed into class, will store the coordinates of the complex plane later
            cx = 0
            cy = 0
            c = complex(cx, cy)
            input_choice = int(input("No.1 Input zoom values manually\nNo.2 View sample 1\nNo.3 View sample 2"))
            if input_choice == 1:
                # allows user to input zoom values and color map
                x1 = float(input("Please input x1 value"))
                x2 = float(input("Please input x2 value"))
                y1 = float(input("Please input y1 value"))
                y2 = float(input("Please input y2 value"))
                color_map = str(input("Please input the color type"))
                x_axis = np.linspace(x1, x2, 1000)
                y_axis = np.linspace(y1, y2, 1000)
                x_axis_len = len(x_axis)
                y_axis_len = len(y_axis)
                atlas = np.empty((x_axis_len, y_axis_len))
                general_mandelbrot = MandelbrotGenerator(120, 1000, x_axis, y_axis, x_axis_len, y_axis_len, atlas, z, cx
                                                         , cy, c, color_map)
                # values to be passed to and organized by mandelbrot function which in turn calls mandelbrot_calculator
                # - function so that the z values are tested
                general_mandelbrot.mandelbrot()
                # plots and save the mandelbrot image
```

```
            general_mandelbrot.return_atlas()


        # View sample 1
        elif input_choice == 2:
            x_axis = np.linspace(-2.55, 0.75, 1000)
            y_axis = np.linspace(-1.5, 1.5, 1000)
            x_axis_len = len(x_axis)
            y_axis_len = len(y_axis)
            atlas = np.empty((x_axis_len, y_axis_len))


            general_mandelbrot = MandelbrotGenerator(120, 1000, x_axis, y_axis, x_axis_len,
y_axis_len, atlas, z, cx
                                                     , cy, c, "hot")
            general_mandelbrot.mandelbrot()
            general_mandelbrot.return_atlas()


        # View Sample 2
        elif input_choice == 3:
            x_axis = np.linspace(-0.22, -0.21, 1000)
            y_axis = np.linspace(-0.70, -0.69, 1000)
            x_axis_len = len(x_axis)
            y_axis_len = len(y_axis)
            atlas = np.empty((x_axis_len, y_axis_len))
            general_mandelbrot = MandelbrotGenerator(120, 1000, x_axis, y_axis, x_axis_len,
y_axis_len, atlas, z, cx
                                                     , cy, c, "hot")
            general_mandelbrot.mandelbrot()
            general_mandelbrot.return_atlas()


    elif input_fractal_type == 4:
            break
```

```
main()


class AxiomTransformer():
    def __init__(self, sequence, transformations, iterations, angle2):
        self.sequence = sequence
        self.transformations = transformations
        self.iterations = iterations
        self.angle2 = angle2

    def iteration(self):
        for j in range(self.iterations):
            # method get(c, c) gets the value of the key of a dictionary(transformations)
            # the function of ".join is to join strings without any separator
            self.sequence = ".join(self.transformations.get(c, c) for c in self.sequence)
        return self.sequence

    def get_angle(self):
        return self.angle2

# ref:[https://bitaesthetics.com/posts/fractal-generation-with-l-systems.html]


import matplotlib.pyplot as plt


# Plot the x and y coordinates
class PlotCords:
```

```
    def __init__(self, x_cords, y_cords):
        self.x_cords = x_cords
        self.y_cords = y_cords

    def plot_graph(self):
        plt.xlabel("x coordinates")
        plt.ylabel("y_coordinates")
        plt.margins(x=0.1, y=0.1)
        plt.plot(self.x_cords, self.y_cords)
        plt.show()
```

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

class JuliaGenerator:

```
    # kindly see the function main() to see the values which are passed to these variables.
    def __init__(self, screen_width, screen_height, scale, x_cords, y_cords, Z, C, M, N,
colormap_type):
        self.screen_width = screen_width
        self.screen_height = screen_height
        self.scale = scale
        self.x_cords = x_cords
        self.y_cords = y_cords
        self.Z = Z
        self.C = C
        self.M = M
        self.N = N
        self.colormap_type = colormap_type
```

```
    def julia_calculator(self):
        # note that Z, C, M, N are 2D arrays.
        for i in range(256):
            # We want to process the whole coordinates at once, hence we used 2D arrays. Z contains all the coordinates
            # on the complex plane. (the coordinates are already converted into complex form).
            # self.Z[self.M] means boolean indexing. Values of Z for which M contains true are selected.
            self.Z[self.M] = self.Z[self.M] * self.Z[self.M] + self.C[self.M]
            # for the magnitude of the values of array Z which exceed 2, M will store false. So that next time, the
            # values which are processed are the "True" values (values of Z which do not exceed 2)
            self.M[abs(self.Z) > 2] = False
            # to color the Julia fractals
            self.N[self.M] = i


    # plot the Julia fractals, remove the axes and save the figure.
    def plot_julia(self):
        fig = plt.figure()
        fig.set_size_inches(self.screen_width / 100, self.screen_height / 100)
        ax = fig.add_axes([0, 0, 1, 1], frameon=False, aspect=1)
        ax.set_xticks([])
        ax.set_yticks([])
        plt.imshow(np.flipud(self.N), cmap=self.colormap_type)
        plt.savefig('julia-plt.png')
        plt.close()



import matplotlib.pyplot as plt
```

class MandelbrotGenerator:

```
    def __init__(self, max_iter, density,  x_axis, y_axis, x_axis_len, y_axis_len, atlas, z, cx, cy, c, color_map):
        self.max_iter = max_iter
        self.density = density
        self.x_axis = x_axis
        self.y_axis = y_axis
        self.x_axis_len = x_axis_len
        self.y_axis_len = y_axis_len
        self.atlas = atlas
        self.z = z
        self.cx = cx
        self.cy = cy
        self.c = c
        self.color_map = color_map


    def mandelbrot(self):
        # iterates through each and every coordinates
        for ix in range(self.x_axis_len):
            for iy in range(self.y_axis_len):
                self.cx = self.x_axis[ix]
                self.cy = self.y_axis[iy]
                self.c = complex(self.cx, self.cy)
                # test the c values and iteration stored in atlas
                self.atlas[ix, iy] = self.mandelbrot_calculator(self.c, self.max_iter)
                pass
            pass


    # In mandelbrot set the z value starts with 0
    def mandelbrot_calculator(self, c, max_iter):
        # z = complex(0, 0) is the same as 0 + 0j
```

```
        self.z = complex(0, 0)
        for iteration in range(max_iter):
            self.z = (self.z * self.z) + self.c
            # checks if the magnitude of z i greater than 4
            if abs(self.z) > 4:
                break
                pass
            pass
        # if magnitude of z greater than 4, return i value
        return iteration

    # generates mandelbrot image
    def return_atlas(self):
        plt.imshow(self.atlas.T, cmap=self.color_map, interpolation="nearest")
        plt.savefig('mandelbrot-plt.png')
        plt.close()


# reference:[Make Your Own Mandelbrot by Tariq Rashid and
# https://github.com/danyaal/mandelbrot/blob/master/mandelbrot.py]
```