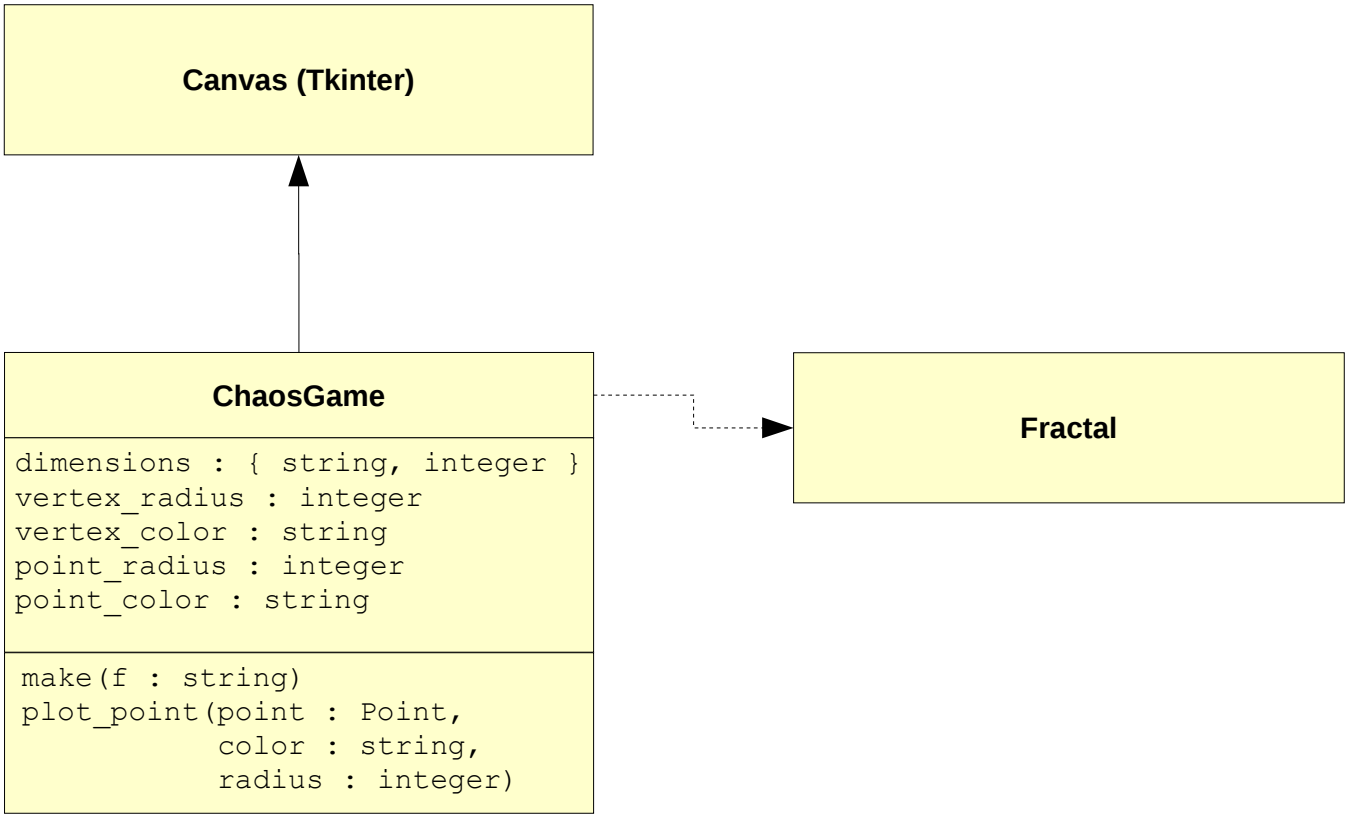
Program: The Chaos Game...Reloaded

Your task in this programming assignment is to re-implement the chaos game from a previous program, while also augmenting it. You must implement the variation that results in the Sierpinski triangle (as before). In addition, you must implement at least two of the other fractals that are described in the Python application demonstrated in a previous lesson. Recall the Sierpinski Carpet, the Pentagon, the Hexagon, the Octagon, etc. Of the many fractals demonstrated, the aforementioned are the easiest to implement as they do not involve rotations about a vertex when computing a new point to plot. You must also utilize inheritance (specifically for the fractals themselves). More details on this later.

The big picture
To provide a frame of reference with respect to the program's design, let's take a look at the overall class diagram:



Note how the ChaosGame class inherits from Tkinter's Canvas class. That is, the ChaosGame **is a** Tkinter Canvas. This is similar to the last chaos game program. Your program should iteratively generate the various fractals that you wish to implement. That is, it should play the chaos game for the Sierpinski Triangle, then play the chaos game for another fractal, and so on. Each iteration of the chaos game should create a new window (with a new canvas) and produce a fractal. Therefore, the ChaosGame class **has a** fractal that it **makes** by **plotting points** on the canvas.

Here's an example of how to layout the main part of your program to accomplish this:

```
# the default size of the canvas is 600x520
WIDTH = 600
HEIGHT = 520

# the implemented fractals
FRACTALS = [ "SierpinskiTriangle", "SierpinskiCarpet",\
             "Pentagon", "Hexagon", "Octagon" ]

# create the fractals in individual (sequential) windows
for f in FRACTALS:
     window = Tk()
     window.geometry("{}x{}".format(WIDTH, HEIGHT))
     window.title("The Chaos Game...Reloaded")
     # create the game as a Tkinter canvas inside the window
     s = ChaosGame(window)
     # make the current fractal
     s.make(f)
     # wait for the window to close
     window.mainloop()
```

Of course, the example illustrates all of the aforementioned fractals (you only need to implement three in total). As before, you should maintain various canvas dimensions and constraints in order to more easily calculate vertices and points. They were defined in the previous chaos game program as follows:

- `WIDTH`: the width of the window (600 by default);
- `HEIGHT`: the height of the window (520 by default);
- `MIN_X`: the minimum *x*-coordinate value for a visible point (5 by default);
- `MAX_X`: the maximum *x*-coordinate value for a visible point (`WIDTH` - 5 by default);
- `MIN_Y`: the minimum *y*-coordinate value for a visible point (5 by default);
- `MAX_Y`: the maximum *y*-coordinate value for a visible point (`HEIGHT` - 5 by default);
- `MID_X` = (`MIN_X` + `MAX_X`) / 2; and
- `MID_Y` = (`MIN_Y` + `MAX_Y`) / 2.

Instead of being globals throughout the entire program, these will now stored in a dictionary (more details below). The ChaosGame class has the following state:

(1) **dimensions**: a dictionary of strings to integers that represents the various canvas constraints and their values (e.g., `dimensions["min_x"] = 5`);
(2) **vertex_radius**: the number of pixels of plotted vertices (2 pixels by default);
(3) **vertex_color**: the color of a vertex (red by default);
(4) **point_radius**: the number of pixels of plotted points (0 pixels by default – i.e., points are a single pixel); and
(5) **point_color**: the color of a point (black by default).

The ChaosGame has the following behavior (through functions):

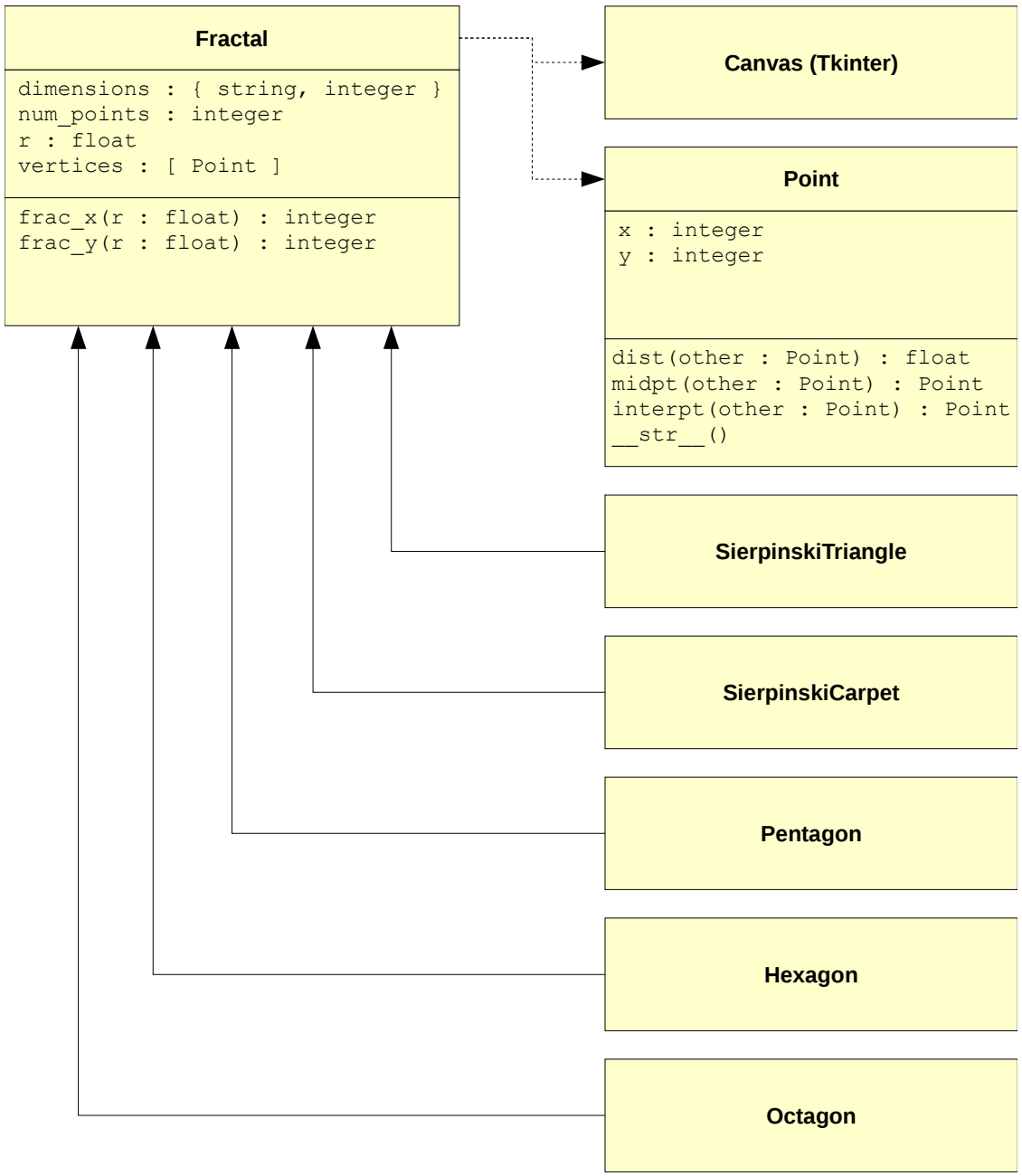
(1) **make**: a function that takes a string that represents the fractal to create (e.g., "SierpinskiTriangle") and plots it on the canvas; and

(2) **plot_point**: a function that takes an instance of the Point class, a string representing the point's color, and an integer representing the point's radius; and plots the point on the canvas.

Note that the **make** function should determine which fractal to make, create an instance of the appropriate fractal class, and play the chaos game to create the fractal on the canvas.

The fractals

Since this version of the chaos game generates at least three different fractals, let's utilize inheritance to implement them. This makes sense since, for example, the Sierpinski Triangle is a fractal. So is the Sierpinski Carpet. Implement the following class diagram in a separate module (that you will import in your main program):



The Fractal class is a superclass (i.e., it is inherited by other subclasses). It has the following state:

(1) **dimensions**: a dictionary representing the canvas dimensions/constraints (which is used to calculate the placement of the vertices and the generated points on the canvas);
(2) **num_points**: the number of points to plot for this fractal (50,000 by default);
(3) **r**: the distance ratio to calculate new points with for this fractal (0.5 by default); and
(4) **vertices**: a list of Point instances that identifies the vertices for this fractal.

When instantiating a fractal, the dictionary of canvas dimensions/constraints that are defined in the ChaosGame class is passed in. This is useful because the dimensions/constraints makes calculating vertices and points much easier and allows the fractals to be properly created on any window size. Specific values for **num_points** and **r** for the various fractals is provided below. Here's an example of what the Fractal's constructor could look like:

```
def __init__(self, dimensions):
    # the canvas dimensions
    self.dimensions = dimensions
    # the default number of points to plot is 50,000
    self.num_points = 50000
    # the default distance ratio is 0.5 (halfway)
    self.r = 0.5
```

The Fractal class has the following behavior:

(1) **frac_x**: a function that takes a single float that represents the distance ratio as input and returns the *x*-coordinate on the canvas that corresponds to this distance ratio; and
(2) **frac_y**: a function that takes a single float that represents the distance ratio as input and returns the *y*-coordinate on the canvas that corresponds to this distance ratio.

The two functions **frac_x** and **frac_y** are used to calculate a fractal's vertices on the canvas. For example, the Sierpinski Triangle's top vertex is located at the top-center of the canvas. Therefore, it's *x*-component is halfway between MIN_X and MAX_X – a distance ratio of 0.5. The **frac_x** function can be implemented as follows (note that this may change based on your specific implementation of the program):

```
def frac_x(self, r):
    return int((self.dimensions["max_x"] - \
            self.dimensions["min_x"]) * r) + \
            self.dimensions["min_x"]
```

The function first calculates the distance between the min and max values, scales them by the distance ratio, and finally adds back the min value – in order to obtain the *x*-coordinate with the specified distance ratio within the min and max range. The function **frac_y** is similarly implemented.

The Point class is essentially the same as in the previous programs that you have submitted. There is a slight addition, however. Since some of the fractals have distance ratios that vary from the default 0.5 (a point that lies exactly halfway – a midpoint), there must be a method that can generate a new point that is *any* distance ratio from the last generated point to a randomly chosen vertex. The **frac_x** and **frac_y** functions listed above implement a similar process (only with respect to the canvas dimensions/constraints). In the Point class, let's call this new method **interpt** (for intermediate point) and define it as follows:

```
def interpt(self, other, r):
```

```
        # make sure that the distance ratio is expressed from a
        #  smaller component value to a larger one
        # first, the x-component
        rx = r
        if (self.x > other.x):
             rx = 1.0 - r
        # next, the x-component
        ry = r
        if (self.y > other.y):
             ry = 1.0 - r

        # calculate the new point's coordinates
        # the difference in the components (distance between the
        #  points) is first scaled by the specified distance ratio
        # the minimum of the components is then added back in order
        #  to obtain the coordinates in between the two points (and
        #  not with respect to the origin)
        x = abs(self.x - other.x) * rx + min(self.x, other.x)
        y = abs(self.y - other.y) * ry + min(self.y, other.y)

        return Point(x, y)
```

The various fractals inherit from the Fractal class and define a fractal's vertices. As mentioned above, they may additionally redefine **num_points** and **r**. Note that it is assumed that the various dimensions and constraints defined by and/or on the canvas have been defined in the ChaosGame class. The individual fractal subclasses are defined below. Note that all vertex calculations with respect to canvas dimensions/constraints have already been calculated for you! Also, the dimensions/constraints should be obtained from the dictionary in the Fractal class (i.e., dimensions). For example, any reference to MIN_X in the vertices below can be obtained via a statement as follows:

```
self.dimensions["min_x"]
```

The fractals

The **SierpinskiTriangle** is plotted with the default **50,000** points. The points are generated using the default distance ratio of **0.5**. It has the following **three** vertices:

- (MID_X, MIN_Y)
- (MIN_X, MAX_Y)
- (MAX_X, MAX_Y)

The **SierpinskiCarpet** is plotted with **100,000** points. The points are generated using a distance ratio of **0.66**. It has the following **eight** vertices:

- (MIN_X, MIN_Y)
- (MID_X, MIN_Y)
- (MAX_X, MIN_Y)
- (MIN_X, MID_Y)
- (MAX_X, MID_Y)
- (MIN_X, MAX_Y)
- (MID_X, MAX_Y)

- (MAX_X, MAX_Y)

The **Pentagon** is plotted with the default **50,000** points. The points are generated using a distance ratio of **0.618**. It has the following **five** vertices:

- (MID_X + MID_X * cos(2 * pi / 5 + 60), (**frac_y**(0.5375) + MID_Y * sin(2 * pi / 5 + 60))
- (MID_X + MID_X * cos(4 * pi / 5 + 60), (**frac_y**(0.5375) + MID_Y * sin(4 * pi / 5 + 60))
- (MID_X + MID_X * cos(6 * pi / 5 + 60), (**frac_y**(0.5375) + MID_Y * sin(6 * pi / 5 + 60))
- (MID_X + MID_X * cos(8 * pi / 5 + 60), (**frac_y**(0.5375) + MID_Y * sin(8 * pi / 5 + 60))
- (MID_X + MID_X * cos(10 * pi / 5 + 60), (**frac_y**(0.5375) + MID_Y * sin(10 * pi / 5 + 60))

The **Hexagon** is plotted with the default **50,000** points. The points are generated using a distance ratio of **0.665**. It has the following **six** vertices:

- (MID_X, MIN_Y)
- (MIN_X, **frac_y**(0.25))
- (MAX_X, **frac_y**(0.25))
- (MIN_X, **frac_y**(0.75))
- (MAX_X, **frac_y**(0.75))
- (MID_X, MAX_Y)

The **Octagon** is plotted with **75,000** points. The points are generated using a distance ratio of **0.705**. It has the following **eight** vertices:

- (**frac_x**(0.2925), MIN_Y)
- (**frac_x**(0.7075), MIN_Y)
- (MINX, **frac_y**(0.2925))
- (MAX_X, **frac_y**(0.2925))
- (MIN_X, **frac_y**(0.7075))
- (MAX_X, **frac_y**(0.7075))
- (**frac_x**(0.2925), MAX_Y)
- (**frac_x**(0.7075), MAX_Y)

You should now have everything that you need to get started on this program! It is ambitious, to be sure; however, break it down into multiple individual steps. Implement it incrementally, a little at a time. Implement a little, test a little, implement a little more, test a little more, and so on.

To implement the chaos game, the ChaosGame class has the following **additional** requirements:

- Vertices should be plotted in a color of your choice and have a radius greater than 0 (i.e., be larger than the generated points that make up the fractal);
- Points making up the fractal should be formed from the last plotted point and a randomly chosen vertex (i.e., through the `interpt` function of your 2D point class);
- You are free to plot the first point using any two vertices;
- Points making up the fractal should be plotted in a color that is different than the vertices;
- You are free to set the canvas background to a color of your choice (however, ensure that the vertices and fractal are clearly visible); and
- You are free to define other functions in the ChaosGame class as appropriate.

As always, keep the following things in mind (as you will be graded on them):

(1) You must include meaningful headers, use good coding style, use meaningful variable names, and comment your source code where appropriate; and

(2) You must submit your source code as a **single ZIP** file that contains the **two** individual .py files: (1) the main program with the ChaosGame class; and (2) the module that contains the Point class, the Fractal superclass, and the various fractal subclasses.

Your prof should demonstrate a working program so that you can see the end result.