2D Points

Overview

The purpose of this programming assignment is to gain experience writing an object-oriented program that utilizes a GUI. You will do this by creating a 2D (two-dimensional) `Point` class, instantiating randomly generated points within a specified coordinate range, and plotting those points on a canvas. See the sample output shown below for more detail.

Specification

This program primarily consists of two parts:

- A `Point` class that defines a blueprint for a 2D point; and
- A `CoordinateSystem` class that manages a Tkinter `Canvas` and plots `Point` objects in random colors onto the canvas.

The `Point` class

When designing your `Point` class, make sure to address the following:

- A 2D point is made up of an X-component and a Y-component. Each component is a **floating point** value;
- A constructor should initialize a point either with specified values for the X- and Y-components or the point (0.0, 0.0) by default;
- Instance variables should be appropriately named (i.e., meaningful variable names that begin with underscores);
- Accessors and mutators should “wrap” and provide read access of and write access to the instance variables;
- A function named `dist()` should take **two** points as input and calculate and return the **floating point distance** between the two points;
- A function named `midpt()` should take **two** points as input and calculate and return the **midpoint** of the two points (i.e., a `Point` instance of the midpoint); and
- The magic `__str__()` function should provide a string representation of a point in the format `(x,y)`.

To test your *Point* class, feel free to use the following code as the main part of a test program:

```
# create some points
p1 = Point()
p2 = Point(3, 0)
p3 = Point(3, 4)
# display them
print("p1:", p1)
print("p2:", p2)
print("p3:", p3)
# calculate and display some distances
print("distance from p1 to p2:", p1.dist(p2))
print("distance from p2 to p3:", p2.dist(p3))
print("distance from p1 to p3:", p1.dist(p3))
# calculate and display some midpoints
print("midpt of p1 and p2:", p1.midpt(p2))
print("midpt of p2 and p3:", p2.midpt(p3))
print("midpt of p1 and p3:", p1.midpt(p3))
# just a few more things...
p4 = p1.midpt(p3)
print("p4:", p4)
print("distance from p4 to p1:", p4.dist(p1))
```

Make sure that your output matches the following to confirm that your *Point* class works properly:

```
p1: (0.0,0.0)
p2: (3.0,0.0)
p3: (3.0,4.0)
distance from p1 to p2: 3.0
distance from p2 to p3: 4.0
distance from p1 to p3: 5.0
midpt of p1 and p2: (1.5,0.0)
midpt of p2 and p3: (3.0,2.0)
midpt of p1 and p3: (1.5,2.0)
p4: (1.5,2.0)
distance from p4 to p1: 2.5
```

The *CoordinateSystem* class

This class (which will inherit from Tkinter's *Canvas* class) will initialize a canvas on which the points will be plotted and include two functions: one to instantiate a number of randomly generated *Point* instances, and another to plot the generated points. Specifically:

- The *CoordinateSystem* class must inherit from Tkinter's *Canvas* class and fill the entire Tkinter window (except for the title bar, of course);
- A function named *plot()* should take **an instance** of a *Point* and a **color** as input – and plot the specified point on the canvas in the specified color;
- A function named *plotPoints()* should take the **number of points** to plot as input (set to **5,000** by default), generate the points (each should be randomly generated and of a randomly chosen color), and call the *plot()* function to plot each point;
- Plotted points should be individual instances of your *Point* class, each with random X- and Y-components that are within the width and height of the canvas (set to **800x800** by default);
- Plotted points should have a radius of 0 (i.e., a point is made up of a single pixel);
- Points should be plotted in a random color from the following set of colors: black, red, green, blue, cyan, yellow, and magenta;
- The *CoordinateSystem* class must include class variables for point radius and point colors – that are referred to as appropriate within the class; and
- You are free to define other functions in the *CoordinateSystem* class as appropriate.

Constraints and notes

Note the following constraints and notes (see the rubric below for more detail):

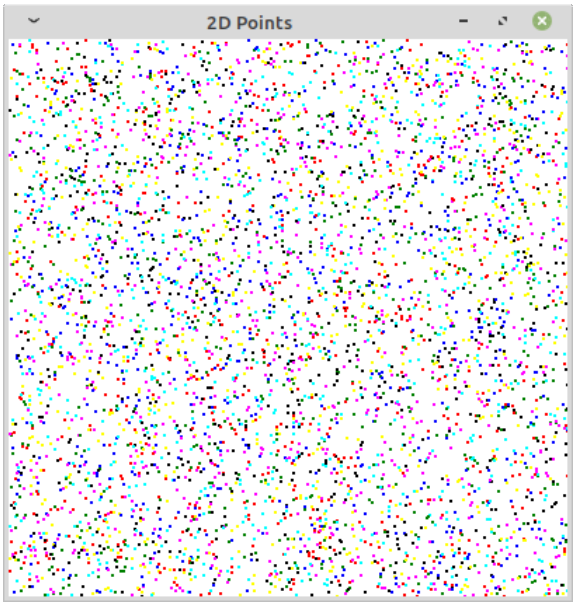
- Your output must be similar to the output of the sample run shown below. Since the points are randomly generated in random colors, your plotted points will likely be different.
- You must comment your source code appropriately, include an informative header at the top of your program, and use good coding style.
- You must use meaningful variable names.
- You must adhere to the requirements of the *Point* and *CoordinateSystem* classes outlined above.

Deliverable

Submit a Python 3 source file that can be executed through an IDE (such as Thonny) or via the command line (e.g., `python3 2DPoints.py`).

Sample output

Here is sample output of a run of a **correct** solution:



Note that the points are randomly generated and will therefore look different for every run.

Hints

Here are some hints to help get you started:

- Of course, feel free to Google things to help you. As always, do not Google solutions and make sure to **cite sources**.
- In the constructor of the `Point` class, convert the X- and Y-components of a point to floating point values.
- Recall the formula to calculate the distance between two points, (x_1, y_1) and (x_2, y_2): $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$.

- Recall the formula to calculate the midpoint of two points, (x_1, y_1) and (x_2, y_2): $\left(\frac{x_1+x_2}{2}, \frac{y_1+y_2}{2}\right)$.
- Consider declaring a Python list of all supported point colors (e.g., “black”, “red”, etc) and using the *`randint()`* or *`choice()`* function from Python’s **random** library to randomly select a color for a specific point.

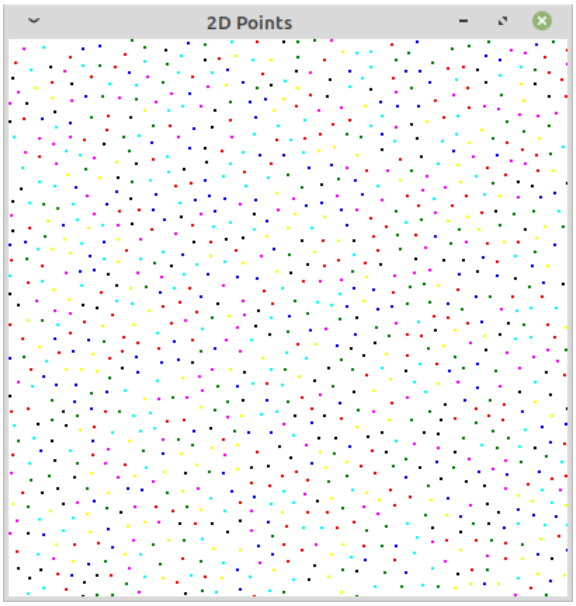
Rubric

Please note the following rubric for this programming assignment:

2D Points		
#	**Item**	**Points**
1	Good coding style	1
2	Appropriate comments	1
3	Appropriate header	1
4	Meaningful variable names	1
5	Output is correct	1
6	Point class: constructor	2
7	Point class: instance variables	2
8	Point class: getters and setters	2
9	Point class: dist() function	4
10	Point class: midpt() function	4
11	Point class: __str__() function	2
12	CoordinateSystem class: inherits from Tkinter Canvas	2
13	CoordinateSystem class: plot() function	5
14	CoordinateSystem class: plotPoints() function	5
15	CoordinateSystem class: class variables	3
16	CoordinateSystem class: points are plotted in random colors	2
17	CoordinateSystem class: points are potted as a single pixel	2
TOTAL		**40**

An observation

Did you notice in the sample output above that there are small clusters of points in addition to gaps where no points exist (i.e., the points are not evenly distributed across the canvas)? Generally, plotting random points does not guarantee an even distribution of the points on the canvas. We can, however, implement what is known as a **Poisson disc** method that encourages points to “spread out” a bit. This is accomplished by setting a **threshold** for the **minimum** distance between points. If a randomly generated point is not at least the threshold distance from every other point, it is discarded (and a new point is randomly generated). Here is sample output with the Poisson disc method implemented:



Of course, this can be computationally intensive – especially when trying to fill the canvas with the maximum possible number of points. That is, once a good number of points have been plotted, it will be unlikely that a randomly generated point is far enough away from every other point on the canvas. It will possibly take many tries to find a valid point. If too many points are plotted, it may be impossible to find a valid point. This motivates several strategies to attempt to address this:

(1) Pick an appropriate number of points based both on the size of the canvas and the threshold; and

(2) Set a limit on how many times to regenerate a point if one is not far enough away from every other point (i.e., after some number of tries, give up and go to the next point).

An appropriate number of points to generate can be calculated as follows:

$$n = \frac{WIDTH * HEIGHT}{THRESHOLD^2}$$

For an 800x800 canvas and a threshold of 10 pixels, for example, 6,400 points can be plotted. Note, however, that this would take a very long time. Reducing the number of points is recommended.

Similarly, a reasonable threshold can be calculated as follows:

$$t = \sqrt{\frac{WIDTH * HEIGHT}{n}}$$

For an 800x800 canvas on which 1,250 points are to be plotted, for example, a threshold of approximately 22 pixels should work. Again, this is likely to take a long time, and picking a smaller threshold is recommended.

Implementing a limit on how many times a point is regenerated if it is found to be too close to every other point is fairly simple. Simply set up a counter that increments each time a point is found to be invalid. Once a valid point is found, reset the counter and go to the next point. If the counter reaches the limit, skip the current point, reset the counter, and go on to the next point.

If you wish to implement the Poisson disc method, please ensure that the number of points plotted and the threshold are selected to ensure that the execution time of your program does not exceed **five seconds** on a typical system. **Please note that this enhancement is not required (i.e., you do not need to implement the Poisson disc method)!**