CS 1: Introduction To Computer Programming, Fall 2013

Assignment 5: A Whopping 2000 Calculations Per Second!!!

Due: *Thursday, November 21, 02:00:00*

Coverage

This assignment covers the material up to lecture 15.

Part What to hand in

You will be handing in three files for this assignment.

You should collect the answers to sections B and C into a file called `lab5_bc.py`.

For section D, you should hand in the files `lab5_d_1.py` and `lab5_d_2.py` containing your programs.

There is a test script for section B only. It's called lab5_b_tests.py. You should download it and run it to make sure your code from section B works properly. Code from section C will not be tested.

All three files of your code should be handed in to csman as usual.

Part A: Installing new Python packages

There are no new Python packages that you need to install for this week's assignment.

Part B: Exercises

For this section, you will be writing a few simple (and fairly short) classes. Make sure that your classes have a class docstring, and that all the methods in each class have docstrings.

1. **[15]** Write a class called `Point` which represents a point in three-dimensional Euclidean space (with real-valued coordinates). This class will contain only two methods. The constructor method (`__init__`) will take as inputs the `x`, `y` and `z` coordinates of the point and will store them in the object. In addition, a method called `distanceTo` will take another `Point` as its input and will compute the distance between that point and the point being acted on, using the Euclidean distance formula (the distance between two points is the square root of the sum of the squares of the differences between the coordinates).

   Examples:

   ```
   >>> p1 = Point(1.2, 2.4, 0.1)
   >>> p2 = Point(0.0, -3.2, -1.3)
   >>> p1.distanceTo(p2)
   5.89576118919347
   ```

The distance is `sqrt((1.2 - 0.0)**2 + (2.4 - (-3.2))**2 + (0.1 - (-1.3))**2)`, which evaluates to the printed value.

2. **[15]** Write a class called `Triangle`. Instances of this class will contain three `Point`s (as defined in the previous problem), which should be supplied as arguments to the constructor (don't just supply the (x, y) coordinates; give actual `Point` objects as arguments). The class will contain one method besides its constructor: `area`. This method will compute the area of a triangle using Heron's formula:

```
a = length of side 1
b = length of side 2
c = length of side 3
s = (a + b + c) / 2
area = sqrt(s * (s - a) * (s - b) * (s - c))
```

Examples:

```
>>> p1 = Point(0.0, 0.0, 0.0)
>>> p2 = Point(1.0, 0.0, 0.5)
>>> p3 = Point(0.0, 1.0, 0.5)
>>> t = Triangle(p1, p2, p3)
>>> t.area()
0.6123724356957944
```

3. **[30]** In this problem you will be designing a (slightly) more complicated class called `Averager`. The purpose of this class is to store a list of numbers and perform various operations on it. Specifically, there will be methods to do the following actions (method names in square brackets):

    1. [`getNums`] Return a copy of the list of numbers stored so far.
    2. [`append`] Append a new number to the list.
    3. [`extend`] Append a list of numbers to the existing list.
    4. [`average`] Compute the average of the stored list. If the list is empty, return 0.
    5. [`limits`] Compute the minimum and maximum of the stored list. If the list is empty, return the tuple `(0, 0)`.

    Make sure that the `getNums` method returns a *copy* of the stored list, not the stored list itself (the test script will check for this). (*Hint:* list slices may be useful here.) Methods should not return fields of an object which are mutable, because that permits the object's internal state to be altered by any external code, which destroys one of the big advantages of using objects (secure encapsulation of state).

    Examples:

```
>>> a = Averager()
>>> a.average()
0
>>> a.getNums()
[]
>>> a.append(1)
>>> a.append(2)
>>> a.append(3)
>>> a.getNums()
[1, 2, 3]
>>> a.extend(range(4, 11))
>>> a.getNums()
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> a.average()
5.5
>>> a.limits()
(1, 10)
```

Part C: Pitfalls: Poor design

Each of the following snippets of code has one or more design flaws. Design flaws include (but are not limited to):

- unnecessary code
- excessively complex code
- unnecessarily inefficient code (doing more computation than is necessary to get the result)

These will be the only design flaws in the functions in this section. For each case,

a. explain what the problem or problems are
b. write out what the code should be

NOTE: You are *not* looking for style errors here. All of these functions are written in a clean coding style.

1. **[20]** This function has a minor but very common design flaw.

```
def is_positive(x):
    '''Returns True if x is positive.'''
    if x > 0:
        return True
    else:
        return False
```

2. **[20]** *N.B.* don't use the Python `index` method in your version of this function.

```
def find(x, lst):
    '''Returns the index into a list where x is found, or -1 otherwise.
    Assume that x is found at most once in the list.'''
    found = False
    location = -1
    for i, item in enumerate(lst):
        if item == x:
            found = True
            location = i
    if found == True:
        return location
    else:
        return -1
```

3. **[20]**

```
def categorize(x):
    '''Return a string categorizing the number 'x', which should be
    an integer.'''
    if x < 0:
        category = 'negative'
    if x == 0:
        category = 'zero'
    if x > 0 and x < 10:
        category = 'small'
    if x >= 10:
        category = 'large'
    return category
```

4. **[20]** *N.B.* don't use the Python `sum` function in your version of this function.

```
def sum_list(lst):
    '''Returns the sum of the elements of a list of numbers.'''

    if len(lst) == 0:
        answer = 0
    elif len(lst) == 1:
        answer = lst[0]
    elif len(lst) == 2:
        answer = lst[0] + lst[1]
```

```
    elif len(lst) > 2:
        total = 0
        for item in lst:
            total += item
        answer = total
    return answer
```

Part D: Miniproject: Interactive picture drawing

For this week's miniproject, we will continue making drawings using the Python graphics package `Tkinter`. This week, we will focus on event handling, which is the process by which the user can directly effect a graphical program while it's running by pressing a key on the keyboard or clicking the mouse (or other possibilities that we won't consider here).

You will write two programs for this section, each in their own files (called `lab5_d_1.py` and `lab5_d_2.py` for problems D.1 and D.2 respectively). To keep you from having to type in a lot of standard code (what is called in the programming world "boilerplate" code), we are providing you with a file called lab5_d_template.py. You should use this as a starting point for your programs and fill in the places marked:

```
<your code goes here>
```

with your own code (removing that line, of course, and renaming the file).

1. **[60]** This program will be an incredibly simple painting program. You will create an 800x800 window which is initially blank. It will respond to the following events with the corresponding actions.

    1. When the user presses the `q` key, the program will exit (`q` stands for "quit").
    2. When the user clicks the left mouse button, a filled circle of the current color and a diameter between 10 to 50 pixels (randomly-chosen) will be drawn centered at the current position of the mouse cursor.
    3. When the user presses the `c` key, the current color will change to a new randomly-chosen color (`c` stands for "change color"). That will become the color that is used to paint circles until the `c` is pressed again. Previously-drawn circles don't change color.
    4. When the user presses the `x` key, all the circles are cleared from the screen. This means that every time a circle is created, the handle that is returned must be stored somewhere.

    The current color is initially set to a random color value.

    Here's a picture generated by this program:

Pretty cool, huh? ;-) OK, maybe not.

Here are some things to keep in mind when writing this program:

- All of your event handling code (aside from the `bind` method calls) should be in two functions; one of them will handle key presses and the other will handle left mouse button clicks.
- The key value from the event object is `event.keysym`. Confusingly, this doesn't use the same notation as the `bind` command. So the `q` key is represented just by the letter `'q'`, not by `'<q>'`. Some keys, such as the `+` and `-` keys, are represented by names (`'plus'` and `'minus'` respectively). `Tkinter` is not very consistent!
- You will need global variables in this program representing the root window, the canvas, the current color and the list of circles (actually circle handles). You will need to use the Python `global` statement, but don't overuse it! Only use it when it's absolutely necessary. Remember

that you only need to use the `global` statement when you are going to *change* the value of a global variable from inside a function.

- When you delete the circles from the canvas, make sure you also empty out the global list of circle handles.
- Use a helper function to do the actual drawing of circles. You can also write whatever other helper functions seem useful. Don't try to cram all of your code into the event handling functions! That's very poor design.
- Don't define any classes for this problem (or any of the problems in this section). We'll cover defining classes for use in conjunction with `Tkinter` next assignment. Here we just want to get you working with `Tkinter` and event handling.

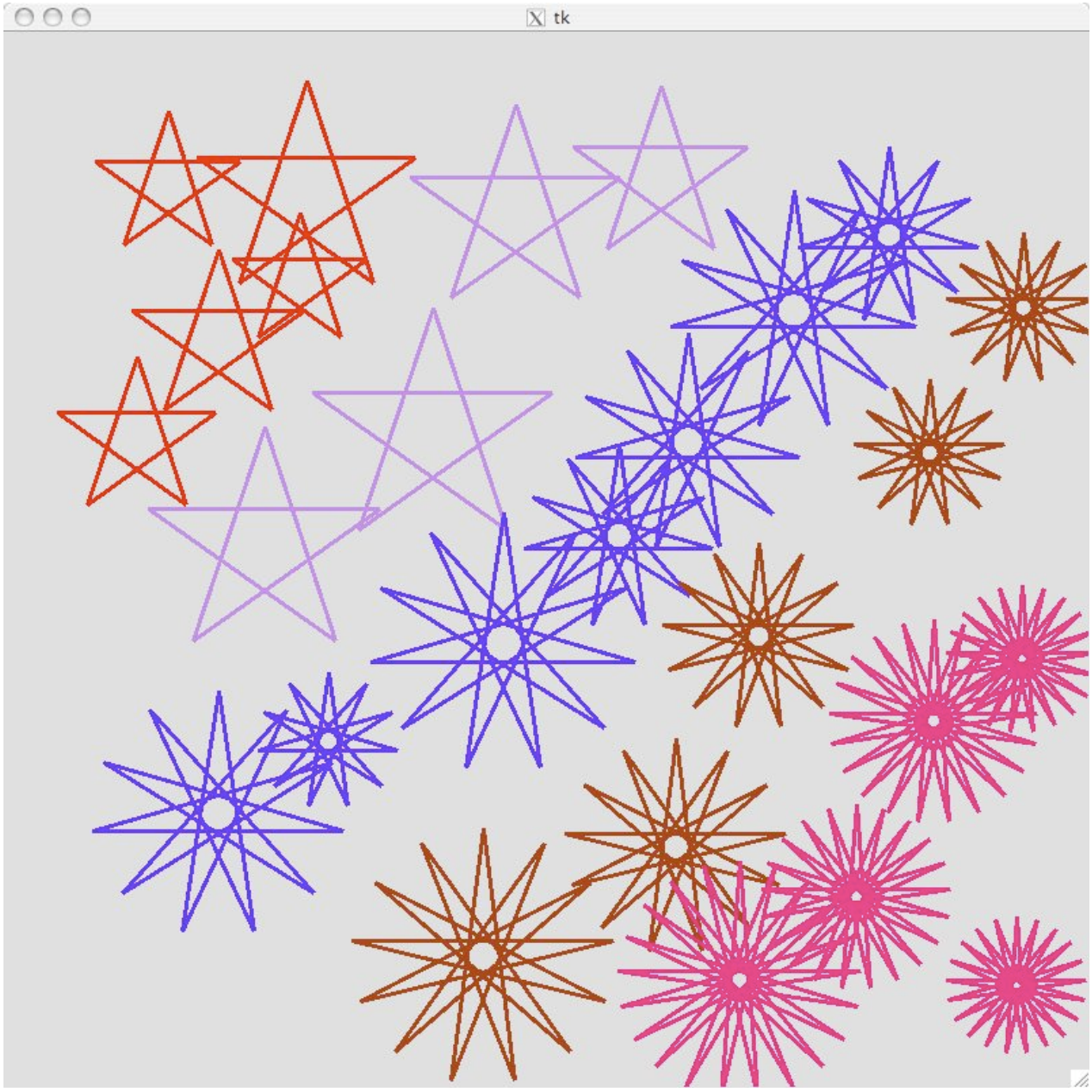
This program does not have to be very long; our solution is about 60 lines long including comments and docstrings.

2. **[120]** For this program, you will be creating a variant of the previous program. Here is what will change:

   1. When the left mouse button is clicked, instead of a colored circle, a colored N-pointed star shape is drawn. N is a positive odd integer no smaller than 5 (so N could be 5, 7, 9, ...). The star shape is made of N individual lines of the same length *i.e.* it's not a filled star. The size of a star is the radius of the circle circumscribing the star; this size for any particular star is randomly chosen to be between 50 and 100 pixels.
   2. When the `+` key is pressed, N is increased by 2. So if clicking the left mouse button previously caused a five-pointed star to be drawn, after clicking `+` once, clicking the left mouse button would then cause a seven-pointed star to be drawn. Previously-drawn stars don't change.
   3. When the `-` key is pressed, N is decreased by 2. N is not allowed to go below 5, however. If N is 5 and the `-` key is pressed, N stays at 5 (no error message is printed either).

   Everything else will stay the same. In particular, the `c` key will again change the current color, the `x` key will again clear the display and empty out the global list of handles, and the `q` key will again quit the program. Once again, the current color is initially set to a random color value.

   Here's a picture generated by this program:



You are encouraged to re-use as much code as possible from the previous program.

NOTE: This program is harder than the ones you have been writing up to this point. Here are some tips to help you get this program working correctly.

1. You will need to keep track of these global variables: the root window, the canvas, the current color, the value of `N`, and a list of the handles of the lines drawn on the canvas (so you can delete them when the `x` key is pressed).

2. As mentioned above, the event keysyms for the `+` and `-` keys are `'plus'` and `'minus'`, not `'+'` and `'-'` as you might expect.

3. By far the most difficult part of this program is the function that draws individual stars. (You'd better put this code into a separate function — if it's inside *e.g.* the event handler you'll have to re-write the program because of poor design.) The way to handle this task is to

break it into pieces (known in programming as "divide and conquer"). Here are the separate steps we recommend you follow.

1. Write a separate function called `draw_line` that does nothing but draws individual lines on a canvas with a particular starting location, ending location, and color, and will return the handle of the line just drawn. This function should be easy to write.

2. Use the `draw_line` function to write a function called `draw_star` which draws an N-pointed star centered at a particular location, using lines of a particular length and color. The center point of the star points vertically upward (towards the top of the canvas). The function will return a list of the handles of all the lines drawn.

   This function may be a bit tricky. The way to approach it is to remember that the points of a star will be located on the perimeter of a circle around the center point, equally spaced around the circle, with one of the points located at the top of the circle. (The radius of this circle is the "size" of the star.) So the first task is to figure out the locations of these points. Once you have these locations, you simply need to connect them up with lines. To do this, you connect a point with one of the points located as far as possible around the circle. If the coordinates of the points are in a list with nearby points adjacent in the list, then for N points you need to go (N - 1) / 2 points in the list to get the point to connect a particular point to (going around the end of the list to the beginning if necessary). So for N = 5 you need to go 2 points along the list, for N = 7 you need to go 3 points along the list, etc. Thus, if you have a list of 5 points, the lines you need to draw are between points 0 and 2, 1 and 3, 2 and 4, 3 and 0, and 4 and 1. For 7 points you'll need to connect points 0 and 3, 1 and 4, 2 and 5, 3 and 6, 4 and 0, 5 and 1, and 6 and 2. For larger values of N the process is similar.

   This is all well and good, but how do you compute the coordinates of the points in the first place? To do this, it's really helpful to know about polar coordinates. The points of a star are spaced around a circle at equal distances, which means that lines connecting them to the center of the circle (which are not the lines that you will draw) form equal angles with lines from nearby star points to the center. The angle (in radians) is (2 * pi / N) for N points. Normal (x, y) coordinates can be derived from polar coordinates using these equations:

   ```
   x = r * cos(theta)
   y = r * sin(theta)
   ```

   You know what `r` is (it's the radius of the circle in which the star fits). And `theta` is the angle between the x-axis and the line connecting the (x, y) coordinates of a point to the origin of the circle.

   Even with this knowledge, there are still a couple of problems. First is that the (x, y) positions have to be moved so that they are centered on the cursor position, not on the point (0, 0). This is easy. Second is that the y-axis on the screen goes in the opposite direction to the y-axis we use when doing math. And third is that the first point we want is at (0, -r) relative to the origin of the circle (the top of the circle), not at (r, 0). *Hint:* One clever approach involves pretending that the first point is at (r, 0), computing all the other points, and then flipping the x and y axes.

   Our version of the `draw_star` function is 21 lines of code (not including docstrings and comments).

3. Once the `draw_star` function is working, the rest of the program should be easy.

Because of all this, expect to spend a fair bit of time debugging your `draw_star` drawing function, and don't be shy about asking for help from your TA! When you get it working, though, it'll be a lot of fun to play with.

Again, this program is not that long. Our solution is about 110 lines long, including comments and docstrings.
