CS 1110, LAB 10: SUBCLASSES AND ENCAPSULATION

http://www.cs.cornell.edu/courses/cs1110/2016fa/labs/lab10/

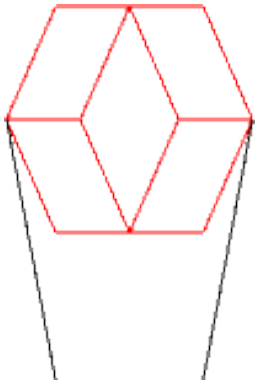
First Name: _____ **Last Name**: _____ **NetID**: _____

This lab demonstrates the power of subclasses, particularly in GUI applications. Subclasses are a great way to customize visual behavior of a GUI object. In addition to basic subclasses, you this lab will also give you some experience with encapsulation: the process of protecting attributes with getters and setters.

We know that you have a lot to do this week. The exam is coming up, and Assignment 5 and Assignment 6 have been pushed out simulatenously (though Assignment 6 is not due until well after the exam). We have kept this lab very short. In our experience, you will find you spend more time reading the lab than doing it. Most people are able to complete this lab during the lab section.

Lab Materials. We have created several Python files for this lab. You can download all of the from the Labs section of the course web page.

http://www.cs.cornell.edu/courses/cs1110/2016fa/labs



When you are done, you should have the following three files.

- `drawapp.py` (the primary script to run the application)
- `shapes.py` (a support module, which you will not touch)
- `lab10.py` (the module that you must modify)

You should create a *new* directory on your hard drive and download all of the files into that directory. Alternatively, you can get all of the files bundled in a single ZIP file called `lab10.zip` from the Labs section of the course web page. Open a command prompt and type

```
python drawapp.py
```

A figure like that on the right should appear in the window. Your command prompt will fill with the usual messages from Kivy. In addition, you should see some test ouput about the various shapes that are drawn in the window.

Getting Credit for the Lab. Once again, you have a choice between getting credit through the online system or your instructor. The online lab is available at the web page

http://www.cs.cornell.edu/courses/cs1110/2016fa/labs/lab10/

By now you should have a good idea of which version you are more comfortable with. This lab is fairly robust with respect to the online system. However, you should be modifying the file `lab09.py` directly, and not typing all of your answers into the online system directly. Therefore, there is not really that much difference between which one you chose to do.

Course authors: D. Gries, L. Lee, S. Marschner, W. White

As with the previous lab, if you do not finish during the lab, you have **until the beginning of lab next week to finish it**. You should always do your best to finish during lab hours. Remember that labs are graded on effort, not correctness.

1. Understanding the Application

Before you get started on this lab, we should first describe a bit how `drawapp` works. If you open it up you will notice that it contains two classes. The first is `DrawApp`; it is a subclass of the Kivy class `App` and has a single important method: `run()`. This `run()` method is called in the script code of this module, and is what opens up a new Window.

By itself, `DrawApp` just draws a blank window. The second class, `Panel` represents the contents of the Window, and it is responsible for drawing the figure above. When the application is built, a `Panel` object is "placed inside" the Window. The initializer for `Panel` calls the function `draw_shapes()`, which is located in module `lab10`. This function draws the figure you see.

The function `draw_shapes()` creates several objects, all of which are subclasses of class `Shape`. The `Shape` class, which is defined in the module `shapes.py`, contains information about the position of the shape, as well as its size and geometry. When we call the `Panel` method `draw()`, the shape is added to the panel and remains there until you quit the application.

The values of `x` and `y`, as well as the side lengths, are given in pixels. Pixels in this application are indexed by integer coordinates *starting at the bottom left corner*, as follows:

```
...
(0,2)  (1,2)  (2,2)  ...
(0,1)  (1,1)  (2,1)  ...
(0,0)  (1,0)  (2,0)  ...
```

In a pixel coordinate (x,y), x is the horizontal coordinate; it increases from left to right. Similarly, y is the vertical coordinate; it increases from bottom to top.

When working with graphics, each shape is going to have its own drawing code. That is why we never actually make objects of class `Shape`. Instead, we create subclasses of `Shape` for specific shapes, and work with those instead. In this lab, we are working with the following classes:

Class	Subclass Of	Description
`Line`	`Shape`	Line segment between two points.
`Parallelogram`	`Shape`	Four sided shape where each pair of opposing sides is parallel.
`Rhombus`	`Parallelogram`	Parallelogram where all four sides are equal length.
`Rectangle`	`Parallelogram`	Parallelogram where all angles are right angles.

You should now look at the documentation of `Parallelogram` (which is in `shapes.py`) to see how it works. You can either look at the source code directly, or import the module and use the `help()` function as shown in class. You should pay particular attention to the concept of the "leaning factor" in the definition of a parallelogram.

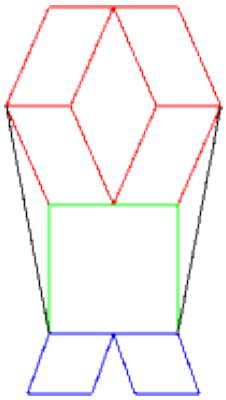
You are welcome to look at the other the classes in `shapes.py`; particularly the base class `Shape`. However, you will notice that it contains a lot of Kivy code. We are not going to explain this code, and you do not need to understand it to do the lab.

2. Lab Instructions

This lab is broken up into four tasks. The first three are rather short, though you might want to review the presentation slides for Lecture 20.

Task 1: Complete the Initializers.

The image in the window is incomplete. Only `Parallelogram` and `Line` draw properly. The initializers in `Rhombus` and `Rectangle` are unfinished. If you open `lab10.py`, you will see that their `__init__` methods are empty.



This is unacceptable. In order to initialize properly, each of these subclasses must call the `__init__` method in their parent class as a helper function. Fortunately,the class `Parallelogram` does all the work for drawing, so that is the only thing we need to do.

We showed how to do this in class. Call the method `__init__` as an explicit function in the parent class. If you are unsure of how to do this, open up `shapes.py` and look at the classes `Line` and `Parallelogram`.

Use this technique to complete the initializers of `Rhombus` and `Rectangle`. For `Rhombus`, this is a single line. However, `Rectangle` has an extra attribute named `_fill`. Assign this to `colormodel.WHITE`. As this is a default value, and not defined by a parameter, there is no precondition to enforce.

When you are done, run `drawapp.py` again. You should now get the image on the right.

Task 2: Add Getters and Setters to `Rhombus` and `Rectangle`.

The class `Rectangle` has a single (mutable) attribute in addition to those in `Parallelogram`: `_fill`. You should create a single getter and setter for this attribute (called `setFill` and `getFill`). The setter should enforce the invariant that `_fill` is an RGB object. You should use `isinstance` to check the type, rather than the `type` function, as shown in class.

The class `Rhombus` does not have any additional attributes beyond those in `Parallelogram`. Normally this means we would not add any setters or getters. Instead, we just inherit the setters and getters from `Parallelogram`. But we are going to add them, because we want to make a attribute that was previously not mutable (the side length) now mutable.

However, `Rhombus` has an additional invariant on top of those of `Parallogram`. The two attributes `_l1` and `_l2` must be equal at all times. We cannot allow a user to change just one of these, as that would break the invariant. To protect the invariant, we are going to create a getter and setter for a new "virtual" attribute `side`. **You should not create an attribute called `_side`.** Instead, this getter and setter will "group" the existing attributes `_l1` and `_l2` together.

The getter `getSide` should return either attribute `_l1` or attribute `_l2`. It does not matter.

The setter `setSide` should have a second parameter after self, called `value`. This setter

- Asserts that `value` satisfies the class invariant for `_l1` and `_l2`
- Assigns `value` to **both** attributes `_l1` and `_l2`

Implement this getter and setter. If you cannot remember the syntax, look at the slides in Lecture 19 covering getters and setters.

Task 3: Add Some Methods to `Rectangle`.

When you added the `getSide()` method to `Rhombus` you might have noticed that it changed what was being printed out when you ran `drawapp.py`. That is because `getSide()` was used in the `__str__` method (previously this method was crashing and returning only the empty string).

The class `Rectangle` has a similar problem. Right now, nothing is printed out at all for the `Rectangle` objects. That is because the `__str__` method is empty. You need to implement it.

In creating a string for a square object, we want you to give the `x` and `y` position, the side-length, and the area. For example, when you complete the implementation of `__str__`, and run `drawapp.py`, you should see the following:

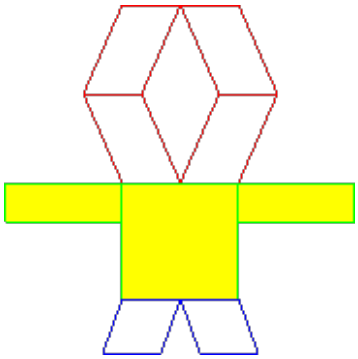
```
rectangle at (95, 69), dimension 60x60, area 3600
```

To see how to do this, look at the implementation of `__str__` in `Rhombus`. Your solution will be very similar (but without the `try-except`). Notice that this method invokes the `__str__` method in `Shape` to get the `x` and `y` position, as that is the class that first defines these attributes.

Your implementation of `__str__` requires that you implement the method `getArea` that has been provided. You should implement this as well.

Task 4: Add Some Arms.

The shape being drawn almost looks like a person. It has a head (the `Parallelograms`), a body (the `Rectangle`), and some legs (the `Rhombuses`). You will give the person arms and fill in the rectangles to make it look like it is wearing a sweater. When you are done, the image will look like the one to the right.



All the changes you will make will be in the function `draw_shapes()`. This function takes a `Panel` as an argument and adds multiple shapes to it. Read through this function now. Once you have read the function, you should add an extra line before drawing object `s5`. In this extra line, you should use the setter `setFill` to make the center square filled with yellow.

Next, look for where the shape color is set to black. This is where the two black lines are drawn. Comment this out, as you will be replacing this code with the arms. Each arm is should be a green rectangle with yellow filling. It should be also be 60 pixels long and 20 pixels high. The arms should be attached at the top right and top left of the square that makes up the body. The tops of the arms should be parallel to the top line of the square.

In writing the code that draws rectangles, use the variables that are defined at the top of function `draw_shapes()`. Use variables to contain all the constants that you need, as we did in functions `draw_shapes()`. You should avoid using numbers directly in the constructor `Rectangle`.

Hint: To determine the arm coordinates, look at the position of the green square. For debugging purposes, you may want to include print statements for the objects you create, as we did.

When you get the shape to the right, you are finished.