Python Practice 8: Object Oriented Programming
— *Animating shapes*

The aim of this lab is to provide an opportunity to practice using an *object oriented* approach to programming in Python. We'll do this through a task that involves animating simple shapes. The task gives us the chance to practice using both **classes** and **objects**, and also **inheritance**.

Warning: The code used in this lab **does not work** with the `IPython Console` in `Spyder`. **Instead**, you must get Spyder to run the code using an **external system terminal**, as follows. The first time that you run any code file, press **ctrl-F6** to open a menu window that allows you to configure how the code is run. In this window, select "`Execute in an external system terminal`" and (lower down) "`Interact with the Python console after execution`", and then press `Run`. Thereafter, you can run the file, in the same way, just by pressing **F5**.

Please note: After running code, you will typically have open both the *terminal window*, and a *graphics window*. You can close *both* windows, by clicking on × in the top-right corner of the *terminal window*. DO NOT allow multiple terminal windows to remain open (or your PC may grind to a halt), i.e. be sure to close extra terminal windows once you have finished with them.

Start by downloading the lab files:

`MovingShapes.py`, `Shapes.py`, `test_interacting_shapes.py`, `test_multi_shapes.py`, `load_shapes.py`, `test_one_shape.py`.

To begin, open the code file `load_shapes.py`, and inspect its contents, which are as follows:

```
from Shapes import *
frame = Frame()
s1 = Shape('square',100)
```

The code in `load_shapes.py` starts by importing code from the file `Shapes.py`, which provides some very basic graphics capability. (You don't need to understand how the code in `Shapes.py` works – just how to call it.) The next line of code in `load_shapes.py` creates an *instance* of the `Frame` class, and causes a graphics window to open, containing an outer 'frame' line, marked with min/max x and y coordinate values. The second command creates an *instance* of the `Shape` class, and assigns it to the variable `s1`, also causing a square figure to appear in the window. The figure appears with its *central point* positioned at coordinate position (0,0). The figure's shape is set by the first argument (`'square'`, with `'diamond'` or `'circle'` being other possible values). The second argument sets the figure's diameter. Run the code (as indicated above), to see all this happen. Then close the terminal (and thereby also the graphics window), to tidy up.

Now extend the code in `load_shapes.py`, by adding the following command:

```
s1.goto(200,100)
```

The command calls the figure's `goto` method, causing it to move to the specified (x, y) coordinates. Re-run the code file, to see its effect. That's really all we need to know about the `Shapes` module. Again, tidy up by closing the terminal window.

The above is sufficient for simple animation, i.e. by making a figure `goto` successive new locations. For example, we can animate our sample square by adding the following code (try this out):

```
for n in range(100):
    s1.goto(n * 8, n * 5)
```

Part 1: Creating our first moving figure

Our task is to define classes for *shapes that move*. A start has been made in `MovingShapes.py`. Study this code to make sure that you understand it. The `MovingShape` class has an incomplete definition. Its initialisation method currently only creates variables to store the figure object created (an instance of the `Shape` class), plus associated details, and its position, as recorded by the instance variables `self.x` and `self.y`. Instances should also store information about their rate of movement, so that their position can be *iteratively updated* – but do not do so as yet. There is a method `goto_curr_xy`, which calls the figure's `goto` method, to move the figure to its current x, y position. There is also a `moveTick` method, which should update a `MovingShape`'s position with each tick of the clock. This currently has an empty definition (here "`pass`" is a command that performs no action).

The file also contains incomplete definitions of classes `Square`, `Circle` and `Diamond`, which *inherit* from `MovingShape`. Each has an `__init__` method that invokes the superclass `__init__` method, giving its `shape` parameter appropriately, as `'square'`, etc. The code in the file `test_one_shape.py` tests the code in `MovingShapes.py`, by creating an instance of `Square`, and attempting to move it, using a loop to repeatedly call its `moveTick` method. This currently has no effect, as the method has an *empty definition*. Observe that the `while` loop's condition is just `True`, allowing it to repeat indefinitely. Hence, the process will terminate only when the terminal window is manually closed.

Begin by adding *instance variables* (i.e. ones with names of the form `self.`α) to the `__init__` method of the `MovingShape` class, to store the *rate of movement* in x and y directions. Call them `self.dx` (for '*delta-x*') and `self.dy`, and try out different start values (reasonable values being $\approx 10 \pm 5$).

Finally, to achieve animation, complete the definition of `moveTick`. This should first update the x,y position variables by adding the corresponding position-change values (e.g. adding `self.dx` to `self.x`, etc). It should then call the `goto_curr_xy` method, so that the figure moves to the new location. If everything is correct, then running the test file (`test_one_shape.py`) should open a graphics window, in which there is a square that travels across the graphics window.

Part 2: Adding random variation — velocities

Study the code in the second test file `test_multi_shapes.py`, which attempts to animate several shapes at the same time. Observe that it creates several instances of the `Square` class — how many is determined by the variable `numshapes`. These instances are stored in a list (`shapes`). When the second '*animation*' loop of the code runs, there is an embedded loop that calls the `moveTick` method of each `Shape` object. The results, however, are not very interesting, as the shapes all start at the same position and have the same speed/direction, and so travel in unison.

`MovingShapes.py` imports the function `random` from `pylab`, renaming it as `r`. Hence, a call `r()` returns a random value between 0 and 1, with which we introduce variation to movement vectors. Thus, as 10 has been a reasonable value for `self.dx` so far, we might now assign a value such as "`5 + 10 * r()`", which will be a random value between 5 and 15. Implement this idea for both the x and y velocity components. Run the test file several times to see the variation introduced.

Although the shapes should no longer travel together, we've introduced only limited variation, as all values are +ve. The test "`r() < 0.5`" returns `True` 50% of the time, so it can be used in a conditional to randomly to flip individual velocity values from +ve to −ve. Again, try this out.

Part 3: Adding random variation — start positions

Figures should ideally start at random locations, and preferably *within* the line frame of the graphics window. Note that when we create a `MovingShape` instance, `frame` is provided as a parameter. This object has attributes `frame.width` and `frame.height` (i.e. the frame's dimensions), which can be used when computing a suitable random start location for the figure.

Consider first the x component of a shape's (x, y) position. To avoid the figure overlapping the border, there is a *minimum* x value that should ever be assigned, which is determined by its diameter d. To take the case of a `Square`, this value should not be less than $d/2$. Likewise, the *maximum* value for x should not be more than `frame.width` *minus* $d/2$. Similar reasoning produces *min* and *max* values for y. Compute these values and assign them to instance variables (`self.minx`, `self.maxx`, etc), as they will be useful again later on.

When assigning a random position to a shape, a suitable x value is one that lies randomly between `self.minx` and `self.maxx`, which might be computed as immediately below. Implement this idea to assign random positions to objects, and test that it works as expected.

```
self.minx + r() * (self.maxx - self.minx)
```

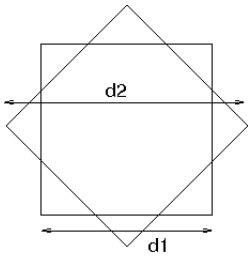
Part 4: Hitting the wall

We can now look at adding some more interesting behaviour to figures. We treat the border in the window as a wall against which figures bounce. The 'physics' here is simple. Consider a square which is moving upwards to the right, which hits the right hand wall. On bouncing, it would continue to move upwards, but now away from the wall. Assuming no energy is lost, we can achieve this result simply by reversing the x velocity component (i.e. multiplying it by -1). The same is true for a figure bouncing off the left wall. For a figure hitting the top or bottom wall, we need only reverse the y velocity component.

The natural place to include code for this behaviour is in the definition of the `moveTick` method. Thus, after we've updated the position of the figure, we can check if the new position is one that counts as 'hitting the wall'. For the x component, this is true if the new x position (say `self.x`) is *less than the minimum* x position, or is *more than the maximum* x position, which are values we computed in the previous section and stored as variables `self.minx`, `self.maxx`. Implement this idea by adding conditionals to the `moveTick` method, and test that your code behaves correctly.

Part 5: Diamonds vs. Squares

It's easy to see that the behaviour we've created for squares is not correct for diamonds. This is because the tilted orientation of a diamond makes its width and height *in practice* more than that of a square with the same specified diameter, i.e. so that in the figure on the right, the *practical* diameter `d2` is such that $\texttt{d2} = \sqrt{2} \cdot \texttt{d1}$. Hence, our calculation of min/max x, y values is incorrect for diamonds, with the consequence that the 'initial positions' allowed for diamonds include ones where their corners overlap the boundary, and this is also true of the point at which diamonds 'bounce'.



It seems we need different figures to have different behaviour — here in the calculation of min/max x/y values. We could do this in various ways, but since the calculation done previously for this works for both squares and circles, let's keep this as the *default* in the `MovingShape` class, and get the `Diamond` sub-class to override it. To do this, take the segment of code that calculates the min/max values in the `__init__` method of the `MovingShape` class, and put it into a separate method. This new method should then instead called within the `__init__` method, in place of

the original code segment. If you then add an alternative definition of this method to the `Diamond` class, which does the different calculation required for diamonds, this will serve to *override* the superclass definition for the case of diamonds. For squares and circles, the superclass definition will be used.

Part 6: Chatty shapes

Let's add some further behaviour that differs between shapes. When any shape bounces (i.e. hits the wall), have it print a statement (which shows in the interpreter window) such as:

```
I'm a bouncing diamond - my area is 6400 sq.units
```

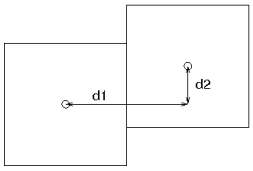
To this end, define a method (`report_bounce`, perhaps) in the superclass, that is called whenever a bounce is made. This method will call upon another method to calculate the shape's area (perhaps called `my_area`), which will need to be *separately defined* in each of the subclasses, to give an area calculation appropriate to that class.

Part 7: Interacting shapes

The file `test_interacting_shapes.py` shows how we might implement some *interaction* behaviours between shapes. The extra code is placed within the 'animation' loop of the test file, and with each 'tick' of the animation clock, it does the following: for each possible pair of shapes (i.e. shape instances in `shapes`), the `check_interaction` method of the first is called, with the second supplied as argument. As an instance method, this method can of course access/modify its own instance's attributes. However, since the second shape is supplied to it as an argument, it can also access/modify the latter's attributes. As such, it can calculate if the two shapes are appropriately placed to interact, e.g. if they are close enough to collide. And, it can implement the effect of their interaction, e.g. by modifying their velocities.

Colliding squares: A relatively simple case to consider is allowing squares to collide and bounce off each other. To implement this, we might have a `check_interaction` method for the `MovingShape` class with *null* behaviour (i.e. it does nothing). For the `Square` class, we can override this with a new definition that does the following: (i) check that the second shape is also a square, (ii) check that the two shapes are close enough to collide and (iii) decide if the collision is *side-to-side* or *top-to-bottom*, then (iv) implement an appropriate velocity change. If we treat this as an *elastic collision*, the appropriate effect for the *side-to-side* case is that the two square *swap over* their `dx` values (or their `dy` values, for a *top-to-bottom* collision). (See the wikipedia page for "Elastic collision", especially the section on 1-dimensional collisions.)

The figure to the right shows an example case of two squares colliding *side-to-side*. We can see that a collision situation arises where the distance between their positions, in **both** x and y directions, is *less than or equal to* their diameters. In this case, for a *side-to-side* collision, the *smaller* distance is in the vertical direction. For a *top-to-bottom* collision, the smaller distance would be in the horizontal direction.



If you fancy a challenge . . . Shapes go to War

For a more challenging task, consider implementing a version of the classic *Scissors–Paper–Stone* game, with diamonds as scissors, square as paper, and circle as stone. Shapes that 'lose' in any pair-wise collision might just disappear. (Note that instances of the `Shape` class have a method `vanish` that causes the figure to become invisible.) The game would be over when all the shapes that remain belong to just one of the subclasses (e.g. if they were all squares, or all circles).