Review cs1graphics

Things to know:

- All our graphics programs should start with: from cs1graphics import *, so that names of objects and functions defined in this library can be recognized by Python, when we use them in our programs.
- As next step, we need to create a Canvas object which creates the window where we’ll draw. See slide 1-22 in the Big Ideas lecture.
- cs1graphics defines many graphics shapes that we can use in our program by calling the respective special functions that create them, such as Rectangle, Circle, Polygon, etc.
- The created shapes (known as objects) are referred through the names we assign them with the assignment statements. Using these names and the . (dot) operator, we call special functions (known as methods) to manipulate the state of the shape (position, color, size, borders, etc.). For example: torso.setFillColor("gray").
- The coordinate system for drawing shapes has its origin (0, 0) in the top-left corner of the canvas window.
- By default, every new shape you create without a reference point (or center point), will be assigned the reference (0, 0).
- To make objects appear on the canvas window, we need to add them through the method add.

Motivating Problem

Sometimes graphics scenes contain repeated elements. But, a programmer should follow the DRY principle: Don't Repeat Yourself. This makes our code easy to read and maintain.

The shown fishtank has six scaled, rotated, and flipped versions of the fish pattern in the picture. How to make use of it?

In preparation to learning to build scenes like this, we will discuss:
1. Cloning (making a copy of a shape)
2. Transformation operation:
   1. Scaling (increase or decrease the size by a factor)
   2. Rotating (moving around a center with a certain angle)
   3. Flipping (turning over, usually across one of three axes)
### Cloning

We can make a copy of a drawable object using the `clone` method.

```python
wedge = Polygon(Point(0,100), Point(0,0), Point(200,100))
wedge.setFillColor('green')
wedge.moveTo(100,200) # the grid below shows position after this statement
wedge2 = wedge.clone()
wedge2.moveTo(525,200)
```

**Concepts in this slide:**
For a Polygon, the reference point is the first provided point (see yellow circle).

### Scaling

We can change the size of a drawable object using the `scale` method.

This method takes one number argument. A number > 1 increases the size, a number < 1 decreases the size.

```python
wedge = Polygon(Point(0,100), Point(0,0), Point(200,100))
wedge.setFillColor('green')
wedge.moveTo(100,200)
wedge2 = wedge.clone()
wedge2.scale(0.5)
```

### Rotating

We can rotate a drawable object using the `rotate` method. This causes the shape to move around its center by a certain angle, which is given as argument to the method.

```python
wedge = Polygon(Point(0,100), Point(0,0), Point(200,100))
wedge.setFillColor('green')
wedge.moveTo(100,200)
wedge2 = wedge.clone()
wedge2.rotate(180)
```

**Notice negative value for angle.**

### Flipping

We can flip a drawable object around an axis using the `flip` method.

```python
wedge = Polygon(Point(0,100), Point(0,0), Point(200,100))
wedge.setFillColor('green')
wedge.moveTo(100,200)
wedge2 = wedge.clone()
wedge2.flip(90)
```

* On this and subsequent slides, to avoid clutter we omit invocations of the Canvas `add` method for each of the Polygon objects, e.g. `paper.add(wedge)` `paper.add(wedge2)` etc.
Returning to our motivating problem

Remember the problem from slide 4: How can we draw this fishtank with six similar fish?

We learned to transform individual shapes by cloning and then performing operations such as rotation, scaling, and flipping to them. However, a fish is composed of three parts: body, tail, and eye. These however, are currently not tied to one-another.

One approach we can try is to separately draw the body, tail, and eye for each of the six fish. Is this easy or hard? What about DRY?

A better way: Abstraction!

Layers capture patterns

Drawing the three parts (yellow body, green fin, and black eye) of every fish over and over again would be tedious, and getting the coordinates right for all the scalings, rotations, and flippings would be extremely challenging. We will make lots of mistakes.

Fortunately, there is a better way! We can abstract over the notion of a fish pattern by creating a Layer object, which you can think of as a mini-canvas for grouping together other shapes.

We can add items to a Layer just like we can add items to a canvas. The "push pin" reference point of a Layer is (0,0) in its own coordinate system.
A Layer containing one fish

# A layer is a "mini-canvas" for combining drawable objects
fish = Layer()

# yellow body of the fish
body = Ellipse(100,50,Point(0,0))
body.setFillColor('yellow')
fish.add(body)

# green tail of the fish
tail = Polygon(Point(-50,0),Point(-75,25),Point(-75,-25))
tail.setFillColor('green')
fish.add(tail)

# black eye of the fish
eye = Circle(5,Point(25,-5))
eye.setFillColor('black')
fish.add(eeye)

To notice:
The (0,0) point is in the center of Layer, not at the top left corner as for Canvas.

Cloning a Layer: a second fish

# Add fish to a canvas
tank = Canvas(600, 400, 'skyblue', 'Where is Dory?')
tank.add(fish)
fish.moveTo(100, 50)

# Add a second, bigger fish
fish2 = fish.clone()
tank.add(fish2)
fish2.moveTo(350, 100)
fish2.scale(2)

# Third fish
fish3 = fish.clone()
tank.add(fish3)

# Fourth fish
fish4 = fish.clone()
tank.add(fish4)

# Fifth fish
fish5 = fish.clone()
tank.add(fish5)

# Sixth fish
fish6 = fish.clone()
tank.add(fish6)

For fish 3 to 6 specify the coordinates and the transformation operations.

A tank with one fish

# Add fish to a canvas
tank = Canvas(600, 400, 'skyblue', 'Where is Dory?')
tank.add(fish)
fish.moveTo(100, 50)

After add
This is how the canvas looks like after adding the fish, why?

After moveTo
This is how the canvas looks like after moving the fish to a new point.

Populate the tank

# Add a second, bigger fish
fish2 = fish.clone()
tank.add(fish2)
fish2.moveTo(350, 100)
fish2.scale(2)

# Third fish
fish3 = fish.clone()
tank.add(fish3)

# Fourth fish
fish4 = fish.clone()
tank.add(fish4)

# Fifth fish
fish5 = fish.clone()
tank.add(fish5)

# Sixth fish
fish6 = fish.clone()
tank.add(fish6)

For fish 3 to 6 specify the coordinates and the transformation operations.
The power of abstraction

Suppose we want to add pink hats to all of our fish.

Do we want to change each fish individually?
No! We need only change the prototypical fish (the original fish prior to any cloning).

```python
# Add pink hat *before* any clones are made
hat = Polygon(Point(21,-21),Point(-10,-14),Point(-23,-36),Point(4,-30),Point(30,-46))
hat.setFillColor('pink')
fish.add(hat)
```

Now all our fishies have pink hats.

This illustrates the power of abstraction. When we abstract over a pattern, then a change to the pattern affects all instantiations of the pattern.

Drawbacks of Layers

Although Layers are powerful, they do not let us abstract over all the properties of our fish that we might want to change.

What if we want different fish to have different colors?

```python
# Add pink hat *before* any clones are made
hat = Polygon(Point(21,-21),Point(-10,-14),Point(-23,-36),Point(4,-30),Point(30,-46))
hat.setFillColor('pink')
fish.add(hat)
```

Now all our fishies have pink hats.

This illustrates the power of abstraction. When we abstract over a pattern, then a change to the pattern affects all instantiations of the pattern.

Digging Deeper

In this course we learn problem solving and happen to use Python to write our solutions.

Python has many details that initially seem daunting. Although you might not need to know them right now, you might need them eventually, so we want to make you aware of them. We'll put them in slides marked “Digging Deeper”:

```python
# Create canvas
paper = Canvas(400, 400, 'white', 'Test Problem')

# first rectangle
rec1 = Rectangle(100, 200, Point(150, 150))
rec1.setBorderWidth(5)

# create second by cloning
rec2 = rec1.clone()

# set different border colors
rec1.setBorderColor('red')
rec2.setBorderColor('blue')

# add to paper
paper.add(rec1)
paper.add(rec2)

# rotate first rectangle
rec1.rotate(45)

# adjust reference for 2nd rectangle
rec2.adjustReference(50, 100)
rec2.rotate(45)

# print new reference
print(rec2.getReferencePoint())
```

The method adjustReference

Initially, both rectangles have the same reference point (150, 150)

Here, the blue rect's reference was adjusted by (50, 100), becoming (200, 250). The rotation happened around this new reference point.
**Classes**

- A **class** is a description of the shared characteristics (state and behaviors) of a set of objects.
- A class is like a mold for making objects.
- Example of classes include:
  - `Canvas`
  - `Rectangle`
  - `Point`
  - `Polygon`
  - `Circle`
  - `Text`
  - `Image`
  - `Layer`

- An object is made from a class by calling the **constructor function** with the same name as the class. E.g.,
  - `Canvas(900, 600, 'cyan', 'sky')`
  - `Rectangle(75, 300)`
  - `Circle(50, Point(200, 100))`

- Each object made from a class is an **instance** of the class.

**Contracts**

- Every method/function has a **contract** or Application Program Interface (API) that specifies the behavior of the method/function.
- Every class has a contract/API for all of its methods.
- Any user of a method/function/class can expect that it will behave as described in the contract.
- Any implementer of the method/function/class must ensure that it fulfills the contract.
- Example of contracts include:
  - `cs1graphics API`
  - `Python built-in functions API`
  - `math module API`

**The Canvas Contract**

<table>
<thead>
<tr>
<th>Canvas</th>
</tr>
</thead>
<tbody>
<tr>
<td>Canvas(w, h, bgColor, title, autoRefresh)</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

This is just the names of the methods and their arguments in the Canvas class contract. The full Contract specifies what each method *does*.

**Inheritance**

Some classes are generalizations of another class

We arrange classes in an **inheritance hierarchy** in which more specific **subclasses** below inherit state and behavior from more general **superclasses** above.
Test your knowledge

1. Looking at the diagrams in Slide 3-3, how does the model for storing an object into a variable compare to that of other values we saw in the last lecture?
2. Can you give examples of state and behavior for cs1graphics objects?
3. What is the principle DRY and why is important to follow it?
4. Is the reference point always the center point of a shape? Explain by giving examples.
5. If we were to change the state of a cloned object, would the state of the original object change as well? Explain.
6. What is the difference between rotating and flipping?
7. Study the coordinates system in the slide 3-13. How does it differ from the one for the Canvas object that we have been working so far?
8. Were you able to figure out the needed transformations for the fishies in slide 3-15?
9. How can the two modules: graphicsState and cs1graphicsHelper assist you during programming?
10. Which class is more specific: Rectangle or Square? [digging deeper]