Creating a Class for Pygame

In a typical programming class, a great deal of time would be spent on creating traditional classes which show the features of object oriented design. Since this course is so short and the focus of the course is on creating games via Pygame, this lecture will be devoted to the use of the token class, which will be useful within Pygame.

The token class manages an item that can be displayed in Pygame. Since Python doesn't allow abstract classes (which is what the token class really should be), in this example, the a token object will be displayed as a circle with a given diameter and a given color (both provided when the constructor is called.)

The methods for this class will make it easy to manage a ball that is bouncing off each of the four walls of the display surface. Let's just dive in and look at the full listing of the class:

```
import random
import math
import time
import pygame, sys
from pygame.locals import *

# Token Class we'll use for drawing objects in pyGame
class token:

    # Default settings.
    x = 0
    y = 0
    dx = 0
    dy = 0
    side = 0
    color = pygame.Color(0,0,255)

    # Constructor.
    def __init__(self,myx,myy,mydx,mydy,myside,mycolor):
        self.x = myx
        self.y = myy
        self.dx = mydx
        self.dy = mydy
        self.side = myside
        self.color = mycolor

    # Call each frame.
    def move(self):
        self.x += self.dx
        self.y += self.dy

    # Executes updating dx as necessary for bouncing off the left wall.
    def bounceLeft(self):
        if self.x + self.dx < 0:
            self.dx = -self.dx
```

```
    # Executes updating dx as necessary for bouncing off the right wall.
    def bounceRight(self, SCREEN_W):
        if self.x + self.dx > SCREEN_W-self.side:
            self.dx = -self.dx

    # Executes updating y as necessary for bouncing off the top wall.
    def bounceUp(self):
        if self.y + self.dy < 0:
            self.dy = -self.dy

    # Executes updating y as necessary for bouncing off the bottom wall.
    def bounceDown(self, SCREEN_H):
        if self.y + self.dy > SCREEN_H-self.side:
            self.dy = -self.dy

    # Adds addDX to the the change in x per frame, and adds addDY to
    # the change in y per frame.
    def changeVelocity(self,addDX,addDY):
        self.dx += addDX
        self.dy += addDY

    # Update for a single frame. Maybe this will typically be overridden.
    def updateFrame(self, DISPLAYSURF):
        self.move()
        self.bounceLeft()
        self.bounceRight(DISPLAYSURF.get_width())
        self.bounceUp()
        self.bounceDown(DISPLAYSURF.get_height())
        self.draw(DISPLAYSURF)

    # Draws this object on the display surface as a circle.
    # Likely to be overridden most of the time.
    def draw(self, DISPLAYSURF):
        pygame.draw.ellipse(DISPLAYSURF,    self.color,    (self.x,    self.y,
self.side, self.side), 0)

    # Returns true iff the two circles represented by self and other intersect.
    # Should be overridden for different objects.
    def collide(self, other):

        # I know these aren't the centers, but this difference is the same as
the
        # difference of the x and y coordinates of the centers.
        center_dx = self.x - other.x
        center_dy = self.y - other.y
        dist_sq = center_dx*center_dx + center_dy*center_dy
        dist_sq_radii   =   (self.side//2   +   other.side//2)*(self.side//2   +
other.side//2)
        return dist_sq <= dist_sq_radii
```

Let's take this apart. The constructor is fairly straightforward. It simply assigns each of the instance variables directly to the information passed to it from the formal parameters. The move function should look familiar. It simply updates the x and y coordinates of self via the current values of dx and dy. The four bounce methods take care of updating dx and dy in the case we would like for our object to bounce off of those particular walls. The updateFrame method should only be used if we want to bounce off of all four walls. Otherwise, a combination of the move and draw methods should be used. Finally, the collide method checks for a collision between self and another token. This assumes that both objects are circles.

Since it's typical in other languages to include a new class in its own file, we'll introduce how to write multi-file Python programs now. Let's assume we store the token class in the file, TokenFile, and that we wanted to use a token object (or a few of them) in the file bounceballsplit.py. Then in the latter file, we would need the following import statement:

```
from TokenFile import token
```

Finally, we must place TokenFile in the same directory as bounceballsplit. Once we do all of this, we can then instantiate (create) a token object in bounceballsplit and then use it.

For this example, we'll create ten tokens (all blue for this example) and have each of them randomly bouncing around. Each will be given a random x and y position as well as a random dx and dy. For this example, the magnitude of the velocity of each token will never change. (Specifically, |dx| and |dy| will remain the same.)

Before the start of our game loop, we will create ten tokens in random locations with random directions and store them in a list. Then, in our game loop, we'll call the appropriate token methods as needed to get our desired behavior: the balls continuing to bounce around each of the walls. Just to show that the collision function is working, whenever two tokens collide, a print statement (going to the console) is executed.

There is only one function other than main in bounceballsplit:

```
# Returns a random integer in between low and high not equal to 0.
def myrand(low,high):
    res = 0
    while res == 0:
        res = random.randint(low, high)
    return res
```

This function returns a non-zero integer in between low and high, inclusive. This will help us create the initial direction of movement for each token.

The first part of the main method (included on the next page) is the code to create the ten bouncing balls at their initial position:

```
mytokens = []
for i in range(NUM_BALLS):

    # Somewhere on the screen.
    x = random.randint(1, SCREEN_W-BALL_D)
    y = random.randint(1, SCREEN_H-BALL_D)

    # Random non-zero movement in both directions.
    dx = myrand(-2,2)
    dy = myrand(-2,2)

    mytok = token(x,y,dx,dy,BALL_D,BLUE)
    mytokens.append(mytok)
```

We generate some appropriate random values and then use those to create a token object, which we then add to our list, mytokens.

In the game loop, we just update the location of each of the tokens, and for kicks, look for collisions, printing out which token has collided with which token (this prints out the indexes into the main list):

```
# Game loop.
while True:

    # Look for events - we aren't using this right now.
    for event in pygame.event.get():

        if event.type == QUIT:
            pygame.quit()
            sys.exit()

    # White background.
    DISPLAYSURF.fill(WHITE)

    # Essentially update each ball.
    for item in mytokens:
        item.updateFrame(DISPLAYSURF)

    # Update what we put on the canvas.
    pygame.display.update()

    # Just testing collision code.
    for i in range(len(mytokens)):
        for j in range(i+1, len(mytokens)):
            if mytokens[i].collide(mytokens[j]):
                print("collide",i,j)

    # Wait a bit!
    clock.tick(100)
```

The key here is to notice that the token class is really doing the heavy lifting. Once we have all of movement and display logic in the class, we can fairly easily call what we need from here and it's pretty easy to look at the whole thing to see the overall flow of what's happening. Eventually, we'll

see how inheritance can really help. We can create a class that uses all of this code (inherits it) but then just adds the specific code that it needs that isn't already included here.

Rain Example

For the remainder of this lecture we'll show one more application of the token class: we'll simulate rain falling. Similar to the previous example, raindrops will be circles that are randomly generated. In this example, all the raindrops will fall straight down. (Though in Florida, we are known to have slanty rain!) Also, different raindrops will be given different random colors. There are two key things different about the movement and displaying raindrops:

1. The raindrops won't bounce off any of the surfaces. (So we'll no longer use the update Frame method. It's still there, but we just won't call it. This is true of many built in classes, where we use objects but not all of the methods provided for them.)

2. All the raindrops shouldn't persist, because once they hit the ground, they should no longer appear. Thus, we'll need to add the ability to remove a drop from our list of drops. We'll do this through a static method outside of the class.

The two functions (other than main) in rain.py are as follows:

```
# This function handles moving each item listed in items.
def move(items):
    for item in items:
        item.move()

# This function removes all items that will never be visible again.
def removeUseless(items):
    for item in items:
        if item.y > SCREEN_H:
            items.remove(item)
```

The first will move all of the items in the list passed to it. This just calls the move method (instead of the updateFrame method).

The second will remove all raindrops that have fallen below the bottom of the screen so that we don't waste processing time on those drops when they don't show on the display.

In this program, every frame, we'll add in between 1 and 10 new raindrops. To do this, we'll generate several random x values (y will be 0 initially) and create token object representing the rain for each of those x values. Each of the raindrops will have the same initial dy value. Then, each of these token objects will be added to the list of current raindrops. After that, each raindrop is drawn and the screen updated.

Finally, some processing is done to get ready for the next frame: to mimic gravity somewhat, every five frames, the dy value of the drop is increased by 1. (To do this, we keep a frame counter and use mod.) Then, we move the drops, remove the ones which have gone off the screen, update the frame count and wait a bit. Here's the code:

```
def main():

    # Basic setup.
    pygame.init()
    DISPLAYSURF = pygame.display.set_mode((SCREEN_W, SCREEN_H))
    pygame.display.set_caption("Let it rain!")
    clock = pygame.time.Clock()

    # Store all raindrops here.
    rain = []

    frame = 0

    # Game loop.
    while True:

        for event in pygame.event.get():
            if event.type == QUIT:
                pygame.quit()
                sys.exit()

        # Calculate number of new drops to add this iteration, then randomly
        # generate that many unique values.
        numNewRain = random.randint(1, 10)
        xvals = set()
        while len(xvals) < numNewRain:
            x = random.randint(1, SCREEN_W)
            xvals.add(x)

        # Add each if these items into the list rain.
        for val in xvals:
            r = random.randint(0,255)
            g = random.randint(0,255)
            b = random.randint(0,255)
            rain.append(token(x,0,0,DY,RADIUS,pygame.Color(r,g,b)))

        DISPLAYSURF.fill(BLACK)

        # Draw each raindrop individually.
        for item in rain:
            item.draw(DISPLAYSURF)

        pygame.display.update()

        if frame%5 == 0:
            for item in rain:
                item.changeVelocity(0,1)

        # Move the drops for the next iteration and remove useless ones.
        move(rain)
        removeUseless(rain)

        frame += 1
        clock.tick(30)
```