Name: _____

Paddling Around

Professor Bob Brown – Kennesaw State University
Bob.Brown@Kennesaw.edu

Class and Object

As we said earlier, Python is an *object oriented language*. Objects contain both code, called *methods*, which define the behavior of the object, and data, called *properties*. Each object is described by a *class* definition. In Python, class names begin with a capital letter.

The class is not the object; instead, you can think of it as a template or mold for creating objects. Creating an object from its class definition is called *instantiating* an object. The object created is called an *instance* of the class.

Sometimes, but not always, making an instance of an object involves executing code. That code is called a *constructor*, and by convention in Python, constructors are named __init__. (That's two underscores.)

Inheritance

Inheritance is a key feature of true object oriented languages. That means we can create new classes (not objects) with all the properties and methods of an existing class, and some new properties and methods as well. The existing class from which the new class inherits is called the *parent class* or *superclass*. For example, if there is an existing class Mammal, we can define a new class, perhaps Dog, that inherits properties and methods from Mammal. We don't have to repeat them; we just add or change those properties and methods the separate Dog from the generic Mammal.

Paddles for Our Pong Game

In game development, a *sprite* is something that can move around in the game space. The PyGames library defines a generic class called Sprite. The code below creates a class called Paddle that inherits the properties of Sprite. This class definition has its own constructor, called __init__, and also calls the constructor of the superclass, called super().__init__, which refers to the constructor of the parent, or super class.

Last update: 2020-03-09

It is important to remember that this doesn't create any paddles, it just provides a template for building them later. The *self* keyword refers to the object being constructed. Type the code below and save it as a file called paddle.py.

```
import pygame
BLACK = (0,0,0)

class Paddle(pygame.sprite.Sprite):
    # This class represents a paddle.
    # It derives from the "Sprite" class in Pygame.
    def __init__(self, color, width, height):
        # Call the parent class (Sprite) constructor
        super().__init__()

        # Pass in the color of the paddle,
        # and its x and y position, width and height.
        # Set the background color and set it to be transparent
        self.image = pygame.Surface([width, height])
        self.image.fill(BLACK)
        self.image.set_colorkey(BLACK)

        # Draw the paddle (a rectangle!)
        pygame.draw.rect(self.image, color, [0, 0, width, height])

        # Fetch the rectangle object that has the dimensions of the image.
        self.rect = self.image.get_rect()
```

Using the Paddle Class

Return to the code from the second class, the skeleton of our Pong game. We are going to instantiate two paddles, one for each player, using the Paddle class we just wrote.

Just below the line that says `import pygame` add a new line, `from paddle import Paddle`

After `pygame.display.set_caption("Pong")` add a blank line, then the following, not indented.

```
paddleA = Paddle(WHITE, 10, 100)
paddleA.rect.x = 20
paddleA.rect.y = 200

paddleB = Paddle(WHITE, 10, 100)
paddleB.rect.x = 670
paddleB.rect.y = 200
```

This defines two white paddles named paddleA and PaddleB, 10 pixels wide and 100 pixels tall, one 20 pixels from the left edge of the screen and one 670 pixels from the edge. They're both 200 pixels from the top.

Immediately after the code you just entered, add another blank line and the following:

```
# This will be a list that will contain all the sprites we
# intend to use in our game.
all_sprites_list = pygame.sprite.Group()

# Add the paddles to the list of sprites
all_sprites_list.add(paddleA)
all_sprites_list.add(paddleB)
```

This defines a list of the sprites used in the game so far and adds paddleA and paddle to the list.

Under the comment line `# --- Game logic should go here` add a line:

```
  all_sprites_list.update()
```

It should be indented the same amount as for event in pygame.event.get() five lines up. Depending on how you entered the first code, this should be one tab. This updates the list of sprites once on each pass through the game's main loop.

```
Finally, under pygame.draw.line(screen, WHITE, [349, 0], [349, 500], 5) add a
blank line and the following:
    #Now let's draw all the sprites in one go.
    all_sprites_list.draw(screen)
```

It should be indented the same as the other lines you just added, one tab.

Try running your program. If you get error messages, read them carefully and see whether you can find this mistakes.

When everything runs, you'll have a screen with two paddles! Close the screen with the STOP icon near the top pf the Thonny window.

Make the Paddles Move

One of the great things about object oriented programming is that by changing a class definition and re-running our program, we can change all the objects created from that class definition, no matter how many there were!

We're going to add two methods, `moveUp` and `moveDown`, to our paddles. Open paddle.py and add the following code at the end:

```
    def moveUp(self, pixels):
        self.rect.y -= pixels
            #Check that you are not going too far (off the screen)
        if self.rect.y < 0:
          self.rect.y = 0

    def moveDown(self, pixels):
        self.rect.y += pixels
      #Check that you are not going too far (off the screen)
        if self.rect.y > 400:
          self.rect.y = 400
```

Let's take a closer look at **moveUp**. The things in parentheses on the first line are what's passed to the method. As we said, *self* refers to whichever paddle gets the method call. *Pixels* is how far to move, and of course, we're moving up. Since the Y coordinate of the paddle starts at the top of the screen, we must *subtract* from the Y coordinate to move up. The **-=** operator says to subtract the value on the right, *pixels*, from the value on the left, the Y coordinate, and store the result back in the value on the left. We can't just subtract, however; we have to be sure a player can't move the paddle off the top of the screen. That's what the check for the Y coordinate being less than zero does.

The **moveDown** method adds to the Y coordinate, and checks that we haven't moved off the bottom of the 400 pixel tall screen.

To actually use these two methods, we have to call them from the Pong program. This is a two-player game. The left player will use the w and s keys on the keyboard to move up and down. The right player will use the up and down arrows. Add the following code *before* the # --- Game logic should go here comment, indented by one tab or four spaces, to line up with the comment itself.

```
# Moving the paddles when the user uses the arrow keys (player A)
# or "W/S" keys (player B)
keys = pygame.key.get_pressed()
if keys[pygame.K_w]:
    paddleA.moveUp(5)
if keys[pygame.K_s]:
    paddleA.moveDown(5)
if keys[pygame.K_UP]:
    paddleB.moveUp(5)
if keys[pygame.K_DOWN]:
    paddleB.moveDown(5)
```

Try running your program again. Correct any errors and move the paddles!

Now that we can use the keyboard, we can make the Escape key end the game. After these two lines:

```
if event.type == pygame.QUIT: # If user clicked close
    carryOn = False # Flag that we are done so we exit
```

add these:

```
elif event.type==pygame.KEYDOWN:
    if event.key==pygame.K_ESCAPE: #Pressing the Esc Key will quit the game
        carryOn=False
```

The **elif** should line up with the **if** a couple of lines up.

Run it again and test that the paddles still move and the Escape key ends the game.