# CS: Objects in Python

# Inheritance

# Lab - Inheritance

# Inheritance Lab 1

---

## Lab 1 - Setting up PyGame

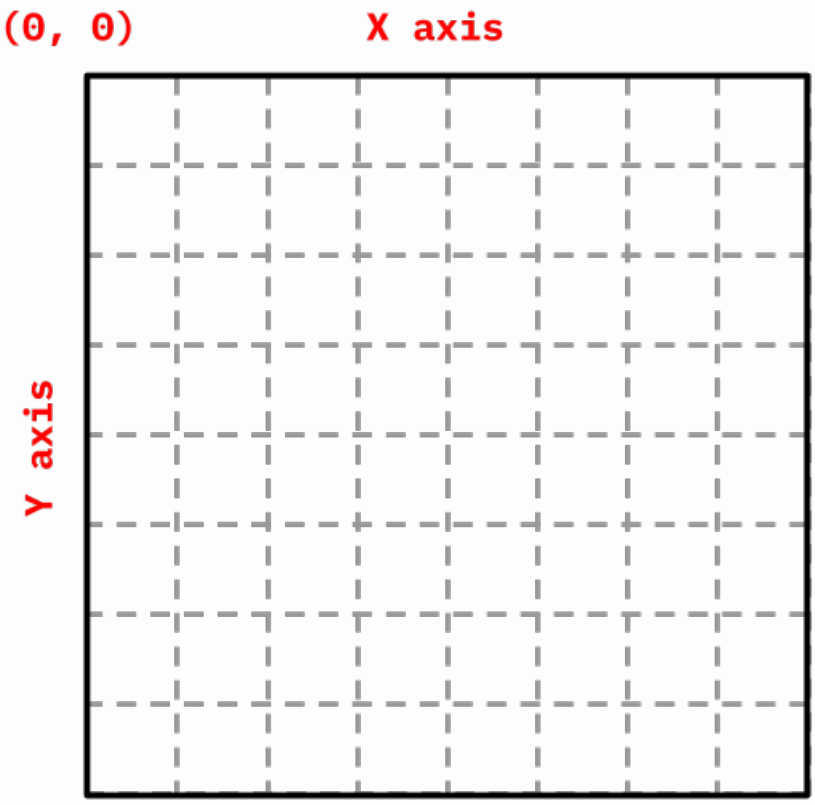
This lab is going to use PyGame to create some animations using inheritance. Here are steps needed to set up a PyGame project.

1. Import the Pygame module.
2. Initialize the project. This, essentially, “turns on” the different features of Pygame like sound, the keyboard, the mouse, etc.
3. Create a window. This window will be 400 pixels wide by 400 pixels tall. There needs to be two sets of parentheses when creating the window.
4. Create a caption for the window
5. The clock controls how fast the animation runs
6. Create a main loop. The main loop is where the drawing and animating takes place. Without this loop, the window would appear and then disappear almost immediately. The loop is checking for the `pygame.QUIT` event, which happens when the user closes the Pygame window. The main loop also fills the window with a color and updates the animation with any changes.
7. Quit Pygame once the main loop stops running.

```
import pygame

pygame.init()
window = pygame.display.set_mode((400, 400))
pygame.display.set_caption("Inheritance Animation")
clock = pygame.time.Clock()

run = True
while run:
  pygame.time.delay(100)
  window.fill((12, 24, 186)) #change the color of the window

  for event in pygame.event.get():
    if event.type == pygame.QUIT:
      run = False

  pygame.display.update() #update the window with any changes
  clock.tick(30)
pygame.quit()
```

## PyGame Refresher

- The Window - Pygame programs are run in a window, which is a grid of x- and y-coordinates. The origin point (0, 0) is located in the top-left corner. Use window coordinates to position objects in the window.




- Colors - PyGame uses the RGB (red, green, blue) system of defining colors. Color values range between 0 and 255, and these values are mixed together. Use this website to help you find the RGB value you want.
- This lab will use the polygon shape from the PyGame library. The command draw a polygon is `pygame.draw.polygon`. The parameters for a polygon are a surface, a color, and a list of points for each vertex of the polygon. Add the following line of code to the main loop after `window.fill`. You should see a red diamond.

```
pygame.draw.polygon(window, (231, 76, 60), [(200, 0), (400, 200), (200
```

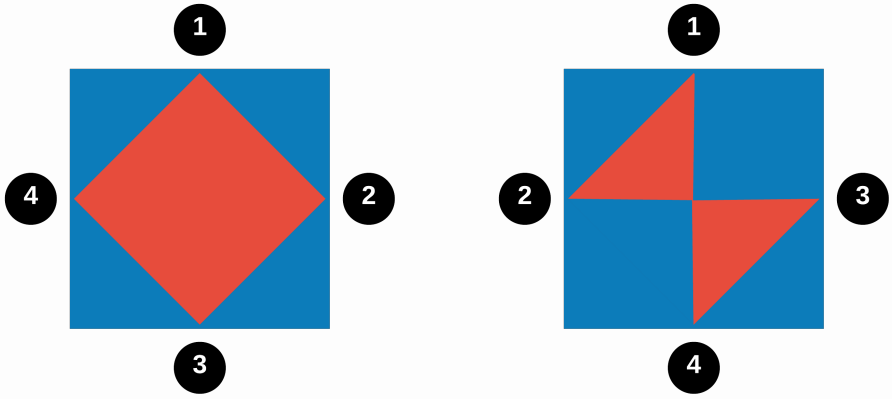
- Tuples - Using PyGame you will see numbers grouped together with parentheses. This data structure is called a tuple. You will often see them for coordinate pairs and RGB colors. You can reference the information inside a tuple just as you would a list, with square brackets and an index.

```
(255,218,185) # peach color in RGB
```

challenge

## Try these variations:

- Change the order of coordinates to be `[(200, 0), (0, 200), (400, 200), (200, 400)]`.
  ▼ **What happened?**
  PyGame draws a polygon in the order of the coordinates. If you want a regular polygon, the coordinates need to go in either clockwise or counterclockwise order. The first polygon used a clockwise pattern, while the second one used a zigzag pattern.



# Inheritance Lab 2

---

## Lab 2 - Creating the Parent Class

This lab is all about inheritance, so we need to create a parent class from which we can inherit. We are going to create a class called `Hexagon` that draws a hexagon to window.

### Setting up PyGame

This lab comes with some code pre-populated in the IDE. This is the basic code needed to get a main loop running. The window is 400 by 400 and has a dark gray background. Notice too that the `math` module has been imported as well.

### The `Hexagon` Class

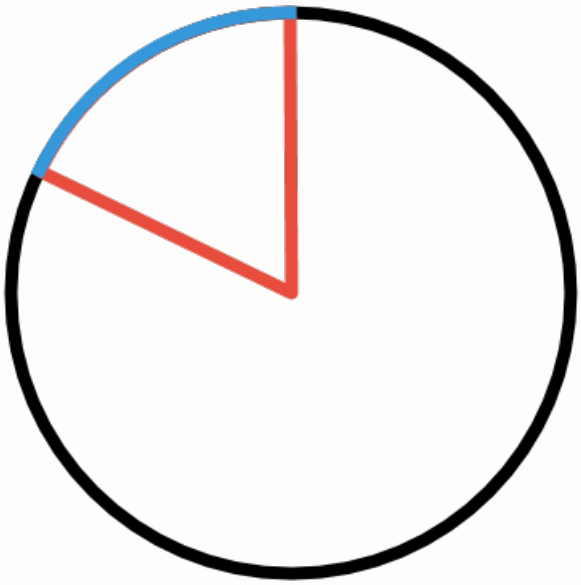
There is a comment in the code indicating where to start the `Hexagon` declaration. The class should be declared before the main loop. Even though there is a command to draw a polygon, all of the information will be stored as an object.

```
# Declare the Hexagon class here
class Hexagon:
  def __init__(self, center, radius, window):
    self.center = center
    self.radius = radius
    self.window = window
    self.points = 6
    self.angle = math.radians(360 / self.points)
    self.color = (179, 55, 113)
    self.stroke_weight = 3
```

▼ **What are radians?**

In the constructor, `self.angle` is converted from degrees to radians. A radian is a unit of measure for angles. One radian is about 57.3 degrees. The are exactly 2π radians in any circle. The image below shows that the a

radian (blue) has the same length as the radius (red).



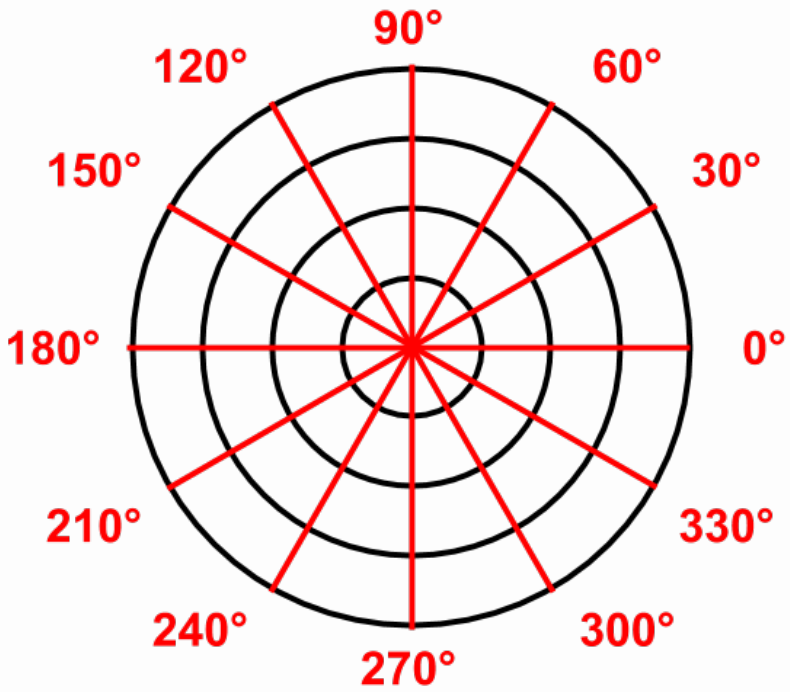
## Drawing the Hexagon

Create a method called `Draw`. In it, use the PyGame command to create a polygon. Start by using `self.window` and `self.center`. The problem arises when you need a list of coordinate pairs. There are no attributes in the constructor that correspond to the position of the vertices. We need a helper method that returns a list of six coordinate pairs. Where the list of coordinates goes, use `self.calculate_vertices()` (our helper method). Finish the polygon with `self.stroke_weight`. This tells PyGame to not fill the hexagon, but draw an outline with a stroke weight of 3. Declare the method `calculate_vertices` but set its body to `pass` for now.

```
def calculate_vertices(self):
  pass

def draw(self):
  pygame.draw.polygon(self.window, self.color, self.calculate_vertic
```

## Polar to Cartesian Transformation

The `Hexagon` class is designed to position the shape according to `self.center`. The six vertices radiate out from the center according to `self.radius`, and the spacing between each vertex is controlled by `self.angle`. Using angles and a distance from a central point to calculate a position is referred to as the polar coordinate system. The red lines show different angles and the black concentric circles represent a distance from the center. The intersection of the red line and black circle is a location.




Polar Coordinates

The problem is that the PyGame window uses the Cartesian coordinate system. With a little bit of trigonometry, you can convert the polar coordinate system to the Cartesian coordinate system. The x- and y-coordinates for each point of the hexagon can be calculated with the following formulas.

```
x = cosine(angle) * radius
y = sine(angle) * radius
```

These formulas need to be tweaked a bit for Python. First, the above formulas assume you are setting a position relative to the origin point (the top-left corner of the window). Instead, we want to position each vertex in relation to the center of the hexagon. So offset the x-coordinate of the

vertex with the x-position of the center of the hexagon. Similarly, offset the y-coordinate of the vertex with the y-position of the center of the hexagon. Since there are six vertices, use a for loop to calculate each vertex. Group the x- and y-coordinates together as a tuple, and append this to a list of coordinate pairs. Once the loop has run, return the list

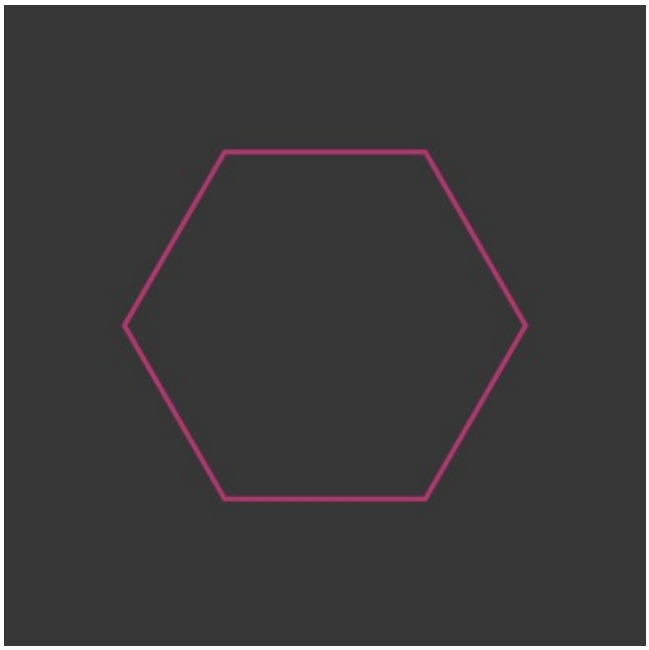
```
  def calculate_vertices(self):
    vertices = []
    for i in range(self.points + 1):
      x = math.cos(i * self.angle) * self.radius + self.center[0]
      y = math.sin(i * self.angle) * self.radius + self.center[1]
      vertices.append((x, y))
    return vertices

  def draw(self):
    pygame.draw.polygon(self.window, self.color, self.calculate_vertic
```

### Testing the `Hexagon` Class

To be sure that the `Hexagon` class works, we are going to add an object and draw it to the window. Look for the global variables section and instantiate a `Hexagon` object. We want it to be centered in the window, so set its coordinates to `(200, 200)`, give it a radius of `125`, and pass it the PyGame `window`. Go into the main loop and call `hexagon.draw()` after the window has been filled. Run the code; you should see a hexagon in the window.

```
# global variables
run = True
hexagon = Hexagon((200, 200), 125, window)

# main loop
while run:
  pygame.time.delay(100)
  window.fill((55, 55, 55))
  hexagon.draw()
```

▼ **Expected Output**



challenge

## Try these variations:

- Add more `Hexagon` objects to the window. Name them `hexagon2`, `hexagon3`, etc. and draw them to the window.

# Inheritance Lab 3

---

## Lab 3 - Extending the Parent Class

Now that we have a parent class, it is time to extend it through inheritance. We want a more general `Polygon` class that can have a user-defined number of vertices. We also want to control the color. Start by creating the `Polygon` class as a child of the `Hexagon` class. Then override the constructor so `points` and `color` are user-defined. The parameter `color` has a default value of `(179, 55, 113)` so you do not have to provide a color. **Note:** PyGame requires that there be at least three vertices when drawing a polygon.

```
class Polygon(Hexagon):
  def __init__(self, center, radius, window, points, color=(179, 55, 1

    self.center = center
    self.radius = radius
    self.window = window
    self.points = points
    self.angle = math.radians(360 / self.points)
    self.color = color
    self.stroke_weight = 3
```

Next, go to the global variable section and declare the object `polygon`. In the main loop, call the `draw` method for `polygon`. Run your code to make sure you see an octagon. **Note:** you might want to remove references to `Hexagon` objects.

```
# global variables
run = True
polygon = Polygon((200, 200), 140, window, 8, (255, 19, 30))

# main loop
while run:
  pygame.time.delay(100)
  window.fill((55, 55, 55))
  polygon.draw()
```

## Animating the Polygon

We are going to animate the polygon by overriding the `calculate_vertices` method. Instead of having a fixed radius (the distance from the vertices from the center of the polygon), we are going to create a radius that increases and decreases. This will give the appearance that the polygon is growing and shrinking.

The new version of `calculate_vertices` adds the variable `length`. This replaces `self.radius` as the distance between the vertex and the center of the polygon. The value for `length` is determined by the helper method `calculate_length`.

```
  def calculate_vertices(self):
    vertices = []
    length = self.calculate_length()
    for i in range(self.points + 1):
      x = math.cos(i * self.angle) * length + self.center[0]
      y = math.sin(i * self.angle) * length + self.center[1]
      vertices.append((x, y))
    return vertices
```

The `calculate_length` method needs to access a persistent value that changes over time. That means this value should be an attribute of the `Polygon` class. Add `self.grow = 0` to the end of the constructor.

```
class Polygon(Hexagon):
  def __init__(self, center, radius, window, points, color=(179, 55, 1

    self.center = center
    self.radius = radius
    self.window = window
    self.points = points
    self.angle = math.radians(360 / self.points)
    self.color = color
    self.stroke_weight = 3
    self.grow = 0
```

Now define the `calculate_length` method. Increment `self.grow`. The rate at which `self.grow` changes is the rate at which the polygon grows and shrinks; 0.1 is a good starting value. The variable `length` represents the new distance between the vertex and the center of the polygon. `math.sin` is used because this function will return a value between -1 and 1. When the result is positive, the polygon grows. When the result is negative, the polygon shrinks. It is important to multiply `math.sin(self.grow)` by `self.radius`. If not, maximum size the polygon can have is 1 pixel. Return `length` so it can be used to calculate the coordinate pairs for each vertex.

```
  def calculate_length(self):
    self.grow += 0.1
    length = math.sin(self.grow) * self.radius
    return length
```

▼ **Expected Output**