Chapter 11: Getting Started with Pygame with Bouncing Balls

Contents

🚦 No AI use

Time required: 90 minutes

DRY

Don't **R**epeat **Y**ourself

Code Shape

Please group program code as follows.

- Declare constants and variables

- Get input
- Calculate
- Display

Visualize and Debug Programs

The website www.pythontutor.com helps you create visualizations for the code in all the listings and step through those programs one instruction at a time. As you do that, pay attention to the movements of the red and green arrows on the left, the diagram on the right, and the printed material at the bottom. This will help you to better understand the behavior of the programs we are working on.

This is a great way to debug your code. You can see the variables change as you step through the program.

www.pythontutor.com

What is Pygame?

The Pygame library is the most well-known python library for making games. It's not the most advanced or high-level library. It is simple and easy to learn (comparatively). Pygame serves as a great entry point into the world of graphics and game development.

Pygame is installed with Python. Pygame is a framework that includes several modules with functions for drawing graphics, playing sounds, handling mouse input, and other game like things. Pygame provides functions for creating programs with a **graphical user interface,** or **GUI** (pronounced, "gooey").

Install and Update Pygame with pip

pip is a package manager for **Python**. It's a tool that allows you to install and manage additional libraries and dependencies that are not distributed as part of the standard library.

Install Pygame-ce:

```
pip install pygame-ce
```

If Pygame is already installed, you may want to update Pygame:

```
pip install pygame-ce -U
```

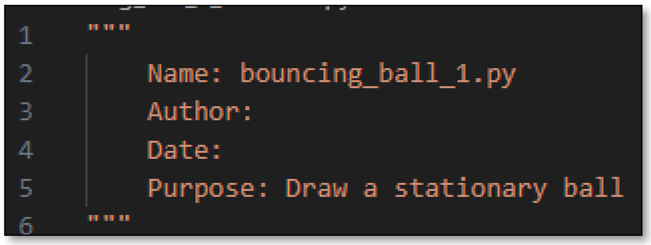
You are now ready to embark on your successful and prosperous game development career in Python!

Tutorial 1: Draw a Stationary Ball

This code is a Python script using the Pygame library to create a window and draw a stationary ball on it.

Type in the following code and save the file as **bouncing_ball_1.py**.

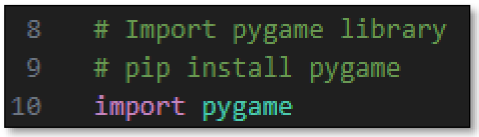
Header Comments:



```
"""
    Name: bouncing_ball_1.py
    Author:
    Date:
    Purpose: Draw a stationary ball
"""
```

The header provides information about the script, such as its name, author, date, and purpose.

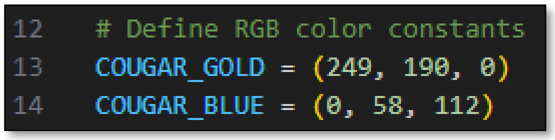
Import pygame:



```
# Import pygame library
# pip install pygame
import pygame
```

This line imports the Pygame library, which is used for developing games and multimedia applications in Python.

Define RGB Color Constants:



```
# Define RGB color constants
COUGAR_GOLD = (249, 190, 0)
COUGAR_BLUE = (0, 58, 112)
```

RGB (Red, Green, Blue) color constants are defined for later use. COUGAR_GOLD is a golden color, and COUGAR_BLUE is a specific shade of blue. These are the official RGB values for WNCC.

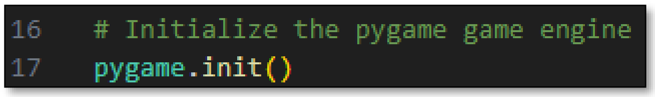
RGB

RGB stands for Red, Green, Blue, which are the primary colors of light used in digital displays. In the RGB color model, colors are created by combining different intensities of these three primary colors. Each color is represented by a set of three values, indicating the intensity of red, green, and blue respectively.

- **Red (R)**: Controls the amount of red light. Higher values result in more intense red color.
- **Green (G)**: Controls the amount of green light. Higher values result in more intense green color.
- **Blue (B)**: Controls the amount of blue light. Higher values result in more intense blue color.

Each of the RGB values typically ranges from 0 to 255, where 0 means none of that color and 255 means the maximum intensity. By combining different intensities of these three colors, a wide range of colors can be represented on digital screens. For example, (255, 0, 0) represents pure red, (0, 255, 0) represents pure green, and (0, 0, 255) represents pure blue.

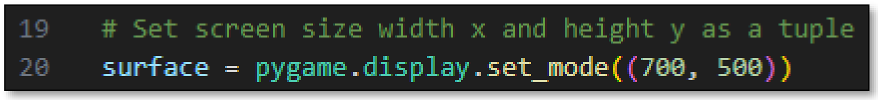
Initialize pygame:



```python
# Initialize the pygame game engine
pygame.init()
```

Initializes the Pygame game engine. This must be done before using any Pygame functions.

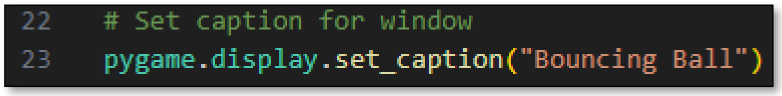
Set Screen Size:



```python
# Set screen size width x and height y as a tuple
surface = pygame.display.set_mode((700, 500))
```

This line creates a Pygame display surface with a size of 700 pixels in width and 500 pixels in height. The size is specified as a tuple **(700, 500)**.

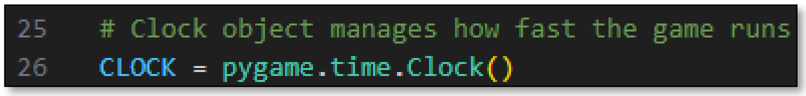
Set Window Caption:



```python
# Set caption for window
pygame.display.set_caption("Bouncing Ball")
```

This sets the caption or title for the Pygame window to "Bouncing Ball". It is the text that typically appears at the top of the window.

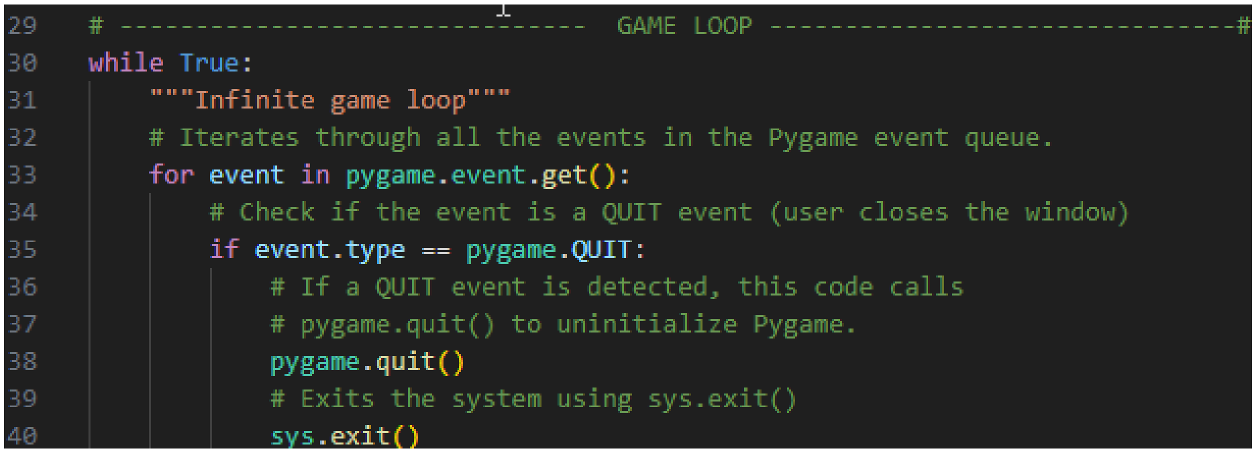
Clock Object:



```python
# Clock object manages how fast the game runs
CLOCK = pygame.time.Clock()
```

This line creates a Pygame clock object (`CLOCK`). The clock is used to control the frame rate of the game, ensuring that the game runs at a consistent speed across different systems. It is often used with the **tick()** method to regulate the frame rate.

Game Loop:



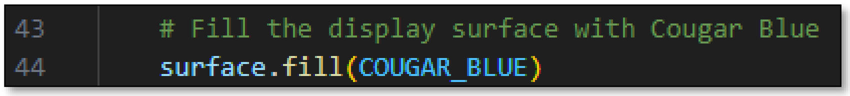
```python
# ------------------------------  GAME LOOP ------------------------------#
while True:
    """Infinite game loop"""
    # Iterates through all the events in the Pygame event queue.
    for event in pygame.event.get():
        # Check if the event is a QUIT event (user closes the window)
        if event.type == pygame.QUIT:
            # If a QUIT event is detected, this code calls
            # pygame.quit() to uninitialize Pygame.
            pygame.quit()
            # Exits the system using sys.exit()
            sys.exit()
```

This creates an infinite loop, commonly known as the game loop. It continuously repeats the code inside, managing the game's ongoing processes.

A game loop keeps the game running smoothly by handling input, updating the game state, rendering graphics, and repeating these steps in a loop. It allows games to react to user actions in real-time and maintain a coherent and engaging experience. The loop continues until the game is exited or a specific condition is met.

All code from this point is inside the game loop.

Fill the Display Surface:

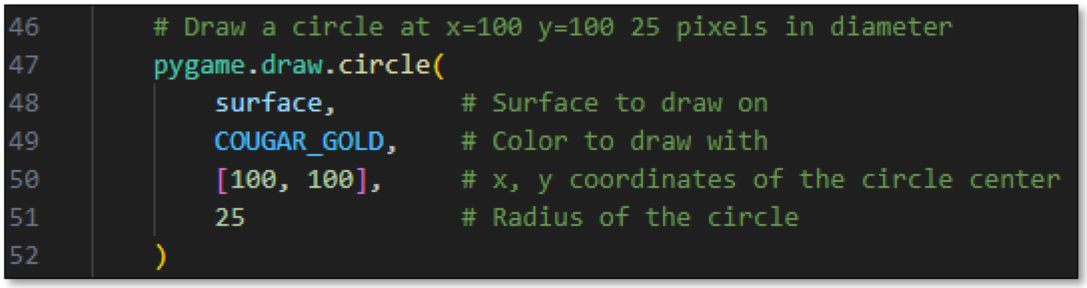


```python
    # Fill the display surface with Cougar Blue
    surface.fill(COUGAR_BLUE)
```

Fills the entire display surface with the color defined as **COUGAR_BLUE**. This effectively clears the screen before drawing the next frame.

Draw a Circle:

Draws a filled circle on the display surface at coordinates **(x=100, y=100)** with a radius of **25** pixels. The circle is filled with the color defined as **COUGAR_GOLD**



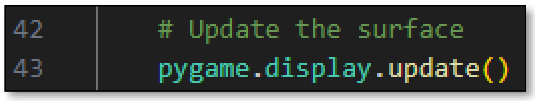
```python
    # Draw a circle at x=100 y=100 25 pixels in diameter
    pygame.draw.circle(
        surface,        # Surface to draw on
        COUGAR_GOLD,    # Color to draw with
        [100, 100],     # x, y coordinates of the circle center
        25              # Radius of the circle
    )
```

Draw Circle:

This code uses the Pygame library to draw a filled circle on a surface.

- **pygame.draw.circle -** This is the function used to draw a circle.
- **surface -** The surface on which the circle will be drawn.
- **COUGAR_GOLD -** The color of the circle.
- **[100, 100] -** The coordinates of the center of the circle. In this case, it's [100, 100].
- **25 -** The radius of the circle.

Update the Surface:



```python
    # Update the surface
    pygame.display.update()
```

Updates the display to show the changes made in the current frame. This is essential for rendering the drawn circle on the screen.

Backbuffer

The term "backbuffer" refers to a secondary or off-screen buffer used during the rendering process. The backbuffer is closely related to the concept of double buffering, which is employed to prevent visual artifacts like flickering in animations.

Here's how it relates to pygame.display.update():

1. **Double Buffering:**

    a. In a double-buffering system, there are two buffers: the front buffer and the back buffer. The front buffer is the buffer currently being displayed on the screen, while the back buffer is where rendering operations occur without being immediately visible to the user.

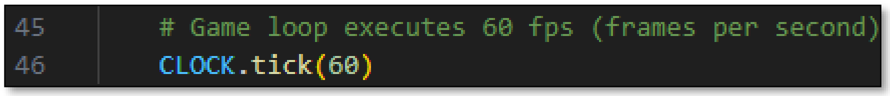
2. **Rendering to the Back Buffer:**
    a. When you perform drawing operations using Pygame, you are often drawing to the back buffer. This includes drawing shapes, images, or any other visual elements that make up the current frame of your game or application.

3. **Flipping Buffers:**
    a. After rendering the entire frame to the back buffer, the buffers are "flipped." This means the back buffer becomes the front buffer, and vice versa.

4. **pygame.display.update():**
    a. When you call **pygame.display.update()**, it tells Pygame to update the contents of the front buffer, making the changes from the back buffer visible on the screen. It's the point where the rendered frame becomes visible to the user.

5. **Preventing Flickering:**
    a. Double buffering helps prevent flickering issues that can occur when directly rendering to the front buffer. The back buffer provides a stable environment for drawing, and the switch between buffers is done quickly to minimize any visible artifacts.

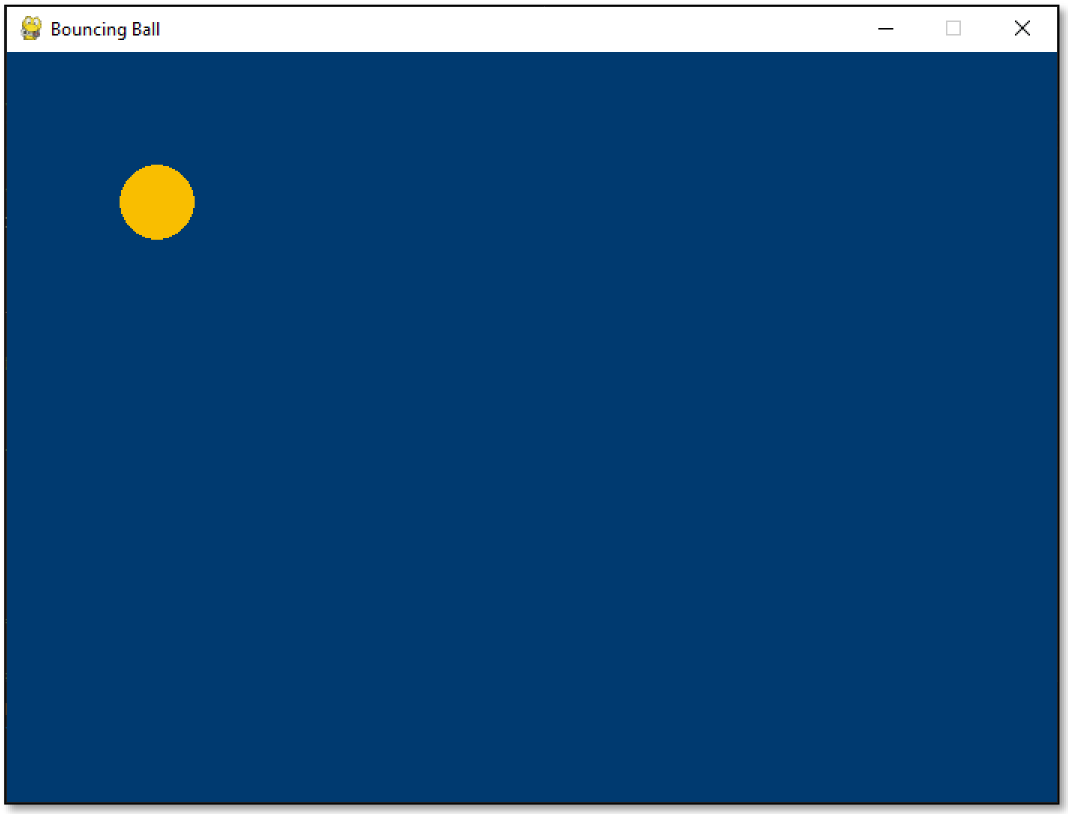
Control Frame Rate:



```
    # Game loop executes 60 fps (frames per second)
    CLOCK.tick(60)
```

Limits the frame rate to 60 frames per second (fps). It uses the clock object (`CLOCK`) created earlier to control the speed of the game loop. This ensures that the game runs at a consistent frame rate, providing smooth animation.

Example run:



Yay! You've just made the world's most boring video game! It's just a blank window with Hello World! at the top of the window in the title bar.

Creating a window and drawing an image is the first step to making graphical games.

Game Loops and Game States

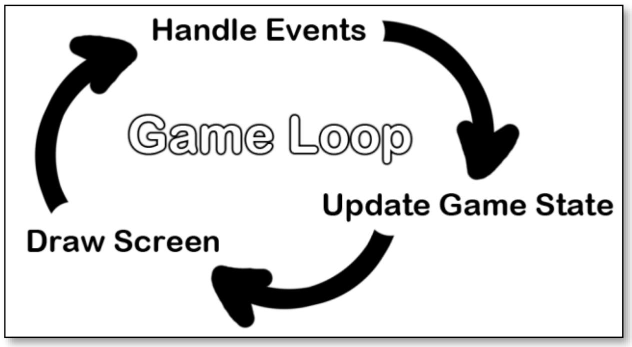
Every game that has what is called a "Game Loop". It's a standard game development concept.

The Game Loop is where all the game events occur, update, and get drawn to the screen. Once the initial setup and initialization of variables is out of the way, the Game Loop begins where the program keeps looping over and over until an event of type **QUIT** occurs.

A game loop (also called a main loop) is a loop where the code does three things:

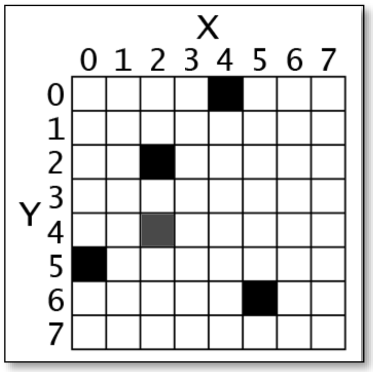
1. Handles events
2. Updates the game state
3. Draws the game state to the screen

Since the game state is usually updated in response to events (such as mouse clicks or keyboard presses) or the passage of time, the game loop is constantly checking and re-checking many times a second for any new events that have happened. Inside the main loop is code that looks at which events have been created (with Pygame, this is done by calling the **pygame.event.get()** function). The main loop also has code that updates the game state based on which events have been created. This is usually called event handling.



Pixel Coordinates

The window that the "POng" program creates is composed of little square dots on the screen called pixels. Each pixel starts off as black but can be set to a different color. Imagine that instead of a Surface object that is 300 pixels wide and 300 pixels tall, we just had a Surface object that was 8 pixels by 8 pixels. If that tiny 8x8 Surface was enlarged so that each pixel looks like a square in a grid, and we added numbers for the X and Y axis, then a good representation of it could look something like this:



We can refer to a specific pixel by using a Cartesian Coordinate system. Each column of the X-axis and each row of the Y-axis will have an "address" that is an integer from 0 to 7 so that we can locate any pixel by specifying the X and Y axis integers.

For example, in the above 8x8 image, we can see that the pixels at the XY coordinates (4, 0), (2, 2), (0, 5), and (5, 6) have been painted black, the pixel at (2, 4) has been painted gray, while all the other pixels are painted white. XY coordinates are also called points. If you've taken a math class and learned about Cartesian Coordinates, you might notice that the Y-axis starts at 0 at the top and then increases going down, rather than increasing as it goes up. This is just how Cartesian Coordinates work in Pygame (and almost every programming language).

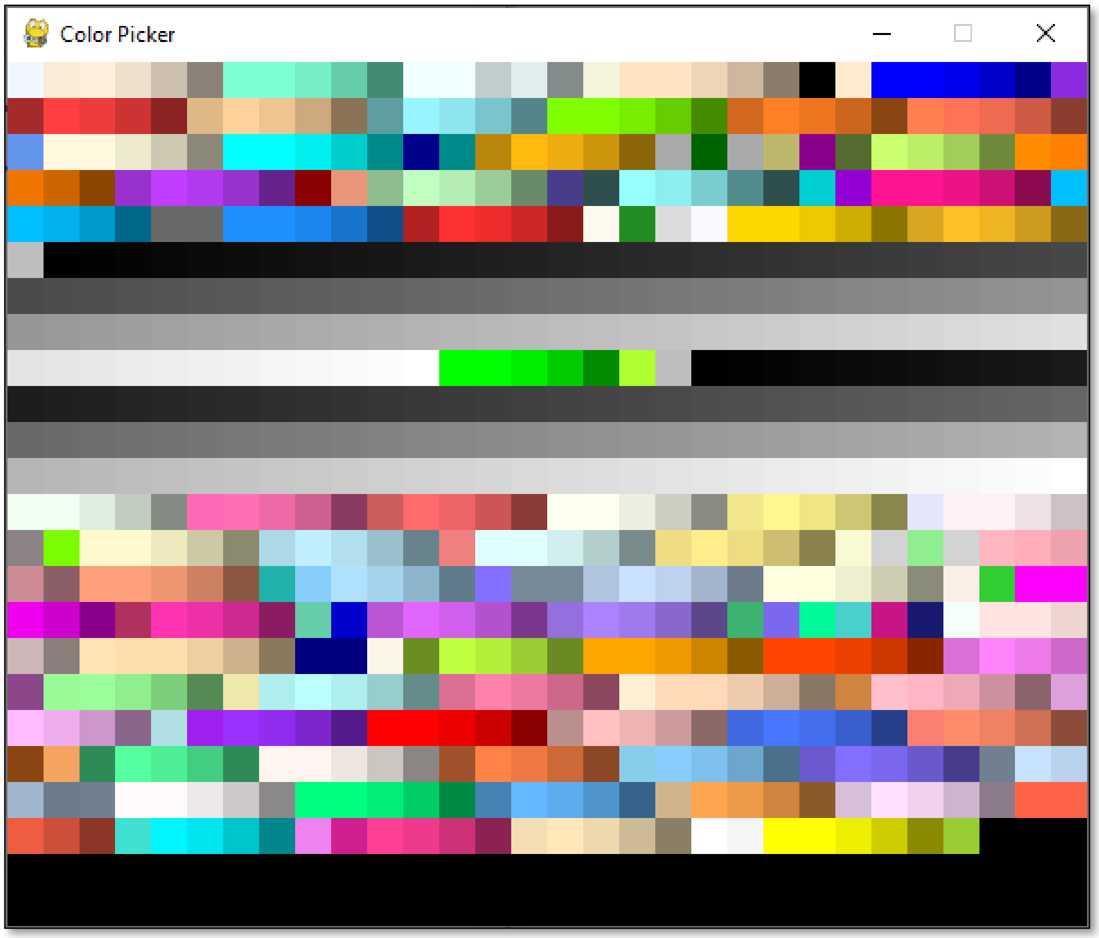
The Pygame framework often represents Cartesian Coordinates as a tuple of two integers, such as (4, 0) or (2, 2). The first integer is the X coordinate and the second is the Y coordinate.

Colors

Colors are a big part of any game development framework or engine.

Pygame uses the RGB system of colors. This stand for Red, Green and Blue respectively. These three colors combined (in varying ratios) are used to create all the colors you see on computers, or any device that has a screen.

Some of these colors are predefined in Pygame. See the **pygame_color_picker.py** program attached to this assignment.



There are three primary colors of light: red, green and blue. (Red, blue, and yellow are the primary colors for paints and pigments, but the computer monitor uses light, not paint.) By combining different amounts of these three colors you can form any other color.

In Pygame, we represent colors with tuples of three integers. The first value in the tuple is how much red is in the color. An integer value of 0 means there is no red in this color, and a value of 255 means there is the maximum amount of red in the color. The second value is for green and the third value is for blue. These tuples of three integers used to represent a color are often called RGB values.

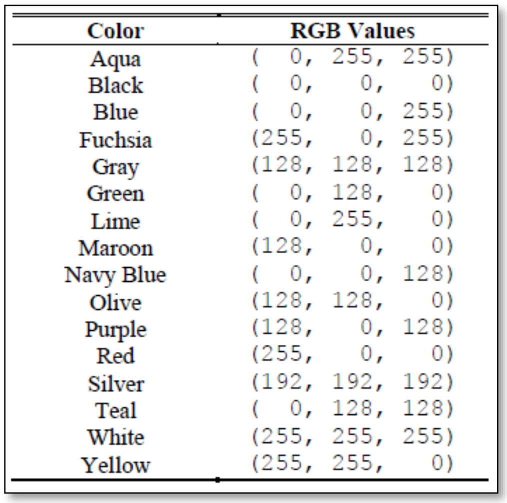
Because you can use any combination of 0 to 255 for each of the three primary colors, this means Pygame can draw 16,777,216 different colors (that is, 256 x 256 x 256 colors).

For example, we will create the tuple (0, 0, 0) and store it in a variable named BLACK. With no amount of red, green, or blue, the resulting color is completely black. The color black is the absence of any color. The tuple (255, 255, 255) for a maximum amount of red, green, and blue to result in white. The color white is the full combination of red, green, and blue.

The tuple (255, 0, 0) represents the maximum amount of red but no amount of green and blue, so the resulting color is red. Similarly, (0, 255, 0) is green and (0, 0, 255) is blue.

The values for each color range from 0 – 255, a total of 256 values. You can find the total number of possible color combinations by evaluating 256 x 256 x 256, which results in a value well over 16 million.

You can mix the amount of red, green, and blue to form other colors. Here are the RGB values for a few common colors:



Color	RGB Values
Aqua	(0, 255, 255)
Black	(0, 0, 0)
Blue	(0, 0, 255)
Fuchsia	(255, 0, 255)
Gray	(128, 128, 128)
Green	(0, 128, 0)
Lime	(0, 255, 0)
Maroon	(128, 0, 0)
Navy Blue	(0, 0, 128)
Olive	(128, 128, 0)
Purple	(128, 0, 128)
Red	(255, 0, 0)
Silver	(192, 192, 192)
Teal	(0, 128, 128)
White	(255, 255, 255)
Yellow	(255, 255, 0)

To use colors in Pygame, we first create Color objects using RGB values. RGB values must be in a tuple format, with three values, each corresponding to a respective color. Here are some examples below showing some color objects in pygame.

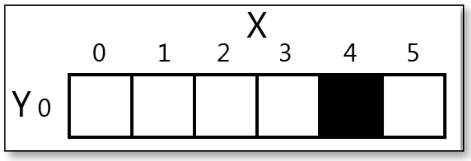
```
BLACK = pygame.Color(0, 0, 0)          # Black
WHITE = pygame.Color(255, 255, 255)    # White
GREY = pygame.Color(128, 128, 128)     # Grey
RED = pygame.Color(255, 0, 0)          # Red
```

The **fill(color)** method is used to fill in objects. For instance, assigning a rectangle the color green will only turn the borders green. If you use the **fill()** method and pass a green color object, the rectangle will become completely green.

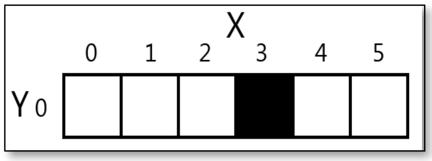
Animation

Now that we know how to get the Pygame framework to draw to the screen, let’s learn how to make animated pictures. A game with only still, unmoving images is dull. (Sales of the game “Look At This Rock” have been disappointing.)

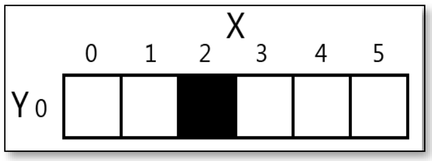
Animated images are the result of drawing an image on the screen, then a split second later drawing a slightly different image on the screen. Imagine the program's window was 6 pixels wide and 1 pixel tall, with all the pixels white except for a black pixel at 4, 0. It would look like this:



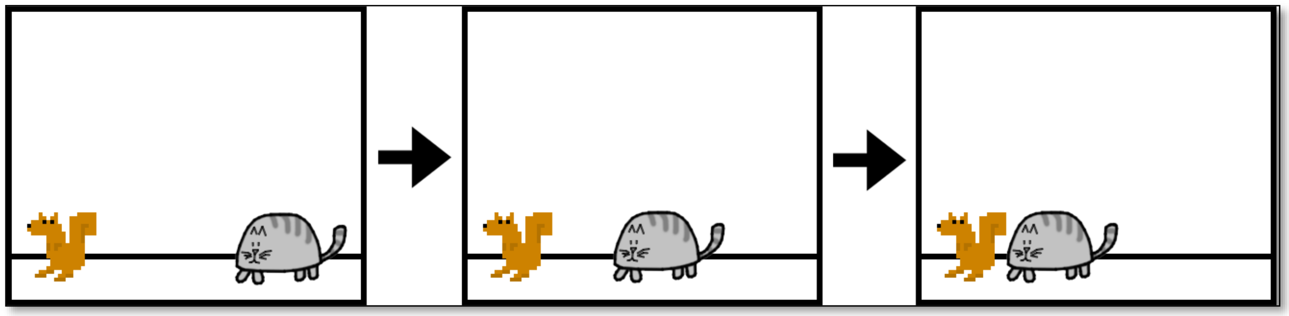
If you changed the image so that 3, 0 was black and 4, 0 was white, it would look like this:



To the user, it looks like the black pixel has "moved" over to the left. If you redrew the window to have the black pixel at 2, 0, it would continue to look like the black pixel is moving left:



It may look like the black pixel is moving, but this is just an illusion. To the computer, it is just showing three different images that each just happen to have one black pixel. Consider if the three following images were rapidly shown on the screen:



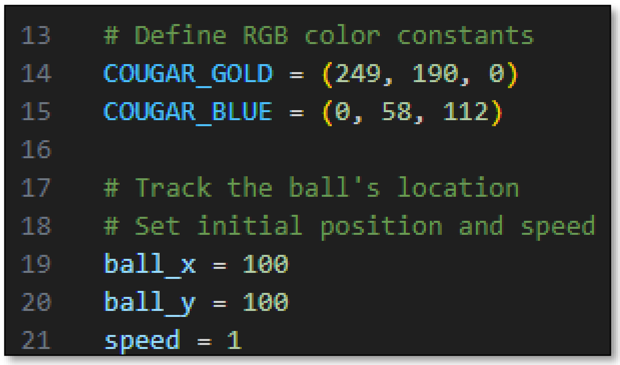
To the user, it would look like the cat is moving towards the squirrel. But to the computer, they're just a bunch of pixels. The trick to making believable looking animation is to have your program draw a picture to the window, wait a fraction of a second, and then draw a slightly different picture.

Tutorial 2: Ball Movement

Yes, we are finally going to make something move. By changing the (x, y) values each time through the game loop, we animate our ball.

Open **bouncing_ball_1.py** Save as **bouncing_ball_2.py**

Modify the existing code.



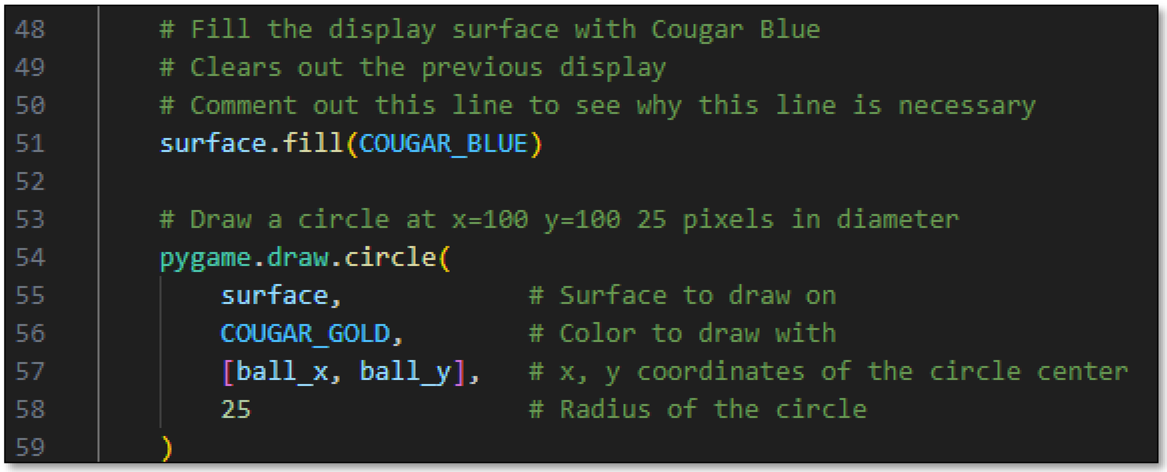
```python
# Define RGB color constants
COUGAR_GOLD = (249, 190, 0)
COUGAR_BLUE = (0, 58, 112)

# Track the ball's location
# Set initial position and speed
ball_x = 100
ball_y = 100
speed = 1
```

ball_x and ball_y:

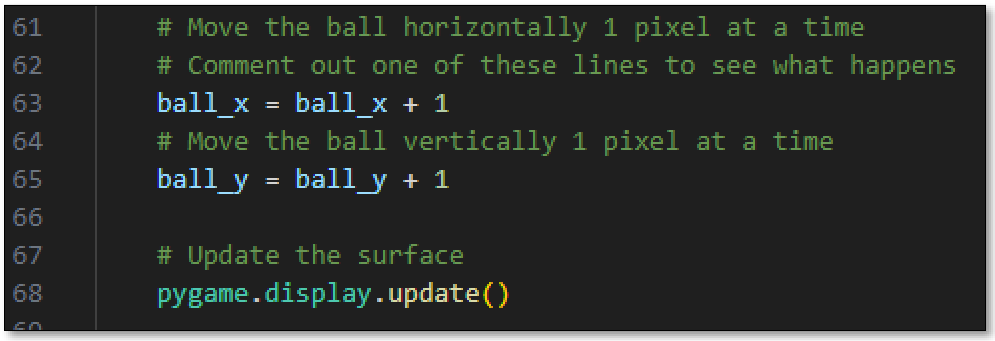
These variables represent the initial x and y coordinates of the ball on the game screen. In this case, the ball is positioned at (100, 100) on the coordinate plane. These values can be modified during the game to update the ball's position.

speed:

This variable represents the speed of the ball. The value is set to 1, indicating a relatively slow speed. You can adjust this value based on the desired movement speed of the ball in your game.

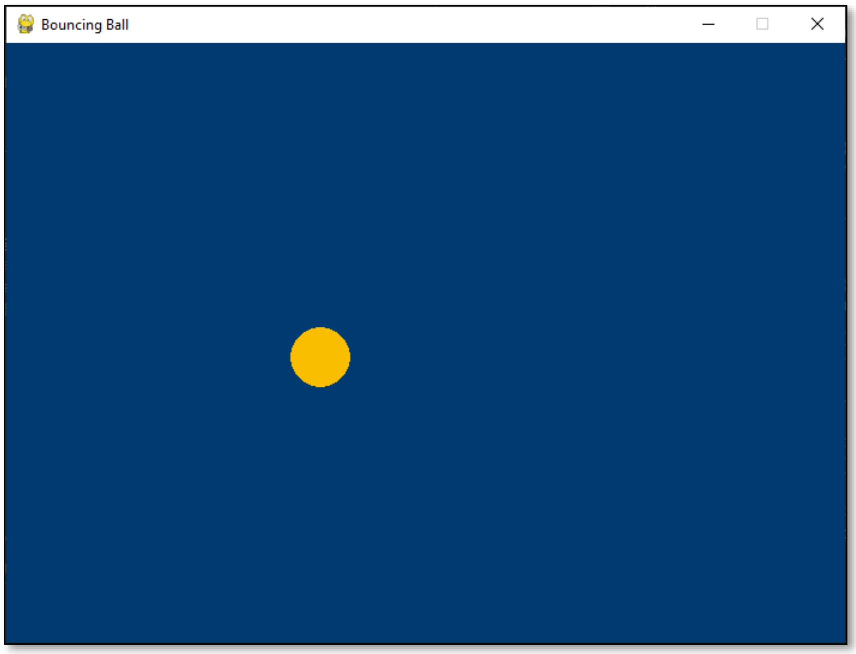


```python
        # Fill the display surface with Cougar Blue
        # Clears out the previous display
        # Comment out this line to see why this line is necessary
        surface.fill(COUGAR_BLUE)

        # Draw a circle at x=100 y=100 25 pixels in diameter
        pygame.draw.circle(
            surface,            # Surface to draw on
            COUGAR_GOLD,        # Color to draw with
            [ball_x, ball_y],   # x, y coordinates of the circle center
            25                  # Radius of the circle
        )
```



```python
        # Move the ball horizontally 1 pixel at a time
        # Comment out one of these lines to see what happens
        ball_x = ball_x + 1
        # Move the ball vertically 1 pixel at a time
        ball_y = ball_y + 1

        # Update the surface
        pygame.display.update()
```

Example run:



Yes, we have liftoff! You can't tell by the screenshot. Our ball is moving down and to the right 1 pixel a frame. Our Frames per Second is 60; the ball is moving 60 pixels a second.

Notice that we are using **surface.fill(COUGAR_GOLD)** to clear the screen each time through the game loop. Comment that line out to see why.

Tutorial 3: Bouncing Ball Class

We will modify the single file to incorporate OOP (Object Oriented Programming) concepts with separate class files.

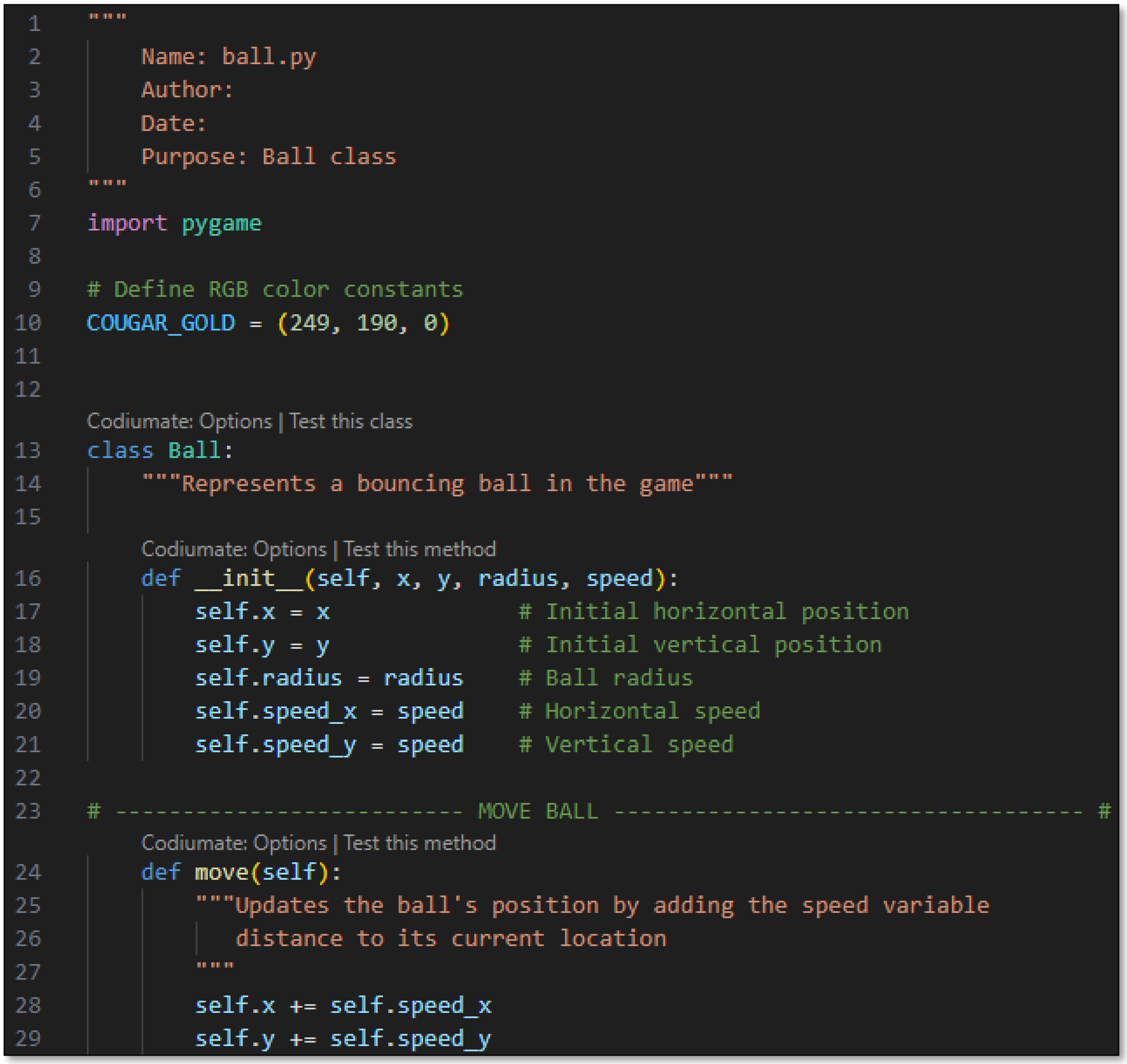
Two advantages of using OOP is to separate the game components into separate files and create multiple "sprites" from a single class. We can create hundreds of aliens from a single Alien class.

What is a sprite? A **sprite** is a two-dimensional image that is part of the larger graphical scene. Typically, a **sprite** will be an object of some sort **in** the scene that will be interacted with like a car, frog, or little plumber guy.

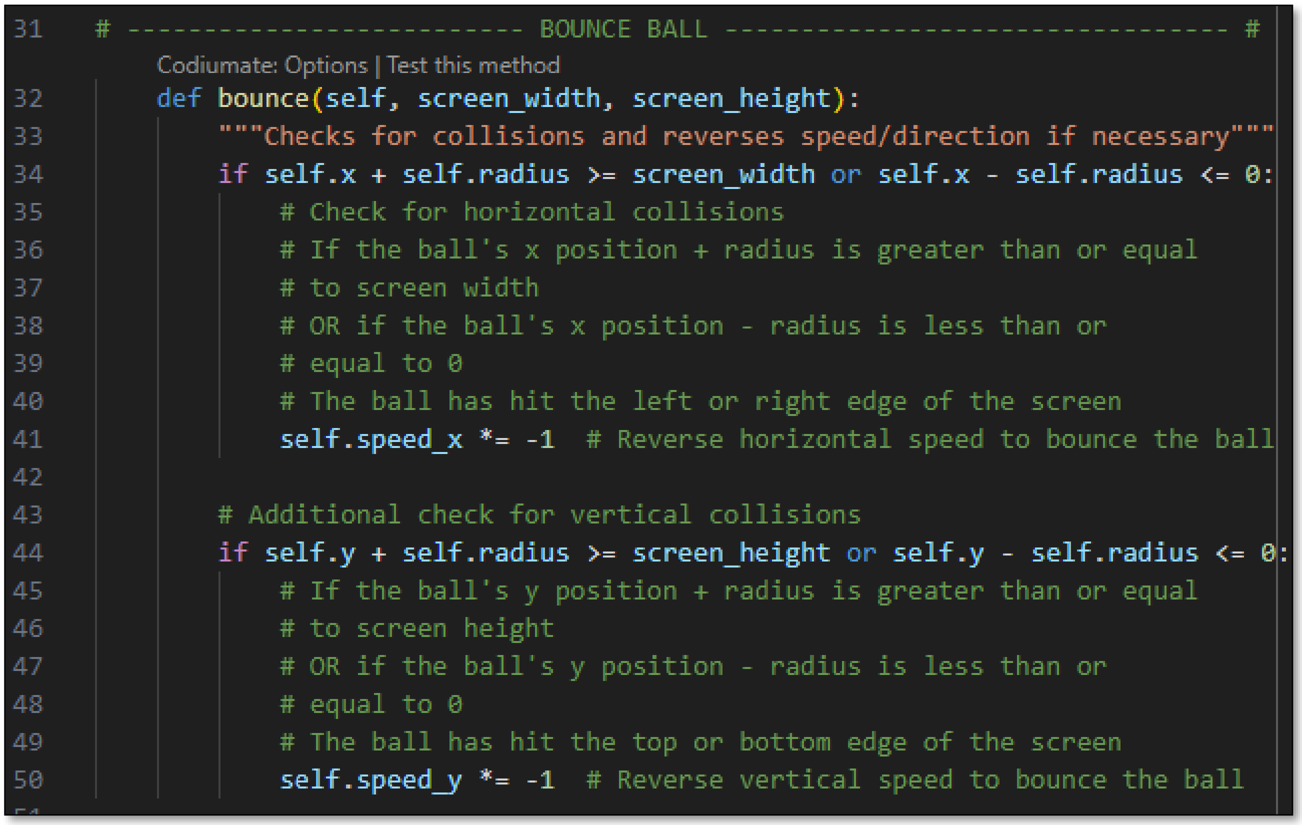
Create a file called **ball.py** with the following code. Some can be copied from **bouncing_ball.py**

Notice how all the game logic for the Ball is now in the Ball class.

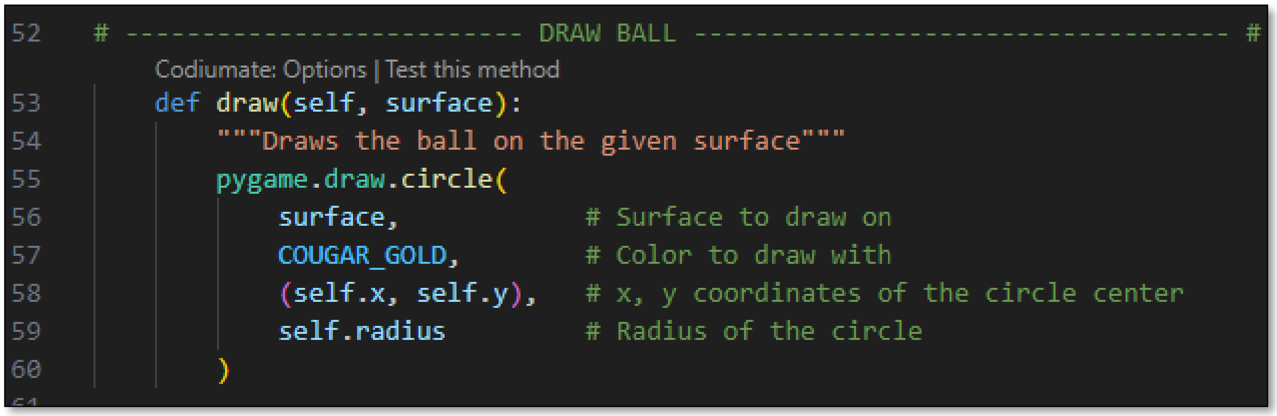
Create a file named **config.py** This file contains shared configurations for all classes.



```
"""
    Name: ball.py
    Author:
    Date:
    Purpose: Ball class
"""
import pygame

# Define RGB color constants
COUGAR_GOLD = (249, 190, 0)


Codiumate: Options | Test this class
class Ball:
    """Represents a bouncing ball in the game"""

    Codiumate: Options | Test this method
    def __init__(self, x, y, radius, speed):
        self.x = x              # Initial horizontal position
        self.y = y              # Initial vertical position
        self.radius = radius    # Ball radius
        self.speed_x = speed    # Horizontal speed
        self.speed_y = speed    # Vertical speed

# ------------------------- MOVE BALL ---------------------------------- #
    Codiumate: Options | Test this method
    def move(self):
        """Updates the ball's position by adding the speed variable
           distance to its current location
        """
        self.x += self.speed_x
        self.y += self.speed_y
```



```
# ------------------------- BOUNCE BALL --------------------------------- #
    Codiumate: Options | Test this method
    def bounce(self, screen_width, screen_height):
        """Checks for collisions and reverses speed/direction if necessary"""
        if self.x + self.radius >= screen_width or self.x - self.radius <= 0:
            # Check for horizontal collisions
            # If the ball's x position + radius is greater than or equal
            # to screen width
            # OR if the ball's x position - radius is less than or
            # equal to 0
            # The ball has hit the left or right edge of the screen
            self.speed_x *= -1  # Reverse horizontal speed to bounce the ball

        # Additional check for vertical collisions
        if self.y + self.radius >= screen_height or self.y - self.radius <= 0:
            # If the ball's y position + radius is greater than or equal
            # to screen height
            # OR if the ball's y position - radius is less than or
            # equal to 0
            # The ball has hit the top or bottom edge of the screen
            self.speed_y *= -1  # Reverse vertical speed to bounce the ball
```



```
# ------------------------- DRAW BALL ----------------------------------- #
    Codiumate: Options | Test this method
    def draw(self, surface):
        """Draws the ball on the given surface"""
        pygame.draw.circle(
            surface,            # Surface to draw on
            COUGAR_GOLD,        # Color to draw with
            (self.x, self.y),   # x, y coordinates of the circle center
            self.radius         # Radius of the circle
        )
```

Open **bouncing_ball_2.py Save as bouncing_ball_3.py** Change the code to the following:

Run the program to confirm that it works.

Assignment Submission

1. Attach all tutorials and assignments.
2. Attach screenshots showing the successful operation of each tutorial program.
3. Submit in Blackboard.