Lesson 38. Capstone project: game of tag

After reading lesson 38, you'll be able to

- Write a simple game by using the tkinter library
- Use classes and object-oriented programming to organize code for a GUI
- Write code that interacts with the user using the keyboard
- Use a canvas to draw shapes in your program

When you think of a program that uses a GUI, one of the most common kinds of programs that comes to mind are games. Games that are short and interactive offer quick distractions. They're even more fun to play when you write them yourself!

The problem

Write a GUI game using the tkinter library. The game simulates a game of tag. You should create two players inside a window. The players' position and size can be randomized at the start. Both players will use the same keyboard: one will use the W, A, S, D keys, and the other will use the I, J, K, L keys to move their piece. Users decide which one will try to catch the other. Then, they'll move around the window using their respective keys to try to touch the other player. When they touch the other player, the word *Tag* should appear somewhere on the screen.

This is a simple game, and the code to write it won't be long. When writing GUIs or visual applications such as games, it's important to not be too ambitious at the beginning. Start with a simpler problem and build upon it as you get things working.

38.1. IDENTIFYING THE PARTS TO THE PROBLEM

Given the problem, it's time to identify its parts. You'll find it easier to write the code if you do it incrementally. Ultimately, you need to accomplish three tasks:

- Create two shapes
- Move them around the window when certain keys are pressed
- Detect whether the two shapes have touched

Each of these can be written as a separate piece of code that can be tested separately.

38.2. CREATING TWO SHAPES IN A WINDOW

As with the other GUIs you've seen, the first step is to create a window and add any widgets your game will use to it. The next listing shows the code for this. The window will hold only one widget, a canvas. The canvas widget is a rectangular area in which you can place shapes and other graphical objects.

Listing 38.1. Initializing a window and widgets

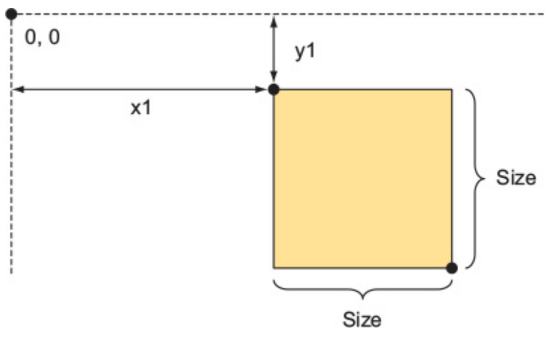
```
import tkinter
window = tkinter.Tk()
window.geometry("800x800")
window.title("Tag!")
canvas = tkinter.Canvas(window)                    1
canvas.pack(expand=1, fill='both')                 2
```

- *1* **Canvas widget that will contain the player shapes**
- *2* **Adding the canvas so that it fills entire window and scales if the window is resized**

not the window itself. Because you'll be creating more than one player, it's a good idea to think in a modular way. You'll create a class for the player, which will initialize the player piece on the canvas.

To make the game more interesting, you can use random numbers to set up the starting position and size of the shape. Figure 38.1 shows how you'll construct the rectangle. You'll choose a random coordinate for the top-left corner, `x1` and `y1`. Then, you'll choose a random number for the size of the rectangle. The `x2` and `y2` coordinates are calculated by adding the size to `x1` and `y1`.

Figure 38.1. Constructing the rectangle playing piece by choosing a coordinate for the top-left corner and a random number for the size



The random aspect of creating the rectangle means that every time you create a new player, the rectangle will be placed at a random position in the window and will be a random size.

Listing 38.2 shows the code for creating the player's rectangle piece. The code creates a class named `Player`. You'll use a rectangle to denote the player's piece. Any object on a canvas is denoted by a tuple of four integers, `x1`, `y1`, `x2`, `y2`, where (`x1`, `y1`) is the top-left corner, and (`x2`, `y2`) is the bottom-right corner of the shape.

Listing 38.2. A class for the player

```
import random
class Player(object):
    def __init__(self, canvas, color):
        self.color = color
        size = random.randint(1,100)
        x1 = random.randint(100,700)
        y1 = random.randint(100,700)
        x2 = x1+size
        y2 = y1+size
        self.coords = [x1, y1, x2, y2]
        self.piece = canvas.create_rectangle(self.coo
        canvas.itemconfig(self.piece, fill=color)
```

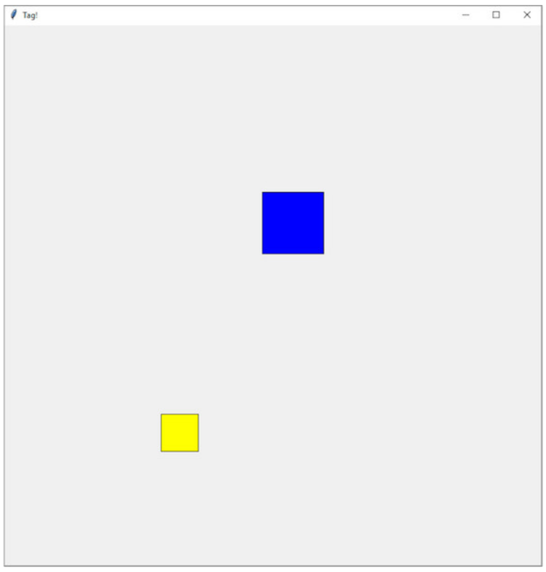
- ***1* init method to create an object that takes in the canvas on which to add the shape, and the shape's color**
- ***2* Sets the color of the object as a data attribute**
- ***3* Chooses a random number between 1 and 100 for the size of the player piece**
- ***4* The x-value of the top-left corner of the object, selected randomly between the range specified**
- ***5* The y-value of the top-left corner of the object, selected randomly between the range specified**
- ***6* The x-value of the bottom-right coordinate of the object**
- ***7* The y-value of the bottom-right coordinate of the object**
- ***8* Sets the coordinates of the object as a data attribute, with type list**
- ***9* Sets the player piece data attribute as a rectangle, placed at the position given by the coordinates above**
- ***10* Sets the color of the player piece, referencing it by its variable name, self.piece, from the preceding line**

After creating the window, you can add the players to the canvas with the following code, which creates two `Player` objects on the same canvas, one yellow and one blue:

```
player1 = Player(canvas, "yellow")
player2 = Player(canvas, "blue")
```

If you run the code, you'll get a window that looks something like figure 38.2. You have two shapes of different colors at a random position and with a random size. Nothing happens when the mouse is clicked or when a key is

Figure 38.2. The game after creating two player objects. The position and size of each square varies each time the program is run.



38.3. MOVING SHAPES INSIDE THE CANVAS

Each shape responds to the same type of event, keypresses:

- To move the shape up, press W for one shape and I for the other.
- To move the shape left, press A for one shape and J for the other.
- To move the shape down, press S for one shape and K for the other.
- To move the shape right, press D for one shape and L for the other.

You'll have to create a function that acts as the event handler for any key-press on the canvas. Inside the function, you'll move one player or the other, depending on which button is pressed. The following listing shows the code. In this code, `move` is a method you'll define in the `Player` class. It'll move the player's position with `"u"` for up, `"d"` for down, `"r"` for right, and `"l"` for left.

Listing 38.3. Event handler function when any key is pressed on the canvas

```
def handle_key(event):                                  1
    if event.char == 'w' :                              2
        player1.move("u")                               3
    if event.char == 's' :
        player1.move("d")
    if event.char == 'a' :
        player1.move("l")
    if event.char == 'd' :
        player1.move("r")
    if event.char == 'i' :                              4
        player2.move("u")                               5
    if event.char == 'k' :
        player2.move("d")
    if event.char == 'j' :
        player2.move("l")
    if event.char == 'l' :
        player2.move("r")

window = tkinter.Tk()
window.geometry("800x800")
window.title("Tag!")
canvas = tkinter.Canvas(window)
canvas.pack(expand=1, fill='both')

player1 = Player(canvas, "yellow")
player2 = Player(canvas, "blue")
canvas.bind_all('<Key>', handle_key)                     6
```

- ***1* Event handler function**
- ***2* Checks whether the keypress from the event is a W**
- ***3* move is a method that you'll define in the player class, so call that method to move the shape up.**
- ***4* Checks whether the keypress from the event is an I**
- ***5* Because move is defined in the player class, you call it on the other player to move the shape up.**
- ***6* For the canvas, any keypress event will call the handle_key function.**

Notice how nicely modular this code is. It's easy to understand what's going player to move and in which direction, denoted by "u" for up, "d" for down, "l" for left, and "r" for right.

Inside the Player class, you can write code that handles moving the shape by changing the values of the coordinates. Listing 38.4 shows the code. For each of the directions that the player can move, you modify the coordinate data attribute. Then, you update the canvas coordinates for the shape to the new coordinates. Two players can't move simultaneously. After a player starts moving, it'll stop moving as soon as the other player presses a key.

Listing 38.4. How to move a shape inside the canvas

```
class Player(object):
    def __init__(self, canvas, color):
        size = random.randint(1,100)
        x1 = random.randint(100,700)
        y1 = random.randint(100,700)
        x2 = x1+size
        y2 = y1+size
        self.color = color
        self.coords = [x1, y1, x2, y2]
        self.piece = canvas.create_rectangle(self.coo
        canvas.itemconfig(self.piece, fill=color)

    def move(self, direction):
        if direction == 'u':
            self.coords[1] -= 10
            self.coords[3] -= 10
            canvas.coords(self.piece, self.coords)

        if direction == 'd':
            self.coords[1] += 10
            self.coords[3] += 10
            canvas.coords(self.piece, self.coords)
        if direction == 'l':
            self.coords[0] -= 10
            self.coords[2] -= 10
            canvas.coords(self.piece, self.coords)
        if direction == 'r':
            self.coords[0] += 10
            self.coords[2] += 10
            canvas.coords(self.piece, self.coords)
```

- *1* **Method to move the shape; takes in a direction: one of 'u', 'd', 'l', 'r'**
- *2* **Does something different for each of the four possible inputs ('u', 'd', 'l', 'r')**
- *3* **If you're moving up, changes the y1 and y2 values by decreasing them by indexing into the list coords**
- *4* **Changes the coordinates of the rectangle, denoted by self.piece, to the new coordinates**

This code is used by any player object created. It follows the abstraction and modularity principles because it's under the Player class, which means you have to write it only once but it can be reused by any of the objects.

Now when you run the program, the set of keys W, A, S, D will move the yellow shape around the window, and the keys I, J, K, L will move the blue shape. You can even ask someone to play with you to test the code. You'll notice that if you hold a key down, the shape will move continuously, but as soon as you press another key, the shape will stop and move according to the other keypress (until another key is pressed). Chasing the shapes around is fun, but nothing happens when they touch.

38.4. DETECTING A COLLISION BETWEEN SHAPES

The last piece to the game is to add the code logic for detecting whether two shapes collide. After all, this is a game of tag, and it'd be nice to be notified when one shape has touched the other one. The code logic will consist of making two method calls on the canvas. It'll be implemented inside the same event function that deals with keypresses in the canvas. This is because after every keypress, you'd like to see whether a collision has occurred.

Listing 38.5 shows the code for detecting the collision between two shapes. By design, in the tkinter library, every shape added to the canvas is assigned an ID. The first shape added gets an ID of 1, the second of 2, and so on. The first shape you added to the canvas was the yellow one. The idea behind the code is that you'll get the coordinates of the first shape in the canvas by calling the method bbox, which finds the bounding box around the shape. In the case of the rectangle, the bounding box is the rectangle itself, but in other cases, the bounding box is a rectangle that the shape barely fits in. Then, you call the find_overlapping method on the canvas, with the coordinates of the bounding box as a parameter. The method returns a tuple that tells you all IDs that are within that box. Because the coordinates given as a parameter are those of the bounding box for one shape, the method will give do is to check whether the shape with ID of 2 is in the returned tuple. If it is, then add text to the canvas.

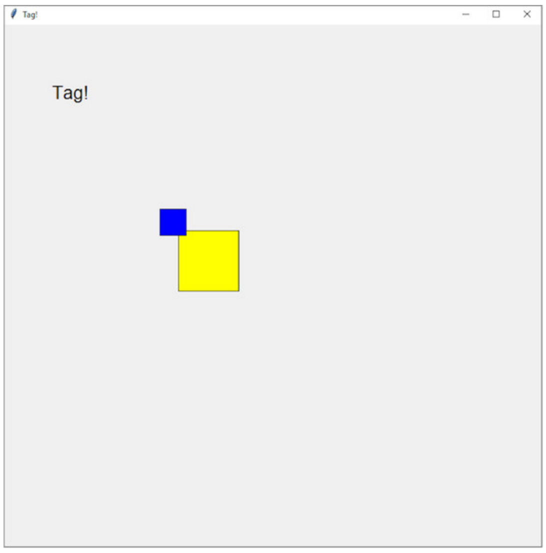
Listing 38.5. Detecting a collision

```
def handle_key(event):
    yellow_xy = canvas.bbox(1)
    overlapping = canvas.find_overlapping(
                   yellow_xy[0],yellow_xy[1],yellow_x
    if 2 in overlapping:
        canvas.create_text(100,100,font=("Arial",20),
```

- *1* **Gets coordinates around one of the shapes**
- *2* **Finds IDs of all shapes that are within the box formed by those coordinates**
- *3* **Checks whether the ID of the other shape is in the overlapping IDs**
- *4* **Adds text to the canvas**

As soon as one shape touches the other one, the screen will look something like figure 38.3.

Figure 38.3. When one shape overlaps with the bounding box of the other shape, the text *Tag!* is printed on the canvas.



38.5. POSSIBLE EXTENSIONS

Many possibilities for extensions exist with this game. Your coding in this lesson is a great start. Here are some ideas:

- Instead of closing the window and restarting it to play again, add a button to ask the user to play again. When they do, choose another random position and size for your shapes.
- Allow the shape to escape. If the shapes aren't touching after having touched once, remove the text from the canvas.
- Allow the players to customize their shapes by changing the color or changing the shape to a circle.

SUMMARY

In this lesson, my goal was to teach you how to use more advanced GUI elements to make a game. You used a canvas to add shapes to the GUI, and you added an event handler that moved shapes around depending on which key was pressed. You also saw how to detect collisions between shapes in a canvas. You saw how to write neat, organized, and easy-to-read code by using classes and functions for major parts of the code that you know will be reused.

PREV
Lesson 37. A library for graphical user interfaces

NEXT
Appendix A. Answers to lesson exercises