Computer Science 1 — CSci 1100
Lab 11 — Drawing, Classes, Testing
Fall Semester 2017

Lab Overview

So far, the outputs from our programs have been text files and, occasionally, images. In this lab, we will draw on a graphical display using a package called `Tk` (through the module `Tkinter`). This should give you an initial feel for graphical interface programming. Along the way we will practice a bit more with writing and testing classes. This material is covered in Lecture 22 and Chapter 16 of the Python book, but all the specific information necessary to complete the lab is provided here.

It is highly recommended that you review the Lecture notes on `Tkinter` before you start this lab:

`http://www.cs.rpi.edu/academics/courses/fall17/csci1100/lecture_notes/lec22_tkinter.html`

While we explain mechanics of the program here, we will not go into details of Tkinter.

You will be asked to work with an infinite loop in this lab, which is actually how you animate objects: keep drawing them until the program is terminated.

Checkpoint 0 – Get the ball moving...

Before we start with the real work of this lab, you need to first get familiar with the methods we are going to use. `Tkinter` allows you to create animations as well as create graphical user interfaces. We will work with two classes from `Tkinter` called `Canvas` that allows you to draw objects in a window, and `Button` that allows a user to interact with the program.

Please download `lab11files.zip` and unzip it. Open `check0.py`, take a quick look, and then run it. You will see a single ball object move across the screen until it reaches the right edge and then stop. The program ends when you close the window.

To begin to understand what is going on, let's take a closer look at `check0.py`. We start by initializing a canvas and four buttons, all are under `root` window. There are different frames enclosing these elements.

Read all the way through `check0.py`, using the comments to guide your understanding. Explore the code by making the following changes. For each change, rerun the code, and then restore the value you changed:

- Change the color to green.
- Change the starting location and the dx,dy values
- Change the wait time by clicking the slower and faster buttons
- Comment out the code that deletes all previous drawings.

The last line of the program

```
root.mainloop()
```

means that the `root` window listens continuously to any commands (events) that are sent to it. We have added code to listen for the four buttons, "Restart"', "Slower", "Faster" and "Quit" .

Once you have completed your experiments, please move onto Checkpoint 1, where the graded activity of the lab starts.

One small issue that we have noticed, is that the current version of Wing does not seem to interact well with tkinter. If you have problems running your program a second time, please restart the Python shell.

Checkpoint 1 — Ball class

Notice that the information about the ball object is stored in several separate variables of the class `BallDraw`. This is a good indication that a class is needed to gather and encapsulate this information.

For Checkpoint 1, define a new Ball class in a new file. This class will encapsulate all the relevant functionality of a `Ball` object. We will tell you exactly what this is below:

A `Ball` object stores the position of the object (x,y), dx and dy values, radius, and color as attributes. The `Ball` class must have the following methods:

- `__init__` to initialize a Ball object that takes as input x,y coordinates, dx,dy values, radius and the color,
- `position()` returns a 2-tuple containing the x and y positions of the center of the ball,
- `move()` changes the current location of the ball by adding the dx and dy offsets to the ball. It does not return anything,
- `bounding_box()` returns a box containing the ball in the form of a 4-tuple, (x0, y0, x1, y1), completely containing the Ball,
- `get_color()` returns the ball's color, and
- `some_inside(maxx,maxy)` returns `True` if any part of the Ball object is inside the window with coordinates `(0,0,maxx,maxy)` and `False` otherwise (Hint. the logic for this is already given in the while loop of `check0.py`.)

Place the class in a separate module called `Ball.py`.

Test your Ball class by running `test_Ball.py`, a Python program that uses a module called **doctest** to test code. Notice that `test_Ball.py` imports `Ball`. The code in `test_Ball.py` has been written to run and check methods from the Ball class.

Make some changes to `test_Ball.py` to see what happens when the tests and the output of code do not match. Finally, observe that the code in `test_Ball.py` will only run correctly if you have built the `Ball` class methods parameter lists and returns correctly. If your class passes all the tests, you are ready for the animation part of the work.

Copy `check0.py` to `check1.py`. Import the new class `Ball` into this. Now, change the animation code such that at `__init__`, it creates a ball object with the relevant information. At any point where you need to change the `Ball` object or access one of its attributes, change your code to use your `Ball` object and its methods. All of the relevant code that you must change will involve variables that start with `ball_`. For example, instead of the long condition in the while loop for the `animate` function, you can simply call the `some_inside` method of the ball.

To complete Checkpoint 1: Show your class implementation, doctest test execution results (including the introduction of some errors), and the working version of `check1.py` to a TA or mentor.

Checkpoint 2 — Bouncing off the Walls

Now things will get interesting. Instead of your ball disappearing from one end of the canvas, what if the ball just bounced when it hit the edge of the canvas and started moving in a different direction? In particular if it hits the left (`x==0`) or right (`x==maxx`) wall, it should negate its dx value, while if it hits the top (`y==0`) or bottom (`y==maxy`) wall, it should negate its dy value. Be sure to account for the ball radius when doing your calculations. The ball should bounce when the edge of the ball contacts the wall. This creates the effect of a bouncing ball with no friction and no spin. Accomplish this in a new method of the `Ball` class called `check_and_reverse` which should have two arguments: `maxx` and `maxy`. Copy `check1.py` into `check2.py`. In the new file, you will need to call this function from the main code each time you move the ball.

Add text in `test_Ball.py` to test `check_and_reverse`.

To complete Checkpoint 2: Show your class implementation, your new test cases and doctest execution, and the working program to the TA or a mentor.

Checkpoint 3 — Multiple Balls

We finish the lab with a fun exercise. First, copy your `check2.py` file to `check3.py`. Now, using the `random` module create 10 balls with random initial locations, random dx and dy values, random radii, and random colors. To do so, start by including the `random` module in your code. Then, for random integers, use the function:

```
random.randint(min,max)
```

All initial positions should be within the canvas (10,390). Each dx and dy value should be in the range -8 to 8. Each radius should be between 5 and 10. The colors should come from

```
colorList = ["blue", "red", "green", "yellow", "magenta", "orange"]
```

and you can use the function

```
random.choice(colorList)
```

to pick a random color from this list.

Move each ball by its dx and dy values in each step. Reorganize the code for animation by only checking that the animation has not stopped (removing boundary checks). Then merge the animation and drawing functionality to simplify the program flow. The code should first remove all objects from canvas, draw the ball objects in their new location, move the balls, and then once all the balls have been updated, update the canvas. At this point you will also need to give the restart button some attention by making it reset every ball to its original (x, y) position and velocity.

You can also see some new effects by changing the frequency with which you execute `self.canvas.delete("all")`. Instead of doing it at every step of the animation, do it in every 10th or 100th step. See how that changes the look of the animation. You can explore further by

changing the code so that when the bouncing balls collide they create a physically realistic elastic collision (momentum conserved).

To complete Checkpoint 3: Show your working program to a TA or mentor.