CS 120 (Fall 21): Introduction to Computer Programming II

Long Project #7

Three Shapes

due at 5pm, Tue 12 Oct 2021

WARNING: The `graphics.py` that I use has a few more features than the one that Ben Dicken uses. If you still have `graphics.py` left over from 110, please download my version to get the latest features.

1 Creativity

This is a **creative project!** This means that you won't be auto-graded; instead, your TA will look at your code manually (and run your programs), and assign you a grade based on that.

Please note: you are going to be graded on **engagement** with the project, not on artistry! While I'm sure that some students will do amazing, colorful projects, that's not required - you can get full credit from a very simple design - provided that you address the problems required of you in the spec.

1.1 Limitation

The purpose of this project is to give you some experience with using a library - and especially with using "callbacks," which are methods that **you** define, but the library calls.

Therefore, both of your programs **must** be built around the `Game` class that I provide. Don't try to write your own graphics program from scratch - **use the library!**

2 What You Must Turn In

You will write two programs. One of them, named `three_shapes.py`, will (approximately!) re-create a demo program that I will show you in the video for this project. The other will be a creative program, of your own devising.

Each of the two programs you turn in must have their own `main()` function; see just below for an example of a good `main()` function for your files. Other details on the requirements for both programs are in the spec below.

You must turn in:

- `three_shapes.py`
- Your other program
- A README file (it must be a text file or PDF) describing your other program: how to run it, what it does, etc.

3 Overview

The purpose of this project is to give you practice **writing classes.** I needed some sort of environment where you could write a few methods which would affect how your class worked, and what I came up with was the idea of small 2D games.

In this project, I have provided the framework for a small 2D space, represented by a window on the screen; you will write classes that represent objects which move around in that space. All of the classes will be required to implement the same methods, but **you** will decide what the methods do! When you run your program, you will watch these objects interact on the screen. (Although I call this a "game," it's probably better called a "simulation" - I assume that most student projects will have **zero** interaction with the user. Instead, the objects will interact with each other.)

I have provided a file (`three_shapes_game.py`) which implements a `Game` class for you; the `Game` class creates the window, and then keeps track of the objects you've created; it also calls the various methods of your objects from time to time.[1]

In addition, I've provided the "game loop." This is the `main()` function which you can use in your own program - it creates the `Game` object, and then goes into an infinite loop, updating the objects, and re-drawing the screen. **You are not required to use this code,** but if you want to use this as your own `main()` function (or edit it for your own purposes), that's perfectly fine!

(spec continues on next page)

[1] Normally, when I give you a file, I forbid you to modify it. But in this case, since we're not testing you against an autograder, you are **welcome** to tweak this file if necessary. Two rules:

- Keep the basic architecture (the `Game` calls methods on your objects from time to time)
- Submit the updated file, so that your TA can run the code!

```
def main():
    # This is the size of the window; feel free to tweak it.  However,
    # please don't make this gigantic (about 800x800 should be max),
    # since your TA may not have a screen with crazy-large resolution.
    wid = 400
    hei = 600

    # This creates the Game object.  The first param is the window name;
    # the second is the framerate you want (20 frames per second, in this
    # example); the last two are the window / game space size.
    game = Game("Three Shapes", 20, wid,hei)

    # This affects how the distance calculation in the "nearby" calls
    # works; the default is to measure center-to-center.  But if anybody
    # wants to measure edge-to-edge, they can turn on this feature.
        # game.config_set("account_for_radii_in_dist", True)

    # You get to decide what spawn() does.  This sets up the initial
    # objects that you want to create, at the beginning of the game (if
    # any).  Of course, you can remove this is you want to create objects
    # some other way.
    spawn(game, wid,hei)

    # game loop.  Runs forever, unless the game ends.
    while not game.is_over():
        game.do_nearby_calls()
        game.do_move_calls()
        game.do_edge_calls()
        game.draw()

        # You get to decide what spawn_more() does (if anything).  This
        # allows you to spawn additional objects over time.  It is
        # called once per game tick.
        spawn_more(game, wid,hei)
```

Spec continues on next page.

4 The `Game` Class

The `Game` class has a number of methods, most of which you don't need to know anything about; the game loop that I've given you above will call them at the proper times. However, there are two critical methods that you must know about.

4.1 `Game.add_obj(obj)`

The `add_obj()` method is used to add a new object to the game. The object that you pass it should be one of the objects that you've created; it can be any class (and many games will include mixes, with objects from several classes), but it must implement all of the methods that we define below. You can call this method at any time.

Example:

```
# you get to decide what parameters to pass to your classes
new_obj = MyClass(parameterA, parameterB)
game.add_obj(new_obj)
```

NOTE: It's illegal to add the same object to the game twice.

4.2 `Game.remove_obj(obj)`

The `remove_obj()` method is used to remove an object from the game. However, it doesn't remove it from the game instantly, as it turns out that it's kind of annoying to "remove" an object that might already be queued up to be handled later, as part of a loop. Therefore, objects will be removed later, all at once, when the game loop calls `Game.draw()` .

Therefore, the following rules apply:

- `remove_obj()` should only be called during a `nearby()` or `move()` method.
- It is illegal to try to remove an object which isn't part of the current game.
- However, it is **legal** to call `remove_obj()` multiple times on the **same** object - so long as you do it all in the same tick.

Example:

```
class Example:
    def move(self, game):
        if ... some check to decide when to remove an object ... :
            game.remove_obj(self)
```

5 Your Classes

Each program that you will write will declare at least one class (sometimes several), to represent the objects that are moving around in the game. You get to design **all of the internals** of your class - it's entirely up to you! However, you will be required to implement a few methods, which the `Game` class will be calling. (If you don't define one of these, or if you use the wrong number of parameters, the game will crash when `Game` tries to call them.)

You are also allowed to add **additional** functions if you wish; however, you **must** implement (at least) the required methods.

5.1 Dummy Class

Looking for an example of what the class should look like? Take a look at the file `dummy_class.py`, which I've included in the zipfile for this project.

5.2 Private Variables

All of your variables must be private; none of the variables may be public.

Remember, however, that it is permissible for your code to access the private variables of your own class. So if you declare class `Foo`, with private variable `_asdf` inside of it, is permissible for your function to access **both** `self._asdf` **and** the `_asdf` fields inside other `Foo` objects.

5.3 Constructor

Every one of your classes will need a constructor. But **you** will be creating the objects, not me - so you have complete flexibility about how many parameters the constructor has, and what they mean.

5.4 Required Getters: `get_xy(self)`, `get_radius(self)`

WHY DOES THIS EXIST?
To allow the `Game` object to know where your object is, and how large it is.

Every object must have a position in the 2D space of the game. (We generally assume that your object won't go outside the bounds of the window, but I don't enforce this requirement.) Every object also has a size, which is reported as a "radius." If your object is not a circle, that's OK - give some sort of value that **approximately** represents the size.

The method `get_xy()` must return a 2-element tuple, which are numeric values (integers or floats), which give the position of the object.

The method `get_radius()` must return a numeric value which represents the size of the object (or at least, a rough approximation of that). You can define what this size means - the game doesn't really care. If your objects are all circles, then it's easy to make this their radius. If your objects are squares,

then this might be half the width. If they are points, the radius might be hard-coded to zero. It's up to you!

NOTE: You can define whatever units you find handy. For my example projects, I just made the (x,y) coordinates line up with the drawing coordinates - so (0,0) was the upper left, and a change of +1 would move the object by one pixel. That's easy! But if you want to use some other system - such as having a grid, where +1 means "move over one tile," the game won't care. Have fun!

5.5 `nearby(self, other, dist, game)`

WHY DOES THIS EXIST?
To allow you to have interactions between objects. While it's especially useful for objects that are close to each other, it can be used to define interactions at any distance.

Every tick, the `Game` will call every object's `nearby()` method multiple times. It will pass three parameters:

- A reference to another object
- The distance between the two
- A reference to the `Game` object (in case you want to add or remove objects)

You can decide what this does. You might even make it a NOP! But generally, this is the call that will let you define interactions between the various objects - when they get close to each other, you can decide to do something. (Or, if you want to do a physics simulation, such as magnetism or gravity, you might use this call to update all of the forces applied to an object.)

5.5.1 Sorting the Calls

The `Game` will call you multiple times - once for every "other" object, in the entire game. (That is, if there are 100 objects, then each one gets 99 `nearby()` calls; there will be 9900 of them, altogether.) When the `Game` calls all of the `nearby()` methods for some object, it will sort things by the distance to the *nearest* object, so you will get calls about very-close objects first, and distant objects later.

Your `nearby()` method should return a boolean. If it returns `True`, then you will keep getting `nearby()` calls; if it returns `False`, that object will not get any more `nearby()` calls until the next tick.

What does this mean for you? It doesn't have to mean anything! If you want to make your program simple, just return `True` at all times, and you will get all of the calls. But if you are doing something where you only care about the **close** objects (like my Bouncing Balls example), then you can return `False` when the calls are "getting far away."'

5.6 edge(self, dir, position)

WHY DOES THIS EXIST?
So that you can make objects bounce off the walls, or at least not go beyond them.

Every tick, the `Game` will check to see if an object is at (or beyond) the edge of the board. It will call the `edge()` method on the object if so. The first parameter indicates what edge you are close to; it is a string, and will be one of `"top", "left", "bottom", "right"` . The second is the position of the edge; for instance, for `"right"`, this will be equal to the width of the board, and `"top"` will be 0.

Note that, if you used funny coordinates (see `get_xy()` above), this mechanism will not work properly. It only works if your (x,y) position uses the same coordinates as we do for drawing on the screen.

5.7 move(self, game)

WHY DOES THIS EXIST?
So that you can update the position of an object - after all of the `nearby()` calls have already happened. In some games, you will use the `nearby()` and `edge()` calls to change the velocity of an object (bouncing off things), and then `move()` will actually update the (x,y) position.

Every tick, the `Game` will call this once on each object. Use it to update the position of your object.

If you have a class where all of the objects never move, then this can be a NOP. However, most classes will move the object around in some fashion inside of this method.

5.8 draw(self, win)

Every tick, the `Game` will call this once on each object. The object will draw itself onto the window.

The `win` parameter is a `graphics` object, based on the same library as Ben Dicken uses in 110. So you can use the same drawing methods, like this:

```
x,y = ... my xy position on the screen ...
rad = ... my radius ...

color = ... color string. ...
        ... Either a standard color, like "black", ...
        ... or an HTML hex color ...

win.ellipse(x,y,
            rad*2, rad*2,      # width and height
            color)
```

6 Requirements for `three_shapes.py`

Watch the intro video for this project. You must do something **similar** to the Three Shapes demo that I showed you. Note that you are **NOT** required to replicate it exactly!

Your Three Shapes program should:

- Have three different object classes: `Circle, Square, Triangle`
- Periodically spawn new objects, at random locations and with random velocities.
- Each of the three objects must have different rules for how they move. **Feel free to come up with your own rules** - in my demo, the circles fall under the influence of gravity, the squares move in straight lines unless they bounce off a wall, and the triangles move randomly (meaning that, most of the time, they just "quiver.").
- When objects get near each other, they should disappear based on the following rules:

  - If a circle touches a square, the circle is removed

  - If a square touches a triangle, the square is removed

  - If a triangle touches a circle, the triangle is removed

- Each class must have two or more randomly-chosen properties. For my demo, I assigned every object a random size, and a random RGB color. If you want to do the same, that's OK - or, you can make the objects vary in some other way.
- When you draw the objects on the screen, their type, as well as their randomly-chosen properties, should be something that the user can perceive (such as size or color). Since you have flexibility about what the random properties are, feel free to be creative: for instance, you might have some objects that move differently than others (based on their random property). If you want some sort of property which isn't obvious - such as a point value, mass, or whatever - then you probably will need to type out a label for each object, with this info.

(spec continues on the next page)

7 Requirements for your Creative Program

You must implement another program; watch the intro video for this project to get an idea of the sorts of projects you might build.

It must:

- Have at least one new object class, different than the ones from Three Shapes. (It is permissible to have only one class in your program.)
- Create some objects. If your system wants to create them all at once, when the game starts, that's fine. Or, it can add and remove them as the game runs.
- Move the objects around. Note that if you want to create multiple classes, and some of the objects are fixed in space, that's fine - but **some** of them must move.
- Define some sort of interaction between the objects. They must affect each other, somehow. In one of my demos, they bounce off each other; in another, objects swallow each other if they touch; in another, I tried to model magnetic attraction and repulsion.

8 Turning in Your Solution

You must turn in your code using GradeScope.