CS1410 Assignment 6: Asteroids

In this assignment, you will write the classic arcade game "Asteroids". In the process, you will explore class hierarchies and practice using inheritance and polymorphism as part of a larger object-oriented program.

The assignment is split into three parts, each with its own due date.

In the first part, you will implement a basic ship that can move around, and asteroids that move around the screen. You will also implement the basic framework that will control painting the screen and updating the state of the game.

In part two, you will refactor some of the code you have written to form a simple class hierarchy, with common code implemented in a base class. You will add user control of the ship, and collision detection, making it possible for your ship to blow up when it is hit by an asteroid.

In the third part, you will give the ship a way to fight back by implementing a bullet that can destroy asteroids, and you will implement stars that twinkle in the background. You will do additional refactoring, creating a more complex class hierarchy.

As in the previous assignments, you can see what the finished product will look like by running some sample code, but you will not simply be replacing one class at a time with your own code. Instead, you will start with some minimal skeleton code and gradually build your game, testing each step along the way, but not having a fully functional game until you have implemented everything.

Installing pygame

To handle the graphics and user input, you will use the `pygame` library. It is already installed on the lab computers (for all operating systems), but if you want to install it on your own machine you can find the files and instructions here:

- http://pygame.org/download.shtml

To install it on your own Ubuntu Linux machine, install the package `python-pygame` using this command:

```
sudo apt-get install python-pygame
```

This document will provide all of the information you need to use `pygame` for this assignment, but you may find it helpful to try out a `pygame` tutorial or use the references, all of which are available here:

- http://www.pygame.org/docs/

The most important `pygame` classes for this game are `Draw`, which you will use to plot graphics on the screen, and `Surface`, which represents an image in memory. The game window is a `Surface`, and you will use it to render everything on the screen.

Getting started

Download the starter kit here:

- asteroids.zip

Unzip it and run `asteroids.pyc` in the `sample` subdirectory to try it out. The starter kit consists of the following classes. Note that each class is implemented in a file by the same name (except the filename uses only lowercase letters):

- `Game`: this class implements the main game initialization code and the main loop. The constructor opens the window and prepares it for rendering. The `main_loop` method sets a timer to run the game at a set number of frames per second. For each frame, it monitors the keyboard for user input, and calls two other methods:
  1. `game_logic`: this method is responsible for updating the state of the game based on time passing and keyboard input. It does not draw anything, but implements the logic and control of the game.
  2. `paint`: this method draws everything to the screen. It does not do anything else. In particular, it does nothing to modify the game state.

  Note that the complete source code for `game.py` is provided. You should not need to change anything in it.

The methods `game_logic` and `paint` do not do anything except raise an exception, so running them directly will cause the game to crash. You will not modify the file, but will instead *subclass* `Game` to make your own `Asteroids` class. `Asteroids` will inherit the basic functionality of the `Game` class, but it will override the `game_logic` and `paint` methods, providing new implementations that do useful work.

- `Polygon`: this class represents a single polygon on the screen. The ship and the rocks are implemented as polygons. The beginnings of this class are provided for you. You will extend and modify this class as you implement your game.
- `Point`: this class is very simple; it represents a single point on the cartesian plane, represented as a pair of coordinates. You will use it throughout the game. Note that `Point` objects should never be changed once they are created. If you ever need to move a point, you should create a new object to replace the old one. You should not need to modify this class.
- `config.py`: this file controls some of the game parameters. You can change values in here to change how fast the ship moves, how many rocks there are, etc.

You can use `game.py`, `point.py`, and `config.py` without any modifications, though you are free to change them to customize your game. In particular, if you wish to change the look and feel of your game, you should modify the values in `config.py`.

This assignment is long and complex. It is presented progressively: each step will build on what you have done before. You must be careful to test each step as you go. There is little point in moving on until you are confident that everything you have written already works correctly.

Part I

The first step is to familiarize yourself with the code that is provided. The `Game` class is the same as you used in the Platformer game.

Remember that the main work of the game will be provided by *abstract methods*, namely `game_logic` and `paint`. They are called abstract because they define an interface, but they do not implement anything. If the versions implemented in `Game` are ever called, they will raise an exception and end the program.

Asteroids class

Your first step should be to create a new class (in a new file) called "`Asteroids`", which inherits from `Game`. You may want to examine `platformer.py` to see how the `Platformer` class inherited from the `Game` class.

The `Asteroids` class has the following methods:

- `__init__(self, name, screen_x, screen_y, frames_per_second)`: the constructor takes four parameters, the name of the game as a string, the desired width of the game in pixels, the desired height of the game in pixels, and the desired number of frames per second.

  The constructor should call the constructor of the base class, `Game`. Remember that `Asteroids` objects are extended versions of `Game` objects. This line will let the `Game` class initialize the parts of the object that are inherited. As you proceed, you will create the ship, rocks, the bullet, and stars in the `Asteroids` constructor, but for now it need not do anything else.

- `game_logic(self, keys, newkeys)`: the two parameters are python `set` objects. `keys` are all keys currently held down, and `newkeys` are all keys that were pressed during the previous frame. This method is where you will update the game state information. For example, the position of the ship and the rocks will be updated here. For now, just make this method a stub by putting in the python command `pass`. As you progress, you will add to this method.

- `paint(self, surface)`: this method is where you will draw the current status of the game. The single parameter is a pygame surface. For now, just make this method a stub by putting in the python command `pass`. As you progress, you will add to this method.

As you implement functionality in other classes, you will come back to this class to drive everything. `game_logic` gets called once per frame before `paint`, and it is responsible for updating all of the game logic. You will add methods with the same names and parameters to every class you implement. `Asteroids` will implement some logic directly, but mostly it will just call the `game_logic` methods of other objects, deferring work to them.

The `paint` method is similar: its main job is to call the `paint` methods of other objects. These act as "heartbeat" methods:

the heartbeat originates in `Game`, but it is passed to every active object in the game, giving each object a chance to update itself. Then `paint` is called in the same fashion, letting every object draw itself.

Outside of the `Asteroids` class but within the same file, define a `main()` method that creates a single `Asteroids` object and calls its `main_loop()` method. Note that this method is inherited from `Game`, so you do not need to write it.

When you instantiate the `Asteroids` object, you'll need to pass the values of `config.TITLE`, `config.SCREEN_X`, `config.SCREEN_Y`, and `config.FRAMES_PER_SECOND` to the constructor.

Call `main()` as the last step in this file. Run the `asteroids.py` file, and you should get a black window that closes when you click its close box or hit `<escape>` from the keyboard.

Polygon class

Much of the code needed in the `Polygon` class is provided. You should read through it and become familiar with what is available. If you do not understand some of the math, that is okay. You will not need to modify any of those methods.

- `__init__(self, outline, position, rotation, color)`: the parameters to the constructor are important. `outline` is a list of `Point` objects that define the outline of the polygon. `position` is a single `Point` object that defines where the polygon is located on the screen. `rotation` is a floating point number that defines the current direction that the polygon is facing. This is measured in degrees, with 0 degrees facing the right, 90 degrees facing down, 180 degrees facing left, and 270 degrees facing up. `color` is a 3-tuple of RGB values used to draw the polygon.

  Note: the angles are the reverse of what you learned in geometry or trigonometry. This is because the *y* coordinate increases as you go down.

Ship class

Create a subclass from `Polygon` called `Ship`. This class will be used to represent and display the player's vehicle.

Methods of `Ship`:

- `__init__(self, position, rotation, color)`: the three parameters will be passed on to the `Polygon` constructor. Read about their purpose above.

  The `Ship` constructor also needs to create a list of `Point` objects to send as the outline of the ship to the `Polygon` constructor.

  You are free to design the ship however you like, but make sure the first point in the list is where bullets should come out, normally the front tip of the ship. You should design it so that it is facing to the right (zero degrees).

- `paint(self, surface)`: the parameter is a pygame surface object that the ship should be drawn to. Use the `pygame.draw.polygon()` function to draw the ship to the screen.

  The `Point`s to draw can be obtained with the `getPoints()` method that `Ship` inherits from `Polygon`. This method adjusts the outline of the ship according to the position and rotation.

  See the `pygame` documentation for more on how to use this `pygame.draw.polygon()`. Note that it takes a color in the format of a triple of three values: red, green, and blue, where each value is an integer from 0 to 255. For the list of points, it wants a list of pairs (2-tuples), not a list of `Point` objects, so you will have to convert the list you get from `getPoints` into the right format. Use the `pair()` method in the `Point` class to do this for each point.

  The color should be available by inheritance from the `Polygon` class.

- `move(self):` have this method move the ship one pixel to the right. Note that this is just an increase of one in the position's x value. The current position is stored as a `Point` object in `self.position`, a data member inherited from `Polygon`. You should never modify a `Point` object directly, so instead create a new one with the *x* position incremented by one, and store it in place of the old `position` field.

- `game_logic(self, keys, newkeys):` have this method call the `move()` method. You will modify it later.

Drawing the ship

In the `Asteroids` constructor, create a `Ship` object, and save it as part of the `Asteroids` object. Place the ship in the center of

the screen, in the `config.SHIP_INITIAL_DIRECTION` and with the `config.SHIP_COLOR`. The screen size is available from `config.SCREEN_X` and `config.SCREEN_Y`.

Also, have the `Asteroids.paint()` method call the `paint()` method of the ship. At this point, running your game should show the ship in the center of the screen. Trace the sequence of `paint` calls necessary to call `paint` in the `Ship` class and make sure you understand how they are all connected.

This is a checkpoint. If you don't see your ship, go back up and fix something!

Moving the ship

Call `game_logic()` on the ship within `Asteroids.game_logic()`. When you run the game now, the ship should start out in the middle of the screen and fly off the right edge, leaving a trail. Go fix `move()` so that when the ship flies off the right edge, it wraps around to the left edge. You'll need `config.SCREEN_X`.

While you are at it, make it so flying off the left edge wraps around to the right edge, and do the same for the top and bottom. You'll need `config.SCREEN_Y` for the top and bottom checks. In `move()`, change the motion (by hand) to test each of these boundary conditions, i.e., have the ship move right, left, up, and down.

This is a checkpoint. If your ship doesn't wrap around in each of the four directions, go back up and fix something!

Removing the ship's trail

At this point, the ship will still leave a trail behind it. Fix this by blanking the screen each frame before drawing anything else. Do this in `Asteroids.paint()`. Use the `fill` method of the `Surface` class (see the PyGame documentation). Set the background to the color given in `config`.

This is a checkpoint. If your ship isn't visible, or if it is still leaving a trail as it moves, go back up and fix something!

Rock class

Create a subclass from `Polygon` called `Rock`. This class will be used to represent and display the rocks flying around the player's ship. Each `Rock` object will represent a single rock in the game.

Methods of `Rock`:

- `__init__(self, position, rotation, color)`: the three parameters will be passed on to the `Polygon` constructor. Read about their purpose above.

  As with the ship, have the constructor define the outline as a fixed series of points in a list, and pass that along to the `Polygon` constructor. You are free to design the rock outline however you like. Don't put too much effort into it. You will be creating random outlines later.

- `paint(self, surface)`: like `Ship.paint()`, have the rock be displayed by this method.

- `game_logic(self, keys, newkeys)`: for now, just make this method a stub by putting in the python command `pass`. As you progress, you will add to this method.

- `rotate(self, degrees)`: the parameter is the number of degrees to rotate the rock. For now, just make this method a stub by putting in the python command `pass`. As you progress, you will add to this method.

- `move(self):` for now, just make this method a stub by putting in the python command `pass`. As you progress, you will add to this method.

Drawing the asteroids

In the `Asteroids` constructor, create `config.ROCK_COUNT` `Rock` objects., and save them as part of the `Asteroids` object. Place each `Rock` randomly on the screen. Use `config.ROCK_COLOR` for the color. Use the `uniform` function in the `random` class to pick the rotation between 0.0 and 359.99 (look it up in the `random` module documentation).

Have `Asteroids.paint()` and `Asteroids.game_logic()` call the same methods for each of the rocks.

This is a checkpoint. If your asteroids aren't visible, or they are not randomly placed and rotated, go back up and fix something!

Rotating the asteroids

After testing this much, make your rocks rotate randomly. First, modify `Rock.rotate()` so that is adds the degrees to rotate to `self.rotataion`, which is inherited from the `Polygon` class. Make sure that `self.rotation` stays between 0.0 and 360.0.

Next, modify the `Rock` constructor to accept another parameter, the rotation speed for the rock. Modify the `Asteroids` constructor to randomly choose a rotation speed for each rock, and pass it to the `Rock` constructor. Use `random.uniform` to pick a number in the range specified in `config`. Make sure that your rocks can rotate in either direction: do this by randomly picking a random speed of rotation, then randomly deciding if the direction should be positive or negative.

Then within the `Rock.game_logic()` method, call the `rotate()` method with the speed that was assigned in the constructor.

By this point, you should have a ship that moves across the screen endlessly, and a bunch of rocks that rotate in-place at random speeds.

This is a checkpoint. If your asteroids aren't rotating at different speeds and directions, go back up and fix something!

Moving the asteriods

Now, make the rocks move. For now, it is sufficient to have them all move in the same direction at the same speed (similar to how you implemented it in `Ship`). We will see a better way to do it later.

Modify `Rock.move()` to increase or decrease the *x* or *y* position of each the rock by 1. Be sure to handle all of the edge wrap conditions.

Modify `rock.game_logic()` to call the `move()` method.

This is a checkpoint. If your asteriods don't wrap around in each of the four directions, go back up and fix something!

Variety in the asteriods

To make your rocks more interesting, give each one a random outline. Here is one way to do it:

- Loop through the range of degrees in a circle, starting at zero and ending before 360.0. Do it with the number of steps specified in `config.ROCK_POLYGON_SIZE`. For example, if that value was 4, you would loop over the values 0, 90, 180, and 270. If it was 6, you would loop over the values 0, 60, 120, 180, 240, and 300.
- At each position, pick a random radius (use `random.uniform` to pick a value in the range specified in `config`).
- Convert the number of degrees to radians using `math.radians`.
- Calculate the polygon point `x = math.cos(radians) * radius` and `y = math.sin(radians) * radius`. This will give you a polygon with a fixed number of sides, but with a random outline.

Now you should have a screen full of randomly-shaped rocks spinning at random speeds and drifting across the screen. At this point your ship should be of a fixed outline and not rotating, but drifting across the screen.

This is a checkpoint. If your game doesn't appear as described above, go back up and fix something!

Congratulations, you have finished the first part! Go pass it off with the lab assistant, then zip up your files and submit them.

Part II

The second part is to consolidate common code into base classes, implement control over the ship (and improve motion of rocks), and implement collision detection.

Refactoring

Start by considering what you have done so far in the `Ship` and `Rock` classes:

1. Each has a constructor that sets the outline, initial position, and initial rate of motion (if any).
2. Each has a `move` method to move it across the screen with each `game_logic` heartbeat.
3. Each has a `game_logic` method to call the `move` method.
4. Each has a method to paint the object to the screen using the outline, position, rotation, and color stored in the object

Some of these methods are (or at least could be) identical for `Ship` and `Rock` instances. Make sure that both classes use the same internal field names for storing their respective rotation angles and positions, and then make sure they each have identical method definitions when possible.

Next, factor these identical methods into the base class, namely `Polygon`, and delete them from `Ship` and `Rock`. You should be able to move `paint`, `move`, and `rotate` into `Polygon`. If you cannot, go back and rewrite them until they are all identical for both `Ship` and `Rock`, and then move them over.

Note that the `rotate` method should take a single argument (some number of degrees), and it should rotate the object that number of degrees. The `Rock` class should call it every time it receives a `game_logic` heartbeat, while the `Ship` class will only call it in response to keyboard input. Still, the action taken is the same in either case, so the `rotate` method itself can be factored into the `Polygon` class.

When you have done this refactoring, you should have code that does exactly the same thing as before, except that now it will be a bit shorter.

This is a checkpoint. If your game doesn't appear as described above, go back up and fix something!

Ship motion

The next step is to rotate the ship under user control. The arguments to `game_logic` are two python `set`s, which are objects that collect data in no particular order (but with each element appearing at most once in the set). The first set, called `keys`, tracks all of the keys that are currently being held down by the user. In the `game_logic` method for the `Ship` class, check if the user is pressing the left key:

```
if pygame.K_LEFT in keys:
    # do something
```

If so, rotate the ship left by the amount specified in `config.SHIP_ROTATION_RATE` (to rotate left, add a negative number). If the user is holding down the right arrow, rotate the ship right. Test it, and you should now be able to control the ship's rotation.

This is a checkpoint. If you can't rotate the ship as describe above, go back up and fix something!

Next, program the ship's thrusters to allow it to move forward and backward. To get the zero gravity effect, your ship should accelerate in the direction it is currently facing.

Modify the `Polygon` constructor to create two data members `dx` and `dy` to represent the current velocity of the ship. Initialize them both to 0.

Modify `Polygon.move()` to add `dx` to the x position and `dy` to the y position.

Create the method `Ship.accelerate(self, acceleration)`. This method should modify the values of `self.dx` and `self.dy` according to the following formulas:

```
self.dx = self.dx + acceleration * math.cos(angleinradians)
self.dy = self.dy + acceleration * math.sin(angleinradians)
```

Where `acceleration` is the value sent to the method, and `angleinradians` is the the ship's angle converted to radians (using `math.radians`).

Modify `Ship.game_logic()` to call `accelerate` when the up arrow or down arrow is held down. The amount to accelerate is `config.SHIP_ACCELERATION_RATE`. Use a positive value to speed up, and a negative value to slow down.

After implementing this, you should have full control over how the ship moves.

This is a checkpoint. If you can't accelerate or decelerate the ship as describe above, go back up and fix something!

Rock motion

Next, apply the same system of motion to the rocks.

Add the method `Rock.accelerate(self, acceleration)`, that works like `Ship.accelerate()`.

Modify the `Rock` constructor to receive a parameter for the forward speed of the rock, then it should call the `accelerate()` method to start the rock moving.

Modify the `Asteroids` constructor to pick a random acceleration value from the range specified in `config`, and pass it to the `Rock` constructor.

At this point, you should have randomly generated rocks that rotate at random speeds (in random directions), and float around the screen in straight, randomly-chosen directions. You should also be able to control the ship and direct it around the screen with a full range of rotation and forward and reverse thrusters.

This is a checkpoint. If your asteroids aren't moving at random speeds in random directions, go back up and fix something!

Next, take a few minutes to review `config`, and make sure you are using all of the relevant parameters defined in there. Some of them will not apply yet, but you should be using all that do (and not hard-coding values in your source code).

Collision detection

Exact collision detection is hard to implement, so we will approximate it. The `Polygon` class already has a `contains` method that tests whether or not a given point is inside the polygon.

We will say that two polygons collide or intersect if any point from the one polygon is contained in the other polygon. We must check every point from the first polygon to see if it is contained in the second polygon, then check every point in the second polygon to see if it is contained in the first polygon. Any single point being contained will be enough to be classified as an intersection.

Implement an `Polygon.intersect(self, other_polygon)` method. It returns `True` if the two polygons intersect each other, and `False` if they do not.

Use `getPoints()` to get the points of the first polygon. For each of the points, check to see if is contained (`contains()`) in the second polygon. If any point is contained, return `True`.

Use `getPoints()` to get the points of the second polygon. For each of the points, check to see if is contained (`contains()`) in the first polygon. If any point is contained, return `True`.

If no point was reported as contained, then return `False`.

Now use the `intersect()` method from within `Asteroids.game_logic()` to test if any of the rocks collides with the ship.

To test this out, change the color of a rock when it collides with the ship, and then fly around for a bit and see if it works as expected.

This is a checkpoint. If collisions to not occur as described, go back up and fix something!

Blowing up the ship

Of course, the ship is not supposed to paint rocks different colors when it hits them, it is supposed to explode. Each `Polygon` object has an `active` field that is initially set to `True`. When this field is set to `False`, the polygon should act as though it has been deleted. In particular:

- The `paint` method of `Polygon` or any derivative class should return without doing anything if `active` is `False`.
- The `game_logic` method of `Polygon` or any derivative class should return without doing anything if `active` is `False`.
- When doing collision detection, you should ignore any `Ship` or `Rock` that is inactive. Do this test *before* checking for a collision, not after, since collision detection is relatively expensive.

Implement this change with the following modifications:

- Add `Polygon.is_active(self)`: returns the value of the `active` data member.

- Add `Polygon.set_inactive(self)`: sets the `active` field to `False`.
- Modify all `paint()` methods in `Polygon` and its derived classes to check the `active` field as described above.
- Modify all `game_logic()` methods in `Polygon` and its derived classes to check the `active` field as described above.
- Modify `Asteroids.game_logic()` to use the `is_active()` method to determine which collisions to check.
- Modify `Asteroids.game_logic()` to use the `set_inactive()` method to deactivate the ship on collision.

Now, when you detect a collision between the ship and a rock the ship should disappear from the game. If you still have rocks changing color when they collide with the ship, you should not see any rocks changing color (at least not after the one that blows up the ship).

Remove the color changing code now.

This is a checkpoint. If the ship doesn't disappear on collsion, go back up and fix something!

Just for fun (and to test if you have implemented deactivation properly), change the behavior so that a collision between the ship and a rock destroys (deactivates) the rock instead of the ship. Have the collision detection code print a message out each time it does this, and make sure you are not encountering "ghost" rocks that still hit the ship after they have been destroyed. You should be able to kill all of the rocks and then fly around the screen freely without detecting any further collisions.

Do not forget how to switch the behavior back to normal, but leave it as it is for now. You will pass off this part with the ship in destroyer mode.

This is a checkpoint. If the asteroids don't disappear on collsion, go back up and fix something!

Further refactoring

Review the new code that you have written, and check if there is anything in your derivative classes that could be moved into a base class. Make sure your `accelerate` method is in `Polygon`, for example. Also, make sure you have a `move` method that uses `dx` and `dy`, and make sure both of these are implemented in `Polygon` instead of `Rock` or `Ship`. `intersect` should also be in `Polygon`.

In fact, the only things that should be implemented directly in the `Ship` class are the constructor and the `game_logic` method, and the same is true of the `Rock` class. Everything else should be part of `Polygon`.

This process of writing code and later refining it and refactoring it is something you should always follow. Object hierarchies rarely spring into existence fully formed. Usually you will discover commonalities in classes as you implement them that were not apparent in the planning stages.

Congratulations, you have finished the second part! Go pass it off with the lab assistant, then zip up your files and submit them.

Part III

The third part is to expand the class hierarchy to include circles, refactor the code further, implement bullets that can destroy rocks, and add stars that twinkle.

Circle class

Bullets and stars are circles, not polygons. As such, it does not make sense to create them by subclassing the polygon class. Instead, create a new `Circle` class to act as the base class for bullets and stars.

A circle is defined by its center point (a position) and its radius. It also needs a color and a rotation. Rotating a circle may not seem very useful at first, but having a circle facing a certain direction can help with controlling motion. In fact, motion and acceleration can be exactly the same as for polygons.

Circles should also have an `active` flag, similar to ships and rocks. This will help later when implementing the bullet, which is not always on the screen.

Implement the `Circle` class. It needs to have at least the following methods:

- `__init__(self, position, radius, rotation, color)`
- `paint(self, surface)`
- `game_logic(self, keys, newkeys)`
- `move(self)`
- `rotate(self, degrees)`
- `accelerate(self, acceleration)`
- `is_active(self)`
- `set_inactive(self)`

You should be able to figure out the details of the implementations of these methods. You may want to read about `pygame.draw.circle` for the `paint` method. The `game_logic` method need not do anything (leave that to the subclasses), but should still exist. Use `NotImplementedError()`, similar to how it is already done in other places.

You may want to use the python interpreter or a simple test program to test the basic functionality of this class.

Expanding the shape hierarchy

Even before creating any bullets or stars, it is clear that circles have much in common with the other shapes in the game. To take advantage of this, create a new `Shape` base class.

`Shape` class methods:

- `__init__(self, position, rotation, color)`: the constructor accepts and stores the position, rotation, and color of the shape, and gives it an initial motion, `dx` and `dy`, (which should have the shape standing still).
- `paint(self, surface)`: this should be abstract and do nothing (except raise the `NotImplementedError()` exception).
- `game_logic(self, keys, newkeys)`: this should be abstract and do nothing (except raise the `NotImplementedError()` exception).
- `move(self)` :
- `accelerate(self, acceleration)`: both of these methods you have already defined for polygons will apply equally well to circles, so move them up ("promote" them) into the `Shape` base class.

Have `Polygon` and `Circle` inherit from `Shape`. Make sure that both `Circle` and `Polygon` call the `Shape` constructor from within their respective constructors.

Since `Polygon` and `Circle` both inherit from `Shape`, they do not need to have `paint` and `game_logic` placeholder methods any more. Only include those two in any of the other classes if they actually do something; otherwise they can inherit the `NotImplementedError` behavior from `Shape`.

This is a checkpoint. With the inheritance from `Shape` to `Polygon` to `Ship` and `Rock`, test that you system still behaves the way it did before `Shape`. If there is a problem, go back up and fix something!

Bullet class

A bullet is a circle, so create a `Bullet` class that inherits from `Circle`. Like `Ship` and `Rock`, `Bullet` will inherit most of its functionality and extend it only in a few key ways.

Instead of creating a new bullet every time the player shoots, you will create a single `Bullet` in the `Asteroids` class. It will be inactive when it is not in use, and it will be active when the player fires the bullet.

`Bullet` class methods:

- `__init__(self, position, radius, rotation, color)`: the constructor should call the `Circle` constructor. After that, it should set itself to be inactive, since a bullet is only visible when the user requests one.

- `game_logic(self, keys, newkeys)`: if the bullet is inactive, `game_logic` should return immediately. Otherwise, it should move the bullet, and then check if the bullet is still on the screen. Instead of having it wrap around like the polygon shapes, have it go inactive once it leaves the boundaries of the screen. Be careful when making this check: if you just call the `move()` method and then check if the bullet is off the screen, it never will be, because `move()` will wrap it around to the other side. Instead, figure out where the bullet will end up once move is called (but without actually calling it), and then test that location to see if it is off the screen. Only call `move()` after doing the test.

- `fire(self, position, rotation)`: this method takes a starting position and a rotation as arguments. It should set the

bullet's position and rotation to match the arguments, activate it, and set its current `dx` and `dy` to zero. Then it should call `accelerate` to fire the bullet in its specified direction (similar to how rocks are accelerated a single time). Use the speed specified in `config`.

It will be useful for you to add methods `Shape.get_rotation(self)` and `Shape.set_active(self)`. Can you guess what they need to do?

Create a single bullet in the `Asteroids` constructor and store it. In `Asteroids.game_logic()`, check when the user first presses the space bar (use `newkeys` instead of `keys`), and when that happens (and the ship is active), fire the bullet. The initial position should be the nose of the ship. Call the ship's `getPoints` method, and take the first point in the list that is returned (you must use `getPoints` or the ship's position and rotation will not be factored in). Take the ship's rotation and use it for the bullet's rotation, so the bullet is fired in the same direction as the ship is facing.

After plugging in the bullet's `game_logic` and `paint` heartbeats in `Asteroids`, you should be able to fire bullets, which disappear once they leave the screen (make sure they do not continue flying forever off the edge of the screen).

This is a checkpoint. If the bullet does not appear when fired, or does not stop when it goes off of the screen, back up and fix something!

Collision detection and bullets

Bullets collide with rocks and destroy them. The collision detection you have implemented already only works for polygons. However, it can be extended to work for circles as well.

First, implement a `Circle.getPoints` method. Note that this is only an approximation of the points in a circle, and it will only be used for collision detection. `Circle` objects will still be drawn circular by `pygame.draw.circle`.

Approximate the circle by taking some number of points evenly distributed around the circle (use `config.BULLET_POINT_COUNT` to determine how many). This is similar to how you generated random rock shapes, but use the circle's radius instead of generating a random radius at each point. Add the circle's current position, and ignore the rotation.

Next, implement a `Circle.contains` method. This can just use the distance formula, and check if the distance from the center of the circle to the given point is less than or equal to the radius of the circle:

```
def contains(self, point):
    dist_x = self.position.x - point.x
    dist_y = self.position.y - point.y
    return dist_x*dist_x + dist_y*dist_y <= self.radius*self.radius
```

Now move the `intersect` method from `Polygon` to `Shape`. As long as it only uses `contains` and `getPoints`, it should be able to detect intersections between all kinds of shapes.

Since you are relying on all shapes having `contains` and `getPoints` methods, put abstract methods for both of them in the `Shape` class (the kind that just raise `NotImplementedError()` exceptions).

Finally, check for collisions from within the `Asteroids` class. As you check active rocks for collisions with the ship, also test them for collisions with the bullet. If an (active) bullet hits an rock, the rock and the bullet should both be made inactive. If an active rock hits the (active) ship, the ship should be made inactive. If you still have destroyer mode active (the ship destroys rocks by running into them), change to the normal game mode (rocks destroy the ship by running into it).

At this point, the game should be fully playable.

This is a checkpoint. If the bullet does not destroy a rock, rocks do not destroy the ship, etc., back up and fix something!

Star class

Stars make space feel less lonely and austere. Create a `Star` class that inherits from `Circle`. Its constructor should pick a random position and a random brightness (set all three values of the color triple to the same random value between 0 and 255). It should have a rotation of 0.0, and a radius as defined in `config`.

Stars do not do much. In particular, they do not interact in any way with the other objects of the game. They should twinkle, however. Pick a random amount to change the brightness each frame. Add or subtract a random amount between zero and `config.STAR_TWINKLE_SPEED`. Add (or subtract) that amount from the red, green, and blue values (so they remain in lockstep with each other), but make sure they are always in the range from 0 to 255.

Create a bunch of these in `Asteroids` (use the number defined in `config`, as usual). Make sure to give each one a `game_logic` heartbeat and a `paint` call. Stars should be painted before anything else so they are always behind the other shapes.

At this point, everything should be implemented. Take a few minutes to make sure you have not forgotten anything. You should have used all of the values in `config`. Look through your classes and make sure everything that is shared by multiple classes is shared through inheritance, not through code duplication.

Optional fun

Here is a list of option additions you can make to your game:

- Instead of destroying a rock the first time it is hit with a bullet, split it into two pieces. Start with large rocks, which split into two medium rocks. When a medium rock is hit, replace it with two small rocks. Only small rocks are completely destroyed. Have large rocks move slowly, medium rocks move twice as fast (and moving away from each other initially), and small rocks twice as fast again.
- Create an image for the ship instead of a polygon. Make a polygon that approximates the shape of the ship (and is used for collision detection), but add a custom `paint` method that draws an image loaded from disk instead. Methods to rotate images can be found in `pygame.transform`.
- Add sound effects and a music track.
- Add points, levels, and multiple lives. When the ship is destroyed, keep displaying everything as normal for five seconds or so (to let it sink in), and then restart if the player has lives left. Harder levels could have more rocks, faster moving rocks, etc.
- Add additional controls to the ship. Be careful that you do not make it too easy to control. Part of the challenge of the game is controlling a ship in zero gravity. A random teleport option is one possibility: if you are about to be destroyed, you can take your chances and teleport to a random location.
- Make a two-player mode. Put the controls for one player on the left side of the keyboard and the controls for the other on the right side. Create two ships that are independent of each other. You could have cooperative mode (where the ships cannot harm each other), or competitive (where the ships have to worry about hitting each other, dodging bullets, and still staying out of the way of rocks).

What to turn in

For each deadline, put all of your files in a `.zip` file and submit it.