Game: Asteroids

Paul Wright Revision 1.10, October 7, 2001

Credits

For the LaTeX source of this sheet, and for more information on LiveWires and on this course, see the LiveWires web site at http://www.livewires.org.uk/python/

Introduction

This is a Python games worksheet. When you've finished it, you'll have written a version of a classic arcade game called Asteroids.

What you need to know

- The basics of Python (from Sheets 1 and 2)
- Functions (from Sheet 3; you might want to look at Sheet F too)
- Classes and Objects (from Sheet O)

> *You should definitely look at Sheet O before starting this worksheet, even if you did the Beginners' course, as this sheet assumes you know about classes and objects, which aren't in that course.*

What is Asteriods?

I'll begin by explaining how the game works.

Asteroids is a 2-dimensional game: what's on the screen is flat rather than being a view of a 3-D world (so it's like Worms, say, rather than Quake).

The game starts with the player's space ship on the screen, along with some asteroids. The ship starts off not moving, but the asteroids start off moving at a random speed in a random direction.

In the game, the player guides the ship around the screen, shooting bullets at asteroids. When an asteroid is hit, it splits up into smaller asteroids which fly off. When a small asteroid is hit, it disappears from the screen. If an asteroid or bullet hits the player's ship, the player loses the game. If the player manages to shoot all the asteroids on the screen, the player wins the game.

The player controls the space ship by rotating it left and right and using the rocket thrusters on the back of the space ship to push it in the direction it's pointing in. When the player fires, the bullets come out of the front of the space ship and travel in the direction the ship is pointing in.

When the asteroids hit each other, they just pass right through each other. When anything goes off one edge of the screen, it

comes back on to the screen on the opposite side. In general, things on the screen keep moving at the same speed and in the same direction unless one of the things described above happens to them.

Things on the screen

In the outline of the game we saw above, there are 3 kinds of objects which can be on the screen. They are:

- asteroids,
- space ships,
- bullets fired by a space ship.

The LiveWires games library provides us with a way of putting shapes on the screen, and also a way for them move across the screen by themselves (so we don't have to keep telling them to move, as you might have done if you've written games in BASIC before). These things are provided as classes, as you might expect from Sheet O. We are going to make sub-classes of the useful shape classes to create the things we want on the screen.

The Asteroid class

Let's start with the asteroids. Type the following into the editor window (the editor window is the one without the >>> characters before each line. In this window, Python doesn't run what you type as soon as you type it, but lets you type a lot of things to be run when you tell it).

```
from livewires import games
from livewires import colour
import math
import random
```

First, we tell Python we wanted to access the things in the LiveWires `games` module, and also the colour names supplied by the `colour` module. We also want to use the `math` module, which comes with Python ("math" is the American abbreviation for mathematics, the equivalent of "maths" in British English), and the `random` module, which is used to generate random numbers.

```
SCREENWIDTH=640
SCREENHEIGHT=480
```

Then we set up `SCREENWIDTH` and `SCREENHEIGHT` to hold the width and the height of our graphics window. We did this so we can easily change the width or height if we want to, just by changing the program at that point. If we'd used the numbers 640 (the width) and 480 (the height) all over the place and we decided we wanted to change them to make our screen bigger, we'd have to go through the program looking for all the times we've used those numbers, work out whether they were being used for the screen size or something else (eg if we score 640 points for hitting an asteroid, we don't want to change that when we change the screen size), and change the ones that relate to the screen size. So, we define some variables so we can refer to the sizes by name instead. Programmers who don't do things like this tend to spend a lot of time trying to work out how to alter simple things!

Another thing which we're doing here is following a set way of naming our variables. Things which we won't change during the program have names which are ALL IN CAPITALS. Python *doesn't* force us to do this, but doing it makes it easier for someone reading our program to know what sort of thing a variable is without having to puzzle through the program to work it out.

A thing which we won't change is called a *constant*, by the way.

```
class KeepsToScreen:
  def keep_to_screen (self):

    oldpos = (x,y) = self.pos ()

    if x > SCREENWIDTH:
      x = 0
    if x < 0:
      x = SCREENWIDTH
    if y > SCREENHEIGHT:
      y = 0
    if y < 0:
      y = SCREENHEIGHT

    if (x,y) != oldpos:
      self.move_to ((x,y))
```

The next thing we do is make a class called `KeepsToScreen`. Objects from this class only know how to do one thing, the `keep_to_screen` method. Notice we've got another rule for naming classes: the first letter of each word in the class name is a capital letter.

> See if you can work out what the `keep_to_screen` method does. You probably need to know that the `self.pos` method returns the x and y co-ordinates of the object.

```
class Asteroid (games.Sprite,
        games.Mover,
        KeepsToScreen):

  def __init__ (self, screen, x, y, size=2):

    self.init_sprite (screen=screen,
      x=x, y=y, image=Asteroid.image)

    self.init_mover (dx=1, dy=1)

  def moved (self):
    self.keep_to_screen ()
```

Now we create the `Asteroid` class, which all our asteroids will belong to. Notice that `Asteroid` has more than one parent class (or superclass). That's because an asteroid is more than one kind of thing. It's a `Sprite`, which is a picture taken from a file you've make using a paint package. It's a `Mover`, something which moves across the screen by itself. It's a `KeepsToScreen` (you worked out what that is, right?). That means our asteroid will inherit a whole load of methods from the `Sprite` class, and some from `Mover`, as well as `KeepsToScreen`'s one method. The `Sprite` and the `Mover` class come from the `games` module, so we have to tell Python to find them in that module by prefixing their names with `games`.

Now we define the `__init__` method, which is called when we create a new object from the class. When we create an asteroid, we need to tell it which screen to be on, and where it is on the screen. Those are the `screen`, `x` and `y` variables in the definition.

Next, we call the asteroid's `init_sprite` method, which it inherited from the `Sprite` class. We've referred to something called `Asteroid.image` which we've not created yet: it will hold a picture of the asteroid which we've loaded from a file.

We give the `init_sprite` method a screen for the asteroid to be on, the x and y co-ordinates of the centre of the image of the asteroid, and the image itself.

Then we call the `init_mover` method. The `Asteroid` class inherits this method from the `Mover` class. A `Mover` knows how to move itself around the screen: the things we tell it are how far to move each time it moves, in the x and y directions. We've told the `Asteroid` to move one pixel right and one pixel upwards every time it moves.

An object from the `Mover` class will call its own `moved` method every time it moves. We have to write the `moved` method ourselves. Our `moved` method so it checks whether the Asteroid has moved off the screen and puts it back on the screen at the other side if it has.

```
                                                    These aren't inside the class, so start from the left
                                                    of the window.
my_screen = games.Screen (width=SCREENWIDTH,
                          height=SCREENHEIGHT)

Asteroid.image = games.load_image ("asteroid.bmp")
stars = games.load_image ("stars.bmp")

my_screen.set_background (stars)

Asteroid (screen = my_screen,
  x = SCREENWIDTH/2,
  y = SCREENHEIGHT/2)

my_screen.mainloop ()
```

Finally, to get things going, we tell Python to make a `Screen` object with the width and height given by `SCREENWIDTH` and `SCREENHEIGHT`, and to put a single `Asteroid` on the screen. Notice that we had to tell the `Asteroid` which screen it was on: if you look at the `__init__` method of `Asteroid`, you'll see it takes the screen we give it and puts a circle on it.

Before you run the program, you'll need a picture of an asteroid and a picture of a starry sky. The `asteroid.bmp` and `stars.bmp` files should have come with the worksheets[1]. Move them into the same directory as the Python program you're writing.

We load these pictures into the `image` of the `Asteroid` class and into a variable called `stars`. We need to do this after the `Screen` is created, as this sets up some information which the `load_image` needs.

Having given the `Asteroid` class a picture to use, we create a new asteroid.

Then we start the `Screen` running: that's what the `mainloop` method does.

Now, when you run your program, you should see an asteroid moving across a starry sky. You'll need to close the window to make it stop, otherwise when it reaches one side of the screen it'll re-appear on the other side and start again.

So far, so good. There is one thing you can do to make the `Asteroid` class a bit more useful, though. We don't really want all our asteroids to start off moving slowly upwards and to the right. We'll also want them to come in different sizes, so that when they are hit by the player's bullets they can split up into smaller versions of themselves.

The `games.scale_image` function takes an image and scales it by the number you give it, and gives you back a new image. If you were to say

```
new_image = games.scale_image (old_image, 0.5)
```

you'll end up with `new_image`, which will be a copy of `old_image` but half the size in each direction.

[1]The stars picture shows NGC 1818, a young globular cluster. It was taken by Diedre Hunter using the Hubble Space Telescope. It was Astronomy Picture of the Day on March 11, 2001. See http://antwrp.gsfc.nasa.gov/apod/ap010311.html for details.

> Instead of having a single `Asteroid.image`, make it a list called `Asteroid.images` (look at Sheet L if you're not sure about lists). When you set up the list at the bottom of your program, make the first thing in a 0.25 scaled asteroid picture, the second a 0.5 scaled asteroid picture, and the last thing the asteroid picture not scaled at all.
> You'll notice that the `__init__` method of the `Asteroid` class to takes a parameter called `size`, which isn't used at the moment. `size` is going to be between 0 and 2. (The `size=2` in at the top of the method means that `size` will be 2 unless the person creating the asteroid asks for something different). Use `size` to pick one of the images of the asteroid from the list, and give that to the `init_sprite` method as the `surface` parameter it wants.

It'd be good for smaller asteroids to be a bit faster than larger ones, too. Let's make the maximum speed of the asteroid increase as its size decreases. Type this in the `__init__` method, just above the call to `init_mover`.

```
max_speed = 20.0/(size + 1)
```

When `size` is 2, `max_speed` is 10. When `size` is 0, `max_speed` is 20 (you can see why the program uses `size + 1` if you think about what would happen when `size` is zero if the formula above just divided by `size`). So, as the size of the asteroid decreases, its maximum speed will increase.

The `random` module provides the `randint` function. What the function does is to take a number and a higher number, and return a randomly chosen number which lies between the two numbers you gave it. So `random.randint (0, 20)` returns a random number from 0 to 20.

> Using `random.randint`, change the call to `init_mover` to make the asteroids move by a random amount up to `max_speed` in the x and y directions.
> Hint: the lower number you give to `random.randint` doesn't want to be zero here. (Why not? Try it and see what happens.)

We'll come back to the `Asteroid` later. Let's look at the player's space ship now.

The Ship class

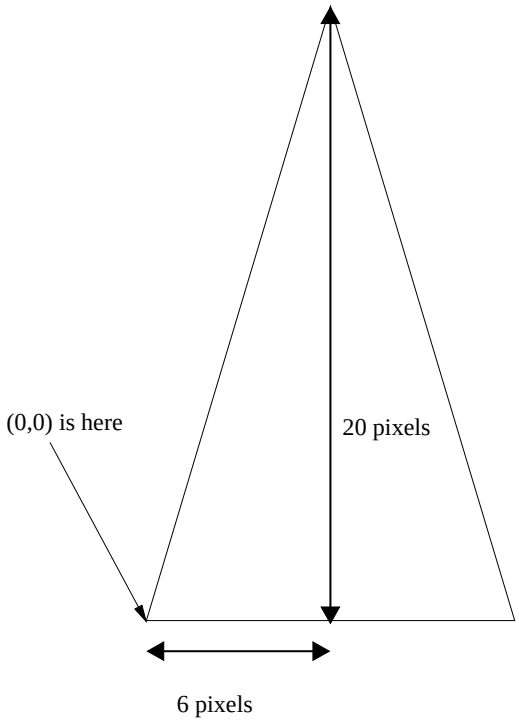
Let's think about the player's space ship. We're going to make the `Ship` class out of the `Polygon` class, plus the `Mover` and `KeepsToScreen` classes we've seen previously. A "polygon" is a many-sided shape, like a triangle or a square.

The ship in Asteroids is usually a triangle. It turns out that an isosceles triange that's about 20 pixels high (that is, from the base to the point) looks about right on our screen. Let's make the base 12 pixels across.

The `Polygon` class needs a list the co-ordinates of the points which make up the polygon. We'll give it a list of three points to make the triangle.

When we're writing co-ordinates, we usually write them with the first number presenting the x co-ordinate and the second representing the y co-ordinate. So 3 pixels along and 2 up would be $(3, 2)$.

The list we give is going to be based on a triangle at the $(0, 0)$ position. When the triangle is put somewhere else on the screen the `Polygon` class adds the co-ordinates of the place on the screen to the co-ordinates of each of the points of the triangle, so the whole triangle is shifted into the right place.



> Work out the co-ordinates of the points on the spaceship shown in the picture of the ship. The arrow on the diagram points to $(0, 0)$.

We want the `Ship` to respond to the player pressing some keys. The way we do this is by getting the `moved` method of the `Ship` class to check which keys are pressed: each time it moves, the `Ship` responds to the keys that are pressed at the time.

Luckily for us, the `Polygon` object already has methods to rotate it, so all we need to do is write a method to make it go faster when the player presses the "thrust" key. That needs some maths, so I've done it for you.

> *But I've not done it all for you. In the listing below, you'll need to fill some things in yourself. Wherever you see a symbol like this: "∞", you'll need to look at the text on the right of the listing to tell you what to fill in.*

You'll need to fill in the co-ordinates of the ship which you worked out from the diagram: I've told you where to do that. You'll need to list them as $((x_1, y_1), (x_2, y_2), (x_3, y_3))$, where x_1 is the x co-ordinate of the first point, y_2 is the y co-ordinate of the second point, and so on. You get the idea. You'll also need to fill in some methods: the notes on the right of the text tell you what to do.

We'll need to make some changes to the things we've done before: you need to get rid of the lines beginning `my_screen = Screen` and `asteroid = Asteroid`, and the call to the `mainloop` method. We'll define the `Ship` class underneath the `Asteroid class`.

```
                                                    This goes underneath the end of the Asteroid class.
class Ship (games.Polygon, games.Mover, KeepsToScreen):

  THRUST = 1                                        This is how much the speed of the ship will
                                                    increase by when we press the ''thrust'' key.

  LIST_POINTS = ( ∞ )                               Put the numbers for the co-ordinates of the ship
                                                    between the ( and )
  def __init__ (self, screen):

                                                    The line below will create our triangle.
    self.init_polygon (screen=screen, x= SCREENWIDTH/2, y= SCREENHEIGHT/2
      shape = Ship.LIST_POINTS, colour = colour.red)

    self.init_mover ( ∞ )                           Fill in between the brackets so the ship
                                                    stars off not moving.

  def moved (self):
    ∞                                               Fill in this method so that if the ship goes off
                                                    one side of the screen, it comes back on the other
                                                    side. Hint: you've already got a method to do
                                                    this.

    if self.screen.is_pressed (games.K_a):
      self.rotate_by (-4)                           rotate_by comes from the Polgyon class.
                                                    It takes an angle in degrees in the brackets.

    ∞                                               Write something to rotate the ship in the
                                                    opposite direction you press [S].


    if self.screen.is_pressed (games.K_SPACE):
      self.thrust ()

  def thrust (self):

    angle = self.angle ()                           The angle method is from Polygon, and tells us what
                                                    angle the ship is at. It's zero when the ship is pointing
                                                    straight up.

    change_in_v = angle_and_length (angle, Ship.THRUST)
    v = self.get_velocity ()                        This gets the ship's velocity.
    new_v = add_vectors (v, change_in_v)
    self.set_velocity (new_v)                       This sets the ship's velocity.
```

Colours are defined in the `colour` module. If you type:

```
>>> from livewires import colour
>>> dir (colour)
```

you'll see a list of the available colours in the `colour` module.

In the `thrust` method, we used a couple of new functions.

The first one, `angle_and_length`, turns an angle and a length into x and y co-ordinates. We use this to work out how much the ship's velocity changes by when we thrust: we know what direction the ship is pointing in and how much thrust to give it, and we convert this to a change to the x and y parts of the ship's velocity.

The second function, `add_vectors`, adds two vectors together, by adding their x and y parts. We use this to add the little bit of thrust we've just given the ship to the velocity it already has.

These two functions are defined below for you to type in. You don't have to understand how they work, but if you know about trigonometry and vectors, you might like to work it out. (If you don't know the maths, then don't worry about it!)

These aren't inside the class, so start from the left of the window.

```
def angle_and_length (angle, length):
  radian_angle = angle * math.pi / 180
  x = -(length * math.sin (radian_angle))
  y = length * math.cos (radian_angle)
  return (x, y)

def add_vectors (first, second):
  x = first [0] + second [0]
  y = first [1] + second [1]
  return (x, y)
```

Now, at the bottom of your program, make the following changes:

- Make more than one asteroid rather than just one (you could use a `for` loop to do this).
- Make each asteroid start in a random place on the screen rather than in the middle
- Create a `Ship` before calling the `mainloop` method of the `Screen`.

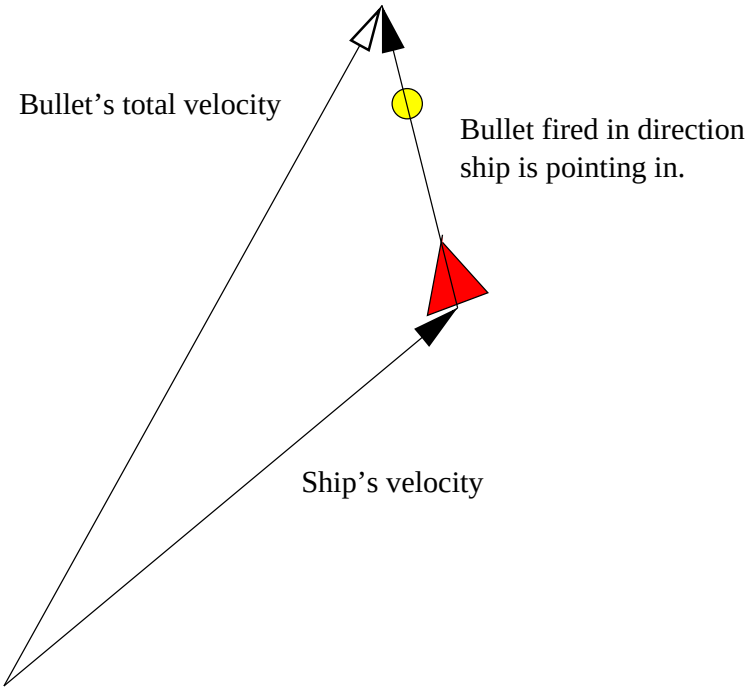
If you run your program, you should be able to control the ship with the A and S keys to rotate left and right and Space to thrust. The asteroids on the screen should also move around.

Next, we'll think about how we can let the ship shoot at the asteroids.

The Bullet class

Bullets are what the ship fires at the asteroids. When we create a bullet, it goes off with a fixed speed in the direction the ship is pointing in, but it also gets the ship's current speed and direction added on to that. The word for a speed and a direction is *velocity*. Think about a moving train. If you're on the train and you throw something across the train, you see it moving with the velocity at which you threw it. But someone watching from the trackside would see if moving with that velocity plus the velocity of the train. Since we're looking at things from the viewpoint where the space ship is moving, as well as the bullet, we need to add the velocity of the ship to the velocity at which the ship fires the bullet.

If you've learned about adding vectors in maths lessons, the diagram below illustrates what's going on. Again, if you haven't learned about vectors, don't worry about it too much.



Let's make a `Bullet` class. The `__init__` method will need to take the ship's velocity, and add its own velocity on to that. It'll need to know the angle the ship is pointing at, to know which direction to go off in, and where the ship is, to know where to start. All this involves some maths, so I'll have to do some of it for you. But, as usual, there are some things for you to do too.

```
class Bullet (games.Circle, games.Mover, KeepsToScreen):
```
This goes under the `Bullet` *class, but above the two lines which start the game.*

```
  SPEED = 5
  TIME_TO_LIVE = 50
  SIZE = 2
```
Here are some constants for the bullets: their speed, how long they last, and their size.

```
  def __init__ (self, screen, ship_pos, ship_velocity, ship_angle):
    offset = Bullet.SIZE + 21
```
We make sure the bullet starts a little offset from the ship, otherwise it'll collide with it straight away!

```
    offset_vec = angle_and_length (ship_angle, offset)
    (start_x, start_y) = add_vectors (ship_pos, offset_vec)
```

In the lines below, we're doing the addition of vectors illustrated in the diagram above.

```
    bullet_velocity = angle_and_length (ship_angle, Bullet.SPEED)
    (start_vx, start_vy) = add_vectors (ship_velocity, bullet_velocity)

    ∞
```
Create a circle with an x co-ordinate of `start_x` *and a y co-ordinate of* `start_y`*. Make its radius* `Bullet.size`*, and make it yellow.*

```
    self.init_mover (dx = start_vx, dy = start_vy)

    self.time_left = Bullet.TIME_TO_LIVE
```
We'll see what this is for in a moment...

Now we're going to write the bullet's `moved` method, which it needs to have for it to be a working `Mover`. We don't want the bullet to keep going across the screen until it hits something: bullets last a fixed time and then peter out. We're going to use the `time_left` variable as a counter of how much time the bullet has left on the screen.

> Write the `moved` method for the `Bullet` class. It should do the following things.
>
> - Ensure that the bullet stays on the screen.
> - Subtract one from `self.time_left`, the counter of how much time the bullet has left to live.
> - If the `self.time_left` is less than or equal to zero then call `self.destroy ()` to remove the circle from the screen.

Now we've got a bullet class, we need to arrange it so that pressing a key fires a bullet. Let's give the `Ship` class a `fire` method which will make it fire a bullet.

This is a method in the `Ship` *class. You've already created that class, so type this inside it.*

```
def fire (self):
  Bullet (screen = self.screen, ship_pos = self.pos (),
   ship_velocity = self.get_velocity (),
   ship_angle = self.angle ())
```

We also need to make some other changes to get this to work.

> To make the bullet work, change the `moved` method of the `Ship` class, which handles key presses, so that when we press `Return`, the ship fires a bullet. `games.K_RETURN` is the key value for the `Return` key.

Try running the program after making these changes. You should be able to fire a bullet by pressing `Return`.

We've now got the asteroids, the ship and the bullets moving around on the screen. You've probably notice that they can all fly through each other at the moment. The next thing we need to look at is what happens when things collide with each other.

Collisions

There are various things which can happen when something collides. Here's a list of the outcome we want from each type of collision.

- Ship and asteroid: ship dies, player loses the game.
- Ship and bullet: ship dies, player loses the game.
- Asteroid and bullet: the asteroid blows up. It may divide into smaller, faster moving asteroids, unless it's already quite small, in which case it just disappears.
- Asteroid and asteroid: nothing (they just pass through each other: this is a bit silly but that's what the original game does).

Let's define a `hit` method for our `Asteroid`, `Ship` and `Bullet` classes. The `hit` method will define what happens when the object is hit by something. We'll give the method the object that hit it as a parameter, as it might want to use the class of that object to decide what to do.

The easiest one is the bullet, so let's do that first:

This is a method for the `Bullet` *class, so you should put it inside the class.*

```
def hit (self, what_hit_me):
  self.destroy ()
```
This removes the `Circle` *from the screen.*

The next one is the ship, which is also easy:

This is a method for the `Ship` *class, so you should put it inside that class.*

```
def hit (self, what_hit_me):
  self.destroy ()
  self.lose_game ()
```
We'll write the `Ship`*'s* `lose_game` *method in a minute.*

In the `hit` method for the `Ship`, we've referred to the `lose_game` method of the `Ship`. We'll make it clear the screen and print "Game Over". We don't restart the game for now.

This is a method for the `Ship` *class, so you should put it inside that class.*

```
def lose_game (self):

  message = games.Text (screen=self.screen,
   x=SCREENWIDTH/2, y=SCREENHEIGHT/2, text = "Game Over",
   size = 90, colour = colour.red)
```

Finally, there's the hit method for the `Asteroid` class. This is a bit more complicated: if what hit an asteroid is another asteroid, nothing should happen. In fact, that's true if what hit the asteroid was the ship, too, because the game will end at that point so there's no point in doing anything.

But if it was a bullet, the asteroid might split two if it's a big one, or disappear if it's a smaller one.

This is a method for the `Asteroid` *class, so you should put it inside that class.*

```
def hit (self, what_hit_me):

  if isinstance (what_hit_me, Bullet):
```
If what hit me is a bullet...

```
    new_size = self.size - 1

    if new_size >= 0:
      ∞
```
Create two new asteroids here: they should both have a size of `new_size`*, and start from the same position as the old asteroid.*

```
    self.destroy ()
```
The old asteroid dies, whether new ones are created or not.

In the `hit` method of `Asteroid`, we've mentioned `self.size`: this is meant to be the current size of the asteroid, between 0 and 2. In the `__init__` method of `Asteroid` we used this number to pick which of the images of the asteroid to use. But we didn't store the number that we used in `self.size`: the `size` variable in the `__init__` method is lost once that method is finished.

> Change the `__init__` method of `Asteroid` so it remembers the size of the asteroid in `self.size`

Finally, we need to make sure these `hit` methods get called. The `Sprite`, `Circle` and `Polygon` classes provide an `overlapping_objects` method which gives a list of other objects which are overlapping that object. We can use this method in the `moved` method of our asteroids, ships and bullets to see whether they've hit anything, and if they have, we can call the ship's, asteroid's or bullet's own `hit` method, and the one for the object it's hit.

In fact, each object will do the same thing when it's hit, which is:

- Run its own `hit` method, passing it the object that hit it.
- Run the `hit` method of the object it hit, giving that method the `self` object (which how an object calls itself).

When we realise that each class of object will do the same thing when it's hit, we should straight away think of putting the code for doing that thing in a method (if we're using classes) or a function, so we only have to write the code once (and we only have to change it in one place if we want to make a change).

> Make a new class, `Hittable`, which contains one method `check_for_hits`. That method should do the following:
>
> - For every object in the `self.overlapping_objects ()` list:
>   - If the object is `Hittable` (look back the `hit` method for the `Asteroid` to see how we test for that) then
>   * Run `self.hit (` the object `)`, to tell ourselves we've been hit.
>   * Run the object's `hit` method, giving it `self` as the thing it's been hit by, to tell the object we've hit it.
>
> Make `Asteroid`, `Bullet` and `Ship` inherit from the `Hittable` class. Make the `Ship` and `Bullet` call `self.check_for_hits ()` in their `moved` methods.

You don't need to make the asteroids call `check_for_hits` because the only things that can hit them are ships and bullets, and ships and bullets are checking for collisions themselves. Since there can be a lot of asteroids on the screen at once, making them all check for hits slows the game down a lot.

If you run your game now, the ship should be able to shoot at the asteroids, and when they're hit, they should split in two until they get too small, at which point they'll disappear. If an asteroid hits the ship, the game should print the "Game Over" message.

We've handled losing, but not winning. We need a way to know when we've won the game. We win by shooting all the asteroids. The easiest way to know when we've won is to make the `Asteroid` class store a number which keeps track of how many asteroids there are. Every time a new `Asteroid` is created, we'll add one to this number. Every time we destroy an asteroid, we'll subtract one from it and check whether it is zero. If it is, we'll call the `win_game` method of `Ship` (which we're going to write).

- Underneath the constants for the `Asteroid` (`SMALLEST_RADIUS` and so on), make a variable called `number_of_asteroids` and set it to zero. This variable will be shared by all the asteroids.
- In the `__init__` method for the `Asteroid` class, add one to `Asteroid.number_of_asteroids`.
- In the `hit` method for the `Asteroid` class, just after we destroy the asteroid, subtract one from the number of asteroids.
- Make a `win_game` method in the `Ship` class which prints a congratulatory message on the screen.
- In the `moved` method of the `Ship` class, check whether `Asteroid.number_of_asteroids` is zero. If it is, the `Ship` should call its own `win_game` method.

We've now got a working game of Asteroids. Congratulations!

Where to next?

If you've got time, you might like to try one or more of these extensions to the game. You might need to look up some things about the `games` module. Sheet W provides a list of all the objects and methods in the module.

- Sometimes the game starts with the player's ship directly on top of an asteroid, so the ship is destroyed straight away. Think of a way to prevent this.
- You could add a score which increases when you hit an asteroid. You can change the text of a `games.Text` object (like the one we used for the "Game Over" message) by using `self.set_text ("your text here")`.
- The original Asteroids game had an alien space ship in it. The alien ship would appear a random times and zig-zag across the screen, shooting at the player's ship. The player gets more points for shooting the alien ship as it's harder to hit because it's zig-zagging. You might like to add this to your game.
- In the original game, the player's ship had a shield. When the player pressed the shield key, the ship was shielded and so wouldn't be destroyed by things colloding with it. There was a limited amount of power in the shield, though, so it would run out after a while. More fuel for the shield would float across the screen sometimes, and the player's ship could pick this up by running into it. (This requires a change to the `hit` method of `Ship`, as hitting the fuel doesn't destroy the ship but rather powers up its shields).
- You could allow the player to restart the game when the ship collides with something, without having to re-run the program. There are lots of ways you could do this. If you make a sub-class of the `Screen` class to use in your game, you can provide a `tick` method which is called every time the screen is updated. That would be a handy place to keep track of whether the player has won or lost (better than doing it in the `Ship` class, in fact, since the `Ship` is destroyed when the player loses).
- You could make the game re-start on a new level with more asteroids when the player destroys all the asteroids on the screen.