Game: Space War

Paul Wright

Revision 1.15, October 7, 2001

Credits

Introduction

This is a Python games worksheet. When you've finished it, you'll have written a version of a classic two-player arcade game called Space War.

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 Space War?

I'll begin by explaining how the game works.

Space war 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 two players' space ships on the screen, along with a planet at the centre of the screen.

In the game, each player guides their ship around the screen, shooting bullets at the other player's ship. If a player's ship is hit by a bullet, that player loses the game.

Each player controls their 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.

Each ship has a limited amount of fuel, some of which is used up every time the ship's thrusters are used. When there's no more fuel left on the ship, the thrusters stop working: the ship can rotate but it cannot thrust.

Everything on the screen feels the gravity of the planet at the centre: if a ship ventures too close to the planet, it will be sucked in and collide with it. If a player's ship collides with the planet, that player loses the game.

If both ships collide, the game is a draw.

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:

- the planet,
- space ships,
- bullets fired by a space ship.

The Livewires games library provides us with some shapes which we can put 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 Planet class

Let's start with the planet, as it's the easiest thing because it doesn't move. 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
```

First, we tell Python we wanted to access the things in the LiveWires games module, using the `import` statement. We also want to use the `math` module, which comes with Python. ("Math" is the American abbreviation for mathematics).

```
SCREENWIDTH=640
SCREENHEIGHT=480

CENTRE_X = SCREENWIDTH / 2
CENTRE_Y = SCREENHEIGHT / 2
```

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 another ship, 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.

We work out a couple of other constants from the width and height of the screen: they're the x and y co-ordinates of the centre of the screen.

```
class Planet (games.Circle):
  RADIUS = 50

  def __init__ (self, screen):

    self.init_circle (screen = screen, x = CENTRE_X, y = CENTRE_Y,
      radius = Planet.RADIUS, colour = colour.grey)
```

Now we create the `Planet` class. The `Planet` class is a sub-class of the `Circle` class. The `Circle` class is provided by the LiveWires games module, so we had to prefix its name by `games.` to tell Python where to find it.

The only method we're defining for the moment is the `__init__` method, which, you'll remember from Sheet O, is called when we create a new object from the `Planet` class. The only thing that the `Planet` needs to know is which screen to be on, which we tell it. It works out where it ought to be using the `CENTRE_X` and `CENTRE_Y` constants, and it has a radius defined inside the class. Planets are made of space rock, which is a sort of grey colour.

If you want to see the planet on the screen, add the following lines to your program:

```
my_screen = games.Screen (width = SCREENWIDTH, height = SCREENHEIGHT)

stars = games.load_image ("stars.bmp")
my_screen.set_background (stars)

my_planet = Planet (screen = my_screen)
my_screen.mainloop ()
```

This will create a `Screen` (that is, a window) for the objects in the game to be in, and then put a `Planet` in the window. The `mainloop` method of the `Screen` class tells Python to start drawing the screen (and to start moving things about on it, as we'll see in a minute).

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

If you run the program, you should see the planet at the centre of the screen. You'll need to close the window to stop the program.

Now we create the `Ship` class, to which the players' ships will belong.

The Ship class

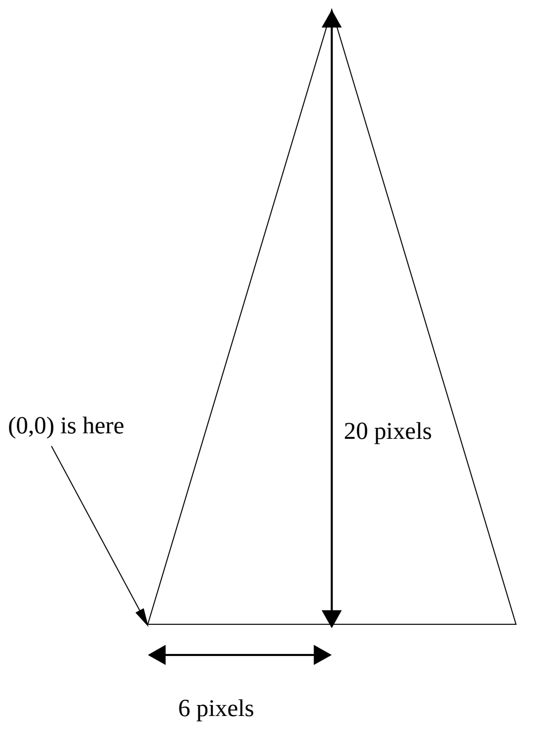
Let's think about the player's space ship. We're going to make the `Ship` class out of the `Polygon` class. A "polygon" is a many-sided shape, like a triangle or a square.

The ship is 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 $(3, 2)$.

[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.



> 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 $(0, 0)$.

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.

Below, 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.

```
class Ship (games.Polygon):
  LIST_POINTS = ( ∞ )                        Put the numbers for the co-ordinates of the ship
                                             between the ( and )
  def __init__ (self, screen, x, y, colour):

                                             The line below will create our triangle.
    self.init_polygon (screen=screen, x=x, y=y,
      shape = Ship.LIST_POINTS, colour = colour)
```

If you want to see what the ship looks like, add the line

```
my_ship = Ship (screen = my_screen, x = SCREENWIDTH/6, y = CENTRE_Y,
  colour = colour.red)
```

after you create a screen at the bottom of your program, but before the `mainloop ()` method is called. This will create a new `Ship` called `my_ship`. If you run the program, you should see a red ship appear to the left of the planet. Again, you'll need to close the window to stop the program.

We want our ships to move around the screen. Luckily for us, the LiveWires Games module provides a class called `Mover` which is for making objects which move themselves around the screen. When we create a `Mover`, we tell it how far in the x and y directions to move each time it moves. Each time a `Mover` moves, it calls its own `moved` method. Inside that method we can do things like check whether the object has collided with anything or has gone off the screen.

- We want the `Ship` to be a sub-class of both the `Polygon` and `Mover` classes. To do this, change the start of the class so we say `class Ship (games.Polygon, games.Mover):` We've now told Python that a `Ship` is both a `Polygon` and a `Mover`.
- We set up the things the `Ship` inherits from the `Mover` class by calling the `init_mover` method. Do this in the `Ship`'s `__init__` method, underneath where we set up the circle. A `Mover` needs to be told how far to move in the x and y directions when it moves, in the `dx` and `dy` parameters. Set the movement in the x direction to zero, and the movement in the y direction to 10.
- We need to make a `moved` method for something which is a `Mover`, otherwise Python will complain about not being able to find one. Make the method underneath the `__init__` method. Inside it, just say `pass` for now: this will tell Python that this method does nothing.

If you run your game after making these changes, the ship should move up the screen from its starting position until it leaves the screen.

Gravity

Now is a good time to add gravity to the game. Gravity causes a change in the *velocity* of the ship. We told the ship its velocity when we created it, in the `dx` and `dy` parameters we gave to the `init_mover` method. Now we're going to want to get the `Ship`'s current velocity, and then change it according to the Law of Gravitation. (The equations we're using were discovered by a man called Sir Isaac Newton, who you might have heard of).

For a `Mover`, the `get_velocity` method either gets velocity and the `set_velocity` method sets the velocity. The velocity is given as a pair of numbers in brackets, eg `(3,2)`, a bit like co-ordinates: the first number is the speed in the x direction and the second one is the speed in the y direction.

Both the ship and the bullets are going to feel gravity, so rather than write out a method to handle gravity for both the `Ship` class and the `Bullet` class we're going to write in a minute, we'll make a new class called `FeelsGravity`. Both the `Ship` and `Bullet` classes will be sub-classes of `FeelsGravity`, so they'll both know how to respond to gravity.

Applying gravity requires some maths, so I've given you the `gravitate` method of `FeelsGravity` and a function which it needs, `add_vectors`. If you really want to know how it works, ask a someone who knows some physics!

```
class FeelsGravity:
  STRENGTH = 2000

  def gravitate (self):
    (x,y) = self.pos ()
    dist_x = CENTRE_X - x
    dist_y = CENTRE_Y - y
    dist_cubed = math.sqrt (dist_x * dist_x + dist_y * dist_y) ** 3

    strength = FeelsGravity.STRENGTH / dist_cubed

    a = (dist_x * strength, dist_y * strength)

    v = self.get_velocity ()
    v = add_vectors (v, a)
    self.set_velocity (v)

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

So, now we can make objects on the screen feel gravity.

> Make the `Ship` class a sub-class of `FeelsGravity` as well as `Polygon` and `Mover`. In the `Ship`'s `moved` method, call `self.gravitate ()`. This will make the ship update its velocity according to gravity every time it moves.

If you run the game now, the ship will respond to gravity: it might plunge into the planet (although we've not told the game what to do when things collide yet, so it'll just go straight through), or it might go into orbit around the planet.

Staying on the screen

Right now, there's nothing stopping the ships from going off the screen. There are a couple of things which we could do when the ship hits the edge of the screen. The ship could come back on the screen at the other side, or it could stop at the edge of the screen. If we stop it at the edge of the screen, we probably want to set its speed in the direction it was trying to leave the screen to zero.

> Decide what you want to do when a ship hits the edge of the screen. Now, write a method in the `Ship` class which will be called every time the ship moves, to check whether the ship has left the screen and to perform whatever action you'd like to do when that happens. Some things you'll need to remember:
>
> - The `pos` and `move_to` methods get and set the ship's position. The `get_velocity` and `set_velocity` methods get and set the ship's velocity. So, you can get the current position by saying `(x, y) = self.pos ()`. You can get the ship's velocity by saying `self.set_velocity ((dx, dy))`.
> - You need to make sure you've handled all four sides of the screen.
> - You need to make sure your new method gets called inside the ship's `moved` method.
> - Hint: you'll probably end up writing things like `if x > SCREENWIDTH` and so on.

Thrust

We want the `Ship` to respond to the player pressing some keys. The way we do this is by getting our `Screen` object, which receives the key presses, to talk to the `Ship` object and tell it that it ought to do something. Luckily for us, the `Polygon` object already has methods to rotate it. We need to 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.

```
def thrust (self):

    angle = self.angle ()

    change_in_v = angle_and_length (angle, Ship.THRUST)
    v = self.get_velocity ()
    new_v = add_vectors (v, change_in_v)
    self.set_velocity (new_v)
```

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.

This gets the ship's velocity.

This sets the ship's velocity.

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

We've already defined `add_vectors`. The `angle_and_length` function is defined below for you to type in. You don't have to understand how it works, but if you know about trigonometry and vectors, you might like to work it out.

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)
```

In the `thrust` method, we also used the `Ship.THRUST` constant to tell the program by how much to increase the speed of the ship. Add the constant to the definition of the `Ship` class, and set it to 1.

Handling the keyboard

Each ship needs to check for key presses every time it moves. In each ship's `moved` method, the ships will ask the `games` module what keys are pressed, and then rotate or thrust if the right key is pressed.

Because each ship will come from the same class, we can't just put the names of the keys directly into the `moved` method, otherwise both ships will respond to the same set of keys. We'll need some way of telling each ship which keys it should respond to.

- Change the `__init__` method of the `Ship` class so that it takes `key_left`, `key_right`, `key_thrust` and `key_fire` as parameters (that's those things in brackets at the top of the function).
- Still in the `__init__` method of the `Ship` class, make the `Ship` remember what values it got for `key_left` by setting `self.key_left` to the `key_left` parameter and so on. If don't do this, Python forgets about `key_left` (and the other parameters) once it finishes the `__init__` method.
- Do the same thing to make the `Ship`'s `__init__` method remember the `key_right` , `key_thrust` and `key_fire` parameters.

Now we can change the `moved` method of `Ship` so that every time if moves it checks whether a key one of the keys pressed and does the right thing.

This goes in the `moved` *method of* `Ship`.

```
  if self.screen.is_pressed (self.key_left):
    self.rotate_by (-4)

  ∞

  ∞
```

Write something to rotate the ship in the opposite direction when the player presses the key to go right.

Write something to call the ship's own `thrust` *method direction when the player presses the key to thrust.*

You'll have noticed we've not covered firing yet: we'll do that in a minute.

Now we need to specify which keys the ship responds to when we create the ship (this happens at the bottom of your program).

- You can refer to keys from A to Z as `games.K_a`, `games.K_b` and so on up to `games.K_z`.
- Change the line which creates a ship so that it gives the ship the keys you want it to respond to, using the `key_left`, `key_right`, `key_thrust` and `key_fire` parameters you added to the `__init__` method of `Ship`.

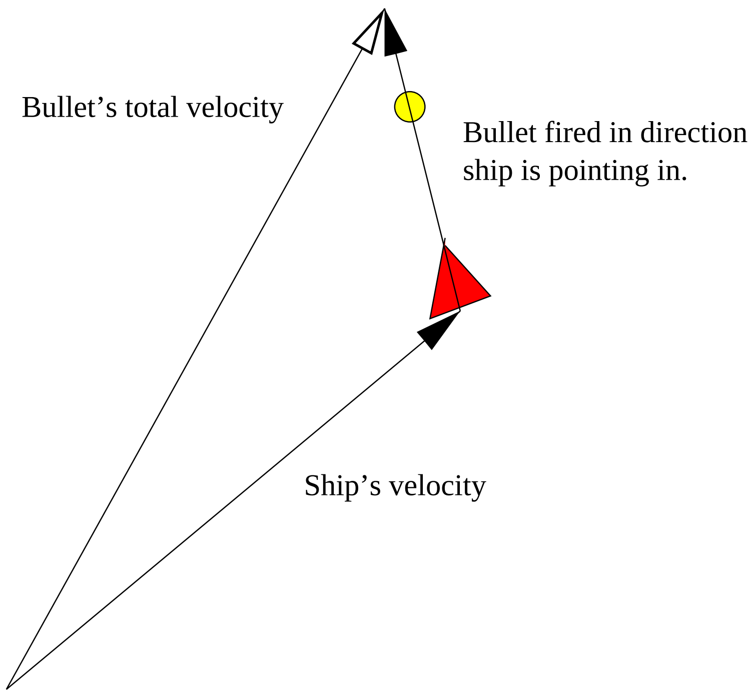
If you run your program you should be able to steer the ship and make it move in the direction in which it's pointing. If you like, you can change your program to create more than one ship and have the other ship respond to different keys.

Next, we'll think about how we can let the ships shoot at each other.

The Bullet class

Bullets are what the ships fire at each other. 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.



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, FeelsGravity):
```

This goes under the GameScreen 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.
>
> - 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 or the bullet has left the screen then call `self.destroy ()` to remove the circle from the screen. (You might want to write a function to tell whether something has left the screen given its x and y co-ordinates).

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 = 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 firing work, you need to change the `moved` method of `Ship`, which handles key presses, so that when we press the key specified in `self.key_fire`, the `fire` method is called.

Try running the program after making these changes. You should be able to make ships fire by pressing the key you've specified as `key_fire` when you created the ship.

We've now got the ships 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 outcomes we want from each type of collision.

- Ship and bullet: the ship is destroyed, that player loses the game. The bullet is destroyed too.
- Ship and planet: the ship is destroyed, that player loses the game. The bullet is destroyed too.
- Ship and ship: the game is a draw.
- Bullet and bullet: both bullets are destroyed.
- Bullet and planet: the bullet is destroyed, but the planet isn't!

Let's define a `hit` method for our `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. Whatever hits the bullet, the bullet is destroyed.

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 a `Circle` *from the screen.*

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

> If what has hit the ship is another ship, the game is drawn, otherwise, the ship that's been hit has lost the game. In any case, the ship is destroyed. Write the `hit` method. The `hit` method of the ship class should take the parameters `self` and `what_hit_me` and do the following:
>
> - If `what_hit_me` is a `Ship`, then call `self.drawn_game`. You can check whether something comes from a particular class by using the Python keyword `isinstance` eg `isinstance (what_hit_me, Ship)`
> - Otherwise, the ship that's been hit has lost the game, so call `self.lost_game`.
> - Remove the ship from the screen.

Finally, we'll need a `hit` method for the `Planet` class. Since we can't destroy the planet, it's easy:

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

```
def hit (self, what_hit_me):
  pass
```

This tells Python to do nothing.

Winning and losing

In the `hit` method for the `Ship`, we've referred to the `drawn_game` and `lost_game` method of the `Ship`. We've not yet defined those methods, so let's do that now.

First the `lost_game` method. We'll just make it clear the screen and print "You lost", in the colour of the ship that was destroyed.

To print a message on the screen, we use the `Text` class which is found in the Livewires Games module.

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

```
def lost_game (self):
  if not self.game_over:
    self.game_over = 1
    message = games.Text (screen=self.screen, x=SCREENWIDTH/8, y=SCREENHEIGHT/2,
      text = "You lost", size = 50, colour = ∞)
```

Make the colour of the text the colour of the dead ship. The `Ship`*'s* `get_colour` *method returns its colour.*

> Write the `drawn_game` method of the `Ship` class. It should print a message on the screen saying the game is a draw. Choose a different colour to that of either of the ships.

Finally, we need to make sure these methods get called. The `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 ships and bullets to see whether they've hit anything, and if they have, we can call the ship'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 it's 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 `Ship` 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 `Bullet`, `Planet` and `Ship` belong to the `Hittable` class as well as the other classes they currently belong to. Make the `Ship` and `Bullet` call `self.check_for_hits ()` in their `moved` methods.

You don't need to make the planet call `check_for_hits` because the only things that can hit it are ships and bullets, and ships and bullets are checking for collisions themselves.

If you run your game now, each ship should be able to shoot at the other ship, and if the other ship is hit, the game should print a message in the appropriate colour. If both ships collide, the game should tell the players the game is a draw. Bullets and ships should be destroyed by hitting the planet.

Fuel

We're almost there now. There's one more thing to add to the game. We've said that each ship has a limited amount of fuel, and once that runs out, it cannot thrust any more. We need to add this to our program.

It'd be good if each player could see how much fuel they had left by looking a bar at the bottom of the screen: the bar should start off reaching all the way across the screen, and then decrease in length as the ship uses up its fuel.

We can make the fuel display bar out of the `Polygon` class we've seen already. This time, it'll be a four sided polygon, or rectangle as they're called. We'll provide an `__init__` method to create the bar, and an `update` method for the ship to tell us how much fuel it has left.

We've got a bit of a problem when we're asked to create a new fuel bar: we can't create it in a fixed position, or the blue ship's bar will be on top of the red ship's. We need the class to keep track of how many fuel bars it's created. We can do this easily, because as well as having constants which are visible to the class, we can also have variables associated with the class as a whole. (In fact, our constants aren't really constants at all as we've said Python won't stop us from changing them during the game if we want: what makes them constants is that we've decided we won't change them, not that Python has).

So, we'll keep track of how many bars we've made with a `num_bars` variable.

```
class FuelBar (games.Polygon):            Our fuel bar is a kind of Polygon.
  THICKNESS = 4                           This will be the height of the bar, as it goes
                                          longways across the screen.
  INIT_POINTS = ( ∞ )                     Put the four co-ordinates for a bar as
                                          wide as the screen and THICKNESS high
                                          here.

  num_bars = 0

  def __init__ (self, screen, start_fuel, colour): We tell the bar which screen
                                          to be on, how much fuel there is
                                          to start with, and what colour to
                                          be.

                                          We work out a number we'll need to
    self.conversion_factor = float (SCREENWIDTH)/start_fuel
                                          multiply the fuel level by, to
                                          convert it to a distance along the
                                          screen.

    y_pos = FuelBar.num_bars * FuelBar.THICKNESS + 1
    FuelBar.num_bars = FuelBar.num_bars + 1

    self.init_polygon (screen=screen, x = 0, y=y_pos, shape = INIT_POINTS,
      fill_colour = colour)

  def update (self, fuel_level):          The ship will use this to update
                                          the fuel display.

    length = self.conversion_factor * fuel_level  Work out new bar length.

    new_shape = ( ∞ )                     Put the four co-ordiantes for a bar
                                          length long and FuelBar.THICKNESS high here.

    self.shape (new_shape)                Change our shape to the new one.
```

We've got a class which displays a fuel bar. Now all we need to do is make the `Ship` class use it.

Make the following changes to the `Ship` class:

- Make a constant inside the `Ship` class to hold the amount of fuel the ship starts out with. Let's call it `START_FUEL`, say. Set it to a couple of hundred for now.
- We'll need a variable to keep track of how much fuel the ship has left. Let's call it `fuel_left`. Set `self.fuel_left` to `Ship.START_FUEL` in the `__init__` method for `Ship`, so it starts out with the right amount of fuel.
- We'll need each ship to have a `FuelBar`. Set `self.fuelbar` to a new `FuelBar` in the `Ship`'s `__init__` method. You'll need to tell the `FuelBar` how much fuel to start out with and also what colour to be. If you look at the `__init__` method, you should be able to work out where to get that information from.
- In the `thrust` method, we should do nothing if `self.fuel_left` is zero. Otherwise, we should work out our new velocity and update it as before, but after that we should subtract one from `self.fuel_left` and then call `self.fuelbar.update`, giving it the amount of fuel left as a parameter.

Now when you play the game, you should find that each ship has a limited amount of fuel. Once that fuel is gone, the ship can no longer thrust, so it's stuck either orbitting the planet or plunging down into it. The ship can still rotate so it may be able to fight the other ship off, but it'll be quite difficult to do this. Adding fuel to the game makes it a bit more interesting.

Conclusion

You've now got a workable Space War game. If you've got time, you might like to think of ways to add to the game, such as adding powerups to enable the player to pick up extra fuel, adding things like homing missiles or mines (which just stay in orbit from where they're laid), and so on. If you need to know more about the `games` module to do this, have a look at Sheet W, where you'll find a list of all the classes and methods in the module.

Have fun.