Game: The Robots Are Coming

Rhodri James

Revision 1.00, August 10, 2007

Credits

Introduction

This is a Python games worksheet. When you've finished it, you'll have written another computerised version of the *Robots* game we introduced in Sheet 5.

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)
- The Robots game (from Sheet 5)
- 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. You don't need to have done all of Sheet 5, as we'll recap the features of the game as we go along.

The Robots Are Coming!

If you remember back in the mists of time when you were just doing Beginners' Programming, the very last worksheet was all about programming a game. In it, the player was being pursued around the screen by robots bent on death and destruction, and had to trick them into running into each other before they could run into him.

In this worksheet, we're going to write the same program, but using the `games` library instead of the `beginners` library. The `games` library does an awful lot of the work for us that we had to do for ourselves in Sheet 5, and you should find it a lot easier to write this version. Once you have, you can add more complications to your heart's content.

Before we start, let's take a moment to write down just what we need from the game.

- a player (to be hunted by the robots)
- some robots (to hunt the player)
- a window on the computer (for the hunt to happen in)

- some way of telling the player how to move
- some way of telling when things hit each other

All these things are easy enough to do with the `games` library.

First Things First

Before we start on the serious bits of the game, we need to tell Python to get all the bits and pieces that we will want to use. It's best to do all this in one place in the program, right at the beginning, so that you can find them later on when you decide to tweak something.

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 random
```

First, we tell Python we want to access the things in the LiveWires `games` module, and the colour names in the `colour` module. We also need the standard library `random`, which contains lots of functions for picking numbers or things out of a list at random.

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

Then we set up some values for how big a screen we will want. We do this here so that we can easily change the size if we want to, just by changing the program at this point. We won't have to hunt through it to find out exactly where we call the function to make the screen, though actually that'll be easy to find – other things won't be, and it's a good idea to get into the habit of collecting all these constant values in one place early on.

The reason we give names to these *constants* is again to make it easier to change things. Suppose that we decide (as we will) to represent the player with a circle of radius 5, but after trying the game out for a while we decide that the circle needs to be bigger. If we were sensible and included a line like `SIZE=5` at the start of the program, all we need to do is change that to `SIZE=6` or whatever we decide. If we weren't sensible, we have to find everywhere that the size of the circle mattered and change the 5 to a 6. We would also need to avoid changing any instances of `5` that *weren't* anything to do with the radius.

Worse, if we have any values that we calculate from the circle size, we might not remember to change them when we change the number we use. If we let the program calculate those numbers for us, it will all work without looking silly. Suppose, for instance, that we decide to put a "GAME OVER" message in the middle of the window when the game finishes:

```
something_to_make_a_message("GAME OVER", 320, 240)
```

If we now make the screen twice as large but don't notice this line of the program, the "GAME OVER" message will turn up in a peculiar-looking place. Worse, if we halve the size of the screen, the message will be disappearing off the bottom right of the window! It's better to let Python do the hard work for use, by saying something like:

```
something_to_make_a_message("GAME OVER",
                            SCREENWIDTH/2,
                            SCREENHEIGHT/2)
```

Notice that we're following a convention for naming our Python variables. Both of the variables we've created so far are constants, ones that we don't intend to change while the program is running. To be distinctive, we give them names that are `ALL_IN_CAPITALS_AND_UNDERLINES`. This gives us a hint that they are constants, and if we change them in the program something is going wrong somewhere. Python doesn't force us to do this, but it makes life a lot easier for someone else reading our program to see what is going on. It's well worth doing; in six months' time, you won't remember what's what yourself!

The Skeleton of a Player

The next thing that we will do is put together the beginnings of our Player on the screen. We'll represent the player by a red circle, with a five unit radius – we can change these later if we decide they don't do quite what we want. We won't, but we could. That immediately suggests two constants that we ought to add to the ones at the top of our program:

```
PLAYER_SIZE = 5
PLAYER_COLOUR = colour.red
PLAYER_SPEED = 1
```

If you remember, in Sheet 5 we introduced the idea of classes as a convenient means of holding all the information about the player and the robots in one place. We're going to do much the same now, except that if you've read Sheet O you'll know that classes are capable of much more than we used them for then.

The `games` library provides a handy class that already does most of the work involved with drawing circles on the screen, and another class that handles objects that move. We want our "player" to be both of these things, or as Sheet O would put it, we want it to be a subclass of both of them. We also want our new player to be put at a random point on the screen, moving in a random direction.

```
class Player(games.Circle, games.Mover):
    def __init__(self, screen):
        self.screen = screen
        x = random.randint(0, SCREENWIDTH-1)
        y = random.randint(0, SCREENHEIGHT-1)
        self.init_circle(screen, x, y, PLAYER_SIZE, PLAYER_COLOUR)
        self.init_mover(random.randint(-1,1) * PLAYER_SPEED,
                        random.randint(-1,1) * PLAYER_SPEED)

    def moved(self):                           Every Mover needs one of these
        pass                                   But we don't know what to do with it yet
```

Notice that we've got another naming convention here. We've given our class a name that has capital letters at the start of the word, and if the name had contained more than one word we would have put capital letters at the start of each word and not used any underlines. Once again, Python doesn't care whether we do this or not, it's just something that we do to make our program more readable.

`random.randint` is a new function to us. As you might guess, it's a function in the `random` library that comes with Python as standard, and it picks a *rand*om *int*eger (whole number) for us. In fact you have seen `randint` before; we hid it in the `beginners` library under the name `random_between`!

Finally, we had to write a little function in `Player` called `moved`. Every `Mover`-type object needs to have a function called `moved`, which gets called after the object has been moved on the screen. We will make a lot of use of that later, but for now we just tell Python not to do anything (or `pass`, as a Mastermind contestant might say).

The skeleton of a game

Once we have the beginnings of a class for our Player, we need a window to show the player off in. The `games` library has a class for doing this, the `Screen` class. We will want to do one or two extra things, though, so we'll make our own subclass of it.

```
class RobotGame(games.Screen):
    def __init__(self):
        self.init_screen(SCREENWIDTH, SCREENHEIGHT)
        self.set_background_colour(colour.white)
        self.game_over = 0
        self.player = Player(self)
```

This tells Python that our `RobotGame` class is a kind of `Screen`, makes sure that the screen is started up correctly, turns it white, sets the `game_over` flag to remind us that the game hasn't finished yet, and creates a `Player` to move around the window.

All we need to make this into a complete Python program is to add two more lines to our file. We need to tell Python to make us a `RobotGame` and then do all the work of running the window for us.

```
game = RobotGame()
game.mainloop()
```

And that's it. If you now save your program and run it, you will see it open a new window on the screen with a red blob in it that rapidly disappears off the screen in a random direction.

Moving About

The next thing we ought to do is something to keep the player from disappearing off the screen. This is nice and straightforward, if a bit tedious; all we have to do is to check if the player has gone off the edge of the and move it back on. As a courtesy we should change the velocity of the player so that it stops trying to move off the screen.

These are all things we do after the player object has moved, and we already know a function that gets called every time it moves. So you'll need to replace the little stub of a function that we wrote for `moved` earlier with the following:

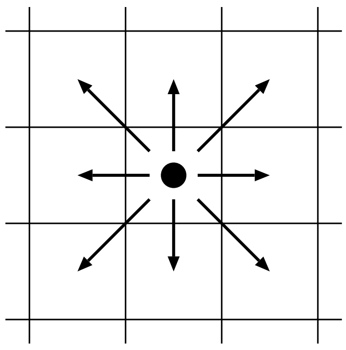
```
                                      This goes in the Player class
    def moved(self):                  Make sure that the lines starting ``def'' line up
        if not self.screen.game_over:
            (x, y) = self.pos()
            (dx, dy) = self.get_velocity()

            if x < 0:                 Have we gone off the left?
                x = 0
                dx = 0
            if x >= SCREENWIDTH:      Have we gone off the right?
                x = SCREENWIDTH-1
                dx = 0
            if y < 0:                 Similarly for top & bottom
                y = 0
                dy = 0
            if y >= SCREENHEIGHT:
                y = SCREENHEIGHT-1
                dy = 0

            self.move_to(x, y)        Set the updated position and speed
            self.set_velocity(dx, dy)
```

If you run the program now, the red blob stops at the edge of the screen.

You may find the choice of `dx` *and* `dy` *as names for the x and y components of the velocity a bit confusing. It's something that you'll find more common if you do more advanced mathematics or physics; mathematicians tend to use the word "delta" when talking about the change in something. Here, the x and y parts of the velocity of the player are the changes (delta) in its* `x` *and* `y` *position, so we call them* `dx` *and* `dy` *for short.*

Controlling the Player

Now that the player's blob is on the screen the whole time, we can work on controlling exactly where it goes. Just like in Sheet 5, we'll allow movement in 8 directions …



… and allow the player to stand still as well, and we'll do all this by using the "numeric keypad" on the right of your computer keyboard. So, for example, we want the number 1 on the keypad to make the blob move down and to the left.

Unlike Sheet 5, we do this by changing the player's *velocity* (speed) rather than its position, and once again the convenient place to do that is in the `moved` function that we've just added to. Put the following lines in *before* the calls to `self.move_to` and `self.set_velocity`:

```
                                        Remember to line this up!
if self.screen.is_pressed(games.K_KP1):
    dx = -PLAYER_SPEED                  Change the velocity to move left…
    dy = PLAYER_SPEED                   … and down
```

Notice that "down" is the positive direction here, unlike the way we normally draw graphs and the like. Sadly that's just the way most computer graphics work, and we haven't hidden it from you the way we did in the `beginners` *library.*

If I tell you that the number 2 key on the key pad is called `games.K_KP2` by the library, can you figure out how to get the blob moving in all the directions?

> On my computer, the player moves smoothly but slowly across the screen. Try changing the value of `PLAYER_SPEED` to find a speed that suits you — fast enough to be fun, but not impossible. Then make a note of what you've got and set it back to `1` so that you can test the rest of the program!

I, Robot

That's enough of the player for now, let's turn our attention to the robots. First the easy decisions: we'll make our robots be black square blocks, ten units on each side to match the size of the player. The `games` library gives us the `Polygon` class which allows us to build squares, but we need to tell it where each corner of the square is. All in all, that gives us a few more constants to add to the start of the program:

```
ROBOT_SHAPE = ((-PLAYER_SIZE, PLAYER_SIZE),
               (PLAYER_SIZE,  PLAYER_SIZE),
               (PLAYER_SIZE, -PLAYER_SIZE),
               (-PLAYER_SIZE, -PLAYER_SIZE))
ROBOT_COLOUR = colour.black
ROBOT_SPEED = PLAYER_SPEED
```

Notice that the place we consider the robot to be is the centre of this square that we've defined, and that we've been a little bit clever to make sure that the robots and the player are of similar sizes.

Beyond that, the things that we need to do to start up a robot are quite similar to the things that we need to do to start up a player. We pick a place to put it on the screen, draw it there, and start it moving in the right direction.

```
class Robot(games.Polygon, games.Mover):
    def __init__(self, screen, player):
        self.junk = 0
        self.screen = screen
        self.target = player
        x = random.randint(0, SCREENWIDTH-1)
        y = random.randint(0, SCREENHEIGHT-1)
        self.init_polygon(screen, x, y, ROBOT_SHAPE, ROBOT_COLOUR)
        (dx, dy) = self.determine_velocity()
        self.init_mover(dx, dy)

    def moved(self):                            Again, we need a trivial one of these temporarily
        pass

    def determine_velocity(self):               Just for now, we'll have a trivial one of these too
        return (0,0)                            Make the robot stand still
```

Then we just need to add one line to the `__init__` routine of our `RobotGame`:

```
                                                Line it up right!
        self.robots = Robot(self, self.player)
```

If you run this now, you'll get a black square sitting on your screen not moving, in addition to the moving red blob.

This looks familiar

You may have noticed that we've done a few things in very much the same way for both robots and players. For one thing, we pick a random spot in the window in exactly the same way for both of them, by calling `random.randint` twice.

We can take advantage of this to shorten our program a bit by putting this little bit of code into a function of its own:

```
def random_location():                          This is outside all of the class definitions
    x = random.randint(0, SCREENWIDTH-1)
    y = random.randint(0, SCREENHEIGHT-1)
    return (x, y)
```

Then we just replace both of the places where we pick a random location with

```
(x, y) = random_location()
```

This may not look like it saves us much (and indeed it doesn't), but if you remember Sheet 5 you might remember that we'll be calling that function a few more times yet. Besides, it's good practice at spotting when we can turn "doing the same thing" into a common function.

Homing in

At the moment, our robot is a bit of a dummy when it comes to homing in on the player. It could at least start out moving in the right general direction to find its target.

To fix that, we need to write the `determine_velocity` function properly, so let's stop and think for a minute about how to do that. If you remember from Sheet 5, the first thing we have to do is to work out where the robot and the target are, and then think about where they are relative to each other. If the target is to the left of the robot, the robot needs to move left. If it's above the robot, the robot needs to move up, and so on. From that, we can work out what velocity (i.e. what direction) we want the robot to be going at.

Since all of the objects on the screen have a method (a fancy name for a function belonging to them) called `pos` which tells you where they are, that list of things to do translates fairly simply into Python instructions:

Replace ''determine_velocity'' with this

```
def determine_velocity(self):
    (x, y) = self.pos()
    (tx, ty) = self.target.pos()
    (dx, dy) = (0, 0)
    if x < tx:
        dx = ROBOT_SPEED
    if x > tx:
        dx = -ROBOT_SPEED
    if y < ty:
        dy = ROBOT_SPEED
    if y > ty:
        dy = -ROBOT_SPEED
    return (dx, dy)
```

This starts the robot heading off towards the player's blob, but it doesn't follow when the player goes off in a different direction. To do that, we need to correct our course at the end of each move. We can do that in the `moved` function, just like we did for the `Player` responding to key presses. Thinking ahead a little, we don't want to do that when we've turned into junk or when the game is finished. The changes we need to make should be obvious from the what we did in the `__init__` function in the first place.

A new ''moved'' for the ''Robot'' class

```
def moved(self):
    if not self.junk and not self.screen.game_over:
        (dx, dy) = self.determine_velocity()
        self.set_velocity(dx, dy)
```

We could have written those last two lines as just one line:
`self.set_velocity(self.determine_velocity())`
since `set_velocity` *wants two values, and* `determine_velocity` *returns two values. We just wanted to make it a little more obvious what was going on.*

Terminator Conditions

So now our robot chases around after the player, but when it catches him neither of them even notice. We need some way of working out whether the robot has hit the player or not. We'll do this by defining a brand new class of "things which can be hit by other things" and have it do the figuring out for us.

Checking for collisions is something that the library makes fairly easy for us. We can ask for a list of all the objects on the screen that overlap with our object with the function `overlapping_objects`. We can go through that list looking for other objects that can be hit (which will be all of them for me, but we might have some things in future versions of our program that can't be hit) and causing them and ourselves to do whatever we should when we're hit.

Finding out if an object "can be hit" is the same as asking if it's a type of our "can be hit" class. The way to ask that question in Python is with the `isinstance` function: `isinstance(` *obj, class*`)` asks if *obj* is a member of the class *class*. Armed with that, we can make our new class.

```
class Hittable:
    def check_for_collisions(self):
        for o in self.overlapping_objects():
            if isinstance(o, Hittable):
                o.collided()
                self.collided()
```

> Now you have to make both `Player` and `Robot` subclasses of `Hittable`. Can you remember how to do that?

Junking robots

Our new `Hittable` class means that all the hittable things must have a function called `collided`, otherwise our program will moan incessantly when it notices that two objects have collided. Let's think about what happens to robots who collide.

From the basic description of the game, any robots that collide with each other will turn into junk. That's nice and simple: we flip the flag that says "we're junk, don't do anything", stop the robot moving, and change it to look like junk.

First, we'll define what junk looks like. We'll cook up a little shape that looks like a small pile with smoke coming off it. If you squint.

Add these to the other constants

```
JUNK_SHAPE = ((-PLAYER_SIZE, PLAYER_SIZE),
              (PLAYER_SIZE, PLAYER_SIZE),
              (PLAYER_SIZE/5, 0),
              (PLAYER_SIZE-1, -PLAYER_SIZE/2),
              (PLAYER_SIZE/5, 1-PLAYER_SIZE),
              (-PLAYER_SIZE/5, 1-PLAYER_SIZE),
              (PLAYER_SIZE/2, -PLAYER_SIZE/2),
              (-PLAYER_SIZE/5, 0))
JUNK_COLOUR = colour.grey
```

After that, our `collided` routine is easy!

```
def collided(self):                       Add this to the ‘‘Robot’’ class
        if not self.junk                  If we aren’t already junk . . .
            self.junk = 1               . . . become junk                 :
            self.set_velocity(0, 0)
            self.set_shape(JUNK_SHAPE)
            self.set_colour(JUNK_COLOUR)
```

Junking the player

Obviously when something hits the player, the game is over. What exactly should we do then? The simple answer is that we raise the flag that we carefully set aside earlier to say that the game is done. Since we're thoughtful people, we also want to put up a message commiserating with the player for becoming robot fodder. While this is all about what happens after the player is hit, it's logically something do to with the game, not with the player as such, so we'll do the work in the `RobotGame` object.

The library gives us the `Text` object to display messages with. That just needs to be told what to say, where to say it, and what size and colour to use. We'll put the message in the middle of the screen, in blue, and 48 points high.

Add this to the ‘‘RobotGame’’ class

```
    def lose_game(self):
        self.game_over = 1
        games.Text(self, SCREENWIDTH/2, SCREENHEIGHT/2,
                   "GAME OVER: YOU LOSE", 48, colour.blue)
```

Yes, the "magic number" 48 and the colour `blue` *should really be named constants like the all the other changeable things. Why don't you do that?*

> So all we need to do when the player has collided with a robot is to stop the player moving (set its velocity to (0,0)) and tell the screen that we've lost the game. Can you write those three lines of Python?

So are we junk then?

Finally, `Hittable` objects won't check whether things have collided by magic. We must call `check_for_collisions` from somewhere, generally the `moved` method of anything that's `Hittable`. In fact for our case, it's enough that we check when robots move. All we have to do is to add `self.check_for_collisions()` to the end of the `Robot` class `moved` function.

> So do that then!

The Rest Of The Robots

Having only one robot doesn't give us much of a game. Now it's time to add more.

Let's say that we want five robots, at least for now:

```
NUMBER_OF_ROBOTS = 5
```

This is a constant . . .

Currently we make just the one robot and assign it to `self.robots` in the `RobotGame` class. We need to change that to making a list of robots, adding each new robot to the end of the list. Can you remember how to do this? We have to start with an empty list in `self.robots`, and *for* each *robot* in the *range* zero to `NUMBER_OF_ROBOTS` we need to *append* a brand new *Robot* to the list.

> With that heavy hint, you can write the three lines of program to start playing with more robots.

If you try this out, you'll notice that the robots head off into the sunset once the game is over. We could let them do that, but it's tidier and avoids some very silly problems if we make all the robots stop instead when the player loses. That means adding a couple of lines to our `lose_game` function to run through the list of robots telling them all to stop.

> We've told things to stop moving before, so you should know how to do that. If you can't remember how to work your way through a list, check through Sheet A. It's only two more lines of Python!

Initial placement

If you try this out lots of time (particularly if you try it out with lots and lots of robots), you'll notice that sometimes the robots appear on the screen on top of each other and crash before they've even had a chance to move. Worse yet, sometimes the robots appear on top of the player and kill him immediately. This doesn't make for a very fun game.

What we need to do is to check where our robots' start position is, and pick a different position if it's going to cause a crash straight away. If that sounds like we're going to use a loop to you, then have a pat on the back; we are. After we've created our robot, we can use the same trick that the `Hittable` class does to see if the robot has already collided with something, using the `overlapping_objects` function in the library. Notice that we have to create our robot's `Polygon` object to do this, otherwise the library doesn't know what bit of the window to check. Notice also that we can't determine the direction we want the robot to go in (its *velocity*) until we know where it is.

That means we need to add the following lines of code to our `Robot` class `__init__` function, in between the call to `init_polygon` and `determine_velocity`:

While there are some overlapping objects . . .

```
while self.overlapping_objects() != []:
    (x, y) = random_location()      ... try again
    self.move_to(x, y)
```

If we want to be even more clever, we can keep robots from showing up within a short distance of the player. We'll write ourselves a little function to make that easier, returning 1 (*true*) if the robot is too close to the target and 0 (*false*) if it isn't.

The fast and inaccurate way of checking if a robot is too close is just to make sure that the difference between both the x-coordinates and the y-coordinates of the robot and its target are big enough. This difference can be positive or negative, and difference that is big enough in either direction will do. To help us, Python has a function called `abs` which gives us the *absolute value* of a number, i.e. the number itself if it's positive, or minus the number if its negative (so `abs(-1)` is 1).

Armed with that information, we'll write a new function for our `Robot` class.

Line this up with all the other ''defs''

```
def is_too_close(self, target):
    (x, y) = self.pos()
    (tx, ty) = target.pos()
    if (abs(x - tx) < DISTANCE and abs(y - ty) < DISTANCE):
        return 1
    return 0
```

We'll need to create our constant `DISTANCE` and set it to, say, 30, but you already know how to do that. All you then need to do is to call `is_too_close` in the condition of your `while` statement:

This replaces the previous ''while''

```
while (self.overlapping_objects() != [] and
       self.is_too_close(self.target)):
```

. . . and you're done!

> This "quick and dirty" method of finding how far apart the robot and the target are is good enough for our purposes. However, if you want to do it properly you could use *Pythagoras' Theorem* to find out what the distance really is. If that doesn't ring any bells, ask a leader.

> If you want to be more generous to the robots, you could make sure that they start up further apart too. See if you can work out how to do that using the tools and functions you've already put together.

Now Get Out Of That

If you remember Sheet 5, you'll have noticed that there's something the player can't do yet. If you aren't familiar with Sheet 5, you may be wondering how the player ever wins because unless you're lucky, the robots can always pin you against one edge of the window.

The answer is that the player gets to teleport! We want to watch out for the player pressing a particular key — say, the `T` key — and then move the player to a new random location.

That's easy to do. See if you can guess where to put this new part of the program.

Indent this to match its surroundings

```
        if self.screen.is_pressed(games.K_t):
            (x, y) = random_location()
             dx = random.randint(-1, 1) * PLAYER_SPEED
             dy = random.randint(-1, 1) * PLAYER_SPEED
```

And that's all. If you want, you can work a little bit harder and make sure that the player's blob doesn't appear on top of a robot. If you're feeling really enthusiastic and very generous, you can make sure that the player doesn't appear too near to any robot, but that's a lot more work. We prefer to be mean and let the player take his chances when teleporting.

Winning the Game

So now the program knows how lose, and the player has a decent chance of crashing all the robots into each other now that he can teleport about the place. The only thing missing is that the game doesn't yet spot when the player *has* tricked all the robots into dying, and has therefore won. It's time to straighten that out.

Looking at what we've just said, it seems that the player will have won the game when all the robots have turned into junk. When that happens we want to stop everything moving and do something similar to what we do in `lost_game` to cheer the player on. We can combine this and cheat a bit, using the nature of functions to get out of doing any more work when we find a non-junked robot.

Add this to the `RobotGame` *class*

```
    def check_for_win(self):
        for robot in self.robots:
            if not robot.junk:             Is this robot not junk?
                return                     Yes, bail out now!
        self.win_game()                    If we are here, all robots are junk
                                           Stop everything from moving
        self.player.set_velocity(0, 0)     (Only the player is left to move!)
```

So when do we need to check for a win? We could do it in any of several places, such as after the player moves. However, there's only any chance that the game is over after a robot is turned into junk. The easy thing to do, then, is to call `self.screen.check_for_win()` at the very end of the `Robot` class `collided` function.

So what are you waiting for?

> You also need to write the `win_game` function. That's easy though; it's exactly the same as the `lose_game` function except for the message you show to the player, and also you don't have to stop the robots moving because they are already not moving!

That's it. You now have a fully functioning game of Robots!

Bells and Whistles

There are several things you could do to make your game more fun.

- At the moment, the symbols for the player, active robots and junk are a bit blobby. That's what you get for letting me do your design work for you. You could take some time to come up with better icons to represent the different things on the screen. If you're feeling very adventurous, you could ask a leader about `Sprites`.
- In the same spirit, you could display little explosions when robots collide. Ask a leader about `Animations` and the pictures of an explosion we have ready and waiting.
- At the moment, once you've won or lost the game it's all over. You could change your program so that once the game is over, the program waits for a short while then `destroys` everything on the screen and restarts. *Note: you'll need to ask a leader about exactly what you need to do*

- If you've done that, the next thing you could do is to increase the number of robots on the screen every time that the player wins.
- If you do that, you could keep a score for the player of how many levels they've got through. You can display that with a `Text` object like the messages in `lose_game` and `win_game`. Have a look at Sheet S as well to see how to put numbers into a text message.
- Alternatively, you could create new robots every time a robot turns into junk, and keep score of how many robots have died.
- Think of other things you could add to the game, such as randomly appearing bombs or zones the player can't go into. Be inventive!