Game: Scramfender

Rhodri James Revision 1.15, 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 horizontal scrolling game we call *Scramfender*.

What you need to know

- The basics of Python (from Sheets 1 and 2)
- Functions (from Sheet 3; it will help if you 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 Scramfender?

Scramfender is an amalgam of bits and pieces. It was originally intended to be a version of an old arcade game called *Defender*. It turned out that what was being remembered was actually an old arcade game called *Scramble*, except that a couple of bits of Defender had crept in. Since the end result is still a fun game, we've written up the worksheet anyway.

The game consists of a spaceship flying through a cavern. The player has to navigate his ship through various obstacles that turn up, such as missiles erupting from the cavern floor, UFOs that spit out bullets in all directions, and little ships that flit up and down and get in the way of the player.

The ship can move up and down, and can accelerate forwards a little or fall back as it moves inexorably through the cavern. It can also fire bullets to try to clear its path. It scores points for every opposition ship destroyed. If it collides with the cavern floor or ceiling, or hits any of the other ships or bullets, then it is destroyed and the game ends.

There are quite a lot of different types of enemy ship that can turn up. We won't bother describing them all just yet; they each have different characteristics in how they move and what they do, and we'll leave that until we get round to implementing each of them.

Things on the screen

From the above description, you can see that there are quite a lot of different things that can end up on the screen.

- Our space ship.
- The cavern.
- Bullets fired by the ship.
- Bullets fired by enemy ships.
- Enemy space ships (lots of them).

The LiveWires games library provides us with a way of putting shapes on the screen, and also a way of telling them to move across the screen by themselves rather than us having to keep moving them ourselves. These things are provided as classes, as you might expect from Sheet O, and we are going to make sub-classes of these useful shape classes to make the objects we actually want.

Making our graphics window

The very first thing we need to make our game is a window to play it in. 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 that we want to access the things in the LiveWires `games` module, and also the colour names supplied in the `colour` module. We also want to use the `random` module that comes as part of the Python standard library, so that we can generate some random numbers.

```
SCREENWIDTH=640
SCREENHEIGHT=480
FLIGHT_SPEED=1
```

Then we set up two values, `SCREENWIDTH` and `SCREENHEIGHT`, to hold the width and height of our graphics window, and a third value, `FLIGHT_SPEED`, which will control how fast we fly through the cave system. We did this so that we can easily change the width or height or speed if we want to, just by changing the program at that point. If we'd used the numbers 640 (for width) and 480 (for height) by themselves all over the place and we later decided that 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, and change only the right ones. This is a lot of work, and a lot of opportunities to mistype the number and get something very strange happening, so instead we define some variables so that we can refer to the height and width by name. 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 AND UNDERLINES. 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 ScramScreen(games.Screen):
  def __init__(self):
    self.init_screen(SCREENWIDTH, SCREENHEIGHT)
    self.set_background_colour(colour.black)
    self.game_over = 0
```

Then we tell Python we want to make a class called `ScramScreen` that is a kind of (*i.e.* a sub-class of) the `Screen` class from the LiveWires `games` library. We tell it what size the screen is, ask it colour it in black, and set up one further useful flag that we will eventually use to remind us that the game has finished.

> *Can you remember what the `__init__` method of a class does? If you can't, go back and read Sheet O again — it's quite important!*

Notice that we've made two more rules for how to name things here. First, our class name has capital letters at the start of each word, which makes it a bit easier to read. Second, our other names for variables (and later functions) are all in lower case letters with underlines between the words. Once again, Python doesn't force us to do this, but it does make it easier to figure out exactly what is going on. That may seem easy now, but when you come back to your program in six months' time it suddenly becomes a lot more difficult!

```
my_screen = ScramScreen()
my_screen.mainloop()
```

Finally, we tell the program to actually create a graphics window and start everything running. It's very short, and there's no more to the main program than that because the library hides everything away in the `mainloop` function!

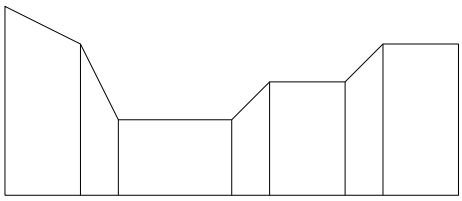
If you run your program now, you'll get a black window with nothing in it. Not very exciting, but it's a start.

The Cavern

The next thing for us to put in is the cavern through which our ship will fly. For simplicity's sake we'll start off by just putting in the floor.

At first, this may seem like a pretty tall order. Scramble is an example of a type of game we call a *horizontal scroller*, meaning that we have an awful lot of cavern floor that our spaceship flies over, that zigzags up and down all over the place. That sounds like a horribly complicated polygon with lots and lots of points.

A complicated piece of terrain



The same terrain as a series of parallelograms

Fortunately there's an easier way to do things. As you can see from the picture above, instead of thinking of our cave floor as one huge complicated lump, we can break it down into lots of individual pieces which are much easier to handle. What we'll do is to chop the terrain into strips (called *parallelograms* because two of their sides are parallel), and we'll do the chopping at the points where the slope changes.

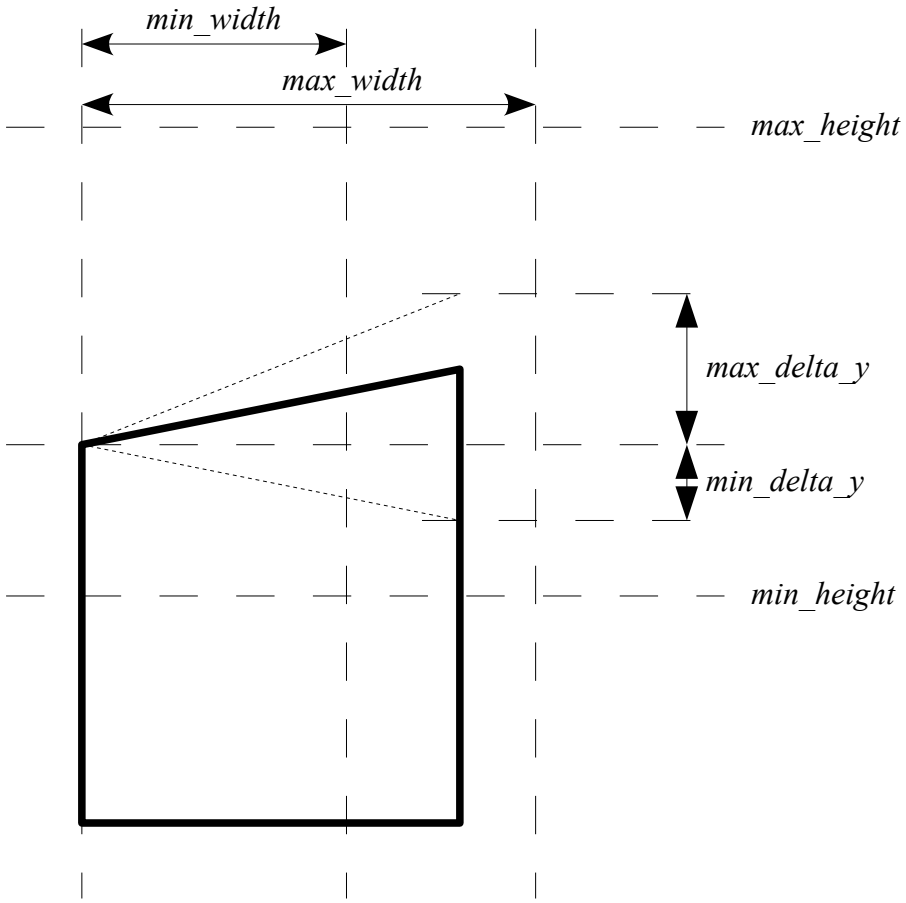
These strips are pretty easy for us to define. The bottom corners will be at the bottom of our screen. The left-hand side

corners will be in the same place as the right-hand side corners of the previous strip (though we'll have to do something special for the very first strip we make). We'll make the width of the strip random, though we'll have to decide what the thinnest and thickest strips we want will be. And finally, we'll make the height of the right-hand top corner random as well, though again we'll have to decide what the limits are.

All these decisions that we're going to make about limits will affect what the cave floor looks like. If our strips are fairly wide or not very different in height from one to the next, then the cave will look smooth; if they are narrow and very different in height, then it will be jagged and harder to negotiate. Thinking ahead, we'll allow ourselves to change these limits as time goes by, so that our cave can look different to make different challenges.

That implies that we have two different sorts of things here. We have the individual pieces of cave floor, and we also have the general nature of the cave which we need to record so that we can refer to the limits when we make a new piece of cave. In Python terms, that means two different classes. Let's start with the overall cave:

```
class Cave:
  def __init__(self, screen, is_floor):
    self.min_width = 6                        Set some default values
    self.max_width = 20
    self.min_delta_y = -10
    self.max_delta_y = 10
    self.min_height = 10
    self.max_height = 100
    self.screen = screen
    self.is_floor = is_floor
    # More to follow...
```



So far, this just collects together a load of limits and other useful things. The diagram will help you understand what we are going to do with these limits; `min_width` and `max_width` will (eventually) control how thin or fat our pieces of cave are, and `min_height` and `max_height` will be similarly the tallest and shortest it can be. You might be a little confused by `min_delta_y` and `max_delta_y`, however. We'll use these values to control how different the heights of the left- and right-hand sides can be — like scientists and mathematicians, we use the greek letter delta to mean a difference or change in something, in this case our y-coordinate. The numbers that we've picked here are intended to mean that our end-point can be up to ten units lower than the start (because `min_delta_y` is `-10`), or up to ten units higher (because `max_delta_y` is `10`).

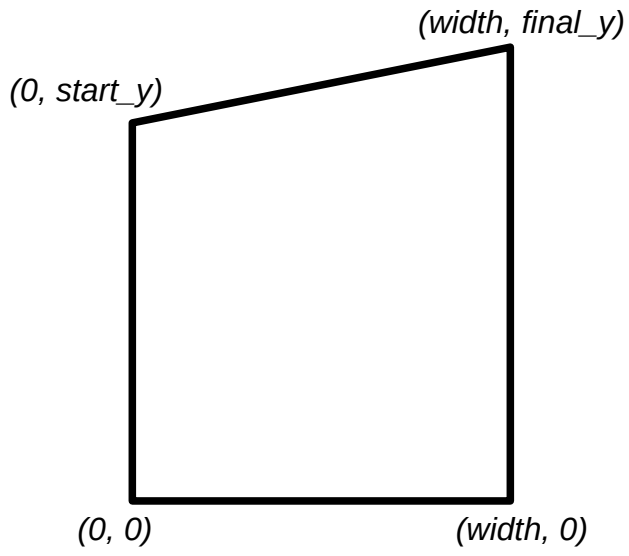
Of course, this doesn't make any actual cave on the screen yet. To do that we need our strips:

```
class CavePiece(games.Polygon):
  def __init__(self, screen, cave, previous, is_floor):
    if previous is None:
      (start_x, start_y) = (START_X, START_Y)
    else:
      (start_x, start_y) = previous.get_end_pos()
    self.start_y = start_y
                                                    Pick a random width, within limits
    self.width = random.randint(cave.min_width, cave.max_width)
                                                    Pick a random height change, within limits
    delta = random.randint(cave.min_delta_y, cave.max_delta_y)
    self.final_y = start_y + delta
                                                    Make sure the new height is also within limits
    if self.final_y < cave.min_height:
      self.final_y = cave.min_height
    if self.final_y > cave.max_height:
      self.final_y = cave.max_height

    # Adjust for screen coordinates
    y_base = SCREENHEIGHT
    start_y = -start_y
    final_y = -self.final_y

    self.init_polygon(screen, start_x, y_base,
                      ((0,0), (0, start_y),
                       (self.width, final_y), (self.width, 0)),
                      colour.grey)
    self.cave = cave
    self.is_floor = is_floor
```

We're declaring our strips to be something that the `games` library calls a `Polygon` – a many-sided shape. The library needs to know what window it's dealing with, a starting point in the window for the shape, what lines to draw relative to the start, and what colour to fill it in with. The picture below should help you to see what the lines are; that's the only tricky bit.



Our strip — our `CavePiece` as we should call it now — gets told about the window that it's going to appear in (`screen`) so that it can be displayed, the `Cave` that it's part of (`cave`) so it can find out what its limits are, the last strip (`previous`) that it's got to join on to, and whether it's part of the floor or the ceiling (`is_floor`). We're thinking ahead with that last one; obviously we're going to want to do pretty similar things for the ceiling of the cave as we're currently doing for the floor, so we're probably going to end up using our `CavePiece` class again.

Some of this may look a little bit strange. Why do we set `y_base` to the height of the screen, when we want to make a

piece of the floor? This is because for historical reasons, our graphics coordinates are upside-down. Unlike the way you draw something on graph paper, our `(0,0)` (the *origin*) is in the top left corner, not the bottom left, and our y-coordinate increases as we go down. That's why we make `y_base` the full height of the screen, and make `start_y` and `final_y` negative numbers to move back up the screen.

The other thing you might be wondering is what the `random.randint` function is. In fact it's something you've seen before in the beginners course, only there we renamed it as `random_between` to make it a little more obvious what it's doing. `randint` is its real name, because it picks a *rand*om *int*eger (whole number). You'll be seeing a lot more of it as we add more to the program.

If you've been paying attention, you'll have noticed that we used two more constants. We'd better define them:

Put this near the top of the program

```
START_X = 100
START_Y = 30
```

Remember that we made a note to ourselves to do something special for the very first piece of cavern that we make? These constants are that "something special". If we haven't got a previous piece of cave (something we'll signal by setting `previous` to the special Python value `None`), we use those constants to tell us where to start. We could have make `START_X` zero rather than a hundred, this just makes it a bit more obvious that we're at the start.

"What is it that we're doing if we do have a previous strip?" you may be asking. We said before that if we have a previous piece of cave, we need to know where its top right corner is so that we can use it as our new top left corner. We find that out by asking it, or in other words by calling its `get_end_pos` function. We'd better write that, then. Put this in the `CavePiece` class:

Put this after the `__init__` function

```
  def get_end_pos(self):
    return (self.xpos() + self.width, self.final_y)
```

Nice and simple. `self.xpos()` asks for the x coordinate of our strip; the way we've written it, that's actually the x coordinate of the left-hand side of the strip, so we need to add the width of the strip to find the right-hand edge.

> *Because we're only going to make one strip to start with, just to test that we've got everything right, we aren't actually going to call* `get_end_pos` *just yet. Because Python is what's called an* interpreted *language, it won't pay any attention to things we don't use, so we could have got away without writing it. We will need it eventually, though, and its good practice to write the function now while we know what we want it to do, rather than later when we might have forgotten.*

Now we need to get our `Cave` to make the first strip of cave floor. To do that, we add one line to our `__init__` function for the `Cave` class:

```
self.pieces = [ CavePiece(screen, self, None, is_floor) ]
```

This tells the program to make a list, putting one brand new `CavePiece` in it.

Finally, we need to create our `Cave` or nothing will happen. We do this by adding a line to our `ScramScreen` class's `__init__` function:

```
self.floor = Cave(self, is_floor=1)
```

And that's it. Try running the program now and seeing what happens.

One Becomes Many

So we've got a single piece of cave floor now. That may fill us with pride, but it's only the one piece. What we really need is a whole string of strips stuck side-by-side to make a cave floor. Let's spend a moment writing down what our program needs to do in words, rather than in Python. Programmers call this *pseudocode* because its sort-of like program code, but not really.

```
Have we run off the end of the screen?
If we have, stop
If we haven't:
  Find the last strip that we created
  Make a new strip next to it
  Add the new strip to the list of strips
  Go back and try again
```

This looks like a loop of some sort. Since we're wanting to stop when a condition is fulfilled instead of after a particular number of times, it's going to be a `while` loop rather than a `for` loop. Let's rewrite our pseudocode with that in mind, and while we're at it we'll think a little harder about what "running off the end of the screen" means.

```
while the last strip's end position is on the screen:
  make a new strip next to the last strip
  add the new strip to the list of strips
```

The last strip we created is also the last thing we added to the list of strips. This might make us scratch our heads a moment; we know how to find the first item in a list, or the second, or even the fifty-third, but how do we find the last item in a list?

If you check Sheet A, you'll discover that actually Python makes this very easy for us. So easy, in fact, that we're going to make you look it up for yourselves; when you see the infinity symbol ∞, put the answer in.

Our three lines of pseudocode actually turn into two lines of Python, because we can combine making the new strip with adding it to the list. We just need to add these two lines to our `Cave` class `__init__` function:

Put this at the end of the function

```
while self.pieces[ ∞ ].get_end_xpos() < SCREENWIDTH:
  self.pieces.append(CavePiece(screen, self,
                     self.pieces[ ∞ ], is_floor)
```

> The `while` statement uses another new function `get_end_xpos` to tell us where the x coordinate of the end of the cave piece is. We could have used the `get_end_pos` function that we wrote earlier and thrown away the y coordinate that gives us as well as the x coordinate, but that would have been a bit untidy. Besides, `get_end_xpos` is really, really easy to write. You shouldn't need our help to figure it out!

Once you've done all that, run the program and luxuriate in your cave floor.

Once More With Ceiling

So that's the floor. Now how do we do the ceiling?

It turns out that it's quite easy. All we're really doing is turning our strips upside-down and hanging them down from the top of the screen instead of up from the bottom. If you look carefully at how the `CavePiece` `__init__` function works, all this means is being careful about choosing `y_base` — starting our y coordinates from zero instead of `SCREENHEIGHT` — and making `start_y` and `final_y` positive numbers (down the screen) instead of negative numbers (up the screen).

All we have to do is change the lines setting `y_base` and its friends so that they look like this:

```
                                          Remember to line this up correctly
    if is_floor:
      y_base = SCREENHEIGHT               What we had before
      start_y = -start_y
      final_y = -self.final_y
    else:
      y_base = 0                          New stuff
      final_y = self.final_y
```

. . . and it will all just work. The only other thing we have to do is to make a cave ceiling:

```
self.ceiling = Cave(self, is_floor=0)
```

You can find where to put that line yourself! Run the program, just to be sure you've got it right.

Flying Through The Cave

We have a cave all right now, but we wanted to fly through it. The way we do that might surprise you, and never mind that we don't have a spaceship yet.

Everything is relative, as Einstein might have said if Newton hadn't beaten him to it. They weren't talking about their aunties; what they meant was that you can look at things from different points of view. We normally think about flying through a cave from the point of view of someone standing on the cave floor. From there, the spaceship is moving through the cave. From the point of view of the spaceship, however, it's the cave that is moving.

We're going to move our cave as well, pushing our strips from right to left across the screen to make it look like the spaceship is moving from left to right. That way we can deal with the spaceship dodging and jiggling about without having to worry about its steady progress through the caves.

As you might hope, the `games` library gives us a lot of help in making things move. It provides us with an entire class, `games.Mover`, to do all the hard work. That means that we have to tell our program that our `CavePiece` class is a kind of `Mover` as well as a kind of `Polygon`. We have to change the class declaration line so that it looks like this:

```
class CavePiece(games.Polygon, games.Mover):
```

We also have to tell the `Mover` class to do its own start-up things. In particular, we have to tell it what speed to move in what direction. We want the bits of cave to move to the left, which means leaving the y coordinate alone and subtracting something from the x coordinate. Subtracting one seems to work well enough, which is why right at the start we conveniently set `FLIGHT_SPEED` to one so that we could use it here. You can change that if you want, but to start with that works well enough.

The upshot of all this is that we need to add this line straight after the call to `self.init_polygon`:

```
self.init_mover(-FLIGHT_SPEED, 0)
```

> *Just so that you know, what the* `Mover` *class does is to change the position of our cave piece every fiftieth of a second. It might not actually manage to do it that fast if it has a lot of things to do, but that's the aim.*

This isn't quite all. The `games.Mover` class insists that every moving object has a method (a function in its class) called `moved` that will be called every time the object moves. Just for now we don't want to do anything after our cave moves, so we'll tell it that there's nothing to do. Add this to the `CavePiece` class:

```
                                          Line up the defs
  def moved(self):
    pass
```

Try running that and see what happens.

Adding More Cave

The problem with what we've got at the moment is that once all of our original cave floor and ceiling have whizzed off the screen, there isn't any more. We need to check after the pieces have moved to see whether we need to add a new strip to the cave.

How do we do that? Let's write it out in pseudocode again.

```
if the last strip's end position is on the screen:
  make a new strip next to the last strip
  add the new strip to the list of strips
```

Does that sound familiar? It should do; it's the same as what we were doing to make enough cave strips in the first place, except that we say "if" instead of "while".

> Knowing this, can you write a new function called `maybe_add_piece` to the `Cave` class? Beware: if you just cut and paste what you've written before, it won't work. This is because two of the variables that we used before, `screen` and `is_floor`, were parameters to the `__init__` function that won't be parameters to your new function. Fortunately we saved copies of them away just in case something like this happened, so if you write `self.screen` and `self.is_floor` instead all will be well.

The only problem is that the `Cave` class doesn't know that its pieces have moved; only they know that. That's easily fixed; all we have to do is to change our `moved` function so that instead of telling Python that we're going to `pass`, we tell it instead to call your new function. We can do that because we've kept a copy for each `CavePiece` of the `Cave` that it's part of, so we can call its functions. The replacement line you need is:

```
self.cave.maybe_add_piece()
```

Tidying Up

What we currently have looks great, but if you leave it running for long enough you'll find it starts to slow down and grind to a halt. This is because the pieces of cave that have slipped off the left-hand side of the screen still exist, and are still being moved left even though we can't see them any more. Eventually this becomes so much work that the library hasn't finished moving all the sections before it should have started the next move.

It's easy enough for us to get rid of strips that have disappeared off the screen; in fact it's very much the same sort of thing that we had to do to add new pieces. In words, what we have to do is to check if the first strip in our list is off the side — in other words if the right-hand edge has an x coordinate less than zero — and if it has, we both take the strip off the list and "remove" it from the screen.

```
                                          Remember to line things up correctly
def maybe_remove_piece():
  if self.pieces[0].get_end_xpos() < 0:
    self.pieces[0].destroy()              Remove from screen
    self.pieces = self.pieces[1:]         See Sheet A
```

You can figure out where to put this function, and where to call it from!

Ship Ahoy

Enough of the cave. It's all very satisfying for us to have created our scrolling cave, but it's not much fun for the player yet. The player needs a ship before he can do anything.

We'll represent our ship as a bright yellow triangle, with the point towards the right. As far as the `games` library is concerned, this is a `Polygon` just like our cave pieces were. It will need to move too, making it a `Mover` as well. We'll start it off stationary, halfway up the screen and as far left as we're going to let it go.

```
class Ship(games.Polygon, games.Mover):
  SHAPE = ((10, 0), (-10, -5), (-10, 5))
  def __init__(self, screen, min_x, max_x):
    self.screen = screen
    self.init_polygon(screen, min_x, SCREENHEIGHT/2,
                      Ship.SHAPE, colour.yellow)
    self.init_mover(0,0)
    self.min_x = min_x
    self.max_x = max_x

  def moved(self):
    pass
```

Again, we need to actually create the ship in our `ScramScreen` class. Add the following line to its `__init__` function:

```
self.ship = Ship(self, START_X, SCREENWIDTH/3)
```

If you later decide that we aren't letting the ship go far enough left or on the screen, this is the line to change. When we later let the ship move itself, we'll stop it going any further left than the first number (the one we've currently set to `START_X`), or any further right than the second number (a third of the way across the screen at the moment).

Moving The Ship

So let's get on with moving the ship. We want to do something slightly different to normal: we will move up and down if the player presses the "up" and "down" arrow keys in the normal way, but we'll move left and right in a slightly odd way. If the "right" arrow key is pressed, we'll move the ship to the right until it's reached its maximum rightwards position. If the "right" arrow key *isn't* pressed, we'll let the ship drift back to the left until it gets as far left as we're going to let it go.

To find out whether a key is pressed at the moment or not, we can call the function `is_pressed` from the `Screen` class. We tell the function which key we want to know about by giving it a constant. The `games` library gives us a whole host of useful constants for this. For our purposes, all you need to know is that the up arrow key is represented by `games.K_UP`, the down arrow by `games.K_DOWN`, and the right arrow by `games.K_RIGHT`.

> We don't need it, but can you guess what the left arrow is called?

When do we worry about moving the ship? Clearly the ship knows where it is and how it's moving at the moment. Conveniently, the ship's `moved` method gets called every time the ship moves (or stays still for that matter), so we can use that. At the moment all it does is `pass` anyway, so delete the `Ship` class `moved` function and replace it with this:

Make sure all your `def`*s line up*

```
  def moved(self):
    (x, y) = self.pos()
    if self.screen.is_pressed(games.K_UP):
      y = y - VERTICAL_SPEED
    if self.screen.is_pressed(games.K_DOWN):
      y = y + VERTICAL_SPEED
```

These lines do the vertical movement. First we get the current position of the ship. We do this at the beginning because

otherwise we'll find ourselves doing it all the time, and that's not a very efficient thing to do.

Then if the "up" key is pressed, we adjust the `y` position by subtracting some value (remember that `(0,0)` is in the *top* left corner, not the bottom left like you might expect). Similarly if the "down" key is pressed, we add to the y coordinate.

Moving horizontally is a bit more involved:

This `if` *should line up with the other* `if`*s above it*

```
    if self.screen.is_pressed(games.K_RIGHT):
      if x < self.max_x:
        x = x + HORIZONTAL_SPEED
    else:
      if x > self.min_x:
        x = x - 1
```

If the "right" key is pressed, then if we aren't already at or past the maximum that we've set for ourselves we'll add something to the `x` position. Otherwise, if we aren't already at or past the minimum that we've set for ourselves we'll subtract one from the x coordinate. Notice that we're only subtracting one this time; no matter how fast we choose to accelerate forwards, we're only going to drift back to our minimum spot slowly.

The last thing our new `moved` function must do now that we've worked out where we want our ship to be is to move it there:

```
    self.move_to(x, y)
```

That's it, except for the two new constants we've created. You'll need to set them at the top of the file; try a value of 2.

Hitting Things

So far so good, but if you play around with what we've done you'll find that you can make the ship disappear into the cave roof or floor. This is no good at all; the ship should explode when it hits the cave!

This is a problem that we're going to have again and again. All sorts of different things are going to need to know when they hit something; not just the player's ship caring about the cave, but enemy ships being hit by bullets, the player's ship being hit by other ships, and so on. When a lot of different objects are react to the same sort of thing, even if they react a bit differently, you ought to think about whether they all belong to some sort of class which you can turn into a real Python class.

In this case, we can think of the ships, bullets and bits of cave as being "hittable", so we'll create a class called `Hittable` which will do all the basic, common work for us.

What does this `Hittable` need to be able to do? Well, it must check to see if the `Hittable` object has hit any other `Hittable` object. In terms of objects on the screen, one hits another when the two objects overlap. This might sound like it's a bit difficult for us to figure out, but fortunately the `games` library does all the hard work for us. It has a function called `overlapping_objects` that returns a list of all the objects which overlap with the thing that's asking. We can easily check that list to find out if anything in it is hittable.

Make this the first class in the file

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

In English, we go through each object that's overlapping us, and if it's `Hittable` we tell it that it's hit us, and we tell ourselves that we've hit it. Clearly we'll need to make our `CavePiece`s and our `Ship` into `Hittable` objects. You should be able to do that on your own. *Hint:* do you remember how you made `CavePiece` a `games.Mover` as well as a

`games.Polygon`?

So what do our objects do when they're hit? Let's take the our bits of cave first. Apart from getting a little scratched, they're not going to care much about the subject. They're a lot bigger than a spaceship, after all. So all we do is add this to the `CavePiece` class:

Keep the `def`*s in line!*

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

For the ship, being hit is a lot more serious. In fact, it's the end of the game. Add this to the `Ship` class:

And again, line the `def`*s up*

```
def hit(self, what_hit_me):
  self.screen.lost_game()
  self.destroy()
```

This removes the ship from the screen and lets the main `screen` object know that it's all over. The screen can then do something useful like display a gloating message and stop anything else from happening. Try adding this to your `ScramScreen` class:

Don't forget to line this up correctly too

```
def lost_game(self):
  if not self.game_over:
    self.game_over = 1
    games.Text(self, SCREENWIDTH/2, SCREENHEIGHT/2,
               "Game over, you lost", 50 colour.white)
```

There's just one last thing to do; find a time to check whether any collisions have happened. Clearly our cave doesn't care much, so we don't need to worry about collisions from its point of view. Our ship cares a lot, though, so we should check every time it moves that it hasn't exploded in fiery death. Or in other words, we should add this line to our `Ship` class `moved` function:

```
self.check_for_hits()
```

First Enemies

So far we can fly our ship through the cavern and avoid crashing into the walls. That's not actually much of a challenge, though. Let's make our game a little more difficult by adding some enemy ships.

The first enemies we'll create are nice simple critters called "Bees" because they buzz up and down making a bit of a nuisance of themselves. They'll give you a nasty sting if you hit them, but otherwise they don't do anything. On the screen they appear as red boxes – not very inspiring, but easy to draw.

Let's define a few constants to help us handle our Bees.

Up near the top, remember?

```
BEE_SIZE = 10
BEE_LIMIT = 100
BEE_VARIANCE = 100
BEE_MAX_SPEED = 10
```

Clearly from the description our Bees are squares (a kind of `Polygon`) which move (*i.e.* are a kind of `Mover`) and are `Hittable`. We'll want to start them off somewhere on the screen, and set upper and lower limits for where they can go. This is the sort of playing with random numbers that we've seen before. That means it isn't too hard for us to start writing our new `Bee` class:

```
class Bee(games.Polygon, games.Mover, Hittable):
  SHAPE = ((-BEE_SIZE/2, -BEE_SIZE/2), (BEE_SIZE/2, -BEE_SIZE/2),
           (BEE_SIZE/2, BEE_SIZE/2), (-BEE_SIZE/2, BEE_SIZE/2))
  def __init__(self, screen, start_x):
    # Set some sensible upper and lower limits
    self.min_y = BEE_LIMIT + random.randint(0, BEE_VARIANCE)
    max_y = BEE_LIMIT + random.randint(0, BEE_VARIANCE)
    self.max_y = SCREENHEIGHT - max_y
    # Then pick a starting point between them
    start_y = random.randint(self.min_y, self.max_y)
    speed = random.randint(-BEE_MAX_SPEED, BEE_MAX_SPEED)
    self.init_polygon(screen, start_x, start_y,
                      Bee.SHAPE, colour.red)
    self.init_mover(-FLIGHT_SPEED, speed)
```

Because it's a `Mover`, we have to give it a `moved` function. This must check to see if it has reached the maximum or minimum height and reverses direction if it has. It also needs to check if it has fallen off the left hand edge of the screen, just like our strips of cave did. Finally, it needs to check if the Bee has been hit.

Make sure the `def`*s are indented the same amount*

```
  def moved(self):
    y = self.ypos()
    if y < self.min_y or y > self.max_y:
      (dx, dy) = self.get_velocity()
      self.set_velocity(dx, -dy)
    if self.xpos() < -BEE_SIZE:
      self.destroy()
    self.check_for_hits()
```

Why do we call the x and y parts of the velocity `dx` *and* `dy`*? It comes from scientists' and mathematicians' habit of using the Greek letter delta for a change in something again; speed is a change in position over time, after all. It's actually a little more convoluted than that, but you don't need to worry about that until you come to do calculus!*

Finally, because it's `Hittable`, our `Bee` needs to have a `hit` function. Bees just die when they are hit; that's easy.

Do we really need to say it?

```
  def hit(self, what_hit_me):
    self.destroy()
```

Now all that remains is for us to actually put some Bees on the screen. To do that, we only need to add a few lines to our `ScramScreen` class's `__init__` function:

This has to line up with what went before too

```
    for x in range(2*SCREENWIDTH/3, SCREENWIDTH, 40):
      Bee(self, x)
```

This creates a new Bee every 40 pixels across the right hand third of the screen. Try it and see.

There's one gotcha with this setup that you probably won't have noticed. Try setting `BEE_LIMIT` to 0 temporarily and see what happens. You'll find that some of the Bees bash themselves against the cave floor or ceiling and die. That's not what we want to happen at all! It may not look like a problem if we put `BEE_LIMIT` back to normal, but if we ever get round to making our cave narrower (which we will eventually) we'll be back to looking silly again.

There are several ways of fixing this. For example, we could just ignore it when our Bees hit the cave, but that would look silly too, even if we made up a story about the Bees having some weird alien technology that let them 'phase' through the floor and ceiling.

What we'll actually do is a little bit cleverer — we'll move our Bees to stop them colliding with the cavern in the future,

by adjusting their maximum and minimum positions. In order to do that we'll need to know how tall the bit of cave we've collided with is. We don't need to know how we figure that out yet, we can ask the `CavePiece` that we've collided with to figure it out for us.

So we have to replace the `hit` routine we just wrote with this:

Keep the same indentation

```
def hit(self, what_hit_me):
  if isinstance(what_hit_me, CavePiece):
    height = what_hit_me.get_height()
    if what_hit_me.is_floor:
      # Remember, y gets bigger as we go down the screen
      self.max_y = SCREENHEIGHT - (height + BEE_SIZE/2)
      self.move_to(self.xpos(), self.max_y)
    else:
      self.min_y = height + BEE_SIZE/2
      self.move_to(self.xpos(), self.min_y)
  else:
    self.destroy()
```

This isn't perfect — we'll get more collisions as our Bee speeds past its end-stop — but it looks good enough for our purposes.

So how do we find out what height our piece of cave is? For our purposes we'll be close enough if we just take the bigger of our starting and ending y-coordinates. That's easy to write:

Add this to your `CavePiece` *class*

```
def get_height(self):
  if self.start_y > self.final_y:
    return self.start_y
  return self.final_y
```

Much better.

Shooting at the Enemy

Now we've got some enemies, we'd better figure out how to fight our way through them. Or in other words, how do we make our bullets work?

Bullets are basically small circles moving across the screen, ideally hitting enemy ships and blowing them up. If that description rings a few bells about what sort of classes we'll use for our `Bullet` class, it should; apart from details of what they look like and exactly how they move, bullets sound a lot like all the other moving things we've created so far!

There is one small wrinkle that we want to introduce; we want our bullets to die off after a while instead of when they disappear off the screen, so that we can limit the reach of our ship's guns. Well, that's our excuse anyway. We'll need to tell our bullets where to start, how fast to move in which direction, and how long it should hang around. While we're at it, we'll also tell them what colour to be.

```
class Bullet(games.Circle, games.Mover, Hittable):
  def __init__(self, screen, x, y, dx, dy, col,
               live_time=BULLET_LIFETIME):
    self.init_circle(screen, x, y, BULLET_SIZE, col)
    self.init_mover(dx, dy)
    self.time_left = live_time

  def moved(self):
    self.check_for_hits()
    self.time_left = self.time_left - 1
    if self.time_left <= 0:
      self.destroy()

  def hit(self, what_hit_me):
    self.destroy()
```

This may look a bit excessive. After all, aren't all of our bullets going to be heading away from our ship at a constant speed and with a constant colour? For now they are, but secretly we know that we're planning to let some enemy ships fire back at us. We might as well write our `Bullet` *class in a general way now, rather than go back and do it later.*

We fire bullets by creating them when the player presses a particular key. Let's make that key the space bar, the handy constant for which is `games.K_SPACE`.

Think carefully about where to put this

```
if self.screen.is_pressed(games.K_SPACE):
  (x, y) = self.pos()
  Bullet(self.screen, x + 17, y, BULLET_SPEED, 0, colour.yellow)
```

That just leaves us with a few constants to define:

```
BULLET_SIZE = 2
BULLET_SPEED = 10
BULLET_LIFETIME = 50
```

And now our ship fires happily at the enemy!

Slowing it down

The only problem with this setup is that our ship fires rather quickly. We can fix that by forcing our `Ship` to wait a few ticks before checking to see if the space bar is pressed. To do that, we'll need to count ticks of our internal clock (or, equivalently, the number of times our ship's `moved` function gets called), which means we'll need a variable to count ticks in. We'll also need to initialise that variable, so add this line to the `Ship` class `__init__` function:

```
self.bullet_delay = 0
```

Then when we want to think about shooting, we first check to see if we're delaying after a previous bullet, and if so we subtract one from our delay counter.

This replaces the `if self.screen.is_pressed(...)` *line*

```
  if self.bullet_delay > 0:
    self.bullet_delay = self.bullet_delay - 1
  elif self.screen.is_pressed(games.K_SPACE):
    self.bullet_delay = SHIP_BULLET_DELAY
```

. . . then the rest of the firing stuff we typed earlier

Experiment a bit to see what value of `SHIP_BULLET_DELAY` works best for you. Setting it to 5 seems to work nicely, but try it and see for yourself.

More Bees

So now we can shoot up our initial collection of Bees, then fly through the rest of the cave. After the initial excitement, that's a bit of a let-down. What we really need is to make more Bees from time to time.

The way we'll do that is to count ticks of our internal clock again, but we won't use our `Ship`'s `moved` method to do it. Making new enemies is really something for our overall `ScramScreen` class to do, since that's what makes the original batch. Unfortunately we don't have anything like `moved` to use there, do we?

Well, actually we do. The library will call a function `tick` in our `ScramScreen` once every "frame" (the same rate that it calls `moved` at). We can use that to decide when to create a new Bee.

> *Hang on a second. If the library calls this* `tick` *function all the time, why hasn't it been complaining every time we've run our program?*
> *The answer is that the* `games.Screen` *class secretly has its own* `tick` *function that does nothing, and the library will call that function if we don't define our own* `tick` *function.*

First we should set up come controls so that our `tick` function will have something to work with. We need to add the following two lines to the `ScramScreen` class `__init__` routine.

Aligned correctly, of course

```
    self.bees_active = 1
    self.bee_time = random.randint(30, 100)
```

Again, we're thinking ahead a little. We may (and will) decide to stop producing Bees for bits of our cave system.

Then we have to write the `tick` function itself:

Line it up with the other `def` *statements*

```
def tick(self):
    if self.game_over:
        return                          Bail out if the game is over
    if self.bees_active:
        self.bee_timer = self.bee_timer - 1
        if self.bee_timer <= 0:
            Bee(self, SCREENWIDTH)
            self.bee_timer = random.randint(30, 100)
```

More Enemies

It's time to introduce a new type of enemy, before we get too comfortable blowing Bees out of space. Now we'll try making some missiles that erupt from the cave floor to try to shoot down our spaceship. We'll make them triangles like our spaceship, only pointing upwards rather than forwards.

The first tricky thing that we have to decide is where the bottom of our missile should be. We want it to be sitting on the cave floor, as best we can given that the cave floor isn't level. That involves asking the floor, which in turn has to hunt through its list of cave pieces for the one that's in the right place. At least we know how to find out where a piece ends from the `maybe_` functions that we wrote earlier: that's what we wrote `get_end_xpos` for!

Since we know that our list of cave pieces is in left-to-right order, it's quite easy for us to find the piece that our x-coordinate is in. All we have to do is look through the list for the first strip that ends *after* the spot we're interested in.

Add this to the `Cave` *class*

```
  def get_height(self, x):
    for piece in self.pieces:
      if piece.get_end_xpos() >= x:
        return piece.get_height()
```

Now we've figured that out, we can write the first part of our new `Missile` class. Don't be too surprised if some of it looks familiar!

```
class Missile(games.Polygon, games.Mover, Hittable):
  SHAPE = ((-5, 0), (0, -20), (5, 0))
  def __init__(self, screen, ship, floor, start_x):
    start_y = SCREENHEIGHT - floor.get_height(start_x)
    self.init_polygon(screen, start_x, start_y,
                      Missile.SHAPE, colour.blue)
    self.init_mover(-FLIGHT_SPEED, 0)
    self.target_ship = ship
    self.is_moving = 0
```

That last bit may look a little odd. We set the missile travelling horizontally along, then set a variable which in effect says that the missile *isn't* moving. How can that be?

From the point of view of the missile, though, it isn't moving. It's just sitting on the cave floor, waiting for the right moment to shoot upwards and (it hopes) destroy our ship. We only think of it as moving because we're looking at it from the point of view of the ship. Everything is relative, remember!

So what is the right moment for the missile to take off? Let's define that the missile rises up as fast as the cave scrolls left (`FLIGHT_SPEED`, in other words). That means that for every space the missile goes up, the ship comes one space closer horizonally. So if the spaceship is flying steady and level (*i.e.* the player isn't pressing any keys), the missile needs to take off when the ship is as far left of it as above it. Then it's up to the player to move out of the way!

Translating all that English into Python code, we get to add this function to our new `Missile` class:

In line with the other `def` *statements*

```
  def moved(self):
    if not self.moving:                    # If we haven't taken off yet
      (tx, ty) = self.target_ship.pos()
      (mx, my) = self.pos()
      if (mx - tx) <= (my - ty):
        self.set_velocity(-FLIGHT_SPEED, -FLIGHT_SPEED)
        self.moving = 1
    elif self.xpos() < -10:                # If we've fallen off the screen
      self.destroy()
    else:
      self.check_for_hits()
```

Again, bits of that should look familiar. Bits of the `hit` function will look familiar too, since missiles should explode when they hit something. The only exception is that missiles shouldn't care about hitting (or rather sitting unevenly on) the cave floor, since it isn't really hitting it as such. It's just a side-effect of the fact that the floor isn't smooth and level, while the bottom of our spaceship is.

Line this up consistently

```
  def hit(self, what_hit_me):
    if not (isinstance(what_hit_me, CavePiece) and
            what_hit_me.is_floor):
      self.destroy()
```

Now all we have to do is to create some `Missile`s both to start with and as the cave goes on.

> That's will look a lot like the way we created our Bees. Try for yourself creating one `Missile` to start with three quarters of the way across the screen, then keep adding another one at the edge of the screen at a random time between 30 and 100 timer ticks later. If this doesn't look *very* like what you did for Bees, as a leader what you've done wrong!

> If you're interested, you could try to make our missiles look slightly better "seated" on the ground than they currently do. That will involve you doing two things: first, working out exactly what height the cave is under the middle of the missile, and allowing for how far below the centre of the missile (its notional position) its bottom is.

Changing the Cave

We've got two different kinds of enemy ships trying to kill us. That's enough for the moment; let's turn our attention back to making the cave itself more interesting.

We would like to change the characteristics of the cave after a while. We could do that by fiddling with the `max` and `min` variables our floor and ceiling keep directly, but that's a bad idea. If we ever changed how our `Cave` worked, we'd find ourselves having to hunt down the places outside the class that knew how it used to work and change them individually. Essentially, it's the same reason that we use our `CONSTANTS` instead of plain numbers.

It's a better idea not to let anything outside a class fiddle with its internals, and provide a set of functions to do the fiddling instead. This means that if we change how `Cave` works, we only need to change the functions and not all the other places in our program.

Here are the functions to control our `Cave` settings:

Line this up with the previous `def`*s*

```
def set_width_limits(self, min_width, max_width):
  self.min_width = min_width
  self.max_width = max_width

def set_height_limits(self, min_height, max_height):
  self.min_height = min_height
  self.max_height = max_height

def set_change_limits(self, min_delta, max_delta):
  self.min_delta_y = min_delta
  self.max_delta_y = max_delta

def set_limits_to_default(self):
  self.min_width = 6
  self.max_width = 20
  self.min_delta_y = -10
  self.max_delta_y = 10
  self.max_height = 100
  self.min_height = 10
```

Now that we've defined `set_limits_to_default`*, it's a good idea to call it rather than just set the variables in our* `__init__` *function. That way if we decide to change the default values of* `min_width` *and so on, we only have to do it in one place.*

Given that, let's suppose that we want to make the cave narrower and more jagged after a full screen of it has scrolled by, and at the same time stop making any more missiles. If that sounds like we should be counting down another timer and doing something when it reaches zero to you, then congratulations on paying attention! That's exactly what we'll do. First we need to set it up:

This goes in the `__init__` *function*

```
self.phase_timer = SCREENWIDTH
```

Then we need to count it down:

This goes in the `tick` *function*

```
self.phase_timer = self.phase_timer - 1
if self.phase_timer <= 0:
  self.end_phase_1()
```

The function that makes those changes we talked about is also pretty easy:

A new function in the `ScramScreen` *class*

```
def end_phase_1(self):
  self.ceiling.set_height_limits(100, 200)
  self.ceiling.set_change_limits(-20, 20)
  self.floor.set_height_limits(100, 200)
  self.floor.set_change_limits(-20, 20)
  self.missiles_active = 0
```

This is a little bit messy, because we'll keep calling `end_phase_1` once the timer expires. However, that only slows us down a bit, and we'll fix it in a minute. Try it now, just to make sure it works.

Changing the Cave some more

Suppose now that after we've been through another screen's width of buzzing bees, we want to make the cave a bit wider again, lose the Bees and have missiles instead. We could create another counter, add another couple of lines to `tick`, and do something to stop our first timer interfering with it. That's a bit horrid, though, and gets more and more horrid as we want to add a fourth, fifth and sixth section to the cave system.

Fortunately, there is a neater way. As you might remember from Sheet F — you did read Sheet F when we suggested it, didn't you? No? Not to worry, we can wait while you do.

OK, as you might remember from Sheet F, as far as Python is concerned, a function is just another object. That means that we can do anything with it that we'd do with any other object. In particular, we can store it in a variable, and then later treat that variable as if it was the object itself.

In other words we can add this line to our `__init__` routine:

```
self.end_phase = self.end_phase_1
```

...and later treat `end_phase` as if it was `end_phase_1`. In particular, when the `phase_timer` runs out in `tick`, we can change what we do to:

```
self.end_phase()
```

...and Python will execute `end_phase_1` for us just like before. Try it and see!

This may seem like a lot of strangeness for no change, but we're not done yet. We can add the following two lines to our `end_phase_1` function:

In line with the rest of the function, of course

```
self.phase_timer = SCREENWIDTH
self.end_phase = self.end_phase_2
```

...and hey presto, when the `phase_timer` runs out the second time it will call `end_phase_2` for us! So we'd better

write that then:

More for the `ScramScreen` *class*

```
  def end_phase_2(self):
    self.ceiling.set_height_limits(50, 120)
    self.floor.set_height_limits(50, 120)
    self.bees_active = 0
    self.missiles_active = 1
    self.phase_timer = 2 * SCREENWIDTH
    self.end_phase = self.end_phase_3
```

In the interests of completeness

```
  def end_phase_3(self):
    self.ceiling.set_limits_to_default()
    self.floor.set_limits_to_default()
    self.bees_active = 1
    self.phase_timer = SCREENWIDTH
    self.end_phase = self.end_phase_1
```

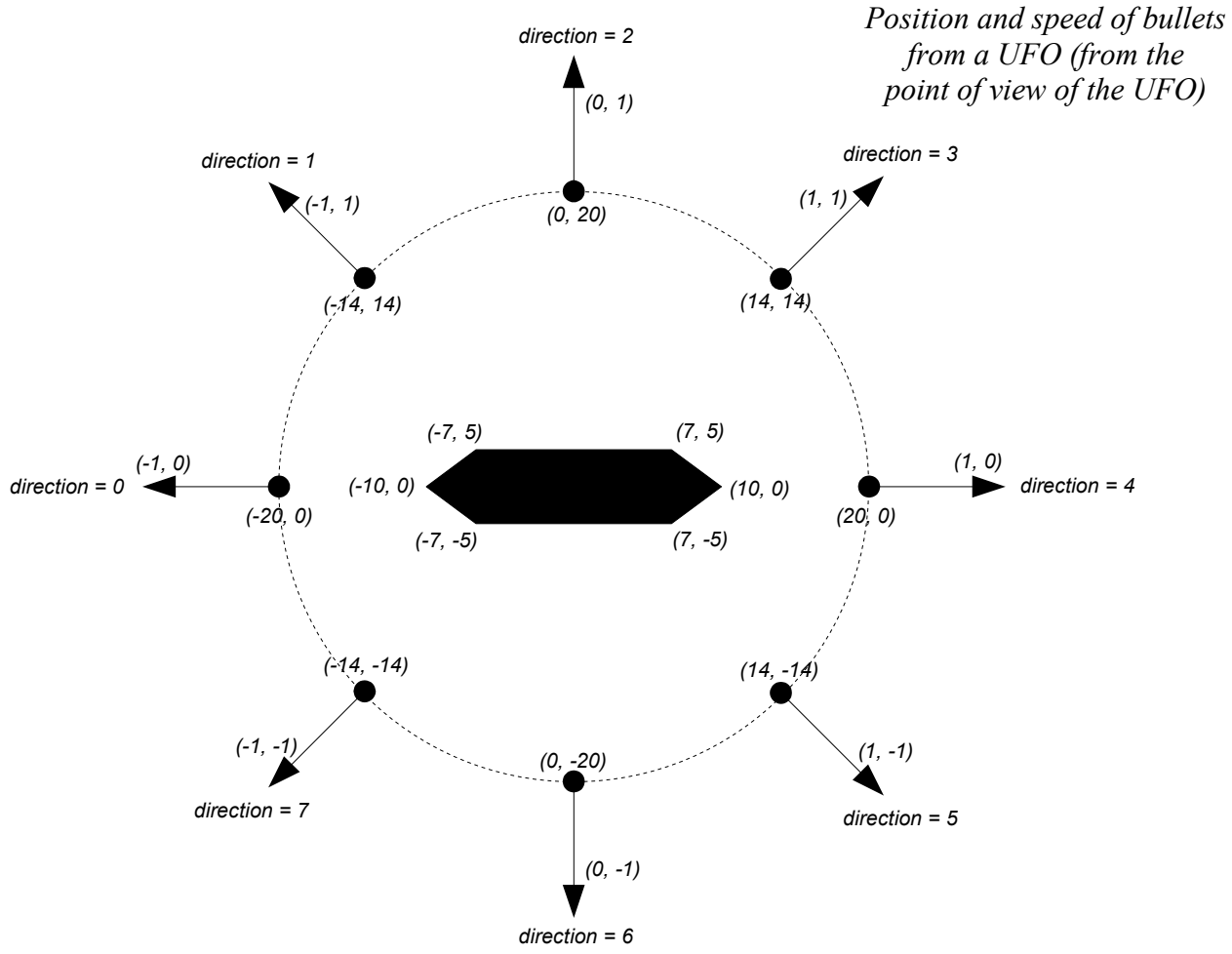
UFOs

Time for another baddy. We'll create ourselves a UFO, something narrow that drifts gently across the screen, spitting out slow bullets all around it. We'll start it off in the middle of the gap between the floor and the ceiling, just to give ourselves a little challenge.

It's not much of a challenge, really. All we have to do is find the y-coordinate of the floor and ceiling (which we've done before), and average them.

```
class UFO(games.Polygon, games.Mover, Hittable):
  SHAPE = ((-10, 0), (-7, 5), (7, 5),
           (10, 0), (7, -5), (-7,-5))
  def __init__(self, screen, floor, ceiling, start_x):
    floor_y = SCREENHEIGHT - floor.get_height(start_x)
    ceiling_y = ceiling.get_height(start_x)
    start_y = (floor_y + ceiling_y) / 2
    self.init_polygon(screen, start_x, start_y,
                      UFO.SHAPE, colour.green)
    self.init_mover(-(FLIGHT_SPEED+1), 0)
    self.direction = random.randint(0, 7)
    self.count = 10
```

Now about the bullets. What we want to do is to spit out one bullet whenever count hits zero in whatever direction `self.direction` is pointing, then reset the counter and move on to the next direction. We also have to create the bullet far enough away from our UFO that it doesn't accidentally hit itself. The diagram below shows you what we mean in more detail.



This isn't quite ideal — the bullets going diagonally should really be going a bit slower than shown (0.7 instead of 1), but it will do for now. Notice that what we've drawn out in the picture is how things look from the point of view of the UFO. From the point of view of the game, the UFO is moving, so we have to add its movement to the bullets' movement if we want them to move right.

We could write this function as a huge chain of `if` statements, and you may remember doing this for the Robots game in the beginners' course. However that's pretty boring to write. Instead, we'll make some (constant) lists and use `direction` to pick the right values out. That makes for a shorter program, and a bit less typing.

```
                                                    It's best to put these at the start of the class
BULLET_X = (-20, -14, 0, 14, 20, 14, 0, -14)
BULLET_Y = (0, 14, 20, 14, 0, -14, -20, -14)
BULLET_DX = (-1, -1, 0, 1, 1, 1, 0, -1)
BULLET_DY = (0, 1, 1, 1, 0, -1, -1, -1)

def moved(self):
  if self.xpos() < -10:                          Check for falling off the screen
    self.destroy()
  else:
    self.count = self.count - 1
    if self.count < 0:
      (x, y) = self.pos()
      (dx, dy) = self.get_velocity()
      Bullet(self.screen,
             x + UFO.BULLET_X[self.direction],
             y + UFO.BULLET_Y[self.direction],
             dx + UFO.BULLET_DX[self.direction],
             dy + UFO.BULLET_DY[self.direction],
             colour.green, UFO_BULLET_LIFETIME)
      self.count = 10
      self.direction = self.direction + 1
      if self.direction > 7:
        self.direction = 0
    self.check_for_hits()                        Line up with the if
```

You'll need to create the constant `UFO_BULLET_LIFETIME`, but that's easy enough. Set it to 100, which should be long enough to make the bullets a danger to the spaceship.

> You also need to write the `hit` function for our `UFO` class. That's dead easy; UFOs just die when they're hit. You don't need any help to write that yourself!

Then just like all our other enemy ships, we'll need to get the `ScramScreen` class to make them when we want some. Let's set our UFOs off in phase 3 (the not quite so narrow bit). That means that we need to add these lines to `__init__`:

```
                                                    Lined up with the other active and timer variables
    self.ufos_active = 0
    self.ufo_timer = 0
```

. . . then this line to `end_phase_2`:

```
self.ufos_active = 1
```

. . . and the corresponding line in `end_phase_3`:

```
self.ufos_active = 0
```

. . . and finally, the bit of program to count the counter and make new `UFO`s when it runs down, which goes in `tick`:

```
                                                    Line this up with the other if statements
    if self.ufos_active:
      self.ufo_timer = self.ufo_timer - 1
      if self.ufo_timer <= 0:
        UFO(self, self.floor, self.ceiling, SCREENWIDTH)
        self.ufo_timer = random.randint(80, 200)
```

One last enemy

There's one last sort of enemy ship we want to create. We'll call them Wasps, because they're a bit like Bees except for being bad tempered. Bees will only sting you if you provoke them (*i.e.* run into them); wasps will come looking for trouble.

Wasps will attempt to home in on our spaceship, a bit like the robots did in the game at the end of the beginners' course. The main difference is that we changed the robots' position to move them closer to the player; with our wasps, we're going to change their *velocity* instead. When the wasp is above the ship, we'll increase its downward speed (or equivalently reduce its upward speed). Similarly, if the wasp is to the left of the ship, we'll increase its rightwards speed to encourage it back to the ship. If the ship manages to dodge, it may take the Wasp quite a while to turn round!

We'll use the same trick that we did with the `UFO` class to start the Wasp off vertically in the middle of the cave:

```
class Wasp(games.Polygon, games.Mover, Hittable):
  SHAPE = ((-8, 0), (0, 5), (8, 0), (0,-5))
  def __init__(self, screen, ship, floor, ceiling, start_x):
    floor_y = SCREENHEIGHT - floor.get_height(start_x)
    ceiling_y = ceiling.get_height(start_x)
    start_y = (floor_y + ceiling_y) / 2
    self.init_polygon(screen, start_x, start_y,
                      Wasp.SHAPE, colour.red)
    self.init_mover(0, 0)
    self.target_ship = ship
```

When our `Wasp` moves, we want to compare its position with that of the ship. That will determine how we change its speed, as we described it above.

The one thing that we won't do here that we did for all our other types of enemy ship is to make it go away when it falls off the screen. This is because unlike all the other enemy ships we've seen, `Wasps` can end up travelling right in an effort to catch up with the player's spaceship. Indeed, if the player dodges the `Wasp` when it first comes at him instead of shooting it, it will shoot past, screech to a halt somewhere off the screen and then come screaming back in to try to clobber him from behind. This being a thoroughly nasty and evil thing to do, we're quite happy to let this happen.

We can do all the changes we want with a couple of quite straightforward `if` statements:

```
                                          # Lined up with the other def statements
  def moved(self):
    (x, y) = self.pos()                   # The wasp's position
    (tx, ty) = self.target_ship.pos()     # The ship's position

    if x < tx:                            # If the wasp is left of the player ...
      ddx = 1                             # ... accelerate right
    elif x > tx:                          # If the wasp is right of the player ...
      ddx = -1                            # ... accelerate left
    else:
      ddx = 0

    if y < ty:                            # Do the same with the y coordinate
      ddy = 1
    elif y > ty:
      ddy = -1
    else:
      ddy = 0

    (dx, dy) = self.get_velocity()
    self.set_velocity(dx + ddx, dy + ddy)
    self.check_for_hits()
```

The `hit` function is exactly like that for `UFO`s, so we'll leave you to write that yourself.

> If we decide to have our Wasps appear in phase 3 with a delay of 50 to 150 ticks between each Wasp, can you write the rest of the program? *Hint:* it's a good idea to start the timer off with an initial delay instead of zero, to make sure that the first Wasp doesn't immediately kill itself on the first UFO.

And that's it. You've written a complete game of Scramfender! We recommend that you spend some time trying out different combinations of enemy ships and different sets of limits for the floor and ceiling until you get results that you like.

Bells and whistles

There are lots of things that you could do to change around and improve on your game.

- The way that `Wasp`s accelerate towards the spaceship is a bit crude at the moment. If you look through the *Space War* worksheet, it explains how to do gravity (which is the same sort of thing really) more accurately. See if you can figure out how to use that make the `Wasp`s behave better. It won't make a lot of difference to your game, but it's an interesting challenge!
- You could keep a score, adding say 10 points every time an enemy ship is destroyed, and display it in the top right hand corner. You'll need to use a `games.Text` object just like you did for the end of game message, and the Python function `str` to convert a number into a string (something in quotes marks). You may also need to ask a leader about keeping the `Text` object "on top" of the cave pieces.
- If you're going to keep a score, you should really do some extra work to only count ships that the player kills, and ignore the ones that crash into each other or the cave walls. See if you can figure out how. Even harder, see if you can figure out a way of telling bullets fired by the `Ship` from bullets fired by a `UFO`.
- If you haven't done it already, you could add some extra phases with different combinations of ships and different styles of cave. Make some of them asymmetric (*i.e.* make the limits on the floor and ceiling different).
- Try your hand at inventing new kinds of alien ships. You can use bits of the ways that the existing ships work to help you out. With a bit of imagination, anything is possible!
- At the moment, the ship flies through the cavern at a steady pace. See if you can figure out how to make everything go faster when you get back to phase 1, and make the game more of a challenge. *Hint:* there's a quick and dirty way with a hidden "gotcha", or the proper way that's more work. You'll have to ask a leader to help you find our what's on the screen already, otherwise life (and the cave) will start looking very peculiar.
- You could give the player a "smart bomb" (just the one) that destroys everything on the screen. Apart from the player's ship and the cave, of course.
- You could make the game start again after the player loses. Ask a leader for help with that, it's a bit involved.