5: The Robots are Coming!

Gareth McCaughan Revision 1.8, May 14, 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 the last numbered sheet in our Python course. When you've finished working through it, you'll have written a program that plays a surprisingly addictive game: You're surrounded by robots who are trying to catch you, and you have to outwit them by fooling them into crashing into one another.

This is quite a difficult sheet, especially since I'm not going to give you quite as much help as you've had in previous sheets. You're supposed to be learning to do things on your own, after all! However, this sheet *does* introduce a lot of things you haven't seen before, so you should feel free to ask for help at any time.

What you need to know

- The basics of Python (from Sheets 1 and 2)
- Simple Python graphics (from Sheet 3)
- Functions (from Sheet 3; you might want to look at Sheet F too)
- Loops (see Sheet L)

The game

I'd better begin by explaining exactly how the game works.

You're being chased by robots. They are armed, and if a robot manages to catch you you're dead. You have no weapons. Fortunately, the robots are rather stupid. They always move towards you, even if there's something in the way. If two robots collide then they both die, leaving a pile of junk. And if a robot collides with a pile of junk, it dies. So, what you're trying to do is to position yourself so that when the robots try to chase you they run into each other!

It turns out that that's a little too hard; it's too easy to get into a position where all you can do is wait for the robots to catch you. So we'll give the player the ability to teleport to a random place on the screen.

It's simplest to make everything happen on a grid of squares. So the player can move in any of the 8 compass directions, and at each turn a robot will move one square in one of those 8 directions; for instance, if the robot is north and east of the player then it will move one square south and one square west.

Planning it out

Let's begin by thinking about what needs to happen when you play the game.

- Position the player and the robots on the screen.
- Repeatedly,
  - move all the robots closer to the player
  - check for collisions between robots, or between robots and piles of junk
  - check also whether the player has lost
    - (if so, the game is over)
  - and whether all the robots are dead
    - (if so, restart the game at a higher level (more robots!)
  - allow the player to move or teleport

There's a lot of stuff here. We'll start. as uaual, by writing some easy bits.

Moving the player around

All the action takes place on a grid. Our graphics window is 640 pixels by 480; let's make the grid 64 by 48 squares, each 10 pixels on a side. That's a pretty good size.

We need to represent the player by something we can draw using the graphics facilities described in Sheet G (*Graphics*). I suggest that a filled-in circle will do as well as anything else.

So, let's write a simple program that lets you move the player around the screen. This involves a bunch of stuff that's new, so I'll lead you through it carefully.

An outline of the program

To make this program easier to follow (remember that it will be getting bigger and bigger as we add bits to it, until it's a program to play the complete game of Robots), we'll divide it up using functions. (See Sheet 3, and Sheet F (*Functions*).) So, begin by typing in the following program.

```
from livewires import *                    As usual
begin_graphics()                           So that we can draw things
allow_moveables()                          This is explained later!
place_player()
finished = 0
while not finished:
  move_player()
end_graphics()                             We've finished
```

As the program develops, we'll add bits to this skeleton (for instance, there'll be a `place_robots()` function added quite soon). For the moment, our task is to define those functions so that the player can move around the screen.

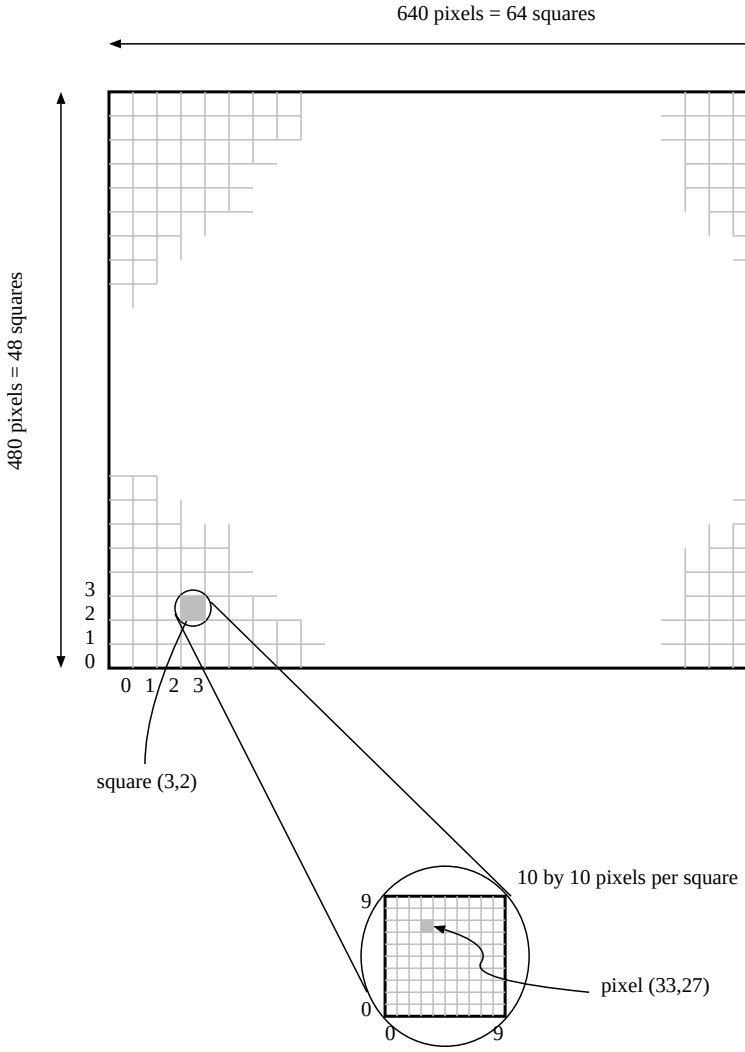
Where to start?

Let's look at `place_player()`, which decides where on the screen the player should begin. Remember to put the definition of `place_player` before you actually use it.

Well, this at least is easy. Let's have two variables called `player_x` and `player_y`, saying where the player is. You could either make them contain the player's coordinates in pixels, or the coordinates in grid squares (which will be 10 times

smaller, because each grid square is 10 pixels on a side). Either is perfectly workable. I prefer the latter, though it's not a big deal; you should probably go along with my preference, because for the rest of this worksheet I'm going to assume that you have done!

I'd better explain this business about "grid coordinates" a bit more carefully. The graphics window is made up of 640×480 pixels. We're chopping them up into 64×48 squares.



So, the bottom-left pixel of square (17,23) is pixel (170,230) and its top-right pixel is pixel (179,239).

Back to `place_player()`. It needs to set the variables `player_x` and `player_y` randomly. `player_x` can have any value from 0 to 63; `player_y` can have any value from 0 to 47. If you can't remember how to do that, look back at Sheet 2 where `random_between()` was introduced.

And then it needs to put the player on the screen by saying something like `circle(player_x, player_y, 5, filled=1)`, except that those obviously aren't the right coordinates for the centre of the circle (because `player_x` etc are measured in grid squares, and `circle()` wants coordinates in pixels). What we actually need is for the centre of the circle to be in the middle of the grid square. So `circle(10*player_x, 10*player_y, 5, filled=1)` should do the trick.

(If you're confused by the `filled=1` bit, you might like to take a look at Sheet F (*Functions*), which describes "keyword arguments".)

Moving the player

Now, we need to move that circle around in response to what the player does at the keyboard. This involves two new ideas – moving graphics, and keyboard handling. Let's deal with the first one first.

In the "Python Shell" window, type the following:

```
>>> from livewires import *
>>> begin_graphics()
>>> allow_moveables()
>>> c = circle(320,200,5)
```

What `allow_moveables()` does is to make a small change to the behaviour of functions like `circle`. That fourth line still does draw the circle, as you'll have seen, but it does something else too: it returns a value, and that value can be used later to move the circle around (or remove it from the window completely).

So, try doing this:

```
>>> move_to(c, 300,220)
```

The circle should move when you do that.

Challenge: Write a loop that makes the circle move smoothly from (0,0) to (640,480): in other words, from the bottom left to the top right of the screen.

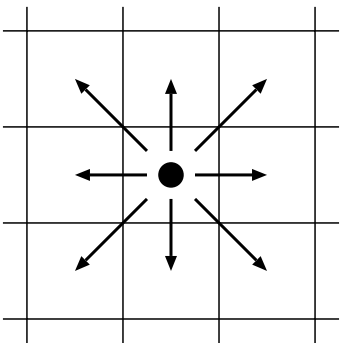
One other thing:

```
>>> remove_from_screen(c)
```

You can probably guess what that does, but you should try it anyway.

Keys

That's all very well, but of course no one will play this game if they don't get to *choose* where the player moves! The easiest way to do this is to let the player use the keyboard. We'll allow movement in 8 different directions ...



... and of course we should allow the player to stay still. So we need a 3×3 grid of keys. Your keyboard almost certainly has a "numeric keypad" on the right: the numbers 1–9 will do fine. So, for instance, pressing "7" should make the player move upwards and to the left.

Therefore, what we have to do inside `move_player()` is to test which of those keys (if any) is pressed.

The function `keys_pressed()` returns a list of the keys that are pressed. Usually this list will either be empty or have exactly one thing in it. (Keys are represented by the characters they produce. Letter keys are represented by *lowercase* letters.

If you're in any doubt about how this works, run the following program, go over to the "Graphics Window" and press some keys. You may find that it doesn't respond very well; that's because the `time.sleep()` (which tells the computer to do nothing for a while) interferes with the computer's ability to watch for new key-presses. (There are better ways of doing the same thing, but they're more complicated.) When you're done, hold down the `Q` key until the machine notices.

```
from livewires import *
import time                                  See Sheet T for more about this
begin_graphics()
while 1:
  keys = keys_pressed()
  print keys
  if 'q' in keys:                            See Sheet C if you don't understand this
    break                                    See Sheet L if you aren't sure what this means
  time.sleep(0.5)                            Wait half a second.
```

So, now you know how to tell what keys are pressed, and you know how to move an object around. So, put the two together:

Change `place_player` so that it puts the value returned from `circle()` in a variable (maybe call it `player_shape` or something); *and*

. . . write a `move_player()` function that uses `keys_pressed()` to see what keys are pressed, and moves the player if any of the keys 1–9 are pressed. Moving the player requires

- Updating `player_x` and `player_y`
- Calling `move_to` to move the player on the screen

Eeek! I bet you find it doesn't work. Specifically, the `move_player()` function will say it's never heard of the variables you set in `place_player()`. *What's going on?*

If you haven't already read Sheet F (*Functions*), now might be a good time to glance at it. The important point is that any variable you set in a function (e.g., `player_x` in `place_player()`) are "local" to that function: in other words, they only exist inside the function, and when the function returns they lose their values. Usually this is a Good Thing (for reasons discussed briefly in Sheet F), but here it's a nuisance. Fortunately, there's a cure.

Suppose you have a function definition that looks like this:

```
def f():
  blah blah blah
  x = 1
  blah blah blah
```

Then `x` is a local variable here, and calling `f` won't make a variable called `x` that exists outside `f`:

```
>>> f()
>>> print x
Blah blah blah ERROR ERROR blah blah
NameError: x
```

But if you add to the definition of `f` a line saying `global x` (just after the `def`, and indented by the same amount as the rest of the definition), then the variable `x` inside `f` will be "global": in other words, `x` will mean just the same inside `f` as it does outside. So the `print x` that gave an error with the other version of `f` will now happily print "`1`".

I hope it's clear that this bit of magic is what we need to fix the problem with `place_player()` and `move_player()`. Add `global` statments to both definitions.

At this point, you should have a little program that puts a circle on the screen and lets you move it around using the keyboard. Fair enough, but (1) there's not much challenge there, without any robots, and (2) you might notice that the player can move off the edge of the window!

Deal with the second of those problems. All you need to do is: After the player's move, see whether he's off the edge (either coordinate negative, or `x>63`, or `y>47`). If so, repair the offending coordinate in what I hope is the obvious way.

Adding a robot

Eventually, we'll have the player being pursued by a horde of robots. First, though, a simpler version of the program in which there's only one robot.

Before the line of your program that says `place_player()`, add another that says `place_robot()`. Write the `place_robot()` function: it should be very much like `place_player()`, except of course that we should (1) use different names for the variables and (2) draw a different symbol. I suggest a `box`, unfilled. You may need to think a bit about exactly where to put the corners of the box.

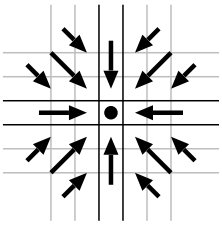
Remember to use the `global` statement as you did in `place_player()`.

Moving the robot

After the `move_player()` line, add another that says `move_robot()`. Now we need to work out how to move the robot. The robot should move according to the following rules:

- If a robot is to the left of the player, it moves right one square.
- If a robot is to the right of the player, it moves left one square.
- If a robot is above the player, it moves down one square.
- If a robot is below the player, it moves up one square.

So, if a robot is both left of the player and above the player, it will move down and to the right. This diagram may make it clearer how the robots move.



Write the `move_robot` function. It needs to look at the positions of player and robot, and decide where to put the robot according to the rules above; and then actually put it there.

This function *doesn't* need to check whether the robot is trying to walk off the edge of the screen. Can you see why?

Try your program. Even if you haven't made any mistakes, it still won't be much fun to play, for two reasons.

Two problems

- You'll probably find that as soon as the game starts, the robot runs towards the player at enormous speed, and then sits on top of him.
- Once that's happened, the game obviously ought to be over, but it isn't. The robot just sits on top of the player and moves wherever he does.

The second problem is easy to fix. After the call `move_robot()`, add another function call: `check_collisions()`. Then write a function `check_collisions()`, which tests whether the player has been caught by the robot. That happens

if, after the robot's move, the player and the robot are in the same place. (And *that* happens if `player_x` and `robot_x` are equal, and `player_y` and `robot_y` are equal. You probably need to look at Sheet C (*Conditions and Conditionals*) to find out how to say "if this is true *and* that is true", if you haven't already done that.)

If they *are* in the same place, the program should print a message saying something like "You've been caught", and set that `finished` variable that gets tested at the top of the main `while` loop to something other than 0. Then the program will finish when the player gets caught.

What about the first problem?

What's going on is just that the computer is much faster than you are. So in the time you're looking at the screen, working out where the player is, and deciding what key to press, the computer has moved the robot 100000000 times or so. (Maybe I'm exaggerating a *little*.)

The easiest way to even out this unfairness is to make the robot have only one move for each move of the player's. And the easiest way to do *that* is to make the `move_player()` function sit and wait until the player has pressed a key. You can do that with a `while` ; the simplest way is probably to make it begin `while 1:` and do a `break` whenever one of the keys 1–9 is pressed.

Two more problems

Once you've fixed those problems and tried your program, you'll probably notice one or two more.

- There's no escape: the robot will just chase the player to the edge of the screen, and then the player has nothing to do other than die.
- Very occasionally, the robot will actually start in the same place as the player, who will then be *instantly* doomed.

Again, the second problem is easier to fix than the first. The trick is to change `place_player` so that it never puts the player in the same place as the robot. How to do that? Just place the player at random; if hes in the same place as the robot, try again (still at random) and check again (and again, and again, if necessary, until the player and robot are in different places). This is just a `while` loop again. Because the test in a `while` loop always has to go at the start, it will have to look (in outline) something like this:

```
choose the player's position at random
while the player and robot haven't collided:
  choose the player's position at random
```

Notice that we have to "choose the player's position at random" twice here. This is a good indication that we should *put it in a function* – whenever you have something that gets done more than once in your program, you should think about making it into a function.

In fact, "choose the player's position at random" is *already* in a function. The function is called `place_player`, and that's exactly what it does. So we need a new function to do the more complicated thing above; call it `really_place_player` or something.

What about "`while` the player and robot haven't collided"? The thing we're testing here ("not collided") ought to go in a function, too: it happens here, and also in `check_collisions`. If you don't already know about "returning a value from a function", have a look at Sheet F now.

OK. So we need a function that checks whether the player and the robot are in the same place. Call it `collided()` or something. It should return 1 if they *have* collided, and 0 if they *haven't*. (If you still haven't read Sheet C (*Conditions and Conditionals*), now would be a very good time.)

Once we have this function, our loop can just say

```
place_player()
while collided():
  place_player()
```

Now that you have the `collided` function, you can also make `check_collisions` a little simpler: instead of looking at `player_x` and `robot_x` (etc) itself, it can just say

```
if collided():
  blah blah                                    do whatever is necessary if they have collided
```

Hmm. I said the second problem (robot being in the same place as player) was easier to fix than the first. That was true, but after fixing the second problem the first is actually really easy. We'll let the player "teleport": move instantaneously to a new place on the screen. We don't want them ever to land on top of a robot. So, in `move_player`, test for the `T` key being pressed; if it's pressed, move the player to a random empty space on the screen ... which we already know how to do: just call `place_player()`. Oh, and don't forget that you need to `break` out of the `while` loop in `move_player` if the player presses `T`.

So far, so good

Let's take a moment to review what we have so far. The player and one robot are placed on the screen, at random. They both move around, in the correct sort of way. If they collide, the game ends. The player can teleport.

This is pretty good. There are just two more really important things to add.

- There ought to be lots of robots, not just one.
- The robots will then be able to crash into each other, as well as into the player. When that happens, we need to remove both the robots from the game and replace them with a pile of junk that doesn't move. We also need to check for robots crashing into junk.

The first of these is a pretty major change. If you haven't already been saving lots of different versions of your program, now would be a very good time to make a copy of the program as it is now. You're about to perform major surgery on it, and it would be wise to keep a copy of how it was before you started operating.

A touch of class

Before we do that, though, some minor surgery that will make the major surgery easier. To explain the minor surgery, we need a major digression, so here is one.

Try doing this in your "Python Shell" window.

```
>>> class Robot:
...   pass
...
>>> fred = Robot()
>>> fred.x = 100
>>> fred.y = 200
>>> print fred.x, fred.y
```

(I'm sure you can guess what that last `print` statement will print.)

What we've done is to define a "class" called `Robot`. Roughly, "class" means "kind of object". In other words, we've told the computer "In the future, I might want to talk about a new kind of thing. Those things are called `Robot`s." You can do all kinds of clever things with classes, but we won't need anything fancy in this sheet; just the very simplest things.

All "pass" means is "I have nothing to say here". It's sometimes used in `if` and `while` statements; so, for instance, to sit and wait until a key is pressed in the graphics window you'd say

```
while not keys_pressed():
  pass
```

which translates as "repeatedly, as long as there are no keys pressed, *do nothing*". In our class definition, it means "There's nothing else to say about the `Robot` class.".

The line `fred = Robot()` means "Remember I told you about a new kind of thing called a `Robot`? Well, I want one of those. Call it `fred`.". The thing that gets named `fred` is what's called an "instance" of the `Robot` class.

Class instances (like `fred` in the little example above) can have things called "attributes". For instance, `fred.x` is an "attribute" of `fred`. Attributes are rather like variables; you can do all the same things with them that you can with variables. But an attribute is really more like an array element, or (if you've read Sheet D) like a "value" in a dictionary: it's part of an object.

For instance, after the example above, suppose we say

```
>>> bill = fred
>>> print bill.x
```

Then the machine will print `100`, just as it would have if we'd asked for `fred.x`, because `fred` and `bill` are just different names for the same object, whose `x` attribute is `100`.

Incidentally, it's usually considered a Good Thing if all the instances of a class have the same attributes. The idea is that all the instances of a class should be "the same kind of object". (If you carry on learning about programming, then one day you'll realise what a huge oversimplification what I've just said is. Never mind.)

What on earth does all this have to do with our game? Well, there are three separate pieces of information associated with the player and with the robot in the game: two coordinates (`player_x`, `player_y`) and one other thing (`player_shape`, used for moving the shape on the screen that represents the player. (Incidentally, the thing called `player_shape` is actually a class instance, though its class definition is slightly more complicated than that of `Robot` in the example above.)) We're about to have, not just one robot, but *lots* of them. Our program will be much neater if all the information about each robot is collected together into a single object.

In fact, this is an important idea to know whenever you're designing a program:

> *Whenever you have several different pieces of information that describe a single object, try to avoid using several different variables for them. Put them together in a class instance, or a list, or a tuple, or a dictionary, or something.*

So, let's improve our program by grouping sets of variables together into class instances.

- At the beginning of the program, add two class definitions:

```
class Robot:
        pass
      class Player:
        pass
```

- At the very beginning of `place_player`, say `player = Player()`.
- At the very beginning of `place_robot`, say `robot = Robot()`.
- Change the `global` statements so that they only "globalise" the variables `player` (instead of `player_x` etc) or `robot` (instead of `robot_x` etc).

- Change all references to `player_x`, `player_y` and `player_shape` to `player.x`, `player.y` and `player.shape`.
- Do the same for `robot`.
- Make sure your program works again.

A list of robots

(Now would be another good time to save a copy of your program!)

You've already met "lists", very briefly, in Sheet 1. It would be a good idea, at this point, to have a quick look at Sheet A (*Lists*); you don't need to absorb everything on it, but reminding yourself of some things lists can do would be a good move.

What we'll do is to have, instead of a single variable `robot`, a list `robots` containing all the robots on the screen. Each element of the list will be an instance of the `Robot` class.

So, what needs to change?

- `place_robot` should be renamed `place_robots` everywhere it occurs. It should place several robots, and instead of setting up the single variable `robot` it should make a new `Robot` instance for each robot, and put them together in a list.
- `check_collisions` and `collided` are going to have to become much more complicated, because we need to check for four different kinds of collision:
  - *Player and robot*: player dies
  - *Robot and robot*: both robots disappear, and they get replaced with a piece of junk
  - *Robot and junk*: robot disappears
  - *Player and junk*: player dies

  (We'll see shortly how we can simplify this a bit.)
- If this means messing with `collided`, then `really_place_player` will probably have to change too.

Before we start ripping the program apart, we need a clear plan of how the new version is going to work. So, here is one.

- *What are we going to do about the junk?* We could have a new class (called, say, `Junk`) and a list of junk objects. On the other hand, a piece of junk actually behaves exactly the same as a robot except that it doesn't move. So what we'll do is to have another attribute for each `Robot`, called `junk`, so that `r.junk` is 1 (i.e., "true") if the robot `r` is actually a piece of junk, and 0 if it's still a working robot.
- *What about collision checking?* We'll have two functions for checking collisions. The first one will check whether the player is in the same place as any robot (or pile of junk), rather like the `collided` function we already have.

  The other thing we need to be able to do is to check whether two robots have collided. It will turn out that we want to know more than just "have some robots collided?"; we need to know *which* robots. The best thing to do is to have a function that determines, for a particular robot, whether any robot *earlier* in the list has collided with it. Then, if we run through the whole list checking this, we'll pick up every collision exactly once. (Can you see why?) The function will return either 0 or the `Robot` instance that has crashed with the robot we're asking about.

  The `check_collisions` function will call the "player dead?" function once, and the "robots crashed?" function once for each robot. If a robot turns out to have crashed into another, it gets removed from the list and the other robot is made into a piece of junk.
- *What do we do to turn a robot into a piece of junk?* Three things.
  - Set its `junk` attribute.

  - Remove its shape object from the screen with `remove_from_screen`.
  - Make a new shape object for it. I suggest a filled box.
    `robot.shape = box(blah,blah,blah,blah,filled=1)`

- *What about moving the robots?* No problem. Just move the robots one by one. The only thing to be careful about is that if a robot is actually junk (i.e., its `junk` attribute is non-0), it shouldn't move.

Placing the robots

We'll place the robots one by one, at random. After placing each robot, we'll call the second collision-testing function I mentioned earlier to see whether it's in the same place as any already-placed robot; if so, we'll try again.

Here's roughly how the new `place_robots()` function should work:

- Make `robots` an empty list.
- Repeat once for each robot we want:
  - Add a randomly-placed robot to the list. (Remember to set its `junk` attribute to 0.)
  - Repeat `while` this robot hasn't collided with any other:
    * Try another random place for it instead.
  - Make a shape object for the robot, and store it in the robot's `shape` attribute.

About the only thing you might need to know is that the way to add a new item to a list is to say something like `my_list.append(new_item)`.

Checking for collisions

Obviously that's no use until we have functions for checking for collisions. Let's work those out next.

Collisions involving the player

Easy. This is just like the old `collided` function with a loop added.

```
def player_caught():
  for robot in robots:
    if player.x == robot.x and player.y == robot.y:
      return 1
  return 0
```

(Do you understand why that `for` loop works?)

Collisions between robots

This isn't much harder. Write a function called `robot_crashed`. It should take a single argument, which will be a `Robot` instance, and loop through the `robots` list (just as `player_caught` does). Each time through the list, we should do two things.

- Check whether the robot we're looking at in the list is the the same as the robot we were asked about. If so, `break`. (*Question*: Why do we do that?)
- Check whether the robot we're looking at in the list is in the same place as the robot we were asked about. If so, return it. ("it" = "the robot we're looking at in the list".)

Finally, if we get to the end of the list without returning from the function, we should `return 0`, because that means "no collision".

Testing for collisions after the robots move

So, now we have functions that check for collisions. How do we use them?

A couple of them, we already know how to use. Checking for the player's death is just the same as it always was (except that we changed the function's name to `player_caught` instead of `collided`). And a moment ago, we saw how to use `robot_crashed` when placing the robots.

The other time when we need to check for collisions is just after the robots have moved. Of course we need to test `player_caught` then, just as before, but we also need to see whether any robots have crashed.

What you need to do is:

- For each robot:
  - Call `robot_crashed` and put the result in a variable.
  - If the result was 0, this robot hasn't collided with any "earlier" robot and we needn't do anything.
  - Otherwise, the result was some earlier robot with which it *has* collided. So:
    - Make the "earlier" robot into junk (I explained how to do that a little while ago, remember?)
    - Remove this robot from the list. (I'll say a bit about how to do this in a moment.)

When a collision happens, we need to remove one robot from circulation completely. If you've read Sheet L (*Loops*), you'll know about the `del` statement, which can remove an item from a list. (It can also do all kinds of other things, some of which are likely to produce really weird errors, so be careful...)

Why 'del' is dangerous

Deleting things from a list while looping over the list is dangerous. Here are a couple of terrible examples of the sort of thing that can happen:

```
>>> my_list = [0,1,2,3,4,5,6,7,8,9]
>>> for i in range(0,len(my_list)):
...   if my_list[i]==3 or my_list[i]==7:
...     del my_list[i]
...
Traceback (innermost last):
  File "<stdin>", line 2, in ?
IndexError: list index out of range
>>> my_list
```

Challenge: Work out exactly what's gone wrong here.

OK, let's try another way.

```
>>> my_list = [0,1,2,3,4,5,6,7,8,9]
>>> for item in my_list:
...   if item==3 or item==7:
...     del my_list[my_list.index(item)]
...
>>> my_list
[0, 1, 2, 4, 5, 6, 8, 9]
```

Looks like it works. Let's try another example.

```
>>> my_list = [0,1,2,3,4,5,6,7,8,9]
>>> for item in my_list:
...     if item==3 or item==4:
...         del my_list[my_list.index(item)]
...
>>> my_list
[0, 1, 2, 4, 5, 6, 7, 8, 9]
```

Uh-oh. 4's still there.

Challenge: Work out what the trouble is this time.

Once you've done that, it might occur to you to try to repair the first example like this:

```
my_list = [0,1,2,3,4,5,6,7,8,9]
for i in range(0,len(my_list)):
  if my_list[i]==3 or my_list[i]==7:
    del my_list[i]
    i = i-1
```

Unfortunately, this behaves in exactly the same way as the other version did.

Challenge: Work out why.

If you've managed all those, you'll probably (1) understand lists and loops pretty well, and (2) be very frustrated. There are a couple of tricks that *will* work for us. For instance, if you repeat our second attempt, but loop *backwards*, that will work. An even simpler way is to build a completely new list to replace the old one; and that's what I suggest we do here. So ...

What we actually do

... our collision-handling code really ought to look like *this*:

- Make an empty list called `surviving_robots`.
- For each robot:
  - Call `robot_crashed` and put the result in a variable.
  - If the result was 0, this robot hasn't collided with any "earlier" robot and we needn't do anything, *except* that we should append the robot to the `surviving_robots` list.
  - Otherwise, the result was some earlier robot with which it *has* collided. So:
    - Make the "earlier" robot into junk (I explained how to do that a little while ago, remember?)
    - Remove this robot from the list. (I'll say a bit about how to do this in a moment.)
- Finally, say `robots = surviving_robots`.
- If there are now no robots left, the player has won! Display a congratulatory message and set `finished=1`.

At this point you should pretty much have a working game. (It'll probably take a little while to get the bugs out of it, though.) Congratulations! This has been a long sheet, with lots of new ideas in it; well done for surviving to the end.

What next?

The game works, but there are plenty of things it would be nice to add.

- The "you have lost" and "you have won" messages are printed in the Python Shell window. It would be nice to have them displayed in the graphics window. Find out how to do this.

- Instead of having the game end when the player eliminates all the robots, it would be good to have it start over with more robots ("a higher level"). You'll need to (1) stick the whole thing you've written so far inside a `while` loop, (2) distinguish between the two ways in which the game has `finished` (since one should make only the inner `while` end, and the other should make them both end), and (3) make the number of robots placed by `place_robots()` a variable and change it each time around the outer loop.
- Give the player a score that starts at 0 and increases every time they kill a robot, or something. First of all, just display the score in the "Python Shell" window using `print`. Then try to get it displayed in the Graphics Window somewhere. *Caution*: if you do this you may need to decrease the size of the playing area: it might be annoying to have a score displayed over the top of where the action is happening.
- Stick the whole thing inside yet another `while` loop, and put a question "Would you like another game?" at the bottom.
- Now add a *high-score* feature.
- Add (perhaps only on higher levels) some extra-tough robots, that don't die as soon as they crash into something. (For instance, the robot might have two "lives", so that when it crashes into something it loses a life.) If you do this, here are two recommendations:
  - Make these special robots a different colour. (You may need to look at Sheet G.)
  - Probably the easiest way to implement this thing is to give *every* robot an attribute called `lives` or something. Most will start with `lives` being 1, but the tough ones will have more lives. Then, when robots crash, you should decrease the number of lives each of them has, and take action accordingly. You can probably lose the `junk` attribute if you do this; a robot will be junk if its number of lives is 0.
  - If you take that advice, here are two things to be careful about: (1) If a robot has no lives left, you obviously don't need to decrease the number of lives. (2) The code that removes robots from the game may need a bit of care. A robot should only be removed when it becomes junk *and the robot it crashed with became junk too*.

You've now reached the end of our Python course. Congratulations! Some things you might not have looked at yet, and might be interesting:

- The more complicated bits of Sheet F (*Functions*).
- Sheet M (*Modules*) tells you a little bit about how the huge collection of (sometimes) useful functions that come with Python is organised.
- Sheet W (*The LiveWires module*) will tell you what things you've been using in these worksheets are part of Python itself, and what things were added by us to make your life easier.
- Sheet D (*Dictionaries and tuples*) will tell you about some things Python does that you haven't had to use yet.
- Sheet O (*Classes and objects*) will tell you much more about classes and their instances.

Warning: Writing computer programs is an addictive activity.