W: The LiveWires module

Gareth McCaughan, Paul Wright and Rhodri James Revision 1.16, October 27, 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

The version of Python you're using comes with a "package" called `livewires`. There are four modules inside the package: `beginners`, `games`, `boards` and `colour`. `beginners` was produced for the LiveWires holiday in 1999, though it has been modified since. `games` and `colour` were produced for the LiveWires holiday in 2000. `games` was modified for 2001 so that it uses the Pygame library, and has been tinkered with ever since. `boards` was written in 2001 while `games` was being modified.

Note: If you've got an old version of the LiveWires package, containing only the `livewires.py` file, you've got the 1999 version of the `beginners` module and no `games` or `colour` modules. You should get the new package if you want to use the games worksheets.

This sheet needs some Python experience, but we've tried to make it as easy to follow as possible.

> *When we want to say something to more experienced programmers who are reading this, we've enclosed it in a box like this.*

Beginners

When you say `from livewires import *` at the start of a program (which you usually should in the Beginners' worksheets), that makes all the things in the `beginners` module available for your use.

> *Saying* `from livewires import *` *causes all the things in* `beginners` *to be imported into the local namespace. This is usually considered to be a Bad Thing, but it is done in the Beginners' worksheets for the sake of simplicity. We may change this at some point in the future. We've chosen not to do it this way in the Games worksheets.*

Some day, you might find yourself having to use Python without our module. In that case, you'll need to know what's ours and what's in standard Python. That's what this sheet is all about.

Graphics

Most of the `livewires` module deals with graphics. *All* of the graphics things discussed in the Beginners' worksheets depend on our module. That doesn't mean you can't do graphics without our module; it's just more difficult.

Currently the graphical parts of the `beginners` *module are implemented as a wrapper round* `Tkinter`*. A canvas object holds all the other objects on the screen.*

Input and output

Just input, actually. The functions `read_number`, `read_string` and `read_yesorno` come from `livewires`. You can get roughly the same effect as `read_number` using the standard Python function `input`, and exactly the same effect as `read_string` using the `raw_input` function. If you need something like `read_yesorno` and can't use our module, you'll need to write your own.

Other things

Our `random_between` function is exactly the same as `random.random_int` in standard Python.

Games

The `games` module uses the ideas of classes and objects explained in Sheet O. There are various classes in the module, which are explained below.

In the Games worksheets, we've told you to use `from livewires import games` when you want to access things in the `games` module. To use the things in the module, you need to put `games` before the name. For example, to refer to the `Screen` class in the `games` module, you'd write `games.Screen`.

In the `games` module, all co-ordinates are in pixels. (0,0) is the top left corner. Co-ordinates increase downwards and to the right. Times are in thousands of a second, that is, milliseconds. Colours are specified in the way described in the `colour` module (see below).

Most of these classes are designed to be extended by subclassing. To avoid some of the syntactic pain that usually attends this kind of design in Python, every class defines an extra method, which takes exactly the same arguments as its `__init__` *method. In a class called* `Foobar`*, this method is called* `init_foobar`*.*

The `games` module uses the Pygame library, which you can find at `http://www.pygame.org/`. You need to install the Pygame library before you can use the `games` module.

The games module is used by setting up objects which do what you want, and then calling the `mainloop` method of the `Screen` class. The `mainloop` method handles moving your objects around and updating the screen. `mainloop` does not return until the player quits the game, so all the behaviour of the game must go inside the objects you create before calling `mainloop`.

Below, we list the classes in the `games` module, and their methods. For each method, we give its arguments. An argument written as `name=value` means the argument gets set to `value` if you don't specify it (so it's not compulsory to specify it).

Screen

The `Screen` class provides a region in which objects can exist and move.

The methods provided by the `Screen` class are listed below. We've listed them as you would call them. If you make a subclass of the screen class and want to redefine these methods, you'll need to add `self` as the first parameter of the method.

- `init_screen (width=640, height=480)`

  This method creates the window on the screen with the specified width, height. You can only do this once in your

program.

- `is_pressed (key)`

  This method returns 1 if the key is pressed, and 0 if it is not.

  The key names are in the games module. The letter keys are `games.K_a` to `games.K_z`. The space bar is `games.K_SPACE`. Return is `games.K_RETURN`. If you import the games library at the >>> prompt and say `dir (games)`, you can get a list of all the names in the games library, which will include the keys. Or you could look at the Pygame documentation at http://www.pygame.org/docs/ref/pygame_constants.html, as the key names are taken from the Pygame library.

  Note: most keyboards can only tell you about 4 keys being pressed a time, so once you're pressing more than 4 keys, `is_pressed` might return 0 even for keys which are pressed.

- `keypress (key)`

  This method is another way of getting key presses. It is called automatically whenever a key is pressed. In the standard `Screen` class, it does nothing. If you override it in a subclass of `Screen` you can get your subclass to handle key presses: the `key` parameter will be the key which has just been pressed, using the same key names as the `is_pressed` function above.

  Whether you want to handle keys by writing your own `keypress` method or using the `is_pressed` method depends on what you're doing: in all the example sheets, other objects call the `Screen`'s `is_pressed` method to find out whether a particular key is pressed, rather than the `Screen` handling the key press itself.

- `mouse_buttons ()`

  In exactly the same way as `is_pressed`, this method returns the pressed-ness of the mouse buttons. It returns a *tuple* of three numbers, being the state of the left, middle and right buttons respectively. The state is set to 1 if the button is pressed, and to 0 if it isn't.

  Note: most Windows computers don't have a middle mouse button, so you might not want to use it for anything important.

- `mouse_position ()`

  This method returns the current position of the pointer in the window as an (x, y) pair. You usually need this information if you're handling a mouse click!

- `mouse_down ((x, y), button)`

  As with key presses, you can wait to be told about mouse clicks rather than actively scanning for them. This method is called automatically whenever a mouse button is pressed. In the standard `Screen` class, it does nothing. If you override it in a subclass of `Screen` you can get your subclass to handle mouse clicks: the `x` and `y` parameters get the position of the pointer on the screen, while `button` is the number of the mouse button being pressed: 0 for left, 1 for middle and 2 for right.

  Whether you want to handle mouse clicks by writing your own `mouse_down` routine method or by using the `mouse_buttons` routine instead depends on what you're doing.

- `mouse_up ((x, y), button)`

  This method is exactly like `mouse_down` except that it's called when the mouse button is released. For a lot of purposes you will want to use this rather than `mouse_down`, unless speed is more important than accuracy!

- `set_background (background)`

  This method sets the background of the screen. You need to give it an image which you've loaded with `load_image`, not the filename of the image. Note that the background image should not have transparency set, or you'll get weird effects.

- `set_background_colour (colour)`

  Sets the background colour of the screen. `colour` is a tuple of three numbers between 0 and 255 indicating how much red, blue and green there is respectively in the colour. The `colours` module provides useful constants for common colours.

- `tick ()`

  This method is called every timer tick (usually 1/50th of a second). In the standard `Screen` class, it does nothing. If you make a subclass of the `Screen` class, you can override this method to do whatever you want to do every tick.

- `handle_events ()`

  This method enables you to write your own handler for the Pygame events, rather than using the standard one which comes with the `Screen` class. It is called every timer tick after all the objects have been updated. You probably don't want to mess with this function unless you understand Pygame's events module.

  The standard handler calls `keypress` to deal with key down events, `mouse_down` to deal with mouse down events, `mouse_up` to deal with mouse up events, and `quit` to deal with quit events.

  If you override this method, you *must* handle the quit event by calling `self.quit ()`.

- `quit ()`

  Calling this method will stop the main loop from running and make the graphics window disappear.

- `clear ()`

  Calling this method destroy all the `Objects` on the screen.

- `mainloop (fps = 50)`

  This method should be called once you've set up all the objects on the screen. This method won't return until the screen's `quit` method is called. It contains the loop which causes the objects on the screen to be drawn and redrawn, so nothing will move on the screen before you call it.

  The `fps` parameter is the number of times the screen should be updated every second. Whether or not your computer will actually achieve this depend on how fast a computer you have.

- `objects_overlapping (box)`

  Returns a list of all the `Objects` on the Screen whose bounding boxes overlap the box you give. A "bounding box" is a rectangle, with the edges either horizontal or vertical, which completely encloses shape you see on the screen.

  The box should be given as a sequence of four elements, which are the co-ordinates of the top left of the box, and its width and height, like this: `(x0, y0, width, height)`.

- `all_objects ()`

  Returns a list of all `Objects` on the screen.

Object

The `Object` class represents a graphical object on the screen. You shouldn't create `Objects` directly: the LiveWires Games module provides subclasses of the `Object` class for you to use. Here is a list of all the methods that the subclasses of `Object` have in common:

- `destroy ()`

  Removes the object from the `Screen`.

  > *This doesn't remove all references to the* `Object` *itself that you might be keeping, so it doesn't guarantee that the* `Object` *will be removed from memory. What it does do is remove any references the* `games` *module is keeping to the* `Object`*.*

- `pos ()`

  Returns position of the object's *reference point*. When we say reference point, we mean some special point of the object: for a circle, this is the centre. The description of each subclass of object will tell you what its reference point is.

  The point is returned as a tuple: so you can say `(x, y) = my_object.pos ()`.

- `xpos ()`

  Returns the x-coordinate of the object's reference point.

- `ypos ()`

  Returns the y-coordinate of the object's reference point.

- `bbox ()`

  Returns a bounding box for the object. A "bounding box" is a rectangle, with the edges either horizontal or vertical, which completely encloses shape you see on the screen.

  The box is returned as a tuple of 4 numbers, which are the co-ordinates of the top left of the box, and its width and height, like this: `(x0, y0, width, height)`.

- `move_to (x,y)` *or* `move_to ((x,y))`

  Moves the object's reference point to (x,y), moving the object with it.

- `move_by (dx,dy)` *or* `move_by ((dx,dy))`

  Moves the object's reference point by `dx` in the x direction and `dy` in the y direction, moving the object with it.

- `rotate_to (angle)`

  Sets the angle by which the object is rotated, in anticlockwise degrees. An angle of 0 means the angle at which the object was originally placed.

  For some objects (circles, for example), this may actually do nothing, except that the angle is remembered for returning from `angle ()`.

- `rotate_by (angle)`

  Adjusts the angle by which the object is rotated, increasing it by "angle" degrees (anticlockwise).

  For some objects (circles, for example), this may actually do nothing, except that the angle is remembered for returning from `angle ()`.

- `angle ()`

  Return the object's current angle, in anticlockwise degrees. The angle is always greater than or equal to zero, and less than 360.

- `overlaps (o)`

  Returns true or false depending on whether this object overlaps another object "o".

- `overlapping_objects ()`

  Returns a list of the objects which are overlapping this object.

- `filter_overlaps (object)`

  You will not need to use this method unless you create your own subclasses of `Object`.

  This is a utility method which allows you to have better accuracy when judging whether two objects have collided. Just checking whether the bounding boxes which enclose them are overlapping will sometimes give false collisions when the objects themselves do not overlap (assuming the objects aren't rectangles).

  This method is called after having established that the bounding boxes touch.

  Some subclasses of `Object` override it (e.g. the `Circle` class). You can also override it in your own subclasses of `Object` to get better collision detection.

  This function should return 1 if the your object really overlaps the other object and 0 otherwise. The standard `Object` class relies on the bounding box check alone, so its `filter_overlaps` method just returns 1.

- `raise_object (above=None)`

  One way of viewing objects on a screen is with the objects being in different layers, the front of the screen being the "topmost" layer. Objects in "higher" layers then appear in front of (or "above") other objects.

  This method brings the object towards the front of the screen, placing it "above" the other object `above`. If `above` is `None` or missing, then the object is pulled to the topmost layer.

- `lower_object (below=None)`

  The reverse of `raise_object`, this puts the object behind `below`. If `below` is missing or `None`, the object is put to the "bottom" of the screen, behind everything except the background.

Colours

`Polygon`, `Circle` and `Text` objects all have a defined colour, and all have the following methods provided by the `ColourMixin` class:

- `set_colour (colour)`

  This sets the main "fill" colour of the object. `colour` is a tuple of three numbers from 0 to 255, specifying how much red, green and blue there is respectively in the colour. The `colours` module provides constants for many colours to avoid you having to experiment.

- `get_colour ()`

  This method returns the "fill" colour of the object.

Outlines

`Polygon` and `Circle` objects all have a defined outline colour, the colour which will be used for the edge of the object. They have the following methods provided by the `OutlineMixin` class:

- `set_outline (colour)`

  This sets the colour of the outline of an object. `colour` is a tuple of three red, green and blue values, as explained above, or `None`: if `None` is specified, the "fill" colour of the object will be used instead.

- `get_outline ()`

  This method returns the current outline colour of the object, or `None` if no outline colour has been specified.

Sprite

This class represents an image you've loaded from a file, for example the image of the asteroid in the Asteroids worksheet. `Sprite` is a subclass of `Object`, so it has all the methods from the `Object` class as well as this:

- `init_sprite (screen, x, y, image, a=0, static=0)`

  `screen` is the `Screen` which the image will be on. The centre of the image will be at `(x,y)`, which is also the reference point.

  `image` should be an image returned from the `load_image` function. See below for more details on that function.

  `a` is the angle of rotation you want the image to start at.

  `static` is a flag that can be set to 1 if the sprite isn't intended to move ever. When there are a lot of static objects and not too much else this can improve redrawing speed, but if there are only one or two it can actually slow things down.

Polygon

This class represents a closed polygon on a `Screen`. When we say *closed*, we mean that the lines making up the polygon completely enclose a single area. A square is an example of a closed polygon. A square with one side removed is not closed.

`Polygon` is a subclass of `Object`, so it inherits all the methods from `Object`. It has these methods of its own:

- `init_polygon (screen, x, y, shape, colour, filled=1, outline=None, static=0)`

  `screen` is the `Screen` that the `Polygon` is on.

  `x` and `y` are the co-ordinates of the reference point of the `Polygon`.

  The `shape` is a list of pairs of co-ordinates `[(x0,y0), (x1,y1), ...]`, giving the x and y co-ordinates of the corners of the shape. The co-ordinates are written relative to the reference point, that is, when you're writing them, assume the reference point is at `(0,0)`. (See the section on the ship in the "Games: Space War" sheet for an example).

  `colour` is the colour of the polygon. This is a tuple of three numbers from 0 to 255, specifying how much red, green and blue respectively there is in the colour. The `colours` module provides a number of useful predefined colours.

  If `filled` is 1, the polygon is filled in with its `colour`. If `filled` is 0, only the outline of the polygon is drawn.

  `outline` is the colour of the outline of the polygon, if it needs to be different from the main `colour`. If `outline` is `None`, then `colour` is used instead.

  `static` is a flag that can be set to 1 if the polygon isn't intended to move ever. It is mostly used for the `GridBox` class in the `boards` module, which form the squares of a board. When there are a lot of static objects and not too much else this can improve redrawing speed, but if there are only one or two it can actually slow things down.

- `set_shape (( (x0,y0), (x1,y1), ... ))`

  Change the shape of the object. The new shape is specified in exactly the same way as the `shape` argument to `init_polygon`. The position of the reference point on the screen stays the same.

- `get_shape ()`

  Return the current shape as a list of pairs of co-ordinates.

Circle

This class represents a circle on a `Screen`. It is a subclass of `Object`, so it inherits all the methods from `Object`. It has these methods of its own:

- `init_circle (screen, x,y, radius, colour, filled=1, outline=None, static=0)`

  `screen` is the `Screen` that the `Circle` is on.

  The centre of the circle is at `(x,y)`. This is also the reference point.

  `radius` is the radius of the circle.

  `colour` is the colour of the circle.

  If `filled` is 1, the polygon is filled. If `filled` is 0, only the outline of the circle is drawn.

  `outline` is the colour of the edge of the circle, if it needs to be different from the main `colour`. If `outline` is `None`, then `colour` is used instead.

  `static` is a flag that can be set to 1 if the circle isn't intended to move ever. When there are a lot of static objects and not too much else this can improve redrawing speed, but if there are only one or two it can actually slow things down.

- `set_radius (r)`

  Sets the circle's radius to r.

- `get_radius (r)`

  Returns the circle's radius.

Text

This class represents some text on a `Screen`. It is a subclass of `Object`, so it inherits all the methods from `Object`. Notice that the reference point is the centre of the text, which is not usually what you expect. `Text` has these methods of its own:

- `init_text (screen, x,y, text, size, colour, static=0)`

  `screen` is the `Screen` that the `Text` is on.

  `x` and `y` are the co-ordinates of the centre of the text. The reference point is `(x,y)`.

  `text` is a string containing the text to be placed on the screen.

  `size` is the height of the text.

  `colour` is the colour of the text.

  `static` is a flag that can be set to 1 if the text isn't intended to move ever. When there are a lot of static objects and not too much else this can improve redrawing speed, but if there are only one or two it can actually slow things down.

- `set_text (text)`

  Sets the current text.

- `get_text ()`

  Returns the current text as a string.

Timer

The `Timer` class is a class you can add to something which is also a subclass of `Object`, to make an object that performs actions at regular intervals. A class which is intended to be used with another class is called a "mix-in". For instance, if you wanted to make a new class of your own which was a `Circle` and also a `Timer`, you would define the class by saying `class MyClass (games.Circle, games.Timer):`

- `init_timer (interval, running=1)`

  `interval` is how often the `tick` method is called, measured in timer ticks. How long a tick is depends on the `fps` argument you give to the `Screen`'s `mainloop` method. Setting `fps` to 50 means a tick is 1/50 of a second.

  `running` controls whether the timer starts immediately on initialisation, or whether you need to explicitly `start` it later.

  You must call this method *after* you have called the `init_` method for the `Object` subclass you are using.

- `stop ()`

  Stop the timer running. It continues to exist, but doesn't count any more.

- `start ()`

  Starts the timer again. A full interval will elapse before it ticks.

- `get_interval ()`

  Gets the current interval.

- `set_interval (interval)`

  Sets the current interval.

- `tick ()`

  This method must be *supplied* by you, by subclassing the `Timer` class. If it is not present, the module will generate an error and quit.

Mover

The `Mover` class is a subclass of the `Timer` class. A `Mover` is something which moves itself around the screen at regular intervals.

Like the `Timer` class itself, the `Mover` class is intended to be used as a mix-in class with a subclass of `Object`. For example, to create your own class representing a moving polygon, you would say `class Ship (games.Polygon, games.Mover)`

- `init_mover (dx, dy, da=0)`

  You must call this method *after* you have called the `init_` method for the `Object` subclass you are using.

  Every tick (see `Timer`, above), your object will `move_by dx` pixels in the x direction and `dy` pixels in the y direction. If `da` is given and is non-zero the object will also be rotated by `da` degrees. The object's `moved ()` method will then be called.

  Note that your object must *have* a `moved` method, so you must provide one in your subclass.

- `set_velocity ([dx,dy])` *or* `velocity ((dx,dy))`

  This method sets `dx,dy`.

- `get_velocity ()`

  This method returns `(dx,dy)`

- `set_angular_speed (da)`

  This method sets `da`.

- `get_angular_speed ()`

  This method returns `da`.

- `moved ()`

  You must supply the `moved` method by subclassing. The `moved` method is called on each tick after the object has been moved by `(dx,dy)` and rotated by `da`.

  Note: if all of `dx`, `dy`, and `da` are zero, this method is still called each tick.

Message

The `Message` class is a subclass of the `Text` class and the `Timer` class. A `Message` is a `Text` object that will go away after a specified period.

- `init_message (screen, x, y, text, size, colour, lifetime, after_death=None)`

  Except for the last two arguments, this method works identically to the `init_text` method (see the `Text` class we mentioned previously).

  `lifetime` is the lifetime of the message in ticks, that is, how long it will last before it goes away.

  When the message goes away, the `after_death` argument will be called if it was supplied. If it is supplied, the `after_death` argument must be something which is callable (a function, for example).

Animation

The `Animation` class is a subclass of the `Sprite` and `Timer` classes. It allows you to produce animations, that is, pictures on the screen which change every tick.

You construct an `Animation` by saying `Animation (screen, x, y, nonrepeating_images, repeating_images, n_repeats, repeat_interval)`, where

`screen` is the screen the animation is on.

`(x,y)` are the co-ordinates of the centre of the animation.

`nonrepeating_images` is a list of images from `load_image` or a list of filenames.

`repeating_images` is a list of images from `load_image` or a list of filenames.

`n_repeats` is the number of times the images in the `repeating_images` list will be shown. Set it to -1 to keep repeating forever.

`repeat_interval` is the number of ticks (see the `Timer` class) between one image and the next.

If the `nonrepeating_images` list is $[a, b, c, d]$ and the `repeating_images` list is $[x, y, z]$ then the sequence of images shown will be $a, b, c, d, x, y, z, x, y, z, x, y, z, ...$

The `Animation` class isn't currently intended to be subclassed: this may change in future.

Useful functions in the games module

- `load_image (file, transparent=1)`

  Loads an image from a file and returns an object which can be passed to the `games` modules' classes as the image for a `Sprite` or `Background`.

  `file` is a string containing the name of the file to load.

  `transparent` indicates whether the image should be transparent. If it is not zero, the image is transparent. The transparent colour is taken from the top left corner of the image. You should not make transparent images which you intend to use as the background for the screen.

The object returned by `load_image` *is a* `pygame.Surface` *object.*

- `load_sound (file)`

  Loads a sound object from a WAV file and returns an object which you can use to play the sound.

  You can use it in the following sort of way:

  ```
  mysound = games.load_sound ('explosion.wav')
  mysound.play ()
  ```

  The object returned by `load_sound` is a `pygame.mixer.Sound` object.

  For more information on the methods of sound objects see the documentation for Pygame at http://www.pygame.org/docs/ref/Sound.html.

- `load_animation (nonrepeating_files, repeating_files=[], transparent=1)`

  Loads a sequence of images from files like `load_image`, returning a pair of lists: first all the `nonrepeating` images, then all the `repeating` images (if any). These lists of images are suitable for making an `Animation`.

- `scale_image (image, x_scale)` *or* `scale_image (image, x_scale, y_scale)`

  This function scales an image object from `load_image` by the amount you specify.

  If you give it one scale factor, the image is scaled by the same amount in the x and y directions (giving a factor of 2.0 would double the size of the image, for example).

  If you give it two scale factors, you can scale the image by different amounts in the x and y directions.

  This function returns the scaled image object. It does not modify the original image you gave it.

Boards

The `boards` module is an extension of the `games` module, providing subclasses and useful functions particularly aimed at writing board games. The definition of "board games" is quite wide – the `boards` module can equally well be used for Snake as for Draughts!

In the Games worksheets, we've told you to use `from livewires import boards` when you want to access things in the `boards` module. You will usually want to access things in the `games` module also. To use the things in the module, you need to put `boards` before the name. For example, to refer to the `SingleBoard` class in the `boards` module, you'd write `boards.SingleBoard`.

> *These classes follow the same conventions as the* `games` *module. Coordinates are given relative to the top left hand corner of the window, for example, and any class called* `FooBar` *has an extra method called* `init_foobar`*.*

Most of the comments about the games module apply to the boards module too. In particular, the `SingleBoard` class is a subclass of `Screen`, so has a `mainloop` method that handles moving everything around the screen.

Below, we list the classes in the `boards` module, and their methods. For each method, we give its arguments. An argument written as `name=value` means the argument gets set to `value` if you don't specify it (so it's not compulsory to specify it).

GameCell

`GameCell` is a subclass of `Polygon` from the `games` module. It is intended to be subclassed itself to fit the exact needs of a particular game, since it provides only fairly general methods and properties. Note, however, that a `GameCell` is a *static* Polygon; you will often need to mix the `Container` class below into your subclass to get it to redraw properly. `GameCell`'s shape is always a square.

Primarily, the `GameCell` class provides a number of useful properties, variables that belong to each `GameCell` separately. These properties are:

- `grid_x`

  The column number of the cell in the grid, counting from zero as the leftmost column.

- `grid_y`

  The row number of the cell in the grid, counting from zero as the top row.

- `neighbours`

  A list of all the `GameCell`s which are next to this `GameCell`. These lists are created by a method in the `GameBoard` class, which allows some control over exactly what "neighbour" should be taken to mean.

- `direction`

  An array containing the `GameCell` which can be found in each direction from this `GameCell`. This array is created by a method in the `GameBoard` class, which allows some control over exactly what "direction" should mean.

The library supplies a set of constants for using the `direction` array in a meaningful way. These are:

- `boards.LEFT`
- `boards.UP_LEFT`
- `boards.UP`
- `boards.UP_RIGHT`
- `boards.RIGHT`
- `boards.DOWN_RIGHT`
- `boards.DOWN`
- `boards.DOWN_LEFT`

You should not rely on these constants being particular values or in a particular order. Instead, the module provides a number of useful functions to turn one direction into another, described later on.

In addition to the methods inherited from `Polygon`, the following methods can be used on a `GameCell`:

- `init_gamecell (board, i, j, line_colour=colour.black, fill_colour=colour.light_grey)`

  `board` is the `GameBoard` (see below) that the `GameCell` is on.

  `(i, j)` is the grid coordinates of the cell. In other words, `i` is the column number of the cell within the grid, while `j` is the row number. Cell `(0, 0)` is the top left-hand corner of the grid.

  `line_colour` is the colour of the line drawn between squares in the grid (i.e. the outline colour of the `Polygon`).

  `fill_colour` is the colour of the square itself.

- `add_neighbour(neighbour)`

  You probably won't need to call this method yourself, but if you do this is how. This method is called by the `GameBoard` class to create the `neighbours` list for a `GameCell`. `neighbour` is a `GameCell` which is next to (in some sense) this cell.

  Note: this only sets up a one way relationship. In other words, the instruction `fred.add_neighbour(bill)` tells `fred` that `bill` is his neighbour, but doesn't tell `bill` anything at all about `fred`.

- `add_direction(dir, cell)`

  Again, you probably won't need to call this method yourself since it is called by the `GameBoard` class. It builds up the `direction` array for this `GameCell`.

  `dir` is the direction under consideration.

  `cell` is the `GameCell` to be found in direction `dir`.

  Note: again, this method only establishes a connection one way. The instruction `lobby.add_direction(LEFT, main_hall)` tells the `lobby` that the `main_hall` is to its left, but doesn't let on to the `main_hall` that the `lobby` is near it at all.

GameBoard

The `GameBoard` class is where you keep the `GameCells` on your board. You will almost always want to subclass `GameBoard` (or more likely the `SingleBoard` class described below) to get exactly what you want for your game.

Each `GameBoard` contains a rectangular grid of `GameCells`. Not all cells in the grid have to actually contain `GameCells`, but all the cells are assumed to be square whether or not they are present.

The `GameBoard` class provides two useful properties (variables):

- `grid` is a list of lists (see Sheet A if you're not sure what that means) which contains every `GameCell` in the table. The cell at `grid[i][j]` is the cell which will appear in column `i`, row `j` on the screen.
- `cursor` is an optional feature of the library. If it is switched on using `enable_cursor` (see below), `cursor` contains the `GameCell` which is currently highlighted. All the boardgame worksheets use `cursor` to control the game from the keyboard, moving the "cursor" around the board to select a particular cell that some other action should happen in.

`GameBoard` provides the following methods:

- `init_gameboard (screen, (x, y), n_cols, n_rows, box_size, line_colour=colour.black, fill_colour=colour.light_grey, cursor_colour=colour.red)`

  `screen` is the `Screen` on which this board lives.

(x,y) are the coordinates of the top left-hand corner of the board; under most circumstances these will be the coordinates of the top left-hand corner of the GameCell in the top left-hand corner of the grid.

n_cols and n_rows are the number of columns and rows of GameCells in the grid respectively.

box_size is the height and width of each square in the grid.

line_colour is the colour of the lines between or around each GameCell (in other words, the outline colour of the Polygon).

fill_colour is the colour of each GameCell.

cursor_colour is the colour of the "cursor", the colour which is used instead of the line colour to highlight a particular GameCell if this feature of the class is used.

- new_gamecell (i, j)

  You will probably want to override this method in any subclass that you create. This method is called automatically to create a new cell to fit into column i, row j of the grid. The version supplied with GameBoard itself creates a GameCell, which is almost certainly not what you want. Instead you should write a version which returns one of the subclasses of GameCell that you have made.

- create_neighbours(orthogonal_only=0)

  This method must not be called before init_gameboard has been called. It scans through the grid, informing each GameCell of which other GameCells are its neighbours.

  orthogonal_only is a flag controlling whether or not to include cells that are only diagonally adjacent to one another. If it is set to 0, cells on the diagonal are included; if it is 1, then only the cells directly up, right, down or left (*orthogonally adjacent*) of each cell are considered to be neighbours.

- create_direction(orthogonal_only=0, wrap=0)

  This method must not be called before init_gameboard has been called. It scans through the grid, informing each GameCell of which other GameCells lie in which directions from it.

  orthogonal_only is a flag controlling whether or not the diagonal directions are filled in. If it is 0 then the diagonals are considered, otherwise they are not. This is not usually a major concern, and may be removed in future versions of the library.

  wrap is a flag which controls whether or not to "wrap" the board's sense of direction. If wrap is 0, then there is considered to be nothing to the left of the leftmost column of cells, nothing above the top row of cells, and so on. If wrap is 1, then the rightmost column of cells are considered to be to the left of the leftmost column of cells, and the bottom row of cells are considered to be above the top row of cells, and vice versa. This allows a piece to move off the top of the board onto the bottom, for example.

- keypress (key)

  This method should be called whenever a key is pressed. This will happen automatically in any class that subclasses both GameBoard and Screen, such as SingleBoard.

  The method checks to see if it understands the key code in key as an instruction to move the cursor. If it doesn't, it calls handle_keypress, allowing the game to do any special processing that it might need to do.

  Key codes that keypress understands are held in the property key_movements. This is a dictionary containing key codes matched to (di, dj) pairs, where di is the required change in the column number and dj is the change in row number for the cursor. The keypress method does not allow these changes to take the cursor off the board, nor does it attempt to "wrap" the cursor by reintroducing it on the right if it tries to fall off the left of the grid. The key codes and movements set up by default are:

  - K_UP: (0, -1)
  - K_DOWN: (0, 1)
  - K_LEFT: (-1, 0)
  - K_RIGHT: (1, 0)

- `handle_keypress(key)`

  This method is called automatically whenever the `boards` library receives a key code that it does not know how to handle. Except as noted above, it is used in exactly the same way as `Screen.handle_keypress` in the `games` module. You should override this method in your subclass if you want to handle keyboard input yourself.

- `enable_cursor(i, j)`

  This method switches on the cursor, placing it on the cell in column `i`, row `j`. If the cursor already exists, it is simply moved to `(i,j)`. If `(i, j)` is not a valid cell on the grid, an error will occur.

- `disable_cursor()`

  This method switches off the cursor, setting `cursor` to `None` and removing the red outline from the screen. It is safe to call this method if the cursor is already disabled — in that case, nothing will happen.

- `move_cursor(i, j)`

  This method moves the cursor to the cell at column `i`, row `j` on the grid. The cursor must already be enabled, and `(i, j)` must be a valid cell on the grid, otherwise an error will occur.

- `cursor_moved()`

  This method is called automatically whenever the cursor is enabled, disabled (provided it was enabled at the time) or moved either explicitly in the program or by pressing keys. By default, it does nothing — you should override this method in your subclass if you want to be notified when the state of the cursor has changed.

  This method may change in the future, since it's not enormously useful at present.

- `on_board(i, j)`

  This method returns `1` if grid position `(i, j)` is really on the board, and `0` otherwise. By default, `(i, j)` is considered to be on the board if both `i` and `j` are zero or more, `i` is less than the maximum number of columns passed to `init_gameboard` and `j` is less than the maximum number of rows.

  `on_board` is used when creating the grid, and when setting up the `direction` and `neighbours` properties. You can override it in your subclass to produce a board that has holes in it (a Ludo board, for instance) automatically.

- `map_grid(fn)`

  `fn` is a callable object (a function, usually) which is expected to take an entry from `grid` as a parameter. The `map_grid` method then calls `fn` on every cell on the board. There is an example of how to use it in the "Games: Minesweeper" sheet: it is most often handy for tidying up at the end of a game.

- `cell_to_coords(i, j)`

  This method returns the pixel coordinates (the `(x,y)` used by the `games` module) of the top left-hand corner of the cell in column `i`, row `j`.

- `coords_to_cell(x, y)`

  This method returns the grid coordinates (the `(i,j)` used by the rest of the `boards` module) of the cell inside which the point `(x,y)` in pixel coordinates lies. It is most often used for working out which `GameCell` the pointer was over when a mouse button was pressed or released. *Note:* the result is not necessarily a legal position for a cell, so check it with `on_board` before using it!

SingleBoard

The `SingleBoard` class is a straightforward subclass of `GameBoard` and `Screen`. It provides a single grid of `GameCells` in a window, which is the basis of most games. This is the class that you will probably want to subclass for your game, unless for some reason you want to have two or more boards in your window.

`SingleBoard` provides one method in addition to those it inherits from `Screen` and `GameBoard`:

- `init_singleboard(margins, n_cols, n_rows, box_size, line_colour=colour.black, fill_colour=colour.light_grey, cursor_colour=colour.red)`

margins is a tuple giving the amount of space to leave on each side between the edge of the window and the edge or the grid. It comes in two alternatives: either (x,y), where x is the amount of space at the left and right and y the space above and below, or (l,t,r,b), where l is the left margin, r the right margin, t the space at the top and b the space at the bottom.

n_cols and n_rows are the number of columns and rows respectively in the grid.

box_size is the height and width of the grid squares.

line_colour is the colour of the lines between or around each GameCell (in other words, the outline colour of the Polygon).

fill_colour is the colour of each GameCell.

cursor_colour is the colour of the "cursor", the colour which is used instead of the line colour to highlight a particular GameCell if this feature of the class is used.

Container

The Container class is a utility class that can be mixed in when you subclass any Object subclass. Its purpose is to keep together, in the right order, any Objects that are supposed to stay together. For example, if a counter is represented on the screen by a white Circle with a small red Circle in the middle of it, the Container class allows us to move the white Circle and have the red Circle tag along automatically.

Container overrides a number of the normal methods of an Object. As such, it must be the first class that you list in the brackets when making a subclass. For example:

```
class MyCounter(boards.Container, games.Circle):
```

will work while

```
class MyCounter(games.Circle, boards.Container):
```

will quietly and frustratingly fail to do what you want.

As a side-effect of GameCells being declared to be static (q.v.), you will usually want to make your game's subclass of GameCell a subclass of Container as well. In fact the only board game worksheet for this isn't true at present is "Snake", which isn't strictly speaking a board game at all!

Container provides one method for general use:

- init_container(contents)

  contents is slightly unusual: it is a list of the names of the objects which are to be contained. For example, suppose we have the white Circle containing a red Circle that we mentioned above, and we set it up like this:

  ```
  self.red_blob = games.Circle(blah, blah, colour.red)
  ```

  Then we would have to call init_container(['red_blob']) to initialise it. Suppose our counter had to have a number on it as well:

  ```
  self.digit = games.Text(blah, blah, "1", blah)
  ```

  Then we would need to call init_container(['red_blob', 'digit']). This would make sure that both the red_blob and the digit stayed on top of our counter. Furthermore, digit would be on top of red_blob!

  Note that the "contents" don't have to overlap the Container or each other, but they usually do. There is also nothing to stop the contents being a Container themselves; for an example of that, see the worksheet "Games: Chain Reaction".

> *Note that the* `init_container` *method must be called before the init routine for any other subclass (*`init_circle` *in the example above), otherwise the library will produce a cryptic* `KeyError` *and quit.*

As a side-effect, `init_container` will create `self.red_blob` and `self.digit`, setting them to `None`. You should therefore not create your objects until after you have initialised your container!

Useful functions in the boards module

The module supplies a number of useful functions specifically for playing with directions. You should always use these functions rather than making any assumptions about what numbers mean what directions. That way, if the library ever changes you won't have to retype everything!

- `turn_45_clockwise(direction)`

  This function returns the direction 45 degrees clockwise of `direction`. In other words, if you call it with `UP`, it will reply with `UP_RIGHT`.

  The remaining functions all follow the same naming scheme, apart from the last one.

- `turn_45_anticlockwise(direction)`
- `turn_90_clockwise(direction)`
- `turn_90_anticlockwise(direction)`
- `turn_180(direction)`
- `random_direction(orthogonal_only=0)`

  To avoid the need for you to write lots of tedious `if`s or cunning look-up code, this function returns a random direction. If `orthogonal_only` is 0, all 8 directions are equally likely to be selected; otherwise only the *orthogonal* directions (`UP`, `DOWN`, `LEFT` and `RIGHT`) will be chosen from.

Colour

The `colour` module provides some pre-defined colours for use with the `games` and `boards` modules: whenever the modules wants a colour, you can give them an object from this module.

The colours are:

- `red`
- `green`
- `blue`
- `black`
- `white`
- `dark_red`
- `dark_green`
- `dark_blue`
- `dark_grey`
- `grey`
- `light_grey`

- `yellow`
- `brown`
- `pink`
- `purple`

To use the module, say `from livewires import colour`. You can now refer to the colours as `colour.red`, `colour.green` and so on.

If you want to make your own colours, you'll need to know that the colours are a tuples of 3 numbers giving the amount of red, green and blue in the colour. The numbers are in the range 0 to 255. For example the `colour.red` is `(255,0,0)`, giving all red, no blue and no green.

The colours from the `colour` module are not intended for use with the `beginners` module.

Who we are

This section contains some background about us and LiveWires.

Contributors

Gareth McCaughan and Richard Crook both specified and wrote the LiveWires package, with the assistance of Neil Turton and Matthew Newton. Paul Wright ported the games library to use Pygame. Rhodri James added the boards library. Gareth wrote most of the Beginners' worksheets. Rhodri, Neil and Paul wrote various combinations of Beginners' and Games sheets. Mark White wrote the `wsheet` LaTeX class which enabled us to produce the Postscript and PDF versions of the sheets.

The rest of the team kept us sane on the LiveWires holiday itself. On the LiveWires 2001 holiday, the rest of the computing team was Rob Pearce and Colin Bell.

The maintainers of the course can be reached at `python@livewires.org.uk`.

LiveWires

LiveWires is a Scripture Union holiday for 12 to 15 year olds, which takes place in the UK every summer. The young people on the holiday have the chance to take part in a variety of computing, electronics and multimedia activities. The LiveWires Python Course was written by to help us to teach Python to the young people. We're making it available to everyone else as a way of giving something back to the Python community.

The LiveWires web site is at `http://www.livewires.org.uk/`

Scripture Union

Scripture Union is an organisation whose aim is to make Jesus known to children, young people and families. SU staff and volunteers work in more than 130 countries; in the UK its work includes schools work, missions, family ministry, helping Christians to read the Bible and supporting the church through training and resources. Scripture Union holidays have been happening for more than 100 years.

For more information on SU, see `http://www.scriptureunion.org.uk/`