W: The LiveWires module

Gareth McCaughan and Paul Wright Revision 1.16, October 27, 2001

Credits

Introduction

The version of Python you're using comes with a "package" called `livewires`. There are three modules inside the package: `beginners`, `games`, 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` has been modified for 2001 so that it uses the Pygame library.

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 `input` function, and exactly the same effect as `read_string` using the `raw_input` function, in standard Python. 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.

- `set_background (background)`

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

- `tick ()`

  This method is called every timer tick. 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, and calls `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 `Object`s 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 one 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 `Object`s on the screen.

Object

The `Object` class represents a graphical object on the screen. You shouldn't create `Object`s 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 commmon:

- `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 (eg 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.

Sprite

This class represents an image you've loaded from a file, for example, the image of the asteroid in the Asteroids worksheet.

- `init_sprite (screen, x, y, image, a=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.

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)`

  `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.

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

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

  Change the shape of the object. The new shape is specified in 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)`

  `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 polygon is drawn.

- `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`. It has these methods of its own:

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

  `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.

- `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)`

  `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.

  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.

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 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 $a, b, c, d, x, y, z, x, y, z, x, y, z, \ldots$.

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`.

- `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.

Colour

The `colour` module provides some pre-defined colours for use with the `games` module: whenever the `games` module wants a colour, you can give it 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. Gareth wrote most of the Beginners' worksheets. Rhodri James, 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/`