Pygame Zero Documentation

Release 1.1

Daniel Pope

August 03, 2015

Pygame Zero is for creating games without boilerplate.

It is intended for use in education, so that teachers can teach basic programming without needing to explain the Pygame API or write an event loop.

CHAPTER 1

Courses

1.1 Introduction to Pygame Zero

1.1.1 Creating a window

First, create an empty file called `intro.py`.

Verify that this runs and creates a blank window by running

```
pgzrun intro.py
```

Everything in Pygame Zero is optional; a blank file is a valid Pygame Zero script!

You can quit the game by clicking on the window's close button or by pressing `Ctrl-Q` (`-Q` on Mac). If the game stops responding for any reason, you may need to terminate it by pressing `Ctrl-C` in your Terminal window.

1.1.2 Drawing a background

Next, let's add a *`draw()`* function. Pygame Zero will call this function whenever it needs to paint the screen.

In `intro.py`, add the following:

```
WIDTH = 300
HEIGHT = 300

def draw():
    screen.fill((128, 0, 0))
```

Re-run `pgzrun intro.py` and the screen should now be a reddish square!

What is this code doing?

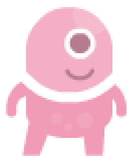
`WIDTH` and `HEIGHT` control the width and height of your window. The code sets the window size to be 300 pixels in each dimension.

`screen` is a built-in that represents the window display. It has a *range of methods for drawing sprites and shapes*. The `screen.fill()` method call is filling the screen with a solid colour, specified as a `(red, green, blue)` colour tuple. `(128, 0, 0)` will be a medium-dark red.

Pygame Zero is actually calling your draw function many times a second. If your `draw()` function draws slightly different things every frame, it will appear as an animation. We'll explore this shortly. For now, let's set up a sprite that we can animate.

1.1.3 Draw a sprite

Before we can draw anything, we'll need to save an alien sprite to use. You can right click on this one and save it ("Save Image As..." or similar).



(This sprite has a transparency (or "alpha") channel, which is great for games! But it's designed for a dark background, so you may not be able to see the alien's space helmet until it is shown in the game).

You need to save the file in the right place so that Pygame Zero can find it. Create a directory called `images` and save the image into it as `alien.png`. Both of those must be lower case. Pygame Zero will complain otherwise, to alert you to a potential cross-platform compatibility pitfall.

If you've done that, your project should look like this:

```
.
-- images/
|   -- alien.png
-- intro.py
```

`images/` is the standard directory that Pygame Zero will look in to find your images.

There's a built-in class called `Actor` that you can use to represent a graphic to be drawn to the screen.

Let's define one now. Change the `intro.py` file to read:

```python
alien = Actor('alien')
alien.pos = 100, 56

WIDTH = 500
HEIGHT = alien.height + 20

def draw():
    screen.clear()
    alien.draw()
```

Your alien should now be appearing on screen! By passing the string `'alien'` to the `Actor` class, it automatically loads the sprite, and has attributes like positioning and dimensions. This allows us to set the `HEIGHT` of the window based on the height of the alien.

The `alien.draw()` method draws the sprite to the screen at its current position.

1.1.4 Moving the alien

Let's set the alien off-screen; change the `alien.pos` line to read:

```python
alien.topright = 0, 10
```

Note how you can assign to `topright` to move the alien actor by its top-right corner. If the right-hand edge of the alien is at `0`, the the alien is just offscreen to the left. Now let's make it move. Add the following code to the bottom of the file:

```
def update():
    alien.left += 2
    if alien.left > WIDTH:
        alien.right = 0
```

Pygame Zero will call your *update()* function once every frame. Moving the alien a small number of pixels every frame will cause it to slide across the screen. Once it slides off the right-hand side of the screen, we reset it back to the left.

1.1.5 Handling clicks

Let's make the game do something when you click on the alien. To do this we need to define a function called *on_mouse_down()*. Add this to the source code:

```
def on_mouse_down(pos):
    if alien.collidepoint(pos):
        print("Eek!")
    else:
        print("You missed me!")
```

You should run the game and try clicking on and off the alien.

Pygame Zero is smart about how it calls your functions. If you don't define your function to take a `pos` parameter, Pygame Zero will call it without a position. There's also a `button` parameter for `on_mouse_down`. So we could have written:

```
def on_mouse_down():
    print("You clicked!")
```

or:

```
def on_mouse_down(pos, button):
    if button == mouse.LEFT and alien.collidepoint(pos):
        print("Eek!")
```

1.1.6 Sounds and images

Now let's make the alien appear hurt. Save these files:

- alien_hurt.png - save this as `alien_hurt.png` in the `images` directory.
- eep.wav - create a directory called `sounds` and save this as `eep.wav` in that directory.

Your project should now look like this:

```
.
-- images/
|   -- alien.png
-- sounds/
|   -- eep.wav
-- intro.py
```

`sounds/` is the standard directory that Pygame Zero will look in to find your sound files.

Now let's change the `on_mouse_down` function to use these new resources:

```python
def on_mouse_down(pos):
    if alien.collidepoint(pos):
        sounds.eep.play()
        alien.image = 'alien_hurt'
```

Now when you click on the alien, you should hear a sound, and the sprite will change to an unhappy alien.

There's a bug in this game though; the alien doesn't ever change back to a happy alien (but the sound will play on each click). Let's fix this next.

1.1.7 Clock

If you're familiar with Python outside of games programming, you might know the `time.sleep()` method that inserts a delay. You might be tempted to write code like this:

```python
def on_mouse_down(pos):
    if alien.collidepoint(pos):
        sounds.eep.play()
        alien.image = 'alien_hurt'
        time.sleep(1)
        alien.image = 'alien'
```

Unfortunately, this is not at all suitable for use in a game. `time.sleep()` blocks all activity; we want the game to go on running and animating. In fact we need to return from `on_mouse_down`, and let the game work out when to reset the alien as part of its normal processing, all the while running your `draw()` and `update()` methods.

This is not difficult with Pygame Zero, because it has a built-in *`Clock`* that can schedule functions to be called later.

First, let's "refactor" (ie. re-organise the code). We can create functions to set the alien as hurt and also to change it back to normal:

```python
def on_mouse_down(pos):
    if alien.collidepoint(pos):
        set_alien_hurt()


def set_alien_hurt():
    alien.image = 'alien_hurt'
    sounds.eep.play()


def set_alien_normal():
    alien.image = 'alien'
```

This is not going to do anything different yet. `set_alien_normal()` won't be called. But let's change `set_alien_hurt()` to use the clock, so that the `set_alien_normal()` will be called a little while after.

```python
def set_alien_hurt():
    alien.image = 'alien_hurt'
    sounds.eep.play()
    clock.schedule_unique(set_alien_normal, 1.0)
```

`clock.schedule_unique()` will cause `set_alien_normal()` to be called after `1.0` second. `schedule_unique()` also prevents the same function being scheduled more than once, such as if you click very rapidly.

Try it, and you'll see the alien revert to normal after 1 second. Try clicking rapidly and verify that the alien doesn't revert until 1 second after the last click.

1.1.8 Summary

We've seen how to load and draw sprites, play sounds, handle input events, and use the built-in clock.

You might like to expand the game to keep score, or make the alien move more erratically.

There are lots more features built in to make Pygame Zero easy to use. Check out the built in objects to learn how to use the rest of the API.

CHAPTER 2

Reference

2.1 Installing Pygame Zero

2.1.1 On Windows

1. Install Pygame for Python 3. This is available as a .msi installer from the Pygame Bitbucket.
2. Install Pygame Zero with pip:

```
pip install pgzero
```

2.1.2 On OSX

homebrew is a package manager for OSX. It will allow you to install nearly everything you need to get Pygame Zero up and running.

All commands will be entered in a Terminal window.

1. Install homebrew:

```
ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

2. Install Python 3:

```
brew install python3
```

3. Install the following dependencies, needed for compiling Pygame:

```
brew install sdl sdl_image sdl_mixer sdl_sound sdl_ttf
```

4. Now pygame can be installed easily using Python's own package manager, pip3:

```
pip3 install hg+http://bitbucket.org/pygame/pygame
```

5. Finally, install Pygame Zero!

```
pip3 install pgzero
```

2.1.3 On Ubuntu Linux

There is a .deb package of Pygame for Python 3 available in this PPA.

1. Add the PPA to your system sources:

```
sudo add-apt-repository ppa:thopiekar/pygame
```

2. Update the package list:

```
sudo apt-get update
```

3. Install the package:

```
sudo apt-get install python3-pygame
```

2. Install Pygame Zero with pip:

```
pip3 install pgzero
```

2.1.4 On Debian 8 (Jessie)

(There is a .deb package of Pygame for Python 3 in Debian unstable "Sid". On Jessie it's relatively simply to compile Pygame yourself.)

1. Install the dependencies:

```
sudo apt-get install mercurial python3-dev python3-numpy libav-tools \
    libsdl-image1.2-dev libsdl-mixer1.2-dev libsdl-ttf2.0-dev libsmpeg-dev \
    libsdl1.2-dev  libportmidi-dev libswscale-dev libavformat-dev \
    libavcodec-dev build-essential
```

2. Grab Pygame source:

```
hg clone https://bitbucket.org/pygame/pygame
```

3. Build Pygame:

```
cd pygame
python3 setup.py build
```

4. Install Pygame:

```
sudo python3 setup.py install
```

5. Install Pygame Zero with pip:

```
pip3 install pgzero
```

2.1.5 On Raspberry Pi

pgzero is likely to make an appearance in the Raspbian repo before long; until then...

(Starting from a vanilla noobs-install Raspbian)

1. sudo apt-get update
2. sudo apt-get install python3-setuptools python3-pip
3. sudo pip-3.2 install pgzero

2.2 Event Hooks

Pygame Zero will automatically pick up and call event hooks that you define. This approach saves you from having to implement the event loop machinery yourself.

2.2.1 Game Loop Hooks

A typical game loop looks a bit like this:

```
while game_has_not_ended():
    process_input()
    update()
    draw()
```

Input processing is a bit more complicated, but Pygame Zero allows you to easily define the `update()` and `draw()` functions within your game module.

draw()

Called by Pygame Zero when it needs to redraw your game window.

`draw()` must take no arguments.

Pygame Zero attempts to work out when the game screen needs to be redrawn to avoid redrawing if nothing has changed. On each step of the game loop it will draw the screen in the following situations:

- If you have defined an `update()` function (see below).
- If a clock event fires.
- If an input event has been triggered.

One way this can catch you out is if you attempt to modify or animate something within the draw function. For example, this code is wrong: the alien is not guaranteed to continue moving across the screen:

```
def draw():
    alien.left += 1
    alien.draw()
```

The correct code uses `update()` to modify or animate things and draw simply to paint the screen:

```
def draw():
    alien.draw()

def update():
    alien.left += 1
```

update() *or update(dt)*

Called by Pygame Zero to step your game logic. This will be called repeatedly, 60 times a second.

There are two different approaches to writing an update function.

In simple games you can assume a small time step (a fraction of a second) has elapsed between each call to `update()`. Perhaps you don't even care how big that time step is: you can just move objects by a fixed number of pixels per frame (or accelerate them by a fixed constant, etc.)

A more advanced approach is to base your movement and physics calculations on the actual amount of time that has elapsed between calls. This can give smoother animation, but the calculations involved can be harder and you must take more care to avoid unpredictable behaviour when the time steps grow larger.

To use a time-based approach, you can change the update function to take a single parameter. If your update function takes an argument, Pygame Zero will pass it the elapsed time in seconds. You can use this to scale your movement calculations.

2.2.2 Event Handling Hooks

Similar to the game loop hooks, your Pygame Zero program can respond to input events by defining functions with specific names.

Somewhat like in the case of `update()`, Pygame Zero will inspect your event handler functions to determine how to call them. So you don't need to make your handler functions take arguments. For example, Pygame Zero will be happy to call any of these variations of an `on_mouse_down` function:

```python
def on_mouse_down():
    print("Mouse button clicked")

def on_mouse_down(pos):
    print("Mouse button clicked at", pos)

def on_mouse_down(button):
    print("Mouse button", button, "clicked")

def on_mouse_down(pos, button):
    print("Mouse button", button, "clicked at", pos)
```

It does this by looking at the names of the parameters, so they must be spelled exactly as above. Each event hook has a different set of parameters that you can use, as described below.

`on_mouse_down`(*[pos]*[, *button*])
Called when a mouse button is depressed.

Parameters

- **`pos`** – A tuple (x, y) that gives the location of the mouse pointer when the button was pressed.
- **`button`** – An integer indicating the button that was pressed (see *below*).

`on_mouse_up`(*[pos]*[, *button*])
Called when a mouse button is released.

Parameters

- **`pos`** – A tuple (x, y) that gives the location of the mouse pointer when the button was released.
- **`button`** – An integer indicating the button that was released (see *below*).

`on_mouse_move`(*[pos]*[, *rel*][, *buttons*])
Called when the mouse is moved.

Parameters

- **`pos`** – A tuple (x, y) that gives the location that the mouse pointer moved to.
- **`rel`** – A tuple (delta_x, delta_y) that represent the change in the mouse pointer's position.
- **`buttons`** – The buttons that were depressed, if any.

`on_key_down`(*[key]*[, *mod*][, *unicode*])
Called when a key is depressed.

Parameters

- **key** – An integer indicating the key that was pressed (see *below*).
- **unicode** – Where relevant, the character that was typed. Not all keys will result in printable characters - many may be control characters. In the event that a key doesn't correspond to a Unicode character, this will be the empty string.
- **mod** – A bitmask of modifier keys that were depressed.

on_key_up([*key*][, *mod*])

Called when a key is released.

Parameters

- **key** – An integer indicating the key that was released (see *below*).
- **mod** – A bitmask of modifier keys that were depressed.

on_music_end()

Called when a *music track* finishes.

Note that this will not be called if the track is configured to loop.

Buttons and Keys

Built-in objects `mouse` and `keys` can be used to determine which buttons or keys were pressed in the above events.

Note that mouse scrollwheel events appear as button presses with the below `WHEEL_UP`/`WHEEL_DOWN` button constants.

class **mouse**

A built-in enumeration of buttons that can be received by the `on_mouse_*` handlers.

LEFT

MIDDLE

RIGHT

WHEEL_UP

WHEEL_DOWN

class **keys**

A built-in enumeration of keys that can be received by the `on_key_*` handlers.

BACKSPACE

TAB

CLEAR

RETURN

PAUSE

ESCAPE

SPACE

EXCLAIM

QUOTEDBL

HASH

DOLLAR

AMPERSAND

QUOTE

LEFTPAREN

RIGHTPAREN

ASTERISK

PLUS

COMMA

MINUS

PERIOD

SLASH

K_0

K_1

K_2

K_3

K_4

K_5

K_6

K_7

K_8

K_9

COLON

SEMICOLON

LESS

EQUALS

GREATER

QUESTION

AT

LEFTBRACKET

BACKSLASH

RIGHTBRACKET

CARET

UNDERSCORE

BACKQUOTE

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z

DELETE

KP0

KP1

KP2

KP3

KP4

KP5

KP6

KP7

KP8

KP9

KP_PERIOD

`KP_DIVIDE`

`KP_MULTIPLY`

`KP_MINUS`

`KP_PLUS`

`KP_ENTER`

`KP_EQUALS`

`UP`

`DOWN`

`RIGHT`

`LEFT`

`INSERT`

`HOME`

`END`

`PAGEUP`

`PAGEDOWN`

`F1`

`F2`

`F3`

`F4`

`F5`

`F6`

`F7`

`F8`

`F9`

`F10`

`F11`

`F12`

`F13`

`F14`

`F15`

`NUMLOCK`

`CAPSLOCK`

`SCROLLOCK`

`RSHIFT`

`LSHIFT`

`RCTRL`

LCTRL

RALT

LALT

RMETA

LMETA

LSUPER

RSUPER

MODE

HELP

PRINT

SYSREQ

BREAK

MENU

POWER

EURO

LAST

Additionally you can access a set of constants that represent modifier keys:

class keymods

Constants representing modifier keys that may have been depressed during an `on_key_up`/`on_key_down` event.

LSHIFT

RSHIFT

SHIFT

LCTRL

RCTRL

CTRL

LALT

RALT

ALT

LMETA

RMETA

META

NUM

CAPS

MODE

2.3 Built-in Objects

Pygame Zero provides useful built-in objects to help you make games easily.

2.3.1 Screen

The `screen` object represents your game screen.

It is a thin wrapper around a Pygame surface that allows you to easily draw images to the screen ("blit" them).

class `Screen`

`surface`
The raw Pygame surface that represents the screen buffer. You can use this for advanced graphics operations.

`clear`()
Reset the screen to black.

`fill`(*(red, green, blue)*)
Fill the screen with a solid color.

`blit`(*image, (left, top)*)
Draw the image to the screen at the given position.

`blit()` accepts either a Surface or a string as its `image` parameter. If `image` is a `str` then the named image will be loaded from the `images/` directory.

`draw.`**`line`**(*start, end, (r, g, b)*)
Draw a line from start to end.

`draw.`**`circle`**(*pos, radius, (r, g, b)*)
Draw the outline of a circle.

`draw.`**`filled_circle`**(*pos, radius, (r, g, b)*)
Draw a filled circle.

`draw.`**`rect`**(*rect, (r, g, b)*)
Draw the outline of a rectangle.

Takes a *Rect*.

`draw.`**`filled_rect`**(*rect, (r, g, b)*)
Draw a filled rectangle.

`draw.`**`text`**(*text*[, *pos*], ***kwargs*)
Draw text.

There's an extremely rich API for positioning and formatting text; see `ptext` for full details.

`draw.`**`textbox`**(*text, rect, **kwargs*)
Draw text, sized to fill the given *Rect*.

There's an extremely rich API for formatting text; see `ptext` for full details.

2.3.2 Rect

The Pygame Rect class is available as a built in. This can be used in a variety of ways, from detecting clicks within a region to drawing a box onto the screen:

For example, you can draw a box with:

```
RED = 200, 0, 0
BOX = Rect((20, 20), (100, 100))

def draw():
    screen.draw.rect(BOX, RED)
```

2.3.3 Resource Loading

The `images` and `sounds` objects can be used to load images and sounds from files stored in the `images` and `sounds` subdirectories respectively. Pygame Zero will handle loading of these resources on demand and will cache them to avoid reloading them.

You generally need to ensure that your images are named with lowercase letters, numbers and underscores only. They also have to start with a letter.

File names like these will work well with the resource loader:

```
alien.png
alien_hurt.png
alien_run_7.png
```

These will not work:

```
3.png
3degrees.png
my-cat.png
sam's dog.png
```

Images

Pygame Zero can load images in `.png`, `.gif`, and `.jpg` formats. PNG is recommended: it will allow high quality images with transparency.

We need to ensure an images directory is set up. If your project contains the following files:

```
space_game.py
images/alien.png
```

Then `space_game.py` could draw the 'alien' sprite to the screen with this code:

```
def draw():
    screen.clear()
    screen.blit('alien', (10, 10))
```

The name passed to `blit()` is the name of the image file within the images directory, without the file extension.

Or using the *Actors* API,

```
alien = Actor('alien')

def draw():
    alien.draw()
```

There are some restrictions on the file names in both cases: they may only contain lowercase latin letters, numbers and underscores. This is to prevent compatibility problems when your game is played on a different operating system that has different case sensitivity.

Image Surfaces

You can also load images from the `images` directory using the `images` object. This allows you to work with the image data itself, query its dimensions and so on:

```
forest = []
for i in range(5):
    forest.append(
        Actor('tree', topleft=(images.tree.width * i, 0))
    )
```

Each loaded image is a Pygame `Surface`. You will typically use `screen.blit(...)` to draw this to the screen. It also provides handy methods to query the size of the image in pixels:

class `Surface`

`get_width``()`
Returns the width of the image in pixels.

`get_height``()`
Returns the height of the image in pixels.

`get_size``()`
Returns a tuple (width, height) indicating the size in pixels of the surface.

`get_rect``()`
Get a `Rect` that is pre-populated with the bounds of the image if the image was located at the origin.

Effectively this is equivalent to:

```
        Rect((0, 0), image.get_size())
```

Sounds

Pygame Zero can load sounds in `.wav` and `.ogg` formats. WAV is great for small sound effects, while OGG is a compressed format that is more suited to music. You can find free .ogg and .wav files online that can be used in your game.

We need to ensure a sounds directory is set up. If your project contains the following files:

```
drum_kit.py
sounds/drum.wav
```

Then `drum_kit.py` could play the drum sound whenever the mouse is clicked with this code:

```
def on_mouse_down():
    sounds.drum_kit.play()
```

Each loaded sound is a Pygame `Sound`, and has various methods to play and stop the sound as well as query its length in seconds:

class `Sound`

`play``()`
Play the sound.

`play``(`*loops*`)`
Play the sound, but loop it a number of times.

Parameters **loops** – The number of times to loop. If you pass `-1` as the number of times to loop, the sound will loop forever (or until you call *Sound.stop()*)

stop()
Stop playing the sound.

get_length()
Get the duration of the sound in seconds.

You should avoid using the `sounds` object to play longer pieces of music. Because the sounds sytem will fully load the music into memory before playing it, this can use a lot of memory, as well as introducing a delay while the music is loaded.

2.3.4 Music

New in version 1.1.

Warning: The music API is experimental and may be subject to cross-platform portability issues.
In particular:

- MP3 may not be available on some Linux distributions.
- Some OGG Vorbis files seem to hang Pygame with 100% CPU.

In the case of the latter issue, the problem may be fixed by re-encoding (possibly with a different encoder).

A built-in object called `music` provides access to play music from within a `music/` directory (alongside your `images/` and `sounds/` directories, if you have them). The music system will load the track a little bit at a time while the music plays, avoiding the problems with using `sounds` to play longer tracks.

Another difference to the sounds system is that only one music track can be playing at a time. If you play a different track, the previously playing track will be stopped.

music.**play**(*name*)
Play a music track from the given file. The track will loop indefinitely.

This replaces the currently playing track and cancels any tracks previously queued with `queue()`.

You do not need to include the extension in the track name; for example, to play the file `handel.mp3` on a loop:

```
music.play('handel')
```

music.**play_once**(*name*)
Similar to `play()`, but the music will stop after playing through once.

music.**queue**(*name*)
Similar to `play_once()`, but instead of stopping the current music, the track will be queued to play after the current track finishes (or after any other previously queued tracks).

music.**stop**()
Stop the music.

music.**pause**()
Pause the music temporarily. It can be resumed by calling `unpause()`.

music.**unpause**()
Unpause the music.

music.**is_playing**()
Returns True if the music is playing (and is not paused), False otherwise.

`music.`**`fadeout`**`(`*duration*`)`
Fade out and eventually stop the current music playback.

Parameters **`duration`** – The duration in seconds over which the sound will be faded out. For example, to fade out over half a second, call `music.fadeout(0.5)`.

`music.`**`set_volume`**`(`*volume*`)`
Set the volume of the music system.

This takes a number between 0 (meaning silent) and 1 (meaning full volume).

`music.`**`get_volume`**`()`
Get the current volume of the music system.

If you have started a music track playing using *`music.play_once()`*, you can use the *`on_music_end() hook`* to do something when the music ends - for example, to pick another track at random.

2.3.5 Clock

Often when writing a game, you will want to schedule some game event to occur at a later time. For example, we may want a big boss alien to appear after 60 seconds. Or perhaps a power-up will appear every 20 seconds.

More subtle are the situations when you want to delay some action for a shorter period. For example you might have a laser weapon that takes 1 second to charge up.

We can use the `clock` object to schedule a function to happen in the future.

Let's start by defining a function `fire_laser` that we want to run in the future:

```
def fire_laser():
    lasers.append(player.pos)
```

Then when the fire button is pressed, we will ask the `clock` to call it for us after exactly 1 second:

```
def on_mouse_down():
    clock.schedule(fire_laser, 1.0)
```

Note that `fire_laser` is the function itself; without parentheses, it is not being called here! The clock will call it for us.

(It is a good habit to write out times in seconds with a decimal point, like `1.0`. This makes it more obvious when you are reading it back, that you are referring to a time value and not a count of things.)

`clock` provides the following useful methods:

class `Clock`

`schedule``(`*callback*, *delay*`)`
Schedule *callback* to be called after the given delay.

Repeated calls will schedule the callback repeatedly.

Parameters

- **`callback`** – A callable that takes no arguments.
- **`delay`** – The delay, in seconds, before the function should be called.

`schedule_unique``(`*callback*, *delay*`)`
Schedule *callback* to be called once after the given delay.

If *callback* was already scheduled, cancel and reschedule it. This applies also if it was scheduled multiple times: after calling `schedule_unique`, it will be scheduled exactly once.

Parameters

- **`callback`** – A callable that takes no arguments.
- **`delay`** – The delay, in seconds, before the function should be called.

`schedule_interval`(*callback*, *interval*)
Schedule *callback* to be called repeatedly.

Parameters

- **`callback`** – A callable that takes no arguments.
- **`interval`** – The interval in seconds between calls to *callback*.

`unschedule`(*callback*)
Unschedule callback if it has been previously scheduled (either because it has been scheduled with `schedule()` and has not yet been called, or because it has been scheduled to repeat with `schedule_interval()`.

Note that the Pygame Zero clock only holds weak references to each callback you give it. It will not fire scheduled events if the objects and methods are not referenced elsewhere. This can help prevent the clock keeping objects alive and continuing to fire unexpectedly after they are otherwise dead.

The downside to the weak references is that you won't be able to schedule lambdas or any other object that has been created purely to be scheduled. You will have to keep a reference to the object.

2.3.6 Actors

Once you have many images moving around in a game it can be convenient to have something that holds in one place the image and where it is on screen. We'll call each moving image on screen an `Actor`. You can create an actor by supplying at least an image name (from the images folder above). To draw the alien talked about above:

```
alien = Actor('alien', (50, 50))

def draw():
    screen.clear()
    alien.draw()
```

You can move the actor around by setting its pos attribute in an update:

```
def update():
    if keyboard.left:
        alien.x -= 1
    elif keyboard.right:
        alien.x += 1
```

And you may change the image used to draw the actor by setting its image attribute to some new image name:

```
alien.image = 'alien_hurt'
```

Actors have all the same attributes as *Rect*. If you assign a new value to one of those attributes then the actor will be moved. For example:

```
alien.right = WIDTH
```

will position the alien so its right-hand side is set to `WIDTH`.

Similarly, you can also set the initial position of the actor in the constructor, by passing one of these as a keyword argument: `pos`, `topleft`, `topright`, `bottomleft`, `bottomright`, `midtop`, `midleft`, `midright`, `midbottom` or `center`. For example:

```
alien = Actor('alien', midbottom=(100, 300))
```

If you don't specify an initial position, the actor will initially be positioned in the top-left corner (equivalent to `topleft=(0, 0)`).

Actors have an "anchor position", which is a convenient way to position the actor in the scene. By default, the anchor position is the center, so the `.pos` attribute refers to the center of the actor (and so do the `x` and `y` coordinates). It's common to want to set the anchor point to another part of the sprite (perhaps the feet - so that you can easily set the Actor to be "standing on" something):

```
alien = Actor('alien', anchor=('center', 'bottom'))
spaceship = Actor('spaceship', anchor=(10, 50))
```

`anchor` is specified as a tuple `(xanchor, yanchor)`, where the values can be floats or the strings `left`, `center`/`middle`, `right`, `top` or `bottom` as appropriate.

2.3.7 The Keyboard

You probably noticed that we used the `keyboard` in the above code. If you'd like to know what keys are pressed on the keyboard, you can query the attributes of the `keyboard` builtin. If, say, the left arrow is held down, then `keyboard.left` will be `True`, otherwise it will be `False`.

There are attributes for every key; some examples:

```
keyboard.a  # The 'A' key
keyboard.left  # The left arrow key
keyboard.rshift  # The right shift key
keyboard.kp0  # The '0' key on the keypad
keyboard.k_0  # The main '0' key
```

The full set of key constants is given in the Buttons and Keys documentation, but the attributes are lowercase, because these are variables not constants.

Deprecated since version 1.1: Uppercase and prefixed attribute names (eg. `keyboard.LEFT` or `keyboard.K_a`) are now deprecated; use lowercase attribute names instead.

New in version 1.1: You can now also query the state of the keys using the keyboard constants themselves:

```
keyboard[keys.A]  # True if the 'A' key is pressed
keyboard[keys.SPACE]  # True if the space bar is pressed
```

2.3.8 Animations

You can animate most things in pygame using the builtin `animate()`. For example, to move an *Actor* from its current position on the screen to the position `(100, 100)`:

```
animate(alien, pos=(100, 100))
```

`animate`(*object*, *tween='linear'*, *duration=1*, ***targets*)

Animate the attributes on object from their current value to that specified in the targets keywords.

Parameters

- **`tween`** – The type of *tweening* to use.
- **`duration`** – The duration of the animation, in seconds.
- **`on_complete`** – Function called when the animation finishes.

- **`targets`** – The target values for the attributes to animate.

The tween argument can be one of the following:

'linear'	Animate at a constant speed from start to finish
'accelerate'	Start slower and accelerate to finish
'decelerate'	Start fast and decelerate to finish
'accel_decel'	Accelerate to mid point and decelerate to finish
'end_elastic'	Give a little wobble at the end
'start_elastic'	Have a little wobble at the start
'both_elastic'	Have a wobble at both ends
'bounce_end'	Accelerate to the finish and bounce there
'bounce_start'	Bounce at the start
'bounce_start_end'	Bounce at both ends

The `animate()` function returns an `Animation` instance:

class `Animation`

`stop`(*complete=False*)
Stop the animation, optionally completing the transition to the final property values.

Parameters **`complete`** – Set the animated attribute to the target value.

`running`
This will be True if the animation is running. It will be False when the duration has run or the `stop()` method was called before then.

`on_finished`
You may set this attribute to a function which will be called when the animation duration runs out. The `on_finished` argument to `animate()` also sets this attribute. It is not called when `stop()` is called. This function takes no arguments.

2.4 Contributing to Pygame Zero

The Pygame Zero project is hosted on bitbucket:

https://bitbucket.org/lordmauve/pgzero

2.4.1 Development installation

It's possible to create a locally-editable install using pip. From the root directory of the checked out source, run:

pip3 install –editable .

The installed version will now reflect any local changes you make.

Alternatively, if you don't want to install it at all, it may be run with:

python3 -m pgzero <name of pgzero script>

For example:

python3 -m pgzero examples/demo1.py

2.4.2 Tests

The tests can be run with

```
python3 setup.py test
```

2.5 Changelog

2.5.1 1.1 - pending

- Added a spell checker that will point out hook or parameter names that have been misspelled when the program starts.
- New ZRect built-in class, API compatible with Rect, but which accepts coordinates with floating point precision.
- Refactor of built-in `keyboard` object to fix attribute case consistency. This also allows querying key state by `keys` constants, eg. `keyboard[keys.LEFT]`.
- Provide much better information when sound files are in an unsupported format.
- `screen.blit()` now accepts an image name string as well as a Surface object, for consistency with Actor.
- Fixed a bug with non-focusable windows and other event bugs when running in a virtualenv on Mac OS X.
- Actor can now be positioned by any of its border points (eg. `topleft`, `midright`) directly in the constructor.
- Added additional example games in the `examples/` directory.

2.5.2 1.0.2 - 2015-06-04

- Fix: ensure compatibility with Python 3.2

2.5.3 1.0.1 - 2015-05-31

This is a bugfix release.

- Fix: Actor is now positioned to the top left of the window if `pos` is unspecified, rather than appearing partially off-screen.
- Fix: repeating clock events can now unschedule/reschedule themselves

  Previously a callback that tried to unschedule itself would have had no effect, because after the callback returns it was rescheduled by the clock.

  This applies also to `schedule_unique`.
- Fix: runner now correctly displays tracebacks from user code
- New: Eliminate redraws when nothing has changed

  Redraws will now happen only if:

  – The screen has not yet been drawn
  – You have defined an update() function
  – An input event has been fired
  – The clock has dispatched an event

2.5.4 1.0 - 2015-05-29

- New: Added `anchor` parameter to Actor, offering control over where its `pos` attribute refers to. By default it now refers to the center.
- New: Added Ctrl-Q/-Q as a hard-coded keyboard shortcut to exit a game.
- New: `on_mouse_*` and `on_key_*` receive `IntEnum` values as `button` and `key` parameters, respectively. This simplifies debugging and enables usage like:

```
if button is button.LEFT:
```

2.5.5 1.0beta1 - 2015-05-19

Initial public (preview) release.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search