pygame-go Documentation

Release 0.1.0-alpha

Matthew Joyce

Mar 29, 2017

Contents:

CHAPTER 1

Tutorial

How do I import pygame-go?

Like this:

```
import pygame_go
```

Alternatively, to have a shorter name, use:

```
import pygame_go as pygo
```

The rest of this sheet will assume the shorter name.

How do I create a window to draw in?

You need to decide the width and height of the window. Say you wanted a window with a width of `600` and a height of `400`:

```
window = pygo.window(600, 400)
```

You can also write it like this:

```
window = pygo.window(width=600, height=400)
```

Or:

```
WINDOW_SIZE = (600, 400)
window = pygo.window(WINDOW_SIZE)
```

Or:

```
WINDOW_SIZE = (600, 400)
window = pygo.window(size=WINDOW_SIZE)
```

I made a window, but then it vanished again???

You need a main loop like this:

```
while window.active():
    window.update()
```

If you do not need to do any new drawing for each frame, you can write is like this:

```
window.loop_forever()
```

Why is the window always called "pygame-go"? I want to call it "MY EPIC THING"!

Do this:

```
window = pygo.window(size=WINDOW_SIZE, title="MY EPIC THING")
```

You can also do this:

```
window.title = "MY EPIC THING"
```

If you want to set the icon:

```
window = pygo.window(size=WINDOW_SIZE, icon=image)
```

Or:

```
window.icon = image
```

`image` can be any image you want.

How can I make the window fill the monitor?

Simple:

```
window = pygo.window(pygo.monitor_size())
```

If this covers your task bar, you can reduce the height a little bit:

```
w, h = pygo.monitor_size()
window = pygo.window(width=w, height=h - 80)
```

But how do I draw stuff? That white screen is boring!

Well, do some drawing in your main loop:

```
while window.active():
    # draw your stuff
    window.update()
```

To draw various things, look below! Remember, the window acts just like any other image. Anything you can do to an image, you can do to the window and vice-versa.

Eh? What's an image?

An image is something you can draw on. You can create images just like you created a window:

```
img = pygo.image(40, 30)
```

That will create an image that has a width of `40` and a height of `30`. A new image will be transparent. You can get the width and height of an image:

```
print("image width is", image.width, "and height is", image.height)
```

Or:

```
print("image size is", image.size)
```

If you want a copy of an image, use `image.copy`:

```
image_copy = image.copy()
```

What use are images?

You can use them to draw on the window! Say you had an image with a face draw on it, and you wanted to draw that face on the window several times. You can do that like this:

```
window.draw_image(face, x=0, y=0)
window.draw_image(face, x=100, y=100)
```

The x and y values specify where to draw the face. If you draw the face with `x=30, y=40` the top-left corner of the face image will be drawn at (30, 40).

How do I get an image of a face?

Well, one way is to have an image of a face, and load it. Say the image is called `/home/bob/face.jpg`. You could load it like this:

```
face = pygo.image("/home/bob/face.jpg")
```

Wow! What if I want to put the face in the middle of the screen? Or a corner?

To draw it in the center:

```
window.draw_image(face, window.center, align=pygo.center)
```

Or:

```
window.draw_image(face, window.center, align=face.center)
```

This says draw face such that the center of `face` is at the center of `window`. If you want to put the top-right corner of face at the center of window, do this:

```
window.draw_image(face, window.center, align=pygo.topright)
```

For the position to draw to you can pick any of:

```
window.center
window.topleft
window.topright
window.bottomleft
window.bottomright
```

For the align you can pick from:

```
pygo.center
pygo.topleft
pygo.topright
pygo.bottomleft
pygo.bottomright
```

Can I make my face bigger?

Just use `image.scale`. If you want it twice as big:

```
face.scale(2)
```

Or you want it twice as small:

```
face.scale(0.5)
```

You can also rotate it (clockwise):

```
face.rotate(90)
```

And flip it:

```
face.flip(vertical=True, horizontal=True)
```

`vertical=True` means that the image is reflected along the x-axis and `horizontal=True` means that the image is reflected along the y-axis.

But the white background is still there! I want it green!

Well, before drawing your faces, do this:

```
window.fill("green")
```

For specifying colors you can give a name:

```
window.fill("tomato")
```

Or an RGB combination:

```
window.fill(255, 127, 0)
```

If you need to fill an image with a see-though (transparent) color:

```
image.fill(255, 0, 0, 127)
```

That will fill image with red and will be 50% transparent. You can also specify the fill color when creating the image:

```
img = pygo.image(40, 30, color="red")
```

And the same for the window:

```
window = pygo.window(size=WINDOW_SIZE, color="green")
```

Ooo! Do I have to make an image if I want to draw a rectangle? It sounds like a lot of work...

No! Say you want to draw a rectangle onto an image. You want the rectangle's top-left corner to be at (10, 20) and you want is to have a width of 50 and a height of 10. You want it filled with blue. Then do:

```
image.draw_rect(x=10, y=20, width=50, height=10, color="blue")
```

You can also write it like:

```
image.draw_rect(position=(10, 20), size=(50, 10), color=(0, 0, 255))
```

But it is less clear that way. You can use `align` with `draw_rect`:

```
image.draw_rect(position=(10, 20), size=(50, 10), color="blue", align=pygo.
↪bottomright)
```

This means that `position` will be the bottom-right of the draw rectangle.

A border! I want a blue rectangle with a yellow border!

Sure! First draw your blue rectangle:

```
image.draw_rect(x=10, y=20, width=50, height=10, color="blue")
```

Then draw your border:

```
image.draw_hollow_rect(x=10, y=20, width=50, height=10, color="blue")
```

This will draw a border that is 1 pixel thick. Want a wider border? Let's say 5 pixels:

```
image.draw_hollow_rect(x=10, y=20, width=50, height=10, color="blue", thickness=5)
```

Using `align`:

```
image.draw_hollow_rect(x=10, y=20, width=50, height=10, color="blue", thickness=5,␣
↪align=pygo.bottomright)
```

Yay! How about a circle? A black one!

To draw a circle at (40, 40) with radius 20 you do:

```
image.draw_circle(x=40, y=40, radius=20, color="black")
```

Remember that you can also specify positions like this:

```
image.draw_circle(position=image.center, radius=20, color="black")
```

Can circles have borders too?

Yup, just like rectangles. Do draw a cyan border of thickness 10 do:

```
image.draw_hollow_circle(position=image.center, radius=20, color="cyan", thickness=10)
```

Triangles?

Use the `draw_polygon` function:

```
image.draw_polygon(points=[(50, 0), (100, 70), (0, 70)], color="tomato")
```

Outlines work too:

```
image.draw_hollow_polygon(points=[(50, 0), (100, 70), (0, 70)], color="blue",␣
↪thickness=10)
```

Any other shapes?

Yes! You can draw ellipses:

```
window.draw_ellipse(position=window.center, radius_x=100, radius_y=50, color="blue")
window.draw_hollow_ellipse(position=window.center, radius_x=100, radius_y=50, color=
↪"blue", thickness=5)
```

Eh, thinking up color names is a pain. Is there a list somewhere?

Yes there is! It is called `pygo.color_names`. Want a random color? Just this way:

```
import random
random.choice(pygo.color_names)
```

Cool! I want to write my name. How?

Just like this:

```
image.draw_text(text="my name", color="black", position=image.topleft)
```

Make sure your image is big enough!

Make my name bold! And italic!

Just like this:

```
image.draw_text(text="my name", color="black", position=image.topleft,
                italic=True, bold=True)
```

Note! This may not change anything unless you change the font as well. To use a different font, set it like this:

```
image.draw_text(text="my name", color="black", position=image.topleft,
                italic=True, bold=True, font="dejavusans")
```

Make my name BIGGER!

OK, OK, here's font size 60:

```
image.draw_text(text="my name", color="black", position=image.topleft,
                italic=True, bold=True, font="dejavusans", size=60)
```

Ha! Show me how to put "YOU DIED!" in the middle of the window!

`draw_text` accepts the same align arguments as `draw`, so do it the same way:

```
window.draw_text(text="YOU DIED!", position=window.center, color="red", size=60,␣
↪align=pygo.center)
```

What if I want to draw a line from A to B?

Well, lets say A and B are coordinates, any you want to draw a red line that has a thickness of 3:

```
A = 20, 30
B = 40, 60
image.draw_line(start=A, end=B, color="red", thickness=3)
```

My program doesn't do much. How can I check if a key is pressed?

Modify your loop to look like this:

```
while window.active():
    for event in window.events():
        # handle events here
    # drawing here
    window.update()
```

To check for a key press, replace `# handle events here` with:

```
if event.is_key():
    print("You pressed", event.key)
```

I just want to check for the space bar, not everything!

Do this:

```
if event.is_key() and event.key == " ":
    print("You pressed the space bar")
```

You can compare to any string you want. If you want to check for the "a" key, do:

```
if event.is_key() and event.key == "a":
    print("You pressed the a key")
```

Some special keys:

If you are looking for	Test for
Return key	`"\n"`
Space bar	`" "`
Shift key	`"<Shift>"`
Ctrl key	`"<Ctrl>"`
Meta (windows) key	`"<Meta>"`
Left arrow	`"<Left>"`
Right arrow	`"<Right>"`
Up arrow	`"<Up>"`
Down arrow	`"<Down>"`
Escape key	`"<Escape>"`
Delete key	`"<Delete>"`
Function keys	`"<F1>"`, `"<F2>"`, ..., `"<F12>"`

How about if they press the mouse?

You can do this:

```
if event.is_mouse_click():
    print("You clicked a mouse button at", event.x, event.y)
```

What about just the left mouse button?

For the left button:

```
if event.is_mouse_click() and event.button is pygo.left_button:
    print("You clicked the left mouse button at", event.position)
```

Right button:

```
if event.is_mouse_click() and event.button is pygo.right_button:
    print("You clicked the right mouse button at", event.position)
```

Middle button:

```
if event.is_mouse_click() and event.button is pygo.middle_button:
    print("You clicked the middle mouse button at", event.position)
```

Scrolling! What about that?

Do this:

```
if event.is_scroll():
    print("You scrolled", event.direction, "at", position)
```

`event.direction` will be one of:

```
pygo.up_scroll
pygo.down_scroll
pygo.left_scroll
pygo.right_scroll
```

What about if they move the mouse?

Write your code like this:

```
if event.is_mouse_motion():
    print("You moved the mouse from", event.start, "to", event.end)
```

You can also see how much the mouse moved:

```
if event.is_mouse_motion():
    print("You moved the mouse by", event.moved_by_x, event.moved_by_y)
```

If you want to see if any buttons were pressed during the movement, test them using `event.is_pressed`:

```
if event.is_pressed(pygo.left_button):
    print("Drag with left button")
elif event.is_pressed(pygo.right_button):
    print("Drag with right button")
elif event.is_pressed(pygo.middle_button):
    print("Drag with middle button")
```

Just tell me where the mouse is now!

Use `pygo.mouse_position`:

```
print("The mouse is at", pygo.mouse_position())
```

Can I move where the mouse is?

Use `pygo.set_mouse_position`:

```
pygo.set_mouse_position(window.center)
```

What about keys? Can I test for them without looking though the events?

Yes. To test for a key, use:

```
if pygo.is_key_pressed("<Shift>"):
    print("Shift is pressed")
```

To test for mouse button, do:

```
if pygo.is_button_pressed(pygo.left_button):
    print("Left mouse button is pressed")
```

I made a snake program, and the snake went really fast!

When you create your window, you can change how fast it updates:

```
window = pygo.window(WINDOW_SIZE, frame_rate=5)
```

`frame_rate` is normally 20. You can make it smaller to slow the game down or larger to speed it up.

Can I tell which frame I am on?

Look at `window.frame_number`:

```
print("You are on frame", window.frame_number)
```

You can use this like a timer, but it will not be very accurate:

```
print("Game playing for", window.frame_number / window.frame_rate)
```

How can I stop the game when the player looses?

Call `window.stop`:

```
if player_lost:
    window.stop()
```

OK, last thing. I want explosion noises!

Sure. If you call your sound file `/home/bob/explosion.wav`, load it like this:

```
explosion = pygo.sound("/home/bob/explosion.wav")
```

Play it using:

```
explosion.play()
```

To stop playing:

```
explosion.stop()
```

You can check if the sound is currently playing:

```
if explosion.is_playing():
    print("BOOM!")
```

To set the volume of the sound at 50%:

```
explosion.volume = 0.5
```

If you need the length of the sound:

```
print("Explosion is", explosion.length, "seconds long")
```

Can I make a sound repeat?

Yup. To make it repeat 10 times, use:

```
explosion.play(times=10)
```

And pause?

Use:

```
explosion.pause()
```

To unpause:

```
explosion.unpause()
```

To check is the sound is paused, use:

```
if explosion.is_paused():
    print("Paused")
```

Really last thing. How can I make it repeat FOREVER!

Simply:

```
explosion.play(forever=True)
```

The sound will only stop if you call (or play too many sounds at once):

```
explosion.stop()
```

CHAPTER 2

Documentation

A note on notation

In the following documentation, the following syntax is used for method signatures:

```
func(a, <size>, *, <align>, b, [thickness=0, <color="white">])
```

This means that:

- `a` is a non-optional argument that can be given positionally.
- `<size>` means a size-like (more on that later) argument, which can be given positionally.
- Any arguments after `*` have to be given using keyword arguments.
- `<align>` means an align-like argument that must be given using keyword arguments.
- `b` is a keyword-only argument that must be given.
- If a parameter is inside square brackets, it is optional and a default used if you do not pass it. If an argument is not inside square brackets it must be given, regardless of whether it is keyword-only or not.
- `thickness` is a keyword-only argument that is optional, and has default `0`.
- `<color>` is a color-like argument that is optional, and has default `"white"`.

`<x>`-like arguments:

- **`<position>`**
  - Can be given as either `x` and `y` or `position`.
  - `x` and `y` must be `int`s or `float`s.
  - `position` must be a 2-`tuple` of `int`s or `float`s.
- **`<size>`**
  - Can be given as either `width` and `height` or `size`.

  - `width` and `height` must be `int`s or `float`s.
  - `size` must be a 2-`tuple` of `int`s or `float`s.

- **<align>**
  - Can be given as either `align` or `align_x` and `align_y`.
  - `align` can be a position or one of *`topleft`*, *`topright`*, *`bottomleft`*, *`bottomright`* or *`center`*.
  - `align_x` and `align_y` must be `int`s or `float`s.

- **<color>**
  - Can be given as either `color` or `r`, `g`, `b` and optionally `a`
  - `color` must either be a `color`, a `str` or a 3-`tuple` of of `int`s or `float`s.
  - If `color` is a `str` the name is looked up in `color.colors` and an error raised if it is not found.
  - `r`, `g`, `b` and optionally `a` must be `int`s or `float`s in the range 0-255. `a` is the transparency.

Constants

left_button
middle_button
right_button
Enumeration representing buttons, used in *`ClickEvent`* and `is_mouse_pressed()`.

up_scroll
down_scroll
left_scroll
right_scroll
Enumeration representing scroll directions, used in *`ScrollEvent`*.

topleft
topright
bottomleft
bottomright
center
Enumeration representing alignment, use for `<align>` parameters.

color_names
List of all the color names recognised.

`color.`**colors**
A dictionary of color names to *`color`*s.

Classes

class image

__init__(*fname*)

Parameters **fname** (*`str or pathlib.Path`*) – Path to image file.

Load an image from a file.

> **Warning:** A window must be created before this function is called! A `RuntimeError` will be raised otherwise.

__init__(<*size*>, *[, <*color="transparent"*>])
Create a new image of size `<size>`. If `<color>` is given, it will be filled with that color, otherwise it will be transparent.

size
Type: 2-tuple of `int`

The width and height of the image. This attribute is not settable.

width
Type: `int`

The width of the image. This attribute is not settable.

height
Type: `int`

The height of the image. This attribute is not settable.

center
Type: 2-tuple of `int`

The position at the center of the image. This attribute can be used as a `<position>` or `<align>`. This attribute is not settable.

topleft
Type: 2-tuple of `int`

The position at the top-left of the image. This attribute can be used as a `<position>` or `<align>`. This attribute is not settable.

topright
Type: 2-tuple of `int`

The position at the top-right of the image. This attribute can be used as a `<position>` or `<align>`. This attribute is not settable.

bottomleft
Type: 2-tuple of `int`

The position at the bottom-left of the image. This attribute can be used as a `<position>` or `<align>`. This attribute is not settable.

bottomright
Type: 2-tuple of `int`

The position at the bottom-right of the image. This attribute can be used as a `<position>` or `<align>`. This attribute is not settable.

copy()

Return type *image*

Returns a copy of the image. Changes to the image will not affect the copy.

fill(<*color*>)

Return type None

The entire image is set to <color>.

draw_image(*source*, <*position*>, *[, <*align=topleft*>])

Return type None

Draw source onto this image such that the point on the source indicated by <align> is at <position>. E.g.:

```
image.draw_image(other, image.bottomright, align=bottomright)
```

Will draw other onto image such that the bottom-right of other is at the bottom-right of image.

draw_rect(*, <*position*>, <*size*>, <*color*>[, <*align=topleft*>])

Return type None

Draw a rectangle of color <color> and size <size> such that <align> is at <position>. The <align> works the same as for *draw_image()*.

draw_hollow_rect(*, <*position*>, <*size*>, <*color*>[, *thickness=1*, <*align=topleft*>])

Return type None

Draw a border of thickness thickness and color <color> in the rectangle defined by <size>, <position> and <align>. The rectangle is defined in the same way as for *draw_rect()*.

draw_circle(*, <*position*>, <*color*>, *radius*)

Return type None

Draw a circle of color <color> with radius radius with its center at <position>.

draw_hollow_circle(*, <*position*>, <*color*>, *radius*[, *thickness=1*])

Return type None

Draw a circular border of color <color> with radius radius and thickness thickness with its center at <position>.

draw_ellipse(*, <*position*>, <*color*>, <*radii*>)

Return type None

Draw a ellipse of color <color> with radius radius with its center at <position>. Its radii are taken from radii or radius_x and radius_y.

draw_hollow_ellipse(*, <*position*>, <*color*>, <*radii*>[, *thickness=1*])

Return type None

Draw a ellipse-shaped border of color <color> with radius radius and thickness thickness with its center at <position>. Its radii are taken from radii or radius_x and radius_y.

draw_line(*, <*start*>, <*end*>, <*color*>[, *thickness=1*])

Return type None

Draw a line from <start> to <end> with color <color> and thickness thickness. For <start>, provide start or start_x and start_y. For <end>, provide end or end_x and end_y. <start> and <end> work the same as <position> in every other way.

draw_arc(*, *start_angle*, *end_angle*, <*position*>, <*color*>, <*radii*>[, *thickness=1*])

Return type None

Draw part of a hollow ellipse defined by <position> and <radii> (see *draw_hollow_ellipse()*) from start_angle to end_angle clockwise. The angles are in degrees, with 0 being directly above the center of the ellipse.

draw_polygon(*points*, *, <*color*>)

Return type None

Draw a polygon with the vertices in points using <color>.

draw_hollow_polygon(*points*, *, <*color*>[, *thickness=1*])

Return type None

Draw a hollow polygon with the vertices in points using <color> and thickness thickness.

draw_text(*, *text*, <*position*>, <*color*>[, *text*, *size=30*, *font=None*, *bold=False*, *italic=False*, <*align=topleft*>])

Return type None

Draw text text in color <color> at <position>. <align> works the same as for *draw_rect()*. size is the height of the font. If font is None, the default font will be used. Otherwise a font called font will be searched for and a ValueError raised if it cannot be found. bold and italic set the function to use the bold and italic variants of the font.

Note: bold and italic may not work on all fonts, especially the default font. If you cannot see any change when using bold or italic, try changing to a different font.

---

flip([*vertical=False*, *horizontal=False*])

Return type None

Flip the image vertically if vertical is True and horizontally if horizontal is True.

rotate(*angle*)

Return type None

Rotate the image by angle degrees clockwise.

scale(*times*)

Return type None

Enlarge the image by factor times. The image will then have a width of times * old_width and a height of times * old_height.

color_at(<*position*>)

Return type *color*

Returns the color of the pixel at <position>

class window

Bases: *image*

__init__(<*size*>, *[, <*color="white"*>, *frame_rate=20*, *autoquit=True*, *title="pygame-go"*, *icon=None*])

Create the window with the size <size>. If <color> is given, the window will be filled with that color, otherwise it is filled with white. frame_rate is the number of updates per second, which is controlled during the *update()* method call. If autoquit is True, then quit events will be processed automatically and *active()* will return False without any event processing by the user. If autoquit

is `False`, the quit events will be accessible though *`events()`*. `title` will be used to set the window title, see *`title`*. `icon` will be used to set the window icon, see *`icon`*.

`active`()

Return type bool

Returns whether the window has quit or not. This should be used in your main loop so that your program exits when the user presses the quit button.

`stop`()

Return type None

Makes *`active()`* return `False`, stopping the program.

`update`()

Return type None

Updates the window, showing the graphics on the window to the user. This function will then delay by the correct amount of time to maintain the correct frame rate.

`loop_forever`()

Return type None

Keep updating the window until the user quits. As no event handling can be done in this function, only use it if you only want to show a static image.

`has_events`()

Return type bool

Returns `True` if there are unprocessed events.

`next_event`()

Return type *Event*

Returns the next event to be processed. Raises a `ValueError` if there are no more events.

`events`()

Return type Iterable[*Event*]

Returns an iterator that yields events in the queue until the queue is empty. This is the preferable way to access events.

`title`

Type: `str`

The title of the window. This attribute is settable, and setting a new value will change the window title.

`icon`

Type: *`image`*

The icon of the window, used in the task bar. This attribute is settable, and setting a new value will change the window icon.

Note: May not work with all DEs

---

class `sound`

`__init__`(*fname*)

Parameters **fname** (*str or pathlib.Path*) – Path to sound file.

Load an sound from a file.

Note: Only `.ogg` and `.wav` files can be loaded. This may change in future releases.

---

play([*times=1*, *forever=False*])

Return type None

Play the sound, repeating it `times` times. If `forever` is `True`, the sound will repeat until *stop()* is called.

stop()

Return type None

Stop the sound. This will also unpause the sound.

pause()

Return type None

Pause the sound if it is not already paused. It can be resumed with *unpause()*.

unpause()

Return type None

If the sound has been paused, unpause it.

is_playing()

Return type bool

Returns whether the sound is currently playing.

is_paused()

Return type bool

Returns whether the sound is currently paused.

length
Type: `float`

The length of the sound in seconds. This attribute is not settable.

volume
Type: `float`

The volume of the sound. This attribute can be set in order to change the volume it is played at.

class color

__init__(<*color*>)
Create a new color.

classmethod fromhex(*value*)
Create a color from a HTML-style color.

r
Type: `int`

The red component of the color. It will be in the range 0-255. This attribute is settable.

g
Type: int

The green component of the color. It will be in the range 0-255. This attribute is settable.

b
Type: int

The blue component of the color. It will be in the range 0-255. This attribute is settable.

transparency
Type: int

The transparency component of the color. It will be in the range 0-255. This attribute is settable.

hex
Type: str

The HTML-style hex representation of this color. This attribute is not settable.

class Event

Note: This type should not be created. Rather, use *window.events()*.

is_mouse_press()

Return type bool

Returns whether this event is a *ClickEvent*.

is_mouse_scroll()

Return type bool

Returns whether this event is a *ScrollEvent*.

is_quit()

Return type bool

Returns whether this event is a *QuitEvent*.

is_mouse_motion()

Return type bool

Returns whether this event is a *MotionEvent*.

is_key()

Return type bool

Returns whether this event is a *KeyEvent*.

class ClickEvent
Bases: *Event*

Note: This type should not be created. Rather, use *window.events()*.

position
Type: 2-tuple of int

The position of the click.

x
Type: `int`

The x-coordinate of the click.

y
Type: `int`

The y-coordinate of the click.

button
Type: One of *left_button*, *right_button* or *middle_button*

The button that was pressed down.

class ScrollEvent
Bases: *Event*

Note: This type should not be created. Rather, use *window.events()*.

---

position
Type: 2-tuple of `int`

The position of the scroll.

x
Type: `int`

The x-coordinate of the scroll.

y
Type: `int`

The y-coordinate of the scroll.

direction
Type: One of *up_scroll*, *down_scroll*, *left_scroll* or *right_scroll*

The direction of the scroll.

class MotionEvent
Bases: *Event*

Note: This type should not be created. Rather, use *window.events()*.

---

start
Type: 2-tuple of `int`

The position the mouse started moving from.

start_x
Type: `int`

The x-coordinate of *start*.

start_y
Type: `int`

The y-coordinate of *start*.

end
Type: 2-tuple of `int`

The position the mouse moved to.

end_x
Type: `int`

The x-coordinate of *end*.

end_y
Type: `int`

The y-coordinate of *end*.

moved_by
Type: 2-tuple of `int`

The amount of movement in the x and y direction

moved_by_x
Type: `int`

The amount of movement in the x direction.

moved_by_y
Type: `int`

The amount of movement in the y direction.

buttons
Type: `set` containing some of *left_button*, *right_button* and *middle_button*

The buttons that were pressed during the motion. See *is_pressed()*.

is_pressed(*[button=None*)

Return type bool

If `button` is one of *left_button*, *right_button* or *middle_button*, returns `True` if that button was pressed during the motion. If `button` is `None`, return `True` if any buttons were pressed during the motion.

class KeyEvent
Bases: *Event*

Note: This type should not be created. Rather, use *window.events()*.

---

key
Type: `str`

The key that was pressed. It can either be a single ASCII character or a modifier / non-printable key like `<Shift>` or `<Ctrl>`. See `pygame_go/events.py` for the full listing.

class QuitEvent
Bases: *Event*

Note: This type should not be created. Rather, use *window.events()*.

Other functions

mouse_position()

Return type 2-tuple of int

Returns the current mouse position.

set_mouse_position(<*position*>)

Return type None

Sets the current mouse position.

is_key_pressed(*key*)

Return type bool

Returns whether the key `key` is currently pressed. `key` should be in the same form as for *KeyEvent*.

is_button_pressed(*button*)

Return type bool

Returns whether the button `button` is currently pressed. `button` should be one of *left_button*, *right_button* or *middle_button*.

monitor_resolution()

Return type 2-tuple of int

Width and height of the monitor.

collides_rect_rect(*, <*position_a*>, <*size_a*>, <*align_a*>, <*position_b*>, <*size_b*>, <*align_b*>)

Return type bool

Returns `True` if the rectangle `a` defined by `<position_a>`, `<size_a>` and `<align_a>` collides with the rectangle `b` defined by `<position_b>`, `<size_b>` and `<align_b>`. Arguments follow the above conventions, with `_a` or `_b` added on the end.

collides_circle_circle(*, <*position_a*>, *radius_a*, <*position_b*>, *radius_b*)

Return type bool

Returns `True` if the circle `a` defined by `<position_a>` and `radius_a` collides with the rectangle `b` defined by `<position_b>``and ``radius_b`. Arguments follow the above conventions, with `_a` or `_b` added on the end.

Index

Symbols

A

B

C

D

E

F

G

H

I