

Migrating from pygame 2.0

This page will help you upgrade your project from pygame 2.0.

Some of pygame 2.1's improvements required small breaking changes, including:

- Arguments for text, UI, and game controller handling
- Locations and names for features
- Data types and annotations
- Changes to math classes

To report a missing change or a bug, please use [GitHub Issues](#) or another [contributor communication](#) channel.

Setting pygame Options

The `pygame.options` attribute is now a `pygame.Options` instance.

The Options Object

Although `Options` is a `dataclass`, you can get and set its attribute values directly via either approach below:

```
# "New" 2.1 attribute-style works with type checkers
pygame.options.dpi_scaling = 'real'

# "Old" dict-style access is backward-compatible to help with porting
pygame.options['dpi_scaling'] = 'real'
```

Window “HiDPI” support

In 2.1, `pygame.options.dpi_scaling` accepts strings to configure fine-grained ‘HiDPI’ behavior.

What is HiDPI?

 [latest](#) ▼

Some systems require scaled drawing due to pixel density, settings, or both.

Note

HiDPI stands for High Dots Per (*Square*) Inch.

In practice, this usually means:

- Apple's "retina" displays on Macs
- Non-Mac displays branded as 4K, 8K, etc
- Other displays with non-100% zoom or scaling

Scaling Modes

The `pyglet` `dpi_scaling` option supports multiple scaling modes.

Each has pros and cons for different platforms and hardware. For example, the default `"real"` mode matches the window, rendering, and input to a computer display's "real" physical pixels.

On a HiDPI screen, this could mean:

- projects with the right art styles can look crisply detailed
- other projects may look tiny or poorly-scaled

In the latter case, other scaling modes may help. You will need to experiment to find the best one(s) for your specific needs. Please see the following to get started:



- `pyglet.options.dpi_scaling`
- `pyglet.Options`
- `Window.dpi`
- `Window.scale`

Labels & Text Layouts

Argument Consistency

The positional arguments for creating `pyglet.text` layouts and labels now all *start* with similar argument orders. This helps you:

- switch between labels and layouts
- create custom subclasses

The order *after* the initial arguments may differ. Please see any relevant API docs  [latest](#)  learn more.

Layout Arguments

All `pyglet.text.layout` types now *start* with the same positional argument order:

```
TextLayout(document, x, y, z, width, height, anchor_x, anchor_y, rotation, ...)
ScrollableTextLayout(document, x, y, z, width, height, anchor_x, anchor_y, rotation, ...)
IncrementalTextLayout(document, x, y, z, width, height, anchor_x, anchor_y, rotation, ...)
```

These types all take a concrete instance of an `AbstractDocument` subclass as their first argument. Subsequent arguments may differ.

Please see the following to learn more:

- `pyglet.text.layout.TextLayout`
- `pyglet.text.layout.ScrollableTextLayout`
- `pyglet.text.layout.IncrementalTextLayout`

Label Arguments

The label classes now also share similar early argument orders.

Only `DocumentLabel` is identical to layouts in its initial arguments. The others both take a string `text` argument as their first argument:

```
DocumentLabel(document, x, y, z, width, height, anchor_x, anchor_y, rotation, ...)
Label(text, x, y, z, width, height, anchor_x, anchor_y, rotation, ...)
HTMLLabel(text, x, y, z, width, height, anchor_x, anchor_y, rotation, ...)
```

As with layouts, the subsequent arguments may vary. Please see the following to learn more:

- `pyglet.text.DocumentLabel`
- `pyglet.text.Label`
- `pyglet.text.HTMLLabel`

Replace Bold With Weight

The string `weight` argument is more flexible than the `bold` argument it replace:

 [latest](#) ▼

This does not apply to `HTMLLabel`.

For `pyglet.text.Label` and `pyglet.text.DocumentLabel`, their `weight` argument now allows choosing a desired font weight. This assumes your specific font and weight combination is:

1. Loaded
2. Supported by the font
3. Supported by the rendering back-end

For known cross-platform `weight` strings, please see `pyglet.text.Weight`.

- Constant names and values mimic OpenType and CSS (`"bold"`, `"thin"`, `"extrabold"`, etc)
- Some rendering back-ends *may* support additional weight string values

Shapes

For consistency with the rest of the library, it was decided to represent all angles in degrees instead of radians. Previously we had a mix of both, which lead to some confusion. Using degrees also makes the API consistent with Sprites and other rotatable objects, which have long used degrees.

The arguments for `Line` have changed slightly. Instead of “width”, we now use “thickness”. This matches with other shapes that are made up of line segments. For example the `Box` shape, which already uses “width” (and height) to mean it’s overall size. Going forward, any shape that is made up of lines should use *thickness* for the thickness/width of those lines.

Controllers

Events from analog sticks and directional pads (d-pads) now pass `Vec2` instances to handler functions.

Vectors offer helper methods in addition to common math operators (+, -, *, etc). Since this makes many tasks easier, we’ll cover the most common ones below.

Handling Diagonals with the D-Pad

`Vec2.normalize()` makes it easy to handle diagonal movement:

```
# In pygame 2.1, this handles diagonals
@controller.event
def on_dpad_motion(controller, vector):
    # Multiplying a vector by an integer or float multiplies all components
    player_position += vector.normalize() * PLAYER_SPEED
```

Without vectors, pre-2.1 code was more verbose:

```
@controller.event
def on_dpad_motion(controller, dleft, dright, dpup, dpdown):
    if dleft:
        # move left
    if dright:
        # move right
    if dright and dpdown:
        # move diagonally, but you have to normalize the values by yourself
```

Getting D-Pad Booleans

If you need boolean data, you can quickly convert from a `Vec2` like this:

```
dleft, dright, dpup, dpdown = vector.x < 0, vector.x > 0, vector.y > 0, vector.y < 0
```

Handling Analog Stick Drift

Analog sticks can “drift” when near zero, but vectors can help.

Circular Dead Zones

The simplest approach to drift is a circular “dead zone” which ignores input with a `length()` beneath a certain threshold:

```
@controller.event
def on_stick_motion(controller, name, vector):
    if vector.length() <= DEADZONE_RADIUS:
        return
    elif name == "leftstick":
        # Do something with the 2D vector
    elif name == "rightstick":
        # Do something with the 2D vector
```

 [latest](#) ▼

Non-Circular Sticks

`Vec2.normalize` <`pyglet.math.Vec2.normalize()`> can also help when an unusual analog stick input could exceed `1.0` in length.

For example, a `Controller` for a device with a non-circular input range could return a value with a combined `length()` greater than `1.0`. Normalizing allows concisely clamping the input to `1.0`:

```
# Avoid a "cheating" / bugged controller for movement
vector = min(vector, vector.normalize())
player.position += vector * PLAYER_SPEED
```

Accessing Vector Components

You can directly access individual `x` and `y` attributes or unpack a vector:

```
# Direct access
x = vector2.x
y = vector2.y

# Unpacking-style access
x, y = vector2
```

Please see the following to learn more about vectors in pyglet 2.1:

- The [Math](#) section of this page
- `pyglet.math.Vec2`


Gui

Widget Event Dispatching

All widget events now dispatch the widget instance itself as the first argument.

This is similar to how `Controller/Joystick` events are implemented. It allows you to re-use a single handler functions across multiple widgets without “forgetting” which widget dispatched an event.

Button Argument Names

The `ToggleButton` and `PushButton` widgets now use `pressed` and `unpressed` for arguments.  [latest](#) ▼

Math

The `math` module includes a number of performance and usability changes.

Immutable Vectors and Matrices

All `math` datatypes are now `typing.NamedTuple` subclasses. This provides multiple benefits:

- More consistent creation syntax
- Vectors and matrices are now hashable
 - They can be `dict` keys or `set` members
 - Combine with `/`, `//`, `round()` or `math.floor()` for easy spatial hashing
- Cleaner controller code (see [Controllers](#))

! Important

The mypy typechecker is incompatible with `pyglet.math`.

When typechecking, it is a good idea to:


1. exclude `pyglet.math` from mypy checks
2. use pyright instead (pylance in VS Code)

Vector Changes

The syntax for the Vec types has changed in several ways. Some of these changes are due to becoming `typing.NamedTuple` subclasses, while others were done for general usability. Where possible, we adopt the behavior of GLM/GLSL for most operations, for a more familiar experience for computer graphics programmers.

- The arguments for `Vec2.from_polar` have been reversed for consistency. The `length` argument also now defaults to 1.0. This will fail silently, so take care to correct this if you are using this method in your code:

```
Vec2.from_polar(angle: 'float', length: 'float' = 1.0)
```

- Vector length is now obtained from the new `length()` method. Previously `l`  [latest](#) could be used, but this is no longer the case. The `len` function will now give you the number of items in the vector (ie: 2, 3, or 4), not the vector length. The `abs` function will give you a

new vector with absolute values:

```
>>> vec = Vec2(1, 9)
>>> vec.length()
9.055385138137417
>>> len(vec)
2
>>>
>>> vec = Vec2(-10, 5)
>>> abs(vec)
Vec2(x=10, y=5)
```

- The vector `heading` property has been replaced with the `heading()` function. This is to better indicate that this is a calculation, not a static attribute. The function call is also marginally faster.
- The `mag` property has been removed. The `length()` function should be used in its place.
- The `Vec2.from_magnitude` function has been removed. For creating a new vector of a certain magnitude, you can simply multiply a normalized vector by the desired length. For example:

```
>>> vec = Vec2(1, 9)
>>> vec.normalize()
Vec2(x=0.11043152607484653, y=0.9938837346736188)
>>> vec.normalize() * 2
Vec2(x=0.22086305214969307, y=1.9877674693472376)
```

Matrix Creation Syntax

`Mat3` and `pyglet.math.Mat4` now accept arguments directly instead of an iterable.

If you create your matrices via the helper methods, nothing changes. If you create matrices directly, pyglet 2.1 allows more efficient code:

```
# pyglet 2.1 requires passing the elements directly
my_mat4 = pyglet.math.Mat4(1, 2, 3, 4, 5, ...)
```

pyglet 2.0 required an intermediate iterable like a list

```
my_mat4 = pyglet.math.Mat4([1, 2, 3, 4, 5, ...])
```

If your pre-existing code has an `intermediate_iterable`, you can use `*` unpacking as a quick fix:

```
# Use * unpacking to unpack the pre-allocated intermediate_iterable
my_mat4 = pyglet.math.Mat4(*intermediate_iterable)
```

 [latest](#) ▼

Models

The `model` module has seen some changes. This is an undocumented WIP module for pyglet 2.0, and it remains so pyglet 2.1. That said, it's in a more usable state now. The first change is that `load()` now returns a `Scene` object instead of a `Model` object. The `Scene` is a new, “pure data” intermediate representation of a 3D scene, that closely mimics the layout of the glTF format. The `create_models()` method can be used to create `Model` instances from the `Scene`, but the `Scene` data can also be manually iterated over for more advanced use cases.

Canvas module

The `pyglet.canvas` module has been renamed to `pyglet.display`. The “canvas” concept was a work-in-progress in legacy pyglet, and was never fully fleshed out. It appears to have been meant to allow arbitrary renderable areas, but this type of functionality can now be easily accomplished with Framebuffers. The name `display` is a more accurate representation of what the code in the module actually relates to. The usage is the same, with just the name change:

```
my_display = pyglet.canvas.get_display()    # old pyglet 2.0
my_display = pyglet.display.get_display()    # new pyglet 2.1
```

pgMustard: a Postgres query plan visualization tool that gives advice. [Try it for free.](#)

Ads by EthicalAds