pygame.sprite

pygame module with basic game object classes

- pygame.sprite.Sprite — Simple base class for visible game objects.
- pygame.sprite.DirtySprite — A subclass of Sprite with more attributes and features.
- pygame.sprite.Group — A container class to hold and manage multiple Sprite objects.
- pygame.sprite.RenderPlain — Same as pygame.sprite.Group
- pygame.sprite.RenderClear — Same as pygame.sprite.Group
- pygame.sprite.RenderUpdates — Group sub-class that tracks dirty updates.
- pygame.sprite.OrderedUpdates — RenderUpdates sub-class that draws Sprites in order of addition.
- pygame.sprite.LayeredUpdates — LayeredUpdates is a sprite group that handles layers and draws like OrderedUpdates.
- pygame.sprite.LayeredDirty — LayeredDirty group is for DirtySprite objects. Subclasses LayeredUpdates.
- pygame.sprite.GroupSingle — Group container that holds a single sprite.
- pygame.sprite.spritecollide — Find sprites in a group that intersect another sprite.
- pygame.sprite.collide_rect — Collision detection between two sprites, using rects.
- pygame.sprite.collide_rect_ratio — Collision detection between two sprites, using rects scaled to a ratio.
- pygame.sprite.collide_circle — Collision detection between two sprites, using circles.
- pygame.sprite.collide_circle_ratio — Collision detection between two sprites, using circles scaled to a ratio.
- pygame.sprite.collide_mask — Collision detection between two sprites, using masks.
- pygame.sprite.groupcollide — Find all sprites that collide between two groups.
- pygame.sprite.spritecollideany — Simple test if a sprite intersects anything in a group.

This module contains several simple classes to be used within games. There is the main Sprite class and several Group classes that contain Sprites. The use of these classes is entirely optional when using pygame. The classes are fairly lightweight and only provide a starting place for the code that is common to most games.

The Sprite class is intended to be used as a base class for the different types of objects in the game. There is also a base Group class that simply stores sprites. A game could create new types of Group classes that operate on specially customized Sprite instances they contain.

The basic Sprite class can draw the Sprites it contains to a Surface. The Group.draw() method requires that each Sprite have a Surface.image attribute and a Surface.rect. The Group.clear() method requires these same attributes, and can be used to erase all the Sprites with background. There are also more advanced Groups: pygame.sprite.RenderUpdates() and pygame.sprite.OrderedUpdates().

Lastly, this module contains several collision functions. These help find sprites inside multiple groups that have intersecting bounding rectangles. To find the collisions, the Sprites are required to have a Surface.rect attribute assigned.

The groups are designed for high efficiency in removing and adding Sprites to them. They also allow cheap testing to see if a Sprite already exists in a Group. A given Sprite can exist in any number of groups. A game could use some groups to control object rendering, and a completely separate set of groups to control interaction or player movement. Instead of adding type attributes or bools to a derived Sprite class, consider keeping the Sprites inside organized Groups. This will allow for easier lookup later in the game.

Sprites and Groups manage their relationships with the add() and remove() methods. These methods can accept a single or multiple targets for membership. The default initializers for these classes also takes a single or list of targets for initial membership. It is safe to repeatedly add and remove the same Sprite from a Group.

While it is possible to design sprite and group classes that don’t derive from the Sprite and AbstractGroup classes below, it is strongly recommended that you extend those when you add a Sprite or Group class.

Sprites are not thread safe. So lock them yourself if using threads.

pygame.sprite.Sprite

Simple base class for visible game objects.

*Sprite(*groups) -> Sprite*

- pygame.sprite.Sprite.update — method to control sprite behavior
- pygame.sprite.Sprite.add — add the sprite to groups
- pygame.sprite.Sprite.remove — remove the sprite from groups
- pygame.sprite.Sprite.kill — remove the Sprite from all Groups
- pygame.sprite.Sprite.alive — does the sprite belong to any groups
- pygame.sprite.Sprite.groups — list of Groups that contain this Sprite

The base class for visible game objects. Derived classes will want to override the Sprite.update() and assign a Sprite.image and Sprite.rect attributes. The initializer can accept any number of Group instances to be added to.

When subclassing the Sprite, be sure to call the base initializer before adding the Sprite to Groups. For example:

```
class Block(pygame.sprite.Sprite):

    # Constructor. Pass in the color of the block,
    # and its x and y position
    def __init__(self, color, width, height):
        # Call the parent class (Sprite) constructor
        pygame.sprite.Sprite.__init__(self)

        # Create an image of the block, and fill it with a color.
        # This could also be an image loaded from the disk.
        self.image = pygame.Surface([width, height])
        self.image.fill(color)

        # Fetch the rectangle object that has the dimensions of the image
        # Update the position of this object by setting the values of rect.x and rect.y
        self.rect = self.image.get_rect()
```

update()

method to control sprite behavior

*update(*args) -> None*

The default implementation of this method does nothing; it's just a convenient "hook" that you can override. This method is called by Group.update() with whatever arguments you give it.

There is no need to use this method if not using the convenience method by the same name in the Group class.

add()

add the sprite to groups

*add(*groups) -> None*

Any number of Group instances can be passed as arguments. The Sprite will be added to the Groups it is not already a member of.

remove()

remove the sprite from groups

*remove(*groups) -> None*

Any number of Group instances can be passed as arguments. The Sprite will be removed from the Groups it is currently a member of.

kill()

remove the Sprite from all Groups

kill() -> None

The Sprite is removed from all the Groups that contain it. This won't change anything about the state of the Sprite. It is possible to continue to use the Sprite after this method has been called, including adding it to Groups.

alive()

does the sprite belong to any groups

alive() -> bool

Returns True when the Sprite belongs to one or more Groups.

groups()

list of Groups that contain this Sprite

groups() -> group_list

Return a list of all the Groups that contain this Sprite.

pygame.sprite.DirtySprite

A subclass of Sprite with more attributes and features.

*DirtySprite(*groups) -> DirtySprite*

Extra DirtySprite attributes with their default values:

dirty = 1

if set to 1, it is repainted and then set to 0 again

if set to 2 then it is always dirty (repainted each frame,

flag is not reset)

0 means that it is not dirty and therefore not repainted again

blendmode = 0

its the special_flags argument of blit, blendmodes

source_rect = None

source rect to use, remember that it is relative to

topleft (0,0) of self.image

visible = 1

normally 1, if set to 0 it will not be repainted

(you must set it dirty too to be erased from screen)

layer = 0

(READONLY value, it is read when adding it to the

LayeredDirty, for details see doc of LayeredDirty)

pygame.sprite.Group

A container class to hold and manage multiple Sprite objects.

*Group(*sprites) -> Group*

- pygame.sprite.Group.sprites — list of the Sprites this Group contains
- pygame.sprite.Group.copy — duplicate the Group
- pygame.sprite.Group.add — add Sprites to this Group
- pygame.sprite.Group.remove — remove Sprites from the Group
- pygame.sprite.Group.has — test if a Group contains Sprites
- pygame.sprite.Group.update — call the update method on contained Sprites
- pygame.sprite.Group.draw — blit the Sprite images
- pygame.sprite.Group.clear — draw a background over the Sprites
- pygame.sprite.Group.empty — remove all Sprites

A simple container for Sprite objects. This class can be inherited to create containers with more specific behaviors. The constructor takes any number of Sprite arguments to add to the Group. The group supports the following standard Python operations:

in	test if a Sprite is contained
len	the number of Sprites contained
bool	test if any Sprites are contained
iter	iterate through all the Sprites

The Sprites in the Group are not ordered, so drawing and iterating the Sprites is in no particular order.

sprites()

list of the Sprites this Group contains

sprites() -> sprite_list

Return a list of all the Sprites this group contains. You can also get an iterator from the group, but you cannot iterator over a Group while modifying it.

copy()

duplicate the Group

copy() -> Group

Creates a new Group with all the same Sprites as the original. If you have subclassed Group, the new object will have the same (sub-)class as the original. This only works if the derived class's constructor takes the same arguments as the Group class's.

add()

add Sprites to this Group

*add(*sprites) -> None*

Add any number of Sprites to this Group. This will only add Sprites that are not already members of the Group.

Each sprite argument can also be a iterator containing Sprites.

remove()

remove Sprites from the Group

*remove(*sprites) -> None*

Remove any number of Sprites from the Group. This will only remove Sprites that are already members of the Group.

Each sprite argument can also be a iterator containing Sprites.

has()

test if a Group contains Sprites

*has(*sprites) -> None*

Return True if the Group contains all of the given sprites. This is similar to using the “in” operator on the Group (“if sprite in group: ...”), which tests if a single Sprite belongs to a Group.

Each sprite argument can also be a iterator containing Sprites.

update()

call the update method on contained Sprites

*update(*args) -> None*

Calls the update() method on all Sprites in the Group. The base Sprite class has an update method that takes any number of arguments and does nothing. The arguments passed to Group.update() will be passed to each Sprite.

There is no way to get the return value from the Sprite.update() methods.

draw()

blit the Sprite images

draw(Surface) -> None

Draws the contained Sprites to the Surface argument. This uses the Sprite.image attribute for the source surface, and Sprite.rect for the position.

The Group does not keep sprites in any order, so the draw order is arbitrary.

clear()

draw a background over the Sprites

clear(Surface_dest, background) -> None

Erases the Sprites used in the last Group.draw() call. The destination Surface is cleared by filling the drawn Sprite positions with the background.

The background is usually a Surface image the same dimensions as the destination Surface. However, it can also be a callback function that takes two arguments; the destination Surface and an area to clear. The background callback function will be called several times each clear.

Here is an example callback that will clear the Sprites with solid red:

```
def clear_callback(surf, rect):
    color = 255, 0, 0
    surf.fill(color, rect)
```

empty()

remove all Sprites

empty() -> None

Removes all Sprites from this Group.

pygame.sprite.RenderPlain

Same as pygame.sprite.Group
This class is an alias to pygame.sprite.Group(). It has no additional functionality.

pygame.sprite.RenderClear
Same as pygame.sprite.Group
This class is an alias to pygame.sprite.Group(). It has no additional functionality.

pygame.sprite.RenderUpdates
Group sub-class that tracks dirty updates.
*RenderUpdates(*sprites) -> RenderUpdates*
pygame.sprite.RenderUpdates.draw — blit the Sprite images and track changed areas
This class is derived from pygame.sprite.Group(). It has an extended draw() method that tracks the changed areas of the screen.

draw()
blit the Sprite images and track changed areas
draw(surface) -> Rect_list
Draws all the Sprites to the surface, the same as Group.draw(). This method also returns a list of Rectangular areas on the screen that have been changed. The returned changes include areas of the screen that have been affected by previous Group.clear() calls.

The returned Rect list should be passed to pygame.display.update(). This will help performance on software driven display modes. This type of updating is usually only helpful on destinations with non-animating backgrounds.

pygame.sprite.OrderedUpdates()
RenderUpdates sub-class that draws Sprites in order of addition.
*OrderedUpdates(*spites) -> OrderedUpdates*
This class derives from pygame.sprite.RenderUpdates(). It maintains the order in which the Sprites were added to the Group for rendering. This makes adding and removing Sprites from the Group a little slower than regular Groups.

pygame.sprite.LayeredUpdates
LayeredUpdates is a sprite group that handles layers and draws like OrderedUpdates.
*LayeredUpdates(*spites, **kwargs) -> LayeredUpdates*

- pygame.sprite.LayeredUpdates.add — add a sprite or sequence of sprites to a group
- pygame.sprite.LayeredUpdates.sprites — returns a ordered list of sprites (first back, last top).
- pygame.sprite.LayeredUpdates.draw — draw all sprites in the right order onto the passed surface.
- pygame.sprite.LayeredUpdates.get_sprites_at — returns a list with all sprites at that position.
- pygame.sprite.LayeredUpdates.get_sprite — returns the sprite at the index idx from the groups sprites
- pygame.sprite.LayeredUpdates.remove_sprites_of_layer — removes all sprites from a layer and returns them as a list.
- pygame.sprite.LayeredUpdates.layers — returns a list of layers defined (unique), sorted from bottom up.
- pygame.sprite.LayeredUpdates.change_layer — changes the layer of the sprite
- pygame.sprite.LayeredUpdates.get_layer_of_sprite — returns the layer that sprite is currently in.
- pygame.sprite.LayeredUpdates.get_top_layer — returns the top layer
- pygame.sprite.LayeredUpdates.get_bottom_layer — returns the bottom layer
- pygame.sprite.LayeredUpdates.move_to_front — brings the sprite to front layer
- pygame.sprite.LayeredUpdates.move_to_back — moves the sprite to the bottom layer
- pygame.sprite.LayeredUpdates.get_top_sprite — returns the topmost sprite
- pygame.sprite.LayeredUpdates.get_sprites_from_layer — returns all sprites from a layer, ordered by how they where added
- pygame.sprite.LayeredUpdates.switch_layer — switches the sprites from layer1 to layer2

This group is fully compatible with pygame.sprite.SpriteSimple base class for visible game objects..

You can set the default layer through kwargs using ‘default_layer’ and an integer for the layer. The default layer is 0.

If the sprite you add has an attribute layer then that layer will be used. If the **kwarg contains ‘layer’ then the sprites passed will be added to that layer (overriding the sprite.layer attribute). If neither sprite has attribute layer nor **kwarg then the default layer is used to add the sprites.

add()
add a sprite or sequence of sprites to a group
*add(*sprites, **kwargs) -> None*
If the sprite(s) have an attribute layer then that is used for the layer. If **kwargs contains ‘layer’ then the sprite(s) will be added to that argument (overriding the sprite layer attribute). If neither is passed then the sprite(s) will be added to the default layer.

sprites()
returns a ordered list of sprites (first back, last top).
sprites() -> sprites

draw()
draw all sprites in the right order onto the passed surface.
draw(surface) -> Rect_list

get_sprites_at()
returns a list with all sprites at that position.
get_sprites_at(pos) -> colliding_sprites
Bottom sprites first, top last.

get_sprite()
returns the sprite at the index idx from the groups sprites
get_sprite(idx) -> sprite
Raises IndexOutOfBounds if the idx is not within range.

remove_sprites_of_layer()
removes all sprites from a layer and returns them as a list.
remove_sprites_of_layer(layer_nr) -> sprites

layers()
returns a list of layers defined (unique), sorted from bottom up.
layers() -> layers

change_layer()
changes the layer of the sprite
change_layer(sprite, new_layer) -> None
sprite must have been added to the renderer. It is not checked.

get_layer_of_sprite()
returns the layer that sprite is currently in.
get_layer_of_sprite(sprite) -> layer
If the sprite is not found then it will return the default layer.

get_top_layer()
returns the top layer
get_top_layer() -> layer

get_bottom_layer()
returns the bottom layer
get_bottom_layer() -> layer

move_to_front()
brings the sprite to front layer
move_to_front(sprite) -> None
Brings the sprite to front, changing sprite layer to topmost layer (added at the end of that layer).

move_to_back()
moves the sprite to the bottom layer
move_to_back(sprite) -> None
Moves the sprite to the bottom layer, moving it behind all other layers and adding one additional layer.

get_top_sprite()
returns the topmost sprite
get_top_sprite() -> Sprite

get_sprites_from_layer()
returns all sprites from a layer, ordered by how they where added
get_sprites_from_layer(layer) -> sprites
Returns all sprites from a layer, ordered by how they where added. It uses linear search and the sprites are not removed from layer.

switch_layer()
switches the sprites from layer1 to layer2
switch_layer(layer1_nr, layer2_nr) -> None
The layers number must exist, it is not checked.

pygame.sprite.LayeredDirty
LayeredDirty group is for DirtySprite objects. Subclasses LayeredUpdates.
*LayeredDirty(*spites, **kwargs) -> LayeredDirty*

- pygame.sprite.LayeredDirty.draw — draw all sprites in the right order onto the passed surface.
- pygame.sprite.LayeredDirty.clear — used to set background
- pygame.sprite.LayeredDirty.repaint_rect — repaints the given area
- pygame.sprite.LayeredDirty.set_clip — clip the area where to draw. Just pass None (default) to reset the clip
- pygame.sprite.LayeredDirty.get_clip — clip the area where to draw. Just pass None (default) to reset the clip
- pygame.sprite.LayeredDirty.change_layer — changes the layer of the sprite
- pygame.sprite.LayeredDirty.set_timing_treshold — sets the threshold in milliseconds

This group requires pygame.sprite.DirtySpriteA subclass of Sprite with more attributes and features. or any sprite that has the following attributes:
image, rect, dirty, visible, blendmode (see doc of DirtySprite).
It uses the dirty flag technique and is therefore faster than the pygame.sprite.RenderUpdatesGroup sub-class that tracks dirty updates. if you have many static sprites. It also switches automatically between dirty rect update and full screen drawing, so

you do no have to worry what would be faster.
Same as for the pygame.sprite.GroupA container class to hold and manage multiple Sprite objects.. You can specify some additional attributes through kwargs:

_use_update: True/False default is False

_default_layer: default layer where sprites without a layer are added.

_time_threshold: threshold time for switching between dirty rect mode

and fullscreen mode, defaults to 1000./80 == 1000./fps

draw()

draw all sprites in the right order onto the passed surface.
draw(surface, bgd=None) -> Rect_list
You can pass the background too. If a background is already set, then the bgd argument has no effect.

clear()

used to set background
clear(surface, bgd) -> None

repaint_rect()

repaints the given area
repaint_rect(screen_rect) -> None
screen_rect is in screen coordinates.

set_clip()

clip the area where to draw. Just pass None (default) to reset the clip
set_clip(screen_rect=None) -> None

get_clip()

clip the area where to draw. Just pass None (default) to reset the clip
get_clip() -> Rect

change_layer()

changes the layer of the sprite
change_layer(sprite, new_layer) -> None
sprite must have been added to the renderer. It is not checked.

set_timing_treshold()

sets the threshold in milliseconds
set_timing_treshold(time_ms) -> None
Default is 1000./80 where 80 is the fps I want to switch to full screen mode. This method’s name is a typo and should be fixed.

pygame.sprite.GroupSingle()

Group container that holds a single sprite.
GroupSingle(sprite=None) -> GroupSingle
The GroupSingle container only holds a single Sprite. When a new Sprite is added, the old one is removed.

There is a special property, GroupSingle.sprite, that accesses the Sprite that this Group contains. It can be None when the Group is empty. The property can also be assigned to add a Sprite into the GroupSingle container.

pygame.sprite.spritecollide()

Find sprites in a group that intersect another sprite.
spritecollide(sprite, group, dokill, collided = None) -> Sprite_list
Return a list containing all Sprites in a Group that intersect with another Sprite. Intersection is determined by comparing the Sprite.rect attribute of each Sprite.

The dokill argument is a bool. If set to True, all Sprites that collide will be removed from the Group.

The collided argument is a callback function used to calculate if two sprites are colliding. it should take two sprites as values, and return a bool value indicating if they are colliding. If collided is not passed, all sprites must have a "rect" value, which is a rectangle of the sprite area, which will be used to calculate the collision.

collided callables:
collide_rect, collide_rect_ratio, collide_circle,
collide_circle_ratio, collide_mask

Example:

```
# See if the Sprite block has collided with anything in the Group block_list
# The True flag will remove the sprite in block_list
blocks_hit_list = pygame.sprite.spritecollide(player, block_list, True)

# Check the list of colliding sprites, and add one to the score for each one
for block in blocks_hit_list:
    score +=1
```

pygame.sprite.collide_rect()
Collision detection between two sprites, using rects.
collide_rect(left, right) -> bool
Tests for collision between two sprites. Uses the pygame rect colliderect function to calculate the collision. Intended to be passed as a collided callback function to the *collide functions. Sprites must have a "rect" attributes.

pygame.sprite.collide_rect_ratio()
Collision detection between two sprites, using rects scaled to a ratio.
collide_rect_ratio(ratio) -> collided_callable
A callable class that checks for collisions between two sprites, using a scaled version of the sprites rects.

Is created with a ratio, the instance is then intended to be passed as a collided callback function to the *collide functions.

A ratio is a floating point number - 1.0 is the same size, 2.0 is twice as big, and 0.5 is half the size.

pygame.sprite.collide_circle()
Collision detection between two sprites, using circles.
collide_circle(left, right) -> bool
Tests for collision between two sprites, by testing to see if two circles centered on the sprites overlap. If the sprites have a "radius" attribute, that is used to create the circle, otherwise a circle is created that is big enough to completely enclose the sprites rect as given by the "rect" attribute. Intended to be passed as a collided callback function to the *collide functions. Sprites must have a "rect" and an optional "radius" attribute.

pygame.sprite.collide_circle_ratio()
Collision detection between two sprites, using circles scaled to a ratio.

collide_circle_ratio(ratio) -> collided_callable
A callable class that checks for collisions between two sprites, using a scaled version of the sprites radius.

Is created with a floating point ratio, the instance is then intended to be passed as a collided callback function to the *collide functions.

A ratio is a floating point number - 1.0 is the same size, 2.0 is twice as big, and 0.5 is half the size.

The created callable tests for collision between two sprites, by testing to see if two circles centered on the sprites overlap, after scaling the circles radius by the stored ratio. If the sprites have a “radius” attribute, that is used to create the circle, otherwise a circle is created that is big enough to completely enclose the sprites rect as given by the “rect” attribute. Intended to be passed as a collided callback function to the *collide functions. Sprites must have a “rect” and an optional “radius” attribute.

pygame.sprite.collide_mask()
Collision detection between two sprites, using masks.
collide_mask(SpriteLeft, SpriteRight) -> point
Returns first point on the mask where the masks collided, or None if there was no collision.

Tests for collision between two sprites, by testing if their bitmasks overlap. If the sprites have a “mask” attribute, that is used as the mask, otherwise a mask is created from the sprite image. Intended to be passed as a collided callback function to the *collide functions. Sprites must have a “rect” and an optional “mask” attribute.

You should consider creating a mask for your sprite at load time if you are going to check collisions many times. This will increase the performance, otherwise this can be an expensive function because it will create the masks each time you check for collisions.

sprite.mask = pygame.mask.from_surface(sprite.image)

pygame.sprite.groupcollide()
Find all sprites that collide between two groups.
groupcollide(group1, group2, dokill1, dokill2, collided = None) -> Sprite_dict
This will find collisions between all the Sprites in two groups. Collision is determined by comparing the Sprite.rect attribute of each Sprite or by using the collided function if it is not None.

Every Sprite inside group1 is added to the return dictionary. The value for each item is the list of Sprites in group2 that intersect.

If either dokill argument is True, the colliding Sprites will be removed from their respective Group.

The collided argument is a callback function used to calculate if two sprites are colliding. It should take two sprites as values and return a bool value indicating if they are colliding. If collided is not passed, then all sprites must have a “rect” value, which is a rectangle of the sprite area, which will be used to calculate the collision.

pygame.sprite.spritecollideany()
Simple test if a sprite intersects anything in a group.
spritecollideany(sprite, group, collided = None) -> Sprite Collision with the returned sprite.
spritecollideany(sprite, group, collided = None) -> None No collision
If the sprite collides with any single sprite in the group, a single sprite from the group is returned. On no collision None is

returned.

If you don’t need all the features of the pygame.sprite.spritecollide() function, this function will be a bit quicker.

The collided argument is a callback function used to calculate if two sprites are colliding. It should take two sprites as values and return a bool value indicating if they are colliding. If collided is not passed, then all sprites must have a “rect” value, which is a rectangle of the sprite area, which will be used to calculate the collision.