Pygame 1.5.5 Reference Manual

Ver. 0.00

By Gabriel Alejandro Zorrilla

gaz082@yahoo.com.ar

Based entirely upon the Pygame Documentation

www.pygame.org

Some words from the “author”

Before you start to read this reference document, I would like to answer some questions that perhaps came to your mind when you read the title of this book:

I have the Pygame Documents, why should I use this guide?

I found the Pygame Documents fine for a quick reference when you are used to the pygame modules and stuff. Im a newbie pygame programmer and because of that I need constant assistance from the Pygame Documents. But, it’s not funny to be opening and closing windows every time I want to read it, so I decided to make a printer friendly version of those documents that suit my tastes.

What are “your tastes”?

Through this document you will notice some reader friendly features:

- Real useful index: as reader of How to Think Like a Computer Scientist: Learning with Python I´m sick of doing math everytime I have to type the page of the topic I want to go, because the 25 pages of introduction and stuff. With my book you just look what topic would you like to read about and the index tells you *exactly* what page it is, whether you are reading this file in Acrobat or printed. This may be not a mayor breakthrough in book desing development, but I´m sure that you would thank this to me!.

- No “cut” paragraphs: aren´t you tired of reading one part of the paragraph in one page and the other part in other one?, and then, when you type some code and want to pick up again that paragraph, you have to look again and figure out in what page the important suff was?. That system may save you paper, but, heck, this is a reference document, the purpose is to save time, not paper.

- PDF: pdf documents rule. You can read it in almost any machine and besides, pdf documents are “in”.

Who are you?

My name is Gabriel Alejandro Zorrilla, and from now it mustn't be named again. I´m a 21 years old fella from Buenos Aires, Argentina. I´m currently studying Industrial Engineering at the UTN. I have many hobbies, and one of them is programming.

That´s it!. Enjoy this manual!.

Main Index

pygame

Contains the core routines that are used by the rest of the pygame modules. It's routines are merged directly into the pygame namespace. This mainly includes the auto-initialization init() and quit() routines.

There is a small module named 'locals' that also gets merged into this namespace. This contains all the constants needed by pygame. Object constructors also get placed into this namespace, you can call functions like Rect() and Surface() to create objects of that type. As a convenience, you can import the members of pygame.locals directly into your module's namespace with 'from pygame.locals import *'. Most of the pygame examples do this if you'd like to take a look.

Rect	- create a new rectangle
Surface	- create a new Surface
get_error	- get current error message
init	- autoinitialize all imported pygame modules
quit	- uninitialize all pygame modules
register_quit	- routine to call when pygame quits

Rect

```
pygame.Rect(rectstyle) -> Rect
```

Creates a new rectangle object. The given rectstyle represents one of the various ways of representing rectangle data. This is usually a sequence of x and y position for the topleft corner, and the width and height.

For some of the Rect methods there are two version. For example, there is move() and move_ip(). The methods witht the '_ip' suffix on the name are the 'in-place' version of those functions. They effect the actual source object, instead of returning a new Rect object.

Surface

```
pygame.Surface(size, [flags, [Surface|depth, [masks]]]) -> Surface
```

Creates a new surface object. Size is a 2-int-sequence containing width and height. Depth is the number of bits used per pixel. If omitted, depth will use the current display depth. Masks is a four item sequence containing the bitmask for r,g,b, and a. If omitted, masks will default to the usual values for the given bitdepth. Flags is a mix of the following flags: SWSURFACE, HWSURFACE, ASYNCBLIT, or SRCALPHA. (flags = 0 is the same as SWSURFACE). Depth and masks can be substituted for another surface object which will create the new surface with the same format as the given one. When using default masks, alpha will always be ignored unless you pass SRCALPHA as a flag. For a plain software surface, 0 can be used for the flag. A plain hardware surface can just use 1 for the flag.

get_error

```
pygame.get_error() -> errorstring
```

SDL maintains an internal current error message. This message is usually given to you when an SDL related exception occurs, but sometimes you may want to call this directly yourself.

init

```
pygame.init() -> passed, failed
```

Initialize all imported pygame modules. Including pygame modules that are not part of the base modules (like font and image).

It does not raise exceptions, but instead silently counts which modules have failed to init. The return argument contains a count of the number of modules initialized, and the number of modules that failed to initialize.

You can always initialize the modules you want by hand. The modules that need it have an init() and quit() routine built in, which you can call directly. They also have a get_init() routine which you can use to doublecheck the initialization. Note that the manual init() routines will raise an exception on error. Be aware that most platforms require the display module to be initialized before others. This init() will handle that for you, but if you initialize by hand, be aware of this constraint.

As with the manual init() routines. It is safe to call this init() as often as you like. If you have imported pygame modules since the.

quit

```
pygame.quit() -> none
```

Uninitialize all pygame modules that have been initialized. Even if you initialized the module by hand, this quit() will uninitialize it for you.

All the pygame modules are uninitialized automatically when your program exits, so you will usually not need this routine. If you program plans to keep running after it is done with pygame, then would be a good time to make this call.

register_quit

```
pygame.register_quit(callback) -> None
```

The given callback routine will be called when. pygame is quitting. Quit callbacks are served on a 'last in, first out' basis. Also be aware that your callback may be called more than once.

pygame.cdrom

The cdrom module provides a few functions to initialize the CDROM subsystem and to manage the CD objects. The CD objects are created with the pygame.cdrom.CD() function. This function needs a cdrom device number to work on. All cdrom drives on the system are enumerated for use as a CD object. To access most of the CD functions, you'll need to init() the CD. (note that the cdrom module will already be initialized). When multiple CD objects are created for the same CDROM device, the state and values for those CD objects will be shared.

You can call the CD.get_name() and CD.get_id() functions without initializing the CD object.

Be sure to understand there is a difference between the cdrom module and the CD objects.

CD	- create new CD object
get_count	- query number of cdroms on system
get_init	- query init of cdrom module
init	- initialize the cdrom subsystem
quit	- uninitialize the cdrom subsystem

CD

```
pygame.cdrom.CD(id) -> CD
```

Creates a new CD object for the given CDROM id. The given id must be less than the value from pygame.cdrom.get_count().

get_count

```
pygame.cdrom.get_count() -> int
```

Returns the number of CDROM drives available on the system.

get_init

```
pygame.cdrom.get_init() -> bool
```

Returns true of the cdrom module is initialized

init

```
pygame.cdrom.init() -> None
```

Initialize the CDROM module manually

quit

```
pygame.cdrom.quit() -> None
```

Uninitialize the CDROM module manually

pygame.constants

These constants are defined by SDL, and needed in pygame. Note that many of the flags for SDL are not needed in pygame, and are not included here. These constants are generally accessed from the pygame.locals module. This module is automatically placed in the pygame namespace, but you will usually want to place them directly into your module's namespace with the following command, 'from pygame.locals import *'.

display

`pygame.constants.display (constants)`

HWSURFACE - surface in hardware video memory. (equal to 1)
RESIZABLE - display window is resizeable
ASYNCBLIT - surface blits happen asynchronously (threaded)
OPENGL - display surface will be controlled by opengl
HWPALETTE - display surface has animatable hardware palette entries
DOUBLEBUF - hardware display surface is page flippable
FULLSCREEN - display surface is fullscreen (nonwindowed)
RLEACCEL - compile for quick alpha blits, only set in alpha or colorkey funcs
NOFRAME - no window decorations

events

`pygame.constants.events (constants)`

NOEVENT - no event, represents an empty event list, equal to 0
ACTIVEEVENT - window has gain/lost mouse/keyboard/visiblity focus
KEYDOWN - keyboard button has been pressed (or down and repeating)
KEYUP - keyboard button has been released
MOUSEMOTION - mouse has moved
MOUSEBUTTONDOWN- mouse button has been pressed
MOUSEBUTTONUP - mouse button has been released
JOYAXISMOTION - an opened joystick axis has changed
JOYBALLMOTION - an opened joystick ball has moved
JOYHATMOTION - an opened joystick hat has moved
JOYBUTTONDOWN - an opened joystick button has been pressed
JOYBUTTONUP - an opened joystick button has been released
VIDEORESIZE - the display window has been resized by the user
QUIT - the user has requested the game to quit
SYSWMEVENT - currently unsupported, system dependant
USEREVENT - all user messages are this or higher
NUMEVENTS - all user messages must be lower than this, equal to 32

keyboard

pygame.constants.keyboard (constants)

There are many keyboard constants, they are used to represent keys on the keyboard. The following is a list of all keyboard constants

KeyASCII	ASCII	Common Name
K_BACKSPACE	\b	*backspace*
K_TAB	\t	*tab*
K_CLEAR		*clear*
K_RETURN	\r	*return*
K_PAUSE		*pause*
K_ESCAPE	^[	*escape*
K_SPACE		*space*
K_EXCLAIM	!	*exclaim*
K_QUOTEDBL	\"	*quotedbl*
K_HASH	#	*hash*
K_DOLLAR	$	*dollar*
K_AMPERSAND	&	*ampersand*
K_QUOTE		*quote*
K_LEFTPAREN	(	*left parenthesis*
K_RIGHTPAREN	)	*right parenthesis*
K_ASTERISK	*	*asterisk*
K_PLUS	+	*plus sign*
K_COMMA	,	*comma*
K_MINUS	-	*minus sign*
K_PERIOD	.	*period*
K_SLASH	/	*forward slash*
K_0	0	*0*
K_1	1	*1*
K_2	2	*2*
K_3	3	*3*
K_4	4	*4*

K_5	5	*5*
K_6	6	*6*
K_7	7	*7*
K_8	8	*8*
K_9	9	*9*
K_COLON	:	*colon*
K_SEMICOLON	;	*semicolon*
K_LESS	<	*less-than sign*
K_EQUALS	=	*equals sign*
K_GREATER	>	*greater-than sign*
K_QUESTION	?	*question mark*
K_AT	@	*at*
K_LEFTBRACKET	[	*left bracket*
K_BACKSLASH	\\	*backslash*
K_RIGHTBRACKET	]	*right bracket*
K_CARET	^	*caret*
K_UNDERSCORE	_	*underscore*
K_BACKQUOTE	`	*grave*
K_a	a	*a*
K_b	b	*b*
K_c	c	*c*
K_d	d	*d*
K_e	e	*e*
K_f	f	*f*
K_g	g	*g*
K_h	h	*h*
K_i	i	*i*
K_j	j	*j*
K_k	k	*k*
K_l	l	*l*

K_m	m	*m*
K_n	n	*n*
K_o	o	*o*
K_p	p	*p*
K_q	q	*q*
K_r	r	*r*
K_s	s	*s*
K_t	t	*t*
K_u	u	*u*
K_v	v	*v*
K_w	w	*w*
K_x	x	*x*
K_y	y	*y*
K_z	z	*z*
K_DELETE		*delete*
K_KP0		*keypad 0*
K_KP1		*keypad 1*
K_KP2		*keypad 2*
K_KP3		*keypad 3*
K_KP4		*keypad 4*
K_KP5		*keypad 5*
K_KP6		*keypad 6*
K_KP7		*keypad 7*
K_KP8		*keypad 8*
K_KP9		*keypad 9*
K_KP_PERIOD	.	*keypad period*
K_KP_DIVIDE	/	*keypad divide*
K_KP_MULTIPLY	*	*keypad multiply*
K_KP_MINUS	-	*keypad minus*
K_KP_PLUS	+	*keypad plus*

K_KP_ENTER	\r	*keypad enter*
K_KP_EQUALS	=	*keypad equals*
K_UP		*up arrow*
K_DOWN		*down arrow*
K_RIGHT		*right arrow*
K_LEFT		*left arrow*
K_INSERT		*insert*
K_HOME		*home*
K_END		*end*
K_PAGEUP		*page up*
K_PAGEDOWN		*page down*
K_F1		*F1*
K_F2		*F2*
K_F3		*F3*
K_F4		*F4*
K_F5		*F5*
K_F6		*F6*
K_F7		*F7*
K_F8		*F8*
K_F9		*F9*
K_F10		*F10*
K_F11		*F11*
K_F12		*F12*
K_F13		*F13*
K_F14		*F14*
K_F15		*F15*
K_NUMLOCK		*numlock*
K_CAPSLOCK		*capslock*
K_SCROLLOCK		*scrollock*
K_RSHIFT		*right shift*

K_LSHIFT	*left shift*
K_RCTRL	*right ctrl*
K_LCTRL	*left ctrl*
K_RALT	*right alt*
K_LALT	*left alt*
K_RMETA	*right meta*
K_LMETA	*left meta*
K_LSUPER	*left windows key*
K_RSUPER	*right windows key*
K_MODE	*mode shift*
K_HELP	*help*
K_PRINT	*print-screen*
K_SYSREQ	*SysRq*
K_BREAK	*break*
K_MENU	*menu*
K_POWER	*power*
K_EURO	*euro*

modifiers

`pygame.constants.modifiers (constants)`

Their states are treated slightly differently than normal keyboard button states, and you can temporarily set their states.

KMOD_NONE, KMOD_LSHIFT, KMOD_RSHIFT, KMOD_SHIFT, KMOD_CAPS, KMOD_LCTRL, KMOD_RCTRL, KMOD_CTRL, KMOD_LALT, KMOD_RALT, KMOD_ALT, KMOD_LMETA, KMOD_RMETA, KMOD_META, KMOD_NUM, KMOD_MODE

zdepracated

pygame.constants.zdepracated (constants)

The flags labeled as readonly should never be used, except when comparing checking flags against Surface.get_flags().

SWSURFACE - not really usable as a surface flag, equates to 0 and is always default
ANYFORMAT - creates a display with in best possible bit depth
HWACCEL - surface is hardware accelerated, readonly
SRCCOLORKEY- surface has a colorkey for blits, readonly
SRCALPHA - surface has alpha enabled, readonly
RLEACCELOK - surface is rle accelrated but uncompiled, readonly

pygame.display

Contains routines to work with the display. Mainly used for setting the display mode and updating the display surface.

Pygame offers a fairly simple interface to the display buffer. The buffer is represented as an offscreen surface to which you can write directly. If you want the screen to show what you have written, the pygame.display.update() function will guarantee the the desired portion of the screen is updated. You can call pygame.display.flip() to update the entire screen, and also flip a hardware surface created with DOUBLEBUF.

There are a number of ways to start the video display. The easiest way is to pick a common screen resolution and depth and just initialize the video, checking for exceptions. You will probably get what you want, but pygame may be emulating your requested mode and converting the display on update (this is not the fastest method). When calling pygame.display.set_mode() with the bit depth omitted or set to zero, pygame will determine the best video mode available and set to that. You can also query for more information on video modes with pygame.display.mode_ok(), pygame.display.list_modes(), and pygame.display.Info().

When using a display depth other than what you graphic resources may be saved at, it is best to call the Surface.convert() routine to convert them to the same format as the display, this will result in the fastest blitting.

Pygame currently supports any but depth >= 8 bits per pixl. 8bpp formats are considered to be 8-bit palettized modes, while 12, 15, 16, 24, and 32 bits per pixel are considered 'packed pixel' modes, meaning each pixel contains the RGB color componsents packed into the bits of the pixel.

After you have initialized your video mode, you can take the surface that was returned and write to it like any other Surface object. Be sure to call update() or flip() to keep what is on the screen synchronized with what is on the surface. Be sure not to call display routines that modify the display surface while it is locked.

Info	- get display capabilities and settings
flip	- update the display
get_active	- get state of display mode
get_caption	- get the current title of the window
get_driver	- get the current sdl video driver
get_init	- get status of display module initialization
get_surface	- get current display surface
gl_get_attribute	- get special OPENGL attributes
gl_set_attribute	- set special OPENGL attributes
iconify	- minimize the display window
init	- initialize the display module

list_modes	- query all resolutions for requested mode
mode_ok	- query a specific display mode
quit	- uninitialize the display module
set_caption	- changes the title of the window
set_gamma	- change the brightness of the display
set_gamma_ramp	- advanced control over the display gamma ramps
set_icon	- changes the window manager icon for the window
set_mode	- set the display mode
set_palette	- set the palette
toggle_fullscreen	- switch the display fullscreen mode
update	- update an area of the display

Info

```
pygame.display.Info() -> VidInfo
```

Gets a vidinfo object that contains information about the capabilities and current state of the video driver. This can be called before the display mode is set, to determine the current video mode of a display. You can print the VidInfo object to see all its members and values.

flip

```
pygame.display.flip() -> None
```

This will update the contents of the entire display. If your display mode is using the flags HWSURFACE and DOUBLEBUF, this will wait for a vertical retrace and swap the surfaces. If you are using a different type of display mode, it will simply update the entire contents of the surface.

When using an OPENGL display mode this will perform a gl buffer swap.

get_active

```
pygame.display.get_active() -> bool
```

Returns true if the current display is active on the screen. This done with the call to pygame.display.set_mode(). It is potentially subject to the activity of a running window manager.

Calling set_mode() will change all existing display surface to reference the new display mode. The old display surface will be lost after this call.

get_caption

```
pygame.display.get_caption() -> title, icontitle
```

Returns the current title and icontitle for the display window.

get_driver

```
pygame.display.get_driver() -> name
```

Once the display is initialized, this will return the name of the currently running video driver. There is no way to get a list of all the supported video drivers.

get_init

```
pygame.display.get_init() -> bool
```

Returns true if SDL's video system is currently intialized.

get_surface

```
pygame.display.get_surface() -> Surface
```

Returns a Surface object representing the current display. Will return None if called before the display mode is set.

gl_get_attribute

```
pygame.display.gl_get_attribute(flag) -> value
```

After calling pygame.display.set_mode() with the OPENGL flag you will likely want to check the value of any special opengl attributes you requested. You will not always get what you requested.

See pygame.display.gl_set_attribute() for a list of flags.

The OPENGL flags are; GL_ALPHA_SIZE, GL_DEPTH_SIZE, GL_STENCIL_SIZE, GL_ACCUM_RED_SIZE, GL_ACCUM_GREEN_SIZE, GL_ACCUM_BLUE_SIZE, GL_ACCUM_ALPHA_SIZE, GL_RED_SIZE, GL_GREEN_SIZE, GL_BLUE_SIZE, GL_DEPTH_SIZE

gl_set_attribute

```
pygame.display.gl_set_attribute(flag, value) -> None
```

When calling pygame.display.set_mode() with the OPENGL flag, pygame automatically handles setting the opengl attributes like color and doublebuffering. OPENGL offers several other attributes you may want control over. Pass one of these attributes as the flag, and its appropriate value.

This must be called before pygame.display.set_mode()

The OPENGL flags are; GL_ALPHA_SIZE, GL_DEPTH_SIZE, GL_STENCIL_SIZE, GL_ACCUM_RED_SIZE, GL_ACCUM_GREEN_SIZE, GL_ACCUM_BLUE_SIZE, GL_ACCUM_ALPHA_SIZE

iconify

```
pygame.display.iconify() -> bool
```

Tells the window manager (if available) to minimize the application. The call will return true if successful. You will receive an APPACTIVE event on the event queue when the window has been minimized.

init

```
pygame.display.init() -> None
```

Manually initialize SDL's video subsystem. Will raise an exception if it cannot be initialized. It is safe to call this function if the video has is currently initialized.

list_modes

```
pygame.display.list_modes([depth, [flags]]) -> [[x,y],...] | -1
```

This function returns a list of possible dimensions for a specified color depth. The return value will be an empty list if no display modes are available with the given arguments. A return value of -1 means that any requested resolution should work (this is likely the case for windowed modes). Mode sizes are sorted from biggest to smallest.

If depth is not passed or 0, SDL will choose the current/best color depth for the display. You will usually want to pass FULLSCREEN when using the flags, if flags is omitted, FULLSCREEN is the default.

mode_ok

```
pygame.display.mode_ok(size, [flags, [depth]]) -> int
```

This uses the same arguments as the call to pygame.display.set_mode(). It is used to determine if a requested display mode is available. It will return 0 if the requested mode is not possible. Otherwise it will return the best and closest matching bit depth for the mode requested.

The size is a 2-number-sequence containing the width and height of the desired display mode. Flags represents a set of different options for the display mode. If omitted or given as 0, it will default to a simple software window. You can mix several flags together with the bitwise-or (|) operator. Possible flags are HWSURFACE (or the value 1), HWPALETTE, DOUBLEBUF, and/or FULLSCREEN. There are other flags available but these are the most usual. A full list of flags can be found in the SDL documentation. The optional depth arguement is the requested bits per pixel. It will usually be left omitted, in which case the display will use the best/fastest pixel depth available.

quit

```
pygame.display.quit() -> None
```

Manually uninitialize SDL's video subsystem. It is safe to call this if the video is currently not initialized.

set_caption

```
pygame.display.set_caption(title, [icontitle]) -> None
```

If the display has a window title, this routine will change the name on the window. Some environments support a shorter icon title to be used when the display is minimized. If icontitle is omittied it will be the same as caption title.

set_gamma

```
pygame.display.set_gamma(r, [g, b]) -> bool
```

Sets the display gamma to the given amounts. If green and blue are ommitted, the red value will be used for all three colors. The color arguments are floating point values with 1.0 being the normal value. If you are using a display mode with a hardware palette, this will simply update the palette you are using. Not all hardware supports gamma. The return value will be true on success.

set_gamma_ramp

```
pygame.display.set_gamma_ramp(r, g, b) -> bool
```

Pass three sequences with 256 elements. Each element must be a '16bit' unsigned integer value. This is from 0 to 65536. If you are using a display mode with a hardware palette, this will simply update the palette you are using. Not all hardware supports gamma. The return value will be true on success.

set_icon

```
pygame.display.set_icon(Surface) -> None
```

Sets the runtime icon that your system uses to decorate the program window. It is also used when the application is iconified and in the window frame.

You likely want this to be a smaller image, a size that your system window manager will be able to deal with. It will also use the Surface colorkey if available.

Some window managers on X11 don't allow you to change the icon after the window has been shown the first time.

set_mode

```
pygame.display.set_mode(size, [flags, [depth]]) -> Surface
```

Sets the current display mode. If calling this after the mode has already been set, this will change the display mode to the desired type. Sometimes an exact match for the requested video mode is not available. In this case SDL will try to find the closest match and work with that instead.

The size is a 2-number-sequence containing the width and height of the desired display mode. Flags represents a set of different options for the new display mode. If omitted or given as 0, it will default to a simple software window. You can mix several flags together with the bitwise-or (|) operator. Possible flags are HWSURFACE (or the value 1), HWPALETTE, DOUBLEBUF, and/or FULLSCREEN. There are other flags available but these are the most usual. A full list of flags can be found in the pygame documentation.

The optional depth arguement is the requested bits per pixel. It will usually be left omitted, in which case the display will use the best/fastest pixel depth available.

You can create an OpenGL surface (for use with PyOpenGL) by passing the OPENGL flag. You will likely want to use the DOUBLEBUF flag when using OPENGL. In which case, the flip() function will perform the GL buffer swaps. When you are using an OPENGL video mode, you will not be able to perform most of the pygame drawing functions (fill, set_at, etc) on the display surface.

set_palette

```
pygame.display.set_palette([[r, g, b], ...]) -> None
```

Displays with a HWPALETTE have two palettes. The display Surface palette and the visible 'onscreen' palette.

This will change the video display's visible colormap. It does not effect the display Surface's base palette, only how it is displayed. Setting the palette for the display Surface will override this visible palette. Also passing no args will reset the display palette back to the Surface's palette.

You can pass an incomplete list of RGB values, and this will only change the first colors in the palette.

toggle_fullscreen

```
pygame.display.toggle_fullscreen() -> bool
```

Tells the window manager (if available) to switch between windowed and fullscreen mode. If available and successfull, will return true. Note, there is currently limited platform support for this call.

update

```
pygame.display.update([rectstyle]) -> None
```

This call will update a section (or sections) of the display screen. You must update an area of your display when you change its contents. If passed with no arguments, this will update the entire display surface. If you have many rects that need updating, it is best to combine them into a sequence and pass them all at once. This call will accept a sequence of rectstyle arguments. Any None's in the list will be ignored.

This call cannot be used on OPENGL displays, and will generate an exception.

pygame.draw

Contains routines to draw onto a surface.

Note that all drawing routines use direct pixel access, so the surfaces must be locked for use. The draw functions will temporarily lock the surface if needed, but if performing many drawing routines together, it would be best to surround the drawing code with a lock/unlock pair.

circle

```
pygame.draw.circle(Surface, color, pos, radius, width=0) -> Rect
```

Draws a circular shape on the Surface. The given position is the center of the circle, and radius is the size. The width argument is the thickness to draw the outer edge. If width is zero then the circle will be filled.

The color argument can be either a RGB sequence or mapped color integer.

This function will temporarily lock the surface.

ellipse

```
pygame.draw.ellipse(Surface, color, Rect, width=0) -> Rect
```

Draws an elliptical shape on the Surface. The given rectangle is the area that the circle will fill. The width argument is the thickness to draw the outer edge. If width is zero then the ellipse will be filled.

The color argument can be either a RGB sequence or mapped color integer.

This function will temporarily lock the surface.

line

```
pygame.draw.line(Surface, color, startpos, endpos, width=1) -> Rect
```

Draws a line on a surface. This will respect the clipping rectangle. A bounding box of the effected area is returned as a rectangle.

The color argument can be either a RGB sequence or mapped color integer.

This function will temporarily lock the surface.

lines

```
pygame.draw.lines(Surface, color, closed, point_array, width=1) -> Rect
```

Draws a sequence on a surface. You must pass at least two points in the sequence of points. The closed argument is a simple boolean and if true, a line will be draw between the first and last points. Note that specifying a linewidth wider than 1 does not fill in the gaps between the lines. Therefore wide lines and sharp corners won't be joined seamlessly.

This will respect the clipping rectangle. A bounding box of the effected area is returned as a rectangle.

The color argument can be either a RGB sequence or mapped color integer.

This function will temporarily lock the surface.

polygon

```
pygame.draw.polygon(Surface, color, pointslist, width=0) -> Rect
```

Draws a polygonal shape on the Surface. The given pointlist is the vertices of the polygon. The width argument is the thickness to draw the outer edge. If width is zero then the polygon will be filled.

The color argument can be either a RGB sequence or mapped color integer.

This function will temporarily lock the surface.

rect

```
pygame.draw.rect(Surface, color, Rect, width=0) -> Rect
```

Draws a rectangular shape on the Surface. The given Rect is the area of the rectangle. The width argument is the thickness to draw the outer edge. If width is zero then the rectangle will be filled.

The color argument can be either a RGB sequence or mapped color integer.

This function will temporarily lock the surface.

Keep in mind the Surface.fill() method works just as well for drawing filled rectangles. In fact the Surface.fill() can be hardware accelerated when the moons are in alignement.

pygame.event

Pygame handles all it's event messaging through an event queue. The routines in this module help you manage that event queue. The input queue is heavily dependent on the pygame display module. If the display has not been initialized and a video mode not set, the event queue will not really work.

The queue is a stack of Event objects, there are a variety of ways to access the data on the queue. From simply checking for the existance of events, to grabbing them directly off the stack.

All events have a type identifier. This event type is in between the values of NOEVENT and NUMEVENTS. All user defined events can have the value of USEREVENT or higher. It is recommended make sure your event id's follow this system.

To get the state of various input devices, you can forego the event queue and access the input devices directly with their appropriate modules; mouse, key, and joystick. If you use this method, remember that pygame requires some form of communication with the system window manager and other parts of the platform. To keep pygame in synch with the system, you will need to call pygame.event.pump() to keep everything current. You'll want to call this function usually once per game loop.

The event queue offers some simple filtering. This can help performance slightly by blocking certain event types from the queue, use the pygame.event.set_allowed() and pygame.event.set_blocked() to work with this filtering. All events default to allowed.

Also know that you will not receive any events from a joystick device, until you have initialized that individual joystick from the joystick module.

An Event object contains an event type and a readonly set of member data. The Event object contains no method functions, just member data. Event objects are retrieved from the pygame event queue. You can create your own new events with the pygame.event.Event() function.

All Event objects contain an event type identifier in the Event.type member. You may also get full access to the Event's member data through the Event.dict method. All other member lookups will be passed through to the Event's dictionary values.

While debugging and experimenting, you can print the Event objects for a quick display of its type and members. Events that come from the system will have a guaranteed set of member items based on the type. Here is a list of the Event members that are defined with each type.

QUIT	*none*
ACTIVEEVENT	gain, state
KEYDOWN	unicode, key, mod
KEYUP	key, mod
MOUSEMOTION	pos, rel, buttons
MOUSEBUTTONUP	pos, button
MOUSEBUTTONDOWN	pos, button
JOYAXISMOTION	joy, axis, value

JOYBALLMOTION	joy, ball, rel
JOYHATMOTION	joy, hat, value
JOYBUTTONUP	joy, button
JOYBUTTONDOWN	joy, button
VIDEORESIZE	size
VIDEOEXPOSE	*none*
USEREVENT	code

Event	- create new event object
clear	- remove all of an event type from the queue
event_name	- name for event type
get	- get all of an event type from the queue
get_blocked	- checks if an event is being blocked
get_grab	- query the state of input grabbing
peek	- query if any of event types are waiting
poll	- get an available event
post	- place an event on the queue
pump	- update the internal messages
set_allowed	- allows certain events onto the queue
set_blocked	- blocks certain events from the queue
set_grab	- grab all input events
wait	- wait for an event

Event

```
pygame.event.Event(type, [dict], [keyword_args]) -> Event
```

Creates a new event object. The type should be one of SDL's event numbers, or above USEREVENT. The given dictionary contains the keys that will be members of the new event.

Also, instead of passing a dictionary to create the event members, you can pass keyword arguments that will become the attributes of the new event.

clear

```
pygame.event.clear([type]) -> None
```

Pass this a type of event to discard, and it will remove all matching event types from the queue. If no types are passed, this will remove all the events from the queue. You may also optionally pass a sequence of event types. For example, to remove all the mouse motion events from the queue, you would call, 'pygame.event.clear(MOUSEMOTION)'.

event_name

```
pygame.event.event_name(event type) -> string
```

Returns the standard SDL name for an event type. Mainly helpful for debugging, when trying to determine what the type of an event is.

get

```
pygame.event.get([type]) -> list of Events
```

Pass this a type of event that you are interested in, and it will return a list of all matching event types from the queue. If no types are passed, this will return all the events from the queue. You may also optionally pass a sequence of event types. For example, to fetch all the keyboard events from the queue, you would call, 'pygame.event.get([KEYDOWN,KEYUP])'.

get_blocked

```
pygame.event.get_blocked(type) -> boolean
```

This returns a true value if the given event type is being blocked from the queue. You can optionally pass a sequence of event types, and it will return TRUE if any of the types are blocked.

get_grab

```
pygame.event.get_grab() -> bool
```

Returns true if the input is currently grabbed to your application.

peek

```
pygame.event.peek([type]) -> bool
```

Pass this a type of event that you are interested in, and it will return true if there are any of that type of event on the queue. If no types are passed, this will return the next event on the queue without removing it. You may also optionally pass a sequence of event types. For example, to find if any keyboard events are on the queue, you would call, 'pygame.event.peek([KEYDOWN,KEYUP])'.

poll

```
pygame.event.poll() -> Event
```

Returns next event on queue. If there is no event waiting on the queue, this will return an event with type NOEVENT.

post

```
pygame.event.post(Event) -> None
```

This will post your own event objects onto the event queue. You can past any event type you want, but some care must be taken. For example, if you post a MOUSEBUTTONDOWN event to the queue, it is likely any code receiving the event will excpect the standard MOUSEBUTTONDOWN attributes to be available, like 'pos' and 'button'.

pump

```
pygame.event.pump() -> None
```

For each frame of your game, you will need to make some sort of call to the event queue. This ensures your program can internally interact with the rest of the operating system. If you are not using other event functions in your game, you should call pump() to allow pygame to handle internal actions.

There are important things that must be dealt with internally in the event queue. The main window may need to be repainted. Certain joysticks must be polled for their values. If you fail to make a call to the event queue for too long, the system may decide your program has locked up.

set_allowed

```
pygame.event.set_allowed(type) -> None
```

By default, all events will appear from the queue. After you have blocked some event types, you can use this to re-enable them. You can optionally pass a sequence of event types.

You can pass None and this will allow no events on the queue.

set_blocked

```
pygame.event.set_blocked(type) -> None
```

By default, all events will appear from the queue. This will allow you to prevent event types from appearing on the queue. You can optionally pass a sequence of event types.

You can pass None and this will allow all events on the queue.

set_grab

```
pygame.event.set_grab(bool) -> None
```

Grabs all mouse and keyboard input for the display. Grabbing the input is not neccessary to receive keyboard and mouse events, but it ensures all input will go to your application. It also keeps the mouse locked inside your window. Set the grabbing on or off with the boolean argument. It is best to not always grab the input, since it prevents the end user from doing anything else on their system.

wait

```
pygame.event.wait() -> Event
```

Returns the current event on the queue. If there are no messages waiting on the queue, this will not return until one is available. Sometimes it is important to use this wait to get events from the queue, it will allow your application to idle when the user isn't doing anything with it.

pygame.font

The font module allows for rendering TrueType fonts into a new Surface object. This module is optional and requires SDL_ttf as a dependency. You may want to check for pygame.font to import and initialize before attempting to use the module.

Most of the work done with fonts are done by using the actual Font objects. The module by itself only has routines to initialize the module and create Font objects with pygame.font.Font().

Font	- create a new font object
get_default_font	- get the name of the default font
get_init	- get status of font module initialization
init	- initialize the display module
quit	- uninitialize the font module

Font

```
pygame.font.Font(file, size) -> Font
```

This will create a new font object. The given file must be a filename to a TTF file. The font loader does not work with python file-like objects. The size represents the height of the font in pixels. The file argument can be 'None', which will use a plain default font.

get_default_font

```
pygame.font.get_default_font() -> string
```

returns the name for the default truetype font.

get_init

```
pygame.font.get_init() -> bool
```

Returns true if the font module is currently intialized.

init

```
pygame.font.init() -> None
```

Manually initialize the font module. Will raise an exception if it cannot be initialized. It is safe to call this function if font is currently initialized.

quit

```
pygame.font.quit() -> none
```

Manually uninitialize SDL's video subsystem. It is safe to call this if font is currently not initialized.

pygame.image

This module contains functions to transfer images in and out of Surfaces. At the minimum the included load() function will support BMP files. If SDL_image is properly installed when pygame is installed, it will support all the formats included with SDL_image. You can call the get_extended() function to test if the SDL_image support is available.

Some functions that communicate with other libraries will require that those libraries are properly installed. For example, the save() function can only save OPENGL surfaces if pyopengl is available.

fromstring

```
pygame.image.fromstring(string, size, format, flipped=0) -> Surface
```

This will create a new Surface from a copy of raw data in a string. This can be used to transfer images from other libraries like PIL's fromstring().

The flipped argument should be set to true if the image in the string is.

The format argument is a string representing which type of string data you need. It can be one of the following, \"P\" for 8bit palette indices. \"RGB\" for 24bit RGB data, \"RGBA\" for 32bit RGB and alpha, or \"RGBX\" for 32bit padded RGB colors.

These flags are a subset of the formats supported the PIL Python Image Library. Note that the \"P\" format only create an 8bit surface, but the colormap will be all black.

get_extended

```
pygame.image.get_extended() -> int
```

This will return a true value if the extended image formats from SDL_image are available for loading.

load

```
pygame.image.load(file, [namehint]) -> Surface
```

This will load an image into a new surface. You can pass it either a filename, or a python file-like object to load the image from. If you pass a file-like object that isn't actually a file (like the StringIO class), then you might want to also pass either the filename or extension as the namehint string. The namehint can help the loader determine the filetype.

If pygame was installed without SDL_image support, the load will only work with BMP images. You can test if SDL_image is available with the get_extended() function. These extended file formats usually include GIF, PNG, JPG, PCX, TGA, and more.

If the image format supports colorkeys and pixel alphas, the load() function will properly load and configure these types of transparency.

save

```
pygame.image.save(Surface, file) -> None
```

This will save your surface as a BMP or TGA image. The given file argument can be either a filename or a python file-like object. This will also work under OPENGL display modes.

The image will default to save with the TGA format. If the filename has the BMP extension, it will use the BMP format.

tostring

```
pygame.image.tostring(Surface, format, flipped=0) -> string
```

This will copy the image data into a large string buffer. This can be used to transfer images to other libraries like PIL's fromstring() and PyOpenGL's glTexImage2D().

The flipped argument will cause the output string to have it's contents flipped vertically.

The format argument is a string representing which type of string data you need. It can be one of the following, \"P\" for 8bit palette indices. \"RGB\" for 24bit RGB data, \"RGBA\" for 32bit RGB and alpha, or \"RGBX\" for 32bit padded RGB colors.

These flags are a subset of the formats supported the PIL Python Image Library. Note that the \"P\" format only will work for 8bit Surfaces.

If you ask for the \"RGBA\" format and the image only has colorkey data. An alpha channel will be created from the colorkey values.

pygame.joystick

The joystick module provides a few functions to initialize the joystick subsystem and to manage the Joystick objects. These objects are created with the pygame.joystick.Joystick() function. This function needs a joystick device number to work on. All joystick devices on the system are enumerated for use as a Joystick object. To access most of the Joystick functions, you'll need to init() the Joystick. (note that the joystick module will already be initialized). When multiple Joysticks objects are created for the same joystick device, the state and values for those Joystick objects will be shared.

You can call the Joystick.get_name() and Joystick.get_id() functions without initializing the Joystick object.

Joystick control values are only updated during the calls to the event queue. Call pygame.event.pump() if you are not using the event queue for any input handling. Once a joystick object has been initialized, it will start to send joystick events to the input queue.

Be sure to understand there is a difference between the joystick module and the Joystick objects.

Joystick	- create new joystick object
get_count	- query number of joysticks on system
get_init	- query initialization of joystick module
init	- initialize joystick module
quit	- uninitialize joystick module

Joystick

```
pygame.joystick.Joystick(id) -> Joystick
```

Creates a new joystick object for the given device id. The given id must be less than the value from pygame.joystick.get_count().

get_count

```
pygame.joystick.get_count() -> int
```

Returns the number of joysticks devices available on the system.

get_init

```
pygame.joystick.get_init() -> bool
```

Returns true when the joystick module is initialized.

init

```
pygame.joystick.init() -> None
```

Initialize the joystick module manually

quit

```
pygame.joystick.quit() -> None
```

Uninitialize the joystick module manually

pygame.key

Contains routines for dealing with the keyboard. All keyboard events can be retreived through the pygame.event module. With the key module, you can get the current state of the keyboard, as well as set the rate of keyboard repeating and lookup names of keysyms.

get_focused

```
pygame.key.get_focused() -> bool
```

Returns true when the application has the keyboard input focus.

get_mods

```
pygame.key.get_mods() -> int
```

Returns a bitwise combination of the pressed state for all modifier keys (KMOD_LSHIFT, etc).

get_pressed

```
pygame.key.get_pressed() -> tuple of bools
```

This gives you a big tuple with the pressed state for all keys. You index the sequence using the keysym constant (K_SPACE, etc)

name

```
pygame.key.name(int) -> string
```

This will provide you with the keyboard name for a keysym. For example 'pygame.key.name(K_SPACE)' will return 'space'.

set_mods

```
pygame.key.set_mods(int) -> None
```

Allows you to control the internal state of the modifier keys. Pass an interger built from using the bitwise-or (|) of all the modifier keys you want to be treated as pressed.

set_repeat

```
pygame.key.set_repeat([delay, interval]) -> None
```

When the keyboard repeat is enabled, you will receive multiple KEYDOWN events when the user holds a key. You can control the repeat timing with the delay and interval values. If no arguments are passed, keyboard repeat will be disabled.

Good values for delay and interval are 500 and 30.

Delay is the amount of milliseconds before the first repeated KEYDOWN event is received. The interval is the amount of milliseconds for each repeated KEYDOWN event after that.

pygame.mixer

Contains sound mixer routines and objects. The mixer module is an optional pygame module, dependent on the SDL_mixer library. This module contains the usual routines needed to initialize the module. One difference is the pygame.mixer.init() function takes several optional arguments. These arguments control the playback rates and datatypes for the sound playback. If you do need specific control over the playback rate, but don't want to bother with hand-initializing the modules, there is a function named pygame.mixer.pre_init() which takes the same arguments as init(), but only sets the new default values. You can call this before pygame.init() and not have to worry about the pygame module initialization order.

Sound objects are created from the pygame.mixer.Sound() function. Simple sound playback can simply use the Sound.play() method to play the sound. Each Sound object can be played multiple times simultaneously. If you desire more specific control over the Sound objects, you can access the Channel objects with functions like pygame.mixer.Channel().

The mixer defaults to supporting 8 simultaneous soundfiles. You can change the number of available sound channels at any time with the set_num_channels() function.

All loaded Sound objects are resampled to match the same format that pygame.mixer is initialized to. The current SDL resampling functions are not that good, so it is best if you initialize pygame.mixer to the same format as your sound resources. Also setting the mixer frequency to even multiples of your sound resources will result in a cleaner conversion.

The mixer also contains a special channel for music. You can control the music channel through pygame.mixer.music.

Channel	- get channel object
Sound	- load a new soundfile
fadeout	- fadeout all channels
find_channel	- find an available sound channel
get_busy	- query busy channels
get_init	- query initialization for the mixer
get_num_channels	- query the number of channels
init	- initialize mixer module
pause	- pause all channels
pre_init	- presets the init default values
quit	- unitializes the mixer
set_num_channels	- sets the number of available channels
set_reserved	- reserves first given channels
stop	- stop all channels

unpause - restart any pause channels

Channel

```
pygame.mixer.Channel(int) -> Channel
```

Get a channel object for the given channel. This number must be less that the current number of channels.

Sound

```
pygame.mixer.Sound(file) -> Sound
```

Loads a new sound object from a WAV file. File can be a filename or a file-like object. The sound will be converted to match the current mode of the mixer.

fadeout

```
pygame.mixer.fadeout(millisec) -> None
```

Fade out all the playing channels over the given number of milliseconds.

find_channel

```
pygame.mixer.find_channel([force]) -> Channel
```

Find a sound channel that is not busy. If the force argument is passed as a nonzero number, this will return the channel of the longest running sound. If not forced, and there are no available channels, returns None.

get_busy

```
pygame.mixer.get_busy() -> int
```

Returns the number of current active channels. This is not the total channels, but the number of channels that are currently playing sound.

get_init

```
pygame.mixer.get_init() -> (frequency,format,stereo)
```

Returns a tuple containing the initialized state of the mixer module. If the module has not been initialized, it will return None.

get_num_channels

```
pygame.mixer.get_num_channels() -> int
```

Gets the current number of channels available for the mixer. This value can be changed with set_num_channels(). This value defaults to 8 when the mixer is first initialized.

init

```
pygame.mixer.init([freq, [size, [stereo, [buffersize]]]]) -> None
```

Initializes the mixer module. Usually no arguments will be needed, the defaults are 22050 frequency data in stereo with signed 16bit data. The size argument can be 8 or 16 for unsigned data, or -8 or -16 for signed data. The default buffersize is 1024 samples, sometimes a larger value is required.

On many platforms it is important that the display module is initialized before the audio. (that is, if the display will be initialized at all). You can easily use the pygame.init() function to cleanly initialize everything, but first use the pygame.mixer.pre_init() function to change the default values for this init().

pause

```
pygame.mixer.pause() -> None
```

Temporarily stops playback on all the mixer channels.

pre_init

```
pygame.mixer.pre_init([freq, [size, [stereo, [buffersize]]]]) -> None
```

This routine is usefull when you want to customize the sound mixer playback modes. The values you pass will change the default values used by pygame.mixer.init(). This way you can still use the pygame automatic initialization to ensure everything happens in the right order, but set the desired mixer mode.

quit

```
pygame.mixer.quit() -> None
```

This will stop all playing sounds and uninitialize the mixer module

set_num_channels

```
pygame.mixer.set_num_channels(int) -> None
```

Sets the current number of channels available for the mixer. This value defaults to 8 when the mixer is first initialized. If the value is decreased, sounds playing in channels above the new value will stop.

set_reserved

```
pygame.mixer.set_reserved(numchans) -> None
```

Reserves numchan channels. Reserved channels won't be used when a sound is played without using a specific channel object. In otherwords, just calling Sound.play() will not use the reserved channels. They must implicitly be used with Channel.play().

stop

```
pygame.mixer.stop() -> None
```

Stop the playback on all mixer channels.

unpause

```
pygame.mixer.unpause() -> None
```

Restarts playback of any paused channels.

pygame.mixer.music

The music module is tied closely to the pygame.mixer module. It is an optional module since it depends on the SDL_mixer library. You should not manually import the music library. Instead it is automatically included as a part of the mixer library. The default module path to music is pygame.mixer.music.

The difference between playback of music and playback of sounds is that the music object is not loaded and decoded all at once, instead the music data is streamed and decoded during playback. There can only be one music file loaded at a single time. Loading a new music file will replace any currently loaded music.

The music module has many of the same types of functions as the Sound objects. The main difference is only one music object can be loaded at a time, with the load() function. Music must be stored in an individual file on the system, it cannot be loaded from special file-like objects through python.

fadeout

```
pygame.mixer.music.fadeout(millisec) -> None
```

Fades out the current playing music and stops it over the given milliseconds.

get_busy

```
pygame.mixer.music.get_busy() -> bool
```

Returns true if music is currently playing

get_endevent

```
pygame.mixer.music.get_endevent([eventid]) -> int
```

When the music has finished playing, you can optionally have pygame place a user defined message on the event queue. If there is no callback event set, NOEVENT will be returned. Otherwise it will return the id of the current music finishe event.

get_pos

```
pygame.mixer.music.get_pos() -> val
```

Get the current (interpolated) time position of the music. Value is in ms, just like get_ticks().

The returned time is only tracking the amount of music played. It will not reflect the result of starting the music at an offset.

get_volume

```
pygame.mixer.music.get_volume() -> val
```

Get the current volume for the music. Value is between 0.0 and 1.0.

load

```
pygame.mixer.music.load(filename) -> None
```

Load a music object as the current music. The music only handles one music as the current. If music is currently playing, it will be stopped and replaced with the given one. Loading music only supports filenames, not file-like objects.

pause

```
pygame.mixer.music.pause() -> None
```

Temporarily stops the current music.

play

```
pygame.mixer.music.play(loops=0, startpos=0.0) -> None
```

Starts playing the current loaded music. This will restart the sound if it is playing. Loops controls how many extra time the sound will play, a negative loop will play indefinitely, it defaults to 0.

The starting position argument controls where in the music the song starts playing. The starting position is dependent on the format of music playing. MP3 and OGG use the position as time (in seconds). MOD music it is the pattern order number. Passing a startpos will raise a NotImplementedError if it cannot set the start position (or your version of SDL_mixer is too old)

queue

```
pygame.mixer.music.queue(soundfile) -> None
```

This will load a music file and queue it. A queued music file will begin as soon as the current music naturally ends. If the current music is ever stopped or changed, the queued song will be lost.

rewind

```
pygame.mixer.music.rewind() -> None
```

Restarts playback of the current music.

set_endevent

```
pygame.mixer.music.set_endevent([eventid]) -> None
```

When the music has finished playing, you can optionally have pygame place a user defined message on the event queue. If the eventid field is omittied or NOEVENT, no messages will be sent when the music finishes playing. Once the endevent is set, it will be called every time the music finished playing.

set_volume

```
pygame.mixer.music.set_volume(val) -> None
```

Sets the current volume for the music. Value is between 0.0 and 1.0.

stop

```
pygame.mixer.music.stop() -> None
```

Stops playback of the current music.

unpause

```
pygame.mixer.music.unpause() -> None
```

Restarts playback of the current music object when paused.

pygame.mouse

Contains routines for dealing with the mouse. All mouse events are retrieved through the pygame.event module. The mouse module can be used to get the current state of the mouse. It can also be used to set the state of the system cursor.

If you hide the mouse cursor with pygame.mouse.set_visible(0) and lock the mouse focus to your game with pygame.event.set_grab(1), the hidden mouse will be forced to the center of the screen. This will help your relative mouse motions keep from getting stuck on the edges of the screen.

get_cursor

```
pygame.mouse.get_cursor() -> size, hotspot, xormasks, andmasks
```

The mouse cursor data is the same as those passed into set_cursor.

get_focused

```
pygame.mouse.get_focused() -> bool
```

Returns true when the application is receiving the mouse input focus.

get_pos

```
pygame.mouse.get_pos() -> x, y
```

Returns the current position of the mouse cursor. This is the absolute mouse position inside your game window.

get_pressed

```
pygame.mouse.get_pressed() -> button1, button2, button3
```

This will return a small sequence containing the pressed state of each mouse button.

get_rel

```
pygame.mouse.get_rel() -> x, y
```

Returns the total distance the mouse has moved since your last call to get_rel(). On the first call to get_rel the movement will always be 0,0.

When the mouse is at the edges of the screen, the relative movement will be stopped. See mouse_visible for a way to resolve this.

set_cursor

```
pygame.mouse.set_cursor(size, hotspot, xormasks, andmasks) -> None
```

When the mouse cursor is visible, it will be displayed as a black and white bitmap using the given bitmask arrays. The size is a sequence containing the cursor width and height. Hotspot is a sequence containing the cursor hotspot position. xormasks is a sequence of bytes containing the cursor xor data masks. Lastly is andmasks, a sequence of bytes containting the cursor bitmask data.

Width must be a multiple of 8, and the mask arrays must be the correct size for the given width and height. Otherwise an exception.

set_pos

```
pygame.mouse.set_pos(pos) -> None
```

Moves the mouse cursor to the specified position. This will generate a MOUSEMOTION event on the input queue. The pos argument is a 2-number-sequence containing the desired x and y position.

set_visible

```
pygame.mouse.set_visible(bool) -> bool
```

Shows or hides the mouse cursor. This will return the previous visible state of the mouse cursor.

Note that when the cursor is hidden and the application has grabbed the input. pygame will force the mouse to stay in the center of the screen. Since the mouse is hidden it won't matter that it's not moving, but it will keep the mouse from the edges of the screen so the relative mouse position will always be true.

pygame.movie

The movie module is an optional pygame module that allows for decoding and playback of MPEG movie files. The module only contains a single function, Movie() which creates a new Movie object.

Movies are played back in background threads, so there is very little management needed on the user end. Just load the Movie, set the destination, and Movie.play()

Movies will only playback audio if the pygame.mixer module is not initialized. It is easy to temporarily call pygame.mixer.quit() to disable audio, then create and play your movie. Finally calling pygame.mixer.init() again when finished with the Movie.

Movie - load a new MPEG stream

Movie

```
pygame.movie.Movie(file) -> Movie
```

Loads a new movie stream from a MPEG file. The file argument is either a filename, or any python file-like object.

pygame.sndarray

Contains routines for mixing numeric arrays with sounds

array

```
pygame.sndarray.array(Sound) -> Array
```

Creates an array with a copy of the sound data.

make_sound

```
pygame.sndarray.make_sound(array) -> Sound
```

Create a new playable Sound object from array data the Sound will be a copy of the array samples.

The array must be 1-dimensional for mono sound, and. 2-dimensional for stereo.

samples

```
pygame.sndarray.samples(Surface) -> Array
```

This will return an array that directly references the samples in the array.

pygame.surfarray

Contains routines for mixing numeric arrays with surfaces

array2d	- get a 2d array copied from a surface
array3d	- get a 3d array copied from a surface
array_alpha	- get an array with a surface pixel alpha values
array_colorkey	- get an array with a surface colorkey values
blit_array	- quickly transfer an array to a Surface
make_surface	- create a new Surface from array data
map_array	- map an array with RGB values into mapped colors
pixels2d	- get a 2d reference array to a surface
pixels3d	- get a 3d reference array to a surface
pixels_alpha	- get a reference array to a surface alpha data

array2d

```
pygame.surfarray.array2d(Surface) -> Array
```

This returns a new contigous 2d array. Think of it as a 2d image array with a mapped pixel value at each index.

This function will temporarily lock the surface.

array3d

```
pygame.surfarray.array3d(Surface) -> Array
```

This returns a new contigous 3d array. Think of it as a 2d image array with an RGB array for each pixel value.

This function will temporarily lock the surface.

array_alpha

```
pygame.surfarray.array_alpha(Surface) -> Array
```

This returns a new contigous 2d array with the alpha values of an image as unsigned bytes. If the surface has no alpha, an array of all opaque values is returned.

This function will temporarily lock the surface.

array_colorkey

```
pygame.surfarray.array_colorkey(Surface) -> Array
```

This returns a new contigous 2d array with the colorkey values of an image as unsigned bytes. If the surface has no colorkey, an array of all opaque values is returned. Otherwise the array is either 0's or 255's.

This function will temporarily lock the surface.

blit_array

```
pygame.surfarray.blit_array(surf, array) -> None
```

Transfer an array of any type (3d or 2d) onto a Surface. The array must be the same dimensions as the destination Surface. While you can assign the values of an array to the pixel referenced arrays, using this blit method will usually be quicker because of it's smarter handling of noncontiguous arrays. Plus it allows you to blit from any image array type to any surface format in one step, no internal conversions.

This function will temporarily lock the surface.

make_surface

```
pygame.surfarray.make_surface(array) -> Surface
```

Create a new software surface that closely resembles the data and format of the image array data.

map_array

```
pygame.surfarray.map_array(surf, array3d) -> array2d
```

Create a new array with the RGB pixel values of a 3d array into mapped color values in a 2D array.

Just so you know, this can also map a 2D array with RGB values into a 1D array of mapped color values

pixels2d

```
pygame.surfarray.pixels2d(Surface) -> Array
```

This returns a new noncontigous 2d array that directly effects a Surface's contents. Think of it as a 2d image array with a mapped pixel value at each index.

This will not work on 24bit surfaces, since there is no native 24bit data type to access the pixel values.

This function will lock the given surface, and it will remained locked for as long as the pixel array exists

pixels3d

```
pygame.surfarray.pixels3d(Surface) -> Array
```

This returns a new noncontigous 3d array that directly effects a Surface's contents. Think of it as a 2d image array with an RGB array for each pixel value.

This will only work for 24 and 32 bit surfaces, where the RGB components can be accessed as 8-bit components.

This function will lock the given surface, and it will remained locked for as long as the pixel array exists

pixels_alpha

```
pygame.surfarray.pixels_alpha(Surface) -> Array
```

This returns a new noncontigous array that directly effects a Surface's alpha contents.

This will only work for 32bit surfaces with a pixel alpha channel enabled.

This function will lock the given surface, and it will remained locked for as long as the pixel array exists

pygame.time

Contains routines to help keep track of time. The timer resolution on most systems is around 10ms.

All times are represented in milliseconds, which is simply Seconds*1000. (therefore 2500 milliseconds is 2.5 seconds)

You can also create Clock instances to keep track of framerate.

Clock	- create a new clock
delay	- accurately delay for a number of milliseconds
get_ticks	- milliseconds since initialization
set_timer	- control timer events
wait	- yielding delay for a number of milliseconds

Clock

```
pygame.time.Clock() -> Clock
```

Clocks are used to track and control the framerate of a game. You create the objects with the time.Clock() function. The clock can be used to limit the framerate of a game, as well as track the time used per frame.

delay

```
pygame.time.delay(millseconds) -> time
```

Will pause for a given number of milliseconds. This function will use the CPU in order to make the delay more accurate than wait().

This returns the actual number of milliseconds used.

get_ticks

```
pygame.time.get_ticks() -> int
```

This is the time in milliseconds since the pygame.time was imported. Always returns 0 before pygame.init() is called.

set_timer

```
pygame.time.set_timer(eventid, milliseconds) -> int
```

Every event id can have a timer attached to it. Calling this will set the timer in milliseconds for that event. setting milliseconds to 0 or less will disable that timer. When a timer for an event is set, that event will be placed on the event queue every given number of milliseconds.

wait

```
pygame.time.wait(millseconds) -> time
```

Will pause for a given number of milliseconds. This function sleeps the process to better share the CPU with other processes. It is less accurate than the delay() function.

This returns the actual number of milliseconds used.

pygame.transform

Contains routines to transform a Surface image.

All transformation functions take a source Surface and return a new copy of that surface in the same format as the original.

Some of the filters are 'destructive', which means if you transform the image one way, you can't transform the image back to the exact same way as it was before. If you plan on doing many transforms, it is good practice to keep the original untransformed image, and only translate that image.

flip	- flips a surface on either axis
rotate	- rotate a Surface
rotozoom	- smoothly scale and/or rotate an image
scale	- scale a Surface to an arbitrary size
scale2x	- doubles the size of the image with advanced scaling

flip

```
pygame.transform.flip(Surface, xaxis, yaxis) -> Surface
```

Flips the image on the x-axis or the y-axis if the argument for that axis is true.

The flip operation is nondestructive, you may flip the image as many times as you like, and always be able to recreate the exact original image.

rotate

```
pygame.transform.rotate(Surface, angle) -> Surface
```

Rotates the image counterclockwise with the given angle (in degrees). The angle can be any floating point value (negative rotation amounts will do clockwise rotations)

Unless rotating by 90 degree increments, the resulting image size will be larger than the original. There will be newly uncovered areas in the image. These will filled with either the current colorkey for the Surface, or the topleft pixel value. (with the alpha channel zeroed out if available)

This transformation is not filtered.

rotozoom

```
pygame.transform.rotozoom(Surface, angle, zoom) -> Surface
```

The angle argument is the number of degrees to rotate counter-clockwise. The angle can be any floating point value. (negative rotation amounts will do clockwise rotations)

This will smoothly rotate and scale an image in one pass. The resulting image will always be a 32bit version of the original surface. The scale is a multiplier for the image size, and angle is the degrees to rotate counter clockwise.
It calls the SDL_rotozoom library which is compiled in. Note that the code in SDL_rotozoom is fairly messy and your resulting image could be shifted and contain artifacts.

scale

```
pygame.transform.scale(Surface, size) -> Surface
```

This will resize a surface to the given resolution. The size is simply any 2 number sequence representing the width and height.

This transformation is not filtered.

scale2x

```
pygame.transform.scale2x(Surface) -> Surface
```

This will return a new image that is double the size of the original. It uses the AdvanceMAME Scale2X algorithm which does a 'jaggie-less' scale of bitmap graphics.

This really only has an effect on simple images with solid colors. On photographic and antialiased images it will look like a regular unfiltered scale.

CD

The CD object represents a CDROM drive and allows you to access the CD inside that drive. All functions (except get_name() and get_id()) require the CD object to be initialized. This is done with the CD.init() function.

Be sure to understand there is a difference between the cdrom module and the CD objects.

eject	- ejects cdrom drive
get_all	- get all track information for the cd
get_busy	- checks if the cd is currently playing
get_current	- get current position of the cdrom
get_empty	- checks for a cd in the drive
get_id	- get device id number for drive
get_init	- check if cd is initialized
get_name	- query name of cdrom drive
get_numtracks	- get number of tracks on cd
get_paused	- checks if the cd is currently paused
get_track_audio	- check if a track has audio data
get_track_length	- check the length of an audio track
get_track_start	- check the start of an audio track
init	- initialize a cdrom device for use
pause	- pause playing cdrom
play	- play music from cdrom
quit	- uninitialize a cdrom device for use
resume	- resume paused cdrom
stop	- stops playing cdrom

eject

```
CD.eject() -> None
```

Ejects the media from the CDROM drive. If the drive is empty, this will open the CDROM drive.

get_all

```
CD.get_all() -> tuple
```

Returns a tuple with values for each track on the CD. Each item in the tuple is a tuple with 4 values for each track. First is a boolean set to true if this is an audio track. The next 3 values are the start time, end time, and length of the track.

get_busy

```
CD.get_busy() -> bool
```

Returns a true value if the cd drive is currently playing. If the drive is paused, this will return false.

get_current

```
CD.get_current() -> track, seconds
```

Returns the current track on the cdrom and the number of seconds into that track.

get_empty

```
CD.get_empty() -> bool
```

Returns a true value if the cd drive is empty.

get_id

```
CD.get_id() -> idnum
```

Returns the device id number for this cdrom drive. This is the same number used in the call to pygame.cdrom.CD() to create this cd device. The CD object does not need to be initialized for this function to work.

get_init

```
CD.get_init() -> bool
```

Returns a true value if the CD is initialized.

get_name

```
CD.get_name(id) -> string
```

Returns the name of the CDROM device, given by the system. This function can be called before the drive is initialized.

get_numtracks

```
CD.get_numtracks() -> numtracks
```

Returns the number of available tracks on the CD. Note that not all of these tracks contain audio data. Use CD.get_track_audio() to check the track type before playing.

get_paused

```
CD.get_paused() -> bool
```

Returns a true value if the cd drive is currently paused.

get_track_audio

```
CD.get_track_audio(track) -> bool
```

Returns true if the cdrom track contains audio data.

get_track_length

```
CD.get_track_length(track) -> seconds
```

Returns the number of seconds in an audio track. If the track does not contain audio data, returns 0.0.

get_track_start

```
CD.get_track_start(track) -> seconds
```

Returns the number of seconds an audio track starts on the cd.

init

```
CD.init() -> None
```

In order to call most members in the CD object, the CD must be initialized. You can initialzie the CD object at anytime, and it is ok to initialize more than once.

pause

```
CD.pause() -> None
```

Pauses the playing CD. If the CD is not playing, this will do nothing.

play

```
CD.play(track, [start, end]) -> None
```

Play an audio track on a cdrom disk. You may also optionally pass a starting and ending time to play of the song. If you pass the start and end time in seconds, only that portion of the audio track will be played. If you only provide a start time and no end time, this will play to the end of the track. You can also pass 'None' as the ending time, and it will play to the end of the cd.

quit

```
CD.quit() -> None
```

After you are completely finished with a cdrom device, you can use this quit() function to free access to the drive. This will be cleaned up automatically when the cdrom module is. uninitialized. It is safe to call this function on an uninitialized CD.

resume

```
CD.resume() -> int
```

Resumes playback of a paused CD. If the CD has not been pause, this will do nothing.

stop

```
CD.stop() -> int
```

Stops the playing CD. If the CD is not playing, this will do nothing.

Channel

Channel objects represent a single channel of sound. Each channel can only playback one Sound object at a time. If your application only requires simply sound playback, you will usually not need to bother with the Channel objects, they exist for finer playback control.

Sound objects can be retrieved from the pygame.mixer module with functions like pygame.mixer.Channel() and pygame.mixer.find_channel(). Also, each time you call Sound.play() a Channel object will be returned, representing the channel that sound is playing on.

fadeout

```
Channel.fadeout(millisec) -> None
```

Fade out the playing sound and stops it over the given milliseconds.

get_busy

```
Channel.get_busy() -> bool
```

Returns true when there is a sound actively playing on this channel.

get_endevent

```
Channel.get_endevent() -> event_type
```

Returns the end event type for this Channel. If the return value is NOEVENT, then no events will be sent when playback ends.

get_queue

```
Channel.get_queue() -> Sound
```

Return the currently queued Sound object on this channel. This will return None if there is nothing queued.

get_sound

```
Channel.get_sound() -> Sound
```

Return the currently playing Sound object on this channel. This will return None if there is nothing playing.

get_volume

```
Channel.get_volume() -> val
```

Returns the current volume for this sound object. The value is between 0.0 and 1.0.

pause

```
Channel.pause() -> None
```

Stops the sound that is playing on this channel, but it can be resumed with a call to unpause()

play

```
Channel.play(Sound, [loops, [maxtime]]) -> None
```

Starts playing a given sound on this channel. If the channels is currently playing a different sound, it will be replaced/restarted with the given sound. Loops controls how many extra times the sound will play, a negative loop will play indefinitely, it defaults to 0. Maxtime is the number of totalmilliseconds that the sound will play. It defaults to forever (-1).

queue

```
Channel.queue(Sound) -> None
```

When you queue a sound on a channel, it will begin playing immediately when the current playing sound finishes. Each channel can only have a single Sound object queued. The queued sound will only play when the current Sound finishes naturally, not from another call to stop() or play().

If there is no currently playing sound on this Channel it will begin playback immediately.

This will only work with SDL_mixer greater than version 1.2.3

set_endevent

```
Channel.set_endevent([event_type]) -> None
```

When you set an endevent for a channel, that event type will be put on the pygame event queue everytime a sound stops playing on that channel. This is slightly different than the music object end event, because this will trigger an event anytime the music stops. If you call stop() or play() on the channel, it will fire an event. An event will also be fired when playback switches to a queued Sound.

Pass no argument to stop this channel from firing events

set_volume

```
Channel.set_volume(val, [stereoval]) -> None
```

Sets the volume for the channel. The channel's volume level is mixed with the volume for the active sound object. The value is between 0.0 and 1.0.

If mixer is using stereo, you can set the panning for audio by supplying a volume for the left and right channels. If SDL_mixer cannot set the panning, it will average the two volumes. Panning requires SDL_mixer-1.2.1.

stop

```
Channel.stop() -> None
```

Stops the sound that is playing on this channel.

unpause

```
Channel.unpause() -> None
```

Restarts a paused channel where it was paused.

Clock

Clocks are used to track and control the framerate of a game. You create the objects with the time.Clock() function. The clock can be used to limit the framerate of a game, as well as track the time used per frame. Use the pygame.time.Clock() function to create new Clock objects.

get_fps

```
Clock.get_fps() -> float
```

This computes the running average of frames per second. This is the number of times the tick() method has been called per second.

get_rawtime

```
Clock.get_rawtime() -> int
```

This is similar to get_time(). It does not include the number of milliseconds that were delayed to keep the clock tick under a given framerate.

get_time

```
Clock.get_time() -> int
```

This is the same value returned from the call to Clock.tick(). it is the number of milliseconds that passed between the last two calls to tick().

tick

```
Clock.tick([ticks_per_sec_delay]) -> milliseconds
```

Updates the number of ticks for this clock. It should usually be called once per frame. If you pass the optional delay argument the function will delay to keep the game running slower than the given ticks per second. The function also returns the number of milliseconds passed since the previous call to tick().

Font

The font object is created only from pygame.font.Font(). Once a font is created it's size and TTF file cannot be changed. The Font objects are mainly used to render() text into a new Surface. The Font objects also have a few states that can be set with set_underline(bool), set_bold(bool), set_italic(bool). Each of these functions contains an equivalent get_XXX() routine to find the current state. There are also many routines to query the dimensions of the text. The rendering functions work with both normal python strings, as well as with unicode strings.

get_ascent	- gets the font ascent
get_bold	- status of the bold attribute
get_descent	- gets the font descent
get_height	- average height of font glyph
get_italic	- status of the italic attribute
get_linesize	- gets the font recommended linesize
get_underline	- status of the underline attribute
render	- render text to a new image
set_bold	- assign the bold attribute
set_italic	- assign the italic attribute
set_underline	- assign the underline attribute
size	- size of rendered text

get_ascent

```
Font.get_ascent() -> int
```

Returns the ascent for the font. The ascent is the number of pixels from the font baseline to the top of the font.

get_bold

```
Font.get_bold() -> bool
```

Get the current status of the font's bold attribute

get_descent

```
Font.get_descent() -> int
```

Returns the descent for the font. The descent is the number of pixels from the font baseline to the bottom of the font.

get_height

```
Font.get_height() -> int
```

Returns the average size of each glyph in the font.

get_italic

```
Font.get_italic() -> bool
```

Get the current status of the font's italic attribute

get_linesize

```
Font.get_linesize() -> int
```

Returns the linesize for the font. Each font comes with it's own recommendation for the spacing number of pixels between each line of the font.

get_underline

```
Font.get_underline() -> bool
```

Get the current status of the font's underline attribute

render

```
Font.render(text, antialias, fore_RGBA, [back_RGBA]) -> Surface
```

Render the given text onto a new image surface. The given text can be standard python text or unicode. Antialiasing will smooth the edges of the font for a much cleaner look. The foreground and background color are both RGBA, the alpha component is ignored if given. If the background color is omitted, the text will have a transparent background.

Note that font rendering is not thread safe, therefore only one thread can render text at any given time.

Also, rendering smooth text with underlines will crash with SDL_ttf less that version 2.0, be careful.

et_bold

```
Font.set_bold(bool) -> None
```

Enables or disables the bold attribute for the font. Making the font bold does not work as well as you expect.

set_italic

```
Font.set_italic(bool) -> None
```

Enables or disables the italic attribute for the font.

set_underline

```
Font.set_underline(bool) -> None
```

Enables or disables the underline attribute for the font.

size

```
Font.size(text) -> width, height
```

Computes the rendered size of the given text. The text can be standard python text or unicode. Changing the bold and italic attributes can change the size of the rendered text.

Joystick

The Joystick object represents a joystick device and allows you to access the controls on that joystick. All functions (except get_name() and get_id()) require the Joystick object to be initialized. This is done with the Joystick.init() function.

Joystick control values are only updated during the calls to the event queue. Call pygame.event.pump() if you are not using the event queue for any input handling. Once a joystick object has been initialized, it will start to send joystick events to the input queue.

Be sure to understand there is a difference between the joystick module and the Joystick objects.

get_axis

```
Joystick.get_axis(axis) -> float
```

Returns the current position of a joystick axis. The value will range from -1 to 1 with a value of 0 being centered. You may want to take into account some tolerance to handle jitter, and joystick drift may keep the joystick from centering at 0 or using the full range of position values.

get_ball

```
Joystick.get_ball(button) -> x, y
```

Returns the relative movement of a joystick button. The value is a x, y pair holding the relative movement since the last call to get_ball()

get_button

```
Joystick.get_button(button) -> bool
```

Returns the current state of a joystick button.

get_hat

```
Joystick.get_hat(button) -> x, y
```

Returns the current position of a position hat. The position is given as two values representing the X and Y position for the hat. (0, 0) means centered. A value of -1 means left/down a value of one means right/up

get_id

```
Joystick.get_id() -> idnum
```

Returns the device id number for this Joystick. This is the same number used in the call to pygame.joystick.Joystick() to create the object. The Joystick does not need to be initialized for this function to work.

get_init

```
Joystick.get_init() -> bool
```

Returns a true value if the Joystick is initialized.

get_name

```
Joystick.get_name(id) -> string
```

Returns the name of the Joystick device, given by the system. This function can be called before the Joystick is initialized.

get_numaxes

```
Joystick.get_numaxes() -> int
```

Returns the number of available axes on the Joystick.

get_numballs

```
Joystick.get_numballs() -> int
```

Returns the number of available trackballs on the Joystick.

get_numbuttons

```
Joystick.get_numbuttons() -> int
```

Returns the number of available buttons on the Joystick.

get_numhats

```
Joystick.get_numhats() -> int
```

Returns the number of available directional hats on the Joystick.

init

```
Joystick.init() -> None
```

In order to call most members in the Joystick object, the Joystick must be initialized. You can initialzie the Joystick object at anytime, and it is ok to initialize more than once.

quit

```
Joystick.quit() -> None
```

After you are completely finished with a joystick device, you can use this quit() function to free access to the drive. This will be cleaned up automatically when the joystick module is. uninitialized. It is safe to call this function on an uninitialized Joystick.

Movie

The Movie object represents an opened MPEG file. You control playback similar to a Sound object.

Movie objects have a target display Surface. The movie is rendered to this Surface in a background thread. If the Surface is the display surface, and the system supports it, the movie will render into a Hardware YUV overlay plane. If you don't set a display Surface, it will default to the display Surface.

Movies are played back in background threads, so there is very little management needed on the user end. Just load the Movie, set the destination, and Movie.play()

Movies will only playback audio if the pygame.mixer module is not initialized. It is easy to temporarily call pygame.mixer.quit() to disable audio, then create and play your movie. Finally calling pygame.mixer.init() again when finished with the Movie.

NOTE: When disabling the mixer so a movie may play audio, you must disable the audio before calling pygame.movie.Movie or the movie will not realise that it may access the audio. Before reinitialising the mixer, You must remove all references to the movie before calling pygame.mixer.init() or the init will fail, leading to errors when you attempt to use the mixer.

eg. pygame.mixer.quit() movie=pygame.movie.Movie(\"my.mpg\") movie.play() # process events until movie finished here movie.stop() movie=None # if you don't do this bit the init will fail pygame.mixer.init()

get_busy	- query the playback state
get_frame	- query the current frame in the movie
get_length	- query playback time of the movie
get_size	- query the size of the video image
get_time	- query the current time in the movie
has_audio	- query if movie stream has audio
has_video	- query if movie stream has video
pause	- pause/resume movie playback
play	- start movie playback
render_frame	- Render a specfic numbered frame.
rewind	- set playback position to the beginning of the movie
set_display	- change the video output surface
set_volume	- change volume for sound
skip	- skip the movie playback position forward
stop	- stop movie playback

get_busy

```
Movie.get_busy() -> bool
```

Returns true if the movie is currently playing.

get_frame

```
Movie.get_frame() -> int
```

Gets the current video frame number for the movie.

get_length

```
Movie.get_length() -> float
```

Returns the total time (in seconds) of the movie.

get_size

```
Movie.get_size() -> width,height
```

Returns the size of the video image the mpeg provides.

get_time

```
Movie.get_time() -> float
```

Gets the current time (in seconds) for the movie. (currently not working? SMPEG always reports 0)

has_audio

```
Movie.has_audio() -> bool
```

Returns a true value when the Movie object has a valid audio stream.

has_video

```
Movie.has_video() -> bool
```

Returns a true value when the Movie object has a valid video stream.

pause

```
Movie.pause() -> None
```

This will temporarily stop playback of the movie. When called a second time, playback will resume where it left off.

play

```
Movie.play(loops=0) -> None
```

Starts playback of a movie. If audio or video is enabled for the Movie, those outputs will be created.

You can specify an optional argument which will be the number of times the movie loops while playing.

render_frame

```
Movie.render_frame(framenum) -> int
```

Returns the current frame number.

rewind

```
Movie.rewind() -> None
```

Sets the movie playback position to the start of the movie.

set_display

```
Movie.set_display(Surface, [pos]) -> None
```

Set the output surface for the Movie's video. You may also specify a position for the topleft corner of the video. The position defaults to (0,0) if not given.

The position argument can optionally be a rectangle, in which case the video will be stretched to fill the rectangular area.

You may also pass None as the destination Surface, and no video will be rendered for the movie playback.

set_volume

```
Movie.set_volume(val) -> None
```

Set the play volume for this Movie. The volume value is between 0.0 and 1.0.

skip

```
Movie.skip(seconds) -> None
```

Sets the movie playback position ahead by the given amount of seconds. the seconds value is a floating point value

stop

```
Movie.stop() -> None
```

Stops playback of a movie. If sound and video are being rendered, both will be stopped at their current position.

Rect

The rectangle object is a useful object representing a rectangle area. Rectangles are created from the pygame.Rect() function. This routine is also in the locals module, so importing the locals into your namespace allows you to just use Rect().

Rect contains helpful methods, as well as a list of modifiable members: top, bottom, left, right, topleft, topright, bottomleft, bottomright, size, width, height, center, centerx, centery, midleft, midright, midtop, midbottom. When changing thesemembers, the rectangle will be moved to the given assignment. (except when changing the size, width, or height member, which will resize the rectangle from the topleft corner)

The rectstyle arguments used frequently with the Rect object (and elsewhere in pygame) is one of the following things. First, an actual Rect object. Second, a sequence of [xpos, ypos, width, height]. Lastly, a pair of sequences, representing the position and size [[xpos, ypos], [width, height]]. Also, if a method takes a rectstyle argument as its only argument, you can simply pass four arguments representing xpos, ypos, width, height. A rectstyle argument can also be _any_ python object with an attribute named 'rect'.

clamp	- move rectangle inside another
clamp_ip	- moves the rectangle inside another
clip	- rectangle cropped inside another
collidedict	- find overlapping rectangle in a dictionary
collidedictall	- find all overlapping rectangles
collidelist	- find overlapping rectangle
collidelistall	- find all overlapping rectangles
collidepoint	- point inside rectangle
colliderect	- check overlapping rectangles
contains	- check if rectangle fully inside another
inflate	- new rectangle with size changed
inflate_ip	- changes the Rect size
move	- new rectangle with position changed
move_ip	- move the Rect by the given offset
normalize	- corrects negative sizes
union	- makes new rectangle covering both inputs
union_ip	- rectangle covering both input
unionall	- rectangle covering all inputs
unionall_ip	- rectangle covering all inputs

clamp

```
Rect.clamp(rectstyle) -> Rect
```

Returns a new rectangle that is moved to be completely inside the argument rectangle. If the base rectangle is too large for the argument rectangle in an axis, it will be centered on that axis.

clamp_ip

```
Rect.clamp_ip(rectstyle) -> None
```

Moves the Rect to be completely inside the argument rectangle. If the given rectangle is too large for the argument rectangle in an axis, it will be centered on that axis.

clip

```
Rect.clip(rectstyle) -> Rect
```

Returns a new rectangle that is the given rectangle cropped to the inside of the base rectangle. If the two rectangles do not overlap to begin with, you will get a rectangle with 0 size.

collidedict

```
Rect.collidedict(dict if rectstyle keys) -> key/value pair
```

Returns the key/value pair of the first rectangle key in the dict that overlaps the base rectangle. Once an overlap is found, this will stop checking the remaining list. If no overlap is found, it will return None.

Remember python dictionary keys must be immutable, Rects are not immutable, so they cannot directly be, dictionary keys. You can convert the Rect to a tuple with the tuple() builtin command.

collidedictall

```
Rect.collidedictall(rectstyle list) -> key/val list
```

Returns a list of the indexes that contain rectangles overlapping the base rectangle. If no overlap is found, it will return an empty sequence.

Remember python dictionary keys must be immutable, Rects are not immutable, so they cannot directly be, dictionary keys. You can convert the Rect to a tuple with the tuple() builtin command.

collidelist

```
Rect.collidelist(rectstyle list) -> int index
```

Returns the index of the first rectangle in the list to overlap the base rectangle. Once an overlap is found, this will stop checking the remaining list. If no overlap is found, it will return -1.

collidelistall

```
Rect.collidelistall(rectstyle list) -> index list
```

Returns a list of the indexes that contain rectangles overlapping the base rectangle. If no overlap is found, it will return an empty sequence.

collidepoint

```
Rect.collidepoint(x, y) -> bool
```

Returns true if the given point position is inside the rectangle. If a point is on the border, it is counted as inside.

colliderect

```
Rect.colliderect(rectstyle) -> bool
```

Returns true if any area of the two rectangles overlaps.

contains

```
Rect.contains(rectstyle) -> bool
```

Returns true when the given rectangle is entirely inside the base rectangle.

inflate

```
Rect.inflate(x, y) -> Rect
```

Returns a new rectangle which has the sizes changed by the given amounts. The rectangle shrinks and expands around the rectangle's center. Negative values will shrink the rectangle.

inflate_ip

```
Rect.inflate_ip(x, y) -> None
```

Changes the Rect by the given amounts. The rectangle shrinks and expands around the rectangle's center. Negative values will shrink the rectangle.

move

```
Rect.move(x, y) -> Rect
```

Returns a new rectangle which is the base rectangle moved by the given amount.

move_ip

```
Rect.move_ip(x, y) -> None
```

Moves the rectangle which by the given amount.

normalize

```
Rect.normalize() -> None
```

If the rectangle has a a negative size in width or height, this will flip that axis so the sizes are positive, and the rectangle remains in the same place.

union

```
Rect.union(rectstyle) -> Rect
```

Returns a new Rect to completely cover the given input. There may be area inside the new Rect that is not covered by either input.

union_ip

```
Rect.union_ip(rectstyle) -> None
```

Resizes the Rect to completely cover the given input. There may be area inside the new dimensions that is not covered by either input.

unionall

```
Rect.unionall(sequence_of_rectstyles) -> Rect
```

Returns a new rectangle that completely covers all the given inputs. There may be area inside the new rectangle that is not covered by the inputs.

unionall_ip

```
Rect.unionall_ip(sequence_of_rectstyles) -> None
```

Resizes the rectangle to completely cover all the given inputs. There may be area inside the new rectangle that is not covered by the inputs.

Sound

Sound objects represent actual sound data. Sound objects are created from the function pygame.mixer.Sound(). Sound objects can be playing on multiple channels simultaneously. Calling functions like Sound.stop() from the sound objects will effect all channels playing that Sound object.

All sound objects have the same frequency and format as the pygame.mixer module's initialization.

fadeout	- fadeout all channels playing this sound
get_num_channels	- number of channels with sound
get_volume	- query volume for sound
play	- play sound
set_volume	- change volume for sound
stop	- stop all channels playing this sound

fadeout

```
Sound.fadeout(millisec) -> None
```

Fade out all the playing channels playing this sound over the. All channels playing this sound will be stopped after the given milliseconds.

get_num_channels

```
Sound.get_num_channels() -> int
```

Returns the number of channels that have been using this sound. The channels may have already finished, but have not started playing any other sounds.

get_volume

```
Sound.get_volume() -> val
```

Returns the current volume for this sound object. The value is 0.0 to 1.0.

play

```
Sound.play([loops, [maxtime]]) -> Channel
```

Starts playing a song on an available channel. If no channels are available, it will not play and return None. Loops controls how many extra times the sound will play, a negative loop will play indefinitely, it defaults to 0. Maxtime is the number of total milliseconds that the sound will play. It defaults to forever (-1).

Returns a channel object for the channel that is selected to play the sound.

set_volume

```
Sound.set_volume(val) -> None
```

Set the play volume for this sound. This will effect any channels currently playing this sound, along with all subsequent calls to play. The value is 0.0 to 1.0.

stop

```
Sound.stop() -> None
```

This will instantly stop all channels playing this sound.

Sound

Sound objects represent actual sound data. Sound objects are created from the function pygame.mixer.Sound(). Sound objects can be playing on multiple channels simultaneously. Calling functions like Sound.stop() from the sound objects will effect all channels playing that Sound object.

All sound objects have the same frequency and format as the pygame.mixer module's initialization.

fadeout	- fadeout all channels playing this sound
get_num_channels	- number of channels with sound
get_volume	- query volume for sound
play	- play sound
set_volume	- change volume for sound
stop	- stop all channels playing this sound

fadeout

```
Sound.fadeout(millisec) -> None
```

Fade out all the playing channels playing this sound over the. All channels playing this sound will be stopped after the given milliseconds.

get_num_channels

```
Sound.get_num_channels() -> int
```

Returns the number of channels that have been using this sound. The channels may have already finished, but have not started playing any other sounds.

get_volume

```
Sound.get_volume() -> val
```

Returns the current volume for this sound object. The value is 0.0 to 1.0.

play

```
Sound.play([loops, [maxtime]]) -> Channel
```

Starts playing a song on an available channel. If no channels are available, it will not play and return None. Loops controls how many extra times the sound will play, a negative loop will play indefinitely, it defaults to 0. Maxtime is the number of total milliseconds that the sound will play. It defaults to forever (-1).

Returns a channel object for the channel that is selected to play the sound.

set_volume

```
Sound.set_volume(val) -> None
```

Set the play volume for this sound. This will effect any channels currently playing this sound, along with all subsequent calls to play. The value is 0.0 to 1.0.

stop

```
Sound.stop() -> None
```

This will instantly stop all channels playing this sound.

Surface

Surface objects represent a simple memory buffer of pixels. Surface objects can reside in system memory, or in special hardware memory, which can be hardware accelerated. Surfaces that are 8 bits per pixel use a colormap to represent their color values. All Surfaces with higher bits per pixel use a packed pixels to store their color values.

Surfaces can have many extra attributes like alpha planes, colorkeys, source rectangle clipping. These functions mainly effect how the Surface is blitted to other Surfaces. The blit routines will attempt to use hardware acceleration when possible, otherwise will use highly optimized software blitting methods.

There is support for pixel access for the Surfaces. Pixel access on hardware surfaces is slow and not recommended. Pixels can be accessed using the get_at() and set_at() functions. These methods are fine for simple access, but will be considerably slow when doing of pixel work with them. If you plan on doing a lot of pixel level work, it is recommended to use the pygame.surfarray module, which can treat the surfaces like large multidimensional arrays (and it's quite quick).

Any functions that directly access a surface's pixel data will need that surface to be lock()'ed. These functions can lock() and unlock() the surfaces themselves without assistance. But, if a function will be called many times, there will be a lot of overhead for multiple locking and unlocking of the surface. It is best to lock the surface manually before making the function call many times, and then unlocking when you are finished. All functions that need a locked surface will say so in their docs.

Also remember that you will want to leave the surface locked for the shortest amount of time needed.

Here is the quick breakdown of how packed pixels work (don't worry if you don't quite understand this, it is only here for informational purposes, it is not needed). Each colorplane mask can be used to isolate the values for a colorplane from the packed pixel color. Therefore PACKED_COLOR & RED_MASK == REDPLANE. Note that the REDPLANE is not exactly the red color value, but it is the red color value bitwise left shifted a certain amount. The losses and masks can be used to convert back and forth between each colorplane and the actual color for that plane. Here are the final formulas used be map and unmap. PACKED_COLOR = RED>>losses[0]<<shifts[0] | GREEN>>losses[1]<<shifts[1] | BLUE>>losses[2]<<shifts[2] RED = PACKED_COLOR & masks[0] >> shifts[0] << losses[0] GREEN = PACKED_COLOR & masks[1] >> shifts[1] << losses[1] BLUE = PACKED_COLOR & masks[2] >> shifts[2] << losses[2] There is also an alpha channel for some Surfaces.

blit	- copy a one Surface to another.
convert	- new copy of surface with different format
convert_alpha	- new copy of surface with different format and per pixel alpha
fill	- fill areas of a Surface
get_abs_offset	- get absolute offset of subsurface
get_abs_parent	- get the toplevel surface for a subsurface
get_alpha	- query alpha information
get_at	- get a pixel color
get_bitsize	- query size of pixel
get_bytesize	- query size of pixel
get_clip	- query the clipping area
get_colorkey	- query colorkey

get_flags - query the surface flags

get_height - query the surface height

get_locked - check if the surface needs locking

get_losses - get mapping losses for each colorplane

get_masks - get mapping bitmasks for each colorplane

get_offset - get offset of subsurface

get_palette - get the palette

get_palette_at - get a palette entry

get_parent - get a subsurface parent

get_pitch - query the surface pitch

get_rect - get a rectangle covering the entire surface

get_shifts - alphashift

get_size - query the surface size

get_width - query the surface width

lock - locks Surface for pixel access

map_rgb - convert RGB into a mapped color

mustlock - check if the surface needs locking

set_alpha - change alpha information

set_at - set pixel at given position

set_clip - assign destination clipping rectangle

set_colorkey - change colorkey information

set_palette - set the palette

set_palette_at - set a palette entry

subsurface - create a new surface that shares pixel data

unlock - locks Surface for pixel access

unmap_rgb - convert mapped color into RGB

blit

```
Surface.blit(source, destpos, [sourcerect]) -> Rect
```

The blitting will copy pixels from the source. It will respect any special modes like colorkeying and alpha. If hardware support is available, it will be used. The given source is the Surface to copy from. The destoffset is a 2-number-sequence that specifies where on the destination Surface the blit happens (see below). When sourcerect isn't supplied, the blit will copy the entire source surface. If you would like to copy only a portion of the source, use the sourcerect argument to control what area is copied.

The blit is subject to be clipped by the active clipping rectangle. The return value contains the actual area blitted.

As a shortcut, the destination position can be passed as a rectangle. If a rectangle is given, the blit will use the topleft corner of the rectangle as the blit destination position. The rectangle sizes will be ignored.

Blitting surfaces with pixel alphas onto an 8bit destination will not use the surface alpha values.

convert

```
Surface.convert([src_surface] OR depth, [flags] OR masks) -> Surface
```

Creates a new copy of the surface with the desired pixel format. Surfaces with the same pixel format will blit much faster than those with mixed formats. The pixel format of the new surface will match the format given as the argument. If no surface is given, the new surface will have the same pixel format as the current display.

convert() will also accept bitsize or mask arguments like the Surface() constructor function. Either pass an integer bitsize or a sequence of color masks to specify the format of surface you would like to convert to. When used this way you may also pass an optional flags argument (whew).

convert_alpha

```
Surface.convert_alpha([src_surface]) -> Surface
```

Creates a new copy of the surface with the desired pixel format. The new surface will be in a format suited for quick blitting to the given format with per pixel alpha. If no surface is given, the new surface will be optimized for blittint to the current display.

Unlike the convert() method, the pixel format for the new image will not be exactly the same as the requested source, but it will be optimized for fast alpha blitting to the destination.

fill

```
Surface.fill(color, [rectstyle])) -> Rect
```

Fills the specified area of the Surface with the mapped color value. If no destination rectangle is supplied, it will fill the entire Surface.

The color argument can be a RGBA sequence or a mapped color integer.

The fill is subject to be clipped by the active clipping rectangle. The return value contains the actual area filled.

get_abs_offset

```
Surface.get_abs_offset() -> x, y
```

Returns the absolute X and Y position a subsurface is positioned inside its top level parent. Will return 0,0 for surfaces that are not a subsurface.

get_abs_parent

```
Surface.get_abs_parent() -> Surface
```

Returns the top level Surface for this subsurface. If this is not a subsurface it will return a reference to itself. You will always get a valid surface from this method.

get_alpha

```
Surface.get_alpha() -> alpha
```

Returns the current alpha value for the Surface. If transparency is disabled for the Surface, it returns None.

get_at

```
Surface.get_at(position) -> RGBA
```

Returns the RGB color values at a given pixel. If the Surface has no per-pixel alpha, the alpha will be 255 (opaque).

This function will need to temporarily lock the surface.

get_bitsize

```
Surface.get_bitsize() -> int
```

Returns the number of bits used to represent each pixel. This value may not exactly fill the number of bytes used per pixel. For example a 15 bit Surface still requires a full 2 bytes.

get_bytesize

```
Surface.get_bytesize() -> int
```

Returns the number of bytes used to store each pixel.

get_clip

```
Surface.get_clip() -> rect
```

Returns the current destination clipping area being used by the Surface. If the clipping area is not set, it will return a rectangle containing the full Surface area.

get_colorkey

```
Surface.get_colorkey() -> RGBA
```

Returns the current mapped color value being used for colorkeying. If colorkeying is not enabled for this surface, it returns None

get_flags

```
Surface.get_flags() -> flags
```

Returns the current state flags for the surface.

get_height

```
Surface.get_height() -> height
```

Returns the height of the Surface.

get_locked

```
Surface.get_locked() -> bool
```

Returns true if the surface is currently locked.

get_losses

```
Surface.get_losses() -> redloss, greenloss, blueloss, alphaloss
```

Returns the bitloss for each color plane. The loss is the number of bits removed for each colorplane from a full 8 bits of resolution. A value of 8 usually indicates that colorplane is not used (like the alpha)

get_masks

```
Surface.get_masks() -> redmask, greenmask, bluemask, alphamask
```

Returns the bitmasks for each color plane. The bitmask is used to isolate each colorplane value from a mapped color value. A value of zero means that colorplane is not used (like alpha)

get_offset

```
Surface.get_offset() -> x, y
```

Returns the X and Y position a subsurface is positioned inside its parent. Will return 0,0 for surfaces that are not a subsurface.

get_palette

```
Surface.get_palette() -> [[r, g, b], ...]
```

This will return the an array of all the color indexes in the Surface's palette.

get_palette_at

```
Surface.get_palette_at(index) -> r, g, b
```

This will retreive an individual color entry from the Surface's palette.

get_parent

```
Surface.get_parent() -> Surface
```

Returns the Surface that is a parent of this subsurface. Will return None if this is not a subsurface.

get_pitch

```
Surface.get_pitch() -> pitch
```

The surface pitch is the number of bytes used in each scanline. This function should rarely needed, mainly for any special-case debugging.

get_rect

```
Surface.get_rect() -> rect
```

Returns a new rectangle covering the entire surface. This rectangle will always start at 0, 0 with a width. and height the same size as the image.

get_shifts

```
Surface.get_shifts() -> redshift, greenshift, blueshift,
```

Returns the bitshifts used for each color plane. The shift is determine how many bits left-shifted a colorplane value is in a mapped color value.

get_size

```
Surface.get_size() -> x, y
```

Returns the width and height of the Surface.

get_width

```
Surface.get_width() -> width
```

Returns the width of the Surface.

lock

```
Surface.lock() -> None
```

On accelerated surfaces, it is usually required to lock the surface before you can access the pixel values. To be safe, it is always a good idea to lock the surface before entering a block of code that changes or accesses the pixel values. The surface must not be locked when performing other pygame functions on it like fill and blit.

You can doublecheck to really make sure a lock is needed by calling the mustlock() member. This should not be needed, since it is usually recommended to lock anyways and work with all surface types. If the surface does not need to be locked, the operation will return quickly with minute overhead.

On some platforms a necessary lock can shut off some parts of the system. This is not a problem unless you leave surfaces locked for long periouds of time. Only keep the surface locked when you need the pixel access. At the same time, it is not a good too repeatedly lock and unlock the surface inside tight loops. It is fine to leave the surface locked while needed, just don't be lazy.

map_rgb

```
Surface.map_rgb(RGBA) -> int
```

Uses the Surface format to convert RGBA into a mapped color value.

This function is not as needed as normal C code using SDL. The pygame functions do not used mapped colors, so there is no need to map them.

mustlock

```
Surface.mustlock() -> bool
```

Returns true if the surface really does need locking to gain pixel access. Usually the overhead of checking before locking outweight the overhead of just locking any surface before access.

set_alpha

```
Surface.set_alpha([alpha, [flags]]) -> None
```

Set the overall transparency for the surface. If no alpha is passed, alpha blending is disabled for the surface. An alpha of 0 is fully transparent, an alpha of 255 is fully opaque. If no arguments or None is passed, this will disable the surface alpha.

If your surface has a pixel alpha channel, it will override the overall surface transparency. You'll need to change the actual pixel transparency to make changes.

If your image also has pixel alpha values, will be used repeatedly, you will probably want to pass the RLEACCEL flag to the call. This will take a short time to compile your surface, and increase the blitting speed.

set_at

```
Surface.set_at(position, RGBA) -> None
```

Assigns color to the image at the give position. Color can be a RGBA sequence or a mapped color integer.

In some situations just using the fill() function with a one-pixel sized rectangle will be quicker. Also the fill function does not require the surface to be locked.

This function will need to temporarily lock the surface.

set_clip

```
Surface.set_clip([rectstyle]) -> None
```

Assigns the destination clipping rectangle for the Surface. When blit or fill operations are performed on the Surface, they are restricted to the inside of the clipping rectangle. If no rectangle is passed, the clipping region is set to the entire Surface area. The rectangle you pass will be clipped to the area of the Surface.

set_colorkey

```
Surface.set_colorkey([color, [flags]]) -> None
```

Set the colorkey for the surface by passing a mapped color value as the color argument. If no arguments or None is passed, colorkeying will be disabled for this surface.

The color argument can be either a RGBA sequence or a mapped integer.

If your image is nonchanging and will be used repeatedly, you will probably want to pass the RLEACCEL flag to the call. This will take a short time to compile your surface, and increase the blitting speed.

set_palette

```
Surface.set_palette([[r, g, b], ...]) -> None
```

This will replace the entire palette with color information you provide.

You can pass an incomplete list of RGB values, and this will only change the first colors in the palette.

set_palette_at

```
Surface.set_palette_at(index, [r, g, b]) -> None
```

This function sets the palette color at a specific entry.

subsurface

```
Surface.subsurface(rectstyle) -> Surface
```

Creates a new surface that shares pixel data of the given surface. Note that only the pixel data is shared. Things like clipping rectangles and colorkeys will be unique for the new surface.

The new subsurface will inherit the palette, colorkey, and surface alpha values from the base image.

You should not use the RLEACCEL flag for parent surfaces of subsurfaces, for the most part it will work, but it will cause a lot of extra work, every time you change the subsurface, you must decode and recode the RLEACCEL data for the parent surface.

As for using RLEACCEL with the subsurfaces, that will work as you'd expect, but changes the the parent Surface will not always take effect in the subsurface.

unlock

```
Surface.unlock() -> None
```

After a surface has been locked, you will need to unlock it when you are done.

You can doublecheck to really make sure a lock is needed by calling the mustlock() member. This should not be needed, since it is usually recommended to lock anyways and work with all surface types. If the surface does not need to be locked, the operation will return quickly with minute overhead.

unmap_rgb

```
Surface.unmap_rgb(color) -> RGBA
```

This function returns the RGBA components for a mapped color value. If Surface has no per-pixel alpha, alpha will be 255 (opaque).

This function is not as needed as normal C code using SDL. The pygame functions do not used mapped colors, so there is no need to unmap them.

pygame.cursors

Set of cursor resources available for use. These cursors come in a sequence of values that are needed as the arguments for pygame.mouse.set_cursor(). to dereference the sequence in place and create the cursor in one step, call like this; pygame.mouse.set_cursor(*pygame.cursors.arrow).

Here is a list of available cursors; arrow, diamond, ball, broken_x, tri_left, tri_right

There is also a sample string cursor named 'thickarrow_strings'. The compile() function can convert these string cursors into cursor byte data.

compile - compile cursor strings into cursor data

load_xbm - reads a pair of XBM files into set_cursor arguments

compile

```
pygame.cursors.compile(strings, black, white) -> data, mask
```

This takes a set of strings with equal length and computes the binary data for that cursor. The string widths must be divisible by 8.

The black and white arguments are single letter strings that tells which characters will represent black pixels, and which characters represent white pixels. All other characters are considered clear.

This returns a tuple containing the cursor data and cursor mask data. Both these arguments are used when setting a cursor with pygame.mouse.set_cursor().

load_xbm

```
pygame.cursors.load_xbm(cursorfile, maskfile) -> cursor_args
```

Arguments can either be filenames or filelike objects with the readlines method. Not largely tested, but should work with typical XBM files.

pygame.sprite

This module contains a base class for sprite objects. Also several different group classes you can use to store and identify the sprites. Some of the groups can be used to draw the sprites they contain. Lastly there are a handful of collision detection functions to help you quickly find intersecting sprites in a group.

The way the groups are designed, it is very efficient at adding and removing sprites from groups. This makes the groups a perfect use for cataloging or tagging different sprites. instead of keeping an identifier or type as a member of a sprite class, just store the sprite in a different set of groups. this ends up being a much better way to loop through, find, and effect different sprites. It is also a very quick to test if a sprite is contained in a given group.

You can manage the relationship between groups and sprites from both the groups and the actual sprite classes. Both have add() and remove() functions that let you add sprites to groups and groups to sprites. Both have initializing functions that can accept a list of containers or sprites.

The methods to add and remove sprites from groups are smart enough to not delete sprites that aren't already part of a group, and not add sprites to a group if it already exists. You may also pass a sequence of sprites or groups to these functions and each one will be used.

The design of the sprites and groups is very flexible. There's no need to inherit from the provided classes, you can use any object you want for the sprites, as long as it contains "add_internal" and "remove_internal" methods, which are called by the groups when they remove and add sprites. The same is true for containers. A container can be any python object that has "add_internal" and "remove_internal" methods that the sprites call when they want add and remove themselves from containers. The containers must also have a member named "_spritegroup", which can be set to any dummy value.

w

Sprite - **(class)** - the base class for your visible game objects.

Sprite.add - add a sprite to container

Sprite.alive - ask the life of a sprite

Sprite.groups - list used sprite containers

Sprite.kill - end life of sprite, remove from all groups

Sprite.remove - remove a sprite from container

groupcollide - collision detection between group and group

spritecollide - collision detection between sprite and group

spritecollideany - finds any sprites that collide

Group

```
pygame.sprite.Group(sprite=())
```

the Group class is a container for sprites This is the base sprite group class. It does everything needed to behave as a normal group. You can easily inherit a new group class from this if you want to add more features.

You can initialize a group by passing it a sprite or sequence of sprites to be contained.

Group.add

```
pygame.sprite.Group.add(sprite)
```

Add a sprite or sequence of sprites to a group.

Group.copy

```
pygame.sprite.Group.copy() -> Group
```

Returns a copy of the group that is the same class type, and has the same contained sprites.

Group.empty

```
pygame.sprite.Group.empty()
```

Removes all the sprites from the group.

Group.has

```
pygame.sprite.Group.has(sprite) -> bool
```

Returns true if the given sprite or sprites are contained in the group

Group.remove

```
pygame.sprite.Group.remove(sprite)
```

Remove a sprite or sequence of sprites from a group.

Group.sprites

```
pygame.sprite.Group.sprites() -> iterator
```

Returns an object that can be looped over with a 'for' loop. (For now it is always a list, but newer version of python could return different objects, like iterators.)

Group.update

```
pygame.sprite.Group.update(...)
```

calls the update method for all sprites in the group. passes all arguments are to the Sprite update function.

GroupSingle

```
pygame.sprite.GroupSingle()
```

a group container that holds a single most recent item This class works just like a regular group, but it only keeps a single sprite in the group. Whatever sprite has been added to the group last, will be the only sprite in the group.

RenderClear

```
pygame.sprite.RenderClear()
```

a group container that can draw and clear its sprites The RenderClear group is just like a normal group, but it can draw and clear the sprites. Any sprites used in this group must contain member elements named "image" and "rect". These are a pygame Surface and Rect, which are passed to a blit call.

RenderClear.clear

```
pygame.sprite.RenderClear.clear(surface, bgd)
```

Clears the area of all drawn sprites. the bgd argument should be Surface which is the same dimensions as the surface. The bgd can also be a function which gets called with the passed surface and the area to be cleared.

RenderClear.draw

```
pygame.sprite.RenderClear.draw(surface)
```

Draws all the sprites onto the given surface.

RenderPlain

```
pygame.sprite.RenderPlain(sprite=())
```

a sprite group that can draw all its sprites The RenderPlain group is just like a normal group, it just adds a "draw" method. Any sprites used with this group to draw must contain two member elements named "image" and "rect". These are a pygame Surface and Rect object that are passed to blit.

You can initialize a group by passing it a sprite or sequence of sprites to be contained.

RenderPlain.draw

```
pygame.sprite.RenderPlain.draw(surface)
```

Draws all the sprites onto the given surface.

RenderUpdates

```
pygame.sprite.RenderUpdates()
```

a sprite group that can draw and clear with update rectangles The RenderUpdates is derived from the RenderClear group and keeps track of all the areas drawn and cleared. It also smartly handles overlapping areas between where a sprite was drawn and cleared when generating the update rectangles.

RenderUpdates.draw

```
pygame.sprite.RenderUpdates.draw(surface)
```

Draws all the sprites onto the given surface. It returns a list of rectangles, which should be passed to pygame.display.update()

Sprite

```
pygame.sprite.Sprite(group=())
```

the base class for your visible game objects. The sprite class is meant to be used as a base class for the objects in your game. It just provides functions to maintain itself in different groups. A sprite is considered 'alive' as long as it is a member of one or more groups. The kill() method simply removes this sprite from all groups.

You can initialize a sprite by passing it a group or sequence of groups to be contained in.

Sprite.add

```
pygame.sprite.Sprite.add(group)
```

Add the sprite to a group or sequence of groups.

Sprite.alive

```
pygame.sprite.Sprite.alive() -> bool
```

Returns true if this sprite is a member of any groups.

Sprite.groups

```
pygame.sprite.Sprite.groups() -> list
```

Returns a list of all the groups that contain this sprite.

Sprite.kill

```
pygame.sprite.Sprite.kill()
```

Removes the sprite from all the groups that contain it. The sprite is still fine after calling this kill() so you could use it to remove a sprite from all groups, and then add it to some other groups.

Sprite.remove

```
pygame.sprite.Sprite.remove(group)
```

Remove the sprite from a group or sequence of groups.

groupcollide

```
pygame.sprite.groupcollide(groupa, groupb, dokilla, dokillb) -> dict
```

given two groups, this will find the intersections between all sprites in each group. it returns a dictionary of all sprites in the first group that collide. the value for each item in the dictionary is a list of the sprites in the second group it collides with. the two dokill arguments control if the sprites from either group will be automatically removed from all groups.

spritecollide

```
pygame.sprite.spritecollide(sprite, group, dokill) -> list
```

given a sprite and a group of sprites, this will return a list of all the sprites that intersect the given sprite. all sprites must have a "rect" value, which is a rectangle of the sprite area. if the dokill argument is true, the sprites that do collide will be automatically removed from all groups.

spritecollideany

```
pygame.sprite.spritecollideany(sprite, group) -> sprite
```

given a sprite and a group of sprites, this will return return any single sprite that collides with with the given sprite. If there are no collisions this returns None. if you don't need all the features of the spritecollide function, this function will be a bit quicker. all sprites must have a "rect" value, which is a rectangle of the sprite area.