

Pygame Documentation

version 2.0.1.dev1

Pygame Developers

December 14, 2020

Contents

Pygame Front Page	1
<code>pygame.BufferProxy</code>	1
<code>pygame.camera</code>	2
<code>pygame.cdrom</code>	4
<code>pygame.Color</code>	7
<code>pygame.cursors</code>	9
<code>pygame.display</code>	12
<code>pygame.draw</code>	20
<code>pygame.event</code>	29
<code>pygame.examples</code>	34
<code>pygame.fastevent</code>	38
<code>pygame.font</code>	39
<code>pygame.freetype</code>	43
<code>pygame.gfxdraw</code>	51
<code>pygame.image</code>	60
<code>pygame.joystick</code>	63
<code>pygame.key</code>	72
<code>pygame.locals</code>	77
<code>pygame.mask</code>	77
<code>pygame.math</code>	85
<code>pygame.midi</code>	93
<code>pygame.mixer</code>	98
<code>pygame.mouse</code>	104
<code>pygame.mixer.music</code>	106
<code>pygame.Overlay</code>	109
<code>pygame.PixelArray</code>	109
<code>pygame.pixelcopy</code>	112
<code>pygame</code>	113
<code>pygame.version</code>	115
<code>pygame.Rect</code>	118
<code>pygame.scrap</code>	122
<code>pygame.sndarray</code>	125
<code>pygame.sprite</code>	126
<code>pygame.Surface</code>	135
<code>pygame.surfarray</code>	144
<code>pygame.tests</code>	146
<code>pygame.time</code>	147
<code>pygame._sdl2.touch</code>	148
<code>pygame.transform</code>	149
Pygame Tutorials - Camera Module Introduction	152

Camera Module Introduction	153
Import and Init	153
Capturing a Single Image	153
Listing Connected Cameras	153
Using Camera Controls	153
Capturing a Live Stream	153
Basic Computer Vision	154
Colorspaces	154
Thresholding	155
Using the Mask Module	156
Pygame Tutorials - Line By Line Chimp Example	157
Line By Line Chimp	157
pygame/examples/chimp.py	157
Introduction	160
Import Modules	160
Loading Resources	161
Game Object Classes	162
Initialize Everything	164
Create The Background	164
Put Text On The Background, Centered	164
Display The Background While Setup Finishes	165
Prepare Game Object	165
Main Loop	165
Handle All Input Events	165
Update the Sprites	166
Draw The Entire Scene	166
Game Over	166
Pygame Tutorials - Setting Display Modes	166
Setting Display Modes	166
Introduction	167
Setting Basics	167
How to Decide	167
Functions	167
Examples	168
Pygame Tutorials - Import and Initialize	168
Import and Initialize	168
Import	169
Init	169
Quit	169
Making Games With Pygame	169
Making Games With Pygame	169
Revision: Pygame fundamentals	169

2. Revision: Pygame fundamentals	169
2.1. The basic Pygame game	169
2.2. Basic Pygame objects	170
2.3. Blitting	171
2.4. The event loop	171
2.5. Ta-da!	171
Kicking things off	171
3. Kicking things off	171
3.1. The first lines, and loading modules	171
3.2. Resource handling functions	172
Game object classes	173
4. Game object classes	173
4.1. A simple ball class	173
4.1.1. Diversion 1: Sprites	174
4.1.2. Diversion 2: Vector physics	174
User-controllable objects	175
5. User-controllable objects	175
5.1. A simple bat class	175
5.1.1. Diversion 3: Pygame events	176
Putting it all together	177
6. Putting it all together	177
6.1. Let the ball hit sides	177
6.2. Let the ball hit bats	177
6.3. The Finished product	178
Table of Contents	182
1. Introduction	182
1.1. A note on coding styles	183
Pygame Tutorials - Help! How Do I Move An Image?	183
Help! How Do I Move An Image?	183
Just Pixels On The Screen	183
Let's Go Back A Step	184
Making The Hero Move	184
Creating A Map	185
Making The Hero Move (Take 2)	185
Definition: "blit"	185
Going From The List To The Screen	186
Screen Coordinates	186
Changing The Background	186
Smooth Movement	186
So, What Next?	187
First, The Mystery Functions	187
Handling Some Input	187

Moving Multiple Images	188
Putting It All Together	188
You Are On Your Own From Here	189
Pygame Intro	189
Python Pygame Introduction	189
HISTORY	189
TASTE	189
PYTHON AND GAMING	191
CLOSING	192
Pygame Modules Overview	192
Pygame Tutorials - Sprite Module Introduction	192
Sprite Module Introduction	192
History Lesson	193
The Classes	193
The Sprite Class	193
The Group Class	193
Mixing Them Together	194
The Many Group Types	194
The Rendering Groups	195
Collision Detection	195
Common Problems	196
Extending Your Own Classes (<i>Advanced</i>)	196
Pygame Tutorials - Surfarray Introduction	197
Surfarray Introduction	197
Introduction	197
Numeric Python	197
Import Surfarray	199
Surfarray Introduction	199
Examples	200
Surface Locking	202
Transparency	202
Other Surfarray Functions	202
More Advanced NumPy	203
Graduation	203
pygame/examples/chimp.py	203
Newbie Guide to Pygame	206
A Newbie Guide to pygame	207
Get comfortable working in Python.	207
Recognize which parts of pygame you really need.	207
Know what a surface is.	207
Use surface.convert().	207
Dirty rect animation.	208

There is NO rule six.	209
Hardware surfaces are more trouble than they're worth.	209
Don't get distracted by side issues.	209
Rects are your friends.	209
Don't bother with pixel-perfect collision detection.	210
Managing the event subsystem.	210
Colorkey vs. Alpha.	211
Do things the pythony way.	211
Revision: Pygame fundamentals	211
2. Revision: Pygame fundamentals	211
2.1. The basic Pygame game	211
2.2. Basic Pygame objects	212
2.3. Blitting	212
2.4. The event loop	213
2.5. Ta-da!	213
Kicking things off	213
3. Kicking things off	213
3.1. The first lines, and loading modules	213
3.2. Resource handling functions	214
Game object classes	214
4. Game object classes	214
4.1. A simple ball class	215
4.1.1. Diversion 1: Sprites	216
4.1.2. Diversion 2: Vector physics	216
User-controllable objects	217
5. User-controllable objects	217
5.1. A simple bat class	217
5.1.1. Diversion 3: Pygame events	218
Putting it all together	218
6. Putting it all together	218
6.1. Let the ball hit sides	219
6.2. Let the ball hit bats	219
6.3. The Finished product	220
pygame C API	224
Slots and c_api - Making functions and data available from other modules	224
High level API exported by pygame.base	224
src_c/base.c	224
Class BufferProxy API exported by pygame.bufferproxy	226
src_c/bufferproxy.c	226
API exported by pygame.cdrom	226
src_c/cdrom.c	226
Class Color API exported by pygame.color	226

src_c/color.c	226
API exported by pygame.display	227
src_c/display.c	227
API exported by pygame.event	227
src_c/event.c	227
API exported by pygame._freetype	228
src_c/_freetype.c	228
API exported by pygame.mixer	228
src_c/mixer.c	228
Class Rect API exported by pygame.rect	229
src_c/rect.c	229
API exported by pygame.rwobject	230
src_c/rwobject.c	230
Class Surface API exported by pygame.surface	230
src_c/surface.c	230
API exported by pygame.surflock	231
src_c/surflock.c	231
API exported by pygame.version	231
src_py/version.py	232
File Path Function Arguments	232
Tutorials	232
Reference	233
Index	235
Python Module Index	247

Pygame Front Page

pygame.BufferProxy

pygame.BufferProxy

pygame object to export a surface buffer through an array protocol

BufferProxy(<parent>) -> BufferProxy

BufferProxy is a pygame support type, designed as the return value of the **Surface.get_buffer()** and **Surface.get_view()** methods. For all Python versions a **BufferProxy** object exports a C struct and Python level array interface on behalf of its parent object's buffer. For CPython 2.6 and later a new buffer interface is also exported. In pygame, **BufferProxy** is key to implementing the **pygame.surfarray** module.

BufferProxy instances can be created directly from Python code, either for a parent that exports an interface, or from a Python dict describing an object's buffer layout. The dict entries are based on the Python level array interface mapping. The following keys are recognized:

"shape" : *tuple*

The length of each array dimension as a tuple of integers. The length of the tuple is the number of dimensions in the array.

"typestr" : *string*

The array element type as a length 3 string. The first character gives byteorder, '<' for little-endian, '>' for big-endian, and '|' for not applicable. The second character is the element type, 'i' for signed integer, 'u' for unsigned integer, 'f' for floating point, and 'V' for a chunk of bytes. The third character gives the bytesize of the element, from '1' to '9' bytes. So, for example, "<u4" is an unsigned 4 byte little-endian integer, such as a 32 bit pixel on a PC, while "|V3" would represent a 24 bit pixel, which has no integer equivalent.

"data" : *tuple*

The physical buffer start address and a read-only flag as a length 2 tuple. The address is an integer value, while the read-only flag is a bool—False for writable, True for read-only.

"strides" : *tuple : (optional)*

Array stride information as a tuple of integers. It is required only of non C-contiguous arrays. The tuple length must match that of "shape".

"parent" : *object : (optional)*

The exporting object. It can be used to keep the parent object alive while its buffer is visible.

"before" : *callable : (optional)*

Callback invoked when the **BufferProxy** instance exports the buffer. The callback is given one argument, the "parent" object if given, otherwise None. The callback is useful for setting a lock on the parent.

"after" : *callable : (optional)*

Callback invoked when an exported buffer is released. The callback is passed on argument, the "parent" object if given, otherwise None. The callback is useful for releasing a lock on the parent.

The BufferProxy class supports subclassing, instance variables, and weak references.

New in version 1.8.0: *New in version 1.8.0.*

parent

Return wrapped exporting object.

parent -> Surface

parent -> <parent>

The **Surface** which returned the **BufferProxy** object or the object passed to a **BufferProxy** call.

length

The size, in bytes, of the exported buffer.

length -> int

The number of valid bytes of data exported. For discontinuous data, that is data which is not a single block of memory, the bytes within the gaps are excluded from the count. This property is equivalent to the `Py_buffer` C struct `len` field.

raw

A copy of the exported buffer as a single block of bytes.

raw -> bytes

The buffer data as a `str/bytes` object. Any gaps in the exported data are removed.

write ()

Write raw bytes to object buffer.

write(buffer, offset=0)

Overwrite bytes in the parent object's data. The data must be C or F contiguous, otherwise a `ValueError` is raised. Argument *buffer* is a `str/bytes` object. An optional offset gives a start position, in bytes, within the buffer where overwriting begins. If the offset is negative or greater than or equal to the buffer proxy's `length` value, an `IndexException` is raised. If `len(buffer) > proxy.length + offset`, a `ValueError` is raised.

pygame.camera

pygame module for camera use

Pygame currently supports only Linux and v4l2 cameras.

EXPERIMENTAL!: This API may change or disappear in later pygame releases. If you use this, your code will very likely break with the next pygame release.

The Bayer to RGB function is based on:

```
Sonix SN9C101 based webcam basic I/F routines
Copyright (C) 2004 Takafumi Mizuno <taka-qce@ls-a.jp>
Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.
THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS ``AS IS'' AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

New in pygame 1.9.0.

`pygame.camera.colorspace ()`

Surface colorspace conversion

`colorspace(Surface, format, DestSurface = None) -> Surface`

Allows for conversion from "RGB" to a destination colorspace of "HSV" or "YUV". The source and destination surfaces must be the same size and pixel depth. This is useful for computer vision on devices with limited processing power. Capture as small of an image as possible, `transform.scale()` it even smaller, and then convert the colorspace to YUV or HSV before doing any processing on it.

`pygame.camera.list_cameras()`

returns a list of available cameras

`list_cameras()` -> [cameras]

Checks the computer for available cameras and returns a list of strings of camera names, ready to be fed into `pygame.camera.Camera`.

`pygame.camera.Camera`

load a camera

`Camera(device, (width, height), format)` -> Camera

Loads a v4l2 camera. The device is typically something like `"/dev/video0"`. Default width and height are 640 by 480. Format is the desired colorspace of the output. This is useful for computer vision purposes. The default is RGB. The following are supported:

- RGB - Red, Green, Blue
- YUV - Luma, Blue Chrominance, Red Chrominance
- HSV - Hue, Saturation, Value

`start()`

opens, initializes, and starts capturing

`start()` -> None

Opens the camera device, attempts to initialize it, and begins recording images to a buffer. The camera must be started before any of the below functions can be used.

`stop()`

stops, uninitializes, and closes the camera

`stop()` -> None

Stops recording, uninitializes the camera, and closes it. Once a camera is stopped, the below functions cannot be used until it is started again.

`get_controls()`

gets current values of user controls

`get_controls()` -> (hflip = bool, vflip = bool, brightness)

If the camera supports it, `get_controls` will return the current settings for horizontal and vertical image flip as bools and brightness as an int. If unsupported, it will return the default values of (0, 0, 0). Note that the return values here may be different than those returned by `set_controls`, though these are more likely to be correct.

`set_controls()`

changes camera settings if supported by the camera

`set_controls(hflip = bool, vflip = bool, brightness)` -> (hflip = bool, vflip = bool, brightness)

Allows you to change camera settings if the camera supports it. The return values will be the input values if the camera claims it succeeded or the values previously in use if not. Each argument is optional, and the desired one can be chosen by supplying the keyword, like `hflip`. Note that the actual settings being used by the camera may not be the same as those returned by `set_controls`.

`get_size()`

returns the dimensions of the images being recorded

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

Returns the current dimensions of the images being captured by the camera. This will return the actual size, which may be different than the one specified during initialization if the camera did not support that size.

`query_image()`

checks if a frame is ready

`query_image()` -> bool

If an image is ready to get, it returns true. Otherwise it returns false. Note that some webcams will always return False and will only queue a frame when called with a blocking function like `get_image()`. This is useful to separate the framerate of the game from that of the camera without having to use threading.

get_image ()

captures an image as a Surface

get_image(Surface = None) -> Surface

Pulls an image off of the buffer as an RGB Surface. It can optionally reuse an existing Surface to save time. The bit-depth of the surface is either 24 bits or the same as the optionally supplied Surface.

get_raw ()

returns an unmodified image as a string

get_raw() -> string

Gets an image from a camera as a string in the native pixelformat of the camera. Useful for integration with other libraries.

pygame.cdrom

pygame module for audio cdrom control

The cdrom module manages the CD and DVD drives on a computer. It can also control the playback of audio CDs. This module needs to be initialized before it can do anything. Each CD object you create represents a cdrom drive and must also be initialized individually before it can do most things.

pygame.cdrom.init ()

initialize the cdrom module

init() -> None

Initialize the cdrom module. This will scan the system for all CD devices. The module must be initialized before any other functions will work. This automatically happens when you call `pygame.init()`.

It is safe to call this function more than once.

pygame.cdrom.quit ()

uninitialize the cdrom module

quit() -> None

Uninitialize the cdrom module. After you call this any existing CD objects will no longer work.

It is safe to call this function more than once.

pygame.cdrom.get_init ()

true if the cdrom module is initialized

get_init() -> bool

Test if the cdrom module is initialized or not. This is different than the `CD.init()` since each drive must also be initialized individually.

pygame.cdrom.get_count ()

number of cd drives on the system

get_count() -> count

Return the number of cd drives on the system. When you create CD objects you need to pass an integer id that must be lower than this count. The count will be 0 if there are no drives on the system.

pygame.cdrom.CD

class to manage a cdrom drive

CD(id) -> CD

You can create a CD object for each cdrom on the system. Use `pygame.cdrom.get_count()` to determine how many drives actually exist. The id argument is an integer of the drive, starting at zero.

The CD object is not initialized, you can only call `CD.get_id()` and `CD.get_name()` on an uninitialized drive.

It is safe to create multiple CD objects for the same drive, they will all cooperate normally.

init ()

initialize a cdrom drive for use

init() -> None

Initialize the cdrom drive for use. The drive must be initialized for most CD methods to work. Even if the rest of pygame has been initialized.

There may be a brief pause while the drive is initialized. Avoid `CD.init()` if the program should not stop for a second or two.

quit()

uninitialize a cdrom drive for use

quit() -> None

Uninitialize a drive for use. Call this when your program will not be accessing the drive for awhile.

get_init()

true if this cd device initialized

get_init() -> bool

Test if this CDROM device is initialized. This is different than the `pygame.cdrom.init()` since each drive must also be initialized individually.

play()

start playing audio

play(track, start=None, end=None) -> None

Playback audio from an audio cdrom in the drive. Besides the track number argument, you can also pass a starting and ending time for playback. The start and end time are in seconds, and can limit the section of an audio track played.

If you pass a start time but no end, the audio will play to the end of the track. If you pass a start time and 'None' for the end time, the audio will play to the end of the entire disc.

See the `CD.get_numtracks()` and `CD.get_track_audio()` to find tracks to playback.

Note, track 0 is the first track on the CD. Track numbers start at zero.

stop()

stop audio playback

stop() -> None

Stops playback of audio from the cdrom. This will also lose the current playback position. This method does nothing if the drive isn't already playing audio.

pause()

temporarily stop audio playback

pause() -> None

Temporarily stop audio playback on the CD. The playback can be resumed at the same point with the `CD.resume()` method. If the CD is not playing this method does nothing.

Note, track 0 is the first track on the CD. Track numbers start at zero.

resume()

unpause audio playback

resume() -> None

Unpause a paused CD. If the CD is not paused or already playing, this method does nothing.

eject()

eject or open the cdrom drive

eject() -> None

This will open the cdrom drive and eject the cdrom. If the drive is playing or paused it will be stopped.

get_id()

the index of the cdrom drive

get_id() -> id

Returns the integer id that was used to create the CD instance. This method can work on an uninitialized CD.

get_name()

the system name of the cdrom drive

get_name() -> name

Return the string name of the drive. This is the system name used to represent the drive. It is often the drive letter or device name. This method can work on an uninitialized CD.

get_busy ()

true if the drive is playing audio

get_busy() -> bool

Returns True if the drive busy playing back audio.

get_paused ()

true if the drive is paused

get_paused() -> bool

Returns True if the drive is currently paused.

get_current ()

the current audio playback position

get_current() -> track, seconds

Returns both the current track and time of that track. This method works when the drive is either playing or paused.

Note, track 0 is the first track on the CD. Track numbers start at zero.

get_empty ()

False if a cdrom is in the drive

get_empty() -> bool

Return False if there is a cdrom currently in the drive. If the drive is empty this will return True.

get_numtracks ()

the number of tracks on the cdrom

get_numtracks() -> count

Return the number of tracks on the cdrom in the drive. This will return zero if the drive is empty or has no tracks.

get_track_audio ()

true if the cdrom track has audio data

get_track_audio(track) -> bool

Determine if a track on a cdrom contains audio data. You can also call `CD.num_tracks()` and `CD.get_all()` to determine more information about the cdrom.

Note, track 0 is the first track on the CD. Track numbers start at zero.

get_all ()

get all track information

get_all() -> [(audio, start, end, length), ...]

Return a list with information for every track on the cdrom. The information consists of a tuple with four values. The audio value is True if the track contains audio data. The start, end, and length values are floating point numbers in seconds. Start and end represent absolute times on the entire disc.

get_track_start ()

start time of a cdrom track

get_track_start(track) -> seconds

Return the absolute time in seconds where at start of the cdrom track.

Note, track 0 is the first track on the CD. Track numbers start at zero.

get_track_length ()

length of a cdrom track

get_track_length(track) -> seconds

Return a floating point value in seconds of the length of the cdrom track.

Note, track 0 is the first track on the CD. Track numbers start at zero.

pygame.Color

pygame.Color

pygame object for color representations

Color(r, g, b) -> Color

Color(r, g, b, a=255) -> Color

Color(color_value) -> Color

The `Color` class represents RGBA color values using a value range of 0 to 255 inclusive. It allows basic arithmetic operations — binary operations `+`, `-`, `*`, `//`, `%`, and unary operation `~` — to create new colors, supports conversions to other color spaces such as HSV or HSL and lets you adjust single color channels. Alpha defaults to 255 (fully opaque) when not given. The arithmetic operations and `correct_gamma()` method preserve subclasses. For the binary operators, the class of the returned color is that of the left hand color object of the operator.

Color objects support equality comparison with other color objects and 3 or 4 element tuples of integers. There was a bug in pygame 1.8.1 where the default alpha was 0, not 255 like previously.

Color objects export the C level array interface. The interface exports a read-only one dimensional unsigned byte array of the same assigned length as the color. For CPython 2.6 and later, the new buffer interface is also exported, with the same characteristics as the array interface.

The floor division, `//`, and modulus, `%`, operators do not raise an exception for division by zero. Instead, if a color, or alpha, channel in the right hand color is 0, then the result is 0. For example:

```
# These expressions are True
Color(255, 255, 255, 255) // Color(0, 64, 64, 64) == Color(0, 3, 3, 3)
Color(255, 255, 255, 255) % Color(64, 64, 64, 0) == Color(63, 63, 63, 0)
```

Parameters:

- **r** (*int*) -- red value in the range of 0 to 255 inclusive
- **g** (*int*) -- green value in the range of 0 to 255 inclusive
- **b** (*int*) -- blue value in the range of 0 to 255 inclusive
- **a** (*int*) -- (optional) alpha value in the range of 0 to 255 inclusive, default is 255
- **color_value** (*Color* or *str* or *int* or *tuple(int, int, int, [int])* or *list(int, int, int, [int])*) -- color value (see note below for the supported formats) Supported `color_value` formats:
 - Color object:** clones the given `Color` object-
 - color name str:** name of the color to use, e.g. `'red'` (all the supported name strings can be found in the `colordict` module)-
 - HTML color format str:** `'#rrggbbaa'` or `'#rrggbb'`, where rr, gg, bb, and aa are 2-digit hex numbers in the range of 0 to 0xFF inclusive, the aa (alpha) value defaults to 0xFF if not provided-
 - hex number str:** `'0xrrggbbaa'` or `'0xrrggbb'`, where rr, gg, bb, and aa are 2-digit hex numbers in the range of 0x00 to 0xFF inclusive, the aa (alpha) value defaults to 0xFF if not provided-
 - int:** int value of the color to use, using hex numbers can make this parameter more readable, e.g. `0xrrggbbaa`, where rr, gg, bb, and aa are 2-digit hex numbers in the range of 0x00 to 0xFF inclusive, note that the aa (alpha) value is not optional for the int format and must be provided-
 - tuple/list of int color values:** `(R, G, B, A)` or `(R, G, B)`, where R, G, B, and A are int values in the range of 0 to 255 inclusive, the A (alpha) value defaults to 255 if not provided

Returns: a newly created `Color` object

Return type: `Color`

Changed in version 2.0.0: *Changed in version 2.0.0:* Support for tuples, lists, and `Color` objects when creating `Color` objects.

Changed in version 1.9.2: *Changed in version 1.9.2:* Color objects export the C level array interface.

Changed in version 1.9.0: *Changed in version 1.9.0:* Color objects support 4-element tuples of integers.

Changed in version 1.8.1: *Changed in version 1.8.1:* New implementation of the class.

r

Gets or sets the red value of the Color.

r -> int

The red value of the Color.

g

Gets or sets the green value of the Color.

g -> int

The green value of the Color.

b

Gets or sets the blue value of the Color.

b -> int

The blue value of the Color.

a

Gets or sets the alpha value of the Color.

a -> int

The alpha value of the Color.

cmY

Gets or sets the CMY representation of the Color.

cmY -> tuple

The CMY representation of the Color. The CMY components are in the ranges $C = [0, 1]$, $M = [0, 1]$, $Y = [0, 1]$. Note that this will not return the absolutely exact CMY values for the set RGB values in all cases. Due to the RGB mapping from 0-255 and the CMY mapping from 0-1 rounding errors may cause the CMY values to differ slightly from what you might expect.

hsva

Gets or sets the HSVA representation of the Color.

hsva -> tuple

The HSVA representation of the Color. The HSVA components are in the ranges $H = [0, 360]$, $S = [0, 100]$, $V = [0, 100]$, $A = [0, 100]$. Note that this will not return the absolutely exact HSV values for the set RGB values in all cases. Due to the RGB mapping from 0-255 and the HSV mapping from 0-100 and 0-360 rounding errors may cause the HSV values to differ slightly from what you might expect.

hsla

Gets or sets the HSLA representation of the Color.

hsla -> tuple

The HSLA representation of the Color. The HSLA components are in the ranges $H = [0, 360]$, $S = [0, 100]$, $V = [0, 100]$, $A = [0, 100]$. Note that this will not return the absolutely exact HSL values for the set RGB values in all cases. Due to the RGB mapping from 0-255 and the HSL mapping from 0-100 and 0-360 rounding errors may cause the HSL values to differ slightly from what you might expect.

i1i2i3

Gets or sets the I1I2I3 representation of the Color.

i1i2i3 -> tuple

The I1I2I3 representation of the Color. The I1I2I3 components are in the ranges $I1 = [0, 1]$, $I2 = [-0.5, 0.5]$, $I3 = [-0.5, 0.5]$. Note that this will not return the absolutely exact I1I2I3 values for the set RGB values in all cases. Due to the RGB mapping from 0-255 and the I1I2I3 mapping from 0-1 rounding errors may cause the I1I2I3 values to differ slightly from what you might expect.

normalize ()

Returns the normalized RGBA values of the Color.

normalize() -> tuple

Returns the normalized RGBA values of the Color as floating point values.

correct_gamma ()

Applies a certain gamma value to the Color.

correct_gamma (gamma) -> Color

Applies a certain gamma value to the Color and returns a new Color with the adjusted RGBA values.

set_length()

Set the number of elements in the Color to 1,2,3, or 4.

set_length(len) -> None

The default Color length is 4. Colors can have lengths 1,2,3 or 4. This is useful if you want to unpack to r,g,b and not r,g,b,a. If you want to get the length of a Color do `len(acolor)`.

New in version 1.9.0: *New in version 1.9.0.*

lerp()

returns a linear interpolation to the given Color.

lerp(Color, float) -> Color

Returns a Color which is a linear interpolation between self and the given Color in RGBA space. The second parameter determines how far between self and other the result is going to be. It must be a value between 0 and 1 where 0 means self and 1 means other will be returned.

New in version 2.0.1: *New in version 2.0.1.*

premul_alpha()

returns a Color where the r,g,b components have been multiplied by the alpha.

premul_alpha() -> Color

Returns a new Color where each of the red, green and blue colour channels have been multiplied by the alpha channel of the original color. The alpha channel remains unchanged.

This is useful when working with the `BLEND_PREMULTIPLIED` blending mode flag for `pygame.Surface.blit()`, which assumes that all surfaces using it are using pre-multiplied alpha colors.

New in version 2.0.0: *New in version 2.0.0.*

update()

Sets the elements of the color

update(r, g, b) -> None

update(r, g, b, a=255) -> None

update(color_value) -> None

Sets the elements of the color. See parameters for `pygame.Color()` for the parameters of this function. If the alpha value was not set it will not change.

New in version 2.0.1: *New in version 2.0.1.*

pygame.cursors

pygame module for cursor resources

Pygame offers control over the system hardware cursor. Pygame supports black and white cursors (bitmap cursors), as well as system variant cursors and color cursors. You control the cursor with functions inside `pygame.mouse`.

This cursors module contains functions for loading and decoding various cursor formats. These allow you to easily store your cursors in external files or directly as encoded python strings.

The module includes several standard cursors. The `pygame.mouse.set_cursor()` function takes several arguments. All those arguments have been stored in a single tuple you can call like this:

```
>>> pygame.mouse.set_cursor(*pygame.cursors.arrow)
```

The following variables can be passed to `pygame.mouse.set_cursor` function:

- `pygame.cursors.arrow`
- `pygame.cursors.diamond`
- `pygame.cursors.broken_x`
- `pygame.cursors.tri_left`
- `pygame.cursors.tri_right`

This module also contains a few cursors as formatted strings. You'll need to pass these to `pygame.cursors.compile()` function before you can use them. The example call would look like this:

```
>>> cursor = pygame.cursors.compile(pygame.cursors.textmarker_strings)
>>> pygame.mouse.set_cursor((8, 16), (0, 0), *cursor)
```

The following strings can be converted into cursor bitmaps with `pygame.cursors.compile()`:

- `pygame.cursors.thickarrow_strings`
- `pygame.cursors.sizer_x_strings`
- `pygame.cursors.sizer_y_strings`
- `pygame.cursors.sizer_xy_strings`
- `pygame.cursors.textmarker_strings`

`pygame.cursors.compile()`

create binary cursor data from simple strings

`compile(strings, black='X', white='.', xor='o') -> data, mask`

A sequence of strings can be used to create binary cursor data for the system cursor. This returns the binary data in the form of two tuples. Those can be passed as the third and fourth arguments respectively of the `pygame.mouse.set_cursor()` function.

If you are creating your own cursor strings, you can use any value represent the black and white pixels. Some system allow you to set a special toggle color for the system color, this is also called the xor color. If the system does not support xor cursors, that color will simply be black.

The height must be divisible by 8. The width of the strings must all be equal and be divisible by 8. If these two conditions are not met, `ValueError` is raised. An example set of cursor strings looks like this

```
thickarrow_strings = (                                #sized 24x24
    "XX",
    "XXX",
    "XXXX",
    "XX.XX",
    "XX..XX",
    "XX...XX",
    "XX....XX",
    "XX.....XX",
    "XX.....XX",
    "XX.....XX",
    "XX.....XX",
    "XX.....XXX",
    "XX.....XXXX",
    "XX.XXX..XX",
    "XXXX XX..XX",
    "XX  XX..XX",
    "    XX..XX",
    "    XX..XX",
    "    XXXX",
    "    XX",
    " ",
    " ",
    " ")
```

`pygame.cursors.load_xbm()`

load cursor data from an XBM file

`load_xbm(cursorfile) -> cursor_args`

`load_xbm(cursorfile, maskfile) -> cursor_args`

This loads cursors for a simple subset of XBM files. XBM files are traditionally used to store cursors on UNIX systems, they are an ASCII format used to represent simple images.

Sometimes the black and white color values will be split into two separate XBM files. You can pass a second maskfile argument to load the two images into a single cursor.

The cursorfile and maskfile arguments can either be filenames or file-like object with the `readlines` method.

The return value `cursor_args` can be passed directly to the `pygame.mouse.set_cursor()` function.

pygame.cursors.Cursor

pygame object representing a cursor

`Cursor(size, hotspot, xormasks, andmasks) -> Cursor`

`Cursor(hotspot, surface) -> Cursor`

`Cursor(constant) -> Cursor`

In pygame 2, there are 3 types of cursors you can create to give your game that little bit of extra polish. There's **bitmap** type cursors, which existed in pygame 1.x, and are compiled from a string or load from an xbm file. Then there are **system** type cursors, where you choose a preset that will convey the same meaning but look native across different operating systems. Finally you can create a **color** cursor, which displays a pygame surface as the cursor.

Creating a system cursor

Choose a constant from this list, pass it into `pygame.cursors.Cursor(constant)`, and you're good to go.

Pygame Cursor Constant	Description
-----	-----
<code>pygame.SYSTEM_CURSOR_ARROW</code>	arrow
<code>pygame.SYSTEM_CURSOR_IBEAM</code>	i-beam
<code>pygame.SYSTEM_CURSOR_WAIT</code>	wait
<code>pygame.SYSTEM_CURSOR_CROSSHAIR</code>	crosshair
<code>pygame.SYSTEM_CURSOR_WAITARROW</code>	small wait cursor (or wait if not available)
<code>pygame.SYSTEM_CURSOR_SIZENWSE</code>	double arrow pointing northwest and southeast
<code>pygame.SYSTEM_CURSOR_SIZENESW</code>	double arrow pointing northeast and southwest
<code>pygame.SYSTEM_CURSOR_SIZWE</code>	double arrow pointing west and east
<code>pygame.SYSTEM_CURSOR_SIZENS</code>	double arrow pointing north and south
<code>pygame.SYSTEM_CURSOR_SIZEALL</code>	four pointed arrow pointing north, south, east, and west
<code>pygame.SYSTEM_CURSOR_NO</code>	slashed circle or crossbones
<code>pygame.SYSTEM_CURSOR_HAND</code>	hand

Creating a color cursor

To create a color cursor, create a `Cursor` from a `hotspot` and a `surface`. `hotspot` is an (x,y) coordinate that determines where in the cursor the exact point is. The `hotspot` position must be within the bounds of the `surface`.

Creating a bitmap cursor

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.

A cursor has a width and height, but a mouse position is represented by a set of point coordinates. So the value passed into the cursor `hotspot` variable helps pygame to actually determine at what exact point the cursor is at. `xormasks` is a sequence of bytes containing the cursor xor data masks. Lastly `andmasks`, a sequence of bytes containing the cursor bitmask data. To create these variables, we can make use of the `pygame.cursors.compile()` function.

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

type

Gets the cursor type

type -> string

The type will be "system", "bitmap", or "color".

data

Gets the cursor data

data -> tuple

Returns the data that was used to create this cursor object, wrapped up in a tuple.

New in version 2.0.1: *New in version 2.0.1.*

Example code for creating and settings cursors. (Click the mouse to switch cursor)

```
# pygame setup
import pygame as pg

pg.init()
screen = pg.display.set_mode([600, 400])
pg.display.set_caption("Example code for the cursors module")

# create a system cursor
system = pg.cursors.Cursor(pg.SYSTEM_CURSOR_NO)

# create bitmap cursors
bitmap_1 = pg.cursors.Cursor(*pg.cursors.arrow)
bitmap_2 = pg.cursors.Cursor(
    (24, 24), (0, 0), *pg.cursors.compile(pg.cursors.thickarrow_strings)
)

# create a color cursor
surf = pg.Surface((40, 40)) # you could also load an image
surf.fill((120, 50, 50))    # and use that as your surface
color = pg.cursors.Cursor((20, 20), surf)

cursors = [system, bitmap_1, bitmap_2, color]
cursor_index = 0

pg.mouse.set_cursor(cursors[cursor_index])

clock = pg.time.Clock()
going = True
while going:
    clock.tick(60)
    screen.fill((0, 75, 30))
    pg.display.flip()

    for event in pg.event.get():
        if event.type == pg.QUIT or (event.type == pg.KEYDOWN and event.key == pg.K_ESCAPE):
            going = False

        # if the mouse is clicked it will switch to a new cursor
        if event.type == pg.MOUSEBUTTONDOWN:
            cursor_index += 1
            cursor_index %= len(cursors)
            pg.mouse.set_cursor(cursors[cursor_index])

pg.quit()
```

pygame.display

pygame module to control the display window and screen

This module offers control over the pygame display. Pygame has a single display Surface that is either contained in a window or runs full screen. Once you create the display you treat it as a regular Surface. Changes are not immediately visible onscreen; you must choose one of the two flipping functions to update the actual display.

The origin of the display, where $x = 0$ and $y = 0$, is the top left of the screen. Both axes increase positively towards the bottom right of the screen.

The pygame display can actually be initialized in one of several modes. By default, the display is a basic software driven framebuffer. You can request special modules like hardware acceleration and OpenGL support. These are controlled by flags passed to `pygame.display.set_mode()`.

Pygame can only have a single display active at any time. Creating a new one with `pygame.display.set_mode()` will close the previous display. If precise control is needed over the pixel format or display resolutions, use the functions `pygame.display.mode_ok()`, `pygame.display.list_modes()`, and `pygame.display.Info()` to query information about the display.

Once the display Surface is created, the functions from this module affect the single existing display. The Surface becomes invalid if the module is uninitialized. If a new display mode is set, the existing Surface will automatically switch to operate on the new display.

When the display mode is set, several events are placed on the pygame event queue. `pygame.QUIT` is sent when the user has requested the program to shut down. The window will receive `pygame.ACTIVEEVENT` events as the display gains and loses input focus. If the display is set with the `pygame.RESIZABLE` flag, `pygame.VIDEORESIZE` events will be sent when the user adjusts the window dimensions. Hardware displays that draw direct to the screen will get `pygame.VIDEOEXPOSE` events when portions of the window must be redrawn.

In pygame 2, there is a new type of event called `pygame.WINDOWEVENT` that is meant to replace all window related events like `pygame.VIDEORESIZE`, `pygame.VIDEOEXPOSE` and `pygame.ACTIVEEVENT`.

Note that the WINDOWEVENT API is considered experimental, and may change in future releases.

The new events of type `pygame.WINDOWEVENT` have an `event` attribute that can take the following values.

Value of event attribute	Short description
<code>WINDOWEVENT_SHOWN</code>	Window became shown
<code>WINDOWEVENT_HIDDEN</code>	Window became hidden
<code>WINDOWEVENT_EXPOSED</code>	Window got updated by some external event
<code>WINDOWEVENT_MOVED</code>	Window got moved
<code>WINDOWEVENT_RESIZED</code>	Window got resized
<code>WINDOWEVENT_SIZE_CHANGED</code>	Window changed it's size
<code>WINDOWEVENT_MINIMIZED</code>	Window was minimised
<code>WINDOWEVENT_MAXIMIZED</code>	Window was maximised
<code>WINDOWEVENT_RESTORED</code>	Window was restored
<code>WINDOWEVENT_ENTER</code>	Mouse entered the window
<code>WINDOWEVENT_LEAVE</code>	Mouse left the window
<code>WINDOWEVENT_FOCUS_GAINED</code>	Window gained focus
<code>WINDOWEVENT_FOCUS_LOST</code>	Window lost focus
<code>WINDOWEVENT_CLOSE</code>	Window was closed
<code>WINDOWEVENT_TAKE_FOCUS</code>	Window was offered focus
<code>WINDOWEVENT_HIT_TEST</code>	Window has a special hit test

If SDL version used is less than 2.0.5, the last two values `WINDOWEVENT_TAKE_FOCUS` and `WINDOWEVENT_HIT_TEST` will not work. See the SDL implementation (in C programming) of the same [over here](#).

Some display environments have an option for automatically stretching all windows. When this option is enabled, this automatic stretching distorts the appearance of the pygame window. In the pygame examples directory, there is example code (`prevent_display_stretching.py`) which shows how to disable this automatic stretching of the pygame display on Microsoft Windows (Vista or newer required).

`pygame.display.init()`

Initialize the display module

`init()` -> None

Initializes the pygame display module. The display module cannot do anything until it is initialized. This is usually handled for you automatically when you call the higher level `pygame.init()`.

Pygame will select from one of several internal display backends when it is initialized. The display mode will be chosen depending on the platform and permissions of current user. Before the display module is initialized the environment variable `SDL_VIDEODRIVER` can be set to control which backend is used. The systems with multiple choices are listed here.

```
Windows : windib, directx
Unix    : x11, dga, fbcon, directfb, ggi, vgl, svgalib, aalib
```

On some platforms it is possible to embed the pygame display into an already existing window. To do this, the environment variable `SDL_WINDOWID` must be set to a string containing the window id or handle. The environment variable is checked when the pygame display is initialized. Be aware that there can be many strange side effects when running in an embedded display.

It is harmless to call this more than once, repeated calls have no effect.

`pygame.display.quit()`

Uninitialize the display module

`quit()` -> None

This will shut down the entire display module. This means any active displays will be closed. This will also be handled automatically when the program exits.

It is harmless to call this more than once, repeated calls have no effect.

`pygame.display.get_init()`

Returns True if the display module has been initialized

`get_init()` -> bool

Returns True if the `pygame.display` module is currently initialized.

`pygame.display.set_mode()`

Initialize a window or screen for display

`set_mode(size=(0, 0), flags=0, depth=0, display=0, vsync=0)` -> Surface

This function will create a display Surface. The arguments passed in are requests for a display type. The actual created display will be the best possible match supported by the system.

The size argument is a pair of numbers representing the width and height. The flags argument is a collection of additional options. The depth argument represents the number of bits to use for color.

The Surface that gets returned can be drawn to like a regular Surface but changes will eventually be seen on the monitor.

If no size is passed or is set to `(0, 0)` and pygame uses SDL version 1.2.10 or above, the created Surface will have the same size as the current screen resolution. If only the width or height are set to 0, the Surface will have the same width or height as the screen resolution. Using a SDL version prior to 1.2.10 will raise an exception.

It is usually best to not pass the depth argument. It will default to the best and fastest color depth for the system. If your game requires a specific color format you can control the depth with this argument. Pygame will emulate an unavailable color depth which can be slow.

When requesting fullscreen display modes, sometimes an exact match for the requested size cannot be made. In these situations pygame will select the closest compatible match. The returned surface will still always match the requested size.

On high resolution displays (4k, 1080p) and tiny graphics games (640x480) show up very small so that they are unplayable. `SCALED` scales up the window for you. The game thinks it's a 640x480 window, but really it can be bigger. Mouse events are scaled for you, so your game doesn't need to do it. Note that `SCALED` is considered an experimental API and may change in future releases.

The flags argument controls which type of display you want. There are several to choose from, and you can even combine multiple types using the bitwise or operator, (the pipe "|" character). If you pass 0 or no flags argument it will default to a software driven window. Here are the display flags you will want to choose from:

<code>pygame.FULLSCREEN</code>	create a fullscreen display
<code>pygame.DOUBLEBUF</code>	recommended for HWSURFACE or OPENGL
<code>pygame.HWSURFACE</code>	hardware accelerated, only in FULLSCREEN
<code>pygame.OPENGL</code>	create an OpenGL-renderable display
<code>pygame.RESIZABLE</code>	display window should be sizeable
<code>pygame.NOFRAME</code>	display window will have no border or controls

Pygame 2 has the following additional flags available.

<code>pygame.SCALED</code>	resolution depends on desktop size and scale graphics
<code>pygame.SHOWN</code>	window is opened in visible mode (default)
<code>pygame.HIDDEN</code>	window is opened in hidden mode

New in version 2.0.0: *New in version 2.0.0:* `SCALED`, `SHOWN` and `HIDDEN`

By setting the `vsync` parameter to 1, it is possible to get a display with vertical sync, but you are not guaranteed to get one. The request only works at all for calls to `set_mode()` with the `pygame.OPENGL` or `pygame.SCALED` flags set, and is still not guaranteed even with one of those set. What you get depends on the hardware and driver

configuration of the system pygame is running on. Here is an example usage of a call to `set_mode()` that may give you a display with vsync:

```
flags = pygame.OPENGL | pygame.FULLSCREEN
window_surface = pygame.display.set_mode((1920, 1080), flags, vsync=1)
```

Vsync behaviour is considered experimental, and may change in future releases.

New in version 2.0.0: *New in version 2.0.0:* vsync

Basic example:

```
# Open a window on the screen
screen_width=700
screen_height=400
screen=pygame.display.set_mode([screen_width, screen_height])
```

The display index 0 means the default display is used.

Changed in version 1.9.5: *Changed in version 1.9.5:* display argument added

`pygame.display.get_surface()`

Get a reference to the currently set display surface

`get_surface()` -> Surface

Return a reference to the currently set display Surface. If no display mode has been set this will return None.

`pygame.display.flip()`

Update the full display Surface to the screen

`flip()` -> None

This will update the contents of the entire display. If your display mode is using the flags `pygame.HWSURFACE` and `pygame.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 `pygame.OPENGL` display mode this will perform a gl buffer swap.

`pygame.display.update()`

Update portions of the screen for software displays

`update(rectangle=None)` -> None

`update(rectangle_list)` -> None

This function is like an optimized version of `pygame.display.flip()` for software displays. It allows only a portion of the screen to be updated, instead of the entire area. If no argument is passed it updates the entire Surface area like `pygame.display.flip()`.

You can pass the function a single rectangle, or a sequence of rectangles. It is more efficient to pass many rectangles at once than to call update multiple times with single or a partial list of rectangles. If passing a sequence of rectangles it is safe to include None values in the list, which will be skipped.

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

`pygame.display.get_driver()`

Get the name of the pygame display backend

`get_driver()` -> name

Pygame chooses one of many available display backends when it is initialized. This returns the internal name used for the display backend. This can be used to provide limited information about what display capabilities might be accelerated. See the `SDL_VIDEODRIVER` flags in `pygame.display.set_mode()` to see some of the common options.

`pygame.display.Info()`

Create a video display information object

`Info()` -> VideoInfo

Creates a simple object containing several attributes to describe the current graphics environment. If this is called before `pygame.display.set_mode()` some platforms can provide information about the default display mode. This can also be called after setting the display mode to verify specific display options were satisfied. The `VidInfo` object has several attributes:

```
hw:          1 if the display is hardware accelerated
wm:          1 if windowed display modes can be used
video_mem:   The megabytes of video memory on the display. This is 0 if
```



```

        unknown
bitsize:    Number of bits used to store each pixel
bytesize:   Number of bytes used to store each pixel
masks:      Four values used to pack RGBA values into pixels
shifts:     Four values used to pack RGBA values into pixels
losses:     Four values used to pack RGBA values into pixels
blit_hw:    1 if hardware Surface blitting is accelerated
blit_hw_CC: 1 if hardware Surface colorkey blitting is accelerated
blit_hw_A:  1 if hardware Surface pixel alpha blitting is accelerated
blit_sw:    1 if software Surface blitting is accelerated
blit_sw_CC: 1 if software Surface colorkey blitting is accelerated
blit_sw_A:  1 if software Surface pixel alpha blitting is accelerated
current_h, current_w: Height and width of the current video mode, or
                      of the desktop mode if called before the display.set_mode
                      is called. (current_h, current_w are available since
                      SDL 1.2.10, and pygame 1.8.0). They are -1 on error, or if
                      an old SDL is being used.

```

`pygame.display.get_wm_info()`

Get information about the current windowing system

`get_wm_info()` -> dict

Creates a dictionary filled with string keys. The strings and values are arbitrarily created by the system. Some systems may have no information and an empty dictionary will be returned. Most platforms will return a "window" key with the value set to the system id for the current display.

New in version 1.7.1: *New in version 1.7.1.*

`pygame.display.list_modes()`

Get list of available fullscreen modes

`list_modes(depth=0, flags=pygame.FULLSCREEN, display=0)` -> list

This function returns a list of possible sizes 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 size should work (this is likely the case for windowed modes). Mode sizes are sorted from biggest to smallest.

If depth is 0, the current/best color depth for the display is used. The flags defaults to `pygame.FULLSCREEN`, but you may need to add additional flags for specific fullscreen modes.

The display index 0 means the default display is used.

Changed in version 1.9.5: *Changed in version 1.9.5: display argument added*

`pygame.display.mode_ok()`

Pick the best color depth for a display mode

`mode_ok(size, flags=0, depth=0, display=0)` -> depth

This function uses the same arguments as `pygame.display.set_mode()`. It is used to determine if a requested display mode is available. It will return 0 if the display mode cannot be set. Otherwise it will return a pixel depth that best matches the display asked for.

Usually the depth argument is not passed, but some platforms can support multiple display depths. If passed it will hint to which depth is a better match.

The most useful flags to pass will be `pygame.HWSURFACE`, `pygame.DOUBLEBUF`, and maybe `pygame.FULLSCREEN`. The function will return 0 if these display flags cannot be set.

The display index 0 means the default display is used.

Changed in version 1.9.5: *Changed in version 1.9.5: display argument added*

`pygame.display.gl_get_attribute()`

Get the value for an OpenGL flag for the current display

`gl_get_attribute(flag)` -> value

After calling `pygame.display.set_mode()` with the `pygame.OPENGL` flag, it is a good idea to check the value of any requested OpenGL attributes. See `pygame.display.gl_set_attribute()` for a list of valid flags.

`pygame.display.gl_set_attribute()`

Request an OpenGL display attribute for the display mode

`gl_set_attribute(flag, value)` -> None

When calling `pygame.display.set_mode()` with the `pygame.OPENGL` flag, Pygame automatically handles setting the OpenGL attributes like color and double-buffering. 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()`.

Many settings are the requested minimum. Creating a window with an OpenGL context will fail if OpenGL cannot provide the requested attribute, but it may for example give you a stencil buffer even if you request none, or it may give you a larger one than requested.

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_MULTISAMPLEBUFFERS, GL_MULTISAMPLES, GL_STEREO
```

GL_MULTISAMPLEBUFFERS

Whether to enable multisampling anti-aliasing. Defaults to 0 (disabled).

Set `GL_MULTISAMPLES` to a value above 0 to control the amount of anti-aliasing. A typical value is 2 or 3.

GL_STENCIL_SIZE

Minimum bit size of the stencil buffer. Defaults to 0.

GL_DEPTH_SIZE

Minimum bit size of the depth buffer. Defaults to 16.

GL_STEREO

1 enables stereo 3D. Defaults to 0.

GL_BUFFER_SIZE

Minimum bit size of the frame buffer. Defaults to 0.

New in version 2.0.0: *New in version 2.0.0:* Additional attributes:

```
GL_ACCELERATED_VISUAL,
GL_CONTEXT_MAJOR_VERSION, GL_CONTEXT_MINOR_VERSION,
GL_CONTEXT_FLAGS, GL_CONTEXT_PROFILE_MASK,
GL_SHARE_WITH_CURRENT_CONTEXT,
GL_CONTEXT_RELEASE_BEHAVIOR,
GL_FRAMEBUFFER_SRGB_CAPABLE
```

GL_CONTEXT_PROFILE_MASK

Sets the OpenGL profile to one of these values:

<code>GL_CONTEXT_PROFILE_CORE</code>	disable deprecated features
<code>GL_CONTEXT_PROFILE_COMPATIBILITY</code>	allow deprecated features
<code>GL_CONTEXT_PROFILE_ES</code>	allow only the ES feature subset of OpenGL

GL_ACCELERATED_VISUAL

Set to 1 to require hardware acceleration, or 0 to force software render. By default, both are allowed.

`pygame.display.get_active()`

Returns True when the display is active on the screen

`get_active()` -> bool

Returns True when the display Surface is considered actively renderable on the screen and may be visible to the user. This is the default state immediately after `pygame.display.set_mode()`. This method may return True even if the application is fully hidden behind another application window.

This will return False if the display Surface has been iconified or minimized (either via `pygame.display.iconify()` or via an OS specific method such as the minimize-icon available on most desktops).

The method can also return False for other reasons without the application being explicitly iconified or minimized by the user. A notable example being if the user has multiple virtual desktops and the display Surface is not on the active virtual desktop.

Note

This function returning True is unrelated to whether the application has input focus. Please see `pygame.key.get_focused()` and `pygame.mouse.get_focused()` for APIs related to input focus.

`pygame.display.iconify()`

Iconify the display surface

`iconify()` -> bool

Request the window for the display surface be iconified or hidden. Not all systems and displays support an iconified display. The function will return True if successful.

When the display is iconified `pygame.display.get_active()` will return False. The event queue should receive an `ACTIVEEVENT` event when the window has been iconified. Additionally, the event queue also receives a `WINDOWEVENT_MINIMIZED` event when the window has been iconified on pygame 2.

`pygame.display.toggle_fullscreen()`

Switch between fullscreen and windowed displays

`toggle_fullscreen()` -> int

Switches the display window between windowed and fullscreen modes. Display driver support is not great when using pygame 1, but with pygame 2 it is the most reliable method to switch to and from fullscreen.

Supported display drivers in pygame 1:

- x11 (Linux/Unix)
- wayland (Linux/Unix)

Supported display drivers in pygame 2:

- windows (Windows)
- x11 (Linux/Unix)
- wayland (Linux/Unix)
- cocoa (OSX/Mac)

`pygame.display.set_gamma()`

Change the hardware gamma ramps

`set_gamma(red, green=None, blue=None)` -> bool

Set the red, green, and blue gamma values on the display hardware. If the green and blue arguments are not passed, they will both be the same as red. Not all systems and hardware support gamma ramps, if the function succeeds it will return True.

A gamma value of 1.0 creates a linear color table. Lower values will darken the display and higher values will brighten.

`pygame.display.set_gamma_ramp()`

Change the hardware gamma ramps with a custom lookup

`set_gamma_ramp(red, green, blue)` -> bool

Set the red, green, and blue gamma ramps with an explicit lookup table. Each argument should be sequence of 256 integers. The integers should range between 0 and 0xffff. Not all systems and hardware support gamma ramps, if the function succeeds it will return True.

`pygame.display.set_icon()`

Change the system image for the display window

`set_icon(Surface)` -> None

Sets the runtime icon the system will use to represent the display window. All windows default to a simple pygame logo for the window icon.

You can pass any surface, but most systems want a smaller image around 32x32. The image can have colorkey transparency which will be passed to the system.

Some systems do not allow the window icon to change after it has been shown. This function can be called before `pygame.display.set_mode()` to create the icon before the display mode is set.

`pygame.display.set_caption()`

Set the current window caption

set_caption(title, icontitle=None) -> None

If the display has a window title, this function will change the name on the window. Some systems support an alternate shorter title to be used for minimized displays.

`pygame.display.get_caption()`

Get the current window caption

get_caption() -> (title, icontitle)

Returns the title and icontitle for the display Surface. These will often be the same value.

`pygame.display.set_palette()`

Set the display color palette for indexed displays

set_palette(palette=None) -> None

This will change the video display color palette for 8-bit displays. This does not change the palette for the actual display Surface, only the palette that is used to display the Surface. If no palette argument is passed, the system default palette will be restored. The palette is a sequence of RGB triplets.

`pygame.display.get_num_displays()`

Return the number of displays

get_num_displays() -> int

Returns the number of available displays. This is always 1 if `pygame.get_sdl_version()` returns a major version number below 2.

New in version 1.9.5: *New in version 1.9.5.*

`pygame.display.get_window_size()`

Return the size of the window or screen

get_window_size() -> tuple

Returns the size of the window initialized with `pygame.display.set_mode()`. This may differ from the size of the display surface if `SCALED` is used.

New in version 2.0.0: *New in version 2.0.0.*

`pygame.display.get_allow_screensaver()`

Return whether the screensaver is allowed to run.

get_allow_screensaver() -> bool

Return whether screensaver is allowed to run whilst the app is running. Default is `False`. By default pygame does not allow the screensaver during game play.

Note

Some platforms do not have a screensaver or support disabling the screensaver. Please see `pygame.display.set_allow_screensaver()` for caveats with screensaver support.

New in version 2.0.0: *New in version 2.0.0.*

`pygame.display.set_allow_screensaver()`

Set whether the screensaver may run

set_allow_screensaver(bool) -> None

Change whether screensavers should be allowed whilst the app is running. The default is `False`. By default pygame does not allow the screensaver during game play.

If the screensaver has been disallowed due to this function, it will automatically be allowed to run when `pygame.quit()` is called.

It is possible to influence the default value via the environment variable `SDL_HINT_VIDEO_ALLOW_SCREENSAVER`, which can be set to either 0 (disable) or 1 (enable).

Note

Disabling screensaver is subject to platform support. When platform support is absent, this function will silently appear to work even though the screensaver state is unchanged. The lack of feedback is due to SDL not

providing any supported method for determining whether it supports changing the screensaver state. `SDL_HINT_VIDEO_ALLOW_SCREENSAVER` is available in SDL 2.0.2 or later. SDL1.2 does not implement this.

New in version 2.0.0: *New in version 2.0.0.*

pygame.draw

pygame module for drawing shapes

Draw several simple shapes to a surface. These functions will work for rendering to any format of surface. Rendering to hardware surfaces will be slower than regular software surfaces.

Most of the functions take a width argument to represent the size of stroke (thickness) around the edge of the shape. If a width of 0 is passed the shape will be filled (solid).

All the drawing functions respect the clip area for the surface and will be constrained to that area. The functions return a rectangle representing the bounding area of changed pixels. This bounding rectangle is the 'minimum' bounding box that encloses the affected area.

All the drawing functions accept a color argument that can be one of the following formats:

- a `pygame.Color` object
- an (RGB) triplet (tuple/list)
- an (RGBA) quadruplet (tuple/list)
- an integer value that has been mapped to the surface's pixel format (see `pygame.Surface.map_rgb()` and `pygame.Surface.unmap_rgb()`)

A color's alpha value will be written directly into the surface (if the surface contains pixel alphas), but the draw function will not draw transparently.

These functions temporarily lock the surface they are operating on. Many sequential drawing calls can be sped up by locking and unlocking the surface object around the draw calls (see `pygame.Surface.lock()` and `pygame.Surface.unlock()`).

Note

See the `pygame.gfxdraw` module for alternative draw methods.

`pygame.draw.rect()`

draw a rectangle

`rect(surface, color, rect) -> Rect`

`rect(surface, color, rect, width=0, border_radius=0, border_top_left_radius=-1, border_top_right_radius=-1, border_bottom_left_radius=-1, border_bottom_right_radius=-1) -> Rect`

Draws a rectangle on the given surface.

Parameters:

- **surface** (*Surface*) -- surface to draw on
- **color** (*Color* or *int* or *tuple(int, int, int, [int])*) -- color to draw with, the alpha value is optional if using a tuple (*RGB[A]*)
- **rect** (*Rect*) -- rectangle to draw, position and dimensions
- **width** (*int*) -- (optional) used for line thickness or to indicate that the rectangle is to be filled (not to be confused with the width value of the `rect` parameter) if `width == 0`, (default) fill the rectangle if `width > 0`, used for line thickness if `width < 0`, nothing will be drawn. When using `width` values `> 1`, the edge lines will grow outside the original boundary of the `rect`. For more details on how the thickness for edge lines grow, refer to the `width` notes of the `pygame.draw.line()` function.
- **border_radius** (*int*) -- (optional) used for drawing rectangle with rounded corners. The supported range is `[0, min(height, width) / 2]`, with 0 representing a rectangle without rounded corners.
- **border_top_left_radius** (*int*) -- (optional) used for setting the value of top left border. If you don't set this value, it will use the `border_radius` value.
- **border_top_right_radius** (*int*) -- (optional) used for setting the value of top right border. If you don't set this value, it will use the `border_radius` value.
- **border_bottom_left_radius** (*int*) -- (optional) used for setting the value of bottom left border. If you don't set this value, it will use the `border_radius` value.
- **border_bottom_right_radius** (*int*) -- (optional) used for setting the value of bottom right border. If you don't set this value, it will use the `border_radius` value. if `border_radius < 1` it will draw rectangle without rounded corners if any of border radii has the value `< 0` it will use value of the `border_radius`. If sum of radii on the same side of the rectangle is greater than the rect size the radii will get scaled

Returns: a `rect` bounding the changed pixels, if nothing is drawn the bounding `rect`'s position will be the position of the given `rect` parameter and its width and height will be 0

Return type: `Rect`

Note

The `pygame.Surface.fill()` method works just as well for drawing filled rectangles and can be hardware accelerated on some platforms with both software and hardware display modes.

Changed in version 2.0.0: *Changed in version 2.0.0:* Added support for keyword arguments.

Changed in version 2.0.0.dev8: *Changed in version 2.0.0.dev8:* Added support for border radius.

`pygame.draw.polygon()`

draw a polygon

`polygon(surface, color, points)` -> `Rect`

`polygon(surface, color, points, width=0)` -> `Rect`

Draws a polygon on the given surface.

Parameters:

- **surface** (*Surface*) -- surface to draw on
- **color** (*Color* or *int* or *tuple(int, int, int, [int])*) -- color to draw with, the alpha value is optional if using a tuple (*RGB[A]*)
- **points** (*tuple(coordinate)* or *list(coordinate)*) -- a sequence of 3 or more (x, y) coordinates that make up the vertices of the polygon, each *coordinate* in the sequence must be a tuple/list/`pygame.math.Vector2` of 2 ints/floats, e.g. [(x1, y1), (x2, y2), (x3, y3)]
- **width** (*int*) -- (optional) used for line thickness or to indicate that the polygon is to be filled if width == 0, (default) fill the polygon if width > 0, used for line thickness if width < 0, nothing will be drawn When using width values > 1, the edge lines will grow outside the original boundary of the polygon. For more details on how the thickness for edge lines grow, refer to the width notes of the `pygame.draw.line()` function.

Returns: a rect bounding the changed pixels, if nothing is drawn the bounding rect's position will be the position of the first point in the `points` parameter (float values will be truncated) and its width and height will be 0

Return type: `Rect`

Raises:

- **ValueError** -- if `len(points) < 3` (must have at least 3 points)
- **TypeError** -- if `points` is not a sequence or `points` does not contain number pairs

Note

For an aapolygon, use `aalines()` with `closed=True`.

Changed in version 2.0.0: *Changed in version 2.0.0:* Added support for keyword arguments.

`pygame.draw.circle()`

draw a circle

`circle(surface, color, center, radius) -> Rect`

`circle(surface, color, center, radius, width=0, draw_top_right=None, draw_top_left=None, draw_bottom_left=None, draw_bottom_right=None) -> Rect`

Draws a circle on the given surface.

Parameters:

- **surface** (*Surface*) -- surface to draw on
- **color** (*Color* or *int* or *tuple(int, int, int, [int])*) -- color to draw with, the alpha value is optional if using a tuple (*RGB[A]*)
- **center** (*tuple(int or float, int or float)* or *list(int or float, int or float)* or *Vector2(int or float, int or float)*) -- center point of the circle as a sequence of 2 ints/floats, e.g. (x, y)
- **radius** (*int or float*) -- radius of the circle, measured from the `center` parameter, nothing will be drawn if the `radius` is less than 1
- **width** (*int*) -- (optional) used for line thickness or to indicate that the circle is to be filled if `width == 0`, (default) fill the circle if `width > 0`, used for line thickness if `width < 0`, nothing will be drawn When using width values > 1, the edge lines will only grow inward.
- **draw_top_right** (*bool*) -- (optional) if this is set to True then the top right corner of the circle will be drawn
- **draw_top_left** (*bool*) -- (optional) if this is set to True then the top left corner of the circle will be drawn
- **draw_bottom_left** (*bool*) -- (optional) if this is set to True then the bottom left corner of the circle will be drawn
- **draw_bottom_right** (*bool*) -- (optional) if this is set to True then the bottom right corner of the circle will be drawn if any of the `draw_circle_part` is True then it will draw all circle parts that have the True value, otherwise it will draw the entire circle.

Returns: a rect bounding the changed pixels, if nothing is drawn the bounding rect's position will be the `center` parameter value (float values will be truncated) and its width and height will be 0

Return type: `Rect`

Raises:

- **TypeError** -- if `center` is not a sequence of two numbers
- **TypeError** -- if `radius` is not a number

Changed in version 2.0.0: *Changed in version 2.0.0:* Added support for keyword arguments. Nothing is drawn when the radius is 0 (a pixel at the `center` coordinates used to be drawn when the radius equaled 0). Floats, and `Vector2` are accepted for the `center` param. The drawing algorithm was improved to look more like a circle.

Changed in version 2.0.0.dev8: *Changed in version 2.0.0.dev8:* Added support for drawing circle quadrants.

`pygame.draw.ellipse()`

draw an ellipse

`ellipse(surface, color, rect) -> Rect`

`ellipse(surface, color, rect, width=0) -> Rect`

Draws an ellipse on the given surface.

Parameters:

- **surface** (`Surface`) -- surface to draw on
- **color** (`Color` or `int` or `tuple(int, int, int, [int])`) -- color to draw with, the alpha value is optional if using a tuple (`RGB[A]`)
- **rect** (`Rect`) -- rectangle to indicate the position and dimensions of the ellipse, the ellipse will be centered inside the rectangle and bounded by it
- **width** (`int`) -- (optional) used for line thickness or to indicate that the ellipse is to be filled (not to be confused with the width value of the `rect` parameter) if `width == 0`, (default) fill the ellipse if `width > 0`, used for line thickness if `width < 0`, nothing will be drawn When using `width` values `> 1`, the edge lines will only grow inward from the original boundary of the `rect` parameter.

Returns: a rect bounding the changed pixels, if nothing is drawn the bounding rect's position will be the position of the given `rect` parameter and its width and height will be 0

Return type: `Rect`

Changed in version 2.0.0: *Changed in version 2.0.0:* Added support for keyword arguments.

`pygame.draw.arc()`

draw an elliptical arc

`arc(surface, color, rect, start_angle, stop_angle) -> Rect`

`arc(surface, color, rect, start_angle, stop_angle, width=1) -> Rect`

Draws an elliptical arc on the given surface.

The two angle arguments are given in radians and indicate the start and stop positions of the arc. The arc is drawn in a counterclockwise direction from the `start_angle` to the `stop_angle`.

Parameters:

- **surface** (*Surface*) -- surface to draw on
- **color** (*Color* or *int* or *tuple(int, int, int, [int])*) -- color to draw with, the alpha value is optional if using a tuple (*RGB[A]*)
- **rect** (*Rect*) -- rectangle to indicate the position and dimensions of the ellipse which the arc will be based on, the ellipse will be centered inside the rectangle
- **start_angle** (*float*) -- start angle of the arc in radians
- **stop_angle** (*float*) -- stop angle of the arc in radians if `start_angle < stop_angle`, the arc is drawn in a counterclockwise direction from the `start_angle` to the `stop_angle` if `start_angle > stop_angle`, `tau` (`tau == 2 * pi`) will be added to the `stop_angle`, if the resulting stop angle value is greater than the `start_angle` the above `start_angle < stop_angle` case applies, otherwise nothing will be drawn if `start_angle == stop_angle`, nothing will be drawn
- **width** (*int*) -- (optional) used for line thickness (not to be confused with the width value of the `rect` parameter) if `width == 0`, nothing will be drawn if `width > 0`, (default is 1) used for line thickness if `width < 0`, same as `width == 0` When using `width` values `> 1`, the edge lines will only grow inward from the original boundary of the `rect` parameter.

Returns: a `rect` bounding the changed pixels, if nothing is drawn the bounding `rect`'s position will be the position of the given `rect` parameter and its width and height will be 0

Return type: `Rect`

Changed in version 2.0.0: *Changed in version 2.0.0:* Added support for keyword arguments.

`pygame.draw.line()`

draw a straight line

`line(surface, color, start_pos, end_pos, width) -> Rect`

`line(surface, color, start_pos, end_pos, width=1) -> Rect`

Draws a straight line on the given surface. There are no endcaps. For thick lines the ends are squared off.

Parameters:

- **surface** (*Surface*) -- surface to draw on
- **color** (*Color* or *int* or *tuple(int, int, int, [int])*) -- color to draw with, the alpha value is optional if using a tuple (*RGB[A]*)
- **start_pos** (*tuple(int or float, int or float)* or *list(int or float, int or float)* or *Vector2(int or float, int or float)*) -- start position of the line, (x, y)
- **end_pos** (*tuple(int or float, int or float)* or *list(int or float, int or float)* or *Vector2(int or float, int or float)*) -- end position of the line, (x, y)
- **width** (*int*) -- (optional) used for line thickness if `width >= 1`, used for line thickness (default is 1) if `width < 1`, nothing will be drawn When using `width` values `> 1`, lines will grow as follows. For odd `width` values, the thickness of each line grows with the original line being in the center. For even `width` values, the thickness of each line grows with the original line being offset from the center (as there is no exact center line drawn). As a result, lines with a slope `< 1` (horizontal-ish) will have 1 more pixel of thickness below the original line (in the y direction). Lines with a slope `>= 1` (vertical-ish) will have 1 more pixel of thickness to the right of the original line (in the x direction).

Returns: a `rect` bounding the changed pixels, if nothing is drawn the bounding `rect`'s position will be the `start_pos` parameter value (float values will be truncated) and its width and height will be 0

Return type: `Rect`

Raises: `TypeError` -- if `start_pos` or `end_pos` is not a sequence of two numbers

Changed in version 2.0.0: *Changed in version 2.0.0:* Added support for keyword arguments.

`pygame.draw.lines()`

draw multiple contiguous straight line segments

lines(surface, color, closed, points) -> Rect

lines(surface, color, closed, points, width=1) -> Rect

Draws a sequence of contiguous straight lines on the given surface. There are no endcaps or miter joints. For thick lines the ends are squared off. Drawing thick lines with sharp corners can have undesired looking results.

Parameters:

- **surface** (*Surface*) -- surface to draw on
- **color** (*Color* or *int* or *tuple(int, int, int, [int])*) -- color to draw with, the alpha value is optional if using a tuple (*RGB[A]*)
- **closed** (*bool*) -- if *True* an additional line segment is drawn between the first and last points in the *points* sequence
- **points** (*tuple(coordinate)* or *list(coordinate)*) -- a sequence of 2 or more (x, y) coordinates, where each *coordinate* in the sequence must be a tuple/list/*pygame.math.Vector2* of 2 ints/floats and adjacent coordinates will be connected by a line segment, e.g. for the points [(x1, y1), (x2, y2), (x3, y3)] a line segment will be drawn from (x1, y1) to (x2, y2) and from (x2, y2) to (x3, y3), additionally if the *closed* parameter is *True* another line segment will be drawn from (x3, y3) to (x1, y1)
- **width** (*int*) -- (optional) used for line thickness if width >= 1, used for line thickness (default is 1) if width < 1, nothing will be drawn When using width values > 1 refer to the width notes of *line()* for details on how thick lines grow.

Returns: a rect bounding the changed pixels, if nothing is drawn the bounding rect's position will be the position of the first point in the *points* parameter (float values will be truncated) and its width and height will be 0

Return type: *Rect*

Raises:

- **ValueError** -- if *len(points) < 2* (must have at least 2 points)
- **TypeError** -- if *points* is not a sequence or *points* does not contain number pairs

Changed in version 2.0.0: *Changed in version 2.0.0:* Added support for keyword arguments.

pygame.draw.aaline()

draw a straight antialiased line

aaline(surface, color, start_pos, end_pos) -> Rect

aaline(surface, color, start_pos, end_pos, blend=1) -> Rect

Draws a straight antialiased line on the given surface.

The line has a thickness of one pixel and the endpoints have a height and width of one pixel each.

The way a line and it's endpoints are drawn:

If both endpoints are equal, only a single pixel is drawn (after rounding floats to nearest integer).

Otherwise if the line is not steep (i.e. if the length along the x-axis is greater than the height along the y-axis):

For each endpoint:

If *x*, the endpoint's x-coordinate, is a whole number find which pixels would be covered by it and draw them.

Otherwise:

Calculate the position of the nearest point with a whole number for it's x-coordinate, when extending the line past the endpoint.

Find which pixels would be covered and how much by that point.

If the endpoint is the left one, multiply the coverage by (1 - the decimal part of *x*).

Otherwise multiply the coverage by the decimal part of *x*.

Then draw those pixels.

e.g.:

The left endpoint of the line ((1, 1.3), (5, 3)) would cover 70% of the pixel (1, 1) and 30% of the pixel (1, 2) while the right one would cover 100% of the pixel (5, 3).

The left endpoint of the line $((1.2, 1.4), (4.6, 3.1))$ would cover 56% (i.e. $0.8 * 70\%$) of the pixel $(1, 1)$ and 24% (i.e. $0.8 * 30\%$) of the pixel $(1, 2)$ while the right one would cover 42% (i.e. $0.6 * 70\%$) of the pixel $(5, 3)$ and 18% (i.e. $0.6 * 30\%$) of the pixel $(5, 4)$ while the right

Then for each point between the endpoints, along the line, whose x-coordinate is a whole number:

Find which pixels would be covered and how much by that point and draw them.

e.g.:

The points along the line $((1, 1), (4, 2.5))$ would be $(2, 1.5)$ and $(3, 2)$ and would cover 50% of the pixel $(2, 1)$, 50% of the pixel $(2, 2)$ and 100% of the pixel $(3, 2)$.

The points along the line $((1.2, 1.4), (4.6, 3.1))$ would be $(2, 1.8)$ (covering 20% of the pixel $(2, 1)$ and 80% of the pixel $(2, 2)$), $(3, 2.3)$ (covering 70% of the pixel $(3, 2)$ and 30% of the pixel $(3, 3)$) and $(4, 2.8)$ (covering 20% of the pixel $(2, 1)$ and 80% of the pixel $(2, 2)$)

Otherwise do the same for steep lines as for non-steep lines except along the y-axis instead of the x-axis (using y instead of x , top instead of left and bottom instead of right).

Note

Regarding float values for coordinates, a point with coordinate consisting of two whole numbers is considered being right in the center of said pixel (and having a height and width of 1 pixel would therefore completely cover it), while a point with coordinate where one (or both) of the numbers have non-zero decimal parts would be partially covering two (or four if both numbers have decimal parts) adjacent pixels, e.g. the point $(1.4, 2)$ covers 60% of the pixel $(1, 2)$ and 40% of the pixel $(2, 2)$.

Parameters:

- **surface** (*Surface*) -- surface to draw on
- **color** (*Color* or *int* or *tuple(int, int, int, [int])*) -- color to draw with, the alpha value is optional if using a tuple (*RGB[A]*)
- **start_pos** (*tuple(int or float, int or float)* or *list(int or float, int or float)* or *Vector2(int or float, int or float)*) -- start position of the line, (x, y)
- **end_pos** (*tuple(int or float, int or float)* or *list(int or float, int or float)* or *Vector2(int or float, int or float)*) -- end position of the line, (x, y)
- **blend** (*int*) -- (optional) if non-zero (default) the line will be blended with the surface's existing pixel shades, otherwise it will overwrite them

Returns: a rect bounding the changed pixels, if nothing is drawn the bounding rect's position will be the `start_pos` parameter value (float values will be truncated) and its width and height will be 0

Return type: *Rect*

Raises: **TypeError** -- if `start_pos` or `end_pos` is not a sequence of two numbers

Changed in version 2.0.0: *Changed in version 2.0.0:* Added support for keyword arguments.

`pygame.draw.aalines()`

draw multiple contiguous straight antialiased line segments

`aalines(surface, color, closed, points) -> Rect`

`aalines(surface, color, closed, points, blend=1) -> Rect`

Draws a sequence of contiguous straight antialiased lines on the given surface.

Parameters:

- **surface** (*Surface*) -- surface to draw on
- **color** (*Color* or *int* or *tuple(int, int, int, [int])*) -- color to draw with, the alpha value is optional if using a tuple (*RGB[A]*)
- **closed** (*bool*) -- if *True* an additional line segment is drawn between the first and last points in the *points* sequence
- **points** (*tuple(coordinate)* or *list(coordinate)*) -- a sequence of 2 or more (x, y) coordinates, where each *coordinate* in the sequence must be a tuple/list/**pygame.math.Vector2** of 2 ints/floats and adjacent coordinates will be connected by a line segment, e.g. for the points [(x1, y1), (x2, y2), (x3, y3)] a line segment will be drawn from (x1, y1) to (x2, y2) and from (x2, y2) to (x3, y3), additionally if the *closed* parameter is *True* another line segment will be drawn from (x3, y3) to (x1, y1)
- **blend** (*int*) -- (optional) if non-zero (default) each line will be blended with the surface's existing pixel shades, otherwise the pixels will be overwritten

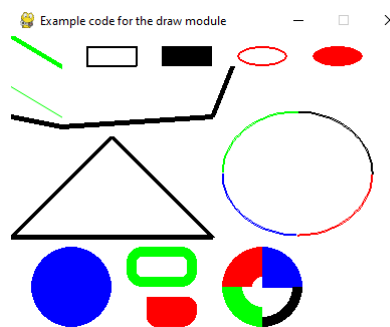
Returns: a rect bounding the changed pixels, if nothing is drawn the bounding rect's position will be the position of the first point in the *points* parameter (float values will be truncated) and its width and height will be 0

Return type: **Rect**

Raises:

- **ValueError** -- if `len(points) < 2` (must have at least 2 points)
- **TypeError** -- if *points* is not a sequence or *points* does not contain number pairs

Changed in version 2.0.0: *Changed in version 2.0.0:* Added support for keyword arguments.



Example code for draw module.

```
# Import a library of functions called 'pygame'
import pygame
from math import pi

# Initialize the game engine
pygame.init()

# Define the colors we will use in RGB format
BLACK = ( 0, 0, 0)
WHITE = (255, 255, 255)
BLUE = ( 0, 0, 255)
GREEN = ( 0, 255, 0)
RED = (255, 0, 0)

# Set the height and width of the screen
size = [400, 300]
screen = pygame.display.set_mode(size)

pygame.display.set_caption("Example code for the draw module")

#Loop until the user clicks the close button.
done = False
```

```

clock = pygame.time.Clock()

while not done:

    # This limits the while loop to a max of 10 times per second.
    # Leave this out and we will use all CPU we can.
    clock.tick(10)

    for event in pygame.event.get(): # User did something
        if event.type == pygame.QUIT: # If user clicked close
            done=True # Flag that we are done so we exit this loop

    # All drawing code happens after the for loop and but
    # inside the main while done==False loop.

    # Clear the screen and set the screen background
    screen.fill(WHITE)

    # Draw on the screen a GREEN line from (0, 0) to (50, 30)
    # 5 pixels wide.
    pygame.draw.line(screen, GREEN, [0, 0], [50,30], 5)

    # Draw on the screen 3 BLACK lines, each 5 pixels wide.
    # The 'False' means the first and last points are not connected.
    pygame.draw.lines(screen, BLACK, False, [[0, 80], [50, 90], [200, 80], [220, 30]], 5)

    # Draw on the screen a GREEN line from (0, 50) to (50, 80)
    # Because it is an antialiased line, it is 1 pixel wide.
    pygame.draw.aaline(screen, GREEN, [0, 50],[50, 80], True)

    # Draw a rectangle outline
    pygame.draw.rect(screen, BLACK, [75, 10, 50, 20], 2)

    # Draw a solid rectangle
    pygame.draw.rect(screen, BLACK, [150, 10, 50, 20])

    # Draw a rectangle with rounded corners
    pygame.draw.rect(screen, GREEN, [115, 210, 70, 40], 10, border_radius=15)
    pygame.draw.rect(screen, RED, [135, 260, 50, 30], 0, border_radius=10, border_top_left_r=
        border_bottom_right_radius=15)

    # Draw an ellipse outline, using a rectangle as the outside boundaries
    pygame.draw.ellipse(screen, RED, [225, 10, 50, 20], 2)

    # Draw an solid ellipse, using a rectangle as the outside boundaries
    pygame.draw.ellipse(screen, RED, [300, 10, 50, 20])

    # This draws a triangle using the polygon command
    pygame.draw.polygon(screen, BLACK, [[100, 100], [0, 200], [200, 200]], 5)

    # Draw an arc as part of an ellipse.
    # Use radians to determine what angle to draw.
    pygame.draw.arc(screen, BLACK,[210, 75, 150, 125], 0, pi/2, 2)
    pygame.draw.arc(screen, GREEN,[210, 75, 150, 125], pi/2, pi, 2)
    pygame.draw.arc(screen, BLUE, [210, 75, 150, 125], pi,3*pi/2, 2)
    pygame.draw.arc(screen, RED, [210, 75, 150, 125], 3*pi/2, 2*pi, 2)

    # Draw a circle
    pygame.draw.circle(screen, BLUE, [60, 250], 40)

```

```

# Draw only one circle quadrant
pygame.draw.circle(screen, BLUE, [250, 250], 40, 0, draw_top_right=True)
pygame.draw.circle(screen, RED, [250, 250], 40, 30, draw_top_left=True)
pygame.draw.circle(screen, GREEN, [250, 250], 40, 20, draw_bottom_left=True)
pygame.draw.circle(screen, BLACK, [250, 250], 40, 10, draw_bottom_right=True)

# Go ahead and update the screen with what we've drawn.
# This MUST happen after all the other drawing commands.
pygame.display.flip()

# Be IDLE friendly
pygame.quit()

```

pygame.event

pygame module for interacting with events and queues

Pygame handles all its 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 may not work properly. The event subsystem should be called from the main thread. If you want to post events into the queue from other threads, please use the `pygame.fastevent` module.

The event queue has an upper limit on the number of events it can hold (128 for standard SDL 1.2). When the queue becomes full new events are quietly dropped. To prevent lost events, especially input events which signal a quit command, your program must handle events every frame (with `pygame.event.get()`, `pygame.event.pump()`, `pygame.event.wait()`, `pygame.event.peek()` or `pygame.event.clear()`) and process them. Not handling events may cause your system to decide your program has locked up. To speed up queue processing use `pygame.event.set_blocked()` to limit which events get queued.

To get the state of various input devices, you can forego the event queue and access the input devices directly with their appropriate modules: `pygame.mouse`, `pygame.key`, and `pygame.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 sync with the system, you will need to call `pygame.event.pump()` to keep everything current. Usually, this should be called once per game loop. Note: Joysticks will not send any events until the device has been initialized.

The event queue contains `pygame.event.EventType` event objects. There are a variety of ways to access the queued events, from simply checking for the existence of events, to grabbing them directly off the stack. The event queue also offers some simple filtering which can slightly help performance by blocking certain event types from the queue. Use `pygame.event.set_allowed()` and `pygame.event.set_blocked()` to change this filtering. By default, all event types can be placed on the queue.

All `pygame.event.EventType` instances contain an event type identifier and attributes specific to that event type. The event type identifier is accessible as the `pygame.event.EventType.type` property. Any of the event specific attributes can be accessed through the `pygame.event.EventType.__dict__` attribute or directly as an attribute of the event object (as member lookups are passed through to the object's dictionary values). The event object has no method functions. Users can create their own new events with the `pygame.event.Event()` function.

The event type identifier is in between the values of `NOEVENT` and `NUMEVENTS`. User defined events should have a value in the inclusive range of `USEREVENT` to `NUMEVENTS - 1`. It is recommended all user events follow this system.

Events support equality and inequality comparisons. Two events are equal if they are the same type and have identical attribute values.

While debugging and experimenting, you can print an event object for a quick display of its type and members. The function `pygame.event.event_name()` can be used to get a string representing the name of the event type.

Events that come from the system will have a guaranteed set of member attributes based on the type. The following is a list event types with their specific attributes.

QUIT	none
ACTIVEEVENT	gain, state

KEYDOWN	key, mod, unicode, scancode
KEYUP	key, mod
MOUSEMOTION	pos, rel, buttons
MOUSEBUTTONUP	pos, button
MOUSEBUTTONDOWN	pos, button
JOYAXISMOTION	joy (deprecated), instance_id, axis, value
JOYBALLMOTION	joy (deprecated), instance_id, ball, rel
JOYHATMOTION	joy (deprecated), instance_id, hat, value
JOYBUTTONUP	joy (deprecated), instance_id, button
JOYBUTTONDOWN	joy (deprecated), instance_id, button
VIDEORESIZE	size, w, h
VIDEOEXPOSE	none
USEREVENT	code

Changed in version 2.0.0: *Changed in version 2.0.0:* The `joy` attribute was deprecated, `instance_id` was added.

You can also find a list of constants for keyboard keys [here](#).

On MacOSX when a file is opened using a pygame application, a `USEREVENT` with its `code` attribute set to `pygame.USEREVENT_DROPFILE` is generated. There is an additional attribute called `filename` where the name of the file being accessed is stored.

```
USEREVENT          code=pygame.USEREVENT_DROPFILE, filename
```

New in version 1.9.2: *New in version 1.9.2.*

When compiled with SDL2, pygame has these additional events and their attributes.

AUDIODEVICEADDED	which, iscapture
AUDIODEVICEREMOVED	which, iscapture
FINGERMOTION	touch_id, finger_id, x, y, dx, dy
FINGERDOWN	touch_id, finger_id, x, y, dx, dy
FINGERUP	touch_id, finger_id, x, y, dx, dy
MOUSEWHEEL	which, flipped, x, y
MULTIGESTURE	touch_id, x, y, pinched, rotated, num_fingers
TEXTEDITING	text, start, length
TEXTINPUT	text
WINDOWEVENT	event

New in version 1.9.5: *New in version 1.9.5.*

pygame can recognize text or files dropped in its window. If a file is dropped, `file` will be its path. The `DROPTTEXT` event is only supported on X11.

```
DROPBEGIN
DROPCOMPLETE
DROPFILE          file
DROPTTEXT         text
```

New in version 2.0.0: *New in version 2.0.0.*

Events reserved for `pygame.midi` use.

```
MIDIIN
MIDIOUT
```

New in version 2.0.0: *New in version 2.0.0.*

SDL2 supports controller hotplugging:

CONTROLLERDEVICEADDED	device_index
JOYDEVICEADDED	device_index
CONTROLLERDEVICEREMOVED	instance_id
JOYDEVICEREMOVED	instance_id
CONTROLLERDEVICEREMAPPED	instance_id

Also in this version, `instance_id` attributes were added to joystick events, and the `joy` attribute was deprecated.

New in version 2.0.0: *New in version 2.0.0.*

`pygame.event.pump()`

internally process pygame event handlers

`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 `pygame.event.pump()` to allow pygame to handle internal actions.

This function is not necessary if your program is consistently processing events on the queue through the other `pygame.event` functions.

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

Caution!

This function should only be called in the thread that initialized `pygame.display`.

`pygame.event.get()`

get events from the queue

`get(eventtype=None)` -> Eventlist

`get(eventtype=None, pump=True)` -> Eventlist

This will get all the messages and remove them from the queue. If a type or sequence of types is given only those messages will be removed from the queue.

If you are only taking specific events from the queue, be aware that the queue could eventually fill up with the events you are not interested.

If `pump` is `True` (the default), then `pygame.event.pump()` will be called.

Changed in version 1.9.5: *Changed in version 1.9.5:* Added `pump` argument

`pygame.event.poll()`

get a single event from the queue

`poll()` -> EventType instance

Returns a single event from the queue. If the event queue is empty an event of type `pygame.NOEVENT` will be returned immediately. The returned event is removed from the queue.

Caution!

This function should only be called in the thread that initialized `pygame.display`.

`pygame.event.wait()`

wait for a single event from the queue

`wait()` -> EventType instance

`wait(timeout)` -> EventType instance

Returns a single event from the queue. If the queue is empty this function will wait until one is created. From pygame 2.0.0, if a `timeout` argument is given, the function will return an event of type `pygame.NOEVENT` if no events enter the queue in `timeout` milliseconds. The event is removed from the queue once it has been returned. While the program is waiting it will sleep in an idle state. This is important for programs that want to share the system with other applications.

Changed in version 2.0.0.dev13: *Changed in version 2.0.0.dev13*: Added `timeout` argument

Caution!

This function should only be called in the thread that initialized `pygame.display`.

`pygame.event.peek()`

test if event types are waiting on the queue

`peek(eventtype=None) -> bool`

`peek(eventtype=None, pump=True) -> bool`

Returns `True` if there are any events of the given type waiting on the queue. If a sequence of event types is passed, this will return `True` if any of those events are on the queue.

If `pump` is `True` (the default), then `pygame.event.pump()` will be called.

Changed in version 1.9.5: *Changed in version 1.9.5*: Added `pump` argument

`pygame.event.clear()`

remove all events from the queue

`clear(eventtype=None) -> None`

`clear(eventtype=None, pump=True) -> None`

Removes all events from the queue. If `eventtype` is given, removes the given event or sequence of events. This has the same effect as `pygame.event.get()` except `None` is returned. It can be slightly more efficient when clearing a full event queue.

If `pump` is `True` (the default), then `pygame.event.pump()` will be called.

Changed in version 1.9.5: *Changed in version 1.9.5*: Added `pump` argument

`pygame.event.event_name()`

get the string name from an event id

`event_name(type) -> string`

Returns a string representing the name (in CapWords style) of the given event type.

"UserEvent" is returned for all values in the user event id range. "Unknown" is returned when the event type does not exist.

`pygame.event.set_blocked()`

control which events are allowed on the queue

`set_blocked(type) -> None`

`set_blocked(typelist) -> None`

`set_blocked(None) -> None`

The given event types are not allowed to appear on the event queue. By default all events can be placed on the queue. It is safe to disable an event type multiple times.

If `None` is passed as the argument, ALL of the event types are blocked from being placed on the queue.

`pygame.event.set_allowed()`

control which events are allowed on the queue

`set_allowed(type) -> None`

`set_allowed(typelist) -> None`

`set_allowed(None) -> None`

The given event types are allowed to appear on the event queue. By default, all event types can be placed on the queue. It is safe to enable an event type multiple times.

If `None` is passed as the argument, ALL of the event types are allowed to be placed on the queue.

`pygame.event.get_blocked()`

test if a type of event is blocked from the queue

`get_blocked(type) -> bool`

`get_blocked(typelist) -> bool`

Returns `True` if the given event type is blocked from the queue. If a sequence of event types is passed, this will return `True` if any of those event types are blocked.

`pygame.event.set_grab()`

control the sharing of input devices with other applications

`set_grab(bool)` -> None

When your program runs in a windowed environment, it will share the mouse and keyboard devices with other applications that have focus. If your program sets the event grab to `True`, it will lock all input into your program. It is best to not always grab the input, since it prevents the user from doing other things on their system.

`pygame.event.get_grab()`

test if the program is sharing input devices

`get_grab()` -> bool

Returns `True` when the input events are grabbed for this application.

`pygame.event.post()`

place a new event on the queue

`post(Event)` -> None

Places the given event at the end of the event queue.

This is usually used for placing `pygame.USEREVENT` events on the queue. Although any type of event can be placed, if using the system event types your program should be sure to create the standard attributes with appropriate values.

If the event queue is full a `pygame.error` is raised.

Caution: In pygame 2.0, calling this function with event types defined by pygame (such as `pygame.KEYDOWN`) may put events into the SDL2 event queue. In this case, an error may be raised if standard attributes of that event are missing or have incompatible values, and unexpected properties may be silently omitted. In order to avoid this behaviour, custom event properties should be used with custom event types. This behaviour is not guaranteed.

`pygame.event.custom_type()`

make custom user event type

`custom_type()` -> int

Reserves a `pygame.USEREVENT` for a custom use.

If too many events are made a `pygame.error` is raised.

New in version 2.0.0.dev3: *New in version 2.0.0.dev3.*

`pygame.event.Event()`

create a new event object

`Event(type, dict)` -> `EventType` instance

`Event(type, **attributes)` -> `EventType` instance

Creates a new event with the given type and attributes. The attributes can come from a dictionary argument with string keys or from keyword arguments.

`pygame.event.EventType`

pygame object for representing events

A pygame object that represents an event. User event instances are created with an `pygame.event.Event()` function call. The `EventType` type is not directly callable. `EventType` instances support attribute assignment and deletion.

type

event type identifier.

type -> int

Read-only. The event type identifier. For user created event objects, this is the `type` argument passed to `pygame.event.Event()`.

For example, some predefined event identifiers are `QUIT` and `MOUSEMOTION`.

__dict__

event attribute dictionary

`__dict__` -> dict

Read-only. The event type specific attributes of an event. The `dict` attribute is a synonym for backward compatibility.

For example, the attributes of a `KEYDOWN` event would be `unicode`, `key`, and `mod`

New in version 1.9.2: *New in version 1.9.2: Mutable attributes.*

pygame.examples

module of example programs

These examples should help get you started with pygame. Here is a brief rundown of what you get. The source code for these examples is in the public domain. Feel free to use for your own projects.

There are several ways to run the examples. First they can be run as stand-alone programs. Second they can be imported and their `main()` methods called (see below). Finally, the easiest way is to use the `python -m` option:

```
python -m pygame.examples.<example name> <example arguments>
```

eg:

```
python -m pygame.examples.scaletest someimage.png
```

Resources such as images and sounds for the examples are found in the `pygame/examples/data` subdirectory.

You can find where the example files are installed by using the following commands inside the python interpreter.

```
>>> import pygame.examples.scaletest
>>> pygame.examples.scaletest.__file__
'/usr/lib/python2.6/site-packages/pygame/examples/scaletest.py'
```

On each OS and version of Python the location will be slightly different. For example on Windows it might be in `'C:/Python26/Lib/site-packages/pygame/examples/'` On Mac OS X it might be in `'/Library/Frameworks/Python.framework/Versions/2.6/lib/python2.6/site-packages/pygame/examples/'`

You can also run the examples in the python interpreter by calling each modules `main()` function.

```
>>> import pygame.examples.scaletest
>>> pygame.examples.scaletest.main()
```

We're always on the lookout for more examples and/or example requests. Code like this is probably the best way to start getting involved with python gaming.

examples as a package is new to pygame 1.9.0. But most of the examples came with pygame much earlier.

`aliens.main()`

play the full aliens example

`aliens.main()` -> None

This started off as a port of the SDL demonstration, Aliens. Now it has evolved into something sort of resembling fun. This demonstrates a lot of different uses of sprites and optimized blitting. Also transparency, colorkeys, fonts, sound, music, joystick, and more. (PS, my high score is 117! goodluck)

`oldalien.main()`

play the original aliens example

`oldalien.main()` -> None

This more closely resembles a port of the SDL Aliens demo. The code is a lot simpler, so it makes a better starting point for people looking at code for the first times. These blitting routines are not as optimized as they should/could be, but the code is easier to follow, and it plays quick enough.

`stars.main()`

run a simple starfield example

`stars.main()` -> None

A simple starfield example. You can change the center of perspective by leftclicking the mouse on the screen.

`chimp.main()`

hit the moving chimp

`chimp.main()` -> None

This simple example is derived from the line-by-line tutorial that comes with pygame. It is based on a 'popular' web banner. Note there are comments here, but for the full explanation, follow along in the tutorial.

`moveit.main()`

display animated objects on the screen

`moveit.main()` -> None

This is the full and final example from the Pygame Tutorial, "How Do I Make It Move". It creates 10 objects and animates them on the screen.

Note it's a bit scant on error checking, but it's easy to read. :] Fortunately, this is python, and we needn't wrestle with a pile of error codes.

`fonty.main()`

run a font rendering example

`fonty.main()` -> None

Super quick, super simple application demonstrating the different ways to render fonts with the font module

`freetype_misc.main()`

run a FreeType rendering example

`freetype_misc.main()` -> None

A showcase of rendering features the `pygame.freetype.Font` class provides in addition to those available with `pygame.font.Font`. It is a demonstration of direct to surface rendering, with vertical text and rotated text, opaque text and semi transparent text, horizontally stretched text and vertically stretched text.

`vgrade.main()`

display a vertical gradient

`vgrade.main()` -> None

Demonstrates creating a vertical gradient with pixelcopy and NumPy python. The app will create a new gradient every half second and report the time needed to create and display the image. If you're not prepared to start working with the NumPy arrays, don't worry about the source for this one :]

`eventlist.main()`

display pygame events

`eventlist.main()` -> None

Eventlist is a sloppy style of pygame, but is a handy tool for learning about pygame events and input. At the top of the screen are the state of several device values, and a scrolling list of events are displayed on the bottom.

This is not quality 'ui' code at all, but you can see how to implement very non-interactive status displays, or even a crude text output control.

`arraydemo.main()`

show various surfarray effects

`arraydemo.main(arraytype=None)` -> None

Another example filled with various surfarray effects. It requires the surfarray and image modules to be installed.

This little demo can also make a good starting point for any of your own tests with surfarray

The `arraytype` parameter is deprecated; passing any value besides 'numpy' will raise ValueError.

`sound.main()`

load and play a sound

`sound.main(file_path=None)` -> None

Extremely basic testing of the mixer module. Load a sound and play it. All from the command shell, no graphics.

If provided, use the audio file 'file_path', otherwise use a default file.

`sound.py` optional command line argument: an audio file

`sound_array_demos.main()`

play various sndarray effects

`sound_array_demos.main(arraytype=None)` -> None

Uses `sndarray` and NumPy to create offset faded copies of the original sound. Currently it just uses hardcoded values for the number of echoes and the delay. Easy for you to recreate as needed.

The `arraytype` parameter is deprecated; passing any value besides 'numpy' will raise ValueError.

`liquid.main()`

display an animated liquid effect

`liquid.main()` -> None

This example was created in a quick comparison with the BlitzBasic gaming language. Nonetheless, it demonstrates a quick 8-bit setup (with colormap).

`glcube.main()`

display an animated 3D cube using OpenGL

glcube.main() -> None

Using PyOpenGL and pygame, this creates a spinning 3D multicolored cube.

scrap_clipboard.main()

access the clipboard

scrap_clipboard.main() -> None

A simple demonstration example for the clipboard support.

mask.main()

display multiple images bounce off each other using collision detection

mask.main(*args) -> None

Positional arguments:

one or more image file names.

This pygame.masks demo will display multiple moving sprites bouncing off each other. More than one sprite image can be provided.

If run as a program then mask.py takes one or more image files as command line arguments.

testsprite.main()

show lots of sprites moving around

testsprite.main(update_rects = True, use_static = False, use_FastRenderGroup = False, screen_dims = [640, 480], use_alpha = False, flags = 0) -> None

Optional keyword arguments:

update_rects - use the RenderUpdate sprite group class
 use_static - include non-moving images
 use_FastRenderGroup - Use the FastRenderGroup sprite group
 screen_dims - pygame window dimensions
 use_alpha - use alpha blending
 flags - additional display mode flags

Like the testsprite.c that comes with SDL, this pygame version shows lots of sprites moving around.

If run as a stand-alone program then no command line arguments are taken.

headless_no_windows_needed.main()

write an image file that is smoothscaled copy of an input file

headless_no_windows_needed.main(fin, fout, w, h) -> None

arguments:

fin - name of an input image file
 fout - name of the output file to create/overwrite
 w, h - size of the rescaled image, as integer width and height

How to use pygame with no windowing system, like on headless servers.

Thumbnail generation with scaling is an example of what you can do with pygame.

NOTE: the pygame scale function uses MMX/SSE if available, and can be run in multiple threads.

If headless_no_windows_needed.py is run as a program it takes the following command line arguments:

-scale inputimage outputimage new_width new_height
 eg. -scale in.png outpng 50 50

fastevents.main()

stress test the fastevents module

fastevents.main() -> None

This is a stress test for the fastevents module.

- Fast events does not appear faster!

-

So far it looks like normal `pygame.event` is faster by up to two times. So maybe fastevent isn't fast at all. Tested on Windows XP SP2 Athlon, and FreeBSD.

However... on my Debian Duron 850 machine fastevents is faster.

`overlay.main()`

play a .pgm video using overlays

`overlay.main(fname) -> None`

Play the .pgm video file given by a path fname.

If run as a program `overlay.py` takes the file name as a command line argument.

`blend_fill.main()`

demonstrate the various surface.fill method blend options

`blend_fill.main() -> None`

A interactive demo that lets one choose which BLEND_xxx option to apply to a surface.

`blit_blends.main()`

uses alternative additive fill to that of surface.fill

`blit_blends.main() -> None`

Fake additive blending. Using NumPy. it doesn't clamp. Press r,g,b Somewhat like `blend_fill`.

`cursors.main()`

display two different custom cursors

`cursors.main() -> None`

Display an arrow or circle with crossbar cursor.

`pixelarray.main()`

display various pixelarray generated effects

`pixelarray.main() -> None`

Display various pixelarray generated effects.

`scaletest.main()`

interactively scale an image using smoothscale

`scaletest.main(imagefile, convert_alpha=False, run_speed_test=True) -> None`

arguments:

```
imagefile - file name of source image (required)
convert_alpha - use convert_alpha() on the surf (default False)
run_speed_test - (default False)
```

A smoothscale example that resized an image on the screen. Vertical and horizontal arrow keys are used to change the width and height of the displayed image. If the `convert_alpha` option is True then the source image is forced to have source alpha, whether or not the original images does. If `run_speed_test` is True then a background timing test is performed instead of the interactive scaler.

If `scaletest.py` is run as a program then the command line options are:

```
ImageFile [-t] [-convert_alpha]
[-t] = Run Speed Test
[-convert_alpha] = Use convert_alpha() on the surf.
```

`midi.main()`

run a midi example

`midi.main(mode='output', device_id=None) -> None`

Arguments:

```
mode - if 'output' run a midi keyboard output example
      'input' run a midi event logger input example
      'list' list available midi devices
      (default 'output')
device_id - midi device number; if None then use the default midi input or
           output device for the system
```

The output example shows how to translate mouse clicks or computer keyboard events into midi notes. It implements a rudimentary button widget and state machine.

The input example shows how to translate midi input to pygame events.

With the use of a virtual midi patch cord the output and input examples can be run as separate processes and connected so the keyboard output is displayed on a console.
new to pygame 1.9.0

`scroll.main()`

run a Surface.scroll example that shows a magnified image

`scroll.main(image_file=None)` -> None

This example shows a scrollable image that has a zoom factor of eight. It uses the `Surface.scroll()` function to shift the image on the display surface. A clip rectangle protects a margin area. If called as a function, the example accepts an optional image file path. If run as a program it takes an optional file path command line argument. If no file is provided a default image file is used.

When running click on a black triangle to move one pixel in the direction the triangle points. Or use the arrow keys. Close the window or press `ESC` to quit.

`camera.main()`

display video captured live from an attached camera

`camera.main()` -> None

A simple live video player, it uses the first available camera it finds on the system.

`playmus.main()`

play an audio file

`playmus.main(file_path)` -> None

A simple music player with window and keyboard playback control. Playback can be paused and rewound to the beginning.

pygame.fastevent

pygame module for interacting with events and queues

pygame.fastevent is a wrapper for Bob Pendleton's fastevent library. It provides fast events for use in multithreaded environments. When using pygame.fastevent, you can not use any of the pump, wait, poll, post, get, peek, etc. functions from pygame.event, but you should use the Event objects.

`pygame.fastevent.init()`

initialize pygame.fastevent

`init()` -> None

Initialize the pygame.fastevent module.

`pygame.fastevent.get_init()`

returns True if the fastevent module is currently initialized

`get_init()` -> bool

Returns True if the pygame.fastevent module is currently initialized.

`pygame.fastevent.pump()`

internally process pygame event handlers

`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.

This function is not necessary if your program is consistently processing events on the queue through the other `pygame.fastevent` functions.

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

`pygame.fastevent.wait()`

wait for an 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

`pygame.fastevent.poll()`

get an available 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`.

`pygame.fastevent.get()`

get all events from the queue

`get()` -> list of Events

This will get all the messages and remove them from the queue.

`pygame.fastevent.post()`

place an event on the queue

`post(Event)` -> None

This will post your own event objects onto the event queue. You can post 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 expect the standard `MOUSEBUTTONDOWN` attributes to be available, like 'pos' and 'button'.

Because `pygame.fastevent.post()` may have to wait for the queue to empty, you can get into a dead lock if you try to append an event on to a full queue from the thread that processes events. For that reason I do not recommend using this function in the main thread of an SDL program.

pygame.font

pygame module for loading and rendering fonts

The font module allows for rendering TrueType fonts into a new Surface object. It accepts any UCS-2 character ('u0001' to 'uFFFF'). This module is optional and requires `SDL_ttf` as a dependency. You should test that `pygame.font` is available and initialized 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()`.

You can load fonts from the system by using the `pygame.font.SysFont()` function. There are a few other functions to help lookup the system fonts.

Pygame comes with a builtin default font. This can always be accessed by passing `None` as the font name.

To use the `pygame.freetype` based `pygame.ftfont` as `pygame.font` define the environment variable `PYGAME_FREETYPE` before the first import of `pygame`. Module `pygame.ftfont` is a `pygame.font` compatible module that passes all but one of the font module unit tests: it does not have the UCS-2 limitation of the `SDL_ttf` based font module, so fails to raise an exception for a code point greater than 'uFFFF'. If `pygame.freetype` is unavailable then the `SDL_ttf` font module will be loaded instead.

`pygame.font.init()`

initialize the font module

`init()` -> None

This method is called automatically by `pygame.init()`. It initializes the font module. The module must be initialized before any other functions will work.

It is safe to call this function more than once.

`pygame.font.quit()`

uninitialize the font module

`quit()` -> None

Manually uninitialize `SDL_ttf`'s font system. This is called automatically by `pygame.quit()`.

It is safe to call this function even if font is currently not initialized.

`pygame.font.get_init()`

true if the font module is initialized

`get_init()` -> bool

Test if the font module is initialized or not.

pygame.font

pygame.font.get_default_font ()

get the filename of the default font

get_default_font() -> string

Return the filename of the system font. This is not the full path to the file. This file can usually be found in the same directory as the font module, but it can also be bundled in separate archives.

pygame.font.get_fonts ()

get all available fonts

get_fonts() -> list of strings

Returns a list of all the fonts available on the system. The names of the fonts will be set to lowercase with all spaces and punctuation removed. This works on most systems, but some will return an empty list if they cannot find fonts.

pygame.font.match_font ()

find a specific font on the system

match_font(name, bold=False, italic=False) -> path

Returns the full path to a font file on the system. If bold or italic are set to true, this will attempt to find the correct family of font.

The font name can also be an iterable of font names, a string of comma-separated font names, or a bytes of comma-separated font names, in which case the set of names will be searched in order. If none of the given names are found, None is returned.

New in version 2.0.1: *New in version 2.0.1:* Accept an iterable of font names.

Example:

```
print pygame.font.match_font('bitstreamverasans')
# output is: /usr/share/fonts/truetype/ttf-bitstream-vera/Vera.ttf
# (but only if you have Vera on your system)
```

pygame.font.SysFont ()

create a Font object from the system fonts

SysFont(name, size, bold=False, italic=False) -> Font

Return a new Font object that is loaded from the system fonts. The font will match the requested bold and italic flags. Pygame uses a small set of common font aliases. If the specific font you ask for is not available, a reasonable alternative may be used. If a suitable system font is not found this will fall back on loading the default pygame font.

The font name can also be an iterable of font names, a string of comma-separated font names, or a bytes of comma-separated font names, in which case the set of names will be searched in order.

New in version 2.0.1: *New in version 2.0.1:* Accept an iterable of font names.

pygame.font.Font

create a new Font object from a file

Font(filename, size) -> Font

Font(object, size) -> Font

Load a new font from a given filename or a python file object. The size is the height of the font in pixels. If the filename is None the pygame default font will be loaded. If a font cannot be loaded from the arguments given an exception will be raised. Once the font is created the size cannot be changed.

Font objects are mainly used to render text into new Surface objects. The render can emulate bold or italic features, but it is better to load from a font with actual italic or bold glyphs. The rendered text can be regular strings or unicode.

bold

Gets or sets whether the font should be rendered in (faked) bold.

bold -> bool

Whether the font should be rendered in bold.

When set to True, this enables the bold rendering of text. This is a fake stretching of the font that doesn't look good on many font types. If possible load the font from a real bold font file. While bold, the font will have a different width than when normal. This can be mixed with the italic and underline modes.

New in version 2.0.0: *New in version 2.0.0.*

italic

Gets or sets whether the font should be rendered in (faked) italics.

italic -> bool

Whether the font should be rendered in italic.

When set to True, this enables fake rendering of italic text. This is a fake skewing of the font that doesn't look good on many font types. If possible load the font from a real italic font file. While italic the font will have a different width than when normal. This can be mixed with the bold and underline modes.

New in version 2.0.0: *New in version 2.0.0.*

underline

Gets or sets whether the font should be rendered with an underline.

underline -> bool

Whether the font should be rendered in underline.

When set to True, all rendered fonts will include an underline. The underline is always one pixel thick, regardless of font size. This can be mixed with the bold and italic modes.

New in version 2.0.0: *New in version 2.0.0.*

render ()

draw text on a new Surface

render(text, antialias, color, background=None) -> Surface

This creates a new Surface with the specified text rendered on it. pygame provides no way to directly draw text on an existing Surface: instead you must use `Font.render()` to create an image (Surface) of the text, then blit this image onto another Surface.

The text can only be a single line: newline characters are not rendered. Null characters ('x00') raise a `TypeError`. Both Unicode and char (byte) strings are accepted. For Unicode strings only UCS-2 characters ('u0001' to 'uFFFF') are recognized. Anything greater raises a `UnicodeError`. For char strings a `LATIN1` encoding is assumed. The antialias argument is a boolean: if true the characters will have smooth edges. The color argument is the color of the text [e.g.: (0,0,255) for blue]. The optional background argument is a color to use for the text background. If no background is passed the area outside the text will be transparent.

The Surface returned will be of the dimensions required to hold the text. (the same as those returned by `Font.size()`). If an empty string is passed for the text, a blank surface will be returned that is zero pixel wide and the height of the font.

Depending on the type of background and antialiasing used, this returns different types of Surfaces. For performance reasons, it is good to know what type of image will be used. If antialiasing is not used, the return image will always be an 8-bit image with a two-color palette. If the background is transparent a colorkey will be set. Antialiased images are rendered to 24-bit RGB images. If the background is transparent a pixel alpha will be included.

Optimization: if you know that the final destination for the text (on the screen) will always have a solid background, and the text is antialiased, you can improve performance by specifying the background color. This will cause the resulting image to maintain transparency information by colorkey rather than (much less efficient) alpha values.

If you render '\n' an unknown char will be rendered. Usually a rectangle. Instead you need to handle new lines yourself.

Font rendering is not thread safe: only a single thread can render text at any time.

size ()

determine the amount of space needed to render text

size(text) -> (width, height)

Returns the dimensions needed to render the text. This can be used to help determine the positioning needed for text before it is rendered. It can also be used for wordwrapping and other layout effects.

Be aware that most fonts use kerning which adjusts the widths for specific letter pairs. For example, the width for "ae" will not always match the width for "a" + "e".

set_underline ()

control if text is rendered with an underline

set_underline(bool) -> None

When enabled, all rendered fonts will include an underline. The underline is always one pixel thick, regardless of font size. This can be mixed with the bold and italic modes.

Note

This is the same as the `underline` attribute.

get_underline ()

check if text will be rendered with an underline

get_underline() -> bool

Return True when the font underline is enabled.

Note

This is the same as the `underline` attribute.

set_bold ()

enable fake rendering of bold text

set_bold(bool) -> None

Enables the bold rendering of text. This is a fake stretching of the font that doesn't look good on many font types. If possible load the font from a real bold font file. While bold, the font will have a different width than when normal. This can be mixed with the italic and underline modes.

Note

This is the same as the `bold` attribute.

get_bold ()

check if text will be rendered bold

get_bold() -> bool

Return True when the font bold rendering mode is enabled.

Note

This is the same as the `bold` attribute.

set_italic ()

enable fake rendering of italic text

set_italic(bool) -> None

Enables fake rendering of italic text. This is a fake skewing of the font that doesn't look good on many font types. If possible load the font from a real italic font file. While italic the font will have a different width than when normal. This can be mixed with the bold and underline modes.

Note

This is the same as the `italic` attribute.

metrics ()

gets the metrics for each character in the passed string

metrics(text) -> list

The list contains tuples for each character, which contain the minimum x offset, the maximum x offset, the minimum y offset, the maximum y offset and the advance offset (bearing plus width) of the character. [(minx, maxx, miny, maxy, advance), (minx, maxx, miny, maxy, advance), ...]. None is entered in the list for each unrecognized character.

`get_italic()`

check if the text will be rendered italic

`get_italic()` -> bool

Return True when the font italic rendering mode is enabled.

Note

This is the same as the `italic` attribute.

`get_linesize()`

get the line space of the font text

`get_linesize()` -> int

Return the height in pixels for a line of text with the font. When rendering multiple lines of text this is the recommended amount of space between lines.

`get_height()`

get the height of the font

`get_height()` -> int

Return the height in pixels of the actual rendered text. This is the average size for each glyph in the font.

`get_ascent()`

get the ascent of the font

`get_ascent()` -> int

Return the height in pixels for the font ascent. The ascent is the number of pixels from the font baseline to the top of the font.

`get_descent()`

get the descent of the font

`get_descent()` -> int

Return the height in pixels for the font descent. The descent is the number of pixels from the font baseline to the bottom of the font.

pygame.freetype

Enhanced pygame module for loading and rendering computer fonts

The `pygame.freetype` module is a replacement for `pygame.font`. It has all of the functionality of the original, plus many new features. Yet it has absolutely no dependencies on the `SDL_ttf` library. It is implemented directly on the FreeType 2 library. The `pygame.freetype` module is not itself backward compatible with `pygame.font`. Instead, use the `pygame.ftfont` module as a drop-in replacement for `pygame.font`.

All font file formats supported by FreeType can be rendered by `pygame.freetype`, namely TTF, Type1, CFF, OpenType, SFNT, PCF, FNT, BDF, PFR and Type42 fonts. All glyphs having UTF-32 code points are accessible (see `Font.ucs4`).

Most work on fonts is done using `Font` instances. The module itself only has routines for initialization and creation of `Font` objects. You can load fonts from the system using the `sysfont()` function.

Extra support of bitmap fonts is available. Available bitmap sizes can be listed (see `Font.get_sizes()`). For bitmap only fonts `Font` can set the size for you (see the `Font.size` property).

For now undefined character codes are replaced with the `.notdef` (not defined) character. How undefined codes are handled may become configurable in a future release.

Pygame comes with a built-in default font. This can always be accessed by passing `None` as the font name to the `Font` constructor.

Extra rendering features available to `pygame.freetype.Font` are direct to surface rendering (see `Font.render_to()`), character kerning (see `Font.kerning`), vertical layout (see `Font.vertical`), rotation of rendered text (see `Font.rotation`), and the strong style (see `Font.strong`). Some properties are configurable, such as strong style strength (see `Font.strong`) and underline positioning (see `Font.underline_adjustment`). Text can be positioned by the upper right corner of the text box or by the text baseline (see `Font.origin`). Finally, a font's vertical and horizontal size can be adjusted separately (see `Font.size`). The `pygame.examples.freetype_misc` example shows these features in use.

The pygame package does not import `freetype` automatically when loaded. This module must be imported explicitly to be used.

```
import pygame
import pygame.freetype
```

New in version 1.9.2: *New in version 1.9.2:* **freetype**

`pygame.freetype.get_error()`

Return the latest FreeType error

`get_error()` -> str

`get_error()` -> None

Return a description of the last error which occurred in the FreeType2 library, or `None` if no errors have occurred.

`pygame.freetype.get_version()`

Return the FreeType version

`get_version()` -> (int, int, int)

Returns the version of the FreeType library in use by this module.

Note that the `freetype` module depends on the FreeType 2 library. It will not compile with the original FreeType 1.0. Hence, the first element of the tuple will always be "2".

`pygame.freetype.init()`

Initialize the underlying FreeType library.

`init(cache_size=64, resolution=72)`

This function initializes the underlying FreeType library and must be called before trying to use any of the functionality of the `freetype` module.

However, `pygame.init()` will automatically call this function if the `freetype` module is already imported. It is safe to call this function more than once.

Optionally, you may specify a default `cache_size` for the Glyph cache: the maximum number of glyphs that will be cached at any given time by the module. Exceedingly small values will be automatically tuned for performance. Also a default pixel `resolution`, in dots per inch, can be given to adjust font scaling.

`pygame.freetype.quit()`

Shut down the underlying FreeType library.

`quit()`

This function closes the `freetype` module. After calling this function, you should not invoke any class, method or function related to the `freetype` module as they are likely to fail or might give unpredictable results. It is safe to call this function even if the module hasn't been initialized yet.

`pygame.freetype.get_init()`

Returns True if the FreeType module is currently initialized.

`get_init()` -> bool

Returns True if the `pygame.freetype` module is currently initialized.

New in version 1.9.5: *New in version 1.9.5.*

`pygame.freetype.was_init()`

DEPRECATED: Use `get_init()` instead.

`was_init()` -> bool

DEPRECATED: Returns True if the `pygame.freetype` module is currently initialized. Use `get_init()` instead.

`pygame.freetype.get_cache_size()`

Return the glyph case size
get_cache_size() -> long
See `pygame.freetype.init()`.

`pygame.freetype.get_default_resolution()`

Return the default pixel size in dots per inch
get_default_resolution() -> long
Returns the default pixel size, in dots per inch, for the module. The default is 72 DPI.

`pygame.freetype.set_default_resolution()`

Set the default pixel size in dots per inch for the module
set_default_resolution([resolution])
Set the default pixel size, in dots per inch, for the module. If the optional argument is omitted or zero the resolution is reset to 72 DPI.

`pygame.freetype.SysFont()`

create a Font object from the system fonts
SysFont(name, size, bold=False, italic=False) -> Font
Return a new Font object that is loaded from the system fonts. The font will match the requested *bold* and *italic* flags. Pygame uses a small set of common font aliases. If the specific font you ask for is not available, a reasonable alternative may be used. If a suitable system font is not found this will fall back on loading the default pygame font.
The font *name* can also be an iterable of font names, a string of comma-separated font names, or a bytes of comma-separated font names, in which case the set of names will be searched in order.
New in version 2.0.1: *New in version 2.0.1:* Accept an iterable of font names.

`pygame.freetype.get_default_font()`

Get the filename of the default font
get_default_font() -> string
Return the filename of the default pygame font. This is not the full path to the file. The file is usually in the same directory as the font module, but can also be bundled in a separate archive.

`pygame.freetype.Font`

Create a new Font instance from a supported font file.
Font(file, size=0, font_index=0, resolution=0, ucs4=False) -> Font
Argument *file* can be either a string representing the font's filename, a file-like object containing the font, or None; if None, a default, Pygame, font is used.
Optionally, a *size* argument may be specified to set the default size in points, which determines the size of the rendered characters. The size can also be passed explicitly to each method call. Because of the way the caching system works, specifying a default size on the constructor doesn't imply a performance gain over manually passing the size on each function call. If the font is bitmap and no *size* is given, the default size is set to the first available size for the font.
If the font file has more than one font, the font to load can be chosen with the *index* argument. An exception is raised for an out-of-range font index value.
The optional *resolution* argument sets the pixel size, in dots per inch, for use in scaling glyphs for this Font instance. If 0 then the default module value, set by `init()`, is used. The Font object's resolution can only be changed by re-initializing the Font instance.
The optional *ucs4* argument, an integer, sets the default text translation mode: 0 (False) recognize UTF-16 surrogate pairs, any other value (True), to treat Unicode text as UCS-4, with no surrogate pairs. See `Font.ucs4`.

name

Proper font name.
name -> string
Read only. Returns the real (long) name of the font, as recorded in the font file.

path

Font file path
path -> unicode
Read only. Returns the path of the loaded font file

size

The default point size used in rendering

size -> float

size -> (float, float)

Get or set the default size for text metrics and rendering. It can be a single point size, given as a Python `int` or `float`, or a font ppm (width, height) `tuple`. Size values are non-negative. A zero size or width represents an undefined size. In this case the size must be given as a method argument, or an exception is raised. A zero width but non-zero height is a `ValueError`.

For a scalable font, a single number value is equivalent to a tuple with width equal height. A font can be stretched vertically with height set greater than width, or horizontally with width set greater than height. For embedded bitmaps, as listed by `get_sizes()`, use the nominal width and height to select an available size.

Font size differs for a non-scalable, bitmap, font. During a method call it must match one of the available sizes returned by method `get_sizes()`. If not, an exception is raised. If the size is a single number, the size is first matched against the point size value. If no match, then the available size with the same nominal width and height is chosen.

get_rect()

Return the size and offset of rendered text

`get_rect(text, style=STYLE_DEFAULT, rotation=0, size=0) -> rect`

Gets the final dimensions and origin, in pixels, of *text* using the optional *size* in points, *style*, and *rotation*. For other relevant render properties, and for any optional argument not given, the default values set for the **Font** instance are used.

Returns a **Rect** instance containing the width and height of the text's bounding box and the position of the text's origin. The origin is useful in aligning separately rendered pieces of text. It gives the baseline position and bearing at the start of the text. See the `render_to()` method for an example.

If *text* is a char (byte) string, its encoding is assumed to be `LATIN1`.

Optionally, *text* can be `None`, which will return the bounding rectangle for the text passed to a previous `get_rect()`, `render()`, `render_to()`, `render_raw()`, or `render_raw_to()` call. See `render_to()` for more details.

get_metrics()

Return the glyph metrics for the given text

`get_metrics(text, size=0) -> [(...), ...]`

Returns the glyph metrics for each character in *text*.

The glyph metrics are returned as a list of tuples. Each tuple gives metrics of a single character glyph. The glyph metrics are:

```
(min_x, max_x, min_y, max_y, horizontal_advance_x, horizontal_advance_y)
```

The bounding box `min_x`, `max_x`, `min_y`, and `max_y` values are returned as grid-fitted pixel coordinates of type `int`. The advance values are float values.

The calculations are done using the font's default size in points. Optionally you may specify another point size with the *size* argument.

The metrics are adjusted for the current rotation, strong, and oblique settings.

If *text* is a char (byte) string, then its encoding is assumed to be `LATIN1`.

height

The unscaled height of the font in font units

height -> int

Read only. Gets the height of the font. This is the average value of all glyphs in the font.

ascender

The unscaled ascent of the font in font units

ascender -> int

Read only. Return the number of units from the font's baseline to the top of the bounding box.

descender

The unscaled descent of the font in font units

descender -> int

Read only. Return the height in font units for the font descent. The descent is the number of units from the font's baseline to the bottom of the bounding box.

get_sized_ascender ()

The scaled ascent of the font in pixels

get_sized_ascender(<size>=0) -> int

Return the number of units from the font's baseline to the top of the bounding box. It is not adjusted for strong or rotation.

get_sized_descender ()

The scaled descent of the font in pixels

get_sized_descender(<size>=0) -> int

Return the number of pixels from the font's baseline to the top of the bounding box. It is not adjusted for strong or rotation.

get_sized_height ()

The scaled height of the font in pixels

get_sized_height(<size>=0) -> int

Returns the height of the font. This is the average value of all glyphs in the font. It is not adjusted for strong or rotation.

get_sized_glyph_height ()

The scaled bounding box height of the font in pixels

get_sized_glyph_height(<size>=0) -> int

Return the glyph bounding box height of the font in pixels. This is the average value of all glyphs in the font. It is not adjusted for strong or rotation.

get_sizes ()

return the available sizes of embedded bitmaps

get_sizes() -> [(int, int, int, float, float), ...]

get_sizes() -> []

Returns a list of tuple records, one for each point size supported. Each tuple containing the point size, the height in pixels, width in pixels, horizontal ppem (nominal width) in fractional pixels, and vertical ppem (nominal height) in fractional pixels.

render ()

Return rendered text as a surface

render(text, fgcolor=None, bgcolor=None, style=STYLE_DEFAULT, rotation=0, size=0) -> (Surface, Rect)

Returns a new **Surface**, with the text rendered to it in the color given by 'fgcolor'. If no foreground color is given, the default foreground color, **fgcolor** is used. If **bgcolor** is given, the surface will be filled with this color. When no background color is given, the surface background is transparent, zero alpha. Normally the returned surface has a 32 bit pixel size. However, if **bgcolor** is **None** and anti-aliasing is disabled a monochrome 8 bit colorkey surface, with colorkey set for the background color, is returned.

The return value is a tuple: the new surface and the bounding rectangle giving the size and origin of the rendered text.

If an empty string is passed for text then the returned Rect is zero width and the height of the font.

Optional *fgcolor*, *style*, *rotation*, and *size* arguments override the default values set for the **Font** instance.

If *text* is a char (byte) string, then its encoding is assumed to be **LATIN1**.

Optionally, *text* can be **None**, which will render the text passed to a previous **get_rect()**, **render()**, **render_to()**, **render_raw()**, or **render_raw_to()** call. See **render_to()** for details.

render_to ()

Render text onto an existing surface

render_to(surf, dest, text, fgcolor=None, bgcolor=None, style=STYLE_DEFAULT, rotation=0, size=0) -> Rect

Renders the string *text* to the **pygame.Surface** *surf*, at position *dest*, a (x, y) surface coordinate pair. If either x or y is not an integer it is converted to one if possible. Any sequence where the first two items are x and y

positional elements is accepted, including a `Rect` instance. As with `render()`, optional *fgcolor*, *style*, *rotation*, and *size* argument are available.

If a background color *bgcolor* is given, the text bounding box is first filled with that color. The text is blitted next. Both the background fill and text rendering involve full alpha blits. That is, the alpha values of the foreground, background, and destination target surface all affect the blit.

The return value is a rectangle giving the size and position of the rendered text within the surface.

If an empty string is passed for text then the returned `Rect` is zero width and the height of the font. The rect will test False.

Optionally, *text* can be set `None`, which will re-render text passed to a previous `render_to()`, `get_rect()`, `render()`, `render_raw()`, or `render_raw_to()` call. Primarily, this feature is an aid to using `render_to()` in combination with `get_rect()`. An example:

```
def word_wrap(surf, text, font, color=(0, 0, 0)):
    font.origin = True
    words = text.split(' ')
    width, height = surf.get_size()
    line_spacing = font.get_sized_height() + 2
    x, y = 0, line_spacing
    space = font.get_rect(' ')
    for word in words:
        bounds = font.get_rect(word)
        if x + bounds.width + bounds.x >= width:
            x, y = 0, y + line_spacing
        if x + bounds.width + bounds.x >= width:
            raise ValueError("word too wide for the surface")
        if y + bounds.height - bounds.y >= height:
            raise ValueError("text too long for the surface")
        font.render_to(surf, (x, y), None, color)
        x += bounds.width + space.width
    return x, y
```

When `render_to()` is called with the same font properties `size`, `style`, `strength`, `wide`, `antialiased`, `vertical`, `rotation`, `kerning`, and `use_bitmap_strikes` as `get_rect()`, `render_to()` will use the layout calculated by `get_rect()`. Otherwise, `render_to()` will recalculate the layout if called with a text string or one of the above properties has changed after the `get_rect()` call.

If *text* is a char (byte) string, then its encoding is assumed to be `LATIN1`.

`render_raw()`

Return rendered text as a string of bytes

`render_raw(text, style=STYLE_DEFAULT, rotation=0, size=0, invert=False) -> (bytes, (int, int))`

Like `render()` but with the pixels returned as a byte string of 8-bit gray-scale values. The foreground color is 255, the background 0, useful as an alpha mask for a foreground pattern.

`render_raw_to()`

Render text into an array of ints

`render_raw_to(array, text, dest=None, style=STYLE_DEFAULT, rotation=0, size=0, invert=False) -> Rect`

Render to an array object exposing an array struct interface. The array must be two dimensional with integer items. The default *dest* value, `None`, is equivalent to position (0, 0). See `render_to()`. As with the other render methods, *text* can be `None` to render a text string passed previously to another method.

The return value is a `pygame.Rect()` giving the size and position of the rendered text.

`style`

The font's style flags

`style -> int`

Gets or sets the default style of the Font. This default style will be used for all text rendering and size calculations unless overridden specifically a render or `get_rect()` call. The style value may be a bit-wise OR of one or more of the following constants:

```
STYLE_NORMAL
STYLE_UNDERLINE
```



```

STYLE_OBLIQUE
STYLE_STRONG
STYLE_WIDE
STYLE_DEFAULT

```

These constants may be found on the FreeType constants module. Optionally, the default style can be modified or obtained accessing the individual style attributes (underline, oblique, strong).

The `STYLE_OBLIQUE` and `STYLE_STRONG` styles are for scalable fonts only. An attempt to set either for a bitmap font raises an `AttributeError`. An attempt to set either for an inactive font, as returned by `Font.__new__()`, raises a `RuntimeError`.

Assigning `STYLE_DEFAULT` to the `style` property leaves the property unchanged, as this property defines the default. The `style` property will never return `STYLE_DEFAULT`.

underline

The state of the font's underline style flag

underline -> bool

Gets or sets whether the font will be underlined when drawing text. This default style value will be used for all text rendering and size calculations unless overridden specifically in a render or `get_rect()` call, via the 'style' parameter.

strong

The state of the font's strong style flag

strong -> bool

Gets or sets whether the font will be bold when drawing text. This default style value will be used for all text rendering and size calculations unless overridden specifically in a render or `get_rect()` call, via the 'style' parameter.

oblique

The state of the font's oblique style flag

oblique -> bool

Gets or sets whether the font will be rendered as oblique. This default style value will be used for all text rendering and size calculations unless overridden specifically in a render or `get_rect()` call, via the 'style' parameter.

The oblique style is only supported for scalable (outline) fonts. An attempt to set this style on a bitmap font will raise an `AttributeError`. If the font object is inactive, as returned by `Font.__new__()`, setting this property raises a `RuntimeError`.

wide

The state of the font's wide style flag

wide -> bool

Gets or sets whether the font will be stretched horizontally when drawing text. It produces a result similar to `pygame.font.Font`'s bold. This style not available for rotated text.

strength

The strength associated with the strong or wide font styles

strength -> float

The amount by which a font glyph's size is enlarged for the strong or wide transformations, as a fraction of the untransformed size. For the wide style only the horizontal dimension is increased. For strong text both the horizontal and vertical dimensions are enlarged. A wide style of strength 0.08333 (1/12) is equivalent to the `pygame.font.Font` bold style. The default is 0.02778 (1/36).

The strength style is only supported for scalable (outline) fonts. An attempt to set this property on a bitmap font will raise an `AttributeError`. If the font object is inactive, as returned by `Font.__new__()`, assignment to this property raises a `RuntimeError`.

underline_adjustment

Adjustment factor for the underline position

underline_adjustment -> float

Gets or sets a factor which, when positive, is multiplied with the font's underline offset to adjust the underline position. A negative value turns an underline into a strike-through or overline. It is multiplied with the ascender. Accepted values range between -2.0 and 2.0 inclusive. A value of 0.5 closely matches Tango underlining. A value of 1.0 mimics `pygame.font.Font` underlining.

fixed_width

Gets whether the font is fixed-width

fixed_width -> bool

Read only. Returns `True` if the font contains fixed-width characters (for example Courier, Bitstream Vera Sans Mono, Andale Mono).

fixed_sizes

the number of available bitmap sizes for the font

fixed_sizes -> int

Read only. Returns the number of point sizes for which the font contains bitmap character images. If zero then the font is not a bitmap font. A scalable font may contain pre-rendered point sizes as strikes.

scalable

Gets whether the font is scalable

scalable -> bool

Read only. Returns `True` if the font contains outline glyphs. If so, the point size is not limited to available bitmap sizes.

use_bitmap_strikes

allow the use of embedded bitmaps in an outline font file

use_bitmap_strikes -> bool

Some scalable fonts include embedded bitmaps for particular point sizes. This property controls whether or not those bitmap strikes are used. Set it `False` to disable the loading of any bitmap strike. Set it `True`, the default, to permit bitmap strikes for a non-rotated render with no style other than **wide** or **underline**. This property is ignored for bitmap fonts.

See also **fixed_sizes** and **get_sizes()**.

antialiased

Font anti-aliasing mode

antialiased -> bool

Gets or sets the font's anti-aliasing mode. This defaults to `True` on all fonts, which are rendered with full 8 bit blending.

Set to `False` to do monochrome rendering. This should provide a small speed gain and reduce cache memory size.

kerning

Character kerning mode

kerning -> bool

Gets or sets the font's kerning mode. This defaults to `False` on all fonts, which will be rendered without kerning.

Set to `True` to add kerning between character pairs, if supported by the font, when positioning glyphs.

vertical

Font vertical mode

vertical -> bool

Gets or sets whether the characters are laid out vertically rather than horizontally. May be useful when rendering Kanji or some other vertical script.

Set to `True` to switch to a vertical text layout. The default is `False`, place horizontally.

Note that the **Font** class does not automatically determine script orientation. Vertical layout must be selected explicitly.

Also note that several font formats (especially bitmap based ones) don't contain the necessary metrics to draw glyphs vertically, so drawing in those cases will give unspecified results.

rotation

text rotation in degrees counterclockwise

rotation -> int

Gets or sets the baseline angle of the rendered text. The angle is represented as integer degrees. The default angle is 0, with horizontal text rendered along the X-axis, and vertical text along the Y-axis. A positive value rotates these axes counterclockwise that many degrees. A negative angle corresponds to a clockwise rotation. The rotation value is normalized to a value within the range 0 to 359 inclusive (eg. 390 -> 390 - 360 -> 30, -45 -> 360 + -45 -> 315, 720 -> 720 - (2 * 360) -> 0).

Only scalable (outline) fonts can be rotated. An attempt to change the rotation of a bitmap font raises an `AttributeError`. An attempt to change the rotation of an inactive font instance, as returned by `Font.__new__()`, raises a `RuntimeError`.

fgcolor

default foreground color

fgcolor -> Color

Gets or sets the default glyph rendering color. It is initially opaque black ■ (0, 0, 0, 255). Applies to `render()` and `render_to()`.

bgcolor

default background color

bgcolor -> Color

Gets or sets the default background rendering color. Initially it is unset and text will render with a transparent background by default. Applies to `render()` and `render_to()`.

New in version 2.0.0: *New in version 2.0.0.*

origin

Font render to text origin mode

origin -> bool

If set `True`, `render_to()` and `render_raw_to()` will take the *dest* position to be that of the text origin, as opposed to the top-left corner of the bounding box. See `get_rect()` for details.

pad

padded boundary mode

pad -> bool

If set `True`, then the text boundary rectangle will be inflated to match that of `font.Font`. Otherwise, the boundary rectangle is just large enough for the text.

ucs4

Enable UCS-4 mode

ucs4 -> bool

Gets or sets the decoding of Unicode text. By default, the freetype module performs UTF-16 surrogate pair decoding on Unicode text. This allows 32-bit escape sequences ('Uxxxxxxx') between 0x10000 and 0x10FFFF to represent their corresponding UTF-32 code points on Python interpreters built with a UCS-2 Unicode type (on Windows, for instance). It also means character values within the UTF-16 surrogate area (0xD800 to 0xDFFF) are considered part of a surrogate pair. A malformed surrogate pair will raise a `UnicodeEncodeError`. Setting `ucs4 True` turns surrogate pair decoding off, allowing access the full UCS-4 character range to a Python interpreter built with four-byte Unicode character support.

resolution

Pixel resolution in dots per inch

resolution -> int

Read only. Gets pixel size used in scaling font glyphs for this `Font` instance.

pygame.gfxdraw

pygame module for drawing shapes

EXPERIMENTAL!: This API may change or disappear in later pygame releases. If you use this, your code may break with the next pygame release.

The pygame package does not import gfxdraw automatically when loaded, so it must be imported explicitly to be used.

```
import pygame
import pygame.gfxdraw
```

For all functions the arguments are strictly positional and integers are accepted for coordinates and radii. The `color` argument can be one of the following formats:

- a `pygame.Color` object
- an (RGB) triplet (tuple/list)
- an (RGBA) quadruplet (tuple/list)

The functions `rectangle()` and `box()` will accept any (x, y, w, h) sequence for their `rect` argument, though `pygame.Rect` instances are preferred.

To draw a filled antialiased shape, first use the antialiased (aa*) version of the function, and then use the filled (filled_*) version. For example:

```
col = (255, 0, 0)
surf.fill((255, 255, 255))
pygame.gfxdraw.aacircle(surf, x, y, 30, col)
pygame.gfxdraw.filled_circle(surf, x, y, 30, col)
```

Note

For threading, each of the functions releases the GIL during the C part of the call.

Note

See the `pygame.draw` module for alternative draw methods. The `pygame.gfxdraw` module differs from the `pygame.draw` module in the API it uses and the different draw functions available. `pygame.gfxdraw` wraps the primitives from the library called `SDL_gfx`, rather than using modified versions.

New in version 1.9.0: *New in version 1.9.0.*

`pygame.gfxdraw.pixel()`

draw a pixel

`pixel(surface, x, y, color) -> None`

Draws a single pixel, at position (x, y), on the given surface.

Parameters:

- **surface** (*Surface*) -- surface to draw on
- **x** (*int*) -- x coordinate of the pixel
- **y** (*int*) -- y coordinate of the pixel
- **color** (*Color* or *tuple(int, int, int, [int])*) -- color to draw with, the alpha value is optional if using a tuple (RGB[A])

Returns: None

Return type: NoneType

`pygame.gfxdraw.hline()`

draw a horizontal line

`hline(surface, x1, x2, y, color) -> None`

Draws a straight horizontal line ((x1, y) to (x2, y)) on the given surface. There are no endcaps.

Parameters:

- **surface** (*Surface*) -- surface to draw on
- **x1** (*int*) -- x coordinate of one end of the line
- **x2** (*int*) -- x coordinate of the other end of the line
- **y** (*int*) -- y coordinate of the line
- **color** (*Color* or *tuple(int, int, int, [int])*) -- color to draw with, the alpha value is optional if using a tuple (*RGB[A]*)

Returns: None**Return type:** NoneType`pygame.gfxdraw.vline()`

draw a vertical line

`vline(surface, x, y1, y2, color) -> None`Draws a straight vertical line ((*x*, *y1*) to (*x*, *y2*)) on the given surface. There are no endcaps.**Parameters:**

- **surface** (*Surface*) -- surface to draw on
- **x** (*int*) -- x coordinate of the line
- **y1** (*int*) -- y coordinate of one end of the line
- **y2** (*int*) -- y coordinate of the other end of the line
- **color** (*Color* or *tuple(int, int, int, [int])*) -- color to draw with, the alpha value is optional if using a tuple (*RGB[A]*)

Returns: None**Return type:** NoneType`pygame.gfxdraw.line()`

draw a line

`line(surface, x1, y1, x2, y2, color) -> None`Draws a straight line ((*x1*, *y1*) to (*x2*, *y2*)) on the given surface. There are no endcaps.**Parameters:**

- **surface** (*Surface*) -- surface to draw on
- **x1** (*int*) -- x coordinate of one end of the line
- **y1** (*int*) -- y coordinate of one end of the line
- **x2** (*int*) -- x coordinate of the other end of the line
- **y2** (*int*) -- y coordinate of the other end of the line
- **color** (*Color* or *tuple(int, int, int, [int])*) -- color to draw with, the alpha value is optional if using a tuple (*RGB[A]*)

Returns: None**Return type:** NoneType`pygame.gfxdraw.rectangle()`

draw a rectangle

`rectangle(surface, rect, color) -> None`Draws an unfilled rectangle on the given surface. For a filled rectangle use `box()`.**Parameters:**

- **surface** (*Surface*) -- surface to draw on
- **rect** (*Rect*) -- rectangle to draw, position and dimensions
- **color** (*Color* or *tuple(int, int, int, [int])*) -- color to draw with, the alpha value is optional if using a tuple (*RGB[A]*)

Returns: None**Return type:** NoneType

Note

The `rect.bottom` and `rect.right` attributes of a `pygame.Rect` always lie one pixel outside of its actual border. Therefore, these values will not be included as part of the drawing.

`pygame.gfxdraw.box()`

draw a filled rectangle

`box(surface, rect, color) -> None`

Draws a filled rectangle on the given surface. For an unfilled rectangle use `rectangle()`.

Parameters:

- **surface** (*Surface*) -- surface to draw on
- **rect** (*Rect*) -- rectangle to draw, position and dimensions
- **color** (*Color* or *tuple(int, int, int, [int])*) -- color to draw with, the alpha value is optional if using a tuple (`RGB[A]`)

Returns: `None`

Return type: `NoneType`

Note

The `rect.bottom` and `rect.right` attributes of a `pygame.Rect` always lie one pixel outside of its actual border. Therefore, these values will not be included as part of the drawing.

Note

The `pygame.Surface.fill()` method works just as well for drawing filled rectangles. In fact `pygame.Surface.fill()` can be hardware accelerated on some platforms with both software and hardware display modes.

`pygame.gfxdraw.circle()`

draw a circle

`circle(surface, x, y, r, color) -> None`

Draws an unfilled circle on the given surface. For a filled circle use `filled_circle()`.

Parameters:

- **surface** (*Surface*) -- surface to draw on
- **x** (*int*) -- x coordinate of the center of the circle
- **y** (*int*) -- y coordinate of the center of the circle
- **r** (*int*) -- radius of the circle
- **color** (*Color* or *tuple(int, int, int, [int])*) -- color to draw with, the alpha value is optional if using a tuple (`RGB[A]`)

Returns: `None`

Return type: `NoneType`

`pygame.gfxdraw.aacircle()`

draw an antialiased circle

`aacircle(surface, x, y, r, color) -> None`

Draws an unfilled antialiased circle on the given surface.

Parameters:

- **surface** (*Surface*) -- surface to draw on
- **x** (*int*) -- x coordinate of the center of the circle
- **y** (*int*) -- y coordinate of the center of the circle
- **r** (*int*) -- radius of the circle
- **color** (*Color* or *tuple(int, int, int, [int])*) -- color to draw with, the alpha value is optional if using a tuple (*RGB[A]*)

Returns: None**Return type:** NoneType`pygame.gfxdraw.filled_circle()`

draw a filled circle

`filled_circle(surface, x, y, r, color) -> None`Draws a filled circle on the given surface. For an unfilled circle use `circle()`.**Parameters:**

- **surface** (*Surface*) -- surface to draw on
- **x** (*int*) -- x coordinate of the center of the circle
- **y** (*int*) -- y coordinate of the center of the circle
- **r** (*int*) -- radius of the circle
- **color** (*Color* or *tuple(int, int, int, [int])*) -- color to draw with, the alpha value is optional if using a tuple (*RGB[A]*)

Returns: None**Return type:** NoneType`pygame.gfxdraw.ellipse()`

draw an ellipse

`ellipse(surface, x, y, rx, ry, color) -> None`Draws an unfilled ellipse on the given surface. For a filled ellipse use `filled_ellipse()`.**Parameters:**

- **surface** (*Surface*) -- surface to draw on
- **x** (*int*) -- x coordinate of the center of the ellipse
- **y** (*int*) -- y coordinate of the center of the ellipse
- **rx** (*int*) -- horizontal radius of the ellipse
- **ry** (*int*) -- vertical radius of the ellipse
- **color** (*Color* or *tuple(int, int, int, [int])*) -- color to draw with, the alpha value is optional if using a tuple (*RGB[A]*)

Returns: None**Return type:** NoneType`pygame.gfxdraw.aaellipse()`

draw an antialiased ellipse

`aaellipse(surface, x, y, rx, ry, color) -> None`

Draws an unfilled antialiased ellipse on the given surface.

Parameters:

- **surface** (*Surface*) -- surface to draw on
- **x** (*int*) -- x coordinate of the center of the ellipse
- **y** (*int*) -- y coordinate of the center of the ellipse
- **rx** (*int*) -- horizontal radius of the ellipse
- **ry** (*int*) -- vertical radius of the ellipse
- **color** (*Color* or *tuple(int, int, int, [int])*) -- color to draw with, the alpha value is optional if using a tuple (*RGB[A]*)

Returns: None**Return type:** NoneType`pygame.gfxdraw.filled_ellipse()`

draw a filled ellipse

`filled_ellipse(surface, x, y, rx, ry, color) -> None`Draws a filled ellipse on the given surface. For an unfilled ellipse use `ellipse()`.**Parameters:**

- **surface** (*Surface*) -- surface to draw on
- **x** (*int*) -- x coordinate of the center of the ellipse
- **y** (*int*) -- y coordinate of the center of the ellipse
- **rx** (*int*) -- horizontal radius of the ellipse
- **ry** (*int*) -- vertical radius of the ellipse
- **color** (*Color* or *tuple(int, int, int, [int])*) -- color to draw with, the alpha value is optional if using a tuple (*RGB[A]*)

Returns: None**Return type:** NoneType`pygame.gfxdraw.arc()`

draw an arc

`arc(surface, x, y, r, start_angle, stop_angle, color) -> None`Draws an arc on the given surface. For an arc with its endpoints connected to its center use `pie()`.The two angle arguments are given in degrees and indicate the start and stop positions of the arc. The arc is drawn in a clockwise direction from the `start_angle` to the `stop_angle`. If `start_angle == stop_angle`, nothing will be drawn**Parameters:**

- **surface** (*Surface*) -- surface to draw on
- **x** (*int*) -- x coordinate of the center of the arc
- **y** (*int*) -- y coordinate of the center of the arc
- **r** (*int*) -- radius of the arc
- **start_angle** (*int*) -- start angle in degrees
- **stop_angle** (*int*) -- stop angle in degrees
- **color** (*Color* or *tuple(int, int, int, [int])*) -- color to draw with, the alpha value is optional if using a tuple (*RGB[A]*)

Returns: None**Return type:** NoneType**Note**This function uses *degrees* while the `pygame.draw.arc()` function uses *radians*.`pygame.gfxdraw.pie()`

draw a pie

`pie(surface, x, y, r, start_angle, stop_angle, color) -> None`

Draws an unfilled pie on the given surface. A pie is an `arc()` with its endpoints connected to its center.

The two angle arguments are given in degrees and indicate the start and stop positions of the pie. The pie is drawn in a clockwise direction from the `start_angle` to the `stop_angle`. If `start_angle == stop_angle`, a straight line will be drawn from the center position at the given angle, to a length of the radius.

Parameters:

- **surface** (*Surface*) -- surface to draw on
- **x** (*int*) -- x coordinate of the center of the pie
- **y** (*int*) -- y coordinate of the center of the pie
- **r** (*int*) -- radius of the pie
- **start_angle** (*int*) -- start angle in degrees
- **stop_angle** (*int*) -- stop angle in degrees
- **color** (*Color* or *tuple(int, int, int, [int])*) -- color to draw with, the alpha value is optional if using a tuple (`RGB[A]`)

Returns: None

Return type: NoneType

`pygame.gfxdraw.trigon()`

draw a trigon/triangle

`trigon(surface, x1, y1, x2, y2, x3, y3, color) -> None`

Draws an unfilled trigon (triangle) on the given surface. For a filled trigon use `filled_trigon()`.

A trigon can also be drawn using `polygon()` e.g.
`polygon(surface, ((x1, y1), (x2, y2), (x3, y3)), color)`

Parameters:

- **surface** (*Surface*) -- surface to draw on
- **x1** (*int*) -- x coordinate of the first corner of the trigon
- **y1** (*int*) -- y coordinate of the first corner of the trigon
- **x2** (*int*) -- x coordinate of the second corner of the trigon
- **y2** (*int*) -- y coordinate of the second corner of the trigon
- **x3** (*int*) -- x coordinate of the third corner of the trigon
- **y3** (*int*) -- y coordinate of the third corner of the trigon
- **color** (*Color* or *tuple(int, int, int, [int])*) -- color to draw with, the alpha value is optional if using a tuple (`RGB[A]`)

Returns: None

Return type: NoneType

`pygame.gfxdraw.aatrigon()`

draw an antialiased trigon/triangle

`aatrigon(surface, x1, y1, x2, y2, x3, y3, color) -> None`

Draws an unfilled antialiased trigon (triangle) on the given surface.

An aatrigon can also be drawn using `aapolygon()` e.g.
`aapolygon(surface, ((x1, y1), (x2, y2), (x3, y3)), color)`

Parameters:

- **surface** (*Surface*) -- surface to draw on
- **x1** (*int*) -- x coordinate of the first corner of the trigon
- **y1** (*int*) -- y coordinate of the first corner of the trigon
- **x2** (*int*) -- x coordinate of the second corner of the trigon
- **y2** (*int*) -- y coordinate of the second corner of the trigon
- **x3** (*int*) -- x coordinate of the third corner of the trigon
- **y3** (*int*) -- y coordinate of the third corner of the trigon
- **color** (*Color* or *tuple(int, int, int, [int])*) -- color to draw with, the alpha value is optional if using a tuple (*RGB[A]*)

Returns: None**Return type:** NoneTypepygame.gfxdraw.**filled_trigon**()

draw a filled trigon/triangle

filled_trigon(surface, x1, y1, x2, y2, x3, y3, color) -> None

Draws a filled trigon (triangle) on the given surface. For an unfilled trigon use **trigon()**.A filled_trigon can also be drawn using **filled_polygon()** e.g.

filled_polygon(surface, ((x1, y1), (x2, y2), (x3, y3)), color)

Parameters:

- **surface** (*Surface*) -- surface to draw on
- **x1** (*int*) -- x coordinate of the first corner of the trigon
- **y1** (*int*) -- y coordinate of the first corner of the trigon
- **x2** (*int*) -- x coordinate of the second corner of the trigon
- **y2** (*int*) -- y coordinate of the second corner of the trigon
- **x3** (*int*) -- x coordinate of the third corner of the trigon
- **y3** (*int*) -- y coordinate of the third corner of the trigon
- **color** (*Color* or *tuple(int, int, int, [int])*) -- color to draw with, the alpha value is optional if using a tuple (*RGB[A]*)

Returns: None**Return type:** NoneTypepygame.gfxdraw.**polygon**()

draw a polygon

polygon(surface, points, color) -> None

Draws an unfilled polygon on the given surface. For a filled polygon use **filled_polygon()**.The adjacent coordinates in the **points** argument, as well as the first and last points, will be connected by line segments. e.g. For the points [(x1, y1), (x2, y2), (x3, y3)] a line segment will be drawn from (x1, y1) to (x2, y2), from (x2, y2) to (x3, y3), and from (x3, y3) to (x1, y1).**Parameters:**

- **surface** (*Surface*) -- surface to draw on
- **points** (*tuple(coordinate)* or *list(coordinate)*) -- a sequence of 3 or more (x, y) coordinates, where each *coordinate* in the sequence must be a tuple/list/**pygame.math.Vector2** of 2 ints/floats (float values will be truncated)
- **color** (*Color* or *tuple(int, int, int, [int])*) -- color to draw with, the alpha value is optional if using a tuple (*RGB[A]*)

Returns: None**Return type:** NoneType**Raises:**

- **ValueError** -- if `len(points) < 3` (must have at least 3 points)
- **IndexError** -- if `len(coordinate) < 2` (each coordinate must have at least 2 items)

`pygame.gfxdraw.aapolygon()`

draw an antialiased polygon

`aapolygon(surface, points, color)` -> None

Draws an unfilled antialiased polygon on the given surface.

The adjacent coordinates in the `points` argument, as well as the first and last points, will be connected by line segments. e.g. For the points `[(x1, y1), (x2, y2), (x3, y3)]` a line segment will be drawn from `(x1, y1)` to `(x2, y2)`, from `(x2, y2)` to `(x3, y3)`, and from `(x3, y3)` to `(x1, y1)`.

Parameters:

- **surface** (*Surface*) -- surface to draw on
- **points** (*tuple(coordinate)* or *list(coordinate)*) -- a sequence of 3 or more (x, y) coordinates, where each *coordinate* in the sequence must be a tuple/list/`pygame.math.Vector2` of 2 ints/floats (float values will be truncated)
- **color** (*Color* or *tuple(int, int, int, [int])*) -- color to draw with, the alpha value is optional if using a tuple (`RGB[A]`)

Returns: None

Return type: NoneType

Raises:

- **ValueError** -- if `len(points) < 3` (must have at least 3 points)
- **IndexError** -- if `len(coordinate) < 2` (each coordinate must have at least 2 items)

`pygame.gfxdraw.filled_polygon()`

draw a filled polygon

`filled_polygon(surface, points, color)` -> None

Draws a filled polygon on the given surface. For an unfilled polygon use `polygons()`.

The adjacent coordinates in the `points` argument, as well as the first and last points, will be connected by line segments. e.g. For the points `[(x1, y1), (x2, y2), (x3, y3)]` a line segment will be drawn from `(x1, y1)` to `(x2, y2)`, from `(x2, y2)` to `(x3, y3)`, and from `(x3, y3)` to `(x1, y1)`.

Parameters:

- **surface** (*Surface*) -- surface to draw on
- **points** (*tuple(coordinate)* or *list(coordinate)*) -- a sequence of 3 or more (x, y) coordinates, where each *coordinate* in the sequence must be a tuple/list/`pygame.math.Vector2` of 2 ints/floats (float values will be truncated)
- **color** (*Color* or *tuple(int, int, int, [int])*) -- color to draw with, the alpha value is optional if using a tuple (`RGB[A]`)

Returns: None

Return type: NoneType

Raises:

- **ValueError** -- if `len(points) < 3` (must have at least 3 points)
- **IndexError** -- if `len(coordinate) < 2` (each coordinate must have at least 2 items)

`pygame.gfxdraw.textured_polygon()`

draw a textured polygon

`textured_polygon(surface, points, texture, tx, ty)` -> None

Draws a textured polygon on the given surface. For better performance, the surface and the texture should have the same format.

A per-pixel alpha texture blit to a per-pixel alpha surface will differ from a `pygame.Surface.blit()` blit. Also, a per-pixel alpha texture cannot be used with an 8-bit per pixel destination.

The adjacent coordinates in the `points` argument, as well as the first and last points, will be connected by line segments. e.g. For the points `[(x1, y1), (x2, y2), (x3, y3)]` a line segment will be drawn from `(x1, y1)` to `(x2, y2)`, from `(x2, y2)` to `(x3, y3)`, and from `(x3, y3)` to `(x1, y1)`.

Parameters:

- **surface** (*Surface*) -- surface to draw on
- **points** (*tuple(coordinate) or list(coordinate)*) -- a sequence of 3 or more (x, y) coordinates, where each *coordinate* in the sequence must be a tuple/list/`pygame.math.Vector2` of 2 ints/floats (float values will be truncated)
- **texture** (*Surface*) -- texture to draw on the polygon
- **tx** (*int*) -- x offset of the texture
- **ty** (*int*) -- y offset of the texture

Returns: None**Return type:** NoneType**Raises:**

- **ValueError** -- if `len(points) < 3` (must have at least 3 points)
- **IndexError** -- if `len(coordinate) < 2` (each coordinate must have at least 2 items)

`pygame.gfxdraw.bezier()`

draw a Bezier curve

`bezier(surface, points, steps, color) -> None`

Draws a Bézier curve on the given surface.

Parameters:

- **surface** (*Surface*) -- surface to draw on
- **points** (*tuple(coordinate) or list(coordinate)*) -- a sequence of 3 or more (x, y) coordinates used to form a curve, where each *coordinate* in the sequence must be a tuple/list/`pygame.math.Vector2` of 2 ints/floats (float values will be truncated)
- **steps** (*int*) -- number of steps for the interpolation, the minimum is 2
- **color** (*Color or tuple(int, int, int, [int])*) -- color to draw with, the alpha value is optional if using a tuple (`RGB[A]`)

Returns: None**Return type:** NoneType**Raises:**

- **ValueError** -- if `steps < 2`
- **ValueError** -- if `len(points) < 3` (must have at least 3 points)
- **IndexError** -- if `len(coordinate) < 2` (each coordinate must have at least 2 items)

pygame.image

pygame module for image transfer

The image module contains functions for loading and saving pictures, as well as transferring Surfaces to formats usable by other packages.

Note that there is no Image class; an image is loaded as a Surface object. The Surface class allows manipulation (drawing lines, setting pixels, capturing regions, etc.).

The image module is a required dependency of pygame, but it only optionally supports any extended file formats. By default it can only load uncompressed BMP images. When built with full image support, the `pygame.image.load()` function can support the following formats.

- JPG
- PNG
- GIF (non-animated)
- BMP
- PCX
- TGA (uncompressed)

- TIF
- LBM (and PBM)
- PBM (and PGM, PPM)
- XPM

Saving images only supports a limited set of formats. You can save to the following formats.

- BMP
- TGA
- PNG
- JPEG

JPEG and JPG refer to the same file format

New in version 1.8: *New in version 1.8*: Saving PNG and JPEG files.

`pygame.image.load_basic()`

load new BMP image from a file (or file-like object)

`load_basic(file)` -> Surface

Load an image from a file source. You can pass either a filename or a Python file-like object.

This function only supports loading "basic" image format, ie BMP format. This function is always available, no matter how pygame was built.

`pygame.image.load()`

load new image from a file (or file-like object)

`load(filename)` -> Surface

`load(fileobj, namehint="")` -> Surface

Load an image from a file source. You can pass either a filename or a Python file-like object.

Pygame will automatically determine the image type (e.g., GIF or bitmap) and create a new Surface object from the data. In some cases it will need to know the file extension (e.g., GIF images should end in ".gif"). If you pass a raw file-like object, you may also want to pass the original filename as the namehint argument.

The returned Surface will contain the same color format, colorkey and alpha transparency as the file it came from. You will often want to call `Surface.convert()` with no arguments, to create a copy that will draw more quickly on the screen.

For alpha transparency, like in .png images, use the `convert_alpha()` method after loading so that the image has per pixel transparency.

pygame may not always be built to support all image formats. At minimum it will support uncompressed BMP. If `pygame.image.get_extended()` returns 'True', you should be able to load most images (including PNG, JPG and GIF).

You should use `os.path.join()` for compatibility.

```
eg. asurf = pygame.image.load(os.path.join('data', 'bla.png'))
```

`pygame.image.load_extended()`

load an image from a file (or file-like object)

`load_extended(filename)` -> Surface

`load_extended(fileobj, namehint="")` -> Surface

This function is similar to `pygame.image.load()`, except that this function can only be used if pygame was built with extended image format support.

From version 2.0.1, this function is always available, but raises an error if extended image formats are not supported. Previously, this function may or may not be available, depending on the state of extended image format support.

Changed in version 2.0.1: *Changed in version 2.0.1*.

`pygame.image.save()`

save an image to file (or file-like object)

`save(Surface, filename)` -> None

`save(Surface, fileobj, namehint="")` -> None

This will save your Surface as either a BMP, TGA, PNG, or JPEG image. If the filename extension is unrecognized it will default to TGA. Both TGA, and BMP file formats create uncompressed files. You can pass a filename or a Python file-like object. For file-like object, the image is saved to TGA format unless a namehint with a recognizable extension is passed in.

Note

To be able to save the JPEG file format to a file-like object, SDL2_Image version 2.0.2 or newer is needed.

Note

When saving to a file-like object, it seems that for most formats, the object needs to be flushed after saving to it to make loading from it possible.

Changed in version 1.8: *Changed in version 1.8: Saving PNG and JPEG files.*

Changed in version 2.0.0.dev11: *Changed in version 2.0.0.dev11: The namehint parameter was added to make it possible to save other formats than TGA to a file-like object.*

`pygame.image.save_extended()`

save a png/jpg image to file (or file-like object)

`save_extended(Surface, filename) -> None`

`save_extended(Surface, fileobj, namehint="") -> None`

This will save your Surface as either a PNG or JPEG image.

Incase the image is being saved to a file-like object, this function uses the namehint argument to determine the format of the file being saved. Saves to JPEG incase the namehint was not specified while saving to file-like object. From version 2.0.1, this function is always available, but raises an error if extended image formats are not supported. Previously, this function may or may not be available, depending on the state of extended image format support.

Changed in version 2.0.1: *Changed in version 2.0.1.*

`pygame.image.get_sdl_image_version()`

get version number of the SDL_Image library being used

`get_sdl_image_version() -> None`

`get_sdl_image_version() -> (major, minor, patch)`

If pygame is built with extended image formats, then this function will return the SDL_Image library's version number as a tuple of 3 integers (major, minor, patch). If not, then it will return None.

New in version 2.0.0.dev11: *New in version 2.0.0.dev11.*

`pygame.image.get_extended()`

test if extended image formats can be loaded

`get_extended() -> bool`

If pygame is built with extended image formats this function will return True. It is still not possible to determine which formats will be available, but generally you will be able to load them all.

`pygame.image.tostring()`

transfer image to string buffer

`tostring(Surface, format, flipped=False) -> string`

Creates a string that can be transferred with the 'fromstring' method in other Python imaging packages. Some Python image packages prefer their images in bottom-to-top format (PyOpenGL for example). If you pass True for the flipped argument, the string buffer will be vertically flipped.

The format argument is a string of one of the following values. Note that only 8-bit Surfaces can use the "P" format. The other formats will work for any Surface. Also note that other Python image packages support more formats than pygame.

- P, 8-bit palettized Surfaces
- RGB, 24-bit image
- RGBX, 32-bit image with unused space

- RGBA, 32-bit image with an alpha channel
- ARGB, 32-bit image with alpha channel first
- RGBA_PREMULT, 32-bit image with colors scaled by alpha channel
- ARGB_PREMULT, 32-bit image with colors scaled by alpha channel, alpha channel first

`pygame.image.fromstring()`

create new Surface from a string buffer

`fromstring(string, size, format, flipped=False) -> Surface`

This function takes arguments similar to `pygame.image.tostring()`. The size argument is a pair of numbers representing the width and height. Once the new Surface is created you can destroy the string buffer.

The size and format image must compute the exact same size as the passed string buffer. Otherwise an exception will be raised.

See the `pygame.image.frombuffer()` method for a potentially faster way to transfer images into pygame.

`pygame.image.frombuffer()`

create a new Surface that shares data inside a bytes buffer

`frombuffer(bytes, size, format) -> Surface`

Create a new Surface that shares pixel data directly from a bytes buffer. This method takes similar arguments to `pygame.image.fromstring()`, but is unable to vertically flip the source data.

This will run much faster than `pygame.image.fromstring()`, since no pixel data must be allocated and copied. It accepts the following 'format' arguments:

- P, 8-bit palettized Surfaces
- RGB, 24-bit image
- BGR, 24-bit image, red and blue channels swapped.
- RGBX, 32-bit image with unused space
- RGBA, 32-bit image with an alpha channel
- ARGB, 32-bit image with alpha channel first

pygame.joystick

Pygame module for interacting with joysticks, gamepads, and trackballs.

The joystick module manages the joystick devices on a computer. Joystick devices include trackballs and video-game-style gamepads, and the module allows the use of multiple buttons and "hats". Computers may manage multiple joysticks at a time.

Each instance of the Joystick class represents one gaming device plugged into the computer. If a gaming pad has multiple joysticks on it, then the joystick object can actually represent multiple joysticks on that single game device.

For a quick way to initialise the joystick module and get a list of Joystick instances use the following code:

```
pygame.joystick.init()
joysticks = [pygame.joystick.Joystick(x) for x in range(pygame.joystick.get_count())]
```

The following event types will be generated by the joysticks

```
JOYAXISMOTION JOYBALLMOTION JOYBUTTONDOWN JOYBUTTONUP JOYHATMOTION
```

And in pygame 2, which supports hotplugging:

```
JOYDEVICEADDED JOYDEVICEREMOVED
```

Note that in pygame 2, joysticks events use a unique "instance ID". The device index passed in the constructor to a Joystick object is not unique after devices have been added and removed. You must call `Joystick.get_instance_id()` to find the instance ID that was assigned to a Joystick on opening.

The event queue needs to be pumped frequently for some of the methods to work. So call one of `pygame.event.get`, `pygame.event.wait`, or `pygame.event.pump` regularly.

`pygame.joystick.init()`

Initialize the joystick module.

`init()` -> None

This function is called automatically by `pygame.init()`.

It initializes the joystick module. The module must be initialized before any other functions will work.

It is safe to call this function more than once.

`pygame.joystick.quit()`

Uninitialize the joystick module.

`quit()` -> None

Uninitialize the joystick module. After you call this any existing joystick objects will no longer work.

It is safe to call this function more than once.

`pygame.joystick.get_init()`

Returns True if the joystick module is initialized.

`get_init()` -> bool

Test if the `pygame.joystick.init()` function has been called.

`pygame.joystick.get_count()`

Returns the number of joysticks.

`get_count()` -> count

Return the number of joystick devices on the system. The count will be 0 if there are no joysticks on the system.

When you create Joystick objects using `Joystick(id)`, you pass an integer that must be lower than this count.

`pygame.joystick.Joystick`

Create a new Joystick object.

`Joystick(id)` -> Joystick

Create a new joystick to access a physical device. The `id` argument must be a value from 0 to

`pygame.joystick.get_count() - 1`.

Joysticks are initialised on creation and are shut down when deallocated. Once the device is initialized the pygame event queue will start receiving events about its input.

Changed in version 2.0.0: *Changed in version 2.0.0:* Joystick objects are now opened immediately on creation.

`init()`

initialize the Joystick

`init()` -> None

Initialize the joystick, if it has been closed. It is safe to call this even if the joystick is already initialized.

Deprecated since version 2.0.0: *Deprecated since version 2.0.0:* In future it will not be possible to reinitialise a closed Joystick object. Will be removed in Pygame 2.1.

`quit()`

uninitialize the Joystick

`quit()` -> None

Close a Joystick object. After this the pygame event queue will no longer receive events from the device.

It is safe to call this more than once.

`get_init()`

check if the Joystick is initialized

`get_init()` -> bool

Return True if the Joystick object is currently initialised.

`get_id()`

get the device index (deprecated)

`get_id()` -> int

Returns the original device index for this device. This is the same value that was passed to the `Joystick()` constructor. This method can safely be called while the Joystick is not initialized.

Deprecated since version 2.0.0: *Deprecated since version 2.0.0:* The original device index is not useful in pygame 2. Use `get_instance_id()` instead. Will be removed in Pygame 2.1.

get_instance_id () → int

get the joystick instance id

get_instance_id() -> int

Get the joystick instance ID. This matches the `instance_id` field that is given in joystick events.

New in version 2.0.0dev11: *New in version 2.0.0dev11.*

get_guid () → str

get the joystick GUID

get_guid() -> str

Get the GUID string. This identifies the exact hardware of the joystick device.

New in version 2.0.0dev11: *New in version 2.0.0dev11.*

get_power_level () → str

get the approximate power status of the device

get_power_level() -> str

Get a string giving the power status of the device.

One of: `empty`, `low`, `medium`, `full`, `wired`, `max`, or `unknown`.

New in version 2.0.0dev11: *New in version 2.0.0dev11.*

get_name ()

get the Joystick system name

get_name() -> string

Returns the system name for this joystick device. It is unknown what name the system will give to the Joystick, but it should be a unique name that identifies the device. This method can safely be called while the Joystick is not initialized.

get_numaxes ()

get the number of axes on a Joystick

get_numaxes() -> int

Returns the number of input axes are on a Joystick. There will usually be two for the position. Controls like rudders and throttles are treated as additional axes.

The `pygame.JOYAXISMOTION` events will be in the range from `-1.0` to `1.0`. A value of `0.0` means the axis is centered. Gamepad devices will usually be `-1`, `0`, or `1` with no values in between. Older analog joystick axes will not always use the full `-1` to `1` range, and the centered value will be some area around `0`.

Analog joysticks usually have a bit of noise in their axis, which will generate a lot of rapid small motion events.

get_axis ()

get the current position of an axis

get_axis(axis_number) -> 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.

The axis number must be an integer from `0` to `get_numaxes() - 1`.

When using gamepads both the control sticks and the analog triggers are usually reported as axes.

get_numballs ()

get the number of trackballs on a Joystick

get_numballs() -> int

Returns the number of trackball devices on a Joystick. These devices work similar to a mouse but they have no absolute position; they only have relative amounts of movement.

The `pygame.JOYBALLMOTION` event will be sent when the trackball is rolled. It will report the amount of movement on the trackball.

get_ball ()

get the relative position of a trackball

get_ball(ball_number) -> 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`.

The ball number must be an integer from 0 to `get_numballs()` - 1.

`get_numbuttons()`

get the number of buttons on a Joystick

`get_numbuttons()` -> int

Returns the number of pushable buttons on the joystick. These buttons have a boolean (on or off) state.

Buttons generate a `pygame.JOYBUTTONDOWN` and `pygame.JOYBUTTONUP` event when they are pressed and released.

`get_button()`

get the current button state

`get_button(button)` -> bool

Returns the current state of a joystick button.

`get_numhats()`

get the number of hat controls on a Joystick

`get_numhats()` -> int

Returns the number of joystick hats on a Joystick. Hat devices are like miniature digital joysticks on a joystick. Each hat has two axes of input.

The `pygame.JOYHATMOTION` event is generated when the hat changes position. The `position` attribute for the event contains a pair of values that are either -1, 0, or 1. A position of (0, 0) means the hat is centered.

`get_hat()`

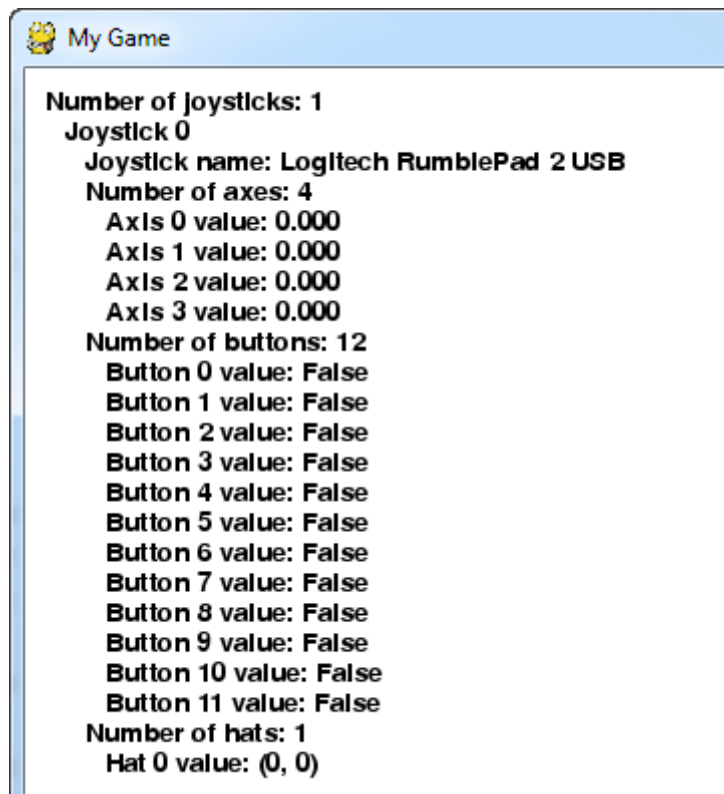
get the position of a joystick hat

`get_hat(hat_number)` -> 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 and a value of 1 means right/up: so (-1, 0) means left; (1, 0) means right; (0, 1) means up; (1, 1) means upper-right; etc.

This value is digital, i.e., each coordinate can be -1, 0 or 1 but never in-between.

The hat number must be between 0 and `get_numhats()` - 1.



Example code for joystick module.

```

import pygame

# Define some colors.
BLACK = pygame.Color('black')
WHITE = pygame.Color('white')

# This is a simple class that will help us print to the screen.
# It has nothing to do with the joysticks, just outputting the
# information.
class TextPrint(object):
    def __init__(self):
        self.reset()
        self.font = pygame.font.Font(None, 20)

    def tprint(self, screen, textString):
        textBitmap = self.font.render(textString, True, BLACK)
        screen.blit(textBitmap, (self.x, self.y))
        self.y += self.line_height

    def reset(self):
        self.x = 10
        self.y = 10
        self.line_height = 15

    def indent(self):
        self.x += 10

    def unindent(self):
        self.x -= 10

pygame.init()

# Set the width and height of the screen (width, height).
screen = pygame.display.set_mode((500, 700))

pygame.display.set_caption("My Game")

# Loop until the user clicks the close button.
done = False

# Used to manage how fast the screen updates.
clock = pygame.time.Clock()

# Initialize the joysticks.
pygame.joystick.init()

# Get ready to print.
textPrint = TextPrint()

# ----- Main Program Loop -----
while not done:
    #
    # EVENT PROCESSING STEP
    #
    # Possible joystick actions: JOYAXISMOTION, JOYBALLMOTION, JOYBUTTONDOWN,

```

```

# JOYBUTTONUP, JOYHATMOTION
for event in pygame.event.get(): # User did something.
    if event.type == pygame.QUIT: # If user clicked close.
        done = True # Flag that we are done so we exit this loop.
    elif event.type == pygame.JOYBUTTONDOWN:
        print("Joystick button pressed.")
    elif event.type == pygame.JOYBUTTONUP:
        print("Joystick button released.")

#
# DRAWING STEP
#
# First, clear the screen to white. Don't put other drawing commands
# above this, or they will be erased with this command.
screen.fill(WHITE)
textPrint.reset()

# Get count of joysticks.
joystick_count = pygame.joystick.get_count()

textPrint.tprint(screen, "Number of joysticks: {}".format(joystick_count))
textPrint.indent()

# For each joystick:
for i in range(joystick_count):
    joystick = pygame.joystick.Joystick(i)
    joystick.init()

    try:
        jid = joystick.get_instance_id()
    except AttributeError:
        # get_instance_id() is an SDL2 method
        jid = joystick.get_id()
    textPrint.tprint(screen, "Joystick {}".format(jid))
    textPrint.indent()

    # Get the name from the OS for the controller/joystick.
    name = joystick.get_name()
    textPrint.tprint(screen, "Joystick name: {}".format(name))

    try:
        guid = joystick.get_guid()
    except AttributeError:
        # get_guid() is an SDL2 method
        pass
    else:
        textPrint.tprint(screen, "GUID: {}".format(guid))

    # Usually axis run in pairs, up/down for one, and left/right for
    # the other.
    axes = joystick.get_numaxes()
    textPrint.tprint(screen, "Number of axes: {}".format(axes))
    textPrint.indent()

    for i in range(axes):
        axis = joystick.get_axis(i)
        textPrint.tprint(screen, "Axis {} value: {:>6.3f}".format(i, axis))
    textPrint.unindent()

    buttons = joystick.get_numbuttons()

```

```

textPrint.tprint(screen, "Number of buttons: {}".format(buttons))
textPrint.indent()

for i in range(buttons):
    button = joystick.get_button(i)
    textPrint.tprint(screen,
                      "Button {:>2} value: {}".format(i, button))
textPrint.unindent()

hats = joystick.get_numhats()
textPrint.tprint(screen, "Number of hats: {}".format(hats))
textPrint.indent()

# Hat position. All or nothing for direction, not a float like
# get_axis(). Position is a tuple of int values (x, y).
for i in range(hats):
    hat = joystick.get_hat(i)
    textPrint.tprint(screen, "Hat {} value: {}".format(i, str(hat)))
textPrint.unindent()

textPrint.unindent()

#
# ALL CODE TO DRAW SHOULD GO ABOVE THIS COMMENT
#

# Go ahead and update the screen with what we've drawn.
pygame.display.flip()

# Limit to 20 frames per second.
clock.tick(20)

# Close the window and quit.
# If you forget this line, the program will 'hang'
# on exit if running from IDLE.
pygame.quit()

```

Common Controller Axis Mappings

Controller mappings are drawn from the underlying SDL library which pygame uses and they differ between pygame 1 and pygame 2. Below are a couple of mappings for two popular game pads.

Pygame 2

Axis and hat mappings are listed from -1 to +1.

X-Box 360 Controller (name: "Xbox 360 Controller")

In pygame 2 the X360 controller mapping has 6 Axes, 11 buttons and 1 hat.

- **Left Stick:**

```

Left -> Right   - Axis 0
Up    -> Down   - Axis 1

```

- **Right Stick:**

```

Left -> Right   - Axis 3
Up    -> Down   - Axis 4

```

- **Left Trigger:**

```

Out -> In       - Axis 2

```

- **Right Trigger:**

```

Out -> In       - Axis 5

```

- **Buttons:**

A Button	- Button 0
B Button	- Button 1
X Button	- Button 2
Y Button	- Button 3
Left Bumper	- Button 4
Right Bumper	- Button 5
Back Button	- Button 6
Start Button	- Button 7
L. Stick In	- Button 8
R. Stick In	- Button 9
Guide Button	- Button 10

- **Hat/D-pad:**

Down -> Up	- Y Axis
Left -> Right	- X Axis

Playstation 4 Controller (name: "PS4 Controller")

In pygame 2 the PS4 controller mapping has 6 Axes and 16 buttons.

- **Left Stick:**

Left -> Right	- Axis 0
Up -> Down	- Axis 1

- **Right Stick:**

Left -> Right	- Axis 2
Up -> Down	- Axis 3

- **Left Trigger:**

Out -> In	- Axis 4
-----------	----------

- **Right Trigger:**

Out -> In	- Axis 5
-----------	----------

- **Buttons:**

Cross Button	- Button 0
Circle Button	- Button 1
Square Button	- Button 2
Triangle Button	- Button 3
Share Button	- Button 4
PS Button	- Button 5
Options Button	- Button 6
L. Stick In	- Button 7
R. Stick In	- Button 8
Left Bumper	- Button 9
Right Bumper	- Button 10
D-pad Up	- Button 11
D-pad Down	- Button 12
D-pad Left	- Button 13
D-pad Right	- Button 14
Touch Pad Click	- Button 15

Pygame 1

Axis and hat mappings are listed from -1 to +1.

X-Box 360 Controller (name: "Controller (XBOX 360 For Windows)")

In pygame 1 the X360 controller mapping has 5 Axes, 10 buttons and 1 hat.

- **Left Stick:**

```
Left -> Right   - Axis 0
Up   -> Down    - Axis 1
```

- **Right Stick:**

```
Left -> Right   - Axis 4
Up   -> Down    - Axis 3
```

- **Left Trigger & Right Trigger:**

```
RT -> LT       - Axis 2
```

- **Buttons:**

```
A Button      - Button 0
B Button      - Button 1
X Button      - Button 2
Y Button      - Button 3
Left Bumper   - Button 4
Right Bumper  - Button 5
Back Button   - Button 6
Start Button  - Button 7
L. Stick In   - Button 8
R. Stick In   - Button 9
```

- **Hat/D-pad:**

```
Down -> Up      - Y Axis
Left  -> Right   - X Axis
```

Playstation 4 Controller (name: "Wireless Controller")

In pygame 1 the PS4 controller mapping has 6 Axes and 14 buttons and 1 hat.

- **Left Stick:**

```
Left -> Right   - Axis 0
Up   -> Down    - Axis 1
```

- **Right Stick:**

```
Left -> Right   - Axis 2
Up   -> Down    - Axis 3
```

- **Left Trigger:**

```
Out -> In       - Axis 5
```

- **Right Trigger:**

```
Out -> In       - Axis 4
```

- **Buttons:**

```
Cross Button   - Button 0
Circle Button  - Button 1
Square Button  - Button 2
Triangle Button - Button 3
Left Bumper    - Button 4
Right Bumper   - Button 5
L. Trigger(Full) - Button 6
R. Trigger(Full) - Button 7
Share Button    - Button 8
Options Button  - Button 9
L. Stick In    - Button 10
R. Stick In    - Button 11
PS Button      - Button 12
Touch Pad Click - Button 13
```

- **Hat/D-pad:**

```
Down -> Up      - Y Axis
Left -> Right   - X Axis
```

pygame.key

pygame module to work with the keyboard

This module contains functions for dealing with the keyboard.

The `pygame.event` queue gets `pygame.KEYDOWN` and `pygame.KEYUP` events when the keyboard buttons are pressed and released. Both events have `key` and `mod` attributes.

- `key`: an integer ID representing every key on the keyboard
- `mod`: a bitmask of all the modifier keys that were in a pressed state when the event occurred

The `pygame.KEYDOWN` event has the additional attributes `unicode` and `scancode`.

- `unicode`: a single character string that is the fully translated character entered, this takes into account the shift and composition keys
- `scancode`: the platform-specific key code, which could be different from keyboard to keyboard, but is useful for key selection of weird keys like the multimedia keys

New in version 2.0.0: *New in version 2.0.0:* The `pygame.TEXTINPUT` event is preferred to the `unicode` attribute of `pygame.KEYDOWN`. The attribute `text` contains the input.

The following is a list of all the constants (from `pygame.locals`) used to represent keyboard keys.

Portability note: The integers for key constants differ between pygame 1 and 2. Always use key constants (`K_a`) rather than integers directly (97) so that your key handling code works well on both pygame 1 and pygame 2.

pygame Constant	ASCII	Description

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 control
K_LCTRL		left control
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

The keyboard also has a list of modifier states (from `pygame.locals`) that can be assembled by bitwise-ORing them together.

pygame Constant	Description

KMOD_NONE	no modifier keys pressed
KMOD_LSHIFT	left shift
KMOD_RSHIFT	right shift
KMOD_SHIFT	left shift or right shift or both
KMOD_LCTRL	left control
KMOD_RCTRL	right control
KMOD_CTRL	left control or right control or both
KMOD_LALT	left alt

KMOD_RALT	right alt
KMOD_ALT	left alt or right alt or both
KMOD_LMETA	left meta
KMOD_RMETA	right meta
KMOD_META	left meta or right meta or both
KMOD_CAPS	caps lock
KMOD_NUM	num lock
KMOD_MODE	AltGr

The modifier information is contained in the `mod` attribute of the `pygame.KEYDOWN` and `pygame.KEYUP` events. The `mod` attribute is a bitmask of all the modifier keys that were in a pressed state when the event occurred. The modifier information can be decoded using a bitwise AND (except for `KMOD_NONE`, which should be compared using equals `==`). For example:

```
for event in pygame.event.get():
    if event.type == pygame.KEYDOWN or event.type == pygame.KEYUP:
        if event.mod == pygame.KMOD_NONE:
            print('No modifier keys were in a pressed state when this '
                  'event occurred.')
        else:
            if event.mod & pygame.KMOD_LSHIFT:
                print('Left shift was in a pressed state when this event '
                      'occurred.')
            if event.mod & pygame.KMOD_RSHIFT:
                print('Right shift was in a pressed state when this event '
                      'occurred.')
            if event.mod & pygame.KMOD_SHIFT:
                print('Left shift or right shift or both were in a '
                      'pressed state when this event occurred.')
```

`pygame.key.get_focused()`

true if the display is receiving keyboard input from the system

`get_focused()` -> bool

Returns `True` when the display window has keyboard focus from the system. If the display needs to ensure it does not lose keyboard focus, it can use `pygame.event.set_grab()` to grab all input.

`pygame.key.get_pressed()`

get the state of all keyboard buttons

`get_pressed()` -> bools

Returns a sequence of boolean values representing the state of every key on the keyboard. Use the key constant values to index the array. A `True` value means the that button is pressed.

Note

Getting the list of pushed buttons with this function is not the proper way to handle text entry from the user. There is no way to know the order of keys pressed, and rapidly pushed keys can be completely unnoticed between two calls to `pygame.key.get_pressed()`. There is also no way to translate these pushed keys into a fully translated character value. See the `pygame.KEYDOWN` events on the `pygame.event` queue for this functionality.

`pygame.key.get_mods()`

determine which modifier keys are being held

`get_mods()` -> int

Returns a single integer representing a bitmask of all the modifier keys being held. Using bitwise operators you can test if specific modifier keys are pressed.

`pygame.key.set_mods()`

temporarily set which modifier keys are pressed

set_mods(int) -> None

Create a bitmask of the modifier key constants you want to impose on your program.

pygame.key.set_repeat ()

control how held keys are repeated

set_repeat() -> None

set_repeat(delay) -> None

set_repeat(delay, interval) -> None

When the keyboard repeat is enabled, keys that are held down will generate multiple `pygame.KEYDOWN` events. The `delay` parameter is the number of milliseconds before the first repeated `pygame.KEYDOWN` event will be sent. After that, another `pygame.KEYDOWN` event will be sent every `interval` milliseconds. If a `delay` value is provided and an `interval` value is not provided or is 0, then the `interval` will be set to the same value as `delay`.

To disable key repeat call this function with no arguments or with `delay` set to 0.

When pygame is initialized the key repeat is disabled.

Raises: **ValueError** -- if `delay` or `interval` is < 0

Changed in version 2.0.0: *Changed in version 2.0.0:* A `ValueError` is now raised (instead of a `pygame.error`) if `delay` or `interval` is < 0.

pygame.key.get_repeat ()

see how held keys are repeated

get_repeat() -> (delay, interval)

Get the `delay` and `interval` keyboard repeat values. Refer to `pygame.key.set_repeat()` for a description of these values.

New in version 1.8: *New in version 1.8.*

pygame.key.name ()

get the name of a key identifier

name(key) -> string

Get the descriptive name of the button from a keyboard button id constant.

pygame.key.key_code ()

get the key identifier from a key name

key_code(name=string) -> int

Get the key identifier code from the descriptive name of the key. This returns an integer matching one of the `K_*` keycodes. For example:

```
>>> pygame.key.key_code("return") == pygame.K_RETURN
True
>>> pygame.key.key_code("0") == pygame.K_0
True
>>> pygame.key.key_code("space") == pygame.K_SPACE
True
```

Raises:

- **ValueError** -- if the key name is not known.

- **NotImplementedError** -- if used with SDL 1.

New in version 2.0.0: *New in version 2.0.0.*

pygame.key.start_text_input ()

start handling Unicode text input events

start_text_input() -> None

Start receiving `pygame.TEXTEDITING` and `pygame.TEXTINPUT` events.

A `pygame.TEXTEDITING` event is received when an IME composition is started or changed. It contains the composition `text`, `length`, and editing `start` position within the composition (attributes `text`, `length`, and `start`, respectively). When the composition is committed (or non-IME input is received), a `pygame.TEXTINPUT` event is generated.

Text input events handling is on by default.

New in version 2.0.0: *New in version 2.0.0.*

`pygame.key.stop_text_input()`

stop handling Unicode text input events

`stop_text_input()` -> None

Stop receiving `pygame.TEXTEDITING` and `pygame.TEXTINPUT` events.

Text input events handling is on by default

New in version 2.0.0: *New in version 2.0.0.*

`pygame.key.set_text_input_rect()`

controls the position of the candidate list

`set_text_input_rect(Rect)` -> None

This sets the rectangle used for typing with an IME. It controls where the candidate list will open, if supported.

New in version 2.0.0: *New in version 2.0.0.*

pygame.locals

pygame constants

This module contains various constants used by pygame. Its contents are automatically placed in the pygame module namespace. However, an application can use `pygame.locals` to include only the pygame constants with a `from pygame.locals import *`.

Detailed descriptions of the various constants can be found throughout the pygame documentation. Here are the locations of some of them.

- The `pygame.display` module contains flags like `HWSURFACE` used by `pygame.display.set_mode()`.
- The `pygame.event` module contains the various event types.
- The `pygame.key` module lists the keyboard constants and modifiers (`K_*` and `MOD_*`) relating to the `key` and `mod` attributes of the `KEYDOWN` and `KEYUP` events.
- The `pygame.time` module defines `TIMER_RESOLUTION`.

pygame.mask

pygame module for image masks.

Useful for fast pixel perfect collision detection. A mask uses 1 bit per-pixel to store which parts collide.

New in version 1.8: *New in version 1.8.*

`pygame.mask.from_surface()`

Creates a Mask from the given surface

`from_surface(Surface)` -> Mask

`from_surface(Surface, threshold=127)` -> Mask

Creates a **Mask** object from the given surface by setting all the opaque pixels and not setting the transparent pixels.

If the surface uses a color-key, then it is used to decide which bits in the resulting mask are set. All the pixels that are **not** equal to the color-key are **set** and the pixels equal to the color-key are not set.

If a color-key is not used, then the alpha value of each pixel is used to decide which bits in the resulting mask are set. All the pixels that have an alpha value **greater than** the `threshold` parameter are **set** and the pixels with an alpha value less than or equal to the `threshold` are not set.

Parameters:

- **surface** (*Surface*) -- the surface to create the mask from
- **threshold** (*int*) -- (optional) the alpha threshold (default is 127) to compare with each surface pixel's alpha value, if the `surface` is color-keyed this parameter is ignored

Returns: a newly created **Mask** object from the given surface

Return type: *Mask*

Note

This function is used to create the masks for `pygame.sprite.collide_mask()`.

`pygame.mask.from_threshold()`

Creates a mask by thresholding Surfaces

`from_threshold(Surface, color) -> Mask`

`from_threshold(Surface, color, threshold=(0, 0, 0, 255), othersurface=None, palette_colors=1) -> Mask`

This is a more featureful method of getting a **Mask** from a surface.

If the optional `othersurface` is not used, all the pixels **within** the `threshold` of the `color` parameter are **set** in the resulting mask.

If the optional `othersurface` is used, every pixel in the first surface that is **within** the `threshold` of the corresponding pixel in `othersurface` is **set** in the resulting mask.

Parameters:

- **surface** (*Surface*) -- the surface to create the mask from
- **color** (*Color* or *int* or *tuple(int, int, int, [int])* or *list[int, int, int, [int]]*) -- color used to check if the surface's pixels are within the given `threshold` range, this parameter is ignored if the optional `othersurface` parameter is supplied
- **threshold** (*Color* or *int* or *tuple(int, int, int, [int])* or *list[int, int, int, [int]]*) -- (optional) the threshold range used to check the difference between two colors (default is `(0, 0, 0, 255)`)
- **othersurface** (*Surface*) -- (optional) used to check whether the pixels of the first surface are within the given `threshold` range of the pixels from this surface (default is `None`)
- **palette_colors** (*int*) -- (optional) indicates whether to use the palette colors or not, a nonzero value causes the palette colors to be used and a 0 causes them not to be used (default is 1)

Returns: a newly created **Mask** object from the given surface

Return type: **Mask**

`pygame.mask.Mask`

pygame object for representing 2D bitmasks

`Mask(size=(width, height)) -> Mask`

`Mask(size=(width, height), fill=False) -> Mask`

A **Mask** object is used to represent a 2D bitmask. Each bit in the mask represents a pixel. 1 is used to indicate a set bit and 0 is used to indicate an unset bit. Set bits in a mask can be used to detect collisions with other masks and their set bits.

A filled mask has all of its bits set to 1, conversely an unfilled/cleared/empty mask has all of its bits set to 0. Masks can be created unfilled (default) or filled by using the `fill` parameter. Masks can also be cleared or filled using the `pygame.mask.Mask.clear()` and `pygame.mask.Mask.fill()` methods respectively.

A mask's coordinates start in the top left corner at `(0, 0)` just like `pygame.Surface`. Individual bits can be accessed using the `pygame.mask.Mask.get_at()` and `pygame.mask.Mask.set_at()` methods.

The methods `overlap()`, `overlap_area()`, `overlap_mask()`, `draw()`, `erase()`, and `convolve()` use an offset parameter to indicate the offset of another mask's top left corner from the calling mask's top left corner. The calling mask's top left corner is considered to be the origin `(0, 0)`. Offsets are a tuple or list of 2 integer values `(x_offset, y_offset)`. Positive and negative offset values are supported.

```

        0 to x (x_offset)
        :
0 ..... +---:-----+
to      |   :       |
y ..... +---:-----+
(y_offset) |   | othermask |
          |   +---:-----+
          | calling_mask |
          +---:-----+
```

Parameters:

- **size** (*tuple(int, int) or list[int, int]*) -- the dimensions of the mask (width and height)
- **fill** (*bool*) -- (optional) create an unfilled mask (default: `False`) or filled mask (`True`)

Returns: a newly created `Mask` object

Return type: `Mask`

Changed in version 2.0.0: *Changed in version 2.0.0:* Shallow copy support added. The `Mask` class supports the special method `__copy__()` and shallow copying via `copy.copy(mask)`.

Changed in version 2.0.0: *Changed in version 2.0.0:* Subclassing support added. The `Mask` class can be used as a base class.

Changed in version 1.9.5: *Changed in version 1.9.5:* Added support for keyword arguments.

Changed in version 1.9.5: *Changed in version 1.9.5:* Added the optional keyword parameter `fill`.

Changed in version 1.9.5: *Changed in version 1.9.5:* Added support for masks with a width and/or a height of 0.

copy()

Returns a new copy of the mask

`copy()` -> `Mask`

Returns: a new copy of this mask, the new mask will have the same width, height, and set/unset bits as the original

Return type: `Mask`

Note

If a mask subclass needs to copy any instance specific attributes then it should override the `__copy__()` method. The overridden `__copy__()` method needs to call `super().__copy__()` and then copy the required data as in the following example code.

```
class SubMask(pygame.mask.Mask):
    def __copy__(self):
        new_mask = super().__copy__()
        # Do any SubMask attribute copying here.
        return new_mask
```

New in version 2.0.0: *New in version 2.0.0.*

get_size()

Returns the size of the mask

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

Returns: the size of the mask, (width, height)

Return type: `tuple(int, int)`

get_rect()

Returns a `Rect` based on the size of the mask

`get_rect(**kwargs)` -> `Rect`

Returns a new `pygame.Rect()` object based on the size of this mask. The rect's default position will be (0, 0) and its default width and height will be the same as this mask's. The rect's attributes can be altered via `pygame.Rect()` attribute keyword arguments/values passed into this method. As an example, `a_mask.get_rect(center=(10, 5))` would create a `pygame.Rect()` based on the mask's size centered at the given position.

Parameters: **kwargs** (*dict*) -- `pygame.Rect()` attribute keyword arguments/values that will be applied to the rect

Returns: a new `pygame.Rect()` object based on the size of this mask with any `pygame.Rect()` attribute keyword arguments/values applied to it

Return type: `Rect`

New in version 2.0.0: *New in version 2.0.0.*

get_at ()

Gets the bit at the given position

get_at((x, y)) -> int

Parameters: **pos** (*tuple(int, int)* or *list[int, int]*) -- the position of the bit to get

Returns: 1 if the bit is set, 0 if the bit is not set

Return type: int

Raises: **IndexError** -- if the position is outside of the mask's bounds

set_at ()

Sets the bit at the given position

set_at((x, y)) -> None

set_at((x, y), value=1) -> None

Parameters:

- **pos** (*tuple(int, int)* or *list[int, int]*) -- the position of the bit to set

- **value** (*int*) -- any nonzero int will set the bit to 1, 0 will set the bit to 0 (default is 1)

Returns: None

Return type: NoneType

Raises: **IndexError** -- if the position is outside of the mask's bounds

overlap ()

Returns the point of intersection

overlap(othermask, offset) -> (x, y)

overlap(othermask, offset) -> None

Returns the first point of intersection encountered between this mask and othermask. A point of intersection is 2 overlapping set bits.

The current algorithm searches the overlapping area in `sizeof(unsigned long int) * CHAR_BIT` bit wide column blocks (the value of `sizeof(unsigned long int) * CHAR_BIT` is platform dependent, for clarity it will be referred to as `W`). Starting at the top left corner it checks bits 0 to `W - 1` of the first row ((0, 0) to (`W - 1`, 0)) then continues to the next row ((0, 1) to (`W - 1`, 1)). Once this entire column block is checked, it continues to the next one (`W` to `2 * W - 1`). This is repeated until it finds a point of intersection or the entire overlapping area is checked.

Parameters:

- **othermask** (*Mask*) -- the other mask to overlap with this mask

- **offset** (*tuple(int, int)* or *list[int, int]*) -- the offset of othermask from this mask, for more details refer to the Mask offset notes

Returns: point of intersection or None if no intersection

Return type: tuple(int, int) or NoneType

overlap_area ()

Returns the number of overlapping set bits

overlap_area(othermask, offset) -> numbits

Returns the number of overlapping set bits between between this mask and othermask.

This can be useful for collision detection. An approximate collision normal can be found by calculating the gradient of the overlapping area through the finite difference.

```
dx = mask.overlap_area(othermask, (x + 1, y)) - mask.overlap_area(othermask, (x - 1, y))
dy = mask.overlap_area(othermask, (x, y + 1)) - mask.overlap_area(othermask, (x, y - 1))
```

Parameters:

- **othermask** (*Mask*) -- the other mask to overlap with this mask

- **offset** (*tuple(int, int)* or *list[int, int]*) -- the offset of othermask from this mask, for more details refer to the Mask offset notes

Returns: the number of overlapping set bits

Return type: int

overlap_mask ()

Returns a mask of the overlapping set bits

overlap_mask(othermask, offset) -> Mask

Returns a **Mask**, the same size as this mask, containing the overlapping set bits between this mask and othermask.

Parameters:

- **othermask** (*Mask*) -- the other mask to overlap with this mask
- **offset** (*tuple(int, int) or list[int, int]*) -- the offset of othermask from this mask, for more details refer to the Mask offset notes

Returns: a newly created **Mask** with the overlapping bits set

Return type: *Mask*

fill ()

Sets all bits to 1

fill() -> None

Sets all bits in the mask to 1.

Returns: None

Return type: *NoneType*

clear ()

Sets all bits to 0

clear() -> None

Sets all bits in the mask to 0.

Returns: None

Return type: *NoneType*

invert ()

Flips all the bits

invert() -> None

Flips all of the bits in the mask. All the set bits are cleared to 0 and all the unset bits are set to 1.

Returns: None

Return type: *NoneType*

scale ()

Resizes a mask

scale((width, height)) -> Mask

Creates a new **Mask** of the requested size with its bits scaled from this mask.

Parameters: **size** (*tuple(int, int) or list[int, int]*) -- the width and height (size) of the mask to create

Returns: a new **Mask** object with its bits scaled from this mask

Return type: *Mask*

Raises: **ValueError** -- if width < 0 or height < 0

draw ()

Draws a mask onto another

draw(othermask, offset) -> None

Performs a bitwise OR, drawing othermask onto this mask.

Parameters:

- **othermask** (*Mask*) -- the mask to draw onto this mask
- **offset** (*tuple(int, int) or list[int, int]*) -- the offset of othermask from this mask, for more details refer to the Mask offset notes

Returns: None

Return type: `NoneType`

erase ()

Erases a mask from another

`erase(othermask, offset) -> None`

Erases (clears) all bits set in `othermask` from this mask.

Parameters:

- **othermask** (*Mask*) -- the mask to erase from this mask
- **offset** (*tuple(int, int) or list[int, int]*) -- the offset of `othermask` from this mask, for more details refer to the Mask offset notes

Returns: `None`

Return type: `NoneType`

count ()

Returns the number of set bits

`count() -> bits`

Returns: the number of set bits in the mask

Return type: `int`

centroid ()

Returns the centroid of the set bits

`centroid() -> (x, y)`

Finds the centroid (the center mass of the set bits) for this mask.

Returns: a coordinate tuple indicating the centroid of the mask, it will return `(0, 0)` if the mask has no bits set

Return type: `tuple(int, int)`

angle ()

Returns the orientation of the set bits

`angle() -> theta`

Finds the approximate orientation (from -90 to 90 degrees) of the set bits in the mask. This works best if performed on a mask with only one connected component.

Returns: the orientation of the set bits in the mask, it will return `0.0` if the mask has no bits set

Return type: `float`

Note

See `connected_component()` for details on how a connected component is calculated.

outline ()

Returns a list of points outlining an object

`outline() -> [(x, y), ...]`

`outline(every=1) -> [(x, y), ...]`

Returns a list of points of the outline of the first connected component encountered in the mask. To find a connected component, the mask is searched per row (left to right) starting in the top left corner.

The `every` optional parameter skips set bits in the outline. For example, setting it to 10 would return a list of every 10th set bit in the outline.

Parameters: **every** (*int*) -- (optional) indicates the number of bits to skip over in the outline (default is 1)

Returns: a list of points outlining the first connected component encountered, an empty list is returned if the mask has no bits set

Return type: `list[tuple(int, int)]`

Note

See `connected_component()` for details on how a connected component is calculated.

convolve()

Returns the convolution of this mask with another mask

`convolve(othermask) -> Mask`

`convolve(othermask, outputmask=None, offset=(0, 0)) -> Mask`

Convolve this mask with the given `othermask`.

Parameters:

- **othermask** (*Mask*) -- mask to convolve this mask with
- **outputmask** (*Mask or NoneType*) -- (optional) mask for output (default is `None`)
- **offset** (*tuple(int, int) or list[int, int]*) -- the offset of `othermask` from this mask, (default is `(0, 0)`)

Returns: a **Mask** with the `(i - offset[0], j - offset[1])` bit set, if shifting `othermask` (such that its bottom right corner is at `(i, j)`) causes it to overlap with this mask. If an `outputmask` is specified, the output is drawn onto it and it is returned. Otherwise a mask of size `(MAX(0, width + othermask's width - 1), MAX(0, height + othermask's height - 1))` is created and returned.

Return type: *Mask*

connected_component()

Returns a mask containing a connected component

`connected_component() -> Mask`

`connected_component((x, y)) -> Mask`

A connected component is a group (1 or more) of connected set bits (orthogonally and diagonally). The SAUF algorithm, which checks 8 point connectivity, is used to find a connected component in the mask.

By default this method will return a **Mask** containing the largest connected component in the mask. Optionally, a bit coordinate can be specified and the connected component containing it will be returned. If the bit at the given location is not set, the returned **Mask** will be empty (no bits set).

Parameters: **pos** (*tuple(int, int) or list[int, int]*) -- (optional) selects the connected component that contains the bit at this position

Returns: a **Mask** object (same size as this mask) with the largest connected component from this mask, if this mask has no bits set then an empty mask will be returned. If the `pos` parameter is provided then the mask returned will have the connected component that contains this position. An empty mask will be returned if the `pos` parameter selects an unset bit.

Return type: *Mask*

Raises: **IndexError** -- if the optional `pos` parameter is outside of the mask's bounds

connected_components()

Returns a list of masks of connected components

`connected_components() -> [Mask, ...]`

`connected_components(min=0) -> [Mask, ...]`

Provides a list containing a **Mask** object for each connected component.

Parameters: **min** (*int*) -- (optional) indicates the minimum number of bits (to filter out noise) per connected component (default is 0, which equates to no minimum and is equivalent to setting it to 1, as a connected component must have at least 1 bit set)

Returns: a list containing a **Mask** object for each connected component, an empty list is returned if the mask has no bits set

Return type: *list[Mask]*

Note

See `connected_component()` for details on how a connected component is calculated.

get_bounding_rects()

Returns a list of bounding rects of connected components

`get_bounding_rects()` -> [Rect, ...]

Provides a list containing a bounding rect for each connected component.

Returns: a list containing a bounding rect for each connected component, an empty list is returned if the mask has no bits set

Return type: list[Rect]

Note

See `connected_component()` for details on how a connected component is calculated.

to_surface()

Returns a surface with the mask drawn on it

`to_surface()` -> Surface

`to_surface(surface=None, setsurface=None, unsetsurface=None, setcolor=(255, 255, 255, 255), unsetcolor=(0, 0, 0, 255), dest=(0, 0))` -> Surface

Draws this mask on the given surface. Set bits (bits set to 1) and unset bits (bits set to 0) can be drawn onto a surface.

Parameters:

- **surface** (*Surface* or *None*) -- (optional) Surface to draw mask onto, if no surface is provided one will be created (default is *None*, which will cause a surface with the parameters `Surface(size=mask.get_size(), flags=SRCALPHA, depth=32)` to be created, drawn on, and returned)
- **setsurface** (*Surface* or *None*) -- (optional) use this surface's color values to draw set bits (default is *None*), if this surface is smaller than the mask any bits outside its bounds will use the `setcolor` value
- **unsetsurface** (*Surface* or *None*) -- (optional) use this surface's color values to draw unset bits (default is *None*), if this surface is smaller than the mask any bits outside its bounds will use the `unsetcolor` value
- **setcolor** (*Color* or *str* or *int* or *tuple(int, int, int, [int])* or *list(int, int, int, [int])* or *None*) -- (optional) color to draw set bits (default is (255, 255, 255, 255), white), use *None* to skip drawing the set bits, the `setsurface` parameter (if set) will takes precedence over this parameter
- **unsetcolor** (*Color* or *str* or *int* or *tuple(int, int, int, [int])* or *list(int, int, int, [int])* or *None*) -- (optional) color to draw unset bits (default is (0, 0, 0, 255), black), use *None* to skip drawing the unset bits, the `unsetsurface` parameter (if set) will takes precedence over this parameter
- **dest** (*Rect* or *tuple(int, int)* or *list(int, int)* or *Vector2(int, int)*) -- (optional) surface destination of where to position the topleft corner of the mask being drawn (default is (0, 0)), if a *Rect* is used as the `dest` parameter, its `x` and `y` attributes will be used as the destination, **NOTE1:** rects with a negative width or height value will not be normalized before using their `x` and `y` values, **NOTE2:** this destination value is only used to position the mask on the surface, it does not offset the `setsurface` and `unsetsurface` from the mask, they are always aligned with the mask (i.e. position (0, 0) on the mask always corresponds to position (0, 0) on the `setsurface` and `unsetsurface`)

- Returns:** the `surface` parameter (or a newly created surface if no `surface` parameter was provided) with this mask drawn on it
- Return type:** `Surface`
- Raises:** **ValueError** -- if the `setsurface` parameter or `unsetsurface` parameter does not have the same format (bytesize/bitsize/alpha) as the `surface` parameter

Note

To skip drawing the set bits, both `setsurface` and `setcolor` must be `None`. The `setsurface` parameter defaults to `None`, but `setcolor` defaults to a color value and therefore must be set to `None`.

Note

To skip drawing the unset bits, both `unsetsurface` and `unsetcolor` must be `None`. The `unsetsurface` parameter defaults to `None`, but `unsetcolor` defaults to a color value and therefore must be set to `None`.

New in version 2.0.0: *New in version 2.0.0.*

pygame.math

pygame module for vector classes

The pygame math module currently provides Vector classes in two and three dimensions, `Vector2` and `Vector3` respectively.

They support the following numerical operations: `vec+vec`, `vec-vec`, `vec*number`, `number*vec`, `vec/number`, `vec//number`, `vec+=vec`, `vec-=vec`, `vec*=number`, `vec/=number`, `vec//=number`.

All these operations will be performed elementwise. In addition `vec*vec` will perform a scalar-product (a.k.a. dot-product). If you want to multiply every element from vector `v` with every element from vector `w` you can use the elementwise method: `v.elementwise() * w`

The coordinates of a vector can be retrieved or set using attributes or subscripts

```
v = pygame.Vector3()

v.x = 5
v[1] = 2 * v.x
print(v[1]) # 10

v.x == v[0]
v.y == v[1]
v.z == v[2]
```

Multiple coordinates can be set using slices or swizzling

```
v = pygame.Vector2()
v.xy = 1, 2
v[:] = 1, 2
```

New in version 1.9.2pre: *New in version 1.9.2pre.*

Changed in version 1.9.4: *Changed in version 1.9.4:* Removed experimental notice.

Changed in version 1.9.4: *Changed in version 1.9.4:* Allow scalar construction like GLSL `Vector2(2) == Vector2(2.0, 2.0)`

Changed in version 1.9.4: *Changed in version 1.9.4:* `pygame.math` required import. More convenient `pygame.Vector2` and `pygame.Vector3`.

`pygame.math.Vector2`

a 2-Dimensional Vector

`Vector2()` -> `Vector2`

`Vector2(int)` -> `Vector2`

`Vector2(float)` -> `Vector2`

`Vector2(Vector2)` -> `Vector2`

`Vector2(x, y)` -> `Vector2`

`Vector2((x, y))` -> `Vector2`

Some general information about the `Vector2` class.

dot ()

calculates the dot- or scalar-product with the other vector

`dot(Vector2)` -> float

cross ()

calculates the cross- or vector-product

`cross(Vector2)` -> `Vector2`

calculates the third component of the cross-product.

magnitude ()

returns the Euclidean magnitude of the vector.

`magnitude()` -> float

calculates the magnitude of the vector which follows from the theorem:

`vec.magnitude()` == `math.sqrt(vec.x**2 + vec.y**2)`

magnitude_squared ()

returns the squared magnitude of the vector.

`magnitude_squared()` -> float

calculates the magnitude of the vector which follows from the theorem:

`vec.magnitude_squared()` == `vec.x**2 + vec.y**2`. This is faster than `vec.magnitude()` because it avoids the square root.

length ()

returns the Euclidean length of the vector.

`length()` -> float

calculates the Euclidean length of the vector which follows from the Pythagorean theorem:

`vec.length()` == `math.sqrt(vec.x**2 + vec.y**2)`

length_squared ()

returns the squared Euclidean length of the vector.

`length_squared()` -> float

calculates the Euclidean length of the vector which follows from the Pythagorean theorem:

`vec.length_squared()` == `vec.x**2 + vec.y**2`. This is faster than `vec.length()` because it avoids the square root.

normalize ()

returns a vector with the same direction but length 1.

`normalize()` -> `Vector2`

Returns a new vector that has `length` equal to 1 and the same direction as self.

normalize_ip ()

normalizes the vector in place so that its length is 1.

`normalize_ip()` -> None

Normalizes the vector so that it has `length` equal to 1. The direction of the vector is not changed.

is_normalized ()

tests if the vector is normalized i.e. has `length` == 1.

`is_normalized()` -> Bool

Returns True if the vector has `length` equal to 1. Otherwise it returns False.

scale_to_length()

scales the vector to a given length.

`scale_to_length(float)` -> None

Scales the vector so that it has the given length. The direction of the vector is not changed. You can also scale to length 0. If the vector is the zero vector (i.e. has length 0 thus no direction) a `ValueError` is raised.

reflect()

returns a vector reflected of a given normal.

`reflect(Vector2)` -> Vector2

Returns a new vector that points in the direction as if self would bounce of a surface characterized by the given surface normal. The length of the new vector is the same as self's.

reflect_ip()

reflect the vector of a given normal in place.

`reflect_ip(Vector2)` -> None

Changes the direction of self as if it would have been reflected of a surface with the given surface normal.

distance_to()

calculates the Euclidean distance to a given vector.

`distance_to(Vector2)` -> float

distance_squared_to()

calculates the squared Euclidean distance to a given vector.

`distance_squared_to(Vector2)` -> float

lerp()

returns a linear interpolation to the given vector.

`lerp(Vector2, float)` -> Vector2

Returns a Vector which is a linear interpolation between self and the given Vector. The second parameter determines how far between self and other the result is going to be. It must be a value between 0 and 1 where 0 means self and 1 means other will be returned.

slerp()

returns a spherical interpolation to the given vector.

`slerp(Vector2, float)` -> Vector2

Calculates the spherical interpolation from self to the given Vector. The second argument - often called t - must be in the range `[-1, 1]`. It parametrizes where - in between the two vectors - the result should be. If a negative value is given the interpolation will not take the complement of the shortest path.

elementwise()

The next operation will be performed elementwise.

`elementwise()` -> VectorElementwiseProxy

Applies the following operation to each element of the vector.

rotate()

rotates a vector by a given angle in degrees.

`rotate(angle)` -> Vector2

Returns a vector which has the same length as self but is rotated counterclockwise by the given angle in degrees.

rotate_rad()

rotates a vector by a given angle in radians.

`rotate_rad(angle)` -> Vector2

Returns a vector which has the same length as self but is rotated counterclockwise by the given angle in radians.

New in version 2.0.0: *New in version 2.0.0.*

rotate_ip()

rotates the vector by a given angle in degrees in place.

rotate_ip(angle) -> None

Rotates the vector counterclockwise by the given angle in degrees. The length of the vector is not changed.

rotate_ip_rad()

rotates the vector by a given angle in radians in place.

rotate_ip_rad(angle) -> None

Rotates the vector counterclockwise by the given angle in radians. The length of the vector is not changed.

New in version 2.0.0: *New in version 2.0.0.*

angle_to()

calculates the angle to a given vector in degrees.

angle_to(Vector2) -> float

Returns the angle between self and the given vector.

as_polar()

returns a tuple with radial distance and azimuthal angle.

as_polar() -> (r, phi)

Returns a tuple (r, phi) where r is the radial distance, and phi is the azimuthal angle.

from_polar()

Sets x and y from a polar coordinates tuple.

from_polar((r, phi)) -> None

Sets x and y from a tuple (r, phi) where r is the radial distance, and phi is the azimuthal angle.

update()

Sets the coordinates of the vector.

update() -> None

update(int) -> None

update(float) -> None

update(Vector2) -> None

update(x, y) -> None

update((x, y)) -> None

Sets coordinates x and y in place.

New in version 1.9.5: *New in version 1.9.5.*

pygame.math.Vector3

a 3-Dimensional Vector

Vector3() -> Vector3

Vector3(int) -> Vector3

Vector3(float) -> Vector3

Vector3(Vector3) -> Vector3

Vector3(x, y, z) -> Vector3

Vector3((x, y, z)) -> Vector3

Some general information about the Vector3 class.

dot()

calculates the dot- or scalar-product with the other vector

dot(Vector3) -> float

cross()

calculates the cross- or vector-product
 cross(Vector3) -> Vector3
 calculates the cross-product.

magnitude ()

returns the Euclidean magnitude of the vector.
 magnitude() -> float
 calculates the magnitude of the vector which follows from the theorem:
 $\text{vec.magnitude}() == \text{math.sqrt}(\text{vec.x}^2 + \text{vec.y}^2 + \text{vec.z}^2)$

magnitude_squared ()

returns the squared Euclidean magnitude of the vector.
 magnitude_squared() -> float
 calculates the magnitude of the vector which follows from the theorem:
 $\text{vec.magnitude_squared}() == \text{vec.x}^2 + \text{vec.y}^2 + \text{vec.z}^2$. This is faster than
 $\text{vec.magnitude}()$ because it avoids the square root.

length ()

returns the Euclidean length of the vector.
 length() -> float
 calculates the Euclidean length of the vector which follows from the Pythagorean theorem:
 $\text{vec.length}() == \text{math.sqrt}(\text{vec.x}^2 + \text{vec.y}^2 + \text{vec.z}^2)$

length_squared ()

returns the squared Euclidean length of the vector.
 length_squared() -> float
 calculates the Euclidean length of the vector which follows from the Pythagorean theorem:
 $\text{vec.length_squared}() == \text{vec.x}^2 + \text{vec.y}^2 + \text{vec.z}^2$. This is faster than $\text{vec.length}()$
 because it avoids the square root.

normalize ()

returns a vector with the same direction but length 1.
 normalize() -> Vector3
 Returns a new vector that has length equal to 1 and the same direction as self.

normalize_ip ()

normalizes the vector in place so that its length is 1.
 normalize_ip() -> None
 Normalizes the vector so that it has length equal to 1. The direction of the vector is not changed.

is_normalized ()

tests if the vector is normalized i.e. has length == 1.
 is_normalized() -> Bool
 Returns True if the vector has length equal to 1. Otherwise it returns False.

scale_to_length ()

scales the vector to a given length.
 scale_to_length(float) -> None
 Scales the vector so that it has the given length. The direction of the vector is not changed. You can also scale to length 0. If the vector is the zero vector (i.e. has length 0 thus no direction) a ValueError is raised.

reflect ()

returns a vector reflected of a given normal.
 reflect(Vector3) -> Vector3
 Returns a new vector that points in the direction as if self would bounce of a surface characterized by the given surface normal. The length of the new vector is the same as self's.

reflect_ip()

reflect the vector of a given normal in place.

reflect_ip(Vector3) -> None

Changes the direction of self as if it would have been reflected of a surface with the given surface normal.

distance_to()

calculates the Euclidean distance to a given vector.

distance_to(Vector3) -> float

distance_squared_to()

calculates the squared Euclidean distance to a given vector.

distance_squared_to(Vector3) -> float

lerp()

returns a linear interpolation to the given vector.

lerp(Vector3, float) -> Vector3

Returns a Vector which is a linear interpolation between self and the given Vector. The second parameter determines how far between self and other the result is going to be. It must be a value between 0 and 1, where 0 means self and 1 means other will be returned.

slerp()

returns a spherical interpolation to the given vector.

slerp(Vector3, float) -> Vector3

Calculates the spherical interpolation from self to the given Vector. The second argument - often called t - must be in the range $[-1, 1]$. It parametrizes where - in between the two vectors - the result should be. If a negative value is given the interpolation will not take the complement of the shortest path.

elementwise()

The next operation will be performed elementwise.

elementwise() -> VectorElementwiseProxy

Applies the following operation to each element of the vector.

rotate()

rotates a vector by a given angle in degrees.

rotate(angle, Vector3) -> Vector3

Returns a vector which has the same length as self but is rotated counterclockwise by the given angle in degrees around the given axis.

rotate_rad()

rotates a vector by a given angle in radians.

rotate_rad(angle, Vector3) -> Vector3

Returns a vector which has the same length as self but is rotated counterclockwise by the given angle in radians around the given axis.

New in version 2.0.0: *New in version 2.0.0.*

rotate_ip()

rotates the vector by a given angle in degrees in place.

rotate_ip(angle, Vector3) -> None

Rotates the vector counterclockwise around the given axis by the given angle in degrees. The length of the vector is not changed.

rotate_ip_rad()

rotates the vector by a given angle in radians in place.

rotate_ip_rad(angle, Vector3) -> None

Rotates the vector counterclockwise around the given axis by the given angle in radians. The length of the vector is not changed.

New in version 2.0.0: *New in version 2.0.0.*

rotate_x()

rotates a vector around the x-axis by the angle in degrees.

rotate_x(angle) -> Vector3

Returns a vector which has the same length as self but is rotated counterclockwise around the x-axis by the given angle in degrees.

rotate_x_rad()

rotates a vector around the x-axis by the angle in radians.

rotate_x_rad(angle) -> Vector3

Returns a vector which has the same length as self but is rotated counterclockwise around the x-axis by the given angle in radians.

New in version 2.0.0: *New in version 2.0.0.*

rotate_x_ip()

rotates the vector around the x-axis by the angle in degrees in place.

rotate_x_ip(angle) -> None

Rotates the vector counterclockwise around the x-axis by the given angle in degrees. The length of the vector is not changed.

rotate_x_ip_rad()

rotates the vector around the x-axis by the angle in radians in place.

rotate_x_ip_rad(angle) -> None

Rotates the vector counterclockwise around the x-axis by the given angle in radians. The length of the vector is not changed.

New in version 2.0.0: *New in version 2.0.0.*

rotate_y()

rotates a vector around the y-axis by the angle in degrees.

rotate_y(angle) -> Vector3

Returns a vector which has the same length as self but is rotated counterclockwise around the y-axis by the given angle in degrees.

rotate_y_rad()

rotates a vector around the y-axis by the angle in radians.

rotate_y_rad(angle) -> Vector3

Returns a vector which has the same length as self but is rotated counterclockwise around the y-axis by the given angle in radians.

New in version 2.0.0: *New in version 2.0.0.*

rotate_y_ip()

rotates the vector around the y-axis by the angle in degrees in place.

rotate_y_ip(angle) -> None

Rotates the vector counterclockwise around the y-axis by the given angle in degrees. The length of the vector is not changed.

rotate_y_ip_rad()

rotates the vector around the y-axis by the angle in radians in place.

rotate_y_ip_rad(angle) -> None

Rotates the vector counterclockwise around the y-axis by the given angle in radians. The length of the vector is not changed.

New in version 2.0.0: *New in version 2.0.0.*

rotate_z()

rotates a vector around the z-axis by the angle in degrees.

`rotate_z(angle) -> Vector3`

Returns a vector which has the same length as self but is rotated counterclockwise around the z-axis by the given angle in degrees.

`rotate_z_rad()`

rotates a vector around the z-axis by the angle in radians.

`rotate_z_rad(angle) -> Vector3`

Returns a vector which has the same length as self but is rotated counterclockwise around the z-axis by the given angle in radians.

New in version 2.0.0: *New in version 2.0.0.*

`rotate_z_ip()`

rotates the vector around the z-axis by the angle in degrees in place.

`rotate_z_ip(angle) -> None`

Rotates the vector counterclockwise around the z-axis by the given angle in degrees. The length of the vector is not changed.

`rotate_z_ip_rad()`

rotates the vector around the z-axis by the angle in radians in place.

`rotate_z_ip_rad(angle) -> None`

Rotates the vector counterclockwise around the z-axis by the given angle in radians. The length of the vector is not changed.

`angle_to()`

calculates the angle to a given vector in degrees.

`angle_to(Vector3) -> float`

Returns the angle between self and the given vector.

`as_spherical()`

returns a tuple with radial distance, inclination and azimuthal angle.

`as_spherical() -> (r, theta, phi)`

Returns a tuple `(r, theta, phi)` where `r` is the radial distance, `theta` is the inclination angle and `phi` is the azimuthal angle.

`from_spherical()`

Sets `x`, `y` and `z` from a spherical coordinates 3-tuple.

`from_spherical((r, theta, phi)) -> None`

Sets `x`, `y` and `z` from a tuple `(r, theta, phi)` where `r` is the radial distance, `theta` is the inclination angle and `phi` is the azimuthal angle.

`update()`

Sets the coordinates of the vector.

`update() -> None`

`update(int) -> None`

`update(float) -> None`

`update(Vector3) -> None`

`update(x, y, z) -> None`

`update((x, y, z)) -> None`

Sets coordinates `x`, `y`, and `z` in place.

New in version 1.9.5: *New in version 1.9.5.*

`pygame.math.enable_swizzling()`

globally enables swizzling for vectors.

`enable_swizzling() -> None`

DEPRECATED: Not needed anymore. Will be removed in a later version.

Enables swizzling for all vectors until `disable_swizzling()` is called. By default swizzling is disabled.

Lets you get or set multiple coordinates as one attribute, eg `vec.xyz = 1, 2, 3`.

`pygame.math.disable_swizzling()`

globally disables swizzling for vectors.

`disable_swizzling()` -> None

DEPRECATED: Not needed anymore. Will be removed in a later version.

Disables swizzling for all vectors until `enable_swizzling()` is called. By default swizzling is disabled.

pygame.midi

pygame module for interacting with midi input and output.

New in version 1.9.0: *New in version 1.9.0.*

The midi module can send output to midi devices and get input from midi devices. It can also list midi devices on the system.

The midi module supports real and virtual midi devices.

It uses the portmidi library. Is portable to which ever platforms portmidi supports (currently Windows, Mac OS X, and Linux).

This uses pyportmidi for now, but may use its own bindings at some point in the future. The pyportmidi bindings are included with pygame.

New in version 2.0.0: *New in version 2.0.0.*

These are pygame events (`pygame.event`) reserved for midi use. The `MIDIIN` event is used by `pygame.midi.midis2events()` when converting midi events to pygame events.

`MIDIIN`
`MIDIOUT`

`pygame.midi.init()`

initialize the midi module

`init()` -> None

Initializes the `pygame.midi` module. Must be called before using the `pygame.midi` module.

It is safe to call this more than once.

`pygame.midi.quit()`

uninitialize the midi module

`quit()` -> None

Uninitializes the `pygame.midi` module. If `pygame.midi.init()` was called to initialize the `pygame.midi` module, then this function will be called automatically when your program exits.

It is safe to call this function more than once.

`pygame.midi.get_init()`

returns True if the midi module is currently initialized

`get_init()` -> bool

Gets the initialization state of the `pygame.midi` module.

Returns: True if the `pygame.midi` module is currently initialized.

Return type: bool

New in version 1.9.5: *New in version 1.9.5.*

`pygame.midi.Input`

Input is used to get midi input from midi devices.

`Input(device_id)` -> None

`Input(device_id, buffer_size)` -> None

Parameters:

- **device_id** (*int*) -- midi device id
- **buffer_size** (*int*) -- (optional) the number of input events to be buffered

close ()

closes a midi stream, flushing any pending buffers.

close() -> None

PortMidi attempts to close open streams when the application exits.

Note

This is particularly difficult under Windows.

poll ()

returns True if there's data, or False if not.

poll() -> bool

Used to indicate if any data exists.

Returns: True if there is data, False otherwise

Return type: bool

Raises: **MidiException** -- on error

read ()

reads num_events midi events from the buffer.

read(num_events) -> midi_event_list

Reads from the input buffer and gives back midi events.

Parameters: **num_events** (*int*) -- number of input events to read

Returns: the format for midi_event_list is
[[[status, data1, data2, data3], timestamp], ...]

Return type: list

pygame.midi.Output

Output is used to send midi to an output device

Output(device_id) -> None

Output(device_id, latency=0) -> None

Output(device_id, buffer_size=256) -> None

Output(device_id, latency, buffer_size) -> None

The **buffer_size** specifies the number of output events to be buffered waiting for output. In some cases (see below) PortMidi does not buffer output at all and merely passes data to a lower-level API, in which case buffersize is ignored.

latency is the delay in milliseconds applied to timestamps to determine when the output should actually occur. If **latency** is <0, 0 is assumed.

If **latency** is zero, timestamps are ignored and all output is delivered immediately. If **latency** is greater than zero, output is delayed until the message timestamp plus the **latency**. In some cases, PortMidi can obtain better timing than your application by passing timestamps along to the device driver or hardware. Latency may also help you to synchronize midi data to audio data by matching midi latency to the audio buffer latency.

Note

Time is measured relative to the time source indicated by **time_proc**. Timestamps are absolute, not relative delays or offsets.

abort ()

terminates outgoing messages immediately

`abort()` -> None

The caller should immediately close the output port; this call may result in transmission of a partial midi message. There is no abort for Midi input because the user can simply ignore messages in the buffer and close an input device at any time.

`close ()`

closes a midi stream, flushing any pending buffers.

`close()` -> None

PortMidi attempts to close open streams when the application exits.

Note

This is particularly difficult under Windows.

`note_off ()`

turns a midi note off (note must be on)

`note_off(note, velocity=None, channel=0)` -> None

Turn a note off in the output stream. The note must already be on for this to work correctly.

`note_on ()`

turns a midi note on (note must be off)

`note_on(note, velocity=None, channel=0)` -> None

Turn a note on in the output stream. The note must already be off for this to work correctly.

`set_instrument ()`

select an instrument, with a value between 0 and 127

`set_instrument(instrument_id, channel=0)` -> None

Select an instrument.

`pitch_bend ()`

modify the pitch of a channel.

`set_instrument(value=0, channel=0)` -> None

Adjust the pitch of a channel. The value is a signed integer from -8192 to +8191. For example, 0 means "no change", +4096 is typically a semitone higher, and -8192 is 1 whole tone lower (though the musical range corresponding to the pitch bend range can also be changed in some synthesizers).

If no value is given, the pitch bend is returned to "no change".

New in version 1.9.4: *New in version 1.9.4.*

`write ()`

writes a list of midi data to the Output

`write(data)` -> None

Writes series of MIDI information in the form of a list.

Parameters: **data** (*list*) -- data to write, the expected format is `[[[status, data1=0, data2=0, ...], timestamp], ...]` with the data# fields being optional

Raises: **IndexError** -- if more than 1024 elements in the data list

Example:

```
# Program change at time 20000 and 500ms later send note 65 with
# velocity 100.
write([[[0xc0, 0, 0], 20000], [[0x90, 60, 100], 20500]])
```

Note

- Timestamps will be ignored if latency = 0
- To get a note to play immediately, send MIDI info with timestamp read from function Time
- Optional data fields: `write([[[0xc0, 0, 0], 20000]])` is equivalent to `write([[[0xc0], 20000]])`

write_short ()

writes up to 3 bytes of midi data to the Output

`write_short(status)` -> None

`write_short(status, data1=0, data2=0)` -> None

Output MIDI information of 3 bytes or less. The data fields are optional and assumed to be 0 if omitted.

Examples of status byte values:

```
0xc0 # program change
0x90 # note on
# etc.
```

Example:

```
# note 65 on with velocity 100
write_short(0x90, 65, 100)
```

write_sys_ex ()

writes a timestamped system-exclusive midi message.

`write_sys_ex(when, msg)` -> None

Writes a timestamped system-exclusive midi message.

Parameters:

- **msg** (*list[int]* or *str*) -- midi message
- **when** -- timestamp in milliseconds

Example:

```
midi_output.write_sys_ex(0, '\xF0\x7D\x10\x11\x12\x13\xF7')

# is equivalent to

midi_output.write_sys_ex(pygame.midi.time(),
                        [0xF0, 0x7D, 0x10, 0x11, 0x12, 0x13, 0xF7])
```

pygame.midi.get_count ()

gets the number of devices.

`get_count()` -> num_devices

Device ids range from 0 to `get_count() - 1`

pygame.midi.get_default_input_id ()

gets default input device number

`get_default_input_id()` -> default_id

The following describes the usage details for this function and the `get_default_output_id()` function.

Return the default device ID or -1 if there are no devices. The result can be passed to the `Input/Output` class.

On a PC the user can specify a default device by setting an environment variable. To use device #1, for example:

```
set PM_RECOMMENDED_INPUT_DEVICE=1
or
set PM_RECOMMENDED_OUTPUT_DEVICE=1
```

The user should first determine the available device ID by using the supplied application "testin" or "testout".

In general, the registry is a better place for this kind of info. With USB devices that can come and go, using integers is not very reliable for device identification. Under Windows, if `PM_RECOMMENDED_INPUT_DEVICE` (or `PM_RECOMMENDED_OUTPUT_DEVICE`) is NOT found in the environment, then the default device is obtained by looking for a string in the registry under:

```
HKEY_LOCAL_MACHINE/SOFTWARE/PortMidi/Recommended_Input_Device
or
HKEY_LOCAL_MACHINE/SOFTWARE/PortMidi/Recommended_Output_Device
```

The number of the first device with a substring that matches the string exactly is returned. For example, if the string in the registry is "USB" and device 1 is named "In USB MidiSport 1x1", then that will be the default input because it contains the string "USB".

In addition to the name, `get_device_info()` returns "interf", which is the interface name. The "interface" is the underlying software system or API used by PortMidi to access devices. Supported interfaces:

```
MMSystem    # the only Win32 interface currently supported
ALSA        # the only Linux interface currently supported
CoreMIDI    # the only Mac OS X interface currently supported
# DirectX   - not implemented
# OSS       - not implemented
```

To specify both the interface and the device name in the registry, separate the two with a comma and a space. The string before the comma must be a substring of the "interf" string and the string after the space must be a substring of the "name" name string in order to match the device. e.g.:

```
MMSystem, In USB MidiSport 1x1
```

Note

In the current release, the default is simply the first device (the input or output device with the lowest `PmDeviceID`).

`pygame.midi.get_default_output_id()`

gets default output device number

`get_default_output_id()` -> `default_id`

See `get_default_input_id()` for usage details.

`pygame.midi.get_device_info()`

returns information about a midi device

`get_device_info(an_id)` -> (interf, name, input, output, opened)

`get_device_info(an_id)` -> `None`

Gets the device info for a given id.

Parameters: `an_id (int)` -- id of the midi device being queried

Returns: if the id is out of range `None` is returned, otherwise a tuple of (interf, name, input, output, opened) is returned. interf: string describing the device interface (e.g. 'ALSA') name: string name of the device (e.g. 'Midi Through Port-0') input: 1 if the device is an input device, otherwise 0 output: 1 if the device is an output device, otherwise 0 opened: 1 if the device is opened, otherwise 0

Return type: tuple or `None`

`pygame.midi.midis2events()`

converts midi events to pygame events

`midis2events(midi_events, device_id)` -> [Event, ...]

Takes a sequence of midi events and returns list of pygame events.

The `midi_events` data is expected to be a sequence of ((status, data1, data2, data3), timestamp) midi events (all values required).

Returns: a list of pygame events of event type `MIDIIN`

Return type: list

`pygame.midi.time()`

returns the current time in ms of the PortMidi timer

`time()` -> time

The time is reset to 0 when the `pygame.midi` module is initialized.

`pygame.midi.frequency_to_midi()`

Converts a frequency into a MIDI note. Rounds to the closest midi note.

`frequency_to_midi(midi_note)` -> `frequency`

example:

```
frequency_to_midi(27.5) == 21
```

New in version 1.9.5: *New in version 1.9.5.*

`pygame.midi.midi_to_frequency()`

Converts a midi note to a frequency.

`midi_to_frequency(midi_note)` -> `frequency`

example:

```
midi_to_frequency(21) == 27.5
```

New in version 1.9.5: *New in version 1.9.5.*

`pygame.midi.midi_to_ansi_note()`

Returns the Ansi Note name for a midi number.

`midi_to_ansi_note(midi_note)` -> `ansi_note`

example:

```
midi_to_ansi_note(21) == 'A0'
```

New in version 1.9.5: *New in version 1.9.5.*

exception `pygame.midi.MidiException`

exception that pygame.midi functions and classes can raise

`MidiException(errno)` -> None

pygame.mixer

pygame module for loading and playing sounds

This module contains classes for loading Sound objects and controlling playback. The mixer module is optional and depends on `SDL_mixer`. Your program should test that `pygame.mixer` is available and initialized before using it.

The mixer module has a limited number of channels for playback of sounds. Usually programs tell pygame to start playing audio and it selects an available channel automatically. The default is 8 simultaneous channels, but complex programs can get more precise control over the number of channels and their use.

All sound playback is mixed in background threads. When you begin to play a Sound object, it will return immediately while the sound continues to play. A single Sound object can also be actively played back multiple times.

The mixer also has a special streaming channel. This is for music playback and is accessed through the `pygame.mixer.music` module.

The mixer module must be initialized like other pygame modules, but it has some extra conditions. The `pygame.mixer.init()` function takes several optional arguments to control the playback rate and sample size. Pygame will default to reasonable values, but pygame cannot perform Sound resampling, so the mixer should be initialized to match the values of your audio resources.

NOTE: For less laggy sound use a smaller buffer size. The default is set to reduce the chance of scratchy sounds on some computers. You can change the default buffer by calling `pygame.mixer.pre_init()` before `pygame.mixer.init()` or `pygame.init()` is called. For example:

```
pygame.mixer.pre_init(44100, -16, 2, 1024)
```

```
pygame.mixer.init()
```

initialize the mixer module

`init(frequency=44100, size=-16, channels=2, buffer=512, devicename=None, allowedchanges=AUDIO_ALLOW_FREQUENCY_CHANGE | AUDIO_ALLOW_CHANNELS_CHANGE) -> None`
 Initialize the mixer module for Sound loading and playback. The default arguments can be overridden to provide specific audio mixing. Keyword arguments are accepted. For backward compatibility where an argument is set zero the default value is used (possible changed by a `pre_init` call).

The `size` argument represents how many bits are used for each audio sample. If the value is negative then signed sample values will be used. Positive values mean unsigned audio samples will be used. An invalid value raises an exception.

The `channels` argument is used to specify whether to use mono or stereo. 1 for mono and 2 for stereo.

The `buffer` argument controls the number of internal samples used in the sound mixer. The default value should work for most cases. It can be lowered to reduce latency, but sound dropout may occur. It can be raised to larger values to ensure playback never skips, but it will impose latency on sound playback. The buffer size must be a power of two (if not it is rounded up to the next nearest power of 2).

Some platforms require the `pygame.mixer` module to be initialized after the display modules have initialized. The top level `pygame.init()` takes care of this automatically, but cannot pass any arguments to the mixer `init`. To solve this, mixer has a function `pygame.mixer.pre_init()` to set the proper defaults before the toplevel `init` is used.

When using `allowedchanges=0` it will convert the samples at runtime to match what the hardware supports. For example a sound card may not support 16bit sound samples, so instead it will use 8bit samples internally. If `AUDIO_ALLOW_FORMAT_CHANGE` is supplied, then the requested format will change to the closest that SDL2 supports.

Apart from 0, `allowedchanges` accepts the following constants ORed together:

- `AUDIO_ALLOW_FREQUENCY_CHANGE`
- `AUDIO_ALLOW_FORMAT_CHANGE`
- `AUDIO_ALLOW_CHANNELS_CHANGE`
- `AUDIO_ALLOW_ANY_CHANGE`

It is safe to call this more than once, but after the mixer is initialized you cannot change the playback arguments without first calling `pygame.mixer.quit()`.

Changed in version 1.8: *Changed in version 1.8:* The default `buffer` size was changed from 1024 to 3072.

Changed in version 1.9.1: *Changed in version 1.9.1:* The default `buffer` size was changed from 3072 to 4096.

Changed in version 2.0.0: *Changed in version 2.0.0:* The default `buffer` size was changed from 4096 to 512. The default frequency changed to 44100 from 22050.

Changed in version 2.0.0: *Changed in version 2.0.0:* `size` can be 32 (32bit floats).

Changed in version 2.0.0: *Changed in version 2.0.0:* `channels` can also be 4 or 6.

New in version 2.0.0: *New in version 2.0.0:* `allowedchanges` argument added

`pygame.mixer.pre_init()`

preset the mixer init arguments

`pre_init(frequency=44100, size=-16, channels=2, buffer=512, devicename=None) -> None`

Call `pre_init` to change the defaults used when the real `pygame.mixer.init()` is called. Keyword arguments are accepted. The best way to set custom mixer playback values is to call `pygame.mixer.pre_init()` before calling the top level `pygame.init()`. For backward compatibility argument values of zero are replaced with the startup defaults.

Changed in version 1.8: *Changed in version 1.8:* The default `buffer` size was changed from 1024 to 3072.

Changed in version 1.9.1: *Changed in version 1.9.1:* The default `buffer` size was changed from 3072 to 4096.

Changed in version 2.0.0: *Changed in version 2.0.0:* The default `buffer` size was changed from 4096 to 512. The default frequency changed to 44100 from 22050.

`pygame.mixer.quit()`

uninitialize the mixer

`quit() -> None`

This will uninitialize `pygame.mixer`. All playback will stop and any loaded Sound objects may not be compatible with the mixer if it is reinitialized later.

`pygame.mixer.get_init()`

test if the mixer is initialized

`get_init() -> (frequency, format, channels)`

If the mixer is initialized, this returns the playback arguments it is using. If the mixer has not been initialized this returns `None`.

`pygame.mixer.stop()`

stop playback of all sound channels

`stop()` -> `None`

This will stop all playback of all active mixer channels.

`pygame.mixer.pause()`

temporarily stop playback of all sound channels

`pause()` -> `None`

This will temporarily stop all playback on the active mixer channels. The playback can later be resumed with

`pygame.mixer.unpause()`

`pygame.mixer.unpause()`

resume paused playback of sound channels

`unpause()` -> `None`

This will resume all active sound channels after they have been paused.

`pygame.mixer.fadeout()`

fade out the volume on all sounds before stopping

`fadeout(time)` -> `None`

This will fade out the volume on all active channels over the time argument in milliseconds. After the sound is muted the playback will stop.

`pygame.mixer.set_num_channels()`

set the total number of playback channels

`set_num_channels(count)` -> `None`

Sets the number of available channels for the mixer. The default value is 8. The value can be increased or decreased. If the value is decreased, sounds playing on the truncated channels are stopped.

`pygame.mixer.get_num_channels()`

get the total number of playback channels

`get_num_channels()` -> `count`

Returns the number of currently active playback channels.

`pygame.mixer.set_reserved()`

reserve channels from being automatically used

`set_reserved(count)` -> `None`

The mixer can reserve any number of channels that will not be automatically selected for playback by Sounds. If sounds are currently playing on the reserved channels they will not be stopped.

This allows the application to reserve a specific number of channels for important sounds that must not be dropped or have a guaranteed channel to play on.

`pygame.mixer.find_channel()`

find an unused channel

`find_channel(force=False)` -> `Channel`

This will find and return an inactive Channel object. If there are no inactive Channels this function will return `None`. If there are no inactive channels and the force argument is `True`, this will find the Channel with the longest running Sound and return it.

If the mixer has reserved channels from `pygame.mixer.set_reserved()` then those channels will not be returned here.

`pygame.mixer.get_busy()`

test if any sound is being mixed

`get_busy()` -> `bool`

Returns `True` if the mixer is busy mixing any channels. If the mixer is idle then this return `False`.

`pygame.mixer.get_sdl_mixer_version()`

get the mixer's SDL version

`get_sdl_mixer_version()` -> (major, minor, patch)

`get_sdl_mixer_version(linked=True) -> (major, minor, patch)`

Parameters: **linked** (*bool*) -- if `True` (default) the linked version number is returned, otherwise the compiled version number is returned

Returns: the mixer's SDL library version number (linked or compiled depending on the `linked` parameter) as a tuple of 3 integers (`major`, `minor`, `patch`)

Return type: tuple

Note

The linked and compile version numbers should be the same.

New in version 2.0.0: *New in version 2.0.0.*

pygame.mixer.Sound

Create a new Sound object from a file or buffer object

`Sound(filename) -> Sound`

`Sound(file=filename) -> Sound`

`Sound(buffer) -> Sound`

`Sound(buffer=buffer) -> Sound`

`Sound(object) -> Sound`

`Sound(file=object) -> Sound`

`Sound(array=object) -> Sound`

Load a new sound buffer from a filename, a python file object or a readable buffer object. Limited resampling will be performed to help the sample match the initialize arguments for the mixer. A Unicode string can only be a file pathname. A Python 2.x string or a Python 3.x bytes object can be either a pathname or a buffer object. Use the 'file' or 'buffer' keywords to avoid ambiguity; otherwise Sound may guess wrong. If the array keyword is used, the object is expected to export a version 3, C level array interface or, for Python 2.6 or later, a new buffer interface (The object is checked for a buffer interface first.)

The Sound object represents actual sound sample data. Methods that change the state of the Sound object will the all instances of the Sound playback. A Sound object also exports an array interface, and, for Python 2.6 or later, a new buffer interface.

The Sound can be loaded from an OGG audio file or from an uncompressed WAV.

Note: The buffer will be copied internally, no data will be shared between it and the Sound object.

For now buffer and array support is consistent with `sndarray.make_sound` for Numeric arrays, in that sample sign and byte order are ignored. This will change, either by correctly handling sign and byte order, or by raising an exception when different. Also, source samples are truncated to fit the audio sample size. This will not change.

New in version 1.8: *New in version 1.8:* `pygame.mixer.Sound(buffer)`

New in version 1.9.2: *New in version 1.9.2:* `pygame.mixer.Sound` keyword arguments and array interface support

play()

begin sound playback

`play(loops=0, maxtime=0, fade_ms=0) -> Channel`

Begin playback of the Sound (i.e., on the computer's speakers) on an available Channel. This will forcibly select a Channel, so playback may cut off a currently playing sound if necessary.

The loops argument controls how many times the sample will be repeated after being played the first time. A value of 5 means that the sound will be played once, then repeated five times, and so is played a total of six times. The default value (zero) means the Sound is not repeated, and so is only played once. If loops is set to -1 the Sound will loop indefinitely (though you can still call `stop()` to stop it).

The maxtime argument can be used to stop playback after a given number of milliseconds.

The fade_ms argument will make the sound start playing at 0 volume and fade up to full volume over the time given. The sample may end before the fade-in is complete.

This returns the Channel object for the channel that was selected.

stop()

stop sound playback

`stop() -> None`

This will stop the playback of this Sound on any active Channels.

fadeout ()

stop sound playback after fading out

fadeout(time) -> None

This will stop playback of the sound after fading it out over the time argument in milliseconds. The Sound will fade and stop on all actively playing channels.

set_volume ()

set the playback volume for this Sound

set_volume(value) -> None

This will set the playback volume (loudness) for this Sound. This will immediately affect the Sound if it is playing. It will also affect any future playback of this Sound.

Parameters: **value** (*float*) -- volume in the range of 0.0 to 1.0 (inclusive) If value < 0.0, the volume will not be changed If value > 1.0, the volume will be set to 1.0

get_volume ()

get the playback volume

get_volume() -> value

Return a value from 0.0 to 1.0 representing the volume for this Sound.

get_num_channels ()

count how many times this Sound is playing

get_num_channels() -> count

Return the number of active channels this sound is playing on.

get_length ()

get the length of the Sound

get_length() -> seconds

Return the length of this Sound in seconds.

get_raw ()

return a bytestring copy of the Sound samples.

get_raw() -> bytes

Return a copy of the Sound object buffer as a bytes (for Python 3.x) or str (for Python 2.x) object.

New in version 1.9.2: *New in version 1.9.2.*

pygame.mixer.Channel

Create a Channel object for controlling playback

Channel(id) -> Channel

Return a Channel object for one of the current channels. The id must be a value from 0 to the value of `pygame.mixer.get_num_channels()`.

The Channel object can be used to get fine control over the playback of Sounds. A channel can only playback a single Sound at time. Using channels is entirely optional since pygame can manage them by default.

play ()

play a Sound on a specific Channel

play(Sound, loops=0, maxtime=0, fade_ms=0) -> None

This will begin playback of a Sound on a specific Channel. If the Channel is currently playing any other Sound it will be stopped.

The loops argument has the same meaning as in `Sound.play()`: it is the number of times to repeat the sound after the first time. If it is 3, the sound will be played 4 times (the first time, then three more). If loops is -1 then the playback will repeat indefinitely.

As in `Sound.play()`, the maxtime argument can be used to stop playback of the Sound after a given number of milliseconds.

As in `Sound.play()`, the fade_ms argument can be used fade in the sound.

stop ()

stop playback on a Channel

stop() -> None

Stop sound playback on a channel. After playback is stopped the channel becomes available for new Sounds to play on it.

pause ()

temporarily stop playback of a channel

pause() -> None

Temporarily stop the playback of sound on a channel. It can be resumed at a later time with `Channel.unpause()`

unpause ()

resume pause playback of a channel

unpause() -> None

Resume the playback on a paused channel.

fadeout ()

stop playback after fading channel out

fadeout(time) -> None

Stop playback of a channel after fading out the sound over the given time argument in milliseconds.

set_volume ()

set the volume of a playing channel

set_volume(value) -> None

set_volume(left, right) -> None

Set the volume (loudness) of a playing sound. When a channel starts to play its volume value is reset. This only affects the current sound. The value argument is between 0.0 and 1.0.

If one argument is passed, it will be the volume of both speakers. If two arguments are passed and the mixer is in stereo mode, the first argument will be the volume of the left speaker and the second will be the volume of the right speaker. (If the second argument is `None`, the first argument will be the volume of both speakers.)

If the channel is playing a Sound on which `set_volume()` has also been called, both calls are taken into account. For example:

```
sound = pygame.mixer.Sound("s.wav")
channel = s.play()           # Sound plays at full volume by default
sound.set_volume(0.9)       # Now plays at 90% of full volume.
sound.set_volume(0.6)       # Now plays at 60% (previous value replaced).
channel.set_volume(0.5)     # Now plays at 30% (0.6 * 0.5).
```

get_volume ()

get the volume of the playing channel

get_volume() -> value

Return the volume of the channel for the current playing sound. This does not take into account stereo separation used by `Channel.set_volume()`. The Sound object also has its own volume which is mixed with the channel.

get_busy ()

check if the channel is active

get_busy() -> bool

Returns `True` if the channel is actively mixing sound. If the channel is idle this returns `False`.

get_sound ()

get the currently playing Sound

get_sound() -> Sound

Return the actual Sound object currently playing on this channel. If the channel is idle `None` is returned.

queue ()

queue a Sound object to follow the current

queue(Sound) -> None

When a Sound is queued on a Channel, it will begin playing immediately after the current Sound is finished. Each channel can only have a single Sound queued at a time. The queued Sound will only play if the current playback finished automatically. It is cleared on any other call to `Channel.stop()` or `Channel.play()`.

If there is no sound actively playing on the Channel then the Sound will begin playing immediately.

get_queue ()

return any Sound that is queued

get_queue() -> Sound

If a Sound is already queued on this channel it will be returned. Once the queued sound begins playback it will no longer be on the queue.

set_endevent ()

have the channel send an event when playback stops

set_endevent() -> None

set_endevent(type) -> None

When an endevent is set for a channel, it will send an event to the pygame queue every time a sound finishes playing on that channel (not just the first time). Use `pygame.event.get()` to retrieve the endevent once it's sent.

Note that if you called `Sound.play(n)` or `Channel.play(sound,n)`, the end event is sent only once: after the sound has been played "n+1" times (see the documentation of `Sound.play()`).

If `Channel.stop()` or `Channel.play()` is called while the sound was still playing, the event will be posted immediately.

The type argument will be the event id sent to the queue. This can be any valid event type, but a good choice would be a value between `pygame.locals.USEREVENT` and `pygame.locals.NUMEVENTS`. If no type argument is given then the Channel will stop sending endevents.

get_endevent ()

get the event a channel sends when playback stops

get_endevent() -> type

Returns the event type to be sent every time the Channel finishes playback of a Sound. If there is no endevent the function returns `pygame.NOEVENT`.

pygame.mouse

pygame module to work with the mouse

The mouse functions can be used to get the current state of the mouse device. These functions can also alter the system cursor for the mouse.

When the display mode is set, the event queue will start receiving mouse events. The mouse buttons generate `pygame.MOUSEBUTTONDOWN` and `pygame.MOUSEBUTTONUP` events when they are pressed and released. These events contain a button attribute representing which button was pressed. The mouse wheel will generate `pygame.MOUSEBUTTONDOWN` and `pygame.MOUSEBUTTONUP` events when rolled. The button will be set to 4 when the wheel is rolled up, and to button 5 when the wheel is rolled down. Whenever the mouse is moved it generates a `pygame.MOUSEMOTION` event. The mouse movement is broken into small and accurate motion events. As the mouse is moving many motion events will be placed on the queue. Mouse motion events that are not properly cleaned from the event queue are the primary reason the event queue fills up.

If the mouse cursor is hidden, and input is grabbed to the current display the mouse will enter a virtual input mode, where the relative movements of the mouse will never be stopped by the borders of the screen. See the functions `pygame.mouse.set_visible()` and `pygame.event.set_grab()` to get this configured.

Mouse Wheel Behavior in pygame 2

There is proper functionality for mouse wheel behaviour with pygame 2 supporting `pygame.MOUSEWHEEL` events. The new events support horizontal and vertical scroll movements, with signed integer values representing the

amount scrolled (x and y), as well as flipped direction (the set positive and negative values for each axis is flipped). Read more about SDL2 input-related changes here <https://wiki.libsdl.org/MigrationGuide#Input>

In pygame 2, the mouse wheel functionality can be used by listening for the `pygame.MOUSEWHEEL` type of an event. When this event is triggered, a developer can access the appropriate Event object with `pygame.event.get()`. The object can be used to access data about the mouse scroll, such as `which` (it will tell you what exact mouse device trigger the event).

Code example of mouse scroll (tested on 2.0.0.dev7)

```
# Taken from husano896's PR thread (slightly modified)
import pygame
from pygame.locals import *
pygame.init()
screen = pygame.display.set_mode((640, 480))
clock = pygame.time.Clock()

def main():
    while True:
        for event in pygame.event.get():
            if event.type == QUIT:
                pygame.quit()
                return
            elif event.type == MOUSEWHEEL:
                print(event)
                print(event.x, event.y)
                print(event.flipped)
                print(event.which)
                # can access properties with
                # proper notation(ex: event.y)
        clock.tick(60)

# Execute game:
main()
```

`pygame.mouse.get_pressed()`

get the state of the mouse buttons

`get_pressed(num_buttons=3)` -> (button1, button2, button3)

`get_pressed(num_buttons=5)` -> (button1, button2, button3, button4, button5)

Returns a sequence of booleans representing the state of all the mouse buttons. A true value means the mouse is currently being pressed at the time of the call.

Note, to get all of the mouse events it is better to use either `pygame.event.wait()` or `pygame.event.get()` and check all of those events to see if they are `MOUSEBUTTONDOWN`, `MOUSEBUTTONUP`, or `MOUSEMOTION`.

Note, that on x11 some X servers use middle button emulation. When you click both buttons 1 and 3 at the same time a 2 button event can be emitted.

Note, remember to call `pygame.event.get()` before this function. Otherwise it will not work as expected.

To support five button mice, an optional parameter `num_buttons` has been added in pygame 2. When this is set to 5, `button4` and `button5` are added to the returned tuple. Only 3 and 5 are valid values for this parameter.

Changed in version 2.0.0: *Changed in version 2.0.0:* `num_buttons` argument added

`pygame.mouse.get_pos()`

get the mouse cursor position

`get_pos()` -> (x, y)

Returns the x and y position of the mouse cursor. The position is relative to the top-left corner of the display. The cursor position can be located outside of the display window, but is always constrained to the screen.

`pygame.mouse.get_rel()`

get the amount of mouse movement

`get_rel()` -> (x, y)

Returns the amount of movement in *x* and *y* since the previous call to this function. The relative movement of the mouse cursor is constrained to the edges of the screen, but see the virtual input mouse mode for a way around this. Virtual input mode is described at the top of the page.

`pygame.mouse.set_pos()`

set the mouse cursor position

`set_pos([x, y])` -> None

Set the current mouse position to arguments given. If the mouse cursor is visible it will jump to the new coordinates. Moving the mouse will generate a new `pygame.MOUSEMOTION` event.

`pygame.mouse.set_visible()`

hide or show the mouse cursor

`set_visible(bool)` -> bool

If the bool argument is true, the mouse cursor will be visible. This will return the previous visible state of the cursor.

`pygame.mouse.get_visible()`

get the current visibility state of the mouse cursor

`get_visible()` -> bool

Get the current visibility state of the mouse cursor. True if the mouse is visible, False otherwise.

New in version 2.0.0: *New in version 2.0.0.*

`pygame.mouse.get_focused()`

check if the display is receiving mouse input

`get_focused()` -> bool

Returns true when pygame is receiving mouse input events (or, in windowing terminology, is "active" or has the "focus").

This method is most useful when working in a window. By contrast, in full-screen mode, this method always returns true.

Note: under MS Windows, the window that has the mouse focus also has the keyboard focus. But under X-Windows, one window can receive mouse events and another receive keyboard events.

`pygame.mouse.get_focused()` indicates whether the pygame window receives mouse events.

`pygame.mouse.set_cursor()`

set the mouse cursor to a new cursor

`set_cursor(pygame.cursors.Cursor)` -> None

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

`set_cursor(hotspot, surface)` -> None

`set_cursor(constant)` -> None

Set the mouse cursor to something new. This function accepts either an explicit `Cursor` object or arguments to create a `Cursor` object.

See `pygame.cursors.Cursor` for help creating cursors and for examples.

Changed in version 2.0.1: *Changed in version 2.0.1.*

`pygame.mouse.get_cursor()`

get the current mouse cursor

`get_cursor()` -> `pygame.cursors.Cursor`

Get the information about the mouse system cursor. The return value contains the same data as the arguments passed into `pygame.mouse.set_cursor()`.

Note

Code that unpacked a `get_cursor()` call into `size`, `hotspot`, `xormasks`, `andmasks` will still work, assuming the call returns an old school type cursor.

Changed in version 2.0.1: *Changed in version 2.0.1.*

pygame.mixer.music

pygame module for controlling streamed audio

The music module is closely tied to `pygame.mixer`. Use the music module to control the playback of music in the sound mixer.

The difference between the music playback and regular Sound playback is that the music is streamed, and never actually loaded all at once. The mixer system only supports a single music stream at once.

Be aware that MP3 support is limited. On some systems an unsupported format can crash the program, e.g. Debian Linux. Consider using OGG instead.

`pygame.mixer.music.load()`

Load a music file for playback

`load(filename)` -> None

`load(object)` -> None

This will load a music filename/file object and prepare it for playback. If a music stream is already playing it will be stopped. This does not start the music playing.

`pygame.mixer.music.unload()`

Unload the currently loaded music to free up resources

`unload()` -> None

This closes resources like files for any music that may be loaded.

New in version 2.0.0: *New in version 2.0.0.*

`pygame.mixer.music.play()`

Start the playback of the music stream

`play(loops=0, start=0.0, fade_ms=0)` -> None

This will play the loaded music stream. If the music is already playing it will be restarted.

`loops` is an optional integer argument, which is 0 by default, it tells how many times to repeat the music. The music repeats indefinitely if this argument is set to -1.

`start` is an optional float argument, which is 0.0 by default, which denotes the position in time, the music starts playing from. The starting position depends on the format of the music played. MP3 and OGG use the position as time in seconds. For mp3s the start time position selected may not be accurate as things like variable bit rate encoding and ID3 tags can throw off the timing calculations. For MOD music it is the pattern order number. Passing a start position will raise a `NotImplementedError` if the start position cannot be set.

`fade_ms` is an optional integer argument, which is 0 by default, makes the music start playing at 0 volume and fade up to full volume over the given time. The sample may end before the fade-in is complete.

Changed in version 2.0.0: *Changed in version 2.0.0:* Added optional `fade_ms` argument

`pygame.mixer.music.rewind()`

restart music

`rewind()` -> None

Resets playback of the current music to the beginning.

`pygame.mixer.music.stop()`

stop the music playback

`stop()` -> None

Stops the music playback if it is currently playing. It Won't Unload the music.

`pygame.mixer.music.pause()`

temporarily stop music playback

`pause()` -> None

Temporarily stop playback of the music stream. It can be resumed with the `pygame.mixer.music.unpause()` function.

`pygame.mixer.music.unpause()`

resume paused music

`unpause()` -> None

This will resume the playback of a music stream after it has been paused.

`pygame.mixer.music.fadeout()`

stop music playback after fading out

`fadeout(time)` -> None

Fade out and stop the currently playing music.

The `time` argument denotes the integer milliseconds for which the fading effect is generated.

Note, that this function blocks until the music has faded out. Calls to `fadeout()` and `set_volume()` will have no effect during this time. If an event was set using `set_endevent()` it will be called after the music has faded.

`pygame.mixer.music.set_volume()`

set the music volume

`set_volume(volume)` -> None

Set the volume of the music playback.

The `volume` argument is a float between 0.0 and 1.0 that sets volume. When new music is loaded the volume is reset to full volume.

`pygame.mixer.music.get_volume()`

get the music volume

`get_volume()` -> value

Returns the current volume for the mixer. The value will be between 0.0 and 1.0.

`pygame.mixer.music.get_busy()`

check if the music stream is playing

`get_busy()` -> bool

Returns True when the music stream is actively playing. When the music is idle this returns False. In pygame 2.0.1 and above this function returns False when the music is paused. In pygame 1 it returns True when the music is paused.

Changed in version 2.0.1: *Changed in version 2.0.1:* Returns False when music paused.

`pygame.mixer.music.set_pos()`

set position to play from

`set_pos(pos)` -> None

This sets the position in the music file where playback will start. The meaning of "pos", a float (or a number that can be converted to a float), depends on the music format.

For MOD files, pos is the integer pattern number in the module. For OGG it is the absolute position, in seconds, from the beginning of the sound. For MP3 files, it is the relative position, in seconds, from the current position. For absolute positioning in an MP3 file, first call `rewind()`.

Other file formats are unsupported. Newer versions of SDL_mixer have better positioning support than earlier ones. An `SDLError` is raised if a particular format does not support positioning.

Function `set_pos()` calls underlining SDL_mixer function `Mix_SetMusicPosition`.

New in version 1.9.2: *New in version 1.9.2.*

`pygame.mixer.music.get_pos()`

get the music play time

`get_pos()` -> time

This gets the number of milliseconds that the music has been playing for. The returned time only represents how long the music has been playing; it does not take into account any starting position offsets.

`pygame.mixer.music.queue()`

queue a sound file to follow the current

`queue(filename)` -> None

This will load a sound file and queue it. A queued sound file will begin as soon as the current sound naturally ends. Only one sound can be queued at a time. Queuing a new sound while another sound is queued will result in the new sound becoming the queued sound. Also, if the current sound is ever stopped or changed, the queued sound will be lost.

The following example will play music by Bach six times, then play music by Mozart once:

```
pygame.mixer.music.load('bach.ogg')
pygame.mixer.music.play(5)          # Plays six times, not five!
pygame.mixer.music.queue('mozart.ogg')
```

`pygame.mixer.music.set_endevent()`

have the music send an event when playback stops

`set_endevent()` -> None

`set_endevent(type)` -> None

This causes pygame to signal (by means of the event queue) when the music is done playing. The argument determines the type of event that will be queued.

The event will be queued every time the music finishes, not just the first time. To stop the event from being queued, call this method with no argument.

`pygame.mixer.music.get_endevent ()`

get the event a channel sends when playback stops

`get_endevent()` -> type

Returns the event type to be sent every time the music finishes playback. If there is no endevent the function returns `pygame.NOEVENT`.

pygame.Overlay

`pygame.Overlay`

pygame object for video overlay graphics

`Overlay(format, (width, height))` -> Overlay

The Overlay objects provide support for accessing hardware video overlays. Video overlays do not use standard RGB pixel formats, and can use multiple resolutions of data to create a single image.

The Overlay objects represent lower level access to the display hardware. To use the object you must understand the technical details of video overlays.

The Overlay format determines the type of pixel data used. Not all hardware will support all types of overlay formats. Here is a list of available format types:

`YV12_OVERLAY, IYUV_OVERLAY, YUY2_OVERLAY, UYVY_OVERLAY, YVYU_OVERLAY`

The width and height arguments control the size for the overlay image data. The overlay image can be displayed at any size, not just the resolution of the overlay.

The overlay objects are always visible, and always show above the regular display contents.

`display ()`

set the overlay pixel data

`display((y, u, v))` -> None

`display()` -> None

Display the YUV data in SDL's overlay planes. The y, u, and v arguments are strings of binary data. The data must be in the correct format used to create the Overlay.

If no argument is passed in, the Overlay will simply be redrawn with the current data. This can be useful when the Overlay is not really hardware accelerated.

The strings are not validated, and improperly sized strings could crash the program.

`set_location ()`

control where the overlay is displayed

`set_location(rect)` -> None

Set the location for the overlay. The overlay will always be shown relative to the main display Surface. This does not actually redraw the overlay, it will be updated on the next call to `Overlay.display()`.

`get_hardware ()`

test if the Overlay is hardware accelerated

`get_hardware(rect)` -> int

Returns a True value when the Overlay is hardware accelerated. If the platform does not support acceleration, software rendering is used.

pygame.PixelArray

`pygame.PixelArray`

pygame object for direct pixel access of surfaces

`PixelArray(Surface)` -> PixelArray

The `PixelArray` wraps a `Surface` and provides direct access to the surface's pixels. A pixel array can be one or two dimensional. A two dimensional array, like its surface, is indexed `[column, row]`. Pixel arrays support slicing, both for returning a subarray or for assignment. A pixel array sliced on a single column or row returns a one dimensional pixel array. Arithmetic and other operations are not supported. A pixel array can be safely assigned to itself. Finally, pixel arrays export an array struct interface, allowing them to interact with `pygame.pixelcopy` methods and NumPy arrays.

A `PixelArray` pixel item can be assigned a raw integer values, a `pygame.Color` instance, or a `(r, g, b[, a])` tuple.

```
pxarray[x, y] = 0xFF00FF
pxarray[x, y] = pygame.Color(255, 0, 255)
pxarray[x, y] = (255, 0, 255)
```

However, only a pixel's integer value is returned. So, to compare a pixel to a particular color the color needs to be first mapped using the `Surface.map_rgb()` method of the `Surface` object for which the `PixelArray` was created.

```
pxarray = pygame.PixelArray(surface)
# Check, if the first pixel at the topleft corner is blue
if pxarray[0, 0] == surface.map_rgb((0, 0, 255)):
    ...
```

When assigning to a range of pixels, a non tuple sequence of colors or a `PixelArray` can be used as the value. For a sequence, the length must match the `PixelArray` width.

```
pxarray[a:b] = 0xFF00FF          # set all pixels to 0xFF00FF
pxarray[a:b] = (0xFF00FF, 0xAACCEE, ...) # first pixel = 0xFF00FF,
                                         # second pixel = 0xAACCEE, ...
pxarray[a:b] = [(255, 0, 255), (170, 204, 238), ...] # same as above
pxarray[a:b] = [(255, 0, 255), 0xAACCEE, ...]         # same as above
pxarray[a:b] = otherarray[x:y]                        # slice sizes must match
```

For `PixelArray` assignment, if the right hand side array has a row length of 1, then the column is broadcast over the target array's rows. An array of height 1 is broadcast over the target's columns, and is equivalent to assigning a 1D `PixelArray`.

Subscript slices can also be used to assign to a rectangular subview of the target `PixelArray`.

```
# Create some new PixelArray objects providing a different view
# of the original array/surface.
newarray = pxarray[2:4, 3:5]
otherarray = pxarray[::2, ::2]
```

Subscript slices can also be used to do fast rectangular pixel manipulations instead of iterating over the x or y axis. The

```
pxarray[::2, :] = (0, 0, 0)          # Make even columns black.
pxarray[::2] = (0, 0, 0)             # Same as [::2, :]
```

During its lifetime, the `PixelArray` locks the surface, thus you explicitly have to `close()` it once its not used any more and the surface should perform operations in the same scope. It is best to use it as a context manager using the with `PixelArray(surf)` as `pixel_array`: style. So it works on pypy too.

A simple `:` slice index for the column can be omitted.

```
pxarray[::2, ...] = (0, 0, 0)        # Same as pxarray[::2, :]
pxarray[...] = (255, 0, 0)           # Same as pxarray[:]
```

A note about `PixelArray` to `PixelArray` assignment, for arrays with an item size of 3 (created from 24 bit surfaces) pixel values are translated from the source to the destinations format. The red, green, and blue color elements of each pixel are shifted to match the format of the target surface. For all other pixel sizes no such remapping occurs. This should change in later pygame releases, where format conversions are performed for all pixel sizes. To avoid code breakage when full mapped copying is implemented it is suggested `PixelArray` to `PixelArray` copies be only between surfaces of identical format.

New in version 1.9.4: *New in version 1.9.4:* `close()` method was added. For explicitly cleaning up. being able to use `PixelArray` as a context manager for cleanup. both of these are useful for when working without reference counting (pypy).

New in version 1.9.2: *New in version 1.9.2:* array struct interface transpose method broadcasting for a length 1 dimension

Changed in version 1.9.2: *Changed in version 1.9.2:* A 2D PixelArray can have a length 1 dimension. Only an integer index on a 2D PixelArray returns a 1D array. For assignment, a tuple can only be a color. Any other sequence type is a sequence of colors.

surface

Gets the Surface the PixelArray uses.
 surface -> Surface
 The Surface the PixelArray was created for.

itemsizes

Returns the byte size of a pixel array item
 itemsizes -> int
 This is the same as `Surface.get_bytesize()` for the pixel array's surface.
 New in version 1.9.2: *New in version 1.9.2.*

ndim

Returns the number of dimensions.
 ndim -> int
 A pixel array can be 1 or 2 dimensional.
 New in version 1.9.2: *New in version 1.9.2.*

shape

Returns the array size.
 shape -> tuple of int's
 A tuple or length `ndim` giving the length of each dimension. Analogous to `Surface.get_size()`.
 New in version 1.9.2: *New in version 1.9.2.*

strides

Returns byte offsets for each array dimension.
 strides -> tuple of int's
 A tuple or length `ndim` byte counts. When a stride is multiplied by the corresponding index it gives the offset of that index from the start of the array. A stride is negative for an array that has is inverted (has a negative step).
 New in version 1.9.2: *New in version 1.9.2.*

make_surface ()

Creates a new Surface from the current PixelArray.
 make_surface() -> Surface
 Creates a new Surface from the current PixelArray. Depending on the current PixelArray the size, pixel order etc. will be different from the original Surface.

```
# Create a new surface flipped around the vertical axis.
sf = pxarray[:,::-1].make_surface ()
```

New in version 1.8.1: *New in version 1.8.1.*

replace ()

Replaces the passed color in the PixelArray with another one.
 replace(color, repcolor, distance=0, weights=(0.299, 0.587, 0.114)) -> None
 Replaces the pixels with the passed color in the PixelArray by changing them them to the passed replacement color.
 It uses a simple weighted Euclidean distance formula to calculate the distance between the colors. The distance space ranges from 0.0 to 1.0 and is used as threshold for the color detection. This causes the replacement to take pixels with a similar, but not exactly identical color, into account as well.
 This is an in place operation that directly affects the pixels of the PixelArray.
 New in version 1.8.1: *New in version 1.8.1.*

extract ()

Extracts the passed color from the PixelArray.

`extract(color, distance=0, weights=(0.299, 0.587, 0.114))` -> PixelArray

Extracts the passed color by changing all matching pixels to white, while non-matching pixels are changed to black. This returns a new PixelArray with the black/white color mask.

It uses a simple weighted Euclidean distance formula to calculate the distance between the colors. The distance space ranges from 0.0 to 1.0 and is used as threshold for the color detection. This causes the extraction to take pixels with a similar, but not exactly identical color, into account as well.

New in version 1.8.1: *New in version 1.8.1.*

compare ()

Compares the PixelArray with another one.

`compare(array, distance=0, weights=(0.299, 0.587, 0.114))` -> PixelArray

Compares the contents of the PixelArray with those from the passed in PixelArray. It returns a new PixelArray with a black/white color mask that indicates the differences (black) of both arrays. Both PixelArray objects must have identical bit depths and dimensions.

It uses a simple weighted Euclidean distance formula to calculate the distance between the colors. The distance space ranges from 0.0 to 1.0 and is used as a threshold for the color detection. This causes the comparison to mark pixels with a similar, but not exactly identical color, as white.

New in version 1.8.1: *New in version 1.8.1.*

transpose ()

Exchanges the x and y axis.

`transpose()` -> PixelArray

This method returns a new view of the pixel array with the rows and columns swapped. So for a (w, h) sized array a (h, w) slice is returned. If an array is one dimensional, then a length 1 x dimension is added, resulting in a 2D pixel array.

New in version 1.9.2: *New in version 1.9.2.*

close ()

Closes the PixelArray, and releases Surface lock.

`transpose()` -> PixelArray

This method is for explicitly closing the PixelArray, and releasing a lock on the Surface.

New in version 1.9.4: *New in version 1.9.4.*

pygame.pixelcopy

pygame module for general pixel array copying

The `pygame.pixelcopy` module contains functions for copying between surfaces and objects exporting an array structure interface. It is a backend for `pygame.surfarray`, adding NumPy support. But `pixelcopy` is more general, and intended for direct use.

The array struct interface exposes an array's data in a standard way. It was introduced in NumPy. In Python 2.7 and above it is replaced by the new buffer protocol, though the buffer protocol is still a work in progress. The array struct interface, on the other hand, is stable and works with earlier Python versions. So for now the array struct interface is the predominate way pygame handles array introspection.

New in pygame 1.9.2.

`pygame.pixelcopy.surface_to_array ()`

copy surface pixels to an array object

`surface_to_array(array, surface, kind='P', opaque=255, clear=0)` -> None

The `surface_to_array` function copies pixels from a Surface object to a 2D or 3D array. Depending on argument `kind` and the target array dimension, a copy may be raw pixel value, RGB, a color component slice, or colorkey alpha transparency value. Recognized `kind` values are the single character codes 'P', 'R', 'G', 'B', 'A', and 'C'. Kind codes are case insensitive, so 'p' is equivalent to 'P'. The first two dimensions of the target must be the surface size (w, h).

The default 'P' kind code does a direct raw integer pixel (mapped) value copy to a 2D array and a 'RGB' pixel component (unmapped) copy to a 3D array having shape (w, h, 3). For an 8 bit colormap surface this means the

table index is copied to a 2D array, not the table value itself. A 2D array's item size must be at least as large as the surface's pixel byte size. The item size of a 3D array must be at least one byte.

For the 'R', 'G', 'B', and 'A' copy kinds a single color component of the unmapped surface pixels are copied to the target 2D array. For kind 'A' and surfaces with source alpha (the surface was created with the SRCALPHA flag), has a colorkey (set with `Surface.set_colorkey()`), or has a blanket alpha (set with `Surface.set_alpha()`) then the alpha values are those expected for a SDL surface. If a surface has no explicit alpha value, then the target array is filled with the value of the optional `opaque` surface_to_array argument (default 255: not transparent).

Copy kind 'C' is a special case for alpha copy of a source surface with colorkey. Unlike the 'A' color component copy, the `clear` argument value is used for colorkey matches, `opaque` otherwise. By default, a match has alpha 0 (totally transparent), while everything else is alpha 255 (totally opaque). It is a more general implementation of `pygame.surfarray.array_colorkey()`.

Specific to `surface_to_array`, a `ValueError` is raised for target arrays with incorrect shape or item size. A `TypeError` is raised for an incorrect kind code. Surface specific problems, such as locking, raise a `pygame.error`.

`pygame.pixelcopy.array_to_surface()`

copy an array object to a surface

`array_to_surface(<surface>, <array>)` -> None

See `pygame.surfarray.blit_array()`.

`pygame.pixelcopy.map_array()`

copy an array to another array, using surface format

`map_array(<array>, <array>, <surface>)` -> None

Map an array of color element values - (w, h, ..., 3) - to an array of pixels - (w, h) according to the format of <surface>.

`pygame.pixelcopy.make_surface()`

Copy an array to a new surface

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

Create a new Surface that best resembles the data and format of the array. The array can be 2D or 3D with any sized integer values.

pygame

the top level pygame package

The pygame package represents the top-level package for others to use. Pygame itself is broken into many submodules, but this does not affect programs that use pygame.

As a convenience, most of the top-level variables in pygame have been placed inside a module named `pygame.locals`. This is meant to be used with `from pygame.locals import *`, in addition to `import pygame`.

When you `import pygame` all available pygame submodules are automatically imported. Be aware that some of the pygame modules are considered *optional*, and may not be available. In that case, pygame will provide a placeholder object instead of the module, which can be used to test for availability.

`pygame.init()`

initialize all imported pygame modules

`init()` -> (numpass, numfail)

Initialize all imported pygame modules. No exceptions will be raised if a module fails, but the total number of successful and failed inits will be returned as a tuple. You can always initialize individual modules manually, but `pygame.init()` is a convenient way to get everything started. The `init()` functions for individual modules will raise exceptions when they fail.

You may want to initialize the different modules separately to speed up your program or to not use modules your game does not require.

It is safe to call this `init()` more than once as repeated calls will have no effect. This is true even if you have `pygame.quit()` all the modules.

`pygame.quit()`

uninitialize all pygame modules

`quit()` -> None

Uninitialize all pygame modules that have previously been initialized. When the Python interpreter shuts down, this method is called regardless, so your program should not need it, except when it wants to terminate its pygame resources and continue. It is safe to call this function more than once as repeated calls have no effect.

Note

Calling `pygame.quit()` will not exit your program. Consider letting your program end in the same way a normal Python program will end.

`pygame.get_init()`

returns True if pygame is currently initialized

`get_init()` -> bool

Returns True if pygame is currently initialized.

New in version 1.9.5: *New in version 1.9.5.*

exception `pygame.error`

standard pygame exception

`raise pygame.error(message)`

This exception is raised whenever a pygame or SDL operation fails. You can catch any anticipated problems and deal with the error. The exception is always raised with a descriptive message about the problem.

Derived from the `RuntimeError` exception, which can also be used to catch these raised errors.

`pygame.get_error()`

get the current error message

`get_error()` -> errorstr

SDL maintains an internal error message. This message will usually be given to you when `pygame.error()` is raised, so this function will rarely be needed.

`pygame.set_error()`

set the current error message

`set_error(error_msg)` -> None

SDL maintains an internal error message. This message will usually be given to you when `pygame.error()` is raised, so this function will rarely be needed.

`pygame.get_sdl_version()`

get the version number of SDL

`get_sdl_version()` -> major, minor, patch

Returns the three version numbers of the SDL library. This version is built at compile time. It can be used to detect which features may or may not be available through pygame.

New in version 1.7.0: *New in version 1.7.0.*

`pygame.get_sdl_byteorder()`

get the byte order of SDL

`get_sdl_byteorder()` -> int

Returns the byte order of the SDL library. It returns 1234 for little endian byte order and 4321 for big endian byte order.

New in version 1.8: *New in version 1.8.*

`pygame.register_quit()`

register a function to be called when pygame quits

`register_quit(callable)` -> None

When `pygame.quit()` is called, all registered quit functions are called. Pygame modules do this automatically when they are initializing, so this function will rarely be needed.

`pygame.encode_string()`

Encode a Unicode or bytes object

`encode_string([obj [, encoding [, errors [, etype]]]])` -> bytes or None

obj: If Unicode, encode; if bytes, return unaltered; if anything else, return None; if not given, raise `SyntaxError`.
 encoding (string): If present, encoding to use. The default is `'unicode_escape'`.
 errors (string): If given, how to handle unencodable characters. The default is `'backslashreplace'`.
 etype (exception type): If given, the exception type to raise for an encoding error. The default is `UnicodeEncodeError`, as returned by `PyUnicode_AsEncodedString()`. For the default encoding and errors values there should be no encoding errors.
 This function is used in encoding file paths. Keyword arguments are supported.
 New in version 1.9.2: *New in version 1.9.2*: (primarily for use in unit tests)

`pygame.encode_file_path()`

Encode a Unicode or bytes object as a file system path
`encode_file_path([obj [, etype]])` -> bytes or None
 obj: If Unicode, encode; if bytes, return unaltered; if anything else, return None; if not given, raise `SyntaxError`.
 etype (exception type): If given, the exception type to raise for an encoding error. The default is `UnicodeEncodeError`, as returned by `PyUnicode_AsEncodedString()`.
 This function is used to encode file paths in pygame. Encoding is to the codec as returned by `sys.getfilesystemencoding()`. Keyword arguments are supported.
 New in version 1.9.2: *New in version 1.9.2*: (primarily for use in unit tests)

pygame.version

small module containing version information

This module is automatically imported into the pygame package and can be used to check which version of pygame has been imported.

`pygame.version.ver`

version number as a string
`ver = '1.2'`
 This is the version represented as a string. It can contain a micro release number as well, e.g. `'1.5.2'`

`pygame.version.vernum`

tupled integers of the version
`vernum = (1, 5, 3)`
 This version information can easily be compared with other version numbers of the same format. An example of checking pygame version numbers would look like this:

```
if pygame.version.vernum < (1, 5):
    print('Warning, older version of pygame (%s)' % pygame.version.ver)
    disable_advanced_features = True
```

New in version 1.9.6: *New in version 1.9.6*: Attributes `major`, `minor`, and `patch`.

```
vernum.major == vernum[0]
vernum.minor == vernum[1]
vernum.patch == vernum[2]
```

Changed in version 1.9.6: *Changed in version 1.9.6*: `str(pygame.version.vernum)` returns a string like `"2.0.0"` instead of `"(2, 0, 0)"`.

Changed in version 1.9.6: *Changed in version 1.9.6*: `repr(pygame.version.vernum)` returns a string like `"PygameVersion(major=2, minor=0, patch=0)"` instead of `"(2, 0, 0)"`.

`pygame.version.rev`

repository revision of the build
`rev = 'a6f89747b551+'`
 The Mercurial node identifier of the repository checkout from which this package was built. If the identifier ends with a plus sign `'+'` then the package contains uncommitted changes. Please include this revision number in bug reports, especially for non-release pygame builds.

Important note: pygame development has moved to github, this variable is obsolete now. As soon as development shifted to github, this variable started returning an empty string "". It has always been returning an empty string since v1.9.5.

Changed in version 1.9.5: *Changed in version 1.9.5*: Always returns an empty string "".

pygame.version.SDL

tupled integers of the SDL library version

SDL = '(2, 0, 12)'

This is the SDL library version represented as an extended tuple. It also has attributes 'major', 'minor' & 'patch' that can be accessed like this:

```
>>> pygame.version.SDL.major
2
```

printing the whole thing returns a string like this:

```
>>> pygame.version.SDL
SDLVersion(major=2, minor=0, patch=12)
```

New in version 2.0.0: *New in version 2.0.0*.

Setting Environment Variables

Some aspects of pygame's behaviour can be controlled by setting environment variables, they cover a wide range of the library's functionality. Some of the variables are from pygame itself, while others come from the underlying C SDL library that pygame uses.

In python, environment variables are usually set in code like this:

```
import os
os.environ['NAME_OF_ENVIRONMENT_VARIABLE'] = 'value_to_set'
```

Or to preserve users ability to override the variable:

```
import os
os.environ['ENV_VAR'] = os.environ.get('ENV_VAR', 'value')
```

If the variable is more useful for users of an app to set than the developer then they can set it like this:

Windows:

```
set NAME_OF_ENVIRONMENT_VARIABLE=value_to_set
python my_application.py
```

Linux/Mac:

```
ENV_VAR=value python my_application.py
```

For some variables they need to be set before initialising pygame, some must be set before even importing pygame, and others can simply be set right before the area of code they control is run.

Below is a list of environment variables, their settable values, and a brief description of what they do.

Pygame Environment Variables

These variables are defined by pygame itself.

```
PYGAME_DISPLAY - Experimental (subject to change)
Set index of the display to use, "0" is the default.
```

This sets the display where pygame will open its window or screen. The value set here will be used if set before calling `pygame.display.set_mode()`, and as long as no 'display' parameter is passed into `pygame.display.set_mode()`.

```
PYGAME_FORCE_SCALE -
Set to "photo" or "default".
```

This forces `set_mode()` to use the SCALED display mode and, if "photo" is set, makes the scaling use the slowest, but highest quality anisotropic scaling algorithm, if it is available. Must be set before calling `pygame.display.set_mode()`.

```
PYGAME_BLEND_ALPHA_SDL2 - New in pygame 2.0.0
Set to "1" to enable the SDL2 blitter.
```

This makes pygame use the SDL2 blitter for all alpha blending. The SDL2 blitter is sometimes faster than the default blitter but uses a different formula so the final colours may differ. Must be set before `pygame.init()` is called.

```
PYGAME_HIDE_SUPPORT_PROMPT -
Set to "1" to hide the prompt.
```

This stops the welcome message popping up in the console that tells you which version of python, pygame & SDL you are using. Must be set before importing pygame.

```
PYGAME_FREETYPE -
Set to "1" to enable.
```

This switches the `pygame.font` module to a pure freetype implementation that bypasses `SDL_ttf`. See the font module for why you might want to do this. Must be set before importing pygame.

```
PYGAME_CAMERA -
Set to "opencv" or "vidcapture"
```

Forces the library backend used in the camera module, overriding the platform defaults. Must be set before calling `pygame.camera.init()`.

SDL Environment Variables

These variables are defined by SDL.

For documentation on the environment variables available in pygame 1 try [here](#). For Pygame 2, some selected environment variables are listed below.

```
SDL_VIDEO_CENTERED -
Set to "1" to enable centering the window.
```

This will make the pygame window open in the centre of the display. Must be set before calling `pygame.display.set_mode()`.

```
SDL_VIDEO_WINDOW_POS -
Set to "x,y" to position the top left corner of the window.
```

This allows control over the placement of the pygame window within the display. Must be set before calling `pygame.display.set_mode()`.

```
SDL_VIDEODRIVER -
Set to "drivername" to change the video driver used.
```

On some platforms there are multiple video drivers available and this allows users to pick between them. More information is available [here](#). Must be set before calling `pygame.init()` or `pygame.display.init()`.

```
SDL_AUDIODRIVER -
Set to "drivername" to change the audio driver used.
```

On some platforms there are multiple audio drivers available and this allows users to pick between them. More information is available [here](#). Must be set before calling `pygame.init()` or `pygame.mixer.init()`.

```
SDL_VIDEO_ALLOW_SCREENSAVER
Set to "1" to allow screensavers while pygame apps are running.
```

By default pygame apps disable screensavers while they are running. Setting this environment variable allows users or developers to change that and make screensavers run again.

```
SDL_VIDEO_X11_NET_WM_BYPASS_COMPOSITOR
Set to "0" to re-enable the compositor.
```

By default SDL tries to disable the X11 compositor for all pygame apps. This is usually a good thing as it's faster, however if you have an app which *doesn't* update every frame and are using linux you may want to disable this bypass. The bypass has reported problems on KDE linux. This variable is only used on x11/linux platforms.

pygame.Rect

pygame.Rect

pygame object for storing rectangular coordinates

`Rect(left, top, width, height) -> Rect`

`Rect((left, top), (width, height)) -> Rect`

`Rect(object) -> Rect`

Pygame uses Rect objects to store and manipulate rectangular areas. A Rect can be created from a combination of left, top, width, and height values. Rects can also be created from python objects that are already a Rect or have an attribute named "rect".

Any pygame function that requires a Rect argument also accepts any of these values to construct a Rect. This makes it easier to create Rects on the fly as arguments to functions.

The Rect functions that change the position or size of a Rect return a new copy of the Rect with the affected changes. The original Rect is not modified. Some methods have an alternate "in-place" version that returns None but affects the original Rect. These "in-place" methods are denoted with the "ip" suffix.

The Rect object has several virtual attributes which can be used to move and align the Rect:

```
x,y
top, left, bottom, right
topleft, bottomleft, topright, bottomright
midtop, midleft, midbottom, midright
center, centerx, centery
size, width, height
w,h
```

All of these attributes can be assigned to:

```
rect1.right = 10
rect2.center = (20,30)
```

Assigning to size, width or height changes the dimensions of the rectangle; all other assignments move the rectangle without resizing it. Notice that some attributes are integers and others are pairs of integers.

If a Rect has a nonzero width or height, it will return `True` for a nonzero test. Some methods return a Rect with 0 size to represent an invalid rectangle. A Rect with a 0 size will not collide when using collision detection methods (e.g. `collidepoint()`, `colliderect()`, etc.).

The coordinates for Rect objects are all integers. The size values can be programmed to have negative values, but these are considered illegal Rects for most operations.

There are several collision tests between other rectangles. Most python containers can be searched for collisions against a single Rect.

The area covered by a Rect does not include the right- and bottom-most edge of pixels. If one Rect's bottom border is another Rect's top border (i.e., `rect1.bottom=rect2.top`), the two meet exactly on the screen but do not overlap, and `rect1.collidect(rect2)` returns false.

New in version 1.9.2: *New in version 1.9.2*: The Rect class can be subclassed. Methods such as `copy()` and `move()` will recognize this and return instances of the subclass. However, the subclass's `__init__()` method is not called, and `__new__()` is assumed to take no arguments. So these methods should be overridden if any extra attributes need to be copied.

copy()

copy the rectangle

`copy()` -> Rect

Returns a new rectangle having the same position and size as the original.

New in pygame 1.9

move()

moves the rectangle

`move(x, y)` -> Rect

Returns a new rectangle that is moved by the given offset. The x and y arguments can be any integer value, positive or negative.

move_ip()

moves the rectangle, in place

`move_ip(x, y)` -> None

Same as the `Rect.move()` method, but operates in place.

inflate()

grow or shrink the rectangle size

`inflate(x, y)` -> Rect

Returns a new rectangle with the size changed by the given offset. The rectangle remains centered around its current center. Negative values will shrink the rectangle. Note, uses integers, if the offset given is too small (< 2 > -2), center will be off.

inflate_ip()

grow or shrink the rectangle size, in place

`inflate_ip(x, y)` -> None

Same as the `Rect.inflate()` method, but operates in place.

update()

sets the position and size of the rectangle

`update(left, top, width, height)` -> None

`update((left, top), (width, height))` -> None

`update(object)` -> None

Sets the position and size of the rectangle, in place. See parameters for `pygame.Rect()` for the parameters of this function.

New in version 2.0.1: *New in version 2.0.1*.

clamp()

moves the rectangle inside another

`clamp(Rect)` -> Rect

Returns a new rectangle that is moved to be completely inside the argument Rect. If the rectangle is too large to fit inside, it is centered inside the argument Rect, but its size is not changed.

clamp_ip()

moves the rectangle inside another, in place
 clamp_ip(Rect) -> None
 Same as the Rect.clamp() method, but operates in place.

clip()

crops a rectangle inside another
 clip(Rect) -> Rect
 Returns a new rectangle that is cropped to be completely inside the argument Rect. If the two rectangles do not overlap to begin with, a Rect with 0 size is returned.

clipline()

crops a line inside a rectangle
 clipline(x1, y1, x2, y2) -> ((cx1, cy1), (cx2, cy2))
 clipline(x1, y1, x2, y2) -> ()
 clipline((x1, y1), (x2, y2)) -> ((cx1, cy1), (cx2, cy2))
 clipline((x1, y1), (x2, y2)) -> ()
 clipline((x1, y1, x2, y2)) -> ((cx1, cy1), (cx2, cy2))
 clipline((x1, y1, x2, y2)) -> ()
 clipline(((x1, y1), (x2, y2))) -> ((cx1, cy1), (cx2, cy2))
 clipline(((x1, y1), (x2, y2))) -> ()
 Returns the coordinates of a line that is cropped to be completely inside the rectangle. If the line does not overlap the rectangle, then an empty tuple is returned.
 The line to crop can be any of the following formats (floats can be used in place of ints, but they will be truncated):

- four ints
- 2 lists/tuples/Vector2s of 2 ints
- a list/tuple of four ints
- a list/tuple of 2 lists/tuples/Vector2s of 2 ints

Returns: a tuple with the coordinates of the given line cropped to be completely inside the rectangle is returned, if the given line does not overlap the rectangle, an empty tuple is returned

Return type: tuple(tuple(int, int), tuple(int, int)) or ()

Raises: **TypeError** -- if the line coordinates are not given as one of the above described line formats

Note

This method can be used for collision detection between a rect and a line. See example code below.

Note

The rect.bottom and rect.right attributes of a **pygame.Rect** always lie one pixel outside of its actual border.

```
# Example using clipline().
clipped_line = rect.clineline(line)

if clipped_line:
    # If clipped_line is not an empty tuple then the line
    # collides/overlaps with the rect. The returned value contains
    # the endpoints of the clipped line.
    start, end = clipped_line
    x1, y1 = start
    x2, y2 = end
```



```
else:
    print("No clipping. The line is fully outside the rect.")
```

New in version 2.0.0: *New in version 2.0.0.*

union ()

joins two rectangles into one

union(Rect) -> Rect

Returns a new rectangle that completely covers the area of the two provided rectangles. There may be area inside the new Rect that is not covered by the originals.

union_ip ()

joins two rectangles into one, in place

union_ip(Rect) -> None

Same as the Rect.union() method, but operates in place.

unionall ()

the union of many rectangles

unionall(Rect_sequence) -> Rect

Returns the union of one rectangle with a sequence of many rectangles.

unionall_ip ()

the union of many rectangles, in place

unionall_ip(Rect_sequence) -> None

The same as the Rect.unionall() method, but operates in place.

fit ()

resize and move a rectangle with aspect ratio

fit(Rect) -> Rect

Returns a new rectangle that is moved and resized to fit another. The aspect ratio of the original Rect is preserved, so the new rectangle may be smaller than the target in either width or height.

normalize ()

correct negative sizes

normalize() -> None

This will flip the width or height of a rectangle if it has a negative size. The rectangle will remain in the same place, with only the sides swapped.

contains ()

test if one rectangle is inside another

contains(Rect) -> bool

Returns true when the argument is completely inside the Rect.

collidepoint ()

test if a point is inside a rectangle

collidepoint(x, y) -> bool

collidepoint((x,y)) -> bool

Returns true if the given point is inside the rectangle. A point along the right or bottom edge is not considered to be inside the rectangle.

Note

For collision detection between a rect and a line the **clipline()** method can be used.

collidect ()

test if two rectangles overlap

`colliderect(Rect)` -> bool

Returns true if any portion of either rectangle overlap (except the top+bottom or left+right edges).

Note

For collision detection between a rect and a line the `clipline()` method can be used.

`collidelist ()`

test if one rectangle in a list intersects

`collidelist(list)` -> index

Test whether the rectangle collides with any in a sequence of rectangles. The index of the first collision found is returned. If no collisions are found an index of -1 is returned.

`collidelistall ()`

test if all rectangles in a list intersect

`collidelistall(list)` -> indices

Returns a list of all the indices that contain rectangles that collide with the Rect. If no intersecting rectangles are found, an empty list is returned.

`collidedict ()`

test if one rectangle in a dictionary intersects

`collidedict(dict)` -> (key, value)

`collidedict(dict)` -> None

`collidedict(dict, use_values=0)` -> (key, value)

`collidedict(dict, use_values=0)` -> None

Returns the first key and value pair that intersects with the calling Rect object. If no collisions are found, None is returned. If `use_values` is 0 (default) then the dict's keys will be used in the collision detection, otherwise the dict's values will be used.

Note

Rect objects cannot be used as keys in a dictionary (they are not hashable), so they must be converted to a tuple/list. e.g. `rect.collidedict({tuple(key_rect) : value})`

`collidedictall ()`

test if all rectangles in a dictionary intersect

`collidedictall(dict)` -> [(key, value), ...]

`collidedictall(dict, use_values=0)` -> [(key, value), ...]

Returns a list of all the key and value pairs that intersect with the calling Rect object. If no collisions are found an empty list is returned. If `use_values` is 0 (default) then the dict's keys will be used in the collision detection, otherwise the dict's values will be used.

Note

Rect objects cannot be used as keys in a dictionary (they are not hashable), so they must be converted to a tuple/list. e.g. `rect.collidedictall({tuple(key_rect) : value})`

pygame.scrap

pygame module for clipboard support.

EXPERIMENTAL!: This API may change or disappear in later pygame releases. If you use this, your code may break with the next pygame release.

The scrap module is for transferring data to/from the clipboard. This allows for cutting and pasting data between pygame and other applications. Some basic data (MIME) types are defined and registered:

pygame constant	string value	description

SCRAP_TEXT	"text/plain"	plain text
SCRAP_BMP	"image/bmp"	BMP encoded image data
SCRAP_PBM	"image/pbm"	PBM encoded image data
SCRAP_PPM	"image/ppm"	PPM encoded image data

pygame.SCRAP_PPM, pygame.SCRAP_PBM and pygame.SCRAP_BMP are suitable for surface buffers to be shared with other applications. pygame.SCRAP_TEXT is an alias for the plain text clipboard type.

Depending on the platform, additional types are automatically registered when data is placed into the clipboard to guarantee a consistent sharing behaviour with other applications. The following listed types can be used as strings to be passed to the respective **pygame.scrap** module functions.

For **Windows** platforms, these additional types are supported automatically and resolve to their internal definitions:

"text/plain;charset=utf-8"	UTF-8 encoded text
"audio/wav"	WAV encoded audio
"image/tiff"	TIFF encoded image data

For **X11** platforms, these additional types are supported automatically and resolve to their internal definitions:

"text/plain;charset=utf-8"	UTF-8 encoded text
"UTF8_STRING"	UTF-8 encoded text
"COMPOUND_TEXT"	COMPOUND text

User defined types can be used, but the data might not be accessible by other applications unless they know what data type to look for. Example: Data placed into the clipboard by `pygame.scrap.put("my_data_type", byte_data)` can only be accessed by applications which query the clipboard for the "my_data_type" data type.

For an example of how the scrap module works refer to the examples page ([pygame.examples.scrap_clipboard.main\(\)](#)) or the code directly in [GitHub \(pygame/examples/scrap_clipboard.py\)](#).

New in version 1.8: *New in version 1.8.*

Note

The scrap module is currently only supported for Windows, X11 and Mac OS X. On Mac OS X only text works at the moment - other types may be supported in future releases.

`pygame.scrap.init()`

Initializes the scrap module.

init() -> None

Initialize the scrap module.

Raises: `pygame.error` -- if unable to initialize scrap module

Note

The scrap module requires `pygame.display.set_mode()` be called before being initialized.

`pygame.scrap.get_init()`

Returns True if the scrap module is currently initialized.

get_init() -> bool

Gets the scrap module's initialization state.

Returns: True if the `pygame.scrap` module is currently initialized, False otherwise

Return type: bool

New in version 1.9.5: *New in version 1.9.5.*

`pygame.scrap.get()`

Gets the data for the specified type from the clipboard.

get(type) -> bytes or str or None

Retrieves the data for the specified type from the clipboard. In python 3 the data is returned as a byte string and might need further processing (such as decoding to Unicode).

Parameters: **type** (*string*) -- data type to retrieve from the clipboard

Returns: data (byte string in python 3 or str in python 2) for the given type identifier or None if no data for the given type is available

Return type: bytes or str or None

```
text = pygame.scrap.get(pygame.SCRAP_TEXT)
if text:
    print("There is text in the clipboard.")
else:
    print("There does not seem to be text in the clipboard.")
```

`pygame.scrap.get_types()`

Gets a list of the available clipboard types.

get_types() -> list

Gets a list of data type string identifiers for the data currently available on the clipboard. Each identifier can be used in the `pygame.scrap.get()` method to get the clipboard content of the specific type.

Returns: list of strings of the available clipboard data types, if there is no data in the clipboard an empty list is returned

Return type: list

```
for t in pygame.scrap.get_types():
    if "text" in t:
        # There is some content with the word "text" in its type string.
        print(pygame.scrap.get(t))
```

`pygame.scrap.put()`

Places data into the clipboard.

put(type, data) -> None

Places data for a given clipboard type into the clipboard. The data must be a string buffer. The type is a string identifying the type of data to be placed into the clipboard. This can be one of the predefined `pygame.SCRAP_PBM`, `pygame.SCRAP_PPM`, `pygame.SCRAP_BMP` or `pygame.SCRAP_TEXT` values or a user defined string identifier.

Parameters:

- **type** (*string*) -- type identifier of the data to be placed into the clipboard
- **data** (*bytes or str*) -- data to be place into the clipboard (in python 3 data is a byte string and in python 2 data is a str)

Raises: `pygame.error` -- if unable to put the data into the clipboard

```
with open("example.bmp", "rb") as fp:
    pygame.scrap.put(pygame.SCRAP_BMP, fp.read())
# The image data is now on the clipboard for other applications to access
# it.
pygame.scrap.put(pygame.SCRAP_TEXT, b"A text to copy")
pygame.scrap.put("Plain text", b"Data for user defined type 'Plain text'")
```

pygame.sndarray

pygame.scrap.contains ()

Checks whether data for a given type is available in the clipboard.

contains(type) -> bool

Checks whether data for the given type is currently available in the clipboard.

Parameters: **type** (*string*) -- data type to check availability of

Returns: True if data for the passed type is available in the clipboard, False otherwise

Return type: bool

```
if pygame.scrap.contains(pygame.SCRAP_TEXT):
    print("There is text in the clipboard.")
if pygame.scrap.contains("own_data_type"):
    print("There is stuff in the clipboard.")
```

pygame.scrap.lost ()

Indicates if the clipboard ownership has been lost by the pygame application.

lost() -> bool

Indicates if the clipboard ownership has been lost by the pygame application.

Returns: True, if the clipboard ownership has been lost by the pygame application, False if the pygame application still owns the clipboard

Return type: bool

```
if pygame.scrap.lost():
    print("The clipboard is in use by another application.")
```

pygame.scrap.set_mode ()

Sets the clipboard access mode.

set_mode(mode) -> None

Sets the access mode for the clipboard. This is only of interest for X11 environments where clipboard modes `pygame.SCRAP_SELECTION` (for mouse selections) and `pygame.SCRAP_CLIPBOARD` (for the clipboard) are available. Setting the mode to `pygame.SCRAP_SELECTION` in other environments will not change the mode from `pygame.SCRAP_CLIPBOARD`.

Parameters: **mode** -- access mode, supported values are `pygame.SCRAP_CLIPBOARD` and `pygame.SCRAP_SELECTION` (`pygame.SCRAP_SELECTION` only has an effect when used on X11 platforms)

Raises: **ValueError** -- if the mode parameter is not `pygame.SCRAP_CLIPBOARD` or `pygame.SCRAP_SELECTION`

pygame.sndarray

pygame module for accessing sound sample data

Functions to convert between NumPy arrays and Sound objects. This module will only be available when pygame can use the external NumPy package.

Sound data is made of thousands of samples per second, and each sample is the amplitude of the wave at a particular moment in time. For example, in 22-kHz format, element number 5 of the array is the amplitude of the wave after 5/22000 seconds.

Each sample is an 8-bit or 16-bit integer, depending on the data format. A stereo sound file has two values per sample, while a mono sound file only has one.

pygame.sndarray.array ()

copy Sound samples into an array

array(Sound) -> array

Creates a new array for the sound data and copies the samples. The array will always be in the format returned from `pygame.mixer.get_init()`.

`pygame.sndarray.samples()`

reference Sound samples into an array

`samples(Sound) -> array`

Creates a new array that directly references the samples in a Sound object. Modifying the array will change the Sound. The array will always be in the format returned from `pygame.mixer.get_init()`.

`pygame.sndarray.make_sound()`

convert an array into a Sound object

`make_sound(array) -> Sound`

Create a new playable Sound object from an array. The mixer module must be initialized and the array format must be similar to the mixer audio format.

`pygame.sndarray.use_arraytype()`

Sets the array system to be used for sound arrays

`use_arraytype(arraytype) -> None`

DEPRECATED: Uses the requested array type for the module functions. The only supported arraytype is 'numpy'. Other values will raise ValueError.

`pygame.sndarray.get_arraytype()`

Gets the currently active array type.

`get_arraytype() -> str`

DEPRECATED: Returns the currently active array type. This will be a value of the `get_arraytypes()` tuple and indicates which type of array module is used for the array creation.

New in version 1.8: *New in version 1.8.*

`pygame.sndarray.get_arraytypes()`

Gets the array system types currently supported.

`get_arraytypes() -> tuple`

DEPRECATED: Checks, which array systems are available and returns them as a tuple of strings. The values of the tuple can be used directly in the `pygame.sndarray.use_arraytype()` method. If no supported array system could be found, None will be returned.

New in version 1.8: *New in version 1.8.*

pygame.sprite

pygame module with basic game object classes

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

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

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

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

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

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

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

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

pygame.sprite.Sprite

Simple base class for visible game objects.

`Sprite(*groups) -> Sprite`

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

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

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

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

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

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

update ()

method to control sprite behavior

`update(*args, **kwargs) -> None`

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

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

add ()

add the sprite to groups

`add(*groups) -> None`

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

remove ()

remove the sprite from groups

`remove(*groups) -> None`

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

kill ()

remove the `Sprite` from all `Groups`

`kill() -> None`

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

alive ()

does the sprite belong to any groups

alive() -> bool

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

groups ()

list of Groups that contain this Sprite

groups() -> group_list

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

pygame.sprite.DirtySprite

A subclass of Sprite with more attributes and features.

DirtySprite(*groups) -> DirtySprite

Extra DirtySprite attributes with their default values:

dirty = 1

```
if set to 1, it is repainted and then set to 0 again
if set to 2 then it is always dirty ( repainted each frame,
flag is not reset)
0 means that it is not dirty and therefore not repainted again
```

blendmode = 0

```
its the special_flags argument of blit, blendmodes
```

source_rect = None

```
source rect to use, remember that it is relative to
topleft (0,0) of self.image
```

visible = 1

```
normally 1, if set to 0 it will not be repainted
(you must set it dirty too to be erased from screen)
```

layer = 0

```
(READONLY value, it is read when adding it to the
LayeredDirty, for details see doc of LayeredDirty)
```

pygame.sprite.Group

A container class to hold and manage multiple Sprite objects.

Group(*sprites) -> Group

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

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

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

sprites ()

list of the Sprites this Group contains

sprites() -> sprite_list

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

copy ()

duplicate the Group

`copy()` -> Group

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

add ()

add Sprites to this Group

`add(*sprites)` -> None

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

Each sprite argument can also be a iterator containing Sprites.

remove ()

remove Sprites from the Group

`remove(*sprites)` -> None

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

Each sprite argument can also be a iterator containing Sprites.

has ()

test if a Group contains Sprites

`has(*sprites)` -> bool

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

Each sprite argument can also be a iterator containing Sprites.

update ()

call the update method on contained Sprites

`update(*args, **kwargs)` -> None

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

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

draw ()

blit the Sprite images

`draw(Surface)` -> None

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

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

clear ()

draw a background over the Sprites

`clear(Surface_dest, background)` -> None

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

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

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

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

empty ()

remove all Sprites

`empty()` -> None
Removes all Sprites from this Group.

`pygame.sprite.RenderPlain`

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

`pygame.sprite.RenderClear`

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

`pygame.sprite.RenderUpdates`

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

`draw()`

blit the Sprite images and track changed areas
`draw(surface)` -> `Rect_list`
Draws all the Sprites to the surface, the same as `Group.draw()`. This method also returns a list of Rectangular areas on the screen that have been changed. The returned changes include areas of the screen that have been affected by previous `Group.clear()` calls.
The returned Rect list should be passed to `pygame.display.update()`. This will help performance on software driven display modes. This type of updating is usually only helpful on destinations with non-animating backgrounds.

`pygame.sprite.OrderedUpdates()`

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

`pygame.sprite.LayeredUpdates`

`LayeredUpdates` is a sprite group that handles layers and draws like `OrderedUpdates`.
`LayeredUpdates(*sprites, **kwargs)` -> `LayeredUpdates`
This group is fully compatible with `pygame.sprite.Sprite`.
You can set the default layer through kwargs using 'default_layer' and an integer for the layer. The default layer is 0.
If the sprite you add has an attribute `_layer` then that layer will be used. If the `**kwargs` contains 'layer' then the sprites passed will be added to that layer (overriding the `sprite.layer` attribute). If neither sprite has attribute `layer` nor `**kwargs` then the default layer is used to add the sprites.
New in version 1.8: *New in version 1.8.*

`add()`

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

`sprites()`

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

`draw()`

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

get_sprites_at ()

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

get_sprite ()

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

remove_sprites_of_layer ()

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

layers ()

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

change_layer ()

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

get_layer_of_sprite ()

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

get_top_layer ()

returns the top layer
get_top_layer() -> layer

get_bottom_layer ()

returns the bottom layer
get_bottom_layer() -> layer

move_to_front ()

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

move_to_back ()

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

get_top_sprite ()

returns the topmost sprite
get_top_sprite() -> Sprite

get_sprites_from_layer ()

returns all sprites from a layer, ordered by how they were added

`get_sprites_from_layer(layer) -> sprites`

Returns all sprites from a layer, ordered by how they were added. It uses linear search and the sprites are not removed from layer.

switch_layer ()

switches the sprites from layer1 to layer2

`switch_layer(layer1_nr, layer2_nr) -> None`

The layers number must exist, it is not checked.

pygame.sprite.LayeredDirty

LayeredDirty group is for DirtySprite objects. Subclasses LayeredUpdates.

`LayeredDirty(*sprites, **kwargs) -> LayeredDirty`

This group requires **pygame.sprite.DirtySprite** or any sprite that has the following attributes:

`image, rect, dirty, visible, blendmode` (see doc of DirtySprite).

It uses the dirty flag technique and is therefore faster than the **pygame.sprite.RenderUpdates** if you have many static sprites. It also switches automatically between dirty rect update and full screen drawing, so you do not have to worry what would be faster.

Same as for the **pygame.sprite.Group**. You can specify some additional attributes through kwargs:

`_use_update: True/False` default is False
`_default_layer: default layer where sprites without a layer are added.`
`_time_threshold: threshold time for switching between dirty rect mode and fullscreen mode, defaults to 1000./80 == 1000./fps`

New in version 1.8: *New in version 1.8.*

draw ()

draw all sprites in the right order onto the passed surface.

`draw(surface, bgd=None) -> Rect_list`

You can pass the background too. If a background is already set, then the bgd argument has no effect.

clear ()

used to set background

`clear(surface, bgd) -> None`

repaint_rect ()

repaints the given area

`repaint_rect(screen_rect) -> None`

screen_rect is in screen coordinates.

set_clip ()

clip the area where to draw. Just pass None (default) to reset the clip

`set_clip(screen_rect=None) -> None`

get_clip ()

clip the area where to draw. Just pass None (default) to reset the clip

`get_clip() -> Rect`

change_layer ()

changes the layer of the sprite

`change_layer(sprite, new_layer) -> None`

sprite must have been added to the renderer. It is not checked.

set_timing_treshold ()

sets the threshold in milliseconds

`set_timing_treshold(time_ms) -> None`

Default is 1000./80 where 80 is the fps I want to switch to full screen mode. This method's name is a typo and should be fixed.

Raises: **TypeError** -- if `time_ms` is not int or float

`pygame.sprite.GroupSingle()`

Group container that holds a single sprite.

`GroupSingle(sprite=None) -> GroupSingle`

The `GroupSingle` container only holds a single `Sprite`. When a new `Sprite` is added, the old one is removed.

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

`pygame.sprite.spritecollide()`

Find sprites in a group that intersect another sprite.

`spritecollide(sprite, group, dokill, collided = None) -> Sprite_list`

Return a list containing all `Sprites` in a `Group` that intersect with another `Sprite`. Intersection is determined by comparing the `Sprite.rect` attribute of each `Sprite`.

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

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

collided callables:

```
collide_rect, collide_rect_ratio, collide_circle,
collide_circle_ratio, collide_mask
```

Example:

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

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

`pygame.sprite.collide_rect()`

Collision detection between two sprites, using rects.

`collide_rect(left, right) -> bool`

Tests for collision between two sprites. Uses the `pygame.rect.collidect` function to calculate the collision. Intended to be passed as a `collided` callback function to the `*collide` functions. Sprites must have a "rect" attributes.

New in version 1.8: *New in version 1.8.*

`pygame.sprite.collide_rect_ratio()`

Collision detection between two sprites, using rects scaled to a ratio.

`collide_rect_ratio(ratio) -> collided_callable`

A callable class that checks for collisions between two sprites, using a scaled version of the sprites rects.

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

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

New in version 1.8.1: *New in version 1.8.1.*

`pygame.sprite.collide_circle()`

Collision detection between two sprites, using circles.

`collide_circle(left, right) -> bool`

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

New in version 1.8.1: *New in version 1.8.1.*

`pygame.sprite.collide_circle_ratio()`

Collision detection between two sprites, using circles scaled to a ratio.

`collide_circle_ratio(ratio)` -> `collided_callable`

A callable class that checks for collisions between two sprites, using a scaled version of the sprites radius.

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

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

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

New in version 1.8.1: *New in version 1.8.1.*

`pygame.sprite.collide_mask()`

Collision detection between two sprites, using masks.

`collide_mask(sprite1, sprite2)` -> (int, int)

`collide_mask(sprite1, sprite2)` -> None

Tests for collision between two sprites, by testing if their bitmasks overlap (uses `pygame.mask.Mask.overlap()`). If the sprites have a `mask` attribute, it is used as the mask, otherwise a mask is created from the sprite's image (uses `pygame.mask.from_surface()`). Sprites must have a `rect` attribute; the `mask` attribute is optional.

The first point of collision between the masks is returned. The collision point is offset from `sprite1`'s mask's topleft corner (which is always (0, 0)). The collision point is a position within the mask and is not related to the actual screen position of `sprite1`.

This function is intended to be passed as a collided callback function to the group collide functions (see `spritecollide()`, `groupcollide()`, `spritecollideany()`).

Note

To increase performance, create and set a `mask` attribute for all sprites that will use this function to check for collisions. Otherwise, each time this function is called it will create new masks.

Note

A new mask needs to be recreated each time a sprite's image is changed (e.g. if a new image is used or the existing image is rotated).

```
# Example of mask creation for a sprite.
sprite.mask = pygame.mask.from_surface(sprite.image)
```

Returns: first point of collision between the masks or None if no collision

Return type: tuple(int, int) or NoneType

New in version 1.8.0: *New in version 1.8.0.*

`pygame.sprite.groupcollide()`

Find all sprites that collide between two groups.

`groupcollide(group1, group2, dokill1, dokill2, collided = None)` -> `Sprite_dict`

This will find collisions between all the Sprites in two groups. Collision is determined by comparing the `Sprite.rect` attribute of each Sprite or by using the collided function if it is not None.

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

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

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

`pygame.sprite.spritecollideany()`

Simple test if a sprite intersects anything in a group.

`spritecollideany(sprite, group, collided = None)` -> Sprite Collision with the returned sprite.

`spritecollideany(sprite, group, collided = None)` -> None No collision

If the sprite collides with any single sprite in the group, a single sprite from the group is returned. On no collision None is returned.

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

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

pygame.Surface

pygame.Surface

pygame object for representing images

`Surface((width, height), flags=0, depth=0, masks=None)` -> Surface

`Surface((width, height), flags=0, Surface)` -> Surface

A pygame Surface is used to represent any image. The Surface has a fixed resolution and pixel format. Surfaces with 8-bit pixels use a color palette to map to 24-bit color.

Call `pygame.Surface()` to create a new image object. The Surface will be cleared to all black. The only required arguments are the sizes. With no additional arguments, the Surface will be created in a format that best matches the display Surface.

The pixel format can be controlled by passing the bit depth or an existing Surface. The flags argument is a bitmask of additional features for the surface. You can pass any combination of these flags:

<code>HWSURFACE</code>	creates the image in video memory
<code>SRCALPHA</code>	the pixel format will include a per-pixel alpha

Both flags are only a request, and may not be possible for all displays and formats.

Advance users can combine a set of bitmasks with a depth value. The masks are a set of 4 integers representing which bits in a pixel will represent each color. Normal Surfaces should not require the masks argument.

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 they will use highly optimized software blitting methods.

There are three types of transparency supported in pygame: colorkeys, surface alphas, and pixel alphas. Surface alphas can be mixed with colorkeys, but an image with per pixel alphas cannot use the other modes. Colorkey transparency makes a single color value transparent. Any pixels matching the colorkey will not be drawn. The surface alpha value is a single value that changes the transparency for the entire image. A surface alpha of 255 is opaque, and a value of 0 is completely transparent.

Per pixel alphas are different because they store a transparency value for every pixel. This allows for the most precise transparency effects, but it also the slowest. Per pixel alphas cannot be mixed with surface alpha and colorkeys.

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 a `pygame.PixelArray`, which gives an array like view of the surface. For involved mathematical manipulations try the `pygame.surfarray` module (It's quite quick, but requires NumPy.)

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. Remember to leave the Surface locked only while necessary.

Surface pixels are stored internally as a single number that has all the colors encoded into it. Use the `map_rgb()` and `unmap_rgb()` to convert between individual red, green, and blue values into a packed integer for that Surface.

Surfaces can also reference sections of other Surfaces. These are created with the `subsurface()` method. Any change to either Surface will effect the other.

Each Surface contains a clipping area. By default the clip area covers the entire Surface. If it is changed, all drawing operations will only effect the smaller area.

blit ()

draw one image onto another

blit(source, dest, area=None, special_flags=0) -> Rect

Draws a source Surface onto this Surface. The draw can be positioned with the dest argument. The dest argument can either be a pair of coordinates representing the position of the upper left corner of the blit or a Rect, where the upper left corner of the rectangle will be used as the position for the blit. The size of the destination rectangle does not effect the blit.

An optional area rectangle can be passed as well. This represents a smaller portion of the source Surface to draw.

New in version 1.8: *New in version 1.8:* Optional special_flags: BLEND_ADD, BLEND_SUB, BLEND_MULT, BLEND_MIN, BLEND_MAX.

New in version 1.8.1: *New in version 1.8.1:* Optional special_flags: BLEND_RGBA_ADD, BLEND_RGBA_SUB, BLEND_RGBA_MULT, BLEND_RGBA_MIN, BLEND_RGBA_MAX BLEND_RGB_ADD, BLEND_RGB_SUB, BLEND_RGB_MULT, BLEND_RGB_MIN, BLEND_RGB_MAX.

New in version 1.9.2: *New in version 1.9.2:* Optional special_flags: BLEND_PREMULTIPLIED

New in version 2.0.0: *New in version 2.0.0:* Optional special_flags: BLEND_ALPHA_SDL2 - Uses the SDL2 blitter for alpha blending, this gives different results than the default blitter, which is modelled after SDL1, due to different approximations used for the alpha blending formula. The SDL2 blitter also supports RLE on alpha blended surfaces which the pygame one does not.

The return rectangle is the area of the affected pixels, excluding any pixels outside the destination Surface, or outside the clipping area.

Pixel alphas will be ignored when blitting to an 8 bit Surface.

For a surface with colorkey or blanket alpha, a blit to self may give slightly different colors than a non self-blit.

blits ()

draw many images onto another

blits(blit_sequence=(source, dest), ...), doreturn=1) -> [Rect, ...] or None

blits((source, dest, area), ...) -> [Rect, ...]

blits((source, dest, area, special_flags), ...) -> [Rect, ...]

Draws many surfaces onto this Surface. It takes a sequence as input, with each of the elements corresponding to the ones of **blit()**. It needs at minimum a sequence of (source, dest).

Parameters:

- **blit_sequence** -- a sequence of surfaces and arguments to blit them, they correspond to the **blit()** arguments

- **doreturn** -- if True, return a list of rects of the areas changed, otherwise return None

Returns: a list of rects of the areas changed if doreturn is True, otherwise None

Return type: list or None

New in pygame 1.9.4.

convert ()

change the pixel format of an image

convert(Surface=None) -> Surface

convert(depth, flags=0) -> Surface

convert(masks, flags=0) -> Surface

Creates a new copy of the Surface with the pixel format changed. The new pixel format can be determined from another existing Surface. Otherwise depth, flags, and masks arguments can be used, similar to the **pygame.Surface()** call.

If no arguments are passed the new Surface will have the same pixel format as the display Surface. This is always the fastest format for blitting. It is a good idea to convert all Surfaces before they are blitted many times.

The converted Surface will have no pixel alphas. They will be stripped if the original had them. See **convert_alpha()** for preserving or creating per-pixel alphas.

The new copy will have the same class as the copied surface. This lets a Surface subclass inherit this method without the need to override, unless subclass specific instance attributes also need copying.

convert_alpha ()

change the pixel format of an image including per pixel alphas

convert_alpha(Surface) -> Surface

`convert_alpha()` -> 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 blitting 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.

As with `convert()` the returned surface has the same class as the converted surface.

`copy()`

create a new copy of a Surface

`copy()` -> Surface

Makes a duplicate copy of a Surface. The new surface will have the same pixel formats, color palettes, transparency settings, and class as the original. If a Surface subclass also needs to copy any instance specific attributes then it should override `copy()`.

`fill()`

fill Surface with a solid color

`fill(color, rect=None, special_flags=0)` -> Rect

Fill the Surface with a solid color. If no rect argument is given the entire Surface will be filled. The rect argument will limit the fill to a specific area. The fill will also be contained by the Surface clip area.

The color argument can be either a RGB sequence, a RGBA sequence or a mapped color index. If using RGBA, the Alpha (A part of RGBA) is ignored unless the surface uses per pixel alpha (Surface has the SRCALPHA flag).

New in version 1.8: *New in version 1.8:* Optional special_flags: BLEND_ADD, BLEND_SUB, BLEND_MULT, BLEND_MIN, BLEND_MAX.

New in version 1.8.1: *New in version 1.8.1:* Optional special_flags: BLEND_RGBA_ADD, BLEND_RGBA_SUB, BLEND_RGBA_MULT, BLEND_RGBA_MIN, BLEND_RGBA_MAX, BLEND_RGB_ADD, BLEND_RGB_SUB, BLEND_RGB_MULT, BLEND_RGB_MIN, BLEND_RGB_MAX.

This will return the affected Surface area.

`scroll()`

Shift the surface image in place

`scroll(dx=0, dy=0)` -> None

Move the image by dx pixels right and dy pixels down. dx and dy may be negative for left and up scrolls respectively. Areas of the surface that are not overwritten retain their original pixel values. Scrolling is contained by the Surface clip area. It is safe to have dx and dy values that exceed the surface size.

New in version 1.9: *New in version 1.9.*

`set_colorkey()`

Set the transparent colorkey

`set_colorkey(Color, flags=0)` -> None

`set_colorkey(None)` -> None

Set the current color key for the Surface. When blitting this Surface onto a destination, any pixels that have the same color as the colorkey will be transparent. The color can be an RGB color or a mapped color integer. If None is passed, the colorkey will be unset.

The colorkey will be ignored if the Surface is formatted to use per pixel alpha values. The colorkey can be mixed with the full Surface alpha value.

The optional flags argument can be set to `pygame.RLEACCEL` to provide better performance on non accelerated displays. An RLEACCEL Surface will be slower to modify, but quicker to blit as a source.

`get_colorkey()`

Get the current transparent colorkey

`get_colorkey()` -> RGB or None

Return the current colorkey value for the Surface. If the colorkey is not set then None is returned.

`set_alpha()`

set the alpha value for the full Surface image

`set_alpha(value, flags=0)` -> None

`set_alpha(None)` -> None

Set the current alpha value for the Surface. When blitting this Surface onto a destination, the pixels will be drawn slightly transparent. The alpha value is an integer from 0 to 255, 0 is fully transparent and 255 is fully opaque. If `None` is passed for the alpha value, then alpha blending will be disabled, including per-pixel alpha. This value is different than the per pixel Surface alpha. For a surface with per pixel alpha, blanket alpha is ignored and `None` is returned.

Changed in version 2.0: *Changed in version 2.0:* per-surface alpha can be combined with per-pixel alpha.

The optional flags argument can be set to `pygame.RLEACCEL` to provide better performance on non accelerated displays. An `RLEACCEL` Surface will be slower to modify, but quicker to blit as a source.

get_alpha ()

get the current Surface transparency value

`get_alpha()` -> `int_value`

Return the current alpha value for the Surface.

lock ()

lock the Surface memory for pixel access

`lock()` -> `None`

Lock the pixel data of a Surface for access. On accelerated Surfaces, the pixel data may be stored in volatile video memory or nonlinear compressed forms. When a Surface is locked the pixel memory becomes available to access by regular software. Code that reads or writes pixel values will need the Surface to be locked.

Surfaces should not remain locked for more than necessary. A locked Surface can often not be displayed or managed by pygame.

Not all Surfaces require locking. The `mustlock()` method can determine if it is actually required. There is no performance penalty for locking and unlocking a Surface that does not need it.

All pygame functions will automatically lock and unlock the Surface data as needed. If a section of code is going to make calls that will repeatedly lock and unlock the Surface many times, it can be helpful to wrap the block inside a lock and unlock pair.

It is safe to nest locking and unlocking calls. The surface will only be unlocked after the final lock is released.

unlock ()

unlock the Surface memory from pixel access

`unlock()` -> `None`

Unlock the Surface pixel data after it has been locked. The unlocked Surface can once again be drawn and managed by pygame. See the `lock()` documentation for more details.

All pygame functions will automatically lock and unlock the Surface data as needed. If a section of code is going to make calls that will repeatedly lock and unlock the Surface many times, it can be helpful to wrap the block inside a lock and unlock pair.

It is safe to nest locking and unlocking calls. The surface will only be unlocked after the final lock is released.

mustlock ()

test if the Surface requires locking

`mustlock()` -> `bool`

Returns `True` if the Surface is required to be locked to access pixel data. Usually pure software Surfaces do not require locking. This method is rarely needed, since it is safe and quickest to just lock all Surfaces as needed.

All pygame functions will automatically lock and unlock the Surface data as needed. If a section of code is going to make calls that will repeatedly lock and unlock the Surface many times, it can be helpful to wrap the block inside a lock and unlock pair.

get_locked ()

test if the Surface is current locked

`get_locked()` -> `bool`

Returns `True` when the Surface is locked. It doesn't matter how many times the Surface is locked.

get_locks ()

Gets the locks for the Surface

`get_locks()` -> `tuple`

Returns the currently existing locks for the Surface.

get_at ()

get the color value at a single pixel

get_at((x, y)) -> Color

Return a copy of the RGBA Color value at the given pixel. If the Surface has no per pixel alpha, then the alpha value will always be 255 (opaque). If the pixel position is outside the area of the Surface an `IndexError` exception will be raised.

Getting and setting pixels one at a time is generally too slow to be used in a game or realtime situation. It is better to use methods which operate on many pixels at a time like with the `blit`, `fill` and `draw` methods - or by using `pygame.surfarray`/`pygame.PixelArray`.

This function will temporarily lock and unlock the Surface as needed.

New in version 1.9: *New in version 1.9:* Returning a Color instead of tuple. Use `tuple(surf.get_at((x,y)))` if you want a tuple, and not a Color. This should only matter if you want to use the color as a key in a dict.

set_at ()

set the color value for a single pixel

set_at((x, y), Color) -> None

Set the RGBA or mapped integer color value for a single pixel. If the Surface does not have per pixel alphas, the alpha value is ignored. Setting pixels outside the Surface area or outside the Surface clipping will have no effect. Getting and setting pixels one at a time is generally too slow to be used in a game or realtime situation.

This function will temporarily lock and unlock the Surface as needed.

get_at_mapped ()

get the mapped color value at a single pixel

get_at_mapped((x, y)) -> Color

Return the integer value of the given pixel. If the pixel position is outside the area of the Surface an `IndexError` exception will be raised.

This method is intended for pygame unit testing. It unlikely has any use in an application.

This function will temporarily lock and unlock the Surface as needed.

New in version 1.9.2: *New in version 1.9.2.*

get_palette ()

get the color index palette for an 8-bit Surface

get_palette() -> [RGB, RGB, RGB, ...]

Return a list of up to 256 color elements that represent the indexed colors used in an 8-bit Surface. The returned list is a copy of the palette, and changes will have no effect on the Surface.

Returning a list of `Color`(with length 3) instances instead of tuples.

New in version 1.9: *New in version 1.9.*

get_palette_at ()

get the color for a single entry in a palette

get_palette_at(index) -> RGB

Returns the red, green, and blue color values for a single index in a Surface palette. The index should be a value from 0 to 255.

New in version 1.9: *New in version 1.9:* Returning `Color`(with length 3) instance instead of a tuple.

set_palette ()

set the color palette for an 8-bit Surface

set_palette([RGB, RGB, RGB, ...]) -> None

Set the full palette for an 8-bit Surface. This will replace the colors in the existing palette. A partial palette can be passed and only the first colors in the original palette will be changed.

This function has no effect on a Surface with more than 8-bits per pixel.

set_palette_at ()

set the color for a single index in an 8-bit Surface palette

set_palette_at(index, RGB) -> None

Set the palette value for a single entry in a Surface palette. The index should be a value from 0 to 255.

This function has no effect on a Surface with more than 8-bits per pixel.

map_rgb ()

convert a color into a mapped color value

map_rgb(Color) -> mapped_int

Convert an `RGBA` color into the mapped integer value for this Surface. The returned integer will contain no more bits than the bit depth of the Surface. Mapped color values are not often used inside pygame, but can be passed to most functions that require a Surface and a color.

See the Surface object documentation for more information about colors and pixel formats.

unmap_rgb ()

convert a mapped integer color value into a Color

unmap_rgb(mapped_int) -> Color

Convert an mapped integer color into the `RGB` color components for this Surface. Mapped color values are not often used inside pygame, but can be passed to most functions that require a Surface and a color.

See the Surface object documentation for more information about colors and pixel formats.

set_clip ()

set the current clipping area of the Surface

set_clip(rect) -> None

set_clip(None) -> None

Each Surface has an active clipping area. This is a rectangle that represents the only pixels on the Surface that can be modified. If `None` is passed for the rectangle the full Surface will be available for changes.

The clipping area is always restricted to the area of the Surface itself. If the clip rectangle is too large it will be shrunk to fit inside the Surface.

get_clip ()

get the current clipping area of the Surface

get_clip() -> Rect

Return a rectangle of the current clipping area. The Surface will always return a valid rectangle that will never be outside the bounds of the image. If the Surface has had `None` set for the clipping area, the Surface will return a rectangle with the full area of the Surface.

subsurface ()

create a new surface that references its parent

subsurface(Rect) -> Surface

Returns a new Surface that shares its pixels with its new parent. The new Surface is considered a child of the original. Modifications to either Surface pixels will effect each other. Surface information like clipping area and color keys are unique to each Surface.

The new Surface will inherit the palette, color key, and alpha settings from its parent.

It is possible to have any number of subsurfaces and subsubsurfaces on the parent. It is also possible to subsurface the display Surface if the display mode is not hardware accelerated.

See `get_offset()` and `get_parent()` to learn more about the state of a subsurface.

A subsurface will have the same class as the parent surface.

get_parent ()

find the parent of a subsurface

get_parent() -> Surface

Returns the parent Surface of a subsurface. If this is not a subsurface then `None` will be returned.

get_abs_parent ()

find the top level parent of a subsurface

get_abs_parent() -> Surface

Returns the parent Surface of a subsurface. If this is not a subsurface then this surface will be returned.

get_offset ()

find the position of a child subsurface inside a parent

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

Get the offset position of a child subsurface inside of a parent. If the Surface is not a subsurface this will return (0, 0).

`get_abs_offset()`

find the absolute position of a child subsurface inside its top level parent

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

Get the offset position of a child subsurface inside of its top level parent Surface. If the Surface is not a subsurface this will return (0, 0).

`get_size()`

get the dimensions of the Surface

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

Return the width and height of the Surface in pixels.

`get_width()`

get the width of the Surface

`get_width()` -> width

Return the width of the Surface in pixels.

`get_height()`

get the height of the Surface

`get_height()` -> height

Return the height of the Surface in pixels.

`get_rect()`

get the rectangular area of the Surface

`get_rect(**kwargs)` -> 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.

You can pass keyword argument values to this function. These named values will be applied to the attributes of the Rect before it is returned. An example would be `mysurf.get_rect(center=(100, 100))` to create a rectangle for the Surface centered at a given position.

`get_bitsize()`

get the bit depth of the Surface pixel format

`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()`

get the bytes used per Surface pixel

`get_bytesize()` -> int

Return the number of bytes used per pixel.

`get_flags()`

get the additional flags used for the Surface

`get_flags()` -> int

Returns a set of current Surface features. Each feature is a bit in the flags bitmask. Typical flags are HWSURFACE, RLEACCEL, SRCALPHA, and SRCCOLORKEY.

Here is a more complete list of flags. A full list can be found in `SDL_video.h`

SWSURFACE	0x00000000	# Surface is in system memory
HWSURFACE	0x00000001	# Surface is in video memory
ASYNCBIT	0x00000004	# Use asynchronous blits if possible

Available for `pygame.display.set_mode()`

ANYFORMAT	0x10000000	# Allow any video depth/pixel-format
HWPALLETTE	0x20000000	# Surface has exclusive palette
DOUBLEBUF	0x40000000	# Set up double-buffered video mode
FULLSCREEN	0x80000000	# Surface is a full screen display
OPENGL	0x00000002	# Create an OpenGL rendering context
OPENGLBLIT	0x0000000A	# OBSOLETE. Create an OpenGL rendering context and use it for blitting
RESIZABLE	0x00000010	# This video mode may be resized
NOFRAME	0x00000020	# No window caption or edge frame

Used internally (read-only)

HWACCEL	0x00000100	# Blit uses hardware acceleration
SRCCOLORKEY	0x00001000	# Blit uses a source color key
RLEACCELOK	0x00002000	# Private flag
RLEACCEL	0x00004000	# Surface is RLE encoded
SRCALPHA	0x00010000	# Blit uses source alpha blending
PREALLOC	0x01000000	# Surface uses preallocated memory

get_pitch()

get the number of bytes used per Surface row

get_pitch() -> int

Return the number of bytes separating each row in the Surface. Surfaces in video memory are not always linearly packed. Subsurfaces will also have a larger pitch than their real width.

This value is not needed for normal pygame usage.

get_masks()

the bitmasks needed to convert between a color and a mapped integer

get_masks() -> (R, G, B, A)

Returns the bitmasks used to isolate each color in a mapped integer.

This value is not needed for normal pygame usage.

set_masks()

set the bitmasks needed to convert between a color and a mapped integer

set_masks((r,g,b,a)) -> None

This is not needed for normal pygame usage.

Note

In SDL2, the masks are read-only and accordingly this method will raise an AttributeError if called.

New in version 1.8.1: *New in version 1.8.1.*

get_shifts()

the bit shifts needed to convert between a color and a mapped integer

get_shifts() -> (R, G, B, A)

Returns the pixel shifts need to convert between each color and a mapped integer.

This value is not needed for normal pygame usage.

set_shifts()

sets the bit shifts needed to convert between a color and a mapped integer

set_shifts((r,g,b,a)) -> None

This is not needed for normal pygame usage.

Note

In SDL2, the shifts are read-only and accordingly this method will raise an AttributeError if called.

New in version 1.8.1: *New in version 1.8.1.*

`get_losses ()`

the significant bits used to convert between a color and a mapped integer

`get_losses()` -> (R, G, B, A)

Return the least significant number of bits stripped from each color in a mapped integer.

This value is not needed for normal pygame usage.

`get_bounding_rect ()`

find the smallest rect containing data

`get_bounding_rect(min_alpha = 1)` -> Rect

Returns the smallest rectangular region that contains all the pixels in the surface that have an alpha value greater than or equal to the minimum alpha value.

This function will temporarily lock and unlock the Surface as needed.

New in version 1.8: *New in version 1.8.*

`get_view ()`

return a buffer view of the Surface's pixels.

`get_view(<kind>='2')` -> BufferProxy

Return an object which exports a surface's internal pixel buffer as a C level array struct, Python level array interface or a C level buffer interface. The pixel buffer is writeable. The new buffer protocol is supported for Python 2.6 and up in CPython. The old buffer protocol is also supported for Python 2.x. The old buffer data is in one segment for kind '0', multi-segment for other buffer view kinds.

The kind argument is the length 1 string '0', '1', '2', '3', 'r', 'g', 'b', or 'a'. The letters are case insensitive; 'A' will work as well. The argument can be either a Unicode or byte (char) string. The default is '2'.

'0' returns a contiguous unstructured bytes view. No surface shape information is given. A `ValueError` is raised if the surface's pixels are discontinuous.

'1' returns a (surface-width * surface-height) array of continuous pixels. A `ValueError` is raised if the surface pixels are discontinuous.

'2' returns a (surface-width, surface-height) array of raw pixels. The pixels are surface-bytesize-d unsigned integers. The pixel format is surface specific. The 3 byte unsigned integers of 24 bit surfaces are unlikely accepted by anything other than other pygame functions.

'3' returns a (surface-width, surface-height, 3) array of RGB color components. Each of the red, green, and blue components are unsigned bytes. Only 24-bit and 32-bit surfaces are supported. The color components must be in either RGB or BGR order within the pixel.

'r' for red, 'g' for green, 'b' for blue, and 'a' for alpha return a (surface-width, surface-height) view of a single color component within a surface: a color plane. Color components are unsigned bytes. Both 24-bit and 32-bit surfaces support 'r', 'g', and 'b'. Only 32-bit surfaces with `SRCALPHA` support 'a'.

The surface is locked only when an exposed interface is accessed. For new buffer interface accesses, the surface is unlocked once the last buffer view is released. For array interface and old buffer interface accesses, the surface remains locked until the BufferProxy object is released.

New in version 1.9.2: *New in version 1.9.2.*

`get_buffer ()`

acquires a buffer object for the pixels of the Surface.

`get_buffer()` -> BufferProxy

Return a buffer object for the pixels of the Surface. The buffer can be used for direct pixel access and manipulation. Surface pixel data is represented as an unstructured block of memory, with a start address and length in bytes. The data need not be contiguous. Any gaps are included in the length, but otherwise ignored.

This method implicitly locks the Surface. The lock will be released when the returned `pygame.BufferProxy` object is garbage collected.

New in version 1.8: *New in version 1.8.*

`_pixels_address`

pixel buffer address

`_pixels_address` -> int

The starting address of the surface's raw pixel bytes.

New in version 1.9.2: *New in version 1.9.2.*

pygame.surfarray

pygame module for accessing surface pixel data using array interfaces

Functions to convert pixel data between pygame Surfaces and arrays. This module will only be functional when pygame can use the external NumPy package.

Every pixel is stored as a single integer value to represent the red, green, and blue colors. The 8-bit images use a value that looks into a colormap. Pixels with higher depth use a bit packing process to place three or four values into a single number.

The arrays are indexed by the x axis first, followed by the y axis. Arrays that treat the pixels as a single integer are referred to as 2D arrays. This module can also separate the red, green, and blue color values into separate indices. These types of arrays are referred to as 3D arrays, and the last index is 0 for red, 1 for green, and 2 for blue.

`pygame.surfarray.array2d()`

Copy pixels into a 2d array

`array2d(Surface) -> array`

Copy the **mapped** (raw) pixels from a Surface into a 2D array. The bit depth of the surface will control the size of the integer values, and will work for any type of pixel format.

This function will temporarily lock the Surface as pixels are copied (see the `pygame.Surface.lock()` - lock the Surface memory for pixel access method).

`pygame.surfarray.pixels2d()`

Reference pixels into a 2d array

`pixels2d(Surface) -> array`

Create a new 2D array that directly references the pixel values in a Surface. Any changes to the array will affect the pixels in the Surface. This is a fast operation since no data is copied.

Pixels from a 24-bit Surface cannot be referenced, but all other Surface bit depths can.

The Surface this references will remain locked for the lifetime of the array (see the `pygame.Surface.lock()` - lock the Surface memory for pixel access method).

`pygame.surfarray.array3d()`

Copy pixels into a 3d array

`array3d(Surface) -> array`

Copy the pixels from a Surface into a 3D array. The bit depth of the surface will control the size of the integer values, and will work for any type of pixel format.

This function will temporarily lock the Surface as pixels are copied (see the `pygame.Surface.lock()` - lock the Surface memory for pixel access method).

`pygame.surfarray.pixels3d()`

Reference pixels into a 3d array

`pixels3d(Surface) -> array`

Create a new 3D array that directly references the pixel values in a Surface. Any changes to the array will affect the pixels in the Surface. This is a fast operation since no data is copied.

This will only work on Surfaces that have 24-bit or 32-bit formats. Lower pixel formats cannot be referenced.

The Surface this references will remain locked for the lifetime of the array (see the `pygame.Surface.lock()` - lock the Surface memory for pixel access method).

`pygame.surfarray.array_alpha()`

Copy pixel alphas into a 2d array

`array_alpha(Surface) -> array`

Copy the pixel alpha values (degree of transparency) from a Surface into a 2D array. This will work for any type of Surface format. Surfaces without a pixel alpha will return an array with all opaque values.

This function will temporarily lock the Surface as pixels are copied (see the `pygame.Surface.lock()` - lock the Surface memory for pixel access method).

`pygame.surfarray.pixels_alpha()`

Reference pixel alphas into a 2d array

`pixels_alpha(Surface) -> array`

Create a new 2D array that directly references the alpha values (degree of transparency) in a Surface. Any changes to the array will affect the pixels in the Surface. This is a fast operation since no data is copied.

This can only work on 32-bit Surfaces with a per-pixel alpha value.

The Surface this array references will remain locked for the lifetime of the array.

`pygame.surfarray.pixels_red()`

Reference pixel red into a 2d array.

`pixels_red(Surface) -> array`

Create a new 2D array that directly references the red values in a Surface. Any changes to the array will affect the pixels in the Surface. This is a fast operation since no data is copied.

This can only work on 24-bit or 32-bit Surfaces.

The Surface this array references will remain locked for the lifetime of the array.

`pygame.surfarray.pixels_green()`

Reference pixel green into a 2d array.

`pixels_green(Surface) -> array`

Create a new 2D array that directly references the green values in a Surface. Any changes to the array will affect the pixels in the Surface. This is a fast operation since no data is copied.

This can only work on 24-bit or 32-bit Surfaces.

The Surface this array references will remain locked for the lifetime of the array.

`pygame.surfarray.pixels_blue()`

Reference pixel blue into a 2d array.

`pixels_blue(Surface) -> array`

Create a new 2D array that directly references the blue values in a Surface. Any changes to the array will affect the pixels in the Surface. This is a fast operation since no data is copied.

This can only work on 24-bit or 32-bit Surfaces.

The Surface this array references will remain locked for the lifetime of the array.

`pygame.surfarray.array_colorkey()`

Copy the colorkey values into a 2d array

`array_colorkey(Surface) -> array`

Create a new array with the colorkey transparency value from each pixel. If the pixel matches the colorkey it will be fully transparent; otherwise it will be fully opaque.

This will work on any type of Surface format. If the image has no colorkey a solid opaque array will be returned.

This function will temporarily lock the Surface as pixels are copied.

`pygame.surfarray.make_surface()`

Copy an array to a new surface

`make_surface(array) -> Surface`

Create a new Surface that best resembles the data and format on the array. The array can be 2D or 3D with any sized integer values. Function `make_surface` uses the array struct interface to acquire array properties, so is not limited to just NumPy arrays. See `pygame.pixelcopy`.

New in pygame 1.9.2: array struct interface support.

`pygame.surfarray.blit_array()`

Blit directly from a array values

`blit_array(Surface, array) -> None`

Directly copy values from an array into a Surface. This is faster than converting the array into a Surface and blitting. The array must be the same dimensions as the Surface and will completely replace all pixel values. Only integer, ASCII character and record arrays are accepted.

This function will temporarily lock the Surface as the new values are copied.

`pygame.surfarray.map_array()`

Map a 3d array into a 2d array

`map_array(Surface, array3d) -> array2d`

Convert a 3D array into a 2D array. This will use the given Surface format to control the conversion. Palette surface formats are supported for NumPy arrays.

`pygame.surfarray.use_arraytype()`

Sets the array system to be used for surface arrays

use_arraytype (arraytype) -> None

DEPRECATED: Uses the requested array type for the module functions. The only supported arraytype is 'numpy'. Other values will raise ValueError.

pygame.surfarray.get_arraytype ()

Gets the currently active array type.

get_arraytype () -> str

DEPRECATED: Returns the currently active array type. This will be a value of the get_arraytypes () tuple and indicates which type of array module is used for the array creation.

New in version 1.8: *New in version 1.8.*

pygame.surfarray.get_arraytypes ()

Gets the array system types currently supported.

get_arraytypes () -> tuple

DEPRECATED: Checks, which array systems are available and returns them as a tuple of strings. The values of the tuple can be used directly in the `pygame.surfarray.use_arraytype ()` () method. If no supported array system could be found, None will be returned.

New in version 1.8: *New in version 1.8.*

pygame.tests

Pygame unit test suite package

A quick way to run the test suite package from the command line is to import the go submodule with the Python -m option:

```
python -m pygame.tests [<test options>]
```

Command line option --help displays a usage message. Available options correspond to the `pygame.tests.run ()` arguments.

The xxxx_test submodules of the tests package are unit test suites for individual parts of pygame. Each can also be run as a main program. This is useful if the test, such as cdrom_test, is interactive.

For pygame development the test suite can be run from a pygame distribution root directory. Program run_tests.py is provided for convenience, though test/go.py can be run directly.

Module level tags control which modules are included in a unit test run. Tags are assigned to a unit test module with a corresponding <name>_tags.py module. The tags module has the global __tags__, a list of tag names. For example, cdrom_test.py has a tag file cdrom_tags.py containing a tags list that has the 'interactive' string. The 'interactive' tag indicates cdrom_test.py expects user input. It is excluded from a run_tests.py or pygame.tests.go run. Two other tags that are excluded are 'ignore' and 'subprocess_ignore'. These two tags indicate unit tests that will not run on a particular platform, or for which no corresponding pygame module is available. The test runner will list each excluded module along with the tag responsible.

pygame.tests.run ()

Run the pygame unit test suite

run(*args, **kwds) -> tuple

Positional arguments (optional):

The names of tests to include. If omitted then all tests are run. Test names need not include the trailing '_test'.

Keyword arguments:

```
incomplete - fail incomplete tests (default False)
nosubprocess - run all test suites in the current process
                (default False, use separate subprocesses)
dump - dump failures/errors as dict ready to eval (default False)
file - if provided, the name of a file into which to dump failures/errors
timings - if provided, the number of times to run each individual test to
            get an average run time (default is run each test once)
exclude - A list of TAG names to exclude from the run
show_output - show silenced stderr/stdout on errors (default False)
```

```

all - dump all results, not just errors (default False)
randomize - randomize order of tests (default False)
seed - if provided, a seed randomizer integer
multi_thread - if provided, the number of THREADS in which to run
                subprocessed tests
time_out - if subprocess is True then the time limit in seconds before
            killing a test (default 30)
fake - if provided, the name of the fake tests package in the
        run_tests__tests subpackage to run instead of the normal
        pygame tests
python - the path to a python executable to run subprocessed tests
         (default sys.executable)

```

Return value:

```

A tuple of total number of tests run, dictionary of error information.
The dictionary is empty if no errors were recorded.

```

By default individual test modules are run in separate subprocesses. This recreates normal pygame usage where `pygame.init()` and `pygame.quit()` are called only once per program execution, and avoids unfortunate interactions between test modules. Also, a time limit is placed on test execution, so frozen tests are killed when their time allotment expired. Use the single process option if threading is not working properly or if tests are taking too long. It is not guaranteed that all tests will pass in single process mode.

Tests are run in a randomized order if the `randomize` argument is `True` or a `seed` argument is provided. If no seed integer is provided then the system time is used.

Individual test modules may have a `__tags__` attribute, a list of tag strings used to selectively omit modules from a run. By default only 'interactive' modules such as `cdrom_test` are ignored. An interactive module must be run from the console as a Python program.

This function can only be called once per Python session. It is not reentrant.

pygame.time

pygame module for monitoring time

Times in pygame are represented in milliseconds (1/1000 seconds). Most platforms have a limited time resolution of around 10 milliseconds. This resolution, in milliseconds, is given in the `TIMER_RESOLUTION` constant.

`pygame.time.get_ticks()`

get the time in milliseconds

`get_ticks()` -> milliseconds

Return the number of milliseconds since `pygame.init()` was called. Before pygame is initialized this will always be 0.

`pygame.time.wait()`

pause the program for an amount of time

`wait(milliseconds)` -> time

Will pause for a given number of milliseconds. This function sleeps the process to share the processor with other programs. A program that waits for even a few milliseconds will consume very little processor time. It is slightly less accurate than the `pygame.time.delay()` function.

This returns the actual number of milliseconds used.

`pygame.time.delay()`

pause the program for an amount of time

`delay(milliseconds)` -> time

Will pause for a given number of milliseconds. This function will use the processor (rather than sleeping) in order to make the delay more accurate than `pygame.time.wait()`.

This returns the actual number of milliseconds used.

`pygame.time.set_timer()`

repeatedly create an event on the event queue

`set_timer(eventid, milliseconds)` -> None

`set_timer(eventid, milliseconds, once)` -> None

Set an event type to appear on the event queue every given number of milliseconds. The first event will not appear until the amount of time has passed.

Every event type can have a separate timer attached to it. It is best to use the value between `pygame.USEREVENT` and `pygame.NUMEVENTS`.

To disable the timer for an event, set the milliseconds argument to 0.

If the once argument is True, then only send the timer once.

New in version 2.0.0.dev3: *New in version 2.0.0.dev3*: once argument added.

`pygame.time.Clock`

create an object to help track time

`Clock()` -> Clock

Creates a new Clock object that can be used to track an amount of time. The clock also provides several functions to help control a game's framerate.

`tick()`

update the clock

`tick(framerate=0)` -> milliseconds

This method should be called once per frame. It will compute how many milliseconds have passed since the previous call.

If you pass the optional framerate argument the function will delay to keep the game running slower than the given ticks per second. This can be used to help limit the runtime speed of a game. By calling `Clock.tick(40)` once per frame, the program will never run at more than 40 frames per second.

Note that this function uses `SDL_Delay` function which is not accurate on every platform, but does not use much CPU. Use `tick_busy_loop` if you want an accurate timer, and don't mind chewing CPU.

`tick_busy_loop()`

update the clock

`tick_busy_loop(framerate=0)` -> milliseconds

This method should be called once per frame. It will compute how many milliseconds have passed since the previous call.

If you pass the optional framerate argument the function will delay to keep the game running slower than the given ticks per second. This can be used to help limit the runtime speed of a game. By calling `Clock.tick_busy_loop(40)` once per frame, the program will never run at more than 40 frames per second.

Note that this function uses `pygame.time.delay()`, which uses lots of CPU in a busy loop to make sure that timing is more accurate.

New in version 1.8: *New in version 1.8*.

`get_time()`

time used in the previous tick

`get_time()` -> milliseconds

The number of milliseconds that passed between the previous two calls to `Clock.tick()`.

`get_rawtime()`

actual time used in the previous tick

`get_rawtime()` -> milliseconds

Similar to `Clock.get_time()`, but does not include any time used while `Clock.tick()` was delaying to limit the framerate.

`get_fps()`

compute the clock framerate

`get_fps()` -> float

Compute your game's framerate (in frames per second). It is computed by averaging the last ten calls to `Clock.tick()`.

pygame.transform

pygame module to work with touch input

New in version 2: *New in version 2*: This module requires SDL2.

`pygame._sdl2.touch.get_num_devices()`

get the number of touch devices

`get_num_devices()` -> int

Return the number of available touch devices.

`pygame._sdl2.touch.get_device()`

get the a touch device id for a given index

`get_device(index)` -> touchid

Parameters: `index (int)` -- This number is at least 0 and less than the **number of devices**.

Return an integer id associated with the given index.

`pygame._sdl2.touch.get_num_fingers()`

the number of active fingers for a given touch device

`get_num_fingers(touchid)` -> int

Return the number of fingers active for the touch device whose id is *touchid*.

`pygame._sdl2.touch.get_finger()`

get information about an active finger

`get_finger(touchid, index)` -> int

Parameters:

- **touchid (int)** -- The touch device id.

- **index (int)** -- The index of the finger to return information about, between 0 and the **number of active fingers**.

Return a dict for the finger *index* active on *touchid*. The dict contains these keys:

<code>id</code>	the id of the finger (an integer).
<code>x</code>	the normalized x position of the finger, between 0 and 1.
<code>y</code>	the normalized y position of the finger, between 0 and 1.
<code>pressure</code>	the amount of pressure applied by the finger, between 0 and 1.

pygame.transform

pygame module to transform surfaces

A Surface transform is an operation that moves or resizes the pixels. All these functions take a Surface to operate on and return a new Surface with the results.

Some of the transforms are considered destructive. These means every time they are performed they lose pixel data. Common examples of this are resizing and rotating. For this reason, it is better to re-transform the original surface than to keep transforming an image multiple times. (For example, suppose you are animating a bouncing spring which expands and contracts. If you applied the size changes incrementally to the previous images, you would lose detail. Instead, always begin with the original image and scale to the desired size.)

`pygame.transform.flip()`

flip vertically and horizontally

`flip(Surface, xbool, ybool)` -> Surface

This can flip a Surface either vertically, horizontally, or both. Flipping a Surface is non-destructive and returns a new Surface with the same dimensions.

`pygame.transform.scale()`

resize to new resolution

`scale(Surface, (width, height), DestSurface = None)` -> Surface

Resizes the Surface to a new resolution. This is a fast scale operation that does not sample the results.

An optional destination surface can be used, rather than have it create a new one. This is quicker if you want to repeatedly scale something. However the destination must be the same size as the (width, height) passed in. Also the destination surface must be the same format.

`pygame.transform.rotate()`

rotate an image

`rotate(Surface, angle) -> Surface`

Unfiltered counterclockwise rotation. The angle argument represents degrees and can be any floating point value. Negative angle amounts will rotate clockwise.

Unless rotating by 90 degree increments, the image will be padded larger to hold the new size. If the image has pixel alphas, the padded area will be transparent. Otherwise pygame will pick a color that matches the Surface colorkey or the topleft pixel value.

`pygame.transform.rotozoom()`

filtered scale and rotation

`rotozoom(Surface, angle, scale) -> Surface`

This is a combined scale and rotation transform. The resulting Surface will be a filtered 32-bit Surface. The scale argument is a floating point value that will be multiplied by the current resolution. The angle argument is a floating point value that represents the counterclockwise degrees to rotate. A negative rotation angle will rotate clockwise.

`pygame.transform.scale2x()`

specialized image doubler

`scale2x(Surface, DestSurface = None) -> 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.

An optional destination surface can be used, rather than have it create a new one. This is quicker if you want to repeatedly scale something. However the destination must be twice the size of the source surface passed in. Also the destination surface must be the same format.

`pygame.transform.smoothscale()`

scale a surface to an arbitrary size smoothly

`smoothscale(Surface, (width, height), DestSurface = None) -> Surface`

Uses one of two different algorithms for scaling each dimension of the input surface as required. For shrinkage, the output pixels are area averages of the colors they cover. For expansion, a bilinear filter is used. For the x86-64 and i686 architectures, optimized MMX routines are included and will run much faster than other machine types. The size is a 2 number sequence for (width, height). This function only works for 24-bit or 32-bit surfaces. An exception will be thrown if the input surface bit depth is less than 24.

New in version 1.8: *New in version 1.8.*

`pygame.transform.get_smoothscale_backend()`

return smoothscale filter version in use: 'GENERIC', 'MMX', or 'SSE'

`get_smoothscale_backend() -> String`

Shows whether or not smoothscale is using MMX or SSE acceleration. If no acceleration is available then "GENERIC" is returned. For a x86 processor the level of acceleration to use is determined at runtime.

This function is provided for pygame testing and debugging.

`pygame.transform.set_smoothscale_backend()`

set smoothscale filter version to one of: 'GENERIC', 'MMX', or 'SSE'

`set_smoothscale_backend(type) -> None`

Sets smoothscale acceleration. Takes a string argument. A value of 'GENERIC' turns off acceleration. 'MMX' uses MMX instructions only. 'SSE' allows SSE extensions as well. A value error is raised if type is not recognized or not supported by the current processor.

This function is provided for pygame testing and debugging. If smoothscale causes an invalid instruction error then it is a pygame/SDL bug that should be reported. Use this function as a temporary fix only.

`pygame.transform.chop()`

gets a copy of an image with an interior area removed

`chop(Surface, rect) -> Surface`

Extracts a portion of an image. All vertical and horizontal pixels surrounding the given rectangle area are removed. The corner areas (diagonal to the rect) are then brought together. (The original image is not altered by this operation.)

NOTE: If you want a "crop" that returns the part of an image within a rect, you can blit with a rect to a new surface or copy a subsurface.

`pygame.transform.laplacian ()`

find edges in a surface

`laplacian(Surface, DestSurface = None) -> Surface`

Finds the edges in a surface using the laplacian algorithm.

New in version 1.8: *New in version 1.8.*

`pygame.transform.average_surfaces ()`

find the average surface from many surfaces.

`average_surfaces(Surfaces, DestSurface = None, palette_colors = 1) -> Surface`

Takes a sequence of surfaces and returns a surface with average colors from each of the surfaces.

`palette_colors` - if true we average the colors in palette, otherwise we average the pixel values. This is useful if the surface is actually greyscale colors, and not palette colors.

Note, this function currently does not handle palette using surfaces correctly.

New in version 1.8: *New in version 1.8.*

New in version 1.9: *New in version 1.9:* `palette_colors` argument

`pygame.transform.average_color ()`

finds the average color of a surface

`average_color(Surface, Rect = None) -> Color`

Finds the average color of a Surface or a region of a surface specified by a Rect, and returns it as a Color.

`pygame.transform.threshold ()`

finds which, and how many pixels in a surface are within a threshold of a 'search_color' or a 'search_surf'.

`threshold(dest_surf, surf, search_color, threshold=(0,0,0,0), set_color=(0,0,0,0), set_behavior=1, search_surf=None, inverse_set=False) -> num_threshold_pixels`

This versatile function can be used for find colors in a 'surf' close to a 'search_color' or close to colors in a separate 'search_surf'.

It can also be used to transfer pixels into a 'dest_surf' that match or don't match.

By default it sets pixels in the 'dest_surf' where all of the pixels NOT within the threshold are changed to `set_color`.

If `inverse_set` is optionally set to True, the pixels that ARE within the threshold are changed to `set_color`.

If the optional 'search_surf' surface is given, it is used to threshold against rather than the specified 'set_color'. That is, it will find each pixel in the 'surf' that is within the 'threshold' of the pixel at the same coordinates of the 'search_surf'.

Parameters:

- **dest_surf** (*pygame.Surface* or None) -- Surface we are changing. See 'set_behavior'. Should be None if counting (set_behavior is 0).
- **surf** (*pygame.Surface*) -- Surface we are looking at.
- **search_color** (*pygame.Color*) -- Color we are searching for.
- **threshold** (*pygame.Color*) -- Within this distance from search_color (or search_surf). You can use a threshold of (r,g,b,a) where the r,g,b can have different thresholds. So you could use an r threshold of 40 and a blue threshold of 2 if you like.
- **set_color** (*pygame.Color* or None) -- Color we set in dest_surf.
- **set_behavior** (*int*) -- set_behavior=1 (default). Pixels in dest_surface will be changed to 'set_color'. set_behavior=0 we do not change 'dest_surf', just count. Make dest_surf=None. set_behavior=2 pixels set in 'dest_surf' will be from 'surf'.
- **search_surf** (*pygame.Surface* or None) -- search_surf=None (default). Search against 'search_color' instead. search_surf=Surface. Look at the color in 'search_surf' rather than using 'search_color'.
- **inverse_set** (*bool*) -- False, default. Pixels outside of threshold are changed. True, Pixels within threshold are changed.

Return type: int

Returns: The number of pixels that are within the 'threshold' in 'surf' compared to either 'search_color' or *search_surf*.

Examples:

See the threshold tests for a full of examples:
https://github.com/pygame/pygame/blob/master/test/transform_test.py

```
def test_threshold_dest_surf_not_change(self):
    """ the pixels within the threshold.

    All pixels not within threshold are changed to set_color.
    So there should be none changed in this test.
    """
    (w, h) = size = (32, 32)
    threshold = (20, 20, 20, 20)
    original_color = (25, 25, 25, 25)
    original_dest_color = (65, 65, 65, 55)
    threshold_color = (10, 10, 10, 10)
    set_color = (255, 10, 10, 10)

    surf = pygame.Surface(size, pygame.SRCALPHA, 32)
    dest_surf = pygame.Surface(size, pygame.SRCALPHA, 32)
    search_surf = pygame.Surface(size, pygame.SRCALPHA, 32)

    surf.fill(original_color)
    search_surf.fill(threshold_color)
    dest_surf.fill(original_dest_color)

    # set_behavior=1, set dest_surface from set_color.
    # all within threshold of third_surface, so no color is set.

    THRESHOLD_BEHAVIOR_FROM_SEARCH_COLOR = 1
    pixels_within_threshold = pygame.transform.threshold(
        dest_surf=dest_surf,
        surf=surf,
        search_color=None,
        threshold=threshold,
        set_color=set_color,
        set_behavior=THRESHOLD_BEHAVIOR_FROM_SEARCH_COLOR,
        search_surf=search_surf,
    )

    # # Return, of pixels within threshold is correct
    self.assertEqual(w * h, pixels_within_threshold)

    # # Size of dest surface is correct
    dest_rect = dest_surf.get_rect()
    dest_size = dest_rect.size
    self.assertEqual(size, dest_size)

    # The color is not the change_color specified for every pixel As all
    # pixels are within threshold

    for pt in test_utils.rect_area_pts(dest_rect):
        self.assertNotEqual(dest_surf.get_at(pt), set_color)
        self.assertEqual(dest_surf.get_at(pt), original_dest_color)
```

New in version 1.8: *New in version 1.8.*

Changed in version 1.9.4: *Changed in version 1.9.4:* Fixed a lot of bugs and added keyword arguments. Test your code.

Camera Module Introduction

Author: by Nirav Patel

Contact: nrp@eclecti.cc

Pygame 1.9 comes with support for interfacing cameras, allowing you to capture still images, watch live streams, and do some simple computer vision. This tutorial will cover all of those use cases, providing code samples you can base your app or game on. You can refer to the [reference documentation](#) for the full API.

Note

As of Pygame 1.9, the camera module offers native support for cameras that use v4l2 on Linux. There is support for other platforms via Videocapture or OpenCV, but this guide will focus on the native module. Most of the code will be valid for other platforms, but certain things like controls will not work. The module is also marked as **EXPERIMENTAL**, meaning the API could change in subsequent versions.

Import and Init

```
import pygame
import pygame.camera
from pygame.locals import *

pygame.init()
pygame.camera.init()
```

As the camera module is optional, it needs to be imported and initialized manually as shown above.

Capturing a Single Image

Now we will go over the simplest case of opening a camera and capturing a frame as a surface. In the below example, we assume that there is a camera at `/dev/video0` on the computer, and initialize it with a size of 640 by 480. The surface called `image` is whatever the camera was seeing when `get_image()` was called.

```
cam = pygame.camera.Camera("/dev/video0", (640, 480))
cam.start()
image = cam.get_image()
```

Listing Connected Cameras

You may be wondering, what if we don't know the exact path of the camera? We can ask the module to provide a list of cameras attached to the computer and initialize the first camera in the list.

```
camlist = pygame.camera.list_cameras()
if camlist:
    cam = pygame.camera.Camera(camlist[0], (640, 480))
```

Using Camera Controls

Most cameras support controls like flipping the image and changing brightness. `set_controls()` and `get_controls()` can be used at any point after using `start()`.

```
cam.set_controls(hflip = True, vflip = False)
print camera.get_controls()
```

Capturing a Live Stream

The rest of this tutorial will be based around capturing a live stream of images. For this, we will be using the class below. As described, it will simply blit a constant stream of camera frames to the screen, effectively showing live

video. It is basically what you would expect, looping `get_image()`, blitting to the display surface, and flipping it. For performance reasons, we will be supplying the camera with the same surface to use each time.

```
class Capture(object):
    def __init__(self):
        self.size = (640,480)
        # create a display surface. standard pygame stuff
        self.display = pygame.display.set_mode(self.size, 0)

        # this is the same as what we saw before
        self.clist = pygame.camera.list_cameras()
        if not self.clist:
            raise ValueError("Sorry, no cameras detected.")
        self.cam = pygame.camera.Camera(self.clist[0], self.size)
        self.cam.start()

        # create a surface to capture to. for performance purposes
        # bit depth is the same as that of the display surface.
        self.snapshot = pygame.surface.Surface(self.size, 0, self.display)

    def get_and_flip(self):
        # if you don't want to tie the framerate to the camera, you can check
        # if the camera has an image ready. note that while this works
        # on most cameras, some will never return true.
        if self.cam.query_image():
            self.snapshot = self.cam.get_image(self.snapshot)

        # blit it to the display surface. simple!
        self.display.blit(self.snapshot, (0,0))
        pygame.display.flip()

    def main(self):
        going = True
        while going:
            events = pygame.event.get()
            for e in events:
                if e.type == QUIT or (e.type == KEYDOWN and e.key == K_ESCAPE):
                    # close the camera safely
                    self.cam.stop()
                    going = False

            self.get_and_flip()
```

Since `get_image()` is a blocking call that could take quite a bit of time on a slow camera, this example uses `query_image()` to see if the camera is ready. This allows you to separate the framerate of your game from that of your camera. It is also possible to have the camera capturing images in a separate thread, for approximately the same performance gain, if you find that your camera does not support the `query_image()` function correctly.

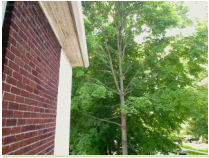
Basic Computer Vision

By using the camera, transform, and mask modules, pygame can do some basic computer vision.

Colorspaces

When initializing a camera, `colorspace` is an optional parameter, with 'RGB', 'YUV', and 'HSV' as the possible choices. YUV and HSV are both generally more useful for computer vision than RGB, and allow you to more easily threshold by color, something we will look at later in the tutorial.

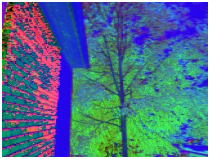
```
self.cam = pygame.camera.Camera(self.clist[0], self.size, "RGB")
```



```
self.cam = pygame.camera.Camera(self.clist[0], self.size, "YUV")
```



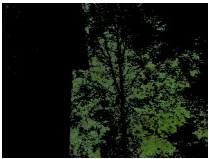
```
self.cam = pygame.camera.Camera(self.clist[0], self.size, "HSV")
```



Thresholding

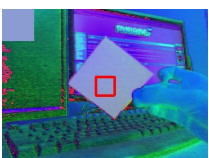
Using the `threshold()` function from the transform module, one can do simple green screen like effects, or isolate specifically colored objects in a scene. In the below example, we threshold out just the green tree and make the rest of the image black. Check the reference documentation for details on the **threshold function**.

```
self.thresholded = pygame.surface.Surface(self.size, 0, self.display)
self.snapshot = self.cam.get_image(self.snapshot)
pygame.transform.threshold(self.thresholded, self.snapshot, (0, 255, 0), (90, 170, 170), (0, 0, 0), 2)
```

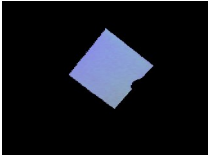


Of course, this is only useful if you already know the exact color of the object you are looking for. To get around this and make thresholding usable in the real world, we need to add a calibration stage where we identify the color of an object and use it to threshold against. We will be using the `average_color()` function of the transform module to do this. Below is an example calibration function that you could loop until an event like a key press, and an image of what it would look like. The color inside the box will be the one that is used for the threshold. Note that we are using the HSV colorspace in the below images.

```
def calibrate(self):
    # capture the image
    self.snapshot = self.cam.get_image(self.snapshot)
    # blit it to the display surface
    self.display.blit(self.snapshot, (0, 0))
    # make a rect in the middle of the screen
    crect = pygame.draw.rect(self.display, (255, 0, 0), (145, 105, 30, 30), 4)
    # get the average color of the area inside the rect
    self.ccolor = pygame.transform.average_color(self.snapshot, crect)
    # fill the upper left corner with that color
    self.display.fill(self.ccolor, (0, 0, 50, 50))
    pygame.display.flip()
```

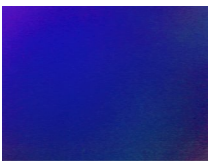


```
pygame.transform.threshold(self.thresholded, self.snapshot, self.ccolor, (30, 30, 30), (0, 0, 0), 2)
```



You can use the same idea to do a simple green screen/blue screen, by first getting a background image and then thresholding against it. The below example just has the camera pointed at a blank white wall in HSV colorspace.

```
def calibrate(self):
    # capture a bunch of background images
    bg = []
    for i in range(0,5):
        bg.append(self.cam.get_image(self.background))
    # average them down to one to get rid of some noise
    pygame.transform.average_surfaces(bg,self.background)
    # blit it to the display surface
    self.display.blit(self.background, (0,0))
    pygame.display.flip()
```



```
pygame.transform.threshold(self.thresholded,self.snapshot,(0,255,0),(30,30,30),(0,0,0),1,self
```



Using the Mask Module

The stuff above is great if you just want to display images, but with the **mask module**, you can also use a camera as an input device for a game. For example, going back to the example of thresholding out a specific object, we can find the position of that object and use it to control an on screen object.

```
def get_and_flip(self):
    self.snapshot = self.cam.get_image(self.snapshot)
    # threshold against the color we got before
    mask = pygame.mask.from_threshold(self.snapshot, self.ccolor, (30, 30, 30))
    self.display.blit(self.snapshot,(0,0))
    # keep only the largest blob of that color
    connected = mask.connected_component()
    # make sure the blob is big enough that it isn't just noise
    if mask.count() > 100:
        # find the center of the blob
        coord = mask.centroid()
        # draw a circle with size variable on the size of the blob
        pygame.draw.circle(self.display, (0,255,0), coord, max(min(50,mask.count()/400),5))
    pygame.display.flip()
```



This is just the most basic example. You can track multiple different colored blobs, find the outlines of objects, have collision detection between real life and in game objects, get the angle of an object to allow for even finer control, and more. Have fun!

Pygame Tutorials - Line By Line Chimp Example

Line By Line Chimp

Author: Pete Shinnars

Contact: pete@shinnars.org

pygame/examples/chimp.py

```
#!/usr/bin/env python
""" pygame.examples.chimp

This simple example is used for the line-by-line tutorial
that comes with pygame. It is based on a 'popular' web banner.
Note there are comments here, but for the full explanation,
follow along in the tutorial.
"""

# Import Modules
import os
import pygame as pg
from pygame.compat import geterror

if not pg.font:
    print("Warning, fonts disabled")
if not pg.mixer:
    print("Warning, sound disabled")

main_dir = os.path.split(os.path.abspath(__file__))[0]
data_dir = os.path.join(main_dir, "data")

# functions to create our resources
def load_image(name, colorkey=None):
    fullname = os.path.join(data_dir, name)
    try:
        image = pg.image.load(fullname)
    except pg.error:
        print("Cannot load image:", fullname)
        raise SystemExit(str(geterror()))
    image = image.convert()
    if colorkey is not None:
        if colorkey == -1:
            colorkey = image.get_at((0, 0))
        image.set_colorkey(colorkey, pg.RLEACCEL)
    return image, image.get_rect()

def load_sound(name):
    class NoneSound:
        def play(self):
            pass

    if not pg.mixer or not pg.mixer.get_init():
        return NoneSound()
    fullname = os.path.join(data_dir, name)
    try:
```

```

        sound = pg.mixer.Sound(fullname)
    except pg.error:
        print("Cannot load sound: %s" % fullname)
        raise SystemExit(str(geterror()))
    return sound

# classes for our game objects
class Fist(pg.sprite.Sprite):
    """moves a clenched fist on the screen, following the mouse"""

    def __init__(self):
        pg.sprite.Sprite.__init__(self) # call Sprite initializer
        self.image, self.rect = load_image("fist.bmp", -1)
        self.punching = 0

    def update(self):
        """move the fist based on the mouse position"""
        pos = pg.mouse.get_pos()
        self.rect.midtop = pos
        if self.punching:
            self.rect.move_ip(5, 10)

    def punch(self, target):
        """returns true if the fist collides with the target"""
        if not self.punching:
            self.punching = 1
            hitbox = self.rect.inflate(-5, -5)
            return hitbox.colliderect(target.rect)

    def unpunch(self):
        """called to pull the fist back"""
        self.punching = 0

class Chimp(pg.sprite.Sprite):
    """moves a monkey critter across the screen. it can spin the
    monkey when it is punched."""

    def __init__(self):
        pg.sprite.Sprite.__init__(self) # call Sprite initializer
        self.image, self.rect = load_image("chimp.bmp", -1)
        screen = pg.display.get_surface()
        self.area = screen.get_rect()
        self.rect.topleft = 10, 10
        self.move = 9
        self.dizzy = 0

    def update(self):
        """walk or spin, depending on the monkeys state"""
        if self.dizzy:
            self._spin()
        else:
            self._walk()

    def _walk(self):
        """move the monkey across the screen, and turn at the ends"""
        newpos = self.rect.move((self.move, 0))
        if not self.area.contains(newpos):
            if self.rect.left < self.area.left or self.rect.right > self.area.right:

```

```

        self.move = -self.move
        newpos = self.rect.move((self.move, 0))
        self.image = pg.transform.flip(self.image, 1, 0)
        self.rect = newpos

def _spin(self):
    """spin the monkey image"""
    center = self.rect.center
    self.dizzy = self.dizzy + 12
    if self.dizzy >= 360:
        self.dizzy = 0
        self.image = self.original
    else:
        rotate = pg.transform.rotate
        self.image = rotate(self.original, self.dizzy)
    self.rect = self.image.get_rect(center=center)

def punched(self):
    """this will cause the monkey to start spinning"""
    if not self.dizzy:
        self.dizzy = 1
        self.original = self.image

def main():
    """this function is called when the program starts.
    it initializes everything it needs, then runs in
    a loop until the function returns."""
    # Initialize Everything
    pg.init()
    screen = pg.display.set_mode((468, 60))
    pg.display.set_caption("Monkey Fever")
    pg.mouse.set_visible(0)

    # Create The Background
    background = pg.Surface(screen.get_size())
    background = background.convert()
    background.fill((250, 250, 250))

    # Put Text On The Background, Centered
    if pg.font:
        font = pg.font.Font(None, 36)
        text = font.render("Pummel The Chimp, And Win $$$", 1, (10, 10, 10))
        textpos = text.get_rect(centerx=background.get_width() / 2)
        background.blit(text, textpos)

    # Display The Background
    screen.blit(background, (0, 0))
    pg.display.flip()

    # Prepare Game Objects
    clock = pg.time.Clock()
    whiff_sound = load_sound("whiff.wav")
    punch_sound = load_sound("punch.wav")
    chimp = Chimp()
    fist = Fist()
    allsprites = pg.sprite.RenderPlain((fist, chimp))

    # Main Loop
    going = True

```

```

while going:
    clock.tick(60)

    # Handle Input Events
    for event in pg.event.get():
        if event.type == pg.QUIT:
            going = False
        elif event.type == pg.KEYDOWN and event.key == pg.K_ESCAPE:
            going = False
        elif event.type == pg.MOUSEBUTTONDOWN:
            if fist.punch(chimp):
                punch_sound.play() # punch
                chimp.punched()
            else:
                whiff_sound.play() # miss
        elif event.type == pg.MOUSEBUTTONUP:
            fist.unpunch()

    allsprites.update()

    # Draw Everything
    screen.blit(background, (0, 0))
    allsprites.draw(screen)
    pg.display.flip()

pg.quit()

# Game Over

# this calls the 'main' function when this script is executed
if __name__ == "__main__":
    main()

```

Introduction

In the *pygame* examples there is a simple example named "chimp". This example simulates a punchable monkey moving around a small screen with promises of riches and reward. The example itself is very simple, and a bit thin on error-checking code. This example program demonstrates many of *pygame*'s abilities, like creating a graphics window, loading images and sound files, rendering TTF text, and basic event and mouse handling.

The program and images can be found inside the standard source distribution of *pygame*. For version 1.3 of *pygame*, this example was completely rewritten to add a couple more features and correct error checking. This about doubled the size of the original example, but now gives us much more to look at, as well as the code I can recommend reusing for your own projects.

This tutorial will go through the code block by block. Explaining how the code works. There will also be mention of how the code could be improved and what error checking could help out.

This is an excellent tutorial for people getting their first look at the *pygame* code. Once *pygame* is fully installed, you can find and run the chimp demo for yourself in the examples directory.

(no, this is not a banner ad, its the screenshot)

Pummel The Chimp, And Win \$\$\$



Full Source

Import Modules

This is the code that imports all the needed modules into your program. It also checks for the availability of some of the optional *pygame* modules.


```
import os, sys
import pygame
from pygame.locals import *

if not pygame.font: print('Warning, fonts disabled')
if not pygame.mixer: print('Warning, sound disabled')
```

First, we import the standard "os" and "sys" python modules. These allow us to do things like create platform independent file paths.

In the next line, we import the pygame package. When pygame is imported it imports all the modules belonging to pygame. Some pygame modules are optional, and if they aren't found, their value is set to *None*.

There is a special *pygame* module named **locals**. This module contains a subset of *pygame*. The members of this module are commonly used constants and functions that have proven useful to put into your program's global namespace. This locals module includes functions like "Rect" to create a rectangle object, and many constants like "QUIT, HWSURFACE" that are used to interact with the rest of *pygame*. Importing the locals module into the global namespace like this is entirely optional. If you choose not to import it, all the members of locals are always available in the *pygame* module.

Lastly, we decide to print a nice warning message if the **font** or **mixer** modules in pygame are not available.

Loading Resources

Here we have two functions we can use to load images and sounds. We will look at each function individually in this section.

```
def load_image(name, colorkey=None):
    fullname = os.path.join('data', name)
    try:
        image = pygame.image.load(fullname)
    except pygame.error as message:
        print('Cannot load image:', name)
        raise SystemExit(message)
    image = image.convert()
    if colorkey is not None:
        if colorkey is -1:
            colorkey = image.get_at((0, 0))
        image.set_colorkey(colorkey, RLEACCEL)
    return image, image.get_rect()
```

This function takes the name of an image to load. It also optionally takes an argument it can use to set a colorkey for the image. A colorkey is used in graphics to represent a color of the image that is transparent.

The first thing this function does is create a full pathname to the file. In this example all the resources are in a "data" subdirectory. By using the *os.path.join* function, a pathname will be created that works for whatever platform the game is running on.

Next we load the image using the **pygame.image.load()** function. We wrap this function in a try/except block, so if there is a problem loading the image, we can exit gracefully. After the image is loaded, we make an important call to the *convert()* function. This makes a new copy of a Surface and converts its color format and depth to match the display. This means blitting the image to the screen will happen as quickly as possible.

Last, we set the colorkey for the image. If the user supplied an argument for the colorkey argument we use that value as the colorkey for the image. This would usually just be a color RGB value, like (255, 255, 255) for white. You can also pass a value of -1 as the colorkey. In this case the function will lookup the color at the topleft pixel of the image, and use that color for the colorkey.

```
def load_sound(name):
    class NoneSound:
        def play(self): pass
    if not pygame.mixer:
        return NoneSound()
    fullname = os.path.join('data', name)
```

```

try:
    sound = pygame.mixer.Sound(fullname)
except pygame.error as message:
    print('Cannot load sound:', fullname)
    raise SystemExit(message)
return sound

```

Next is the function to load a sound file. The first thing this function does is check to see if the `pygame.mixer` module was imported correctly. If not, it returns a small class instance that has a dummy play method. This will act enough like a normal Sound object for this game to run without any extra error checking.

This function is similar to the image loading function, but handles some different problems. First we create a full path to the sound image, and load the sound file inside a try/except block. Then we simply return the loaded Sound object.

Game Object Classes

Here we create two classes to represent the objects in our game. Almost all the logic for the game goes into these two classes. We will look over them one at a time here.

```

class Fist(pygame.sprite.Sprite):
    """moves a clenched fist on the screen, following the mouse"""
    def __init__(self):
        pygame.sprite.Sprite.__init__(self) # call Sprite initializer
        self.image, self.rect = load_image('fist.bmp', -1)
        self.punching = 0

    def update(self):
        """move the fist based on the mouse position"""
        pos = pygame.mouse.get_pos()
        self.rect.midtop = pos
        if self.punching:
            self.rect.move_ip(5, 10)

    def punch(self, target):
        """returns true if the fist collides with the target"""
        if not self.punching:
            self.punching = 1
            hitbox = self.rect.inflate(-5, -5)
            return hitbox.colliderect(target.rect)

    def unpunch(self):
        """called to pull the fist back"""
        self.punching = 0

```

Here we create a class to represent the players fist. It is derived from the *Sprite* class included in the `pygame.sprite` module. The `__init__` function is called when new instances of this class are created. The first thing we do is be sure to call the `__init__` function for our base class. This allows the Sprite's `__init__` function to prepare our object for use as a sprite. This game uses one of the sprite drawing Group classes. These classes can draw sprites that have an "image" and "rect" attribute. By simply changing these two attributes, the renderer will draw the current image at the current position.

All sprites have an `update()` method. This function is typically called once per frame. It is where you should put code that moves and updates the variables for the sprite. The `update()` method for the fist moves the fist to the location of the mouse pointer. It also offsets the fist position slightly if the fist is in the "punching" state.

The following two functions `punch()` and `unpunch()` change the punching state for the fist. The `punch()` method also returns a true value if the fist is colliding with the given target sprite.

```

class Chimp(pygame.sprite.Sprite):
    """moves a monkey critter across the screen. it can spin the
    monkey when it is punched."""
    def __init__(self):

```

```

pygame.sprite.Sprite.__init__(self) # call Sprite initializer
self.image, self.rect = load_image('chimp.bmp', -1)
screen = pygame.display.get_surface()
self.area = screen.get_rect()
self.rect.topleft = 10, 10
self.move = 9
self.dizzy = 0

def update(self):
    """walk or spin, depending on the monkeys state"""
    if self.dizzy:
        self._spin()
    else:
        self._walk()

def _walk(self):
    """move the monkey across the screen, and turn at the ends"""
    newpos = self.rect.move((self.move, 0))
    if not self.area.contains(newpos):
        if self.rect.left < self.area.left or \
            self.rect.right > self.area.right:
            self.move = -self.move
            newpos = self.rect.move((self.move, 0))
            self.image = pygame.transform.flip(self.image, 1, 0)
        self.rect = newpos

def _spin(self):
    """spin the monkey image"""
    center = self.rect.center
    self.dizzy += 12
    if self.dizzy >= 360:
        self.dizzy = 0
        self.image = self.original
    else:
        rotate = pygame.transform.rotate
        self.image = rotate(self.original, self.dizzy)
    self.rect = self.image.get_rect(center=center)

def punched(self):
    """this will cause the monkey to start spinning"""
    if not self.dizzy:
        self.dizzy = 1
        self.original = self.image

```

The *Chimp* class is doing a little more work than the fist, but nothing more complex. This class will move the chimp back and forth across the screen. When the monkey is punched, he will spin around to exciting effect. This class is also derived from the base `sprite` class, and is initialized the same as the fist. While initializing, the class also sets the attribute "area" to be the size of the display screen.

The *update* function for the chimp simply looks at the current "dizzy" state, which is true when the monkey is spinning from a punch. It calls either the `_spin` or `_walk` method. These functions are prefixed with an underscore. This is just a standard python idiom which suggests these methods should only be used by the *Chimp* class. We could go so far as to give them a double underscore, which would tell python to really try to make them private methods, but we don't need such protection. :)

The `_walk` method creates a new position for the monkey by moving the current rect by a given offset. If this new position crosses outside the display area of the screen, it reverses the movement offset. It also mirrors the image using the `pygame.transform.flip()` function. This is a crude effect that makes the monkey look like he's turning the direction he is moving.

The `_spin` method is called when the monkey is currently "dizzy". The dizzy attribute is used to store the current amount of rotation. When the monkey has rotated all the way around (360 degrees) it resets the monkey image back

to the original, non-rotated version. Before calling the `pygame.transform.rotate()` function, you'll see the code makes a local reference to the function simply named "rotate". There is no need to do that for this example, it is just done here to keep the following line's length a little shorter. Note that when calling the `rotate` function, we are always rotating from the original monkey image. When rotating, there is a slight loss of quality. Repeatedly rotating the same image and the quality would get worse each time. Also, when rotating an image, the size of the image will actually change. This is because the corners of the image will be rotated out, making the image bigger. We make sure the center of the new image matches the center of the old image, so it rotates without moving.

The last method is `punched()` which tells the sprite to enter its dizzy state. This will cause the image to start spinning. It also makes a copy of the current image named "original".

Initialize Everything

Before we can do much with pygame, we need to make sure its modules are initialized. In this case we will also open a simple graphics window. Now we are in the `main()` function of the program, which actually runs everything.

```
pygame.init()
screen = pygame.display.set_mode((468, 60))
pygame.display.set_caption('Monkey Fever')
pygame.mouse.set_visible(0)
```

The first line to initialize `pygame` takes care of a bit of work for us. It checks through the imported `pygame` modules and attempts to initialize each one of them. It is possible to go back and check if modules failed to initialize, but we won't bother here. It is also possible to take a lot more control and initialize each specific module by hand. That type of control is generally not needed, but is available if you desire.

Next we set up the display graphics mode. Note that the `pygame.display` module is used to control all the display settings. In this case we are asking for a simple skinny window. There is an entire separate tutorial on setting up the graphics mode, but if we really don't care, `pygame` will do a good job of getting us something that works. Pygame will pick the best color depth, since we haven't provided one.

Last we set the window title and turn off the mouse cursor for our window. Very basic to do, and now we have a small black window ready to do our bidding. Usually the cursor defaults to visible, so there is no need to really set the state unless we want to hide it.

Create The Background

Our program is going to have text message in the background. It would be nice for us to create a single surface to represent the background and repeatedly use that. The first step is to create the surface.

```
background = pygame.Surface(screen.get_size())
background = background.convert()
background.fill((250, 250, 250))
```

This creates a new surface for us that is the same size as the display window. Note the extra call to `convert()` after creating the Surface. The `convert` with no arguments will make sure our background is the same format as the display window, which will give us the fastest results.

We also fill the entire background with a solid whitish color. Fill takes an RGB triplet as the color argument.

Put Text On The Background, Centered

Now that we have a background surface, lets get the text rendered to it. We only do this if we see the `pygame.font` module has imported properly. If not, we just skip this section.

```
if pygame.font:
    font = pygame.font.Font(None, 36)
    text = font.render("Pummel The Chimp, And Win $$$", 1, (10, 10, 10))
    textpos = text.get_rect(centerx=background.get_width()/2)
    background.blit(text, textpos)
```

As you see, there are a couple steps to getting this done. First we must create the font object and render it into a new surface. We then find the center of that new surface and blit (paste) it onto the background.

The font is created with the *font* module's *Font()* constructor. Usually you will pass the name of a TrueType font file to this function, but we can also pass *None*, which will use a default font. The *Font* constructor also needs to know the size of font we want to create.

We then render that font into a new surface. The *render* function creates a new surface that is the appropriate size for our text. In this case we are also telling render to create antialiased text (for a nice smooth look) and to use a dark grey color.

Next we need to find the centered position of the text on our display. We create a "Rect" object from the text dimensions, which allows us to easily assign it to the screen center.

Finally we blit (blit is like a copy or paste) the text onto the background image.

Display The Background While Setup Finishes

We still have a black window on the screen. Lets show our background while we wait for the other resources to load.

```
screen.blit(background, (0, 0))
pygame.display.flip()
```

This will blit our entire background onto the display window. The blit is self explanatory, but what about this flip routine?

In pygame, changes to the display surface are not immediately visible. Normally, a display must be updated in areas that have changed for them to be visible to the user. With double buffered displays the display must be swapped (or flipped) for the changes to become visible. In this case the *flip()* function works nicely because it simply handles the entire window area and handles both single- and double-buffered surfaces.

Prepare Game Object

Here we create all the objects that the game is going to need.

```
whiff_sound = load_sound('whiff.wav')
punch_sound = load_sound('punch.wav')
chimp = Chimp()
fist = Fist()
allsprites = pygame.sprite.RenderPlain((fist, chimp))
clock = pygame.time.Clock()
```

First we load two sound effects using the *load_sound* function we defined above. Then we create an instance of each of our sprite classes. And lastly we create a sprite **Group** which will contain all our sprites.

We actually use a special sprite group named **RenderPlain**. This sprite group can draw all the sprites it contains to the screen. It is called *RenderPlain* because there are actually more advanced Render groups. But for our game, we just need simple drawing. We create the group named "allsprites" by passing a list with all the sprites that should belong in the group. We could later on add or remove sprites from this group, but in this game we won't need to.

The *clock* object we create will be used to help control our game's framerate. we will use it in the main loop of our game to make sure it doesn't run too fast.

Main Loop

Nothing much here, just an infinite loop.

```
while 1:
    clock.tick(60)
```

All games run in some sort of loop. The usual order of things is to check on the state of the computer and user input, move and update the state of all the objects, and then draw them to the screen. You'll see that this example is no different.

We also make a call to our *clock* object, which will make sure our game doesn't run faster than 60 frames per second.

Handle All Input Events

This is an extremely simple case of working the event queue.

```
for event in pygame.event.get():
    if event.type == QUIT:
        return
    elif event.type == KEYDOWN and event.key == K_ESCAPE:
        return
    elif event.type == MOUSEBUTTONDOWN:
        if fist.punch(chimp):
            punch_sound.play() # punch
            chimp.punched()
        else:
            whiff_sound.play() # miss
    elif event.type == MOUSEBUTTONUP:
        fist.unpunch()
```

First we get all the available Events from pygame and loop through each of them. The first two tests see if the user has quit our game, or pressed the escape key. In these cases we just return from the *main()* function and the program cleanly ends.

Next we just check to see if the mouse button was pressed or released. If the button was pressed, we ask the fist object if it has collided with the chimp. We play the appropriate sound effect, and if the monkey was hit, we tell him to start spinning (by calling his *punched()* method).

Update the Sprites

```
allsprites.update()
```

Sprite groups have an *update()* method, which simply calls the update method for all the sprites it contains. Each of the objects will move around, depending on which state they are in. This is where the chimp will move one step side to side, or spin a little farther if he was recently punched.

Draw The Entire Scene

Now that all the objects are in the right place, time to draw them.

```
screen.blit(background, (0, 0))
allsprites.draw(screen)
pygame.display.flip()
```

The first blit call will draw the background onto the entire screen. This erases everything we saw from the previous frame (slightly inefficient, but good enough for this game). Next we call the *draw()* method of the sprite container. Since this sprite container is really an instance of the "DrawPlain" sprite group, it knows how to draw our sprites. Lastly, we *flip()* the contents of pygame's software double buffer to the screen. This makes everything we've drawn visible all at once.

Game Over

User has quit, time to clean up.

Cleaning up the running game in *pygame* is extremely simple. In fact since all variables are automatically destructed, we really don't have to do anything.

Pygame Tutorials - Setting Display Modes

Setting Display Modes

Author: Pete Shinnars

Contact: pete@shinnars.org

Introduction

Setting the display mode in *pygame* creates a visible image surface on the monitor. This surface can either cover the full screen, or be windowed on platforms that support a window manager. The display surface is nothing more than a standard *pygame* surface object. There are special functions needed in the `pygame.display` module to keep the image surface contents updated on the monitor.

Setting the display mode in *pygame* is an easier task than with most graphic libraries. The advantage is if your display mode is not available, *pygame* will emulate the display mode that you asked for. *Pygame* will select a display resolution and color depth that best matches the settings you have requested, then allow you to access the display with the format you have requested. In reality, since the `pygame.display` module is a binding around the SDL library, SDL is really doing all this work.

There are advantages and disadvantages to setting the display mode in this manner. The advantage is that if your game requires a specific display mode, your game will run on platforms that do not support your requirements. It also makes life easier when your getting something started, it is always easy to go back later and make the mode selection a little more particular. The disadvantage is that what you request is not always what you will get. There is also a performance penalty when the display mode must be emulated. This tutorial will help you understand the different methods for querying the platforms display capabilities, and setting the display mode for your game.

Setting Basics

The first thing to learn about is how to actually set the current display mode. The display mode may be set at any time after the `pygame.display` module has been initialized. If you have previously set the display mode, setting it again will change the current mode. Setting the display mode is handled with the function `pygame.display.set_mode((width, height), flags, depth)`. The only required argument in this function is a sequence containing the width and height of the new display mode. The depth flag is the requested bits per pixel for the surface. If the given depth is 8, *pygame* will create a color-mapped surface. When given a higher bit depth, *pygame* will use a packed color mode. Much more information about depths and color modes can be found in the documentation for the display and surface modules. The default value for depth is 0. When given an argument of 0, *pygame* will select the best bit depth to use, usually the same as the system's current bit depth. The flags argument lets you control extra features for the display mode. You can create the display surface in hardware memory with the `HWSURFACE` flag. Again, more information about this is found in the *pygame* reference documents.

How to Decide

So how do you select a display mode that is going to work best with your graphic resources and the platform your game is running on? There are several methods for gathering information about the display device. All of these methods must be called after the display module has been initialized, but you likely want to call them before setting the display mode. First, `pygame.display.Info()` will return a special object type of `VidInfo`, which can tell you a lot about the graphics driver capabilities. The function `pygame.display.list_modes(depth, flags)` can be used to find the supported graphic modes by the system. `pygame.display.mode_ok((width, height), flags, depth)` takes the same arguments as `set_mode()`, but returns the closest matching bit depth to the one you request. Lastly, `pygame.display.get_driver()` will return the name of the graphics driver selected by *pygame*.

Just remember the golden rule. *Pygame* will work with pretty much any display mode you request. Some display modes will need to be emulated, which will slow your game down, since *pygame* will need to convert every update you make to the "real" display mode. The best bet is to always let *pygame* choose the best bit depth, and convert all your graphic resources to that format when they are loaded. You let *pygame* choose it's bit depth by calling `set_mode()` with no depth argument or a depth of 0, or you can call `mode_ok()` to find a closest matching bit depth to what you need.

When your display mode is windowed, you usually must match the same bit depth as the desktop. When you are fullscreen, some platforms can switch to any bit depth that best suits your needs. You can find the depth of the current desktop if you get a `VidInfo` object before ever setting your display mode.

After setting the display mode, you can find out information about it's settings by getting a `VidInfo` object, or by calling any of the `Surface.get*` methods on the display surface.

Functions

These are the routines you can use to determine the most appropriate display mode. You can find more information about these functions in the display module documentation.

`pygame.display.mode_ok(size, flags, depth)`

This function takes the exact same arguments as `pygame.display.set_mode()`. It returns the best available bit depth for the mode you have described. If this returns zero, then the desired display mode is not available without emulation.

`pygame.display.list_modes(depth, flags)`

Returns a list of supported display modes with the requested depth and flags. An empty list is returned when there are no modes. The flags argument defaults to `FULLSCREEN`. If you specify your own flags without `FULLSCREEN`, you will likely get a return value of -1. This means that any display size is fine, since the display will be windowed. Note that the listed modes are sorted largest to smallest.

`pygame.display.Info()`

This function returns an object with many members describing the display device. Printing the `VidInfo` object will quickly show you all the members and values for this object.

```
>>> import pygame.display
>>> pygame.display.init()
>>> info = pygame.display.Info()
>>> print info
<VideoInfo(hw = 1, wm = 1, video_mem = 27354
          blit_hw = 1, blit_hw_CC = 1, blit_hw_A = 0,
          blit_sw = 1, blit_sw_CC = 1, blit_sw_A = 0,
          bitsize = 32, bytesize = 4,
          masks = (16711680, 65280, 255, 0),
          shifts = (16, 8, 0, 0),
          losses = (0, 0, 0, 8)>
```

You can test all these flags as simply members of the `VidInfo` object. The different blit flags tell if hardware acceleration is supported when blitting from the various types of surfaces to a hardware surface.

Examples

Here are some examples of different methods to init the graphics display. They should help you get an idea of how to go about setting your display mode.

```
>>> #give me the best depth with a 640 x 480 windowed display
>>> pygame.display.set_mode((640, 480))

>>> #give me the biggest 16-bit display available
>>> modes = pygame.display.list_modes(16)
>>> if not modes:
...     print '16-bit not supported'
... else:
...     print 'Found Resolution:', modes[0]
...     pygame.display.set_mode(modes[0], FULLSCREEN, 16)

>>> #need an 8-bit surface, nothing else will do
>>> if pygame.display.mode_ok((800, 600), 0, 8) != 8:
...     print 'Can only work with an 8-bit display, sorry'
... else:
...     pygame.display.set_mode((800, 600), 0, 8)
```

Pygame Tutorials - Import and Initialize

Import and Initialize

Author: Pete Shinnars

Contact: pete@shinners.org

Getting pygame imported and initialized is a very simple process. It is also flexible enough to give you control over what is happening. Pygame is a collection of different modules in a single python package. Some of the modules are written in C, and some are written in python. Some modules are also optional, and might not always be present.

This is just a quick introduction on what is going on when you import pygame. For a clearer explanation definitely see the pygame examples.

Import

First we must import the pygame package. Since pygame version 1.4 this has been updated to be much easier. Most games will import all of pygame like this.

```
import pygame
from pygame.locals import *
```

The first line here is the only necessary one. It imports all the available pygame modules into the pygame package. The second line is optional, and puts a limited set of constants and functions into the global namespace of your script.

An important thing to keep in mind is that several pygame modules are optional. For example, one of these is the font module. When you "import pygame", pygame will check to see if the font module is available. If the font module is available it will be imported as "pygame.font". If the module is not available, "pygame.font" will be set to None. This makes it fairly easy to later on test if the font module is available.

Init

Before you can do much with pygame, you will need to initialize it. The most common way to do this is just make one call.

```
pygame.init()
```

This will attempt to initialize all the pygame modules for you. Not all pygame modules need to be initialized, but this will automatically initialize the ones that do. You can also easily initialize each pygame module by hand. For example to only initialize the font module you would just call.

```
pygame.font.init()
```

Note that if there is an error when you initialize with "pygame.init()", it will silently fail. When hand initializing modules like this, any errors will raise an exception. Any modules that must be initialized also have a "get_init()" function, which will return true if the module has been initialized.

It is safe to call the init() function for any module more than once.

Quit

Modules that are initialized also usually have a quit() function that will clean up. There is no need to explicitly call these, as pygame will cleanly quit all the initialized modules when python finishes.

Making Games With Pygame

Making Games With Pygame

Revision: Pygame fundamentals

2. Revision: Pygame fundamentals

2.1. The basic Pygame game

For the sake of revision, and to ensure that you are familiar with the basic structure of a Pygame program, I'll briefly run through a basic Pygame program, which will display no more than a window with some text in it, that should, by the end, look something like this (though of course the window decoration will probably be different on your system):



The full code for this example looks like this:

```
#!/usr/bin/python

import pygame
from pygame.locals import *

def main():
    # Initialise screen
    pygame.init()
    screen = pygame.display.set_mode((150, 50))
    pygame.display.set_caption('Basic Pygame program')

    # Fill background
    background = pygame.Surface(screen.get_size())
    background = background.convert()
    background.fill((250, 250, 250))

    # Display some text
    font = pygame.font.Font(None, 36)
    text = font.render("Hello There", 1, (10, 10, 10))
    textpos = text.get_rect()
    textpos.centerx = background.get_rect().centerx
    background.blit(text, textpos)

    # Blit everything to the screen
    screen.blit(background, (0, 0))
    pygame.display.flip()

    # Event loop
    while 1:
        for event in pygame.event.get():
            if event.type == QUIT:
                return

        screen.blit(background, (0, 0))
        pygame.display.flip()

if __name__ == '__main__': main()
```

2.2. Basic Pygame objects

As you can see, the code consists of three main objects: the screen, the background, and the text. Each of these objects is created by first calling an instance of an in-built Pygame object, and then modifying it to fit our needs. The screen is a slightly special case, because we still modify the display through Pygame calls, rather than calling methods belonging to the screen object. But for all other Pygame objects, we first create the object as a copy of a Pygame object, giving it some attributes, and build our game objects from them.

With the background, we first create a Pygame Surface object, and make it the size of the screen. We then perform the `convert()` operation to convert the Surface to a single pixel format. This is more obviously necessary when we have several images and surfaces, all of different pixel formats, which makes rendering them quite slow. By converting all the surfaces, we can drastically speed up rendering times. Finally, we fill the background surface with white (255, 255, 255). These values are *RGB* (Red Green Blue), and can be worked out from any good paint program.

With the text, we require more than one object. First, we create a font object, which defines which font to use, and the size of the font. Then we create a text object, by using the `render` method that belongs to our font object, supplying three arguments: the text to be rendered, whether or not it should be anti-aliased (1=yes, 0=no), and the color of the text (again in RGB format). Next we create a third text object, which gets the rectangle for the text. The easiest way to understand this is to imagine drawing a rectangle that will surround all of the text; you can then use this rectangle to get/set the position of the text on the screen. So in this example we get the rectangle, set its `centerx` attribute to be the `centerx` attribute of the background (so the text's center will be the same as the background's center, i.e. the text will be centered on the screen on the x axis). We could also set the y coordinate, but it's not any different so I left the text at the top of the screen. As the screen is small anyway, it didn't seem necessary.

2.3. Blitting

Now we have created our game objects, we need to actually render them. If we didn't and we ran the program, we'd just see a blank window, and the objects would remain invisible. The term used for rendering objects is *blitting*, which is where you copy the pixels belonging to said object onto the destination object. So to render the background object, you blit it onto the screen. In this example, to make things simple, we blit the text onto the background (so the background will now have a copy of the text on it), and then blit the background onto the screen.

Blitting is one of the slowest operations in any game, so you need to be careful not to blit too much onto the screen in every frame. If you have a background image, and a ball flying around the screen, then you could blit the background and then the ball in every frame, which would cover up the ball's previous position and render the new ball, but this would be pretty slow. A better solution is to blit the background onto the area that the ball previously occupied, which can be found by the ball's previous rectangle, and then blitting the ball, so that you are only blitting two small areas.

2.4. The event loop

Once you've set the game up, you need to put it into a loop so that it will continuously run until the user signals that he/she wants to exit. So you start an open `while` loop, and then for each iteration of the loop, which will be each frame of the game, update the game. The first thing is to check for any Pygame events, which will be the user hitting the keyboard, clicking a mouse button, moving a joystick, resizing the window, or trying to close it. In this case, we simply want to watch out for user trying to quit the game by closing the window, in which case the game should `return`, which will end the `while` loop. Then we simply need to re-blit the background, and flip (update) the display to have everything drawn. OK, as nothing moves or happens in this example, we don't strictly speaking need to re-blit the background in every iteration, but I put it in because when things are moving around on the screen, you will need to do all your blitting here.

2.5. Ta-da!

And that's it - your most basic Pygame game! All games will take a form similar to this, but with lots more code for the actual game functions themselves, which are more to do your with programming, and less guided in structure by the workings of Pygame. This is what this tutorial is really about, and will now go onto.

Kicking things off

3. Kicking things off

The first sections of code are relatively simple, and, once written, can usually be reused in every game you consequently make. They will do all of the boring, generic tasks like loading modules, loading images, opening networking connections, playing music, and so on. They will also include some simple but effective error handling, and any customisation you wish to provide on top of functions provided by modules like `sys` and `pygame`.

3.1. The first lines, and loading modules

First off, you need to start off your game and load up your modules. It's always a good idea to set a few things straight at the top of the main source file, such as the name of the file, what it contains, the license it is under, and any other helpful info you might want to give those who will be looking at it. Then you can load modules, with some error checking so that Python doesn't print out a nasty traceback, which non-programmers won't understand. The code is fairly simple, so I won't bother explaining any of it:

```
#!/usr/bin/env python
#
```

```

# Tom's Pong
# A simple pong game with realistic physics and AI
# http://www.tomchance.uklinux.net/projects/pong.shtml
#
# Released under the GNU General Public License

VERSION = "0.4"

try:
    import sys
    import random
    import math
    import os
    import getopt
    import pygame
    from socket import *
    from pygame.locals import *
except ImportError, err:
    print "couldn't load module. %s" % (err)
    sys.exit(2)

```

3.2. Resource handling functions

In the Line By Line Chimp example, the first code to be written was for loading images and sounds. As these were totally independent of any game logic or game objects, they were written as separate functions, and were written first so that later code could make use of them. I generally put all my code of this nature first, in their own, classless functions; these will, generally speaking, be resource handling functions. You can of course create classes for these, so that you can group them together, and maybe have an object with which you can control all of your resources. As with any good programming environment, it's up to you to develop your own best practice and style.

It's always a good idea to write your own resource handling functions, because although Pygame has methods for opening images and sounds, and other modules will have their methods of opening other resources, those methods can take up more than one line, they can require consistent modification by yourself, and they often don't provide satisfactory error handling. Writing resource handling functions gives you sophisticated, reusable code, and gives you more control over your resources. Take this example of an image loading function:

```

def load_png(name):
    """ Load image and return image object"""
    fullname = os.path.join('data', name)
    try:
        image = pygame.image.load(fullname)
        if image.get_alpha() is None:
            image = image.convert()
        else:
            image = image.convert_alpha()
    except pygame.error, message:
        print 'Cannot load image:', fullname
        raise SystemExit, message
    return image, image.get_rect()

```

Here we make a more sophisticated image loading function than the one provided by `pygame.image.load()`. Note that the first line of the function is a documentation string describing what the function does, and what object(s) it returns. The function assumes that all of your images are in a directory called `data`, and so it takes the filename and creates the full pathname, for example `data/ball.png`, using the `os` module to ensure cross-platform compatibility. Then it tries to load the image, and convert any alpha regions so you can achieve transparency, and it returns a more human-readable error if there's a problem. Finally it returns the image object, and its `rect`.

You can make similar functions for loading any other resources, such as loading sounds. You can also make resource handling classes, to give you more flexibility with more complex resources. For example, you could make a music class, with an `__init__` function that loads the sound (perhaps borrowing from a `load_sound()` function), a function to pause the music, and a function to restart. Another handy resource handling class is for network

connections. Functions to open sockets, pass data with suitable security and error checking, close sockets, finger addresses, and other network tasks, can make writing a game with network capabilities relatively painless.

Remember the chief task of these functions/classes is to ensure that by the time you get around to writing game object classes, and the main loop, there's almost nothing left to do. Class inheritance can make these basic classes especially handy. Don't go overboard though; functions which will only be used by one class should be written as part of that class, not as a global function.

Game object classes

4. Game object classes

Once you've loaded your modules, and written your resource handling functions, you'll want to get on to writing some game objects. The way this is done is fairly simple, though it can seem complex at first. You write a class for each type of object in the game, and then create an instance of those classes for the objects. You can then use those classes' methods to manipulate the objects, giving objects some motion and interactive capabilities. So your game, in pseudo-code, will look like this:

```
#!/usr/bin/python

# [load modules here]

# [resource handling functions here]

class Ball:
    # [ball functions (methods) here]
    # [e.g. a function to calculate new position]
    # [and a function to check if it hits the side]

def main:
    # [initiate game environment here]

    # [create new object as instance of ball class]
    ball = Ball()

    while 1:
        # [check for user input]

        # [call ball's update function]
        ball.update()
```

This is, of course, a very simple example, and you'd need to put in all the code, instead of those little bracketed comments. But you should get the basic idea. You create a class, into which you put all the functions for a ball, including `__init__`, which would create all the ball's attributes, and `update`, which would move the ball to its new position, before blitting it onto the screen in this position.

You can then create more classes for all of your other game objects, and then create instances of them so that you can handle them easily in the `main` function and the main program loop. Contrast this with initiating the ball in the `main` function, and then having lots of classless functions to manipulate a set ball object, and you'll hopefully see why using classes is an advantage: It allows you to put all of the code for each object in one place; it makes using objects easier; it makes adding new objects, and manipulating them, more flexible. Rather than adding more code for each new ball object, you could simply create new instances of the `Ball` class for each new ball object. Magic!

4.1. A simple ball class

Here is a simple class with the functions necessary for creating a ball object that will, if the `update` function is called in the main loop, move across the screen:

```
class Ball(pygame.sprite.Sprite):
    """A ball that will move across the screen
    Returns: ball object
    Functions: update, calcnewpos
```

```

Attributes: area, vector"""

def __init__(self, vector):
    pygame.sprite.Sprite.__init__(self)
    self.image, self.rect = load_png('ball.png')
    screen = pygame.display.get_surface()
    self.area = screen.get_rect()
    self.vector = vector

def update(self):
    newpos = self.calcnewpos(self.rect, self.vector)
    self.rect = newpos

def calcnewpos(self, rect, vector):
    (angle, z) = vector
    (dx, dy) = (z * math.cos(angle), z * math.sin(angle))
    return rect.move(dx, dy)

```

Here we have the `Ball` class, with an `__init__` function that sets the ball up, an `update` function that changes the ball's rectangle to be in the new position, and a `calcnewpos` function to calculate the ball's new position based on its current position, and the vector by which it is moving. I'll explain the physics in a moment. The one other thing to note is the documentation string, which is a little bit longer this time, and explains the basics of the class. These strings are handy not only to yourself and other programmers looking at the code, but also for tools to parse your code and document it. They won't make much of a difference in small programs, but with large ones they're invaluable, so it's a good habit to get into.

4.1.1. Diversion 1: Sprites

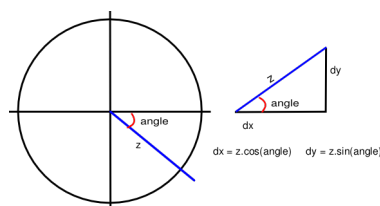
The other reason for creating a class for each object is sprites. Each image you render in your game will be a sprite object, and so to begin with, the class for each object should inherit the `Sprite` class. This is a really nice feature of Python - class inheritance. Now the `Ball` class has all of the functions that come with the `Sprite` class, and any object instances of the `Ball` class will be registered by Pygame as sprites. Whereas with text and the background, which don't move, it's OK to blit the object onto the background, Pygame handles sprite objects in a different manner, which you'll see when we look at the whole program's code.

Basically, you create both a ball object, and a sprite object for that ball, and you then call the ball's update function on the sprite object, thus updating the sprite. Sprites also give you sophisticated ways of determining if two objects have collided. Normally you might just check in the main loop to see if their rectangles overlap, but that would involve a lot of code, which would be a waste because the `Sprite` class provides two functions (`spritecollide` and `groupcollide`) to do this for you.

4.1.2. Diversion 2: Vector physics

Other than the structure of the `Ball` class, the notable thing about this code is the vector physics, used to calculate the ball's movement. With any game involving angular movement, you won't get very far unless you're comfortable with trigonometry, so I'll just introduce the basics you need to know to make sense of the `calcnewpos` function.

To begin with, you'll notice that the ball has an attribute `vector`, which is made up of `angle` and `z`. The angle is measured in radians, and will give you the direction in which the ball is moving. `z` is the speed at which the ball moves. So by using this vector, we can determine the direction and speed of the ball, and therefore how much it will move on the x and y axes:



The diagram above illustrates the basic maths behind vectors. In the left hand diagram, you can see the ball's projected movement represented by the blue line. The length of that line (`z`) represents its speed, and the angle is

the direction in which it will move. The angle for the ball's movement will always be taken from the x axis on the right, and it is measured clockwise from that line, as shown in the diagram.

From the angle and speed of the ball, we can then work out how much it has moved along the x and y axes. We need to do this because Pygame doesn't support vectors itself, and we can only move the ball by moving its rectangle along the two axes. So we need to *resolve* the angle and speed into its movement on the x axis (dx) and on the y axis (dy). This is a simple matter of trigonometry, and can be done with the formulae shown in the diagram.

If you've studied elementary trigonometry before, none of this should be news to you. But just in case you're forgetful, here are some useful formulae to remember, that will help you visualise the angles (I find it easier to visualise angles in degrees than in radians!)

$$\text{radians} = \text{degrees} * \frac{\pi}{180} \quad \text{degrees} = \text{radians} * \frac{180}{\pi}$$

User-controllable objects

5. User-controllable objects

So far you can create a Pygame window, and render a ball that will fly across the screen. The next step is to make some bats which the user can control. This is potentially far more simple than the ball, because it requires no physics (unless your user-controlled object will move in ways more complex than up and down, e.g. a platform character like Mario, in which case you'll need more physics). User-controllable objects are pretty easy to create, thanks to Pygame's event queue system, as you'll see.

5.1. A simple bat class

The principle behind the bat class is similar to that of the ball class. You need an `__init__` function to initialise the ball (so you can create object instances for each bat), an `update` function to perform per-frame changes on the bat before it is blitted the bat to the screen, and the functions that will define what this class will actually do. Here's some sample code:

```
class Bat(pygame.sprite.Sprite):
    """Movable tennis 'bat' with which one hits the ball
    Returns: bat object
    Functions: reinit, update, moveup, movedown
    Attributes: which, speed"""

    def __init__(self, side):
        pygame.sprite.Sprite.__init__(self)
        self.image, self.rect = load_png('bat.png')
        screen = pygame.display.get_surface()
        self.area = screen.get_rect()
        self.side = side
        self.speed = 10
        self.state = "still"
        self.reinit()

    def reinit(self):
        self.state = "still"
        self.movepos = [0,0]
        if self.side == "left":
            self.rect.midleft = self.area.midleft
        elif self.side == "right":
            self.rect.midright = self.area.midright

    def update(self):
        newpos = self.rect.move(self.movepos)
        if self.area.contains(newpos):
            self.rect = newpos
```



```

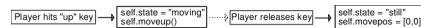
pygame.event.pump()

def moveup(self):
    self.movepos[1] = self.movepos[1] - (self.speed)
    self.state = "moveup"

def movedown(self):
    self.movepos[1] = self.movepos[1] + (self.speed)
    self.state = "movedown"

```

As you can see, this class is very similar to the ball class in its structure. But there are differences in what each function does. First of all, there is a `reinit` function, which is used when a round ends, and the bat needs to be set back in its starting place, with any attributes set back to their necessary values. Next, the way in which the bat is moved is a little more complex than with the ball, because here its movement is simple (up/down), but it relies on the user telling it to move, unlike the ball which just keeps moving in every frame. To make sense of how the ball moves, it is helpful to look at a quick diagram to show the sequence of events:



What happens here is that the person controlling the bat pushes down on the key that moves the bat up. For each iteration of the main game loop (for every frame), if the key is still held down, then the `state` attribute of that bat object will be set to "moving", and the `moveup` function will be called, causing the ball's y position to be reduced by the value of the `speed` attribute (in this example, 10). In other words, so long as the key is held down, the bat will move up the screen by 10 pixels per frame. The `state` attribute isn't used here yet, but it's useful to know if you're dealing with spin, or would like some useful debugging output.

As soon as the player lets go of that key, the second set of boxes is invoked, and the `state` attribute of the bat object will be set back to "still", and the `movepos` attribute will be set back to [0,0], meaning that when the `update` function is called, it won't move the bat any more. So when the player lets go of the key, the bat stops moving. Simple!

5.1.1. Diversion 3: Pygame events

So how do we know when the player is pushing keys down, and then releasing them? With the Pygame event queue system, dummy! It's a really easy system to use and understand, so this shouldn't take long :) You've already seen the event queue in action in the basic Pygame program, where it was used to check if the user was quitting the application. The code for moving the bat is about as simple as that:

```

for event in pygame.event.get():
    if event.type == QUIT:
        return
    elif event.type == KEYDOWN:
        if event.key == K_UP:
            player.moveup()
        if event.key == K_DOWN:
            player.movedown()
    elif event.type == KEYUP:
        if event.key == K_UP or event.key == K_DOWN:
            player.movepos = [0,0]
            player.state = "still"

```

Here assume that you've already created an instance of a bat, and called the object `player`. You can see the familiar layout of the `for` structure, which iterates through each event found in the Pygame event queue, which is retrieved with the `event.get()` function. As the user hits keys, pushes mouse buttons and moves the joystick about, those actions are pumped into the Pygame event queue, and left there until dealt with. So in each iteration of the main game loop, you go through these events, checking if they're ones you want to deal with, and then dealing with them appropriately. The `event.pump()` function that was in the `Bat.update` function is then called in every iteration to pump out old events, and keep the queue current.

First we check if the user is quitting the program, and quit it if they are. Then we check if any keys are being pushed down, and if they are, we check if they're the designated keys for moving the bat up and down. If they are, then we call the appropriate moving function, and set the player state appropriately (though the states `moveup` and `movedown` and changed in the `moveup()` and `movedown()` functions, which makes for neater code, and doesn't

break *encapsulation*, which means that you assign attributes to the object itself, without referring to the name of the instance of that object). Notice here we have three states: still, moveup, and movedown. Again, these come in handy if you want to debug or calculate spin. We also check if any keys have been "let go" (i.e. are no longer being held down), and again if they're the right keys, we stop the bat from moving.

Putting it all together

6. Putting it all together

So far you've learnt all the basics necessary to build a simple game. You should understand how to create Pygame objects, how Pygame displays objects, how it handles events, and how you can use physics to introduce some motion into your game. Now I'll just show how you can take all those chunks of code and put them together into a working game. What we need first is to let the ball hit the sides of the screen, and for the bat to be able to hit the ball, otherwise there's not going to be much game play involved. We do this using Pygame's **collision** methods.

6.1. Let the ball hit sides

The basics principle behind making it bounce off the sides is easy to grasp. You grab the coordinates of the four corners of the ball, and check to see if they correspond with the x or y coordinate of the edge of the screen. So if the top right and top left corners both have a y coordinate of zero, you know that the ball is currently on the top edge of the screen. We do all this in the `update` function, after we've worked out the new position of the ball.

```
if not self.area.contains(newpos):
    tl = not self.area.collidepoint(newpos.topleft)
    tr = not self.area.collidepoint(newpos.topright)
    bl = not self.area.collidepoint(newpos.bottomleft)
    br = not self.area.collidepoint(newpos.bottomright)
    if tr and tl or (br and bl):
        angle = -angle
    if tl and bl:
        self.offcourt(player=2)
    if tr and br:
        self.offcourt(player=1)

self.vector = (angle,z)
```

Here we check to see if the `area` contains the new position of the ball (it always should, so we needn't have an `else` clause, though in other circumstances you might want to consider it). We then check if the coordinates for the four corners are *colliding* with the area's edges, and create objects for each result. If they are, the objects will have a value of 1, or `True`. If they don't, then the value will be `None`, or `False`. We then see if it has hit the top or bottom, and if it has we change the ball's direction. Handily, using radians we can do this by simply reversing its positive/negative value. We also check to see if the ball has gone off the sides, and if it has we call the `offcourt` function. This, in my game, resets the ball, adds 1 point to the score of the player specified when calling the function, and displays the new score.

Finally, we recompile the vector based on the new angle. And that is it. The ball will now merrily bounce off the walls and go offcourt with good grace.

6.2. Let the ball hit bats

Making the ball hit the bats is very similar to making it hit the sides of the screen. We still use the `collide` method, but this time we check to see if the rectangles for the ball and either bat collide. In this code I've also put in some extra code to avoid various glitches. You'll find that you'll have to put all sorts of extra code in to avoid glitches and bugs, so it's good to get used to seeing it.

```
else:
    # Deflate the rectangles so you can't catch a ball behind the bat
    player1.rect.inflate(-3, -3)
    player2.rect.inflate(-3, -3)

    # Do ball and bat collide?
    # Note I put in an odd rule that sets self.hit to 1 when they collide, and unsets it in
```

```

    # iteration. this is to stop odd ball behaviour where it finds a collision *inside* the
    # bat, the ball reverses, and is still inside the bat, so bounces around inside.
    # This way, the ball can always escape and bounce away cleanly
    if self.rect.colliderect(player1.rect) == 1 and not self.hit:
        angle = math.pi - angle
        self.hit = not self.hit
    elif self.rect.colliderect(player2.rect) == 1 and not self.hit:
        angle = math.pi - angle
        self.hit = not self.hit
    elif self.hit:
        self.hit = not self.hit
    self.vector = (angle,z)

```

We start this section with an `else` statement, because this carries on from the previous chunk of code to check if the ball hits the sides. It makes sense that if it doesn't hit the sides, it might hit a bat, so we carry on the conditional statement. The first glitch to fix is to shrink the players' rectangles by 3 pixels in both dimensions, to stop the bat catching a ball that goes behind them (if you imagine you just move the bat so that as the ball travels behind it, the rectangles overlap, and so normally the ball would then have been "hit" - this prevents that).

Next we check if the rectangles collide, with one more glitch fix. Notice that I've commented on these odd bits of code - it's always good to explain bits of code that are abnormal, both for others who look at your code, and so you understand it when you come back to it. The without the fix, the ball might hit a corner of the bat, change direction, and one frame later still find itself inside the bat. Then it would again think it has been hit, and change its direction. This can happen several times, making the ball's motion completely unrealistic. So we have a variable, `self.hit`, which we set to `True` when it has been hit, and `False` one frame later. When we check if the rectangles have collided, we also check if `self.hit` is `True/False`, to stop internal bouncing.

The important code here is pretty easy to understand. All rectangles have a `colliderect` function, into which you feed the rectangle of another object, which returns `True` if the rectangles do overlap, and `False` if not. If they do, we can change the direction by subtracting the current angle from `pi` (again, a handy trick you can do with radians, which will adjust the angle by 90 degrees and send it off in the right direction; you might find at this point that a thorough understanding of radians is in order!). Just to finish the glitch checking, we switch `self.hit` back to `False` if it's the frame after they were hit.

We also then recompile the vector. You would of course want to remove the same line in the previous chunk of code, so that you only do this once after the `if-else` conditional statement. And that's it! The combined code will now allow the ball to hit sides and bats.

6.3. The Finished product

The final product, with all the bits of code thrown together, as well as some other bits of code to glue it all together, will look like this:

```

#
# Tom's Pong
# A simple pong game with realistic physics and AI
# http://www.tomchance.uklinux.net/projects/pong.shtml
#
# Released under the GNU General Public License

VERSION = "0.4"

try:
    import sys
    import random
    import math
    import os
    import getopt
    import pygame
    from socket import *
    from pygame.locals import *
except ImportError, err:

```

```

print "couldn't load module. %s" % (err)
sys.exit(2)

def load_png(name):
    """ Load image and return image object"""
    fullname = os.path.join('data', name)
    try:
        image = pygame.image.load(fullname)
        if image.get_alpha is None:
            image = image.convert()
        else:
            image = image.convert_alpha()
    except pygame.error, message:
        print 'Cannot load image:', fullname
        raise SystemExit, message
    return image, image.get_rect()

class Ball(pygame.sprite.Sprite):
    """A ball that will move across the screen
    Returns: ball object
    Functions: update, calcnewpos
    Attributes: area, vector"""

    def __init__(self, (xy), vector):
        pygame.sprite.Sprite.__init__(self)
        self.image, self.rect = load_png('ball.png')
        screen = pygame.display.get_surface()
        self.area = screen.get_rect()
        self.vector = vector
        self.hit = 0

    def update(self):
        newpos = self.calcnewpos(self.rect, self.vector)
        self.rect = newpos
        (angle, z) = self.vector

        if not self.area.contains(newpos):
            tl = not self.area.collidepoint(newpos.topleft)
            tr = not self.area.collidepoint(newpos.topright)
            bl = not self.area.collidepoint(newpos.bottomleft)
            br = not self.area.collidepoint(newpos.bottomright)
            if tr and tl or (br and bl):
                angle = -angle
            if tl and bl:
                #self.offcourt()
                angle = math.pi - angle
            if tr and br:
                angle = math.pi - angle
                #self.offcourt()
        else:
            # Deflate the rectangles so you can't catch a ball behind the bat
            player1.rect.inflate(-3, -3)
            player2.rect.inflate(-3, -3)

            # Do ball and bat collide?
            # Note I put in an odd rule that sets self.hit to 1 when they collide, and unset
            # iteration. this is to stop odd ball behaviour where it finds a collision *inside
            # bat, the ball reverses, and is still inside the bat, so bounces around inside.
            # This way, the ball can always escape and bounce away cleanly
            if self.rect.colliderect(player1.rect) == 1 and not self.hit:

```

```

        angle = math.pi - angle
        self.hit = not self.hit
    elif self.rect.colliderect(player2.rect) == 1 and not self.hit:
        angle = math.pi - angle
        self.hit = not self.hit
    elif self.hit:
        self.hit = not self.hit
    self.vector = (angle,z)

def calcnewpos(self,rect,vector):
    (angle,z) = vector
    (dx,dy) = (z*math.cos(angle),z*math.sin(angle))
    return rect.move(dx,dy)

class Bat(pygame.sprite.Sprite):
    """Movable tennis 'bat' with which one hits the ball
    Returns: bat object
    Functions: reinit, update, moveup, movedown
    Attributes: which, speed"""

    def __init__(self, side):
        pygame.sprite.Sprite.__init__(self)
        self.image, self.rect = load_png('bat.png')
        screen = pygame.display.get_surface()
        self.area = screen.get_rect()
        self.side = side
        self.speed = 10
        self.state = "still"
        self.reinit()

    def reinit(self):
        self.state = "still"
        self.movepos = [0,0]
        if self.side == "left":
            self.rect.midleft = self.area.midleft
        elif self.side == "right":
            self.rect.midright = self.area.midright

    def update(self):
        newpos = self.rect.move(self.movepos)
        if self.area.contains(newpos):
            self.rect = newpos
        pygame.event.pump()

    def moveup(self):
        self.movepos[1] = self.movepos[1] - (self.speed)
        self.state = "moveup"

    def movedown(self):
        self.movepos[1] = self.movepos[1] + (self.speed)
        self.state = "movedown"

def main():
    # Initialise screen
    pygame.init()
    screen = pygame.display.set_mode((640, 480))
    pygame.display.set_caption('Basic Pong')

    # Fill background

```

```

background = pygame.Surface(screen.get_size())
background = background.convert()
background.fill((0, 0, 0))

# Initialise players
global player1
global player2
player1 = Bat("left")
player2 = Bat("right")

# Initialise ball
speed = 13
rand = ((0.1 * (random.randint(5,8))))
ball = Ball((0,0),(0.47,speed))

# Initialise sprites
playersprites = pygame.sprite.RenderPlain((player1, player2))
ballsprite = pygame.sprite.RenderPlain(ball)

# Blit everything to the screen
screen.blit(background, (0, 0))
pygame.display.flip()

# Initialise clock
clock = pygame.time.Clock()

# Event loop
while 1:
    # Make sure game doesn't run at more than 60 frames per second
    clock.tick(60)

    for event in pygame.event.get():
        if event.type == QUIT:
            return
        elif event.type == KEYDOWN:
            if event.key == K_a:
                player1.moveup()
            if event.key == K_z:
                player1.movedown()
            if event.key == K_UP:
                player2.moveup()
            if event.key == K_DOWN:
                player2.movedown()
        elif event.type == KEYUP:
            if event.key == K_a or event.key == K_z:
                player1.movepos = [0,0]
                player1.state = "still"
            if event.key == K_UP or event.key == K_DOWN:
                player2.movepos = [0,0]
                player2.state = "still"

    screen.blit(background, ball.rect, ball.rect)
    screen.blit(background, player1.rect, player1.rect)
    screen.blit(background, player2.rect, player2.rect)
    ballsprite.update()
    playersprites.update()
    ballsprite.draw(screen)
    playersprites.draw(screen)
    pygame.display.flip()

```

```
if __name__ == '__main__': main()
```

As well as showing you the final product, I'll point you back to TomPong, upon which all of this is based. Download it, have a look at the source code, and you'll see a full implementation of pong using all of the code you've seen in this tutorial, as well as lots of other code I've added in various versions, such as some extra physics for spinning, and various other bug and glitch fixes.

Oh, find TomPong at <http://www.tomchance.uklinux.net/projects/pong.shtml>.

Table of Contents

1. Introduction
 - 1.1. A note on coding styles
2. Revision: Pygame fundamentals
 - 2.1. The basic pygame game
 - 2.2. Basic pygame objects
 - 2.3. Blitting
 - 2.4. The event loop
 - 2.5. Ta-da!
3. Kicking things off
 - 3.1. The first lines, and loading modules
 - 3.2. Resource handling functions
4. Game object classes
 - 4.1. A simple ball class
 - 4.1.1. Diversion 1: Sprites
 - 4.1.2. Diversion 2: Vector physics
5. User-controllable objects
 - 5.1. A simple bat class
 - 5.1.1. Diversion 3: Pygame events
6. Putting it all together
 - 6.1. Let the ball hit sides
 - 6.2. Let the ball hit bats
 - 6.3. The Finished product

1. Introduction

First of all, I will assume you have read the Line By Line Chimp tutorial, which introduces the basics of Python and pygame. Give it a read before reading this tutorial, as I won't bother repeating what that tutorial says (or at least not in as much detail). This tutorial is aimed at those who understand how to make a ridiculously simple little "game", and who would like to make a relatively simple game like Pong. It introduces you to some concepts of game design, some simple mathematics to work out ball physics, and some ways to keep your game easy to maintain and expand.

All the code in this tutorial works toward implementing [TomPong](#), a game I've written. By the end of the tutorial, you should not only have a firmer grasp of pygame, but you should also understand how TomPong works, and how to make your own version.

Now, for a brief recap of the basics of pygame. A common method of organising the code for a game is to divide it into the following six sections:

- **Load modules** which are required in the game. Standard stuff, except that you should remember to import the pygame local names as well as the pygame module itself
- **Resource handling classes**; define some classes to handle your most basic resources, which will be loading images and sounds, as well as connecting and disconnecting to and from networks, loading save game files, and any other resources you might have.
- **Game object classes**; define the classes for your game object. In the pong example, these will be one for the player's bat (which you can initialise multiple times, one for each player in the game), and one for the ball (which can again have multiple instances). If you're going to have a nice in-game menu, it's also a good idea to make a menu class.
- **Any other game functions**; define other necessary functions, such as scoreboards, menu handling, etc. Any code that you could put into the main game logic, but that would make understanding said logic harder, should be put into its own function. So as plotting a scoreboard isn't game logic, it should be moved into a function.
- **Initialise the game**, including the pygame objects themselves, the background, the game objects (initialising instances of the classes) and any other little bits of code you might want to add in.
- **The main loop**, into which you put any input handling (i.e. watching for users hitting keys/mouse buttons), the code for updating the game objects, and finally for updating the screen.

Every game you make will have some or all of those sections, possibly with more of your own. For the purposes of this tutorial, I will write about how TomPong is laid out, and the ideas I write about can be transferred to almost any kind of game you might make. I will also assume that you want to keep all of the code in a single file, but if you're making a reasonably large game, it's often a good idea to source certain sections into module files. Putting the game object classes into a file called `objects.py`, for example, can help you keep game logic separate from game objects. If you have a lot of resource handling code, it can also be handy to put that into `resources.py`. You can then `from objects,resources import *` to import all of the classes and functions.

1.1. A note on coding styles

The first thing to remember when approaching any programming project is to decide on a coding style, and stay consistent. Python solves a lot of the problems because of its strict interpretation of whitespace and indentation, but you can still choose the size of your indentations, whether you put each module import on a new line, how you comment code, etc. You'll see how I do all of this in the code examples; you needn't use my style, but whatever style you adopt, use it all the way through the program code. Also try to document all of your classes, and comment on any bits of code that seem obscure, though don't start commenting the obvious. I've seen plenty of people do the following:

```
player1.score += scoreup           # Add scoreup to player1 score
```

The worst code is poorly laid out, with seemingly random changes in style, and poor documentation. Poor code is not only annoying for other people, but it also makes it difficult for you to maintain.

Pygame Tutorials - Help! How Do I Move An Image?

Help! How Do I Move An Image?

Author: Pete Shinnars

Contact: pete@shinnars.org

Many people new to programming and graphics have a hard time figuring out how to make an image move around the screen. Without understanding all the concepts, it can be very confusing. You're not the first person to be stuck here, I'll do my best to take things step by step. We'll even try to end with methods of keeping your animations efficient.

Note that we won't be teaching you to program with python in this article, just introduce you to some of the basics with pygame.

Just Pixels On The Screen

Pygame has a display Surface. This is basically an image that is visible on the screen, and the image is made up of pixels. The main way you change these pixels is by calling the `blit()` function. This copies the pixels from one image onto another.

This is the first thing to understand. When you blit an image onto the screen, you are simply changing the color of the pixels on the screen. Pixels aren't added or moved, we just change the colors of the pixels already on the screen. These images you blit to the screen are also Surfaces in pygame, but they are in no way connected to the display Surface. When they are blitted to the screen they are copied into the display, but you still have a unique copy of the original.

With this brief description. Perhaps you can already understand what is needed to "move" an image. We don't actually move anything at all. We simply blit the image in a new position. But before we draw the image in the new position, we'll need to "erase" the old one. Otherwise the image will be visible in two places on the screen. By rapidly erasing the image and redrawing it in a new place, we achieve the "illusion" of movement.

Through the rest of this tutorial we will break this process down into simpler steps. Even explaining the best ways to have multiple images moving around the screen. You probably already have questions. Like, how do we "erase" the image before drawing it in a new position? Perhaps you're still totally lost? Well hopefully the rest of this tutorial can straighten things out for you.

Let's Go Back A Step

Perhaps the concept of pixels and images is still a little foreign to you? Well good news, for the next few sections we are going to use code that does everything we want, it just doesn't use pixels. We're going to create a small python list of 6 numbers, and imagine it represents some fantastic graphics we could see on the screen. It might actually be surprising how closely this represents exactly what we'll later be doing with real graphics.

So let's begin by creating our screen list and fill it with a beautiful landscape of 1s and 2s.

```
>>> screen = [1, 1, 2, 2, 2, 1]
>>> print screen
[1, 1, 2, 2, 2, 1]
```

Now we've created our background. It's not going to be very exciting unless we also draw a player on the screen. We'll create a mighty hero that looks like the number 8. Let's stick him near the middle of the map and see what it looks like.

```
>>> screen[3] = 8
>>> print screen
[1, 1, 2, 8, 2, 1]
```

This might have been as far as you've gotten if you jumped right in doing some graphics programming with pygame. You've got some nice looking stuff on the screen, but it cannot move anywhere. Perhaps now that our screen is just a list of numbers, it's easier to see how to move him?

Making The Hero Move

Before we can start moving the character. We need to keep track of some sort of position for him. In the last section when we drew him, we just picked an arbitrary position. Let's do it a little more officially this time.

```
>>> playerpos = 3
>>> screen[playerpos] = 8
>>> print screen
[1, 1, 2, 8, 2, 1]
```

Now it is pretty easy to move him to a new position. We simple change the value of `playerpos`, and draw him on the screen again.

```
>>> playerpos = playerpos - 1
>>> screen[playerpos] = 8
>>> print screen
[1, 1, 8, 8, 2, 1]
```

Whoops. Now we can see two heroes. One in the old position, and one in his new position. This is exactly the reason we need to "erase" the hero in his old position before we draw him in the new position. To erase him, we need to change that value in the list back to what it was before the hero was there. That means we need to keep track of the

values on the screen before the hero replaced them. There's several way you could do this, but the easiest is usually to keep a separate copy of the screen background. This means we need to make some changes to our little game.

Creating A Map

What we want to do is create a separate list we will call our background. We will create the background so it looks like our original screen did, with 1s and 2s. Then we will copy each item from the background to the screen. After that we can finally draw our hero back onto the screen.

```
>>> background = [1, 1, 2, 2, 2, 1]
>>> screen = [0]*6                                #a new blank screen
>>> for i in range(6):
...     screen[i] = background[i]
>>> print screen
[1, 1, 2, 2, 2, 1]
>>> playerpos = 3
>>> screen[playerpos] = 8
>>> print screen
[1, 1, 2, 8, 2, 1]
```

It may seem like a lot of extra work. We're no farther off than we were before the last time we tried to make him move. But this time we have the extra information we need to move him properly.

Making The Hero Move (Take 2)

This time it will be easy to move the hero around. First we will erase the hero from his old position. We do this by copying the correct value from the background onto the screen. Then we will draw the character in his new position on the screen

```
>>> print screen
[1, 1, 2, 8, 2, 1]
>>> screen[playerpos] = background[playerpos]
>>> playerpos = playerpos - 1
>>> screen[playerpos] = 8
>>> print screen
[1, 1, 8, 2, 2, 1]
```

There it is. The hero has moved one space to the left. We can use this same code to move him to the left again.

```
>>> screen[playerpos] = background[playerpos]
>>> playerpos = playerpos - 1
>>> screen[playerpos] = 8
>>> print screen
[1, 8, 2, 2, 2, 1]
```

Excellent! This isn't exactly what you'd call smooth animation. But with a couple small changes, we'll make this work directly with graphics on the screen.

Definition: "blit"

In the next sections we will transform our program from using lists to using real graphics on the screen. When displaying the graphics we will use the term **blit** frequently. If you are new to doing graphics work, you are probably unfamiliar with this common term.

BLIT: Basically, blit means to copy graphics from one image to another. A more formal definition is to copy an array of data to a bitmapped array destination. You can think of blit as just "*assigning*" pixels. Much like setting values in our screen-list above, blitting assigns the color of pixels in our image.

Other graphics libraries will use the word *bitblt*, or just *blt*, but they are talking about the same thing. It is basically copying memory from one place to another. Actually, it is a bit more advanced than straight copying of memory, since it needs to handle things like pixel formats, clipping, and scanline pitches. Advanced blitters can also handle things like transparency and other special effects.

Going From The List To The Screen

To take the code we see in the above to examples and make them work with pygame is very straightforward. We'll pretend we have loaded some pretty graphics and named them "terrain1", "terrain2", and "hero". Where before we assigned numbers to a list, we now blit graphics to the screen. Another big change, instead of using positions as a single index (0 through 5), we now need a two dimensional coordinate. We'll pretend each of the graphics in our game is 10 pixels wide.

```
>>> background = [terrain1, terrain1, terrain2, terrain2, terrain2, terrain1]
>>> screen = create_graphics_screen()
>>> for i in range(6):
...     screen.blit(background[i], (i*10, 0))
>>> playerpos = 3
>>> screen.blit(playerimage, (playerpos*10, 0))
```

Hmm, that code should seem very familiar, and hopefully more importantly; the code above should make a little sense. Hopefully my illustration of setting simple values in a list shows the similarity of setting pixels on the screen (with blit). The only part that's really extra work is converting the player position into coordinates on the screen. For now we just use a crude `(playerpos*10, 0)`, but we can certainly do better than that. Now let's move the player image over a space. This code should have no surprises.

```
>>> screen.blit(background[playerpos], (playerpos*10, 0))
>>> playerpos = playerpos - 1
>>> screen.blit(playerimage, (playerpos*10, 0))
```

There you have it. With this code we've shown how to display a simple background with a hero's image on it. Then we've properly moved that hero one space to the left. So where do we go from here? Well for one the code is still a little awkward. First thing we'll want to do is find a cleaner way to represent the background and player position. Then perhaps a bit of smoother, real animation.

Screen Coordinates

To position an object on the screen, we need to tell the blit() function where to put the image. In pygame we always pass positions as an (X,Y) coordinate. This represents the number of pixels to the right, and the number of pixels down to place the image. The top-left corner of a Surface is coordinate (0, 0). Moving to the right a little would be (10, 0), and then moving down just as much would be (10, 10). When blitting, the position argument represents where the topleft corner of the source should be placed on the destination.

Pygame comes with a convenient container for these coordinates, it is a Rect. The Rect basically represents a rectangular area in these coordinates. It has topleft corner and a size. The Rect comes with a lot of convenient methods which help you move and position them. In our next examples we will represent the positions of our objects with the Rects.

Also know that many functions in pygame expect Rect arguments. All of these functions can also accept a simple tuple of 4 elements (left, top, width, height). You aren't always required to use these Rect objects, but you will mainly want to. Also, the blit() function can accept a Rect as it's position argument, it simply uses the topleft corner of the Rect as the real position.

Changing The Background

In all our previous sections, we've been storing the background as a list of different types of ground. That is a good way to create a tile-based game, but we want smooth scrolling. To make that a little easier, we're going to change the background into a single image that covers the whole screen. This way, when we want to "erase" our objects (before redrawing them) we only need to blit the section of the erased background onto the screen.

By passing an optional third Rect argument to blit, we tell blit to only use that subsection of the source image. You'll see that in use below as we erase the player image.

Also note, now when we finish drawing to the screen, we call `pygame.display.update()` which will show everything we've drawn onto the screen.

Smooth Movement

To make something appear to move smoothly, we only want to move it a couple pixels at a time. Here is the code to make an object move smoothly across the screen. Based on what we already now know, this should look pretty simple.

```
>>> screen = create_screen()
>>> player = load_player_image()
>>> background = load_background_image()
>>> screen.blit(background, (0, 0))           #draw the background
>>> position = player.get_rect()
>>> screen.blit(player, position)             #draw the player
>>> pygame.display.update()                   #and show it all
>>> for x in range(100):                       #animate 100 frames
...     screen.blit(background, position, position) #erase
...     position = position.move(2, 0)           #move player
...     screen.blit(player, position)           #draw new player
...     pygame.display.update()                 #and show it all
...     pygame.time.delay(100)                 #stop the program for 1/10 second
```

There you have it. This is all the code that is needed to smoothly animate an object across the screen. We can even use a pretty background character. Another benefit of doing the background this way, the image for the player can have transparency or cutout sections and it will still draw correctly over the background (a free bonus).

We also throw in a call to `pygame.time.delay()` at the end of our loop above. This slows down our program a little, otherwise it might run so fast you might not see it.

So, What Next?

Well there we have it. Hopefully this article has done everything it promised to do. But, at this point the code really isn't ready for the next best-selling game. How do we easily have multiple moving objects? What exactly are those mysterious functions like `load_player_image()`? We also need a way to get simple user input, and loop for more than 100 frames. We'll take the example we have here, and turn it into an object oriented creation that would make momma proud.

First, The Mystery Functions

Full information on these types of functions can be found in other tutorials and reference. The `pygame.image` module has a `load()` function which will do what we want. The lines to load the images should become this.

```
>>> player = pygame.image.load('player.bmp').convert()
>>> background = pygame.image.load('liquid.bmp').convert()
```

We can see that's pretty simple, the `load` function just takes a filename and returns a new `Surface` with the loaded image. After loading we make a call to the `Surface` method, `convert()`. `Convert` returns us a new `Surface` of the image, but now converted to the same pixel format as our display. Since the images will be the same format at the screen, they will blit very quickly. If we did not convert, the `blit()` function is slower, since it has to convert from one type of pixel to another as it goes.

You may also have noticed that both the `load()` and `convert()` return new `Surfaces`. This means we're really creating two `Surfaces` on each of these lines. In other programming languages, this results in a memory leak (not a good thing). Fortunately Python is smart enough to handle this, and pygame will properly clean up the `Surface` we end up not using.

The other mystery function we saw in the above example was `create_screen()`. In pygame it is simple to create a new window for graphics. The code to create a 640x480 surface is below. By passing no other arguments, pygame will just pick the best color depth and pixel format for us.

```
>>> screen = pygame.display.set_mode((640, 480))
```

Handling Some Input

We desperately need to change the main loop to look for any user input, (like when the user closes the window). We need to add "event handling" to our program. All graphical programs use this Event Based design. The program gets events like "keyboard pressed" or "mouse moved" from the computer. Then the program responds to the different

events. Here's what the code should look like. Instead of looping for 100 frames, we'll keep looping until the user asks us to stop.

```
>>> while 1:
...     for event in pygame.event.get():
...         if event.type in (QUIT, KEYDOWN):
...             sys.exit()
...     move_and_draw_all_game_objects()
```

What this code simply does is, first loop forever, then check if there are any events from the user. We exit the program if the user presses the keyboard or the close button on the window. After we've checked all the events we move and draw our game objects. (We'll also erase them before they move, too)

Moving Multiple Images

Here's the part where we're really going to change things around. Let's say we want 10 different images moving around on the screen. A good way to handle this is to use python's classes. We'll create a class that represents our game object. This object will have a function to move itself, and then we can create as many as we like. The functions to draw and move the object need to work in a way where they only move one frame (or one step) at a time. Here's the python code to create our class.

```
>>> class GameObject:
...     def __init__(self, image, height, speed):
...         self.speed = speed
...         self.image = image
...         self.pos = image.get_rect().move(0, height)
...     def move(self):
...         self.pos = self.pos.move(0, self.speed)
...         if self.pos.right > 600:
...             self.pos.left = 0
```

So we have two functions in our class. The init function constructs our object. It positions the object and sets its speed. The move method moves the object one step. If it's gone too far, it moves the object back to the left.

Putting It All Together

Now with our new object class, we can put together the entire game. Here is what the main function for our program will look like.

```
>>> screen = pygame.display.set_mode((640, 480))
>>> player = pygame.image.load('player.bmp').convert()
>>> background = pygame.image.load('background.bmp').convert()
>>> screen.blit(background, (0, 0))
>>> objects = []
>>> for x in range(10):
...     o = GameObject(player, x*40, x)
...     objects.append(o)
>>> while 1:
...     for event in pygame.event.get():
...         if event.type in (QUIT, KEYDOWN):
...             sys.exit()
...     for o in objects:
...         screen.blit(background, o.pos, o.pos)
...     for o in objects:
...         o.move()
...         screen.blit(o.image, o.pos)
...     pygame.display.update()
...     pygame.time.delay(100)
```

And there it is. This is the code we need to animate 10 objects on the screen. The only point that might need explaining is the two loops we use to clear all the objects and draw all the objects. In order to do things properly, we need to erase all the objects before drawing any of them. In our sample here it may not matter, but when objects are overlapping, using two loops like this becomes important.

You Are On Your Own From Here

So what would be next on your road to learning? Well first playing around with this example a bit. The full running version of this example is available in the pygame examples directory. It is the example named `moveit.py`. Take a look at the code and play with it, run it, learn it.

Things you may want to work on is maybe having more than one type of object. Finding a way to cleanly "delete" objects when you don't want to show them any more. Also updating the `display.update()` call to pass a list of the areas on-screen that have changed.

There are also other tutorials and examples in pygame that cover these issues. So when you're ready to keep learning, keep on reading. :-)

Lastly, you can feel free to come to the pygame mailing list or chatroom with any questions on this stuff. There's always folks on hand who can help you out with this sort of business.

Lastly, have fun, that's what games are for!

Pygame Intro

Python Pygame Introduction

Author: Pete Shinnars

Contact: pete@shinnars.org

This article is an introduction to the [pygame library](#) for [Python programmers](#). The original version appeared in the [Py Zine](#), volume 1 issue 3. This version contains minor revisions, to create an all-around better article. Pygame is a Python extension library that wraps the [SDL](#) library and its helpers.

HISTORY

Pygame started in the summer of 2000. Being a C programmer of many years, I discovered both Python and SDL at about the same time. You are already familiar with Python, which was at version 1.5.2. You may need an introduction to SDL, which is the Simple DirectMedia Layer. Created by Sam Lantinga, SDL is a cross-platform C library for controlling multimedia, comparable to DirectX. It has been used for hundreds of commercial and open source games. I was impressed at how clean and straightforward both projects were and it wasn't long before I realized mixing Python and SDL was an interesting proposal.

I discovered a small project already under-way with exactly the same idea, PySDL. Created by Mark Baker, PySDL was a straightforward implementation of SDL as a Python extension. The interface was cleaner than a generic SWIG wrapping, but I felt it forced a "C style" of code. The sudden death of PySDL prompted me to take on a new project of my own.

I wanted to put together a project that really took advantage of Python. My goal was to make it easy to do the simple things, and straightforward to do the difficult things. Pygame was started in October, 2000. Six months later pygame version 1.0 was released.

TASTE

I find the best way to understand a new library is to jump straight into an example. In the early days of pygame, I created a bouncing ball animation with 7 lines of code. Let's take a look at a friendlier version of that same thing. This should be simple enough to follow along, and a complete breakdown follows.



```
1 import sys, pygame
2 pygame.init()
3
4 size = width, height = 320, 240
5 speed = [2, 2]
6 black = 0, 0, 0
7
```

```

8 screen = pygame.display.set_mode(size)
9
10 ball = pygame.image.load("intro_ball.gif")
11 ballrect = ball.get_rect()
12
13 while 1:
14     for event in pygame.event.get():
15         if event.type == pygame.QUIT: sys.exit()
16
17     ballrect = ballrect.move(speed)
18     if ballrect.left < 0 or ballrect.right > width:
19         speed[0] = -speed[0]
20     if ballrect.top < 0 or ballrect.bottom > height:
21         speed[1] = -speed[1]
22
23     screen.fill(black)
24     screen.blit(ball, ballrect)
25     pygame.display.flip()

```

This is as simple as you can get for a bouncing animation. First we see importing and initializing pygame is nothing noteworthy. The `import pygame` imports the package with all the available pygame modules. The call to `pygame.init()` initializes each of these modules.

On line 8 we create a graphical window with the call to `pygame.display.set_mode()`. Pygame and SDL make this easy by defaulting to the best graphics modes for the graphics hardware. You can override the mode and SDL will compensate for anything the hardware cannot do. Pygame represents images as *Surface* objects. The `display.set_mode()` function creates a new *Surface* object that represents the actual displayed graphics. Any drawing you do to this *Surface* will become visible on the monitor.

At line 10 we load our ball image. Pygame supports a variety of image formats through the *SDL_image* library, including BMP, JPG, PNG, TGA, and GIF. The `pygame.image.load()` function returns us a *Surface* with the ball data. The *Surface* will keep any colorkey or alpha transparency from the file. After loading the ball image we create a variable named `ballrect`. Pygame comes with a convenient utility object type named *Rect*, which represents a rectangular area. Later, in the animation part of the code, we will see what the *Rect* objects can do.

At this point, line 13, our program is initialized and ready to run. Inside an infinite loop we check for user input, move the ball, and then draw the ball. If you are familiar with GUI programming, you have had experience with events and event loops. In pygame this is no different, we check if a *QUIT* event has happened. If so we simply exit the program, pygame will ensure everything is cleanly shutdown.

It is time to update our position for the ball. Lines 17 moves the `ballrect` variable by the current speed. Lines 18 thru 21 reverse the speed if the ball has moved outside the screen. Not exactly Newtonian physics, but it is all we need.

On line 23 we erase the screen by filling it with a black RGB color. If you have never worked with animations this may seem strange. You may be asking "Why do we need to erase anything, why don't we just move the ball on the screen?" That is not quite the way computer animation works. Animation is nothing more than a series of single images, which when displayed in sequence do a very good job of fooling the human eye into seeing motion. The screen is just a single image that the user sees. If we did not take the time to erase the ball from the screen, we would actually see a "trail" of the ball as we continuously draw the ball in its new positions.

On line 24 we draw the ball image onto the screen. Drawing of images is handled by the `Surface.blit()` method. A blit basically means copying pixel colors from one image to another. We pass the blit method a source *Surface* to copy from, and a position to place the source onto the destination.

The last thing we need to do is actually update the visible display. Pygame manages the display with a double buffer. When we are finished drawing we call the `pygame.display.flip()` method. This makes everything we have drawn on the screen *Surface* become visible. This buffering makes sure we only see completely drawn frames on the screen. Without it, the user would see the half completed parts of the screen as they are being created.

That concludes this short introduction to pygame. Pygame also has modules to do things like input handling for the keyboard, mouse, and joystick. It can mix audio and decode streaming music. With the *Surfaces* you can draw simple shapes, rotate and scale the picture, and even manipulate the pixels of an image in realtime as numpy arrays. Pygame also has the ability to act as a cross platform display layer for PyOpenGL. Most of the pygame modules are written in C, few are actually done in Python.

The pygame website has full reference documentation for every pygame function and tutorials for all ranges of users. The pygame source comes with many examples of things like monkey punching and UFO shooting.

PYTHON AND GAMING

"Is Python suitable for gaming?" The answer is, "It depends on the game."

Python is actually quite capable at running games. It will likely even surprise you how much is possible in under 30 milliseconds. Still, it is not hard to reach the ceiling once your game begins to get more complex. Any game running in realtime will be making full use of the computer.



Over the past several years there has been an interesting trend in game development, the move towards higher level languages. Usually a game is split into two major parts. The game engine, which must be as fast as possible, and the game logic, which makes the engine actually do something. It wasn't long ago when the engine of a game was written in assembly, with portions written in C. Nowadays, C has moved to the game engine, while often the game itself is written in higher level scripting languages. Games like Quake3 and Unreal run these scripts as portable bytecode.

In early 2001, developer Rebel Act Studios finished their game, Severance: Blade of Darkness. Using their own custom 3D engine, the rest of the game is written with Python. The game is a bloody action 3rd person perspective fighter. You control medieval warriors into intricate decapitating combination attacks while exploring dungeons and castles. You can download third party add-ons for this game, and find they are nothing more than Python source files.

More recently, Python has been used in a variety of games like Freedom Force, and Humungous' Backyard Sports Series.



Pygame and SDL serve as an excellent C engine for 2D games. Games will still find the largest part of their runtime is spent inside SDL handling the graphics. SDL can take advantage of graphics hardware acceleration. Enabling this can change a game from running around 40 frames per second to over 200 frames per second. When you see your Python game running at 200 frames per second, you realize that Python and games can work together.

It is impressive how well both Python and SDL work on multiple platforms. For example, in May of 2001 I released my own full pygame project, SolarWolf, an arcade style action game. One thing that has surprised me is that one year later there has been no need for any patches, bug fixes, or updates. The game was developed entirely on windows, but runs on Linux, Mac OSX, and many Unixes without any extra work on my end.

Still, there are very clear limitations. The best way to manage hardware accelerated graphics is not always the way to get fastest results from software rendering. Hardware support is not available on all platforms. When a game gets

more complex, it often must commit to one or the other. SDL has some other design limitations, things like full screen scrolling graphics can quickly bring your game down to unplayable speeds. While SDL is not suitable for all types of games, remember companies like Loki have used SDL to run a wide variety of retail quality titles.

Pygame is fairly low-level when it comes to writing games. You'll quickly find yourself needing to wrap common functions into your own game environment. The great thing about this is there is nothing inside pygame to get in your way. Your program is in full control of everything. The side effect of that is you will find yourself borrowing a lot of code to get a more advanced framework put together. You'll need a better understanding of what you are doing.

CLOSING

Developing games is very rewarding, there is something exciting about being able to see and interact with the code you've written. Pygame currently has almost 30 other projects using it. Several of them are ready to play now. You may be surprised to visit the pygame website, and see what other users have been able to do with Python.

One thing that has caught my attention is the amount of people coming to Python for the first time to try game development. I can see why games are a draw for new programmers, but it can be difficult since creating games requires a firmer understanding of the language. I've tried to support this group of users by writing many examples and pygame tutorials for people new to these concepts.

In the end, my advice is to keep it simple. I cannot stress this enough. If you are planning to create your first game, there is a lot to learn. Even a simpler game will challenge your designs, and complex games don't necessarily mean fun games. When you understand Python, you can use pygame to create a simple game in only one or two weeks. From there you'll need a surprising amount of time to add the polish to make that into a full presentable game.

Pygame Modules Overview

cdrom	playback
cursors	load cursor images, includes standard cursors
display	control the display window or screen
draw	draw simple shapes onto a Surface
event	manage events and the event queue
font	create and render TrueType fonts
image	save and load images
joystick	manage joystick devices
key	manage the keyboard
mouse	manage the mouse
sndarray	manipulate sounds with numpy
surfarray	manipulate images with numpy
time	control timing
transform	scale, rotate, and flip images

Pygame Tutorials - Sprite Module Introduction

Sprite Module Introduction

Author: Pete Shinnars

Contact: pete@shinnars.org

Pygame version 1.3 comes with a new module, `pygame.sprite`. This module is written in Python and includes some higher-level classes to manage your game objects. By using this module to its full potential, you can easily manage and draw your game objects. The sprite classes are very optimized, so it's likely your game will run faster with the sprite module than without.

The sprite module is also meant to be very generic. It turns out you can use it with nearly any type of gameplay. All this flexibility comes with a slight penalty, it needs a little understanding to properly use it. The `reference documentation` for the sprite module can keep you running, but you'll probably need a bit more explanation of how to use `pygame.sprite` in your own game.

Several of the pygame examples (like "chimp" and "aliens") have been updated to use the sprite module. You may want to look into those first to see what this sprite module is all about. The chimp module even has it's own line-by-line tutorial, which may help get more an understanding of programming with python and pygame.

Note that this introduction will assume you have a bit of experience programming with python, and are somewhat familiar with the different parts of creating a simple game. In this tutorial the word "reference" is occasionally used. This represents a python variable. Variables in python are references, so you can have several variables all pointing to the same object.

History Lesson

The term "sprite" is a holdover from older computer and game machines. These older boxes were unable to draw and erase normal graphics fast enough for them to work as games. These machines had special hardware to handle game like objects that needed to animate very quickly. These objects were called "sprites" and had special limitations, but could be drawn and updated very fast. They usually existed in special overlay buffers in the video. These days computers have become generally fast enough to handle sprite like objects without dedicated hardware. The term sprite is still used to represent just about anything in a 2D game that is animated.

The Classes

The sprite module comes with two main classes. The first is `Sprite`, which should be used as a base class for all your game objects. This class doesn't really do anything on its own, it just includes several functions to help manage the game object. The other type of class is `Group`. The `Group` class is a container for different `Sprite` objects. There are actually several different types of group classes. Some of the `Groups` can draw all the elements they contain, for example.

This is all there really is to it. We'll start with a description of what each type of class does, and then discuss the proper ways to use these two classes.

The Sprite Class

As mentioned before, the `Sprite` class is designed to be a base class for all your game objects. You cannot really use it on its own, as it only has several methods to help it work with the different `Group` classes. The sprite keeps track of which groups it belongs to. The class constructor (`__init__` method) takes an argument of a `Group` (or list of `Groups`) the `Sprite` instance should belong to. You can also change the `Group` membership for the `Sprite` with the `add()` and `remove()` methods. There is also a `groups()` method, which returns a list of the current groups containing the sprite.

When using the your `Sprite` classes it's best to think of them as "valid" or "alive" when they are belonging to one or more `Groups`. When you remove the instance from all groups pygame will clean up the object. (Unless you have your own references to the instance somewhere else.) The `kill()` method removes the sprite from all groups it belongs to. This will cleanly delete the sprite object. If you've put some little games together, you'll know sometimes cleanly deleting a game object can be tricky. The sprite also comes with an `alive()` method, which returns true if it is still a member of any groups.

The Group Class

The `Group` class is just a simple container. Similar to the sprite, it has an `add()` and `remove()` method which can change which sprites belong to the group. You also can pass a sprite or list of sprites to the constructor (`__init__()` method) to create a `Group` instance that contains some initial sprites.

The `Group` has a few other methods like `empty()` to remove all sprites from the group and `copy()` which will return a copy of the group with all the same members. Also the `has()` method will quickly check if the `Group` contains a sprite or list of sprites.

The other function you will use frequently is the `sprites()` method. This returns an object that can be looped on to access every sprite the group contains. Currently this is just a list of the sprites, but in later version of python this will likely use iterators for better performance.

As a shortcut, the `Group` also has an `update()` method, which will call an `update()` method on every sprite in the group. Passing the same arguments to each one. Usually in a game you need some function that updates the state of a game object. It's very easy to call your own methods using the `Group.sprites()` method, but this is a shortcut that's used enough to be included. Also note that the base `Sprite` class has a "dummy" `update()` method that takes any sort of arguments and does nothing.

Lastly, the `Group` has a couple other methods that allow you to use it with the builtin `len()` function, getting the number of sprites it contains, and the "truth" operator, which allows you to do "if mygroup:" to check if the group has any sprites.

Mixing Them Together

At this point the two classes seem pretty basic. Not doing a lot more than you can do with a simple list and your own class of game objects. But there are some big advantages to using the `Sprite` and `Group` together. A sprite can belong to as many groups as you want. Remember as soon as it belongs to no groups, it will usually be cleared up (unless you have other "non-group" references to that object).

The first big thing is a fast simple way to categorize sprites. For example, say we had a Pacman-like game. We could make separate groups for the different types of objects in the game. Ghosts, Pac, and Pellets. When Pac eats a power pellet, we can change the state for all ghost objects by effecting everything in the Ghost group. This is quicker and simpler than looping through a list of all the game objects and checking which ones are ghosts.

Adding and removing groups and sprites from each other is a very fast operation, quicker than using lists to store everything. Therefore you can very efficiently change group memberships. Groups can be used to work like simple attributes for each game object. Instead of tracking some attribute like "close_to_player" for a bunch of enemy objects, you could add them to a separate group. Then when you need to access all the enemies that are near the player, you already have a list of them, instead of going through a list of all the enemies, checking for the "close_to_player" flag. Later on your game could add multiple players, and instead of adding more "close_to_player2", "close_to_player3" attributes, you can easily add them to different groups or each player.

Another important benefit of using the `Sprites` and `Groups` is that the groups cleanly handle the deleting (or killing) of game objects. In a game where many objects are referencing other objects, sometimes deleting an object can be the hardest part, since it can't go away until it is not referenced by anyone. Say we have an object that is "chasing" another object. The chaser can keep a simple `Group` that references the object (or objects) it is chasing. If the object being chased happens to be destroyed, we don't need to worry about notifying the chaser to stop chasing. The chaser can see for itself that its group is now empty, and perhaps find a new target.

Again, the thing to remember is that adding and removing sprites from groups is a very cheap/fast operation. You may be best off by adding many groups to contain and organize your game objects. Some could even be empty for large portions of the game, there isn't any penalties for managing your game like this.

The Many Group Types

The above examples and reasons to use `Sprites` and `Groups` are only a tip of the iceberg. Another advantage is that the sprite module comes with several different types of `Groups`. These groups all work just like a regular old `Group`, but they also have added functionality (or slightly different functionality). Here's a list of the `Group` classes included with the sprite module.

Group

This is the standard "no frills" group mainly explained above. Most of the other `Groups` are derived from this one, but not all.

GroupSingle

This works exactly like the regular `Group` class, but it only contains the most recently added sprite. Therefore when you add a sprite to this group, it "forgets" about any previous sprites it had. Therefore it always contains only one or zero sprites.

RenderPlain

This is a standard group derived from `Group`. It has a `draw()` method that draws all the sprites it contains to the screen (or any `Surface`). For this to work, it requires all sprites it contains to have a "image" and "rect" attributes. It uses these to know what to blit, and where to blit it.

RenderClear

This is derived from the `RenderPlain` group, and adds a method named `clear()`. This will erase the previous position of all drawn sprites. It uses a background image to fill in the areas where the sprite were. It is smart enough to handle deleted sprites and properly clear them from the screen when the `clear()` method is called.

RenderUpdates

This is the Cadillac of rendering Groups. It is inherited from `RenderClear`, but changes the `draw()` method to also return a list of pygame Rects, which represent all the areas on screen that have been changed.

That is the list of different groups available We'll discuss more about these rendering groups in the next section. There's nothing stopping you from creating your own Group classes as well. They are just python code, so you can inherit from one of these and add/change whatever you want. In the future I hope we can add a couple more Groups to this list. A `GroupMulti` which is like the `GroupSingle`, but can hold up to a given number of sprites (in some sort of circular buffer?). Also a super-render group that can clear the position of the old sprites without needing a background image to do it (by grabbing a copy of the screen before blitting). Who knows really, but in the future we can add more useful classes to this list.

The Rendering Groups

From above we can see there are three different rendering groups. We could probably just get away with the `RenderUpdates` one, but it adds overhead not really needed for something like a scrolling game. So we have a couple tools here, pick the right one for the right job.

For a scrolling type game, where the background completely changes every frame, we obviously don't need to worry about python's update rectangles in the call to `display.update()`. You should definitely go with the `RenderPlain` group here to manage your rendering.

For games where the background is more stationary, you definitely don't want pygame updating the entire screen (since it doesn't need to). This type of game usually involves erasing the old position of each object, then drawing it in a new place for each frame. This way we are only changing what is necessary. Most of the time you will just want to use the `RenderUpdates` class here. Since you will also want to pass this list of changes to the `display.update()` function.

The `RenderUpdates` class also does a good job at minimizing overlapping areas in the list of updated rectangles. If the previous position and current position of an object overlap, it will merge them into a single rectangle. Combine this with the fact that it properly handles deleted objects and this is one powerful Group class. If you've written a game that manages the changed rectangles for the objects in a game, you know this the cause for a lot of messy code in your game. Especially once you start to throw in objects that can be deleted at any time. All this work is reduced down to a `clear()` and `draw()` method with this monster class. Plus with the overlap checking, it is likely faster than if you did it yourself.

Also note that there's nothing stopping you from mixing and matching these render groups in your game. You should definitely use multiple rendering groups when you want to do layering with your sprites. Also if the screen is split into multiple sections, perhaps each section of the screen should use an appropriate render group?

Collision Detection

The sprite module also comes with two very generic collision detection functions. For more complex games, these really won't work for you, but you can easily grab the source code for them, and modify them as needed.

Here's a summary of what they are, and what they do.

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

This checks for collisions between a single sprite and the sprites in a group. It requires a "rect" attribute for all the sprites used. It returns a list of all the sprites that overlap with the first sprite. The "dokill" argument is a boolean argument. If it is true, the function will call the `kill()` method on all the sprites. This means the last reference to each sprite is probably in the returned list. Once the list goes away so do the sprites. A quick example of using this in a loop

```
>>> for bomb in sprite.spritecollide(player, bombs, 1):
...     boom_sound.play()
...     Explosion(bomb, 0)
```

This finds all the sprites in the "bomb" group that collide with the player. Because of the "dokill" argument it deletes all the crashed bombs. For each bomb that did collide, it plays a "boom" sound effect, and creates a new `Explosion` where the bomb was. (Note, the `Explosion` class here knows to add each instance to the appropriate class, so we don't need to store it in a variable, that last line might feel a little "funny" to you python programmers.

`groupcollide(group1, group2, dokill1, dokill2) -> dictionary`

This is similar to the `spritecollide` function, but a little more complex. It checks for collisions for all the sprites in one group, to the sprites in another. There is a `dokill` argument for the sprites in each list. When `dokill1` is true, the colliding sprites in `group1` will be `kill()`'ed. When `dokill2` is true, we get the same results for `group2`. The dictionary it returns works like this; each key in the dictionary is a sprite from `group1` that had a collision. The value for that key is a list of the sprites that it collided with. Perhaps another quick code sample explains it best

```
>>> for alien in sprite.groupcollide.aliens, shots, 1, 1).keys():
...     boom_sound.play()
...     Explosion(alien, 0)
...     kills += 1
```

This code checks for the collisions between player bullets and all the aliens they might intersect. In this case we only loop over the dictionary keys, but we could loop over the `values()` or `items()` if we wanted to do something to the specific shots that collided with aliens. If we did loop over the `values()` we would be looping through lists that contain sprites. The same sprite may even appear more than once in these different loops, since the same "shot" could have collided against multiple "aliens".

Those are the basic collision functions that come with pygame. It should be easy to roll your own that perhaps use something different than the "rect" attribute. Or maybe try to fine-tweak your code a little more by directly effecting the collision object, instead of building a list of the collision? The code in the sprite collision functions is very optimized, but you could speed it up slightly by taking out some functionality you don't need.

Common Problems

Currently there is one main problem that catches new users. When you derive your new sprite class with the `Sprite` base, you **must** call the `Sprite.__init__()` method from your own class `__init__()` method. If you forget to call the `Sprite.__init__()` method, you get a cryptic error, like this

```
AttributeError: 'mysprite' instance has no attribute '_Sprite_g'
```

Extending Your Own Classes (Advanced)

Because of speed concerns, the current `Group` classes try to only do exactly what they need, and not handle a lot of general situations. If you decide you need extra features, you may want to create your own `Group` class.

The `Sprite` and `Group` classes were designed to be extended, so feel free to create your own `Group` classes to do specialized things. The best place to start is probably the actual python source code for the sprite module. Looking at the current `Sprite` groups should be enough example on how to create your own.

For example, here is the source code for a rendering `Group` that calls a `render()` method for each sprite, instead of just blitting an "image" variable from it. Since we want it to also handle updated areas, we will start with a copy of the original `RenderUpdates` group, here is the code:

```
class RenderUpdatesDraw(RenderClear):
    """call sprite.draw(screen) to render sprites"""
    def draw(self, surface):
        dirty = self.lostsprites
        self.lostsprites = []
        for s, r in self.spritedict.items():
            newrect = s.draw(screen) #Here's the big change
            if r is 0:
                dirty.append(newrect)
            else:
                dirty.append(newrect.union(r))
```

```
self.spritedict[s] = newrect
return dirty
```

Following is more information on how you could create your own `Sprite` and `Group` objects from scratch.

The `Sprite` objects only "require" two methods. `"add_internal()"` and `"remove_internal()"`. These are called by the `Group` classes when they are removing a sprite from themselves. The `add_internal()` and `remove_internal()` have a single argument which is a group. Your `Sprite` will need some way to also keep track of the `Groups` it belongs to. You will likely want to try to match the other methods and arguments to the real `Sprite` class, but if you're not going to use those methods, you sure don't need them.

It is almost the same requirements for creating your own `Group`. In fact, if you look at the source you'll see the `GroupSingle` isn't derived from the `Group` class, it just implements the same methods so you can't really tell the difference. Again you need an `"add_internal()"` and `"remove_internal()"` method that the sprites call when they want to belong or remove themselves from the group. The `add_internal()` and `remove_internal()` have a single argument which is a sprite. The only other requirement for the `Group` classes is they have a dummy attribute named `"_spritegroup"`. It doesn't matter what the value is, as long as the attribute is present. The `Sprite` classes can look for this attribute to determine the difference between a "group" and any ordinary python container. (This is important, because several sprite methods can take an argument of a single group, or a sequence of groups. Since they both look similar, this is the most flexible way to "see" the difference.)

You should through the code for the sprite module. While the code is a bit "tuned", it's got enough comments to help you follow along. There's even a `TODO` section in the source if you feel like contributing.

Pygame Tutorials - Surfarray Introduction

Surfarray Introduction

Author: Pete Shinnars

Contact: pete@shinnars.org

Introduction

This tutorial will attempt to introduce users to both NumPy and the pygame surfarray module. To beginners, the code that uses surfarray can be quite intimidating. But actually there are only a few concepts to understand and you will be up and running. Using the surfarray module, it becomes possible to perform pixel level operations from straight python code. The performance can become quite close to the level of doing the code in C.

You may just want to jump down to the *"Examples"* section to get an idea of what is possible with this module, then start at the beginning here to work your way up.

Now I won't try to fool you into thinking everything is very easy. To get more advanced effects by modifying pixel values is very tricky. Just mastering Numeric Python (SciPy's original array package was Numeric, the predecessor of NumPy) takes a lot of learning. In this tutorial I'll be sticking with the basics and using a lot of examples in an attempt to plant seeds of wisdom. After finishing the tutorial you should have a basic handle on how the surfarray works.

Numeric Python

If you do not have the python NumPy package installed, you will need to do that now. You can download the package from the [NumPy Downloads Page](#) To make sure NumPy is working for you, you should get something like this from the interactive python prompt.

```
>>> from numpy import *
>>> a = array((1,2,3,4,5))
>>> a
array([1, 2, 3, 4, 5])
>>> a[2]
3
>>> a*2
array([ 2,  4,  6,  8, 10])

#import numeric
#create an array
#display the array
#index into the array
#new array with twiced values
```

As you can see, the NumPy module gives us a new data type, the *array*. This object holds an array of fixed size, and all values inside are of the same type. The arrays can also be multidimensional, which is how we will use them with images. There's a bit more to it than this, but it is enough to get us started.

If you look at the last command above, you'll see that mathematical operations on NumPy arrays apply to all values in the array. This is called "element-wise operations". These arrays can also be sliced like normal lists. The slicing syntax is the same as used on standard python objects. (*so study up if you need to :]*). Here are some more examples of working with arrays.

```
>>> len(a)                                #get array size
5
>>> a[2:]                                  #elements 2 and up
array([3, 4, 5])
>>> a[:-2]                                #all except last 2
array([1, 2, 3])
>>> a[2:] + a[:-2]                        #add first and last
array([4, 6, 8])
>>> array((1,2,3)) + array((3,4))         #add arrays of wrong sizes
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: operands could not be broadcast together with shapes (3,) (2,)
```

We get an error on the last command, because we try add together two arrays that are different sizes. In order for two arrays to operate with each other, including comparisons and assignment, they must have the same dimensions. It is very important to know that the new arrays created from slicing the original all reference the same values. So changing the values in a slice also changes the original values. It is important how this is done.

```
>>> a                                      #show our starting array
array([1, 2, 3, 4, 5])
>>> aa = a[1:3]                           #slice middle 2 elements
>>> aa                                     #show the slice
array([2, 3])
>>> aa[1] = 13                            #change value in slice
>>> a                                      #show change in original
array([ 1, 2, 13, 4, 5])
>>> aaa = array(a)                        #make copy of array
>>> aaa                                    #show copy
array([ 1, 2, 13, 4, 5])
>>> aaa[1:4] = 0                          #set middle values to 0
>>> aaa                                    #show copy
array([1, 0, 0, 0, 5])
>>> a                                      #show original again
array([ 1, 2, 13, 4, 5])
```

Now we will look at small arrays with two dimensions. Don't be too worried, getting started it is the same as having a two dimensional tuple (*a tuple inside a tuple*). Let's get started with two dimensional arrays.

```
>>> row1 = (1,2,3)                        #create a tuple of vals
>>> row2 = (3,4,5)                        #another tuple
>>> (row1,row2)                            #show as a 2D tuple
((1, 2, 3), (3, 4, 5))
>>> b = array((row1, row2))                #create a 2D array
>>> b                                      #show the array
array([[1, 2, 3],
       [3, 4, 5]])
>>> array(((1,2),(3,4),(5,6)))            #show a new 2D array
array([[1, 2],
       [3, 4],
       [5, 6]])
```

Now with this two dimensional array (*from now on as "2D"*) we can index specific values and do slicing on both dimensions. Simply using a comma to separate the indices allows us to lookup/slice in multiple dimensions. Just

using ":" as an index (*or not supplying enough indices*) gives us all the values in that dimension. Let's see how this works.

```
>>> b                                #show our array from above
array([[1, 2, 3],
       [3, 4, 5]])
>>> b[0,1]                           #index a single value
2
>>> b[1,:]                           #slice second row
array([3, 4, 5])
>>> b[1]                             #slice second row (same as above)
array([3, 4, 5])
>>> b[:,2]                           #slice last column
array([3, 5])
>>> b[:,2]                           #slice into a 2x2 array
array([[1, 2],
       [3, 4]])
```

Ok, stay with me here, this is about as hard as it gets. When using NumPy there is one more feature to slicing. Slicing arrays also allow you to specify a *slice increment*. The syntax for a slice with increment is `start_index : end_index : increment`.

```
>>> c = arange(10)                  #like range, but makes an array
>>> c                               #show the array
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> c[1:6:2]                        #slice odd values from 1 to 6
array([1, 3, 5])
>>> c[4::4]                         #slice every 4th val starting at 4
array([4, 8])
>>> c[8:1:-1]                       #slice 1 to 8, reversed
array([8, 7, 6, 5, 4, 3, 2])
```

Well that is it. There's enough information there to get you started using NumPy with the surfarray module. There's certainly a lot more to NumPy, but this is only an introduction. Besides, we want to get on to the fun stuff, correct?

Import Surfarray

In order to use the surfarray module we need to import it. Since both surfarray and NumPy are optional components for pygame, it is nice to make sure they import correctly before using them. In these examples I'm going to import NumPy into a variable named *N*. This will let you know which functions I'm using are from the NumPy package. (*and is a lot shorter than typing NumPy before each function*)

```
try:
    import numpy as N
    import pygame.surfarray as surfarray
except ImportError:
    raise ImportError, "NumPy and Surfarray are required."
```

Surfarray Introduction

There are two main types of functions in surfarray. One set of functions for creating an array that is a copy of a surface pixel data. The other functions create a referenced copy of the array pixel data, so that changes to the array directly affect the original surface. There are other functions that allow you to access any per-pixel alpha values as arrays along with a few other helpful functions. We will look at these other functions later on.

When working with these surface arrays, there are two ways of representing the pixel values. First, they can be represented as mapped integers. This type of array is a simple 2D array with a single integer representing the surface's mapped color value. This type of array is good for moving parts of an image around. The other type of array uses three RGB values to represent each pixel color. This type of array makes it extremely simple to do types of effects that change the color of each pixel. This type of array is also a little trickier to deal with, since it is essentially a 3D numeric array. Still, once you get your mind into the right mode, it is not much harder than using the normal 2D arrays.

The NumPy module uses a machine's natural number types to represent the data values, so a NumPy array can consist of integers that are 8-bits, 16-bits, and 32-bits. *(the arrays can also use other types like floats and doubles, but for our image manipulation we mainly need to worry about the integer types)*. Because of this limitation of integer sizes, you must take a little extra care that the type of arrays that reference pixel data can be properly mapped to a proper type of data. The functions create these arrays from surfaces are:

`surfarray.pixels2d(surface)`

Creates a 2D array (*integer pixel values*) that reference the original surface data. This will work for all surface formats except 24-bit.

`surfarray.array2d(surface)`

Creates a 2D array (*integer pixel values*) that is copied from any type of surface.

`surfarray.pixels3d(surface)`

Creates a 3D array (*RGB pixel values*) that reference the original surface data. This will only work on 24-bit and 32-bit surfaces that have RGB or BGR formatting.

`surfarray.array3d(surface)`

Creates a 3D array (*RGB pixel values*) that is copied from any type of surface.

Here is a small chart that might better illustrate what types of functions should be used on which surfaces. As you can see, both the arrayXD functions will work with any type of surface.

	32-bit	24-bit	16-bit	8-bit(c-map)
pixel2d	yes		yes	yes
array2d	yes	yes	yes	yes
pixel3d	yes	yes		
array3d	yes	yes	yes	yes

Examples

With this information, we are equipped to start trying things with surface arrays. The following are short little demonstrations that create a NumPy array and display them in pygame. These different tests are found in the `arraydemo.py` example. There is a simple function named `surfdemo_show` that displays an array on the screen.



```
allblack = N.zeros((128, 128))
surfdemo_show(allblack, 'allblack')
```

Our first example creates an all black array. Whenever you need to create a new numeric array of a specific size, it is best to use the `zeros` function. Here we create a 2D array of all zeros and display it.



```
striped = N.zeros((128, 128, 3))
striped[:] = (255, 0, 0)
striped[:, ::3] = (0, 255, 255)
surfdemo_show(striped, 'striped')
```

Here we are dealing with a 3D array. We start by creating an all red image. Then we slice out every third row and assign it to a blue/green color. As you can see, we can treat the 3D arrays almost exactly the same as 2D arrays, just be sure to assign them 3 values instead of a single mapped integer.



```
imgsurface = pygame.image.load('surfarray.png')
rgbarray = surfarray.array3d(imgsurface)
surfdemo_show(rgbarray, 'rgbarray')
```


Here we load an image with the image module, then convert it to a 3D array of integer RGB color elements. An RGB copy of a surface always has the colors arranged as `a[r,c,0]` for the red component, `a[r,c,1]` for the green component, and `a[r,c,2]` for blue. This can then be used without caring how the pixels of the actual surface are configured, unlike a 2D array which is a copy of the `mapped` (raw) surface pixels. We will use this image in the rest of the samples.



```
flipped = rgbarray[:,::-1]
surfdemo_show(flipped, 'flipped')
```

Here we flip the image vertically. All we need to do is take the original image array and slice it using a negative increment.



```
scaledown = rgbarray[::2,::2]
surfdemo_show(scaledown, 'scaledown')
```

Based on the last example, scaling an image down is pretty logical. We just slice out all the pixels using an increment of 2 vertically and horizontally.



```
shape = rgbarray.shape
scaleup = N.zeros((shape[0]*2, shape[1]*2, shape[2]))
scaleup[::2,::2,:] = rgbarray
scaleup[1::2,::2,:] = rgbarray
scaleup[:,1::2] = scaleup[:,::2]
surfdemo_show(scaleup, 'scaleup')
```

Scaling the image up is a little more work, but is similar to the previous scaling down, we do it all with slicing. First we create an array that is double the size of our original. First we copy the original array into every other pixel of the new array. Then we do it again for every other pixel doing the odd columns. At this point we have the image scaled properly going across, but every other row is black, so we simply need to copy each row to the one underneath it. Then we have an image doubled in size.



```
redimg = N.array(rgbarray)
redimg[:,:,1:] = 0
surfdemo_show(redimg, 'redimg')
```

Now we are using 3D arrays to change the colors. Here we set all the values in green and blue to zero. This leaves us with just the red channel.



```
factor = N.array((8,), N.int32)
soften = N.array(rgbarray, N.int32)
soften[1:,:] += rgbarray[:-1,:] * factor
soften[:-1,:] += rgbarray[1:,:] * factor
soften[:,1:] += rgbarray[:, :-1] * factor
soften[:, :-1] += rgbarray[:, 1:] * factor
soften //= 33
surfdemo_show(soften, 'soften')
```

Here we perform a 3x3 convolution filter that will soften our image. It looks like a lot of steps here, but what we are doing is shifting the image 1 pixel in each direction and adding them all together (with some multiplication for

weighting). Then average all the values. It's no Gaussian, but it's fast. One point with NumPy arrays, the precision of arithmetic operations is determined by the array with the largest data type. So if factor was not declared as a 1 element array of type `numpy.int32`, the multiplications would be performed using `numpy.int8`, the 8 bit integer type of each `rgbarray` element. This will cause value truncation. The `soften` array must also be declared to have a larger integer size than `rgbarray` to avoid truncation.



```
src = N.array(rgbarray)
dest = N.zeros(rgbarray.shape)
dest[:] = 20, 50, 100
diff = (dest - src) * 0.50
xfade = src + diff.astype(N.uint)
surfdemo_show(xfade, 'xfade')
```

Lastly, we are cross fading between the original image and a solid bluish image. Not exciting, but the `dest` image could be anything, and changing the 0.50 multiplier will let you choose any step in a linear crossfade between two images.

Hopefully by this point you are starting to see how `surfarray` can be used to perform special effects and transformations that are only possible at the pixel level. At the very least, you can use the `surfarray` to do a lot of `Surface.set_at()` `Surface.get_at()` type operations very quickly. But don't think you are finished yet, there is still much to learn.

Surface Locking

Like the rest of `pygame`, `surfarray` will lock any `Surfaces` it needs to automatically when accessing pixel data. There is one extra thing to be aware of though. When creating the *pixel* arrays, the original surface will be locked during the lifetime of that pixel array. This is important to remember. Be sure to "*del*" the pixel array or let it go out of scope (*ie*, when the function returns, etc).

Also be aware that you really don't want to be doing much (*if any*) direct pixel access on hardware surfaces (`HWSURFACE`). This is because the actual surface data lives on the graphics card, and transferring pixel changes over the PCI/AGP bus is not fast.

Transparency

The `surfarray` module has several methods for accessing a `Surface`'s alpha/colorkey values. None of the alpha functions are affected by overall transparency of a `Surface`, just the pixel alpha values. Here's the list of those functions.

`surfarray.pixels_alpha` (surface)

Creates a 2D array (*integer pixel values*) that references the original surface alpha data. This will only work on 32-bit images with an 8-bit alpha component.

`surfarray.array_alpha` (surface)

Creates a 2D array (*integer pixel values*) that is copied from any type of surface. If the surface has no alpha values, the array will be fully opaque values (255).

`surfarray.array_colorkey` (surface)

Creates a 2D array (*integer pixel values*) that is set to transparent (0) wherever that pixel color matches the `Surface` colorkey.

Other Surfarray Functions

There are only a few other functions available in `surfarray`. You can get a better list with more documentation on the [surfarray reference page](#). There is one very useful function though.

`surfarray.blit_array` (surface, array)

This will transfer any type of 2D or 3D surface array onto a `Surface` of the same dimensions. This `surfarray` blit will generally be faster than assigning an array to a referenced pixel array. Still, it should not be as fast as normal `Surface` blitting, since those are very optimized.

More Advanced NumPy

There's a couple last things you should know about NumPy arrays. When dealing with very large arrays, like the kind that are 640x480 big, there are some extra things you should be careful about. Mainly, while using the operators like `+` and `*` on the arrays makes them easy to use, it is also very expensive on big arrays. These operators must make new temporary copies of the array, that are then usually copied into another array. This can get very time consuming. Fortunately, all the NumPy operators come with special functions that can perform the operation *"in place"*. For example, you would want to replace `screen[:] = screen + brightmap` with the much faster `add(screen, brightmap, screen)`. Anyway, you'll want to read up on the NumPy UFunc documentation for more about this. It is important when dealing with the arrays.

Another thing to be aware of when working with NumPy arrays is the datatype of the array. Some of the arrays (especially the mapped pixel type) often return arrays with an unsigned 8-bit value. These arrays will easily overflow if you are not careful. NumPy will use the same coercion that you find in C programs, so mixing an operation with 8-bit numbers and 32-bit numbers will give a result as 32-bit numbers. You can convert the datatype of an array, but definitely be aware of what types of arrays you have, if NumPy gets in a situation where precision would be ruined, it will raise an exception.

Lastly, be aware that when assigning values into the 3D arrays, they must be between 0 and 255, or you will get some undefined truncating.

Graduation

Well there you have it. My quick primer on Numeric Python and surfarray. Hopefully now you see what is possible, and even if you never use them for yourself, you do not have to be afraid when you see code that does. Look into the vgrade example for more numeric array action. There are also some *"flame"* demos floating around that use surfarray to create a realtime fire effect.

Best of all, try some things on your own. Take it slow at first and build up, I've seen some great things with surfarray already like radial gradients and more. Good Luck.

pygame/examples/chimp.py

```
#!/usr/bin/env python
""" pygame.examples.chimp

This simple example is used for the line-by-line tutorial
that comes with pygame. It is based on a 'popular' web banner.
Note there are comments here, but for the full explanation,
follow along in the tutorial.
"""

# Import Modules
import os
import pygame as pg
from pygame.compat import geterror

if not pg.font:
    print("Warning, fonts disabled")
if not pg.mixer:
    print("Warning, sound disabled")

main_dir = os.path.split(os.path.abspath(__file__))[0]
data_dir = os.path.join(main_dir, "data")

# functions to create our resources
def load_image(name, colorkey=None):
    fullname = os.path.join(data_dir, name)
    try:
```

```

        image = pg.image.load(fullname)
    except pg.error:
        print("Cannot load image:", fullname)
        raise SystemExit(str(geterror()))
    image = image.convert()
    if colorkey is not None:
        if colorkey == -1:
            colorkey = image.get_at((0, 0))
        image.set_colorkey(colorkey, pg.RLEACCEL)
    return image, image.get_rect()

def load_sound(name):
    class NoneSound:
        def play(self):
            pass

    if not pg.mixer or not pg.mixer.get_init():
        return NoneSound()
    fullname = os.path.join(data_dir, name)
    try:
        sound = pg.mixer.Sound(fullname)
    except pg.error:
        print("Cannot load sound: %s" % fullname)
        raise SystemExit(str(geterror()))
    return sound

# classes for our game objects
class Fist(pg.sprite.Sprite):
    """moves a clenched fist on the screen, following the mouse"""

    def __init__(self):
        pg.sprite.Sprite.__init__(self) # call Sprite initializer
        self.image, self.rect = load_image("fist.bmp", -1)
        self.punching = 0

    def update(self):
        """move the fist based on the mouse position"""
        pos = pg.mouse.get_pos()
        self.rect.midtop = pos
        if self.punching:
            self.rect.move_ip(5, 10)

    def punch(self, target):
        """returns true if the fist collides with the target"""
        if not self.punching:
            self.punching = 1
            hitbox = self.rect.inflate(-5, -5)
            return hitbox.colliderect(target.rect)

    def unpunch(self):
        """called to pull the fist back"""
        self.punching = 0

class Chimp(pg.sprite.Sprite):
    """moves a monkey critter across the screen. it can spin the
    monkey when it is punched."""

```

```

def __init__(self):
    pg.sprite.Sprite.__init__(self) # call Sprite initializer
    self.image, self.rect = load_image("chimp.bmp", -1)
    screen = pg.display.get_surface()
    self.area = screen.get_rect()
    self.rect.topleft = 10, 10
    self.move = 9
    self.dizzy = 0

def update(self):
    """walk or spin, depending on the monkeys state"""
    if self.dizzy:
        self._spin()
    else:
        self._walk()

def _walk(self):
    """move the monkey across the screen, and turn at the ends"""
    newpos = self.rect.move((self.move, 0))
    if not self.area.contains(newpos):
        if self.rect.left < self.area.left or self.rect.right > self.area.right:
            self.move = -self.move
            newpos = self.rect.move((self.move, 0))
            self.image = pg.transform.flip(self.image, 1, 0)
        self.rect = newpos

def _spin(self):
    """spin the monkey image"""
    center = self.rect.center
    self.dizzy = self.dizzy + 12
    if self.dizzy >= 360:
        self.dizzy = 0
        self.image = self.original
    else:
        rotate = pg.transform.rotate
        self.image = rotate(self.original, self.dizzy)
    self.rect = self.image.get_rect(center=center)

def punched(self):
    """this will cause the monkey to start spinning"""
    if not self.dizzy:
        self.dizzy = 1
        self.original = self.image

def main():
    """this function is called when the program starts.
    it initializes everything it needs, then runs in
    a loop until the function returns."""
    # Initialize Everything
    pg.init()
    screen = pg.display.set_mode((468, 60))
    pg.display.set_caption("Monkey Fever")
    pg.mouse.set_visible(0)

    # Create The Background
    background = pg.Surface(screen.get_size())
    background = background.convert()
    background.fill((250, 250, 250))

```

```

# Put Text On The Background, Centered
if pg.font:
    font = pg.font.Font(None, 36)
    text = font.render("Pummel The Chimp, And Win $$$", 1, (10, 10, 10))
    textpos = text.get_rect(centerx=background.get_width() / 2)
    background.blit(text, textpos)

# Display The Background
screen.blit(background, (0, 0))
pg.display.flip()

# Prepare Game Objects
clock = pg.time.Clock()
whiff_sound = load_sound("whiff.wav")
punch_sound = load_sound("punch.wav")
chimp = Chimp()
fist = Fist()
allsprites = pg.sprite.RenderPlain((fist, chimp))

# Main Loop
going = True
while going:
    clock.tick(60)

    # Handle Input Events
    for event in pg.event.get():
        if event.type == pg.QUIT:
            going = False
        elif event.type == pg.KEYDOWN and event.key == pg.K_ESCAPE:
            going = False
        elif event.type == pg.MOUSEBUTTONDOWN:
            if fist.punch(chimp):
                punch_sound.play() # punch
                chimp.punched()
            else:
                whiff_sound.play() # miss
        elif event.type == pg.MOUSEBUTTONUP:
            fist.unpunch()

    allsprites.update()

    # Draw Everything
    screen.blit(background, (0, 0))
    allsprites.draw(screen)
    pg.display.flip()

pg.quit()

# Game Over

# this calls the 'main' function when this script is executed
if __name__ == "__main__":
    main()

```

A Newbie Guide to pygame

or **Things I learned by trial and error so you don't have to,**

or **How I learned to stop worrying and love the blit.**

Pygame is a python wrapper for **SDL**, written by Pete Shinnars. What this means is that, using pygame, you can write games or other multimedia applications in Python that will run unaltered on any of SDL's supported platforms (Windows, Unix, Mac, BeOS and others).

Pygame may be easy to learn, but the world of graphics programming can be pretty confusing to the newcomer. I wrote this to try to distill the practical knowledge I've gained over the past year or so of working with pygame, and it's predecessor, PySDL. I've tried to rank these suggestions in order of importance, but how relevant any particular hint is will depend on your own background and the details of your project.

Get comfortable working in Python.

The most important thing is to feel confident using python. Learning something as potentially complicated as graphics programming will be a real chore if you're also unfamiliar with the language you're using. Write a few sizable non-graphical programs in python -- parse some text files, write a guessing game or a journal-entry program or something. Get comfortable with string and list manipulation -- know how to split, slice and combine strings and lists. Know how `import` works -- try writing a program that is spread across several source files. Write your own functions, and practice manipulating numbers and characters; know how to convert between the two. Get to the point where the syntax for using lists and dictionaries is second-nature -- you don't want to have to run to the documentation every time you need to slice a list or sort a set of keys. Resist the temptation to run to a mailing list, `comp.lang.python`, or IRC when you run into trouble. Instead, fire up the interpreter and play with the problem for a few hours. Print out the [Python 2.0 Quick Reference](#) and keep it by your computer.

This may sound incredibly dull, but the confidence you'll gain through your familiarity with python will work wonders when it comes time to write your game. The time you spend making python code second-nature will be nothing compared to the time you'll save when you're writing real code.

Recognize which parts of pygame you really need.

Looking at the jumble of classes at the top of the pygame Documentation index may be confusing. The important thing is to realize that you can do a great deal with only a tiny subset of functions. Many classes you'll probably never use -- in a year, I haven't touched the `Channel`, `Joystick`, `cursors`, `Userrect`, `surfarray` or `version` functions.

Know what a surface is.

The most important part of pygame is the surface. Just think of a surface as a blank piece of paper. You can do a lot of things with a surface -- you can draw lines on it, fill parts of it with color, copy images to and from it, and set or read individual pixel colors on it. A surface can be any size (within reason) and you can have as many of them as you like (again, within reason). One surface is special -- the one you create with `pygame.display.set_mode()`. This 'display surface' represents the screen; whatever you do to it will appear on the user's screen. You can only have one of these -- that's an SDL limitation, not a pygame one.

So how do you create surfaces? As mentioned above, you create the special 'display surface' with `pygame.display.set_mode()`. You can create a surface that contains an image by using `image.load()`, or you can make a surface that contains text with `font.render()`. You can even create a surface that contains nothing at all with `Surface()`.

Most of the surface functions are not critical. Just learn `blit()`, `fill()`, `set_at()` and `get_at()`, and you'll be fine.

Use surface.convert().

When I first read the documentation for `surface.convert()`, I didn't think it was something I had to worry about. 'I only use PNGs, therefore everything I do will be in the same format. So I don't need `convert()`'; It turns out I was very, very wrong.

The 'format' that `convert()` refers to isn't the *file* format (ie PNG, JPEG, GIF), it's what's called the 'pixel format'. This refers to the particular way that a surface records individual colors in a specific pixel. If the surface format isn't

the same as the display format, SDL will have to convert it on-the-fly for every blit -- a fairly time-consuming process. Don't worry too much about the explanation; just note that `convert()` is necessary if you want to get any kind of speed out of your blits.

How do you use `convert`? Just call it after creating a surface with the `image.load()` function. Instead of just doing:

```
surface = pygame.image.load('foo.png')
```

Do:

```
surface = pygame.image.load('foo.png').convert()
```

It's that easy. You just need to call it once per surface, when you load an image off the disk. You'll be pleased with the results; I see about a 6x increase in blitting speed by calling `convert()`.

The only times you don't want to use `convert()` is when you really need to have absolute control over an image's internal format -- say you were writing an image conversion program or something, and you needed to ensure that the output file had the same pixel format as the input file. If you're writing a game, you need speed. Use `convert()`.

Dirty rect animation.

The most common cause of inadequate frame rates in pygame programs results from misunderstanding the `pygame.display.update()` function. With pygame, merely drawing something to the display surface doesn't cause it to appear on the screen -- you need to call `pygame.display.update()`. There are three ways of calling this function:

- `pygame.display.update()` -- This updates the whole window (or the whole screen for fullscreen displays).
- `pygame.display.flip()` -- This does the same thing, and will also do the right thing if you're using double-buffered hardware acceleration, which you're not, so on to...
- `pygame.display.update(a rectangle or some list of rectangles)` -- This updates just the rectangular areas of the screen you specify.

Most people new to graphics programming use the first option -- they update the whole screen every frame. The problem is that this is unacceptably slow for most people. Calling `update()` takes 35 milliseconds on my machine, which doesn't sound like much, until you realize that $1000 / 35 = 28$ frames per second *maximum*. And that's with no game logic, no blits, no input, no AI, nothing. I'm just sitting there updating the screen, and 28 fps is my maximum framerate. Ugh.

The solution is called 'dirty rect animation'. Instead of updating the whole screen every frame, only the parts that changed since the last frame are updated. I do this by keeping track of those rectangles in a list, then calling `update(the_dirty_rectangles)` at the end of the frame. In detail for a moving sprite, I:

- Blit a piece of the background over the sprite's current location, erasing it.
- Append the sprite's current location rectangle to a list called `dirty_rects`.
- Move the sprite.
- Draw the sprite at it's new location.
- Append the sprite's new location to my `dirty_rects` list.
- Call `display.update(dirty_rects)`

The difference in speed is astonishing. Consider that [SolarWolf](#) has dozens of constantly moving sprites updating smoothly, and still has enough time left over to display a parallax starfield in the background, and update that too.

There are two cases where this technique just won't work. The first is where the whole window or screen really is being updated every frame -- think of a smooth-scrolling engine like an overhead real-time strategy game or a side-scroller. So what do you do in this case? Well, the short answer is -- don't write this kind of game in pygame. The long answer is to scroll in steps of several pixels at a time; don't try to make scrolling perfectly smooth. Your player will appreciate a game that scrolls quickly, and won't notice the background jumping along too much.

A final note -- not every game requires high framerates. A strategic wargame could easily get by on just a few updates per second -- in this case, the added complexity of dirty rect animation may not be necessary.

There is NO rule six.***Hardware surfaces are more trouble than they're worth.***

If you've been looking at the various flags you can use with `pygame.display.set_mode()`, you may have thought like this: *Hey, HWSURFACE! Well, I want that -- who doesn't like hardware acceleration. Ooo... DOUBLEBUF; well, that sounds fast, I guess I want that too!* It's not your fault; we've been trained by years of 3-d gaming to believe that hardware acceleration is good, and software rendering is slow.

Unfortunately, hardware rendering comes with a long list of drawbacks:

- It only works on some platforms. Windows machines can usually get hardware surfaces if you ask for them. Most other platforms can't. Linux, for example, may be able to provide a hardware surface if X4 is installed, if DGA2 is working properly, and if the moons are aligned correctly. If a hardware surface is unavailable, SDL will silently give you a software surface instead.
- It only works fullscreen.
- It complicates per-pixel access. If you have a hardware surface, you need to Lock the surface before writing or reading individual pixel values on it. If you don't, Bad Things Happen. Then you need to quickly Unlock the surface again, before the OS gets all confused and starts to panic. Most of this process is automated for you in pygame, but it's something else to take into account.
- You lose the mouse pointer. If you specify HWSURFACE (and actually get it), your pointer will usually just vanish (or worse, hang around in a half-there, half-not flickery state). You'll need to create a sprite to act as a manual mouse pointer, and you'll need to worry about pointer acceleration and sensitivity. What a pain.
- It might be slower anyway. Many drivers are not accelerated for the types of drawing that we do, and since everything has to be blitted across the video bus (unless you can cram your source surface into video memory as well), it might end up being slower than software access anyway.

Hardware rendering has its place. It works pretty reliably under Windows, so if you're not interested in cross-platform performance, it may provide you with a substantial speed increase. However, it comes at a cost -- increased headaches and complexity. It's best to stick with good old reliable SWSURFACE until you're sure you know what you're doing.

Don't get distracted by side issues.

Sometimes, new game programmers spend too much time worrying about issues that aren't really critical to their game's success. The desire to get secondary issues 'right' is understandable, but early in the process of creating a game, you cannot even know what the important questions are, let alone what answers you should choose. The result can be a lot of needless prevarication.

For example, consider the question of how to organize your graphics files. Should each frame have its own graphics file, or each sprite? Perhaps all the graphics should be zipped up into one archive? A great deal of time has been wasted on a lot of projects, asking these questions on mailing lists, debating the answers, profiling, etc, etc. This is a secondary issue; any time spent discussing it should have been spent coding the actual game.

The insight here is that it is far better to have a 'pretty good' solution that was actually implemented, than a perfect solution that you never got around to writing.

Rects are your friends.

Pete Shinnars' wrapper may have cool alpha effects and fast blitting speeds, but I have to admit my favorite part of pygame is the lowly `Rect` class. A rect is simply a rectangle -- defined only by the position of its top left corner, its width, and its height. Many pygame functions take rects as arguments, and they also take 'rectstyles', a sequence that has the same values as a rect. So if I need a rectangle that defines the area between 10, 20 and 40, 50, I can do any of the following:

```
rect = pygame.Rect(10, 20, 30, 30)
rect = pygame.Rect((10, 20, 30, 30))
rect = pygame.Rect((10, 20), (30, 30))
rect = (10, 20, 30, 30)
rect = ((10, 20, 30, 30))
```

If you use any of the first three versions, however, you get access to Rect's utility functions. These include functions to move, shrink and inflate rects, find the union of two rects, and a variety of collision-detection functions.

For example, suppose I'd like to get a list of all the sprites that contain a point (x, y) -- maybe the player clicked there, or maybe that's the current location of a bullet. It's simple if each sprite has a `.rect` member -- I just do:

```
sprites_clicked = [sprite for sprite in all_my_sprites_list if sprite.rect.collidepoint(x, y)]
```

Rects have no other relation to surfaces or graphics functions, other than the fact that you can use them as arguments. You can also use them in places that have nothing to do with graphics, but still need to be defined as rectangles. Every project I discover a few new places to use rects where I never thought I'd need them.

Don't bother with pixel-perfect collision detection.

So you've got your sprites moving around, and you need to know whether or not they're bumping into one another. It's tempting to write something like the following:

- Check to see if the rects are in collision. If they aren't, ignore them.
- For each pixel in the overlapping area, see if the corresponding pixels from both sprites are opaque. If so, there's a collision.

There are other ways to do this, with ANDing sprite masks and so on, but any way you do it in pygame, it's probably going to be too slow. For most games, it's probably better just to do 'sub-rect collision' -- create a rect for each sprite that's a little smaller than the actual image, and use that for collisions instead. It will be much faster, and in most cases the player won't notice the imprecision.

Managing the event subsystem.

Pygame's event system is kind of tricky. There are actually two different ways to find out what an input device (keyboard, mouse or joystick) is doing.

The first is by directly checking the state of the device. You do this by calling, say, `pygame.mouse.get_pos()` or `pygame.key.get_pressed()`. This will tell you the state of that device *at the moment you call the function*.

The second method uses the SDL event queue. This queue is a list of events -- events are added to the list as they're detected, and they're deleted from the queue as they're read off.

There are advantages and disadvantages to each system. State-checking (system 1) gives you precision -- you know exactly when a given input was made -- if `mouse.get_pressed([0])` is 1, that means that the left mouse button is down *right at this moment*. The event queue merely reports that the mouse was down at some time in the past; if you check the queue fairly often, that can be ok, but if you're delayed from checking it by other code, input latency can grow. Another advantage of the state-checking system is that it detects "chording" easily; that is, several states at the same time. If you want to know whether the `t` and `f` keys are down at the same time, just check:

```
if (key.get_pressed[K_t] and key.get_pressed[K_f]):
    print "Yup!"
```

In the queue system, however, each keypress arrives in the queue as a completely separate event, so you'd need to remember that the `t` key was down, and hadn't come up yet, while checking for the `f` key. A little more complicated.

The state system has one great weakness, however. It only reports what the state of the device is at the moment it's called; if the user hits a mouse button then releases it just before a call to `mouse.get_pressed()`, the mouse button will return 0 -- `get_pressed()` missed the mouse button press completely. The two events, `MOUSEBUTTONDOWN` and `MOUSEBUTTONUP`, will still be sitting in the event queue, however, waiting to be retrieved and processed.

The lesson is: choose the system that meets your requirements. If you don't have much going on in your loop -- say you're just sitting in a `while 1` loop, waiting for input, use `get_pressed()` or another state function; the latency will be lower. On the other hand, if every keypress is crucial, but latency isn't as important -- say your user is typing something in an editbox, use the event queue. Some keypresses may be slightly late, but at least you'll get them all.

A note about `event.poll()` vs. `wait()` -- `poll()` may seem better, since it doesn't block your program from doing anything while it's waiting for input -- `wait()` suspends the program until an event is received. However, `poll()` will consume 100% of available CPU time while it runs, and it will fill the event queue with `NOEVENTS`. Use `set_blocked()` to select just those event types you're interested in -- your queue will be much more manageable.

Colorkey vs. Alpha.

There's a lot of confusion around these two techniques, and much of it comes from the terminology used.

'Colorkey blitting' involves telling pygame that all pixels of a certain color in a certain image are transparent instead of whatever color they happen to be. These transparent pixels are not blitted when the rest of the image is blitted, and so don't obscure the background. This is how we make sprites that aren't rectangular in shape. Simply call `surface.set_colorkey(color)`, where `color` is an RGB tuple -- say (0,0,0). This would make every pixel in the source image transparent instead of black.

'Alpha' is different, and it comes in two flavors. 'Image alpha' applies to the whole image, and is probably what you want. Properly known as 'translucency', alpha causes each pixel in the source image to be only *partially* opaque. For example, if you set a surface's alpha to 192 and then blitted it onto a background, 3/4 of each pixel's color would come from the source image, and 1/4 from the background. Alpha is measured from 255 to 0, where 0 is completely transparent, and 255 is completely opaque. Note that colorkey and alpha blitting can be combined -- this produces an image that is fully transparent in some spots, and semi-transparent in others.

'Per-pixel alpha' is the other flavor of alpha, and it's more complicated. Basically, each pixel in the source image has its own alpha value, from 0 to 255. Each pixel, therefore, can have a different opacity when blitted onto a background. This type of alpha can't be mixed with colorkey blitting, and it overrides per-image alpha. Per-pixel alpha is rarely used in games, and to use it you have to save your source image in a graphic editor with a special *alpha channel*. It's complicated -- don't use it yet.

Do things the pythony way.

A final note (this isn't the least important one; it just comes at the end). Pygame is a pretty lightweight wrapper around SDL, which is in turn a pretty lightweight wrapper around your native OS graphics calls. Chances are pretty good that if your code is still slow, and you've done the things I've mentioned above, then the problem lies in the way you're addressing your data in python. Certain idioms are just going to be slow in python no matter what you do. Luckily, python is a very clear language -- if a piece of code looks awkward or unwieldy, chances are its speed can be improved, too. Read over [Python Performance Tips](#) for some great advice on how you can improve the speed of your code. That said, premature optimisation is the root of all evil; if it's just not fast enough, don't torture the code trying to make it faster. Some things are just not meant to be :)

There you go. Now you know practically everything I know about using pygame. Now, go write that game!

David Clark is an avid pygame user and the editor of the Pygame Code Repository, a showcase for community-submitted python game code. He is also the author of Twitch, an entirely average pygame arcade game.

Revision: Pygame fundamentals

2. Revision: Pygame fundamentals

2.1. The basic Pygame game

For the sake of revision, and to ensure that you are familiar with the basic structure of a Pygame program, I'll briefly run through a basic Pygame program, which will display no more than a window with some text in it, that should, by the end, look something like this (though of course the window decoration will probably be different on your system):



The full code for this example looks like this:

```
#!/usr/bin/python

import pygame
from pygame.locals import *

def main():
    # Initialise screen
```

```

pygame.init()
screen = pygame.display.set_mode((150, 50))
pygame.display.set_caption('Basic Pygame program')

# Fill background
background = pygame.Surface(screen.get_size())
background = background.convert()
background.fill((250, 250, 250))

# Display some text
font = pygame.font.Font(None, 36)
text = font.render("Hello There", 1, (10, 10, 10))
textpos = text.get_rect()
textpos.centerx = background.get_rect().centerx
background.blit(text, textpos)

# Blit everything to the screen
screen.blit(background, (0, 0))
pygame.display.flip()

# Event loop
while 1:
    for event in pygame.event.get():
        if event.type == QUIT:
            return

    screen.blit(background, (0, 0))
    pygame.display.flip()

if __name__ == '__main__': main()

```

2.2. Basic Pygame objects

As you can see, the code consists of three main objects: the screen, the background, and the text. Each of these objects is created by first calling an instance of an in-built Pygame object, and then modifying it to fit our needs. The screen is a slightly special case, because we still modify the display through Pygame calls, rather than calling methods belonging to the screen object. But for all other Pygame objects, we first create the object as a copy of a Pygame object, giving it some attributes, and build our game objects from them.

With the background, we first create a Pygame Surface object, and make it the size of the screen. We then perform the `convert()` operation to convert the Surface to a single pixel format. This is more obviously necessary when we have several images and surfaces, all of different pixel formats, which makes rendering them quite slow. By converting all the surfaces, we can drastically speed up rendering times. Finally, we fill the background surface with white (255, 255, 255). These values are *RGB* (Red Green Blue), and can be worked out from any good paint program.

With the text, we require more than one object. First, we create a font object, which defines which font to use, and the size of the font. Then we create a text object, by using the `render` method that belongs to our font object, supplying three arguments: the text to be rendered, whether or not it should be anti-aliased (1=yes, 0=no), and the color of the text (again in RGB format). Next we create a third text object, which gets the rectangle for the text. The easiest way to understand this is to imagine drawing a rectangle that will surround all of the text; you can then use this rectangle to get/set the position of the text on the screen. So in this example we get the rectangle, set its `centerx` attribute to be the `centerx` attribute of the background (so the text's center will be the same as the background's center, i.e. the text will be centered on the screen on the x axis). We could also set the y coordinate, but it's not any different so I left the text at the top of the screen. As the screen is small anyway, it didn't seem necessary.

2.3. Blitting

Now we have created our game objects, we need to actually render them. If we didn't and we ran the program, we'd just see a blank window, and the objects would remain invisible. The term used for rendering objects is *blitting*, which is where you copy the pixels belonging to said object onto the destination object. So to render the background object, you blit it onto the screen. In this example, to make things simple, we blit the text onto the background (so the background will now have a copy of the text on it), and then blit the background onto the screen.

Blitting is one of the slowest operations in any game, so you need to be careful not to blit too much onto the screen in every frame. If you have a background image, and a ball flying around the screen, then you could blit the background and then the ball in every frame, which would cover up the ball's previous position and render the new ball, but this would be pretty slow. A better solution is to blit the background onto the area that the ball previously occupied, which can be found by the ball's previous rectangle, and then blitting the ball, so that you are only blitting two small areas.

2.4. The event loop

Once you've set the game up, you need to put it into a loop so that it will continuously run until the user signals that he/she wants to exit. So you start an open `while` loop, and then for each iteration of the loop, which will be each frame of the game, update the game. The first thing is to check for any Pygame events, which will be the user hitting the keyboard, clicking a mouse button, moving a joystick, resizing the window, or trying to close it. In this case, we simply want to watch out for user trying to quit the game by closing the window, in which case the game should `return`, which will end the `while` loop. Then we simply need to re-blit the background, and flip (update) the display to have everything drawn. OK, as nothing moves or happens in this example, we don't strictly speaking need to re-blit the background in every iteration, but I put it in because when things are moving around on the screen, you will need to do all your blitting here.

2.5. Ta-da!

And that's it - your most basic Pygame game! All games will take a form similar to this, but with lots more code for the actual game functions themselves, which are more to do your with programming, and less guided in structure by the workings of Pygame. This is what this tutorial is really about, and will now go onto.

Kicking things off

3. Kicking things off

The first sections of code are relatively simple, and, once written, can usually be reused in every game you consequently make. They will do all of the boring, generic tasks like loading modules, loading images, opening networking connections, playing music, and so on. They will also include some simple but effective error handling, and any customisation you wish to provide on top of functions provided by modules like `sys` and `pygame`.

3.1. The first lines, and loading modules

First off, you need to start off your game and load up your modules. It's always a good idea to set a few things straight at the top of the main source file, such as the name of the file, what it contains, the license it is under, and any other helpful info you might want to give those who will be looking at it. Then you can load modules, with some error checking so that Python doesn't print out a nasty traceback, which non-programmers won't understand. The code is fairly simple, so I won't bother explaining any of it:

```
#!/usr/bin/env python
#
# Tom's Pong
# A simple pong game with realistic physics and AI
# http://www.tomchance.uklinux.net/projects/pong.shtml
#
# Released under the GNU General Public License

VERSION = "0.4"

try:
    import sys
    import random
```

```

import math
import os
import getopt
import pygame
from socket import *
from pygame.locals import *
except ImportError, err:
    print "couldn't load module. %s" % (err)
    sys.exit(2)

```

3.2. Resource handling functions

In the Line By Line Chimp example, the first code to be written was for loading images and sounds. As these were totally independent of any game logic or game objects, they were written as separate functions, and were written first so that later code could make use of them. I generally put all my code of this nature first, in their own, classless functions; these will, generally speaking, be resource handling functions. You can of course create classes for these, so that you can group them together, and maybe have an object with which you can control all of your resources. As with any good programming environment, it's up to you to develop your own best practice and style.

It's always a good idea to write your own resource handling functions, because although Pygame has methods for opening images and sounds, and other modules will have their methods of opening other resources, those methods can take up more than one line, they can require consistent modification by yourself, and they often don't provide satisfactory error handling. Writing resource handling functions gives you sophisticated, reusable code, and gives you more control over your resources. Take this example of an image loading function:

```

def load_png(name):
    """ Load image and return image object"""
    fullname = os.path.join('data', name)
    try:
        image = pygame.image.load(fullname)
        if image.get_alpha() is None:
            image = image.convert()
        else:
            image = image.convert_alpha()
    except pygame.error, message:
        print 'Cannot load image:', fullname
        raise SystemExit, message
    return image, image.get_rect()

```

Here we make a more sophisticated image loading function than the one provided by `pygame.image.load()`. Note that the first line of the function is a documentation string describing what the function does, and what object(s) it returns. The function assumes that all of your images are in a directory called data, and so it takes the filename and creates the full pathname, for example data/ball.png, using the `os` module to ensure cross-platform compatibility. Then it tries to load the image, and convert any alpha regions so you can achieve transparency, and it returns a more human-readable error if there's a problem. Finally it returns the image object, and its `rect`.

You can make similar functions for loading any other resources, such as loading sounds. You can also make resource handling classes, to give you more flexibility with more complex resources. For example, you could make a music class, with an `__init__` function that loads the sound (perhaps borrowing from a `load_sound()` function), a function to pause the music, and a function to restart. Another handy resource handling class is for network connections. Functions to open sockets, pass data with suitable security and error checking, close sockets, finger addresses, and other network tasks, can make writing a game with network capabilities relatively painless.

Remember the chief task of these functions/classes is to ensure that by the time you get around to writing game object classes, and the main loop, there's almost nothing left to do. Class inheritance can make these basic classes especially handy. Don't go overboard though; functions which will only be used by one class should be written as part of that class, not as a global function.

Game object classes

4. Game object classes

Once you've loaded your modules, and written your resource handling functions, you'll want to get on to writing some game objects. The way this is done is fairly simple, though it can seem complex at first. You write a class for each type of object in the game, and then create an instance of those classes for the objects. You can then use those classes' methods to manipulate the objects, giving objects some motion and interactive capabilities. So your game, in pseudo-code, will look like this:

```
#!/usr/bin/python

# [load modules here]

# [resource handling functions here]

class Ball:
    # [ball functions (methods) here]
    # [e.g. a function to calculate new position]
    # [and a function to check if it hits the side]

def main:
    # [initiate game environment here]

    # [create new object as instance of ball class]
    ball = Ball()

    while 1:
        # [check for user input]

        # [call ball's update function]
        ball.update()
```

This is, of course, a very simple example, and you'd need to put in all the code, instead of those little bracketed comments. But you should get the basic idea. You create a class, into which you put all the functions for a ball, including `__init__`, which would create all the ball's attributes, and `update`, which would move the ball to its new position, before blitting it onto the screen in this position.

You can then create more classes for all of your other game objects, and then create instances of them so that you can handle them easily in the `main` function and the main program loop. Contrast this with initiating the ball in the `main` function, and then having lots of classless functions to manipulate a set ball object, and you'll hopefully see why using classes is an advantage: It allows you to put all of the code for each object in one place; it makes using objects easier; it makes adding new objects, and manipulating them, more flexible. Rather than adding more code for each new ball object, you could simply create new instances of the `Ball` class for each new ball object. Magic!

4.1. A simple ball class

Here is a simple class with the functions necessary for creating a ball object that will, if the `update` function is called in the main loop, move across the screen:

```
class Ball(pygame.sprite.Sprite):
    """A ball that will move across the screen
    Returns: ball object
    Functions: update, calcnewpos
    Attributes: area, vector"""

    def __init__(self, vector):
        pygame.sprite.Sprite.__init__(self)
        self.image, self.rect = load_png('ball.png')
        screen = pygame.display.get_surface()
        self.area = screen.get_rect()
        self.vector = vector

    def update(self):
        newpos = self.calcnewpos(self.rect, self.vector)
        self.rect = newpos
```

```
def calcnewpos(self,rect,vector):
    (angle,z) = vector
    (dx,dy) = (z*math.cos(angle),z*math.sin(angle))
    return rect.move(dx,dy)
```

Here we have the `Ball` class, with an `__init__` function that sets the ball up, an `update` function that changes the ball's rectangle to be in the new position, and a `calcnewpos` function to calculate the ball's new position based on its current position, and the vector by which it is moving. I'll explain the physics in a moment. The one other thing to note is the documentation string, which is a little bit longer this time, and explains the basics of the class. These strings are handy not only to yourself and other programmers looking at the code, but also for tools to parse your code and document it. They won't make much of a difference in small programs, but with large ones they're invaluable, so it's a good habit to get into.

4.1.1. Diversion 1: Sprites

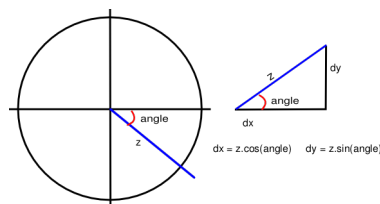
The other reason for creating a class for each object is sprites. Each image you render in your game will be a sprite object, and so to begin with, the class for each object should inherit the `Sprite` class. This is a really nice feature of Python - class inheritance. Now the `Ball` class has all of the functions that come with the `Sprite` class, and any object instances of the `Ball` class will be registered by Pygame as sprites. Whereas with text and the background, which don't move, it's OK to blit the object onto the background, Pygame handles sprite objects in a different manner, which you'll see when we look at the whole program's code.

Basically, you create both a ball object, and a sprite object for that ball, and you then call the ball's update function on the sprite object, thus updating the sprite. Sprites also give you sophisticated ways of determining if two objects have collided. Normally you might just check in the main loop to see if their rectangles overlap, but that would involve a lot of code, which would be a waste because the `Sprite` class provides two functions (`spritecollide` and `groupcollide`) to do this for you.

4.1.2. Diversion 2: Vector physics

Other than the structure of the `Ball` class, the notable thing about this code is the vector physics, used to calculate the ball's movement. With any game involving angular movement, you won't get very far unless you're comfortable with trigonometry, so I'll just introduce the basics you need to know to make sense of the `calcnewpos` function.

To begin with, you'll notice that the ball has an attribute `vector`, which is made up of `angle` and `z`. The angle is measured in radians, and will give you the direction in which the ball is moving. `z` is the speed at which the ball moves. So by using this vector, we can determine the direction and speed of the ball, and therefore how much it will move on the x and y axes:



The diagram above illustrates the basic maths behind vectors. In the left hand diagram, you can see the ball's projected movement represented by the blue line. The length of that line (`z`) represents its speed, and the angle is the direction in which it will move. The angle for the ball's movement will always be taken from the x axis on the right, and it is measured clockwise from that line, as shown in the diagram.

From the angle and speed of the ball, we can then work out how much it has moved along the x and y axes. We need to do this because Pygame doesn't support vectors itself, and we can only move the ball by moving its rectangle along the two axes. So we need to *resolve* the angle and speed into its movement on the x axis (`dx`) and on the y axis (`dy`). This is a simple matter of trigonometry, and can be done with the formulae shown in the diagram.

If you've studied elementary trigonometry before, none of this should be news to you. But just in case you're forgetful, here are some useful formulae to remember, that will help you visualise the angles (I find it easier to visualise angles in degrees than in radians!)

$$\text{radians} = \text{degrees} * \frac{\pi}{180} \quad \text{degrees} = \text{radians} * \frac{180}{\pi}$$

User-controllable objects

5. User-controllable objects

So far you can create a Pygame window, and render a ball that will fly across the screen. The next step is to make some bats which the user can control. This is potentially far more simple than the ball, because it requires no physics (unless your user-controlled object will move in ways more complex than up and down, e.g. a platform character like Mario, in which case you'll need more physics). User-controllable objects are pretty easy to create, thanks to Pygame's event queue system, as you'll see.

5.1. A simple bat class

The principle behind the bat class is similar to that of the ball class. You need an `__init__` function to initialise the bat (so you can create object instances for each bat), an `update` function to perform per-frame changes on the bat before it is blitted to the screen, and the functions that will define what this class will actually do. Here's some sample code:

```
class Bat(pygame.sprite.Sprite):
    """Movable tennis 'bat' with which one hits the ball
    Returns: bat object
    Functions: reinit, update, moveup, movedown
    Attributes: which, speed"""

    def __init__(self, side):
        pygame.sprite.Sprite.__init__(self)
        self.image, self.rect = load_png('bat.png')
        screen = pygame.display.get_surface()
        self.area = screen.get_rect()
        self.side = side
        self.speed = 10
        self.state = "still"
        self.reinit()

    def reinit(self):
        self.state = "still"
        self.movepos = [0,0]
        if self.side == "left":
            self.rect.midleft = self.area.midleft
        elif self.side == "right":
            self.rect.midright = self.area.midright

    def update(self):
        newpos = self.rect.move(self.movepos)
        if self.area.contains(newpos):
            self.rect = newpos
        pygame.event.pump()

    def moveup(self):
        self.movepos[1] = self.movepos[1] - (self.speed)
        self.state = "moveup"

    def movedown(self):
        self.movepos[1] = self.movepos[1] + (self.speed)
        self.state = "movedown"
```

Putting it all together

As you can see, this class is very similar to the ball class in its structure. But there are differences in what each function does. First of all, there is a `reinit` function, which is used when a round ends, and the bat needs to be set back in its starting place, with any attributes set back to their necessary values. Next, the way in which the bat is moved is a little more complex than with the ball, because here its movement is simple (up/down), but it relies on the user telling it to move, unlike the ball which just keeps moving in every frame. To make sense of how the ball moves, it is helpful to look at a quick diagram to show the sequence of events:



What happens here is that the person controlling the bat pushes down on the key that moves the bat up. For each iteration of the main game loop (for every frame), if the key is still held down, then the `state` attribute of that bat object will be set to "moving", and the `moveup` function will be called, causing the ball's y position to be reduced by the value of the `speed` attribute (in this example, 10). In other words, so long as the key is held down, the bat will move up the screen by 10 pixels per frame. The `state` attribute isn't used here yet, but it's useful to know if you're dealing with spin, or would like some useful debugging output.

As soon as the player lets go of that key, the second set of boxes is invoked, and the `state` attribute of the bat object will be set back to "still", and the `movepos` attribute will be set back to [0,0], meaning that when the `update` function is called, it won't move the bat any more. So when the player lets go of the key, the bat stops moving. Simple!

5.1.1. Diversion 3: Pygame events

So how do we know when the player is pushing keys down, and then releasing them? With the Pygame event queue system, dummy! It's a really easy system to use and understand, so this shouldn't take long :) You've already seen the event queue in action in the basic Pygame program, where it was used to check if the user was quitting the application. The code for moving the bat is about as simple as that:

```
for event in pygame.event.get():
    if event.type == QUIT:
        return
    elif event.type == KEYDOWN:
        if event.key == K_UP:
            player.moveup()
        if event.key == K_DOWN:
            player.movedown()
    elif event.type == KEYUP:
        if event.key == K_UP or event.key == K_DOWN:
            player.movepos = [0,0]
            player.state = "still"
```

Here assume that you've already created an instance of a bat, and called the object `player`. You can see the familiar layout of the `for` structure, which iterates through each event found in the Pygame event queue, which is retrieved with the `event.get()` function. As the user hits keys, pushes mouse buttons and moves the joystick about, those actions are pumped into the Pygame event queue, and left there until dealt with. So in each iteration of the main game loop, you go through these events, checking if they're ones you want to deal with, and then dealing with them appropriately. The `event.pump()` function that was in the `Bat.update` function is then called in every iteration to pump out old events, and keep the queue current.

First we check if the user is quitting the program, and quit it if they are. Then we check if any keys are being pushed down, and if they are, we check if they're the designated keys for moving the bat up and down. If they are, then we call the appropriate moving function, and set the player state appropriately (though the states `moveup` and `movedown` and changed in the `moveup()` and `movedown()` functions, which makes for neater code, and doesn't break *encapsulation*, which means that you assign attributes to the object itself, without referring to the name of the instance of that object). Notice here we have three states: still, moveup, and movedown. Again, these come in handy if you want to debug or calculate spin. We also check if any keys have been "let go" (i.e. are no longer being held down), and again if they're the right keys, we stop the bat from moving.

Putting it all together

6. Putting it all together

So far you've learnt all the basics necessary to build a simple game. You should understand how to create Pygame objects, how Pygame displays objects, how it handles events, and how you can use physics to introduce some motion into your game. Now I'll just show how you can take all those chunks of code and put them together into a working game. What we need first is to let the ball hit the sides of the screen, and for the bat to be able to hit the ball, otherwise there's not going to be much game play involved. We do this using Pygame's **collision** methods.

6.1. Let the ball hit sides

The basic principle behind making it bounce off the sides is easy to grasp. You grab the coordinates of the four corners of the ball, and check to see if they correspond with the x or y coordinate of the edge of the screen. So if the top right and top left corners both have a y coordinate of zero, you know that the ball is currently on the top edge of the screen. We do all this in the `update` function, after we've worked out the new position of the ball.

```
if not self.area.contains(newpos):
    tl = not self.area.collidepoint(newpos.topleft)
    tr = not self.area.collidepoint(newpos.topright)
    bl = not self.area.collidepoint(newpos.bottomleft)
    br = not self.area.collidepoint(newpos.bottomright)
    if tr and tl or (br and bl):
        angle = -angle
    if tl and bl:
        self.offcourt(player=2)
    if tr and br:
        self.offcourt(player=1)

self.vector = (angle,z)
```

Here we check to see if the `area` contains the new position of the ball (it always should, so we needn't have an `else` clause, though in other circumstances you might want to consider it). We then check if the coordinates for the four corners are *colliding* with the area's edges, and create objects for each result. If they are, the objects will have a value of 1, or `True`. If they don't, then the value will be `None`, or `False`. We then see if it has hit the top or bottom, and if it has we change the ball's direction. Handily, using radians we can do this by simply reversing its positive/negative value. We also check to see if the ball has gone off the sides, and if it has we call the `offcourt` function. This, in my game, resets the ball, adds 1 point to the score of the player specified when calling the function, and displays the new score.

Finally, we recompile the vector based on the new angle. And that is it. The ball will now merrily bounce off the walls and go offcourt with good grace.

6.2. Let the ball hit bats

Making the ball hit the bats is very similar to making it hit the sides of the screen. We still use the `collide` method, but this time we check to see if the rectangles for the ball and either bat collide. In this code I've also put in some extra code to avoid various glitches. You'll find that you'll have to put all sorts of extra code in to avoid glitches and bugs, so it's good to get used to seeing it.

```
else:
    # Deflate the rectangles so you can't catch a ball behind the bat
    player1.rect.inflate(-3, -3)
    player2.rect.inflate(-3, -3)

    # Do ball and bat collide?
    # Note I put in an odd rule that sets self.hit to 1 when they collide, and unsets it in
    # iteration. this is to stop odd ball behaviour where it finds a collision *inside* the
    # bat, the ball reverses, and is still inside the bat, so bounces around inside.
    # This way, the ball can always escape and bounce away cleanly
    if self.rect.colliderect(player1.rect) == 1 and not self.hit:
        angle = math.pi - angle
        self.hit = not self.hit
    elif self.rect.colliderect(player2.rect) == 1 and not self.hit:
        angle = math.pi - angle
        self.hit = not self.hit
```

```

    elif self.hit:
        self.hit = not self.hit
self.vector = (angle,z)

```

We start this section with an `else` statement, because this carries on from the previous chunk of code to check if the ball hits the sides. It makes sense that if it doesn't hit the sides, it might hit a bat, so we carry on the conditional statement. The first glitch to fix is to shrink the players' rectangles by 3 pixels in both dimensions, to stop the bat catching a ball that goes behind them (if you imagine you just move the bat so that as the ball travels behind it, the rectangles overlap, and so normally the ball would then have been "hit" - this prevents that).

Next we check if the rectangles collide, with one more glitch fix. Notice that I've commented on these odd bits of code - it's always good to explain bits of code that are abnormal, both for others who look at your code, and so you understand it when you come back to it. The without the fix, the ball might hit a corner of the bat, change direction, and one frame later still find itself inside the bat. Then it would again think it has been hit, and change its direction. This can happen several times, making the ball's motion completely unrealistic. So we have a variable, `self.hit`, which we set to `True` when it has been hit, and `False` one frame later. When we check if the rectangles have collided, we also check if `self.hit` is `True/False`, to stop internal bouncing.

The important code here is pretty easy to understand. All rectangles have a `collidirect` function, into which you feed the rectangle of another object, which returns `True` if the rectangles do overlap, and `False` if not. If they do, we can change the direction by subtracting the current angle from `pi` (again, a handy trick you can do with radians, which will adjust the angle by 90 degrees and send it off in the right direction; you might find at this point that a thorough understanding of radians is in order!). Just to finish the glitch checking, we switch `self.hit` back to `False` if it's the frame after they were hit.

We also then recompile the vector. You would of course want to remove the same line in the previous chunk of code, so that you only do this once after the `if-else` conditional statement. And that's it! The combined code will now allow the ball to hit sides and bats.

6.3. The Finished product

The final product, with all the bits of code thrown together, as well as some other bits of code to glue it all together, will look like this:

```

#
# Tom's Pong
# A simple pong game with realistic physics and AI
# http://www.tomchance.uklinux.net/projects/pong.shtml
#
# Released under the GNU General Public License

VERSION = "0.4"

try:
    import sys
    import random
    import math
    import os
    import getopt
    import pygame
    from socket import *
    from pygame.locals import *
except ImportError, err:
    print "couldn't load module. %s" % (err)
    sys.exit(2)

def load_png(name):
    """ Load image and return image object"""
    fullname = os.path.join('data', name)
    try:
        image = pygame.image.load(fullname)
        if image.get_alpha is None:

```

```

        image = image.convert()
    else:
        image = image.convert_alpha()
except pygame.error, message:
    print 'Cannot load image:', fullname
    raise SystemExit, message
return image, image.get_rect()

class Ball(pygame.sprite.Sprite):
    """A ball that will move across the screen
    Returns: ball object
    Functions: update, calcnewpos
    Attributes: area, vector"""

    def __init__(self, (xy), vector):
        pygame.sprite.Sprite.__init__(self)
        self.image, self.rect = load_png('ball.png')
        screen = pygame.display.get_surface()
        self.area = screen.get_rect()
        self.vector = vector
        self.hit = 0

    def update(self):
        newpos = self.calcnewpos(self.rect, self.vector)
        self.rect = newpos
        (angle, z) = self.vector

        if not self.area.contains(newpos):
            tl = not self.area.collidepoint(newpos.topleft)
            tr = not self.area.collidepoint(newpos.topright)
            bl = not self.area.collidepoint(newpos.bottomleft)
            br = not self.area.collidepoint(newpos.bottomright)
            if tr and tl or (br and bl):
                angle = -angle
            if tl and bl:
                #self.offcourt()
                angle = math.pi - angle
            if tr and br:
                angle = math.pi - angle
                #self.offcourt()
        else:
            # Deflate the rectangles so you can't catch a ball behind the bat
            player1.rect.inflate(-3, -3)
            player2.rect.inflate(-3, -3)

            # Do ball and bat collide?
            # Note I put in an odd rule that sets self.hit to 1 when they collide, and unset
            # iteration. this is to stop odd ball behaviour where it finds a collision *inside
            # bat, the ball reverses, and is still inside the bat, so bounces around inside.
            # This way, the ball can always escape and bounce away cleanly
            if self.rect.colliderect(player1.rect) == 1 and not self.hit:
                angle = math.pi - angle
                self.hit = not self.hit
            elif self.rect.colliderect(player2.rect) == 1 and not self.hit:
                angle = math.pi - angle
                self.hit = not self.hit
            elif self.hit:
                self.hit = not self.hit
        self.vector = (angle, z)

```

```

def calcnewpos(self, rect, vector):
    (angle, z) = vector
    (dx, dy) = (z*math.cos(angle), z*math.sin(angle))
    return rect.move(dx, dy)

class Bat(pygame.sprite.Sprite):
    """Movable tennis 'bat' with which one hits the ball
    Returns: bat object
    Functions: reinit, update, moveup, movedown
    Attributes: which, speed"""

    def __init__(self, side):
        pygame.sprite.Sprite.__init__(self)
        self.image, self.rect = load_png('bat.png')
        screen = pygame.display.get_surface()
        self.area = screen.get_rect()
        self.side = side
        self.speed = 10
        self.state = "still"
        self.reinit()

    def reinit(self):
        self.state = "still"
        self.movepos = [0, 0]
        if self.side == "left":
            self.rect.midleft = self.area.midleft
        elif self.side == "right":
            self.rect.midright = self.area.midright

    def update(self):
        newpos = self.rect.move(self.movepos)
        if self.area.contains(newpos):
            self.rect = newpos
        pygame.event.pump()

    def moveup(self):
        self.movepos[1] = self.movepos[1] - (self.speed)
        self.state = "moveup"

    def movedown(self):
        self.movepos[1] = self.movepos[1] + (self.speed)
        self.state = "movedown"

def main():
    # Initialise screen
    pygame.init()
    screen = pygame.display.set_mode((640, 480))
    pygame.display.set_caption('Basic Pong')

    # Fill background
    background = pygame.Surface(screen.get_size())
    background = background.convert()
    background.fill((0, 0, 0))

    # Initialise players
    global player1
    global player2
    player1 = Bat("left")
    player2 = Bat("right")

```

```

# Initialise ball
speed = 13
rand = ((0.1 * (random.randint(5,8))))
ball = Ball((0,0),(0.47,speed))

# Initialise sprites
playersprites = pygame.sprite.RenderPlain((player1, player2))
ballsprite = pygame.sprite.RenderPlain(ball)

# Blit everything to the screen
screen.blit(background, (0, 0))
pygame.display.flip()

# Initialise clock
clock = pygame.time.Clock()

# Event loop
while 1:
    # Make sure game doesn't run at more than 60 frames per second
    clock.tick(60)

    for event in pygame.event.get():
        if event.type == QUIT:
            return
        elif event.type == KEYDOWN:
            if event.key == K_a:
                player1.moveup()
            if event.key == K_z:
                player1.movedown()
            if event.key == K_UP:
                player2.moveup()
            if event.key == K_DOWN:
                player2.movedown()
        elif event.type == KEYUP:
            if event.key == K_a or event.key == K_z:
                player1.movepos = [0,0]
                player1.state = "still"
            if event.key == K_UP or event.key == K_DOWN:
                player2.movepos = [0,0]
                player2.state = "still"

    screen.blit(background, ball.rect, ball.rect)
    screen.blit(background, player1.rect, player1.rect)
    screen.blit(background, player2.rect, player2.rect)
    ballsprite.update()
    playersprites.update()
    ballsprite.draw(screen)
    playersprites.draw(screen)
    pygame.display.flip()

if __name__ == '__main__': main()

```

As well as showing you the final product, I'll point you back to TomPong, upon which all of this is based. Download it, have a look at the source code, and you'll see a full implementation of pong using all of the code you've seen in this tutorial, as well as lots of other code I've added in various versions, such as some extra physics for spinning, and various other bug and glitch fixes.

Oh, find TomPong at <http://www.tomchance.uklinux.net/projects/pong.shtml>.

pygame C API

Slots and `c_api` - Making functions and data available from other modules

One example is `pg_RGBAFromObj` where the implementation is defined in `base.c`, and also exported in `base.c` (and `_pygame.h`).

`base.c` has this exposing the `pg_RGBAFromObj` function to the `c_api` structure:

```
c_api[12] = pg_RGBAFromObj;
```

Then in `src_c/include/_pygame.h` there is an

```
#define pg_RGBAFromObj.
```

Also in `_pygame.h`, it needs to define the number of slots the base module uses. This is `PYGAMEAPI_BASE_NUMSLOTS`. So if you were adding another function, you need to increment this `PYGAMEAPI_BASE_NUMSLOTS` number.

Then to use the `pg_RGBAFromObj` in other files,

1. include the "pygame.h" file,
2. they have to make sure base is imported with:

```
import_pygame_base();
```

Examples that use `pg_RGBAFromObj` are: `_freetype.c`, `color.c`, `gfxdraw.c`, and `surface.c`.

High level API exported by `pygame.base`

`src_c/base.c`

This extension module defines general purpose routines for starting and stopping SDL as well as various conversion routines uses elsewhere in pygame.

C header: `src_c/include/pygame.h`

```
PyObject* pgExc_SDLError
```

This is `pygame.error`, the exception type used to raise SDL errors.

```
int pgVideo_AutoInit ()
```

Initialize the SDL video subsystem. Return 1 on success, 0 for an SDL error, -1 if a Python exception is raised. It is safe to call this function more than once.

```
void pgVideo_AutoQuit ()
```

Close the SDL video subsystem. It is safe to call this function more than once.

```
void pg_RegisterQuit (void (*f)(void))
```

Register function `f` as a callback on Pygame termination. Multiple functions can be registered. Functions are called in the order they were registered.

```
int pg_IntFromObj (PyObject *obj, int *val)
```

Convert number like object `obj` to C int and place in argument `val`. Return 1 on success, else 0. No Python exceptions are raised.

```
int pg_IntFromObjIndex (PyObject *obj, int index, int *val)
```

Convert number like object at position `i` in sequence `obj` to C int and place in argument `val`. Return 1 on success, else raise a Python exception and return 0.

```
int pg_TwoIntsFromObj (PyObject *obj, int *val1, int *val2)
```

Convert the two number like objects in length 2 sequence `obj` to C int and place in arguments `val1` and `val2` respectively. Return 1 on success, else raise a Python exception and return 0.

```
int pg_FloatFromObj (PyObject *obj, float *val)
```

Convert number like object `obj` to C float and place in argument `val`. Returns 1 on success, 0 on failure. No Python exceptions are raised.

```
int pg_FloatFromObjIndex (PyObject *obj, int index, float *val)
```


Convert number like object at position *i* in sequence *obj* to C float and place in argument *val*. Return 1 on success, else raise a Python exception and return 0.

`int pg_TwoFloatsFromObj(PyObject *obj, float *val1, float *val2)`

Convert the two number like objects in length 2 sequence *obj* to C float and place in arguments *val1* and *val2* respectively. Return 1 on success, else raise a Python exception and return 0.

`int pg_UintFromObj(PyObject *obj, Uint32 *val)`

Convert number like object *obj* to unsigned 32 bit integer and place in argument *val*. Return 1 on success, else 0. No Python exceptions are raised.

`int pg_UintFromObjIndex(PyObject *obj, int _index, Uint32 *val)`

Convert number like object at position *i* in sequence *obj* to unsigned 32 bit integer and place in argument *val*. Return 1 on success, else raise a Python exception and return 0.

`int pg_RGBAFromObj(PyObject *obj, Uint8 *RGBA)`

Convert the color represented by object *obj* into a red, green, blue, alpha length 4 C array *RGBA*. The object must be a length 3 or 4 sequence of numbers having values between 0 and 255 inclusive. For a length 3 sequence an alpha value of 255 is assumed. Return 1 on success, 0 otherwise. No Python exceptions are raised.

pg_buffer

`Py_buffer view`

A standard buffer description

`PyObject* consumer`

The object holding the buffer

`pybuffer_releaseproc release_buffer`

A buffer release callback.

`PyObject *pgExc_BufferError`

Python exception type raised for any pg_buffer related errors.

`PyObject* pgBuffer_AsArrayInterface(Py_buffer *view_p)`

Return a Python array interface object representation of buffer *view_p*. On failure raise a Python exception and return *NULL*.

`PyObject* pgBuffer_AsArrayStruct(Py_buffer *view_p)`

Return a Python array struct object representation of buffer *view_p*. On failure raise a Python exception and return *NULL*.

`int pgObject_GetBuffer(PyObject *obj, pg_buffer *pg_view_p, int flags)`

Request a buffer for object *obj*. Argument *flags* are PyBUF options. Return the buffer description in *pg_view_p*. An object may support the Python buffer interface, the NumPy array interface, or the NumPy array struct interface. Return 0 on success, raise a Python exception and return -1 on failure.

`void pgBuffer_Release(Pg_buffer *pg_view_p)`

Release the Pygame *pg_view_p* buffer.

`int pgDict_AsBuffer(Pg_buffer *pg_view_p, PyObject *dict, int flags)`

Write the array interface dictionary buffer description *dict* into a Pygame buffer description struct *pg_view_p*. The *flags* PyBUF options describe the view type requested. Return 0 on success, or raise a Python exception and return -1 on failure.

`void import_pygame_base()`

Import the pygame.base module C API into an extension module. On failure raise a Python exception.

`SDL_Window* pg_GetDefaultWindow(void)`

Return the Pygame default SDL window created by a `pygame.display.set_mode()` call, or *NULL*. Availability: SDL 2.

`void pg_SetDefaultWindow(SDL_Window *win)`

Replace the Pygame default window with *win*. The previous window, if any, is destroyed. Argument *win* may be *NULL*. This function is called by `pygame.display.set_mode()`. Availability: SDL 2.

`pgSurfaceObject* pg_GetDefaultWindowSurface(void)`

Return a borrowed reference to the Pygame default window display surface, or *NULL* if no default window is open. Availability: SDL 2.

```
void pg_SetDefaultWindowSurface (pgSurfaceObject *screen)
```

Replace the Pygame default display surface with object *screen*. The previous surface object, if any, is invalidated. Argument *screen* may be *NULL*. This function is called by `pygame.display.set_mode()`.
Availability: SDL 2.

Class BufferProxy API exported by `pygame.bufferproxy`

`src_c/bufferproxy.c`

This extension module defines Python type `pygame.BufferProxy`.

Header file: `src_c/include/pygame_bufferproxy.h`

```
PyTypeObject *pgBufproxy_Type
```

The pygame buffer proxy object type `pygame.BufferProxy`.

```
int pgBufproxy_Check (PyObject *x)
```

Return true if Python object *x* is a `pygame.BufferProxy` instance, false otherwise. This will return false on `pygame.BufferProxy` subclass instances as well.

```
PyObject* pgBufproxy_New (PyObject *obj, getbufferproc get_buffer)
```

Return a new `pygame.BufferProxy` instance. Argument *obj* is the Python object that has its data exposed. It may be *NULL*. Argument *get_buffer* is the `pg_buffer` get callback. It must not be *NULL*. On failure raise a Python error and return *NULL*.

```
PyObject* pgBufproxy_GetParent (PyObject *obj)
```

Return the Python object wrapped by buffer proxy *obj*. Argument *obj* must not be *NULL*. On failure, raise a Python error and return *NULL*.

```
int pgBufproxy_Trip (PyObject *obj)
```

Cause the buffer proxy object *obj* to create a `pg_buffer` view of its parent. Argument *obj* must not be *NULL*. Return 0 on success, otherwise raise a Python error and return -1.

API exported by `pygame.cdrom`

`src_c/cdrom.c`

The `pygame.cdrom` extension module. Only available for SDL 1.

Header file: `src_c/include/pygame.h`

```
pgCDOBJECT
```

The `pygame.cdrom.CD` instance C struct.

```
PyTypeObject pgCD_Type
```

The `pygame.cdrom.CD` Python type.

```
PyObject* pgCD_New (int id)
```

Return a new `pygame.cdrom.CD` instance for CD drive *id*. On error raise a Python exception and return *NULL*.

```
int pgCD_Check (PyObject *x)
```

Return true if *x* is a `pygame.cdrom.CD` instance. Will return false for a subclass of `CD`. This is a macro. No check is made that *x* is not *NULL*.

```
int pgCD_AsID (PyObject *x)
```

Return the CD identifier associated with the `pygame.cdrom.CD` instance *x*. This is a macro. No check is made that *x* is a `pygame.cdrom.CD` instance or is not *NULL*.

Class Color API exported by `pygame.color`

`src_c/color.c`

This extension module defines the Python type `pygame.Color`.

Header file: `src_c/include/pygame.h`

PyObject* **pgColor_Type**

The Pygame color object type `pygame.Color`.

int **pgColor_Check** (PyObject *obj)

Return true if *obj* is an instance of type `pgColor_Type`, but not a `pgColor_Type` subclass instance. This macro does not check if *obj* is not NULL or indeed a Python type.

PyObject* **pgColor_New** (Uint8 rgba[])

Return a new `pygame.Color` instance for the the four element array *rgba*. On failure, raise a Python exception and return NULL.

PyObject* **pgColor_NewLength** (Uint8 rgba[], Uint8 length)

Return a new `pygame.Color` instance having *length* elements, with element values taken from the first *length* elements of array *rgba*. Argument *length* must be between 1 and 4 inclusive. On failure, raise a Python exception and return NULL.

API exported by `pygame.display`

`src_c/display.c`

This is the `pygame.display` extension module.

Header file: `src_c/include/pygame.h`

pgVidInfoObject

A pygame object that wraps an `SDL_VideoInfo` struct. The object returned by `pygame.display.Info()`.

PyObject* **pgVidInfo_Type**

The `pgVidInfoObject` object Python type.

`SDL_VideoInfo` **pgVidInfo_AsVidInfo** (PyObject *obj)

Return the `SDL_VideoInfo` field of *obj*, a `pgVidInfo_Type` instance. This macro does not check that *obj* is not NULL or an actual `pgVidInfoObject` object.

PyObject* **pgVidInfo_New** (`SDL_VideoInfo` *i)

Return a new `pgVidInfoObject` object for the `SDL_VideoInfo` *i*. On failure, raise a Python exception and return NULL.

int **pgVidInfo_Check** (PyObject *x)

Return true if *x* is a `pgVidInfo_Type` instance

Will return false if *x* is a subclass of `pgVidInfo_Type`. This macro does not check that *x* is not NULL.

API exported by `pygame.event`

`src_c/event.c`

The extsion module `pygame.event`.

Header file: `src_c/include/pygame.h`

pgEventObject

The `pygame.event.EventType` object C struct.

int **type**

The event type code.

pgEvent_Type

The pygame event object type `pygame.event.EventType`.

int **pgEvent_Check** (PyObject *x)

Return true if *x* is a pygame event instance

Will return false if *x* is a subclass of event. This is a macro. No check is made that *x* is not NULL.

PyObject* **pgEvent_New** (`SDL_Event` *event)

Return a new pygame event instance for the `SDL event`. If *event* is NULL then create an empty event object. On failure raise a Python exception and return NULL.

`PyObject* pgEvent_New2(int type, PyObject *dict)`

Return a new pygame event instance of SDL `type` and with attribute dictionary `dict`. If `dict` is `NULL` an empty attribute dictionary is created. On failure raise a Python exception and return `NULL`.

`int pgEvent_FillUserEvent(pgEventObject *e, SDL_Event *event)`

Fill SDL event `event` with information from pygame user event instance `e`. Return 0 on success, -1 otherwise.

API exported by `pygame._freetype`

`src_c/_freetype.c`

This extension module defines Python type `pygame.freetype.Font`.

Header file: `src_c/include/pygame_freetype.h`

`pgFontObject`

The `pygame.freetype.Font` instance C struct.

`pgFont_Type`

The `pygame.freetype.Font` Python type.

`PyObject* pgFont_New(const char *filename, long font_index)`

Open the font file with path `filename` and return a new new `pygame.freetype.Font` instance for that font. Set `font_index` to 0 unless the file contains multiple, indexed, fonts. On error raise a Python exception and return `NULL`.

`int pgFont_Check(PyObject *x)`

Return true if `x` is a `pygame.freetype.Font` instance. Will return false for a subclass of `Font`. This is a macro. No check is made that `x` is not `NULL`.

`int pgFont_IS_ALIVE(PyObject *o)`

Return true if `pygame.freetype.Font` object `o` is an open font file. This is a macro. No check is made that `o` is not `NULL` or not a `Font` instance.

API exported by `pygame.mixer`

`src_c/mixer.c`

Python types and module startup/shutdown functions defined in the `pygame.mixer` extension module.

Header file: `src_c/include/pygame_mixer.h`

`pgSoundObject`

The `pygame.mixer.Sound` instance C structure.

`PyTypeObject *pgSound_Type`

The `pygame.mixer.Sound` Python type.

`PyObject* pgSound_New(Mix_Chunk *chunk)`

Return a new `pygame.mixer.Sound` instance for the SDL mixer chunk `chunk`. On failure, raise a Python exception and return `NULL`.

`int pgSound_Check(PyObject *obj)`

Return true if `obj` is an instance of type `pgSound_Type`, but not a `pgSound_Type` subclass instance. A macro.

`Mix_Chunk* pgSound_AsChunk(PyObject *x)`

Return the SDL `Mix_Chunk` struct associated with the `pgSound_Type` instance `x`. A macro that does no `NULL` or Python type check on `x`.

`pgChannelObject`

The `pygame.mixer.Channel` instance C structure.

`PyTypeObject *pgChannel_Type`

The `pygame.mixer.Channel` Python type.

`PyObject* pgChannel_New(int channelnum)`

Return a new `pygame.mixer.Channel` instance for the SDL mixer channel *channelnum*. On failure, raise a Python exception and return `NULL`.

`int pgChannel_Check(PyObject *obj)`

Return true if *obj* is an instance of type `pgChannel_Type`, but not a `pgChannel_Type` subclass instance. A macro.

`Mix_Chunk *pgChannel_AsInt(PyObject *x)`

Return the SDL mixer music channel number associated with `pgChannel_Type` instance *x*. A macro that does no `NULL` or Python type check on *x*.

`void pgMixer_AutoInit(void)`

Initialize the `pygame.mixer` module and start the SDL mixer.

`void pgMixer_AutoQuit(void)`

Stop all playing channels and close the SDL mixer.

Class Rect API exported by `pygame.rect`

`src_c/rect.c`

This extension module defines Python type `pygame.Rect`.

Header file: `src_c/include/pygame.h`

`GAME_Rect`

`int x`

Horizontal position of the top-left corner.

`int y`

Vertical position of the top-left corner.

`int w`

Width.

`int h`

Height.

A rectangle at (*x*, *y*) and of size (*w*, *h*).

SDL 2: equivalent to `SDL_Rect`.

`pgRectObject`

`GAME_Rect r`

The Pygame rectangle type instance.

`PyTypeObject *pgRect_Type`

The Pygame rectangle object type `pygame.Rect`.

`GAME_Rect pgRect_AsRect(PyObject *obj)`

A macro to access the `GAME_Rect` field of a `pygame.Rect` instance.

`PyObject* pgRect_New(SDL_Rect *r)`

Return a new `pygame.Rect` instance from the `SDL_Rect` *r*. On failure, raise a Python exception and return `NULL`.

`PyObject* pgRect_New4(int x, int y, int w, int h)`

Return a new `pygame.Rect` instance with position (*x*, *y*) and size (*w*, *h*). On failure raise a Python exception and return `NULL`.

`GAME_Rect* pgRect_FromObject(PyObject *obj, GAME_Rect *temp)`

Translate a Python rectangle representation as a Pygame `GAME_Rect`. A rectangle can be a length 4 sequence integers (*x*, *y*, *w*, *h*), or a length 2 sequence of position (*x*, *y*) and size (*w*, *h*), or a length 1 tuple containing a rectangle representation, or have a method *rect* that returns a rectangle. Pass a pointer to a locally declared c:type: `GAME_Rect` as *temp*. Do not rely on this being filled in; use the function's return value instead. On success return a pointer to a `GAME_Rect` representation of the rectangle. One failure may raise a Python exception before returning `NULL`.

`void pgRect_Normalize(GAME_Rect *rect)`

Normalize the given rect. A rect with a negative size (negative width and/or height) will be adjusted to have a positive size.

API exported by *pygame.rwobject*

src_c/rwobject.c

This extension module implements functions for wrapping a Python file like object in a `SDL_RWops` struct for SDL file access.

Header file: `src_c/include/pygame.h`

`SDL_RWops* pgRWops_FromObject(PyObject *obj)`

Return a `SDL_RWops` struct filled to access *obj*. If *obj* is a string then let SDL open the file it names. Otherwise, if *obj* is a Python file-like object then use its `read`, `write`, `seek`, `tell`, and `close` methods. If threads are available, the Python GIL is acquired before calling any of the *obj* methods. On error raise a Python exception and return `NULL`.

`SDL_RWops* pgRWops_FromFileObject(PyObject *obj)`

Return a `SDL_RWops` struct filled to access the Python file-like object *obj*. Uses its `read`, `write`, `seek`, `tell`, and `close` methods. If threads are available, the Python GIL is acquired before calling any of the *obj* methods. On error raise a Python exception and return `NULL`.

`int pgRWops_CheckObject(SDL_RWops *rw)`

Return true if *rw* is a Python file-like object wrapper returned by `pgRWops_FromObject()` or `pgRWops_FromFileObject()`.

`int pgRWops_ReleaseObject(SDL_RWops *context)`

Free a `SDL_RWops` struct. If it is attached to a Python file-like object, decrement its refcount. Otherwise, close the file handle. Return 0 on success. On error, raise a Python exception and return a negative value.

`PyObject* pg_EncodeFilePath(PyObject *obj, PyObject *eclass)`

Return the file path *obj* as a byte string properly encoded for the OS. Null bytes are forbidden in the encoded file path. On error raise a Python exception and return `NULL`, using *eclass* as the exception type if it is not `NULL`. If *obj* is `NULL` assume an exception was already raised and pass it on.

`PyObject* pg_EncodeString(PyObject *obj, const char *encoding, const char *errors, PyObject *eclass)`

Return string *obj* as an encoded byte string. The C string arguments *encoding* and *errors* are the same as for `PyUnicode_AsEncodedString()`. On error raise a Python exception and return `NULL`, using *eclass* as the exception type if it is not `NULL`. If *obj* is `NULL` assume an exception was already raised and pass it on.

Class Surface API exported by *pygame.surface*

src_c/surface.c

This extension module defines Python type `pygame.Surface`.

Header file: `src_c/include/pygame.h`

`pgSurfaceObject`

A `pygame.Surface` instance.

`PyTypeObject *pgSurface_Type`

The `pygame.Surface` Python type.

`int pgSurface_Check(PyObject *x)`

Return true if *x* is a `pygame.Surface` instance

Will return false if *x* is a subclass of *Surface*. This is a macro. No check is made that *x* is not `NULL`.

`pgSurfaceObject* pgSurface_New(SDL_Surface *s)`

Return a new new pygame surface instance for SDL surface *s*. Return `NULL` on error.

`SDL_Surface* pgSurface_AssSurface(PyObject *x)`

Return a pointer the SDL surface represented by the pygame Surface instance *x*.

This is a macro. Argument *x* is assumed to be a Surface, or subclass of Surface, instance.

```
int pgSurface_Blit(PyObject *dstobj, PyObject *srcobj, SDL_Rect *dstrect,
SDL_Rect *srcrect, int the_args)
```

Blit the *srcrect* portion of Surface *srcobj* onto Surface *dstobj* at *srcobj*

Argument *the_args* indicates the type of blit to perform: Normal blit (0), `PYGAME_BLEND_ADD`, `PYGAME_BLEND_SUB`, `PYGAME_BLEND_SUB`, `PYGAME_BLEND_MULT`, `PYGAME_BLEND_MIN`, `PYGAME_BLEND_MAX`, `PYGAME_BLEND_RGBA_ADD`, `PYGAME_BLEND_RGBA_SUB`, `PYGAME_BLEND_RGBA_MULT`, `PYGAME_BLEND_RGBA_MIN`, `PYGAME_BLEND_RGBA_MAX`, `PYGAME_BLEND_ALPHA_SDL2` and `PYGAME_BLEND_PREMULTIPLIED`. Argument *dstrect* is updated to the actual area on *dstobj* affected by the blit. The C version of the `pygame.Surface.blit()` method. Return 1 on success, 0 on an exception.

API exported by `pygame.surflock`

`src_c/surflock.c`

This extension module implements SDL surface locking for the `pygame.Surface` type.

Header file: `src_c/include/pygame.h`

`pgLifetimeLockObject`

`PyObject *surface`

An SDL locked pygame surface.

`PyObject *lockobj`

The Python object which owns the lock on the surface. This field does not own a reference to the object.

The lifetime lock type instance. A lifetime lock pairs a locked pygame surface with the Python object that locked the surface for modification. The lock is removed automatically when the lifetime lock instance is garbage collected.

`PyTypeObject *pgLifetimeLock_Type`

The pygame internal surflock lifetime lock object type.

`int pgLifetimeLock_Check(PyObject *x)`

Return true if Python object *x* is a `pgLifetimeLock_Type` instance, false otherwise. This will return false on `pgLifetimeLock_Type` subclass instances as well.

`void pgSurface_Prep(pgSurfaceObject *surfobj)`

If *surfobj* is a subsurface, then lock the parent surface with *surfobj* the owner of the lock.

`void pgSurface_Unprep(pgSurfaceObject *surfobj)`

If *surfobj* is a subsurface, then release its lock on the parent surface.

`int pgSurface_Lock(pgSurfaceObject *surfobj)`

Lock pygame surface *surfobj*, with *surfobj* owning its own lock.

`int pgSurface_LockBy(pgSurfaceObject *surfobj, PyObject *lockobj)`

Lock pygame surface *surfobj* with Python object *lockobj* the owning the lock.

The surface will keep a weak reference to object *lockobj*, and eventually remove the lock on itself if *lockobj* is garbage collected. However, it is best if *lockobj* also keep a reference to the locked surface and call to `pgSurface_UnLockBy()` when finished with the surface.

`int pgSurface_UnLock(pgSurfaceObject *surfobj)`

Remove the pygame surface *surfobj* object's lock on itself.

`int pgSurface_UnLockBy(pgSurfaceObject *surfobj, PyObject *lockobj)`

Remove the lock on pygame surface *surfobj* owned by Python object *lockobj*.

`PyObject *pgSurface_LockLifetime(PyObject *surfobj, PyObject *lockobj)`

Lock pygame surface *surfobj* for Python object *lockobj* and return a new `pgLifetimeLock_Type` instance for the lock.

This function is not called anywhere within pygame. It and `pgLifetimeLock_Type` are candidates for removal.

API exported by `pygame.version`

src_py/version.py

Header file: `src_c/include/pygame.h`

Version information can be retrieved at compile-time using these macros.

New in version 1.9.5: *New in version 1.9.5.*

PG_MAJOR_VERSION

PG_MINOR_VERSION

PG_PATCH_VERSION

PG_VERSIONNUM (MAJOR, MINOR, PATCH)

Returns an integer representing the given version.

PG_VERSION_ATLEAST (MAJOR, MINOR, PATCH)

Returns true if the current version is at least equal to the specified version.

`src_c/include/` contains header files for applications that use the pygame C API, while `src_c/` contains headers used by pygame internally.

File Path Function Arguments

A pygame function or method which takes a file path argument will accept either a Unicode or a byte (8-bit or ASCII character) string. Unicode strings are translated to Python's default filesystem encoding, as returned by `sys.getfilesystemencoding()`. A Unicode code point above U+FFFF (`\uFFFF`) can be coded directly with a 32-bit escape sequences (`\Uxxxxxxxx`), even for Python interpreters built with an UCS-2 (16-bit character) Unicode type. Byte strings are passed to the operating system unchanged.

Null characters (`\x00`) are not permitted in the path, raising an exception. An exception is also raised if an Unicode file path cannot be encoded. How UTF-16 surrogate codes are handled is Python-interpreter-dependent. Use UTF-32 code points and 32-bit escape sequences instead. The exception types are function-dependent.

Welcome to an early "alpha" version of downloadable docs.

Please email any suggestions to: rene@pygame.org

Tutorials

Newbie Guide

A list of thirteen helpful tips for people to get comfortable using pygame.

Import and Initialize

The beginning steps on importing and initializing pygame. The pygame package is made of several modules. Some modules are not included on all platforms.

How do I move an Image?

A basic tutorial that covers the concepts behind 2D computer animation. Information about drawing and clearing objects to make them appear animated.

Chimp Tutorial, Line by Line

The pygame examples include a simple program with an interactive fist and a chimpanzee. This was inspired by the annoying flash banner of the early 2000s. This tutorial examines every line of code used in the example.

Sprite Module Introduction

Pygame includes a higher level sprite module to help organize games. The sprite module includes several classes that help manage details found in almost all games types. The Sprite classes are a bit more advanced than the regular pygame modules, and need more understanding to be properly used.

Surfarray Introduction

Pygame used the NumPy python module to allow efficient per pixel effects on images. Using the surface arrays is an advanced feature that allows custom effects and filters. This also examines some of the simple effects from the pygame example, `arraydemo.py`.

Camera Module Introduction

Pygame, as of 1.9, has a camera module that allows you to capture images, watch live streams, and do some basic computer vision. This tutorial covers those use cases.

Making Games Tutorial

A large tutorial that covers the bigger topics needed to create an entire game.

Display Modes

Getting a display surface for the screen.

Introduction to Pygame

An introduction to the basics of pygame. This is written for users of Python and appeared in volume two of the Py magazine.

Reference

genindex

A list of all functions, classes, and methods in the pygame package.

pygame.BufferProxy

An array protocol view of surface pixels

pygame.cdrom

How to access and control the CD audio devices.

pygame.Color

Color representation.

pygame.cursors

Loading and compiling cursor images.

pygame.display

Configure the display surface.

pygame.draw

Drawing simple shapes like lines and ellipses to surfaces.

pygame.event

Manage the incoming events from various input devices and the windowing platform.

pygame.examples

Various programs demonstrating the use of individual pygame modules.

pygame.font

Loading and rendering TrueType fonts.

pygame.freetype

Enhanced pygame module for loading and rendering font faces.

pygame.gfxdraw

Anti-aliasing draw functions.

pygame.image

Loading, saving, and transferring of surfaces.

pygame.joystick

Manage the joystick devices.

pygame.key

Manage the keyboard device.

pygame.locals

Pygame constants.

pygame.mixer

Load and play sounds

pygame.mouse

Manage the mouse device and display.

Reference

`pygame.mixer.music`

Play streaming music tracks.

`pygame.Overlay`

Access advanced video overlays.

`pygame`

Top level functions to manage pygame.

`pygame.PixelArray`

Manipulate image pixel data.

`pygame.Rect`

Flexible container for a rectangle.

`pygame.scrap`

Native clipboard access.

`pygame.sndarray`

Manipulate sound sample data.

`pygame.sprite`

Higher level objects to represent game images.

`pygame.Surface`

Objects for images and the screen.

`pygame.surfarray`

Manipulate image pixel data.

`pygame.tests`

Test pygame.

`pygame.time`

Manage timing and framerate.

`pygame.transform`

Resize and move images.

`pygame C API`

The C api shared amongst pygame extension modules.

search

Search pygame documents by keyword.

Index

__dict__ (pygame.event.EventType attribute)
_pixels_address (pygame.Surface attribute)

A

a (pygame.Color attribute)
aacircle() (in module pygame.gfxdraw)
aaellipse() (in module pygame.gfxdraw)
aaline() (in module pygame.draw)
aalines() (in module pygame.draw)
aapolygon() (in module pygame.gfxdraw)
aatrigon() (in module pygame.gfxdraw)
abort() (pygame.midi.Output method)
add() (pygame.sprite.Group method)
 (pygame.sprite.LayeredUpdates method)
 (pygame.sprite.Sprite method)
aliens.main() (in module pygame.examples)
alive() (pygame.sprite.Sprite method)
angle() (pygame.mask.Mask method)
angle_to() (pygame.math.Vector2 method)
 (pygame.math.Vector3 method)
antialiased (pygame.freetype.Font attribute)
arc() (in module pygame.draw)
 (in module pygame.gfxdraw)
array() (in module pygame.sndarray)
array2d() (in module pygame.surfarray)
array3d() (in module pygame.surfarray)
array_alpha() (in module pygame.surfarray)
array_colorkey() (in module pygame.surfarray)
array_to_surface() (in module pygame.pixelcopy)
arraydemo.main() (in module pygame.examples)
as_polar() (pygame.math.Vector2 method)
as_spherical() (pygame.math.Vector3 method)
ascender (pygame.freetype.Font attribute)
average_color() (in module pygame.transform)
average_surfaces() (in module pygame.transform)

B

b (pygame.Color attribute)
bezier() (in module pygame.gfxdraw)

bicolor (pygame.freetype.Font attribute)
blend_fill.main() (in module pygame.examples)
blit() (pygame.Surface method)
blit_array() (in module pygame.surfarray)
blit_blends.main() (in module pygame.examples)
blits() (pygame.Surface method)
bold (pygame.font.Font attribute)
box() (in module pygame.gfxdraw)
BufferProxy (class in pygame)

C

Camera (class in pygame.camera)
camera.main() (in module pygame.examples)
CD (class in pygame.cdrom)
centroid() (pygame.mask.Mask method)
change_layer() (pygame.sprite.LayeredDirty method)
 (pygame.sprite.LayeredUpdates method)
Channel (class in pygame.mixer)
chimp.main() (in module pygame.examples)
chop() (in module pygame.transform)
circle() (in module pygame.draw)
 (in module pygame.gfxdraw)
clamp() (pygame.Rect method)
clamp_ip() (pygame.Rect method)
clear() (in module pygame.event)
 (pygame.mask.Mask method)
 (pygame.sprite.Group method)
 (pygame.sprite.LayeredDirty method)
clip() (pygame.Rect method)
clipline() (pygame.Rect method)
Clock (class in pygame.time)
close() (pygame.midi.Input method)
 (pygame.midi.Output method)
 (pygame.PixelArray method)
cmy (pygame.Color attribute)
collide_circle() (in module pygame.sprite)
collide_circle_ratio() (in module pygame.sprite)
collide_mask() (in module pygame.sprite)
collide_rect() (in module pygame.sprite)
collide_rect_ratio() (in module pygame.sprite)
collidedict() (pygame.Rect method)
collidedictall() (pygame.Rect method)

`collidelist()` (pygame.Rect method)
`collidelistall()` (pygame.Rect method)
`collidepoint()` (pygame.Rect method)
`collidect()` (pygame.Rect method)
`Color` (class in pygame)
`colorspace()` (in module pygame.camera)
`compare()` (pygame.PixelArray method)
`compile()` (in module pygame.cursors)
`connected_component()` (pygame.mask.Mask method)
`connected_components()` (pygame.mask.Mask method)
`contains()` (in module pygame.scrap)
 (pygame.Rect method)
`convert()` (pygame.Surface method)
`convert_alpha()` (pygame.Surface method)
`convolve()` (pygame.mask.Mask method)
`copy()` (pygame.mask.Mask method)
 (pygame.Rect method)
 (pygame.sprite.Group method)
 (pygame.Surface method)
`correct_gamma()` (pygame.Color method)
`count()` (pygame.mask.Mask method)
`cross()` (pygame.math.Vector2 method)
 (pygame.math.Vector3 method)
`Cursor` (class in pygame.cursors)
`cursors.main()` (in module pygame.examples)
`custom_type()` (in module pygame.event)

D

`data` (pygame.cursors.Cursor attribute)
`delay()` (in module pygame.time)
`descender` (pygame.freetype.Font attribute)
`DirtySprite` (class in pygame.sprite)
`disable_swizzling()` (in module pygame.math)
`display()` (pygame.Overlay method)
`distance_squared_to()` (pygame.math.Vector2 method)
 (pygame.math.Vector3 method)
`distance_to()` (pygame.math.Vector2 method)
 (pygame.math.Vector3 method)
`dot()` (pygame.math.Vector2 method)
 (pygame.math.Vector3 method)
`draw()` (pygame.mask.Mask method)

(pygame.sprite.Group method)
(pygame.sprite.LayeredDirty method)
(pygame.sprite.LayeredUpdates method)
(pygame.sprite.RenderUpdates method)

E

`eject()` (pygame.cdrom.CD method)
`elementwise()` (pygame.math.Vector2 method)
 (pygame.math.Vector3 method)
`ellipse()` (in module pygame.draw)
 (in module pygame.gfxdraw)
`empty()` (pygame.sprite.Group method)
`enable_swizzling()` (in module pygame.math)
`encode_file_path()` (in module pygame)
`encode_string()` (in module pygame)
`erase()` (pygame.mask.Mask method)
`error`
`Event()` (in module pygame.event)
`event_name()` (in module pygame.event)
`eventlist.main()` (in module pygame.examples)
`EventType` (class in pygame.event)
`extract()` (pygame.PixelArray method)

F

`fadeout()` (in module pygame.mixer)
 (in module pygame.mixer.music)
 (pygame.mixer.Channel method)
 (pygame.mixer.Sound method)
`fastevents.main()` (in module pygame.examples)
`fgcolor` (pygame.freetype.Font attribute)
`fill()` (pygame.mask.Mask method)
 (pygame.Surface method)
`filled_circle()` (in module pygame.gfxdraw)
`filled_ellipse()` (in module pygame.gfxdraw)
`filled_polygon()` (in module pygame.gfxdraw)
`filled_trigon()` (in module pygame.gfxdraw)
`find_channel()` (in module pygame.mixer)
`fit()` (pygame.Rect method)
`fixed_sizes` (pygame.freetype.Font attribute)
`fixed_width` (pygame.freetype.Font attribute)
`flip()` (in module pygame.display)
 (in module pygame.transform)

Font (class in pygame.font)
(class in pygame.freetype)
fonty.main() (in module pygame.examples)
freetype_misc.main() (in module pygame.examples)
frequency_to_midi() (in module pygame.midi)
from_polar() (pygame.math.Vector2 method)
from_spherical() (pygame.math.Vector3 method)
from_surface() (in module pygame.mask)
from_threshold() (in module pygame.mask)
frombuffer() (in module pygame.image)
fromstring() (in module pygame.image)

G

g (pygame.Color attribute)
GAME_Rect (C type)
GAME_Rect.h (C member)
GAME_Rect.w (C member)
GAME_Rect.x (C member)
GAME_Rect.y (C member)
get() (in module pygame.event)
(in module pygame.fastevent)
(in module pygame.scrap)
get_abs_offset() (pygame.Surface method)
get_abs_parent() (pygame.Surface method)
get_active() (in module pygame.display)
get_all() (pygame.cdrom.CD method)
get_allow_screensaver() (in module pygame.display)
get_alpha() (pygame.Surface method)
get_arraytype() (in module pygame.sndarray)
(in module pygame.surfarray)
get_arraytypes() (in module pygame.sndarray)
(in module pygame.surfarray)
get_ascent() (pygame.font.Font method)
get_at() (pygame.mask.Mask method)
(pygame.Surface method)
get_at_mapped() (pygame.Surface method)
get_axis() (pygame.joystick.Joystick method)
get_ball() (pygame.joystick.Joystick method)
get_bitsize() (pygame.Surface method)
get_blocked() (in module pygame.event)
get_bold() (pygame.font.Font method)
get_bottom_layer() (pygame.sprite.LayeredUpdates method)
get_bounding_rect() (pygame.Surface method)
get_bounding_rects() (pygame.mask.Mask method)
get_buffer() (pygame.Surface method)
get_busy() (in module pygame.mixer)
(in module pygame.mixer.music)
(pygame.cdrom.CD method)
(pygame.mixer.Channel method)
get_button() (pygame.joystick.Joystick method)
get_bytesize() (pygame.Surface method)
get_cache_size() (in module pygame.freetype)
get_caption() (in module pygame.display)
get_clip() (pygame.sprite.LayeredDirty method)
(pygame.Surface method)
get_colorkey() (pygame.Surface method)
get_controls() (pygame.camera.Camera method)
get_count() (in module pygame.cdrom)
(in module pygame.joystick)
(in module pygame.midi)
get_current() (pygame.cdrom.CD method)
get_cursor() (in module pygame.mouse)
get_default_font() (in module pygame.font)
(in module pygame.freetype)
get_default_input_id() (in module pygame.midi)
get_default_output_id() (in module pygame.midi)
get_default_resolution() (in module pygame.freetype)
get_descent() (pygame.font.Font method)
get_device() (in module pygame._sdl2.touch)
get_device_info() (in module pygame.midi)
get_driver() (in module pygame.display)
get_empty() (pygame.cdrom.CD method)
get_endevent() (in module pygame.mixer.music)
(pygame.mixer.Channel method)
get_error() (in module pygame)
(in module pygame.freetype)
get_extended() (in module pygame.image)
get_finger() (in module pygame._sdl2.touch)
get_flags() (pygame.Surface method)
get_focused() (in module pygame.key)
(in module pygame.mouse)
get_fonts() (in module pygame.font)

`get_fps()` (pygame.time.Clock method)
`get_grab()` (in module pygame.event)
`get_guid()` (pygame.joystick.Joystick method)
`get_hardware()` (pygame.Overlay method)
`get_hat()` (pygame.joystick.Joystick method)
`get_height()` (pygame.font.Font method)
 (pygame.Surface method)
`get_id()` (pygame.cdrom.CD method)
 (pygame.joystick.Joystick method)
`get_image()` (pygame.camera.Camera method)
`get_init()` (in module pygame)
 (in module pygame.cdrom)
 (in module pygame.display)
 (in module pygame.fastevent)
 (in module pygame.font)
 (in module pygame.freetype)
 (in module pygame.joystick)
 (in module pygame.midi)
 (in module pygame.mixer)
 (in module pygame.scrap)
 (pygame.cdrom.CD method)
 (pygame.joystick.Joystick method)
`get_instance_id()` (pygame.joystick.Joystick method)
`get_italic()` (pygame.font.Font method)
`get_layer_of_sprite()` (pygame.sprite.LayeredUpdates method)
`get_length()` (pygame.mixer.Sound method)
`get_linesize()` (pygame.font.Font method)
`get_locked()` (pygame.Surface method)
`get_locks()` (pygame.Surface method)
`get_losses()` (pygame.Surface method)
`get_masks()` (pygame.Surface method)
`get_metrics()` (pygame.freetype.Font method)
`get_mods()` (in module pygame.key)
`get_name()` (pygame.cdrom.CD method)
 (pygame.joystick.Joystick method)
`get_num_channels()` (in module pygame.mixer)
 (pygame.mixer.Sound method)
`get_num_devices()` (in module pygame._sdl2.touch)
`get_num_displays()` (in module pygame.display)
`get_num_fingers()` (in module pygame._sdl2.touch)
`get_numaxes()` (pygame.joystick.Joystick method)
`get_numballs()` (pygame.joystick.Joystick method)
`get_numbuttons()` (pygame.joystick.Joystick method)
`get_numhats()` (pygame.joystick.Joystick method)
`get_numtracks()` (pygame.cdrom.CD method)
`get_offset()` (pygame.Surface method)
`get_palette()` (pygame.Surface method)
`get_palette_at()` (pygame.Surface method)
`get_parent()` (pygame.Surface method)
`get_paused()` (pygame.cdrom.CD method)
`get_pitch()` (pygame.Surface method)
`get_pos()` (in module pygame.mixer.music)
 (in module pygame.mouse)
`get_power_level()` (pygame.joystick.Joystick method)
`get_pressed()` (in module pygame.key)
 (in module pygame.mouse)
`get_queue()` (pygame.mixer.Channel method)
`get_raw()` (pygame.camera.Camera method)
 (pygame.mixer.Sound method)
`get_rawtime()` (pygame.time.Clock method)
`get_rect()` (pygame.freetype.Font method)
 (pygame.mask.Mask method)
 (pygame.Surface method)
`get_rel()` (in module pygame.mouse)
`get_repeat()` (in module pygame.key)
`get_sdl_byteorder()` (in module pygame)
`get_sdl_image_version()` (in module pygame.image)
`get_sdl_mixer_version()` (in module pygame.mixer)
`get_sdl_version()` (in module pygame)
`get_shifts()` (pygame.Surface method)
`get_size()` (pygame.camera.Camera method)
 (pygame.mask.Mask method)
 (pygame.Surface method)
`get_sized_ascender()` (pygame.freetype.Font method)
`get_sized_descender()` (pygame.freetype.Font method)
`get_sized_glyph_height()` (pygame.freetype.Font method)
`get_sized_height()` (pygame.freetype.Font method)
`get_sizes()` (pygame.freetype.Font method)
`get_smoothscale_backend()` (in module pygame.transform)
`get_sound()` (pygame.mixer.Channel method)
`get_sprite()` (pygame.sprite.LayeredUpdates method)

`get_sprites_at()` (pygame.sprite.LayeredUpdates method)
`get_sprites_from_layer()` (pygame.sprite.LayeredUpdates method)
`get_surface()` (in module pygame.display)
`get_ticks()` (in module pygame.time)
`get_time()` (pygame.time.Clock method)
`get_top_layer()` (pygame.sprite.LayeredUpdates method)
`get_top_sprite()` (pygame.sprite.LayeredUpdates method)
`get_track_audio()` (pygame.cdrom.CD method)
`get_track_length()` (pygame.cdrom.CD method)
`get_track_start()` (pygame.cdrom.CD method)
`get_types()` (in module pygame.scrap)
`get_underline()` (pygame.font.Font method)
`get_version()` (in module pygame.freetype)
`get_view()` (pygame.Surface method)
`get_visible()` (in module pygame.mouse)
`get_volume()` (in module pygame.mixer.music)
 (pygame.mixer.Channel method)
 (pygame.mixer.Sound method)
`get_width()` (pygame.Surface method)
`get_window_size()` (in module pygame.display)
`get_wm_info()` (in module pygame.display)
`gl_get_attribute()` (in module pygame.display)
`gl_set_attribute()` (in module pygame.display)
`glcube.main()` (in module pygame.examples)
Group (class in pygame.sprite)
`groupcollide()` (in module pygame.sprite)
`groups()` (pygame.sprite.Sprite method)
`GroupSingle()` (in module pygame.sprite)

H

`has()` (pygame.sprite.Group method)
`headless_no_windows_needed.main()` (in module pygame.examples)
`height` (pygame.freetype.Font attribute)
`hline()` (in module pygame.gfxdraw)
`hsla` (pygame.Color attribute)
`hsva` (pygame.Color attribute)

I

`i1i2i3` (pygame.Color attribute)

`iconify()` (in module pygame.display)
`import_pygame_base` (C function)
`inflate()` (pygame.Rect method)
`inflate_ip()` (pygame.Rect method)
`Info()` (in module pygame.display)
`init()` (in module pygame)
 (in module pygame.cdrom)
 (in module pygame.display)
 (in module pygame.fastevent)
 (in module pygame.font)
 (in module pygame.freetype)
 (in module pygame.joystick)
 (in module pygame.midi)
 (in module pygame.mixer)
 (in module pygame.scrap)
 (pygame.cdrom.CD method)
 (pygame.joystick.Joystick method)
Input (class in pygame.midi)
`invert()` (pygame.mask.Mask method)
`is_normalized()` (pygame.math.Vector2 method)
 (pygame.math.Vector3 method)
`italic` (pygame.font.Font attribute)
`itemsiz` (pygame.PixelArray attribute)

J

Joystick (class in pygame.joystick)

K

`ker` (pygame.freetype.Font attribute)
`key_code()` (in module pygame.key)
`kill()` (pygame.sprite.Sprite method)

L

`laplacian()` (in module pygame.transform)
LayeredDirty (class in pygame.sprite)
LayeredUpdates (class in pygame.sprite)
`layers()` (pygame.sprite.LayeredUpdates method)
`length` (pygame.BufferProxy attribute)
`length()` (pygame.math.Vector2 method)
 (pygame.math.Vector3 method)
`length_squared()` (pygame.math.Vector2 method)
 (pygame.math.Vector3 method)
`lerp()` (pygame.Color method)

(pygame.math.Vector2 method)
(pygame.math.Vector3 method)
line() (in module pygame.draw)
(in module pygame.gfxdraw)
lines() (in module pygame.draw)
liquid.main() (in module pygame.examples)
list_cameras() (in module pygame.camera)
list_modes() (in module pygame.display)
load() (in module pygame.image)
(in module pygame.mixer.music)
load_basic() (in module pygame.image)
load_extended() (in module pygame.image)
load_xbm() (in module pygame.cursors)
lock() (pygame.Surface method)
lost() (in module pygame.scrap)

M

magnitude() (pygame.math.Vector2 method)
(pygame.math.Vector3 method)
magnitude_squared() (pygame.math.Vector2 method)
(pygame.math.Vector3 method)
make_sound() (in module pygame.sndarray)
make_surface() (in module pygame.pixelcopy)
(in module pygame.surfarray)
(pygame.PixelArray method)
map_array() (in module pygame.pixelcopy)
(in module pygame.surfarray)
map_rgb() (pygame.Surface method)
Mask (class in pygame.mask)
mask.main() (in module pygame.examples)
match_font() (in module pygame.font)
metrics() (pygame.font.Font method)
midi.main() (in module pygame.examples)
midi_to_ansi_note() (in module pygame.midi)
midi_to_frequency() (in module pygame.midi)
MidiException
midis2events() (in module pygame.midi)
mode_ok() (in module pygame.display)
move() (pygame.Rect method)
move_ip() (pygame.Rect method)
move_to_back() (pygame.sprite.LayeredUpdates
method)

move_to_front() (pygame.sprite.LayeredUpdates
method)
moveit.main() (in module pygame.examples)
mustlock() (pygame.Surface method)

N

name (pygame.freetype.Font attribute)
name() (in module pygame.key)
ndim (pygame.PixelArray attribute)
normalize() (pygame.Color method)
(pygame.math.Vector2 method)
(pygame.math.Vector3 method)
(pygame.Rect method)
normalize_ip() (pygame.math.Vector2 method)
(pygame.math.Vector3 method)
note_off() (pygame.midi.Output method)
note_on() (pygame.midi.Output method)

O

oblique (pygame.freetype.Font attribute)
oldalien.main() (in module pygame.examples)
OrderedUpdates() (in module pygame.sprite)
origin (pygame.freetype.Font attribute)
outline() (pygame.mask.Mask method)
Output (class in pygame.midi)
overlap() (pygame.mask.Mask method)
overlap_area() (pygame.mask.Mask method)
overlap_mask() (pygame.mask.Mask method)
Overlay (class in pygame)
overlay.main() (in module pygame.examples)

P

pad (pygame.freetype.Font attribute)
parent (pygame.BufferProxy attribute)
path (pygame.freetype.Font attribute)
pause() (in module pygame.mixer)
(in module pygame.mixer.music)
(pygame.cdrom.CD method)
(pygame.mixer.Channel method)
peek() (in module pygame.event)
pg_buffer (C type)
pg_buffer.consumer (C member)
pg_buffer.release_buffer (C member)

pg_buffer.view (C member)	pgColor_NewLength (C function)
pg_EncodeFilePath (C function)	pgColor_Type (C variable)
pg_EncodeString (C function)	pgDict_AsBuffer (C function)
pg_FloatFromObj (C function)	pgEvent_Check (C function)
pg_FloatFromObjIndex (C function)	pgEvent_FillUserEvent (C function)
pg_GetDefaultWindow (C function)	pgEvent_New (C function)
pg_GetDefaultWindowSurface (C function)	pgEvent_New2 (C function)
pg_IntFromObj (C function)	pgEvent_Type (C variable)
pg_IntFromObjIndex (C function)	pgEventObject (C type)
PG_MAJOR_VERSION (C macro)	pgEventObject.type (C member)
PG_MINOR_VERSION (C macro)	pgExc_BufferError (C variable)
PG_PATCH_VERSION (C macro)	pgExc_SDLLError (C variable)
pg_RegisterQuit (C function)	pgFont_Check (C function)
pg_RGBAFromObj (C function)	pgFont_IS_ALIVE (C function)
pg_SetDefaultWindow (C function)	pgFont_New (C function)
pg_SetDefaultWindowSurface (C function)	pgFont_Type (C type)
pg_TwoFloatsFromObj (C function)	pgFontObject (C type)
pg_TwoIntsFromObj (C function)	pgLifetimeLock_Check (C function)
pg_UintFromObj (C function)	pgLifetimeLock_Type (C variable)
pg_UintFromObjIndex (C function)	pgLifetimeLockObject (C type)
PG_VERSION_ATLEAST (C function)	pgLifetimeLockObject.lockobj (C member)
PG_VERSIONNUM (C function)	pgLifetimeLockObject.surface (C member)
pgBuffer_AsArrayInterface (C function)	pgMixer_AutoInit (C function)
pgBuffer_AsArrayStruct (C function)	pgMixer_AutoQuit (C function)
pgBuffer_Release (C function)	pgObject_GetBuffer (C function)
pgBufproxy_Check (C function)	pgRect_AsRect (C function)
pgBufproxy_GetParent (C function)	pgRect_FromObject (C function)
pgBufproxy_New (C function)	pgRect_New (C function)
pgBufproxy_Trip (C function)	pgRect_New4 (C function)
pgBufproxy_Type (C variable)	pgRect_Normalize (C function)
pgCD_AsID (C function)	pgRect_Type (C variable)
pgCD_Check (C function)	pgRectObject (C type)
pgCD_New (C function)	pgRectObject.r (C member)
pgCD_Type (C variable)	pgRWops_CheckObject (C function)
pgCDOBJECT (C type)	pgRWops_FromFileObject (C function)
pgChannel_AsInt (C function)	pgRWops_FromObject (C function)
pgChannel_Check (C function)	pgRWops_ReleaseObject (C function)
pgChannel_New (C function)	pgSound_AsChunk (C function)
pgChannel_Type (C variable)	pgSound_Check (C function)
pgChannelObject (C type)	pgSound_New (C function)
pgColor_Check (C function)	pgSound_Type (C variable)
pgColor_New (C function)	pgSoundObject (C type)

pgSurface_AsSurface (C function)
pgSurface_Blit (C function)
pgSurface_Check (C function)
pgSurface_Lock (C function)
pgSurface_LockBy (C function)
pgSurface_LockLifetime (C function)
pgSurface_New (C function)
pgSurface_Prep (C function)
pgSurface_Type (C variable)
pgSurface_UnLock (C function)
pgSurface_UnLockBy (C function)
pgSurface_Unprep (C function)
pgSurfaceObject (C type)
pgVideo_AutoInit (C function)
pgVideo_AutoQuit (C function)
pgVidInfo_AsVidInfo (C function)
pgVidInfo_Check (C function)
pgVidInfo_New (C function)
pgVidInfo_Type (C variable)
pgVidInfoObject (C type)
pie() (in module pygame.gfxdraw)
pitch_bend() (pygame.midi.Output method)
pixel() (in module pygame.gfxdraw)
PixelArray (class in pygame)
pixelarray.main() (in module pygame.examples)
pixels2d() (in module pygame.surfarray)
pixels3d() (in module pygame.surfarray)
pixels_alpha() (in module pygame.surfarray)
pixels_blue() (in module pygame.surfarray)
pixels_green() (in module pygame.surfarray)
pixels_red() (in module pygame.surfarray)
play() (in module pygame.mixer.music)
 (pygame.cdrom.CD method)
 (pygame.mixer.Channel method)
 (pygame.mixer.Sound method)
playmus.main() (in module pygame.examples)
poll() (in module pygame.event)
 (in module pygame.fastevent)
 (pygame.midi.Input method)
polygon() (in module pygame.draw)
 (in module pygame.gfxdraw)
post() (in module pygame.event)

 (in module pygame.fastevent)
pre_init() (in module pygame.mixer)
premul_alpha() (pygame.Color method)
pump() (in module pygame.event)
 (in module pygame.fastevent)
put() (in module pygame.scrap)
pygame (module)
pygame._sdl2.touch (module)
pygame.camera (module)
pygame.cdrom (module)
pygame.cursors (module)
pygame.display (module)
pygame.draw (module)
pygame.event (module)
pygame.examples (module)
pygame.fastevent (module)
pygame.font (module)
pygame.freetype (module)
pygame.gfxdraw (module)
pygame.image (module)
pygame.joystick (module)
pygame.key (module)
pygame.locals (module)
pygame.mask (module)
pygame.math (module)
pygame.midi (module)
pygame.mixer (module)
pygame.mixer.music (module)
pygame.mouse (module)
pygame.pixelcopy (module)
pygame.scrap (module)
pygame.sndarray (module)
pygame.sprite (module)
pygame.surfarray (module)
pygame.tests (module)
pygame.time (module)
pygame.transform (module)
pygame.version (module)

Q

query_image() (pygame.camera.Camera method)
queue() (in module pygame.mixer.music)

(pygame.mixer.Channel method)
quit() (in module pygame)
(in module pygame.cdrom)
(in module pygame.display)
(in module pygame.font)
(in module pygame.freetype)
(in module pygame.joystick)
(in module pygame.midi)
(in module pygame.mixer)
(pygame.cdrom.CD method)
(pygame.joystick.Joystick method)

R

r (pygame.Color attribute)
raw (pygame.BufferProxy attribute)
read() (pygame.midi.Input method)
Rect (class in pygame)
rect() (in module pygame.draw)
rectangle() (in module pygame.gfxdraw)
reflect() (pygame.math.Vector2 method)
(pygame.math.Vector3 method)
reflect_ip() (pygame.math.Vector2 method)
(pygame.math.Vector3 method)
register_quit() (in module pygame)
remove() (pygame.sprite.Group method)
(pygame.sprite.Sprite method)
remove_sprites_of_layer()
(pygame.sprite.LayeredUpdates method)
render() (pygame.font.Font method)
(pygame.freetype.Font method)
render_raw() (pygame.freetype.Font method)
render_raw_to() (pygame.freetype.Font method)
render_to() (pygame.freetype.Font method)
RenderClear (class in pygame.sprite)
RenderPlain (class in pygame.sprite)
RenderUpdates (class in pygame.sprite)
repaint_rect() (pygame.sprite.LayeredDirty method)
replace() (pygame.PixelArray method)
resolution (pygame.freetype.Font attribute)
resume() (pygame.cdrom.CD method)
rev (in module pygame.version)
rewind() (in module pygame.mixer.music)

rotate() (in module pygame.transform)
(pygame.math.Vector2 method)
(pygame.math.Vector3 method)
rotate_ip() (pygame.math.Vector2 method)
(pygame.math.Vector3 method)
rotate_ip_rad() (pygame.math.Vector2 method)
(pygame.math.Vector3 method)
rotate_rad() (pygame.math.Vector2 method)
(pygame.math.Vector3 method)
rotate_x() (pygame.math.Vector3 method)
rotate_x_ip() (pygame.math.Vector3 method)
rotate_x_ip_rad() (pygame.math.Vector3 method)
rotate_x_rad() (pygame.math.Vector3 method)
rotate_y() (pygame.math.Vector3 method)
rotate_y_ip() (pygame.math.Vector3 method)
rotate_y_ip_rad() (pygame.math.Vector3 method)
rotate_y_rad() (pygame.math.Vector3 method)
rotate_z() (pygame.math.Vector3 method)
rotate_z_ip() (pygame.math.Vector3 method)
rotate_z_ip_rad() (pygame.math.Vector3 method)
rotate_z_rad() (pygame.math.Vector3 method)
rotation (pygame.freetype.Font attribute)
rotozoom() (in module pygame.transform)
run() (in module pygame.tests)

S

samples() (in module pygame.sndarray)
save() (in module pygame.image)
save_extended() (in module pygame.image)
scalable (pygame.freetype.Font attribute)
scale() (in module pygame.transform)
(pygame.mask.Mask method)
scale2x() (in module pygame.transform)
scale_to_length() (pygame.math.Vector2 method)
(pygame.math.Vector3 method)
scaletest.main() (in module pygame.examples)
scrap_clipboard.main() (in module pygame.examples)
scroll() (pygame.Surface method)
scroll.main() (in module pygame.examples)
SDL (in module pygame.version)
set_allow_screensaver() (in module pygame.display)
set_allowed() (in module pygame.event)

set_alpha() (pygame.Surface method)
 set_at() (pygame.mask.Mask method)
 (pygame.Surface method)
 set_blocked() (in module pygame.event)
 set_bold() (pygame.font.Font method)
 set_caption() (in module pygame.display)
 set_clip() (pygame.sprite.LayeredDirty method)
 (pygame.Surface method)
 set_colorkey() (pygame.Surface method)
 set_controls() (pygame.camera.Camera method)
 set_cursor() (in module pygame.mouse)
 set_default_resolution() (in module pygame.freetype)
 set_endevent() (in module pygame.mixer.music)
 (pygame.mixer.Channel method)
 set_error() (in module pygame)
 set_gamma() (in module pygame.display)
 set_gamma_ramp() (in module pygame.display)
 set_grab() (in module pygame.event)
 set_icon() (in module pygame.display)
 set_instrument() (pygame.midi.Output method)
 set_italic() (pygame.font.Font method)
 set_length() (pygame.Color method)
 set_location() (pygame.Overlay method)
 set_masks() (pygame.Surface method)
 set_mode() (in module pygame.display)
 (in module pygame.scrap)
 set_mods() (in module pygame.key)
 set_num_channels() (in module pygame.mixer)
 set_palette() (in module pygame.display)
 (pygame.Surface method)
 set_palette_at() (pygame.Surface method)
 set_pos() (in module pygame.mixer.music)
 (in module pygame.mouse)
 set_repeat() (in module pygame.key)
 set_reserved() (in module pygame.mixer)
 set_shifts() (pygame.Surface method)
 set_smoothscale_backend() (in module
 pygame.transform)
 set_text_input_rect() (in module pygame.key)
 set_timer() (in module pygame.time)
 set_timing_treshold() (pygame.sprite.LayeredDirty
 method)
 set_underline() (pygame.font.Font method)

set_visible() (in module pygame.mouse)
 set_volume() (in module pygame.mixer.music)
 (pygame.mixer.Channel method)
 (pygame.mixer.Sound method)
 shape (pygame.PixelArray attribute)
 size (pygame.freetype.Font attribute)
 size() (pygame.font.Font method)
 slerp() (pygame.math.Vector2 method)
 (pygame.math.Vector3 method)
 smoothscale() (in module pygame.transform)
 Sound (class in pygame.mixer)
 sound.main() (in module pygame.examples)
 sound_array_demos.main() (in module
 pygame.examples)
 Sprite (class in pygame.sprite)
 spritecollide() (in module pygame.sprite)
 spritecollideany() (in module pygame.sprite)
 sprites() (pygame.sprite.Group method)
 (pygame.sprite.LayeredUpdates method)
 stars.main() (in module pygame.examples)
 start() (pygame.camera.Camera method)
 start_text_input() (in module pygame.key)
 stop() (in module pygame.mixer)
 (in module pygame.mixer.music)
 (pygame.camera.Camera method)
 (pygame.cdrom.CD method)
 (pygame.mixer.Channel method)
 (pygame.mixer.Sound method)
 stop_text_input() (in module pygame.key)
 strength (pygame.freetype.Font attribute)
 strides (pygame.PixelArray attribute)
 strong (pygame.freetype.Font attribute)
 style (pygame.freetype.Font attribute)
 subsurface() (pygame.Surface method)
 Surface (class in pygame)
 surface (pygame.PixelArray attribute)
 surface_to_array() (in module pygame.pixelcopy)
 switch_layer() (pygame.sprite.LayeredUpdates method)
 SysFont() (in module pygame.font)
 (in module pygame.freetype)

testsprite.main() (in module pygame.examples)
textured_polygon() (in module pygame.gfxdraw)
threshold() (in module pygame.transform)
tick() (pygame.time.Clock method)
tick_busy_loop() (pygame.time.Clock method)
time() (in module pygame.midi)
to_surface() (pygame.mask.Mask method)
toggle_fullscreen() (in module pygame.display)
tostring() (in module pygame.image)
transpose() (pygame.PixelArray method)
trigon() (in module pygame.gfxdraw)
type (pygame.cursors.Cursor attribute)
(pygame.event.EventType attribute)

U

ucs4 (pygame.freetype.Font attribute)
underline (pygame.font.Font attribute)
(pygame.freetype.Font attribute)
underline_adjustment (pygame.freetype.Font attribute)
union() (pygame.Rect method)
union_ip() (pygame.Rect method)
unionall() (pygame.Rect method)
unionall_ip() (pygame.Rect method)
unload() (in module pygame.mixer.music)
unlock() (pygame.Surface method)
unmap_rgb() (pygame.Surface method)
unpause() (in module pygame.mixer)
(in module pygame.mixer.music)
(pygame.mixer.Channel method)
update() (in module pygame.display)
(pygame.Color method)
(pygame.math.Vector2 method)
(pygame.math.Vector3 method)
(pygame.Rect method)
(pygame.sprite.Group method)
(pygame.sprite.Sprite method)
use_arraytype() (in module pygame.sndarray)
(in module pygame.surfarray)
use_bitmap_strikes (pygame.freetype.Font attribute)

V

Vector2 (class in pygame.math)

Vector3 (class in pygame.math)
ver (in module pygame.version)
vernum (in module pygame.version)
vertical (pygame.freetype.Font attribute)
vgrade.main() (in module pygame.examples)
vline() (in module pygame.gfxdraw)

W

wait() (in module pygame.event)
(in module pygame.fastevent)
(in module pygame.time)
was_init() (in module pygame.freetype)
wide (pygame.freetype.Font attribute)
write() (pygame.BufferProxy method)
(pygame.midi.Output method)
write_short() (pygame.midi.Output method)
write_sys_ex() (pygame.midi.Output method)

Python Module Index

.

pygame._sdl2.touch

pygame module to work with touch input

pygame.camera

pygame module for camera use

pygame.cdrom

pygame module for audio cdrom control

pygame.cursors

pygame module for cursor resources

pygame.display

pygame module to control the display window and screen

pygame.draw

pygame module for drawing shapes

pygame.event

pygame module for interacting with events and queues

pygame.examples

module of example programs

pygame.fastevent

pygame module for interacting with events and queues from multiple

threads. **pygame.font**

pygame module for loading and rendering fonts

pygame.freetype

Enhanced pygame module for loading and rendering computer fonts

pygame.gfxdraw

pygame module for drawing shapes

pygame.image

pygame module for image transfer

pygame.joystick

Pygame module for interacting with joysticks, gamepads, and trackballs.

pygame.key

pygame module to work with the keyboard

pygame.locals

pygame constants

pygame.mask

pygame module for image masks.

pygame.math

pygame module for vector classes

pygame.midi

pygame module for interacting with midi input and output.

pygame.mixer

pygame module for loading and playing sounds

pygame.mixer.music

pygame module for controlling streamed audio

pygame.mouse

pygame module to work with the mouse

pygame.pixelcopy

pygame module for general pixel array copying

pygame.scrap

pygame module for clipboard support.

pygame.sndarray

pygame module for accessing sound sample data

pygame.sprite

pygame module with basic game object classes

pygame.surfarray

pygame module for accessing surface pixel data using array interfaces

pygame.tests

Pygame unit test suite package

pygame.time

pygame module for monitoring time

pygame.transform

pygame module to transform surfaces

pygame.version

small module containing version information

p

pygame

the top level pygame package