Sense HAT API Reference

LED Matrix

set_rotation

If you're using the Pi upside down or sideways you can use this function to correct the orientation of the image being shown.

Parameter	Type	Valid values	Explanation
`r`	Integer	`0` `90` `180` `270`	The angle to rotate the LED matrix though. `0` is with the Raspberry Pi HDMI port facing downwards.
`redraw`	Boolean	`True` `False`	Whether or not to redraw what is already being displa

Returned type	Explanation
None	

```
from sense_hat import SenseHat

sense = SenseHat()
sense.set_rotation(180)
# alternatives
sense.rotation = 180
```

flip_h

Flips the image on the LED matrix horizontally.

Parameter	Type	Valid values	Explanation
`redraw`	Boolean	`True` `False`	Whether or not to redraw what is already being displa

Returned type	Explanation
List	A list containing 64 smaller lists of `[R, G, B]` pixels (red, green, blue) representing the flipped image.

```
from sense_hat import SenseHat

sense = SenseHat()
sense.flip_h()
```

flip_v

Flips the image on the LED matrix vertically.

Parameter	Type	Valid values	Explanation
`redraw`	Boolean	`True` `False`	Whether or not to redraw what is already being display

Returned type	Explanation
List	A list containing 64 smaller lists of `[R, G, B]` pixels (red, green, blue) representing the flipped image.

```
from sense_hat import SenseHat

sense = SenseHat()
sense.flip_v()
```

set_pixels

Updates the entire LED matrix based on a 64 length list of pixel values.

Parameter	Type	Valid values	Explanation
`pixel_list`	List	`[[R, G, B] * 64]`	A list containing 64 smaller lists of `[R, G, B]` pixels (red, green, blue). Each R-G-B element must be an i

Returned type	Explanation
None	

latest

```
from sense_hat import SenseHat

sense = SenseHat()

X = [255, 0, 0]  # Red
O = [255, 255, 255]  # White

question_mark = [
O, O, O, X, X, O, O, O,
O, O, X, O, O, X, O, O,
O, O, O, O, O, X, O, O,
O, O, O, O, X, O, O, O,
O, O, O, X, O, O, O, O,
O, O, O, X, O, O, O, O,
O, O, O, O, O, O, O, O,
O, O, O, X, O, O, O, O
]

sense.set_pixels(question_mark)
```

get_pixels

Returned type	Explanation
List	A list containing 64 smaller lists of `[R, G, B]` pixels (red, green, blue) representing the currently displayed image.

```
from sense_hat import SenseHat

sense = SenseHat()
pixel_list = sense.get_pixels()
```

Note: You will notice that the pixel values you pass into `set_pixels` sometimes change when you read them back with `get_pixels`. This is because we specify each pixel element as 8 bit numbers (0 to 255) but when they're passed into the Linux frame buffer for the LED matrix the numbers are bit shifted down to fit into RGB 565. 5 bits for red, 6 bits for green and 5 bits for blue. The loss of binary precision when performing this conversion (3 bits lost for red, 2 for green and 3 for blue) accounts for the discrepancies you see.

The `get_pixels` function provides a correct representation of how the pixels end up in frame buffer memory after you've called `set_pixels`.

set_pixel

Sets an individual LED matrix pixel at the specified X-Y coordinate to the specified colour.

Parameter	Type	Valid values	Explanation
`x`	Integer	`0 - 7`	
`y`	Integer	`0 - 7`	0 is at the to

latest

Parameter	Type	Valid values	Explanation
Colour can either be passed as an RGB tuple:			
`pixel`	Tuple or List	`(r, g, b)`	Each elemen
Or three separate values for red, green and blue:			
`r`	Integer	`0 - 255`	The Red eler
`g`	Integer	`0 - 255`	The Green el
`b`	Integer	`0 - 255`	The Blue ele

Returned type	Explanation
None	

```
from sense_hat import SenseHat

sense = SenseHat()

# examples using (x, y, r, g, b)
sense.set_pixel(0, 0, 255, 0, 0)
sense.set_pixel(0, 7, 0, 255, 0)
sense.set_pixel(7, 0, 0, 0, 255)
sense.set_pixel(7, 7, 255, 0, 255)

red = (255, 0, 0)
green = (0, 255, 0)
blue = (0, 0, 255)

# examples using (x, y, pixel)
sense.set_pixel(0, 0, red)
sense.set_pixel(0, 0, green)
sense.set_pixel(0, 0, blue)
```

get_pixel

Parameter	Type	Valid values	Explanation
`x`	Integer	`0 - 7`	0 is on the left, 7 on the right.
`y`	Integer	`0 - 7`	0 is at the top, 7 at the bottom.

Returned type	Explanation
List	Returns a list of `[R, G, B]` representing the colour of an individual LED matrix pixel at the specified X-Y coor

latest

```
from sense_hat import SenseHat

sense = SenseHat()
top_left_pixel = sense.get_pixel(0, 0)
```

Note: Please read the note under `get_pixels`

load_image

Loads an image file, converts it to RGB format and displays it on the LED matrix. The image must be 8 x 8 pixels in size.

Parameter	Type	Valid values	Explanation
`file_path`	String	Any valid file path.	The file system path to the image file to load.
`redraw`	Boolean	`True` `False`	Whether or not to redraw the loaded image file

```
from sense_hat import SenseHat

sense = SenseHat()
sense.load_image("space_invader.png")
```

Returned type	Explanation
List	A list containing 64 smaller lists of `[R, G, B]` pixels (red, green, blue) representing the loaded image after RGB conversion.

```
from sense_hat import SenseHat

sense = SenseHat()
invader_pixels = sense.load_image("space_invader.png", redraw=False)
```

clear

Sets the entire LED matrix to a single colour, defaults to blank / off.

Parameter	Type	Valid values	Expl
`colour`	Tuple or List	`(r, g, b)`	A tup `0, 0)`
Alternatively, the RGB values can be passed individually:			
`r`	Integer	`0 -`	
`g`	Integer	`0 - 255`	The

Parameter	Type	Valid values	Expl
`b`	Integer	`0 - 255`	The

```
from sense_hat import SenseHat
from time import sleep

sense = SenseHat()

red = (255, 0, 0)

sense.clear()  # no arguments defaults to off
sleep(1)
sense.clear(red)  # passing in an RGB tuple
sleep(1)
sense.clear(255, 255, 255)  # passing in r, g and b values of a colour
```

show_message

Scrolls a text message from right to left across the LED matrix and at the specified speed, in the specified colour and background colour.

Parameter	Type	Valid values	Explanation
`text_string`	String	Any text string.	The message to scroll.
`scroll_speed`	Float	Any floating point number.	The speed at which the text should scr
`text_colour`	List	`[R, G, B]`	A list containing the R-G-B (red, green,
`back_colour`	List	`[R, G, B]`	A list containing the R-G-B (red, green, black / off.

Returned type	Explanation
None	

```
from sense_hat import SenseHat

sense = SenseHat()
sense.show_message("One small step for Pi!", text_colour=[255, 0, 0])
```

show_letter

Displays a single text character on the LED matrix.

latest

Parameter	Type	Valid values	Explanation
`s`	String	A text string of length 1.	The letter to show.
`text_colour`	List	`[R, G, B]`	A list containing the R-G-B (red, green, blue `255, 255]` white.
`back_colour`	List	`[R, G, B]`	A list containing the R-G-B (red, green, blue `0, 0]` black / off.

Returned type	Explanation
None	

```
import time
from sense_hat import SenseHat

sense = SenseHat()

for i in reversed(range(0,10)):
    sense.show_letter(str(i))
    time.sleep(1)
```

low_light

Toggles the LED matrix low light mode, useful if the Sense HAT is being used in a dark environment.

```
import time
from sense_hat import SenseHat

sense = SenseHat()
sense.clear(255, 255, 255)
sense.low_light = True
time.sleep(2)
sense.low_light = False
```

gamma

For advanced users. Most users will just need the `low_light` Boolean property above. The Sense HAT python API uses 8 bit (0 to 255) colours for R, G, B. When these are written to the Linux frame buffer they're bit shifted into RGB 5 6 5. The driver then converts them to RGB 5 5 5 before it passes them over to the ATTiny88 AVR for writing to the LEDs.

The gamma property allows you to specify a gamma lookup table for the final 5 bits of colour used. The lookup table is a list of 32 numbers that must be between 0 and 31. The value of the incoming 5 bit colour is used to index the lookup table and the value found at that position is then written to the LEDs.

Type	Valid values	Explanation
Tuple or List	Tuple or List of length 32 containing Integers between 0 and 31	Gamma lookup

```python
import time
from sense_hat import SenseHat

sense = SenseHat()
sense.clear(255, 127, 0)

print(sense.gamma)
time.sleep(2)

sense.gamma = list(reversed(sense.gamma))
print(sense.gamma)
time.sleep(2)

sense.low_light = True
print(sense.gamma)
time.sleep(2)

sense.low_light = False
```

gamma_reset

A function to reset the gamma lookup table to default, ideal if you've been messing with it and want to get it back to a default state.

Returned type	Explanation
None	

```python
import time
from sense_hat import SenseHat

sense = SenseHat()
sense.clear(255, 127, 0)
time.sleep(2)
sense.gamma = [0] * 32  # Will turn the LED matrix off
time.sleep(2)
sense.gamma_reset()
```

Environmental sensors

get_humidity

Gets the percentage of relative humidity from the humidity sensor.

Returned type	Explanation
Float	The percentage of relative humidity.

```
from sense_hat import SenseHat

sense = SenseHat()
humidity = sense.get_humidity()
print("Humidity: %s %%rH" % humidity)

# alternatives
print(sense.humidity)
```

get_temperature

Calls `get_temperature_from_humidity` below.

```
from sense_hat import SenseHat

sense = SenseHat()
temp = sense.get_temperature()
print("Temperature: %s C" % temp)

# alternatives
print(sense.temp)
print(sense.temperature)
```

get_temperature_from_humidity

Gets the current temperature in degrees Celsius from the humidity sensor.

Returned type	Explanation
Float	The current temperature in degrees Celsius.

```
from sense_hat import SenseHat

sense = SenseHat()
temp = sense.get_temperature_from_humidity()
print("Temperature: %s C" % temp)
```

get_temperature_from_pressure

Gets the current temperature in degrees Celsius from the pressure sensor.

Returned type	Explanation
Float	The current temperature in degrees Celsius.

```
from sense_hat import SenseHat

sense = SenseHat()
temp = sense.get_temperature_from_pressure()
print("Temperature: %s C" % temp)
```

get_pressure

Gets the current pressure in Millibars from the pressure sensor.

Returned type	Explanation
Float	The current pressure in Millibars.

```
from sense_hat import SenseHat

sense = SenseHat()
pressure = sense.get_pressure()
print("Pressure: %s Millibars" % pressure)

# alternatives
print(sense.pressure)
```

IMU Sensor

The IMU (inertial measurement unit) sensor is a combination of three sensors, each with an x, y and z axis. For this reason it's considered to be a 9 dof (degrees of freedom) sensor.

- Gyroscope
- Accelerometer
- Magnetometer (compass)

This API allows you to use these sensors in any combination to measure orientation or as individual sensors in their own right.

set_imu_config

Enables and disables the gyroscope, accelerometer and/or magnetometer contribution to the get orientation functions below.

Parameter	Type	Valid values	Explanation
`compass_enabled`	Boolean	`True` `False`	Whether or not the compass sh
`gyro_enabled`	Boolean	`True` `False`	Whether or not the gyroscope should be enabled

latest

Parameter	Type	Valid values	Explanation
`accel_enabled`	Boolean	`True`	Whether or not the accelerometer should be ena

Returned type	Explanation
None	

```
from sense_hat import SenseHat

sense = SenseHat()
sense.set_imu_config(False, True, False)  # gyroscope only
```

get_orientation_radians

Gets the current orientation in radians using the aircraft principal axes of pitch, roll and yaw.

Returned type	Explanation
Dictionary	A dictionary object indexed by the strings `pitch`, `roll` and `yaw`. The values are Floats representing the angle of the axis in radians.

```
from sense_hat import SenseHat

sense = SenseHat()
orientation_rad = sense.get_orientation_radians()
print("p: {pitch}, r: {roll}, y: {yaw}".format(**orientation_rad))

# alternatives
print(sense.orientation_radians)
```

get_orientation_degrees

Gets the current orientation in degrees using the aircraft principal axes of pitch, roll and yaw.

Returned type	Explanation
Dictionary	A dictionary object indexed by the strings `pitch`, `roll` and `yaw`. The values are Floats representing the angle of the axis in degrees.

```
from sense_hat import SenseHat

sense = SenseHat()
orientation = sense.get_orientation_degrees()
print("p: {pitch}, r: {roll}, y: {yaw}".format(**orientation))
```

latest

get_orientation

Calls `get_orientation_degrees` above.

```
from sense_hat import SenseHat

sense = SenseHat()
orientation = sense.get_orientation()
print("p: {pitch}, r: {roll}, y: {yaw}".format(**orientation))

# alternatives
print(sense.orientation)
```

get_compass

Calls `set_imu_config` to disable the gyroscope and accelerometer then gets the direction of North from the magnetometer in degrees.

Returned type	Explanation
Float	The direction of North.

```
from sense_hat import SenseHat

sense = SenseHat()
north = sense.get_compass()
print("North: %s" % north)

# alternatives
print(sense.compass)
```

get_compass_raw

Gets the raw x, y and z axis magnetometer data.

Returned type	Explanation
Dictionary	A dictionary object indexed by the strings `x`, `y` and `z`. The values are Floats representing the magnetic intensity of the axis in m

```
from sense_hat import SenseHat

sense = SenseHat()
raw = sense.get_compass_raw()
print("x: {x}, y: {y}, z: {z}".format(**raw))

# alternatives
print(sense.compass_raw)
```

latest

get_gyroscope

Calls `set_imu_config` to disable the magnetometer and accelerometer then gets the current orientation from the gyroscope only.

Returned type	Explanation
Dictionary	A dictionary object indexed by the strings `pitch`, `roll` and `yaw`. The values are Floats representing the angle of the axis in degrees.

```
from sense_hat import SenseHat

sense = SenseHat()
gyro_only = sense.get_gyroscope()
print("p: {pitch}, r: {roll}, y: {yaw}".format(**gyro_only))

# alternatives
print(sense.gyro)
print(sense.gyroscope)
```

get_gyroscope_raw

Gets the raw x, y and z axis gyroscope data.

Returned type	Explanation
Dictionary	A dictionary object indexed by the strings `x`, `y` and `z`. The values are Floats representing the rotational intensity of the axis in ra

```
from sense_hat import SenseHat

sense = SenseHat()
raw = sense.get_gyroscope_raw()
print("x: {x}, y: {y}, z: {z}".format(**raw))

# alternatives
print(sense.gyro_raw)
print(sense.gyroscope_raw)
```

get_accelerometer

Calls `set_imu_config` to disable the magnetometer and gyroscope then gets the current orientation from the accelerometer only.

Returned type	Explanation
Dictionary	A dictionary object indexed by the strings `pitch`, `roll` and `yaw`. The values are Floats representing the angle of the axis in degrees.

```
from sense_hat import SenseHat

sense = SenseHat()
accel_only = sense.get_accelerometer()
print("p: {pitch}, r: {roll}, y: {yaw}".format(**accel_only))

# alternatives
print(sense.accel)
print(sense.accelerometer)
```

get_accelerometer_raw

Gets the raw x, y and z axis accelerometer data.

Returned type	Explanation
Dictionary	A dictionary object indexed by the strings `x`, `y` and `z`. The values are Floats representing the acceleration intensity of the axis ir

```
from sense_hat import SenseHat

sense = SenseHat()
raw = sense.get_accelerometer_raw()
print("x: {x}, y: {y}, z: {z}".format(**raw))

# alternatives
print(sense.accel_raw)
print(sense.accelerometer_raw)
```

Joystick

InputEvent

A tuple describing a joystick event. Contains three named parameters:

- `timestamp` - The time at which the event occurred, as a fractional number of seconds (the same format as the built-in `time` function)
- `direction` - The direction the joystick was moved, as a string (`"up"`, `"down"`, `"left"`, `"right"`, `"middle"`)
- `action` - The action that occurred, as a string (`"pressed"`, `"released"`, `"held"`)

This tuple type is used by several joystick methods either as the return type or the type of a parameter.

wait_for_event

Blocks execution until a joystick event occurs, then returns an `InputEvent` representing the event that occurred.

```
from sense_hat import SenseHat
from time import sleep

sense = SenseHat()
event = sense.stick.wait_for_event()
print("The joystick was {} {}".format(event.action, event.direction))
sleep(0.1)
event = sense.stick.wait_for_event()
print("The joystick was {} {}".format(event.action, event.direction))
```

In the above example, if you briefly push the joystick in a single direction you should see two events output: a pressed action and a released action. The optional *emptybuffer* can be used to flush any pending events before waiting for new events. Try the following script to see the difference:

```
from sense_hat import SenseHat
from time import sleep

sense = SenseHat()
event = sense.stick.wait_for_event()
print("The joystick was {} {}".format(event.action, event.direction))
sleep(0.1)
event = sense.stick.wait_for_event(emptybuffer=True)
print("The joystick was {} {}".format(event.action, event.direction))
```

---

get_events

Returns a list of `InputEvent` tuples representing all events that have occurred since the last call to `get_events` or `wait_for_event`.

```
from sense_hat import SenseHat

sense = SenseHat()
while True:
    for event in sense.stick.get_events():
        print("The joystick was {} {}".format(event.action, event.direction))
```

---

direction_up, direction_left, direction_right, direction_down, direction_middle, direction_any

These attributes can be assigned a function which will be called whenever the joystick is pushed in the associated direction (or in any direction in the case of `directio` latest
function assigned must either take no parameters or must take a single param[illegible] which will be passed the associated `InputEvent`.

```
from sense_hat import SenseHat, ACTION_PRESSED, ACTION_HELD, ACTION_RELEASED
from signal import pause

x = 3
y = 3
sense = SenseHat()

def clamp(value, min_value=0, max_value=7):
    return min(max_value, max(min_value, value))

def pushed_up(event):
    global y
    if event.action != ACTION_RELEASED:
        y = clamp(y - 1)

def pushed_down(event):
    global y
    if event.action != ACTION_RELEASED:
        y = clamp(y + 1)

def pushed_left(event):
    global x
    if event.action != ACTION_RELEASED:
        x = clamp(x - 1)

def pushed_right(event):
    global x
    if event.action != ACTION_RELEASED:
        x = clamp(x + 1)

def refresh():
    sense.clear()
    sense.set_pixel(x, y, 255, 255, 255)

sense.stick.direction_up = pushed_up
sense.stick.direction_down = pushed_down
sense.stick.direction_left = pushed_left
sense.stick.direction_right = pushed_right
sense.stick.direction_any = refresh
refresh()
pause()
```

Note that the `direction_any` event is always called *after* all other events making it an ideal hook for things like display refreshing (as in the example above).

Light and colour sensor

The v2 Sense HAT includes a TCS34725 colour sensor that is capable of measuring the amount of Red, Green and Blue (RGB) in the incident light, as well as providing a Clear light (brightness) reading.

You can interact with the colour sensor through the `colour` (or `color`) attribute of the Sense HAT, which corresponds to a `ColourSensor` object.

The example below serves as an overview of how the colour sensor can be used, while the sections that follow provide additional details and explanations.

```
from sense_hat import SenseHat
from time import sleep

sense = SenseHat()
sense.color.gain = 4
sense.color.integration_cycles = 64

while True:
    sleep(2 * sense.colour.integration_time)
    red, green, blue, clear = sense.colour.colour # readings scaled to 0-256
    print(f"R: {red}, G: {green}, B: {blue}, C: {clear}")
```

Obtaining RGB and Clear light readings

The `colour` (or `color`) property of the `ColourSensor` object is a 4-tuple containing the measured values for Red, Green and Blue (RGB), along with a Clear light value, which is a measure of brightness. Individual colour and light readings can also be obtained through the `red`, `green`, `blue` and `clear` properties of the `ColourSensor` object.

`ColourSensor` property	Returned type	Explanation
`red`	int	The amount of incident red light, scaled to 0-256
`green`	int	The amount of incident green light, scaled to 0-256
`blue`	int	The amount of incident blue light, scaled to 0-256
`clear`	int	The amount of incident light (brightness), scaled to 0-256
`colour`	tuple	A 4-tuple containing the RGBC (Red, Green, Blue and Clear) se

These are all read-only properties; they cannot be set.

Note that, in the current implementation, the four values accessed through the `colour` property are retrieved through a single sensor reading. Obtaining these values through the `red`, `green`, `blue` and `clear` properties would require four separate readings.

Gain

In sensors, the term "gain" can be understood as being synonymous to *sensitivity*. A higher gain setting means the output values will be greater for the same input.

There are four possible gain values for the colour sensor: `1`, `4`, `16` and `60`, with the default value being `1`. You can get or set the sensor gain through the `gain` p `ColourSensor` object. An attempt to set the gain to a value that is not valid wil `InvalidGainError` exception being raised.

latest

```
from sense_hat import SenseHAT
from time import sleep

sense = SenseHat()
sense.colour.gain = 1
sleep(1)
print(f"Gain: {sense.colour.gain}")
print(f"RGBC: {sense.colour.colour}")

sense.colour.gain = 16
sleep(1)
print(f"Gain: {sense.colour.gain}")
print(f"RGBC: {sense.colour.colour}")
```

Under the same lighting conditions, the RGBC values should be considerably higher when the gain setting is increased.

When there is very little ambient light and the RGBC values are low, it makes sense to use a higher gain setting. Conversely, when there is too much light and the RGBC values are maximal, the sensor is saturated and the gain should be set to lower values.

Integration cycles and the interval between measurements

You can specify the number of *integration cycles* required to generate a new set of sensor readings. Each integration cycle is 2.4 milliseconds long, so the number of integration cycles determines the *minimum* amount of time required between consecutive readings.

You can set the number of integration cycles to any integer between `1` and `256`, through the `integration_cycles` property of the `ColourSensor` object. The default value is `1`. An attempt to set the number of integration cycles to a value that is not valid will result in a `InvalidIntegrationCyclesError` or `TypeError` exception being raised.

```
from sense_hat import SenseHAT
from time import sleep

sense = SenseHat()
sense.colour.integration_cycles = 100
print(f"Integration cycles: {sense.colour.integration_cycles}")
print(f"Minimum wait time between measurements: {sense.colour.integration_time} seconds")
```

Integration cycles and raw values

The values of the `colour`, `red`, `green`, `blue` and `clear` properties are integers between 0 and 256. However, these are not the actual *raw* values obtained from the sensor; they have been scaled down to this range for convenience.

The range of the raw values depends on the number of integration cycles:

`integration_cycles`	maximum raw value (`max_raw`)
1 - 64	1024 * `integration_cycles`
> 64	65536

What this really means is that the *accuracy* of the sensor is affected by the number of integration cycles, i.e. the time required by the sensor to obtain a reading. A longer integration time will result in more reliable readings that fall into a wider range of values, being able to more accurately distinguish between similar lighting conditions.

The following properties of the `ColourSensor` object provide direct access to the raw values measured by the sensor.

`ColourSensor` property	Returned type	Explanation
`red_raw`	int	The amount of incident red light, between 0 and `max_raw`
`green_raw`	int	The amount of incident green light, between 0 and `max_raw`
`blue_raw`	int	The amount of incident blue light, between 0 and `max_raw`
`clear_raw`	int	The amount of incident light (brightness), between 0 and `max_r`
`colour_raw`	tuple	A 4-tuple containing the RGBC (Red, Green, Blue and Clear) ra
`rgb`	tuple	A 3-tuple containing the RGB raw sensor readings, each betwe
`brightness`	int	An alias to the `clear_raw` property - the amount of incident lig

Here is an example comparing raw values to the corresponding scaled ones, for a given number of integration cycles.

```
from sense_hat import SenseHAT
from time import sleep

sense = SenseHat()
sense.colour.integration_cycles = 64
print(f"Minimum time between readings: {sense.colour.integration_time} seconds")
print(f"Maximum raw sensor reading: {sense.colour.max_raw}")
sleep(sense.colour.integration_time + 0.1)  # try omitting this
print(f"Current raw sensor readings: {sense.colour.colour_raw}")
print(f"Scaled values: {sense.colour.colour}")
```

Exceptions

Custom Sense HAT exceptions are statically defined in the `sense_hat.exceptions` module. The exceptions relate to problems encountered while initialising the colour chip o latest invalid parameters. Each exception includes a message describing the issue encountered, and is subclassed from the base class `SenseHatException`.