nanoleafapi

Release 2.1.0

Jul 31, 2022

Contents:

CHAPTER 1

Getting Started

1.1 Installation

To install run:

1.2 Prerequisites

You must know the IP address of the Nanoleaf device. This can be either be done using your own methods or by using the disovery module. This module uses SSDP and should work __but__ I have found cases of this method not functioning properly. If it doesn't work, and gives an empty dictionary please identify the IP of the Nanoleaf device yourself.

To use the discovery module:

```
from nanoleafapi import discovery
nanoleaf_dict = discovery.discover_devices(timeout=30)
```

This will return a dictionary in the format: `{name: ip}`.

1.3 Usage

There is just one class that contains all relevant functions for controlling the lights. To get started:

```
from nanoleafapi import Nanoleaf
```

Next, a Nanoleaf object can be created with the following section of code. IF you don't have an authentication token yet, hold the power button for 5-7 seconds on your Nanoleaf device before running the following code. This will generate a new token and save it to your user directory to use for future uses of this package.

```
nl = Nanoleaf("ip")
```

You can now use the commands to control the panels as displayed in the example below.

```
nl.toggle_power()
nl.set_color((255, 0, 0))            # Set colour to red
```

CHAPTER 2

Nanoleaf Methods

All of the following methods can be called with the Nanoleaf object you created.

For more information about the Nanoleaf API: https://forum.nanoleaf.me/docs/openapi

For more in-depth documentation about this package visit: https://nanoleafapi.readthedocs.io/en/latest/api.html

2.1 User Management

```
generate_auth_token()      # Generates new authentication token (hold power for 5-7␣
↪before running)
delete_user(auth_token)    # Deletes an authentication token from the device
```

2.2 Power

```
get_power()                # Returns True if lights are on, otherwise False
power_off()                # Powers off the lights
power_on()                 # Powers on the lights
toggle_power()             # Toggles light on/off
```

2.3 Colour

Colours are generated using HSV (or HSB) in the API, and these individual values can be adjusted using methods which are as described, hue, brightness and saturation. The method in this section uses RGB (0-255) and converts this to HSV.

There are already some pre-set colours which can be imported to be used with the `set_color()` method:

```
from nanoleafapi import RED, ORANGE, YELLOW, GREEN, LIGHT_BLUE, BLUE, PINK, PURPLE,␣
↪WHITE
```

The `set_color()` method can then be called, passing in either a pre-set colour or your own RGB colour in the form of a tuple: `(r, g, b)`.

```
set_color((r, g, b))       # Set all lights to RGB colour. Pass the colour as a tuple.
set_color(RED)             # Same result but using a pre-set colour.
```

2.4 Brightness

```
set_brightness(brightness, duration)     # Sets the brightness of the lights (accepts␣
↪values between 0-100)
increment_brightness(value)              # Increments the brightness by set amount␣
↪(can also be negative)
get_brightness()                         # Returns current brightness
```

2.5 Hue

Use these if you want to change the HSV values manually, otherwise use `set_color()` for colour change using RGB.

```
set_hue(value)             # Sets the hue of the lights (accepts values between 0-360)
increment_hue(value)       # Increments the hue by set amount (can also be negative)
get_hue()                  # Returns current hue
```

2.6 Saturation

Use these if you want to change the HSV values manually, otherwise use `set_color()` for colour change using RGB.

```
set_saturation(value)            # Sets the saturation of the lights (accepts value␣
↪between 0-100)
increment_saturation(value)      # Increments the saturation by set amount (can also␣
↪be negative)
get_saturation()                 # Returns current saturation
```

2.7 Identify

This is usually used to identify the current lights by flashing them on and off.

```
identify()
```

2.8 Colour Temperature

```
set_color_temp(value)              # Sets the colour temperature of the lights (accepts␣
↪between 1200-6500)
increment_color_temp(value)        # Increments the colour temperature by set amount␣
↪(can also be negative)
get_color_temp()                   # Returns current colour temperature
```

2.9 Colour Mode

Not really sure what this is for, but included it anyway.

```
get_color_mode()       # Returns current colour mode
```

2.10 Effects

```
get_current_effect()     # Returns either name of current effect if available or␣
↪*Solid*/*Static*/*Dynamic*.
list_effects()           # Returns a list of names of all available effects.
effect_exists(name)      # Helper method which determines whether the given string␣
↪exists as an effect.
set_effect(name)         # Sets the current effect.
```

2.11 Write Effect

```
write_effect(effect_dict)     # Sets a user-created effect.
```

Writing effects is rather complicated; you need to follow the the exact format for the effect dictionary, which can be found here: https://forum.nanoleaf.me/docs/openapi#_u2t4jzmkp8nt

In future updates, I hope to add a way to make this process easier, but for now an example of a valid effect dictionary is provided below:

```
effect_data = {
            "command": "display",
            "animName": "New animation",
            "animType": "random",
            "colorType": "HSB",
            "animData": None,
            "palette": [
                {
                    "hue": 0,
                    "saturation": 100,
                    "brightness": 100
                },
                {
                    "hue": 120,
                    "saturation": 100,
                    "brightness": 100
```

(continues on next page)

(continued from previous page)

```
                },
                {
                    "hue": 180,
                    "saturation": 100,
                    "brightness": 100
                }
            ],
            "brightnessRange": {
                "minValue": 50,
                "maxValue": 100
            },
            "transTime": {
                "minValue": 50,
                "maxValue": 100
            },
            "delayTime": {
                "minValue": 50,
                "maxValue": 100
            },
            "loop": True
        }
```

Inputting an invalid dictionary will result in the function returning False, and it printing to the console *Invalid effect dictionary!*.

2.12 Events

Creates an event listener for the different types of events.

```
register_event(function, event_types)
```

You should pass your own function with one argument (event as a dictionary). This function will run every time a new event is received.

IMPORTANT: You cannot currently call `register_event()` more than **once** due to API limitations. Instead, distinguish between the events in your function using the dictionary data.

A list of event types you would like to listen for should also be passed. You can register up to 4 events (all of them), and these are listed below:

Event IDs:

State (changes in power/brightness): **1**
Layout: **2**
Effects: **3**
Touch (canvas only): **4**

2.12.1 Example Usage

```
def event_function(event):
    print(event)
```

(continues on next page)

(continued from previous page)

```
# Register for all events
nl.register_event(event_function, [1, 2, 3, 4])
```

2.12.2 Example Output

When an event occurs, the `event_function()` will run and therefore in this case, print the event dictionary.

```
{"events":[{"attr":2,"value":65}]}                  # Example of state event (1)
{"events":[{"attr":1,"value":"Falling Whites"}]}    # Example of effects event (3)
{"events":[{"panelId":7397,"gesture":0}]}           # Example of touch event (4)
```

CHAPTER 3

NanoleafDigitalTwin Methods

This class is used to make a digital twin (or copy) of the Nanoleaf device, allowing you to change the colour of individual tiles and then sync all the changes at once to the real device.

To create an instance of this class, you must initialise it with a Nanoleaf object:

```
from nanoleafapi import Nanoleaf, NanoleafDigitalTwin

nl = Nanoleaf("192.168.0.2")
digital_twin = NanoleafDigitalTwin(nl)
```

3.1 Utility

```
get_ids()       # Returns a list of panel IDs
```

3.2 Colour

Setting the colour is all managed by using an RGB tuple, in the format: `(R, G, B)`.

```
set_color(panel_id, (255, 255, 255))    # Sets the panel with specified ID to white
set_all_colors((255, 255, 255))         # Sets all panels to white
get_color(panel_id)                     # Gets the colour of a specified panel
get_all_colors()                        # Returns a dictionary of {panel_id: (R, G, B)}
```

3.3 Sync

The sync method applies the changes to the real Nanoleaf device, based on the changes made here.

```
sync()     # Syncs with the real Nanoleaf counterpart
```

CHAPTER 4

Errors

```
NanoleafRegistrationError()    # Raised when token generation mode not active on device
NanoleafConnectionError()      # Raised when there is a connection error during check_
↪connection() method
NanoleafEffectCreationError() # Raised when there is an error with an effect␣
↪dictionary/method arguments
```

CHAPTER 5

Detailed Documentation

5.1 Nanoleaf

nanoleafapi

This module is a Python 3 wrapper for the Nanoleaf OpenAPI. It provides an easy way to use many of the functions available in the API. It supports the Light Panels (previously Aurora), Canvas and Shapes (including Hexgaons).

class nanoleaf.**Nanoleaf**(*ip: str*, *auth_token: str = None*, *print_errors: bool = False*)

The Nanoleaf class for controlling the Light Panels and Canvas

Variables

- **ip** – IP of the Nanoleaf device
- **url** – The base URL for requests
- **auth_token** – The authentication token for the API
- **print_errors** – True for errors to be shown, otherwise False

check_connection() → None

Ensures there is a valid connection

create_auth_token() → Optional[str]

Creates or retrives the device authentication token

The power button on the device should be held for 5-7 seconds, then this method should be run. This will set both the auth_token and url instance variables, and save the token in a file for future instances of the Nanoleaf object.

Returns Token if successful, None if not.

delete_auth_token(*auth_token: str*) → bool

Deletes an authentication token

Deletes an authentication token and the .nanoleaf_token file if it contains the auth token to delete. This token can no longer be used as part of an API call to control the device. If required, generate a new one using create_auth_token().

Parameters **auth_token** – The authentication token to delete.

Returns True if successful, otherwise False

effect_exists(*effect_name: str*) → bool
Verifies whether an effect exists

Parameters **effect_name** – Name of the effect to verify

Returns True if effect exists, otherwise False

enable_extcontrol() → bool
Enables the extControl UDP streaming mode

Returns True if successful, otherwise False

flow(*rgb_list: List[Tuple[int, int, int]], speed: float = 1*) → bool
Displays a sequence of specified colours on the device.

Parameters

- **rgb** – A list of tuples containing RGB colours to flow between in the format (r, g, b).
- **speed** – The speed of the transition between colours in seconds, with a maximum of 1 decimal place.

Raises ***NanoleafEffectCreationError*** – When an invalid rgb_list is provided.

Returns True if the effect was created and displayed successfully, otherwise False

get_auth_token() → Optional[str]
Returns the current auth token or None

get_brightness() → int
Returns the current brightness value of the lights

get_color_mode() → str
Returns the colour mode of the lights

get_color_temp() → int
Returns the current colour temperature of the lights

get_current_effect() → str
Returns the currently selected effect

If the name of the effect isn't available, this will return *Solid*, *Dynamic* or *Static* instead.

Returns Name of the effect or type if unavailable.

static get_custom_base_effect(*anim_type: str = 'custom', loop: bool = True*) → Dict[str, Any]
Returns base custom effect dictionary

get_hue() → int
Returns the current hue value of the lights

get_ids() → List[int]
Returns a list of all device ids

get_info() → Dict[str, Any]
Returns a dictionary of device information

get_layout() → Dict[str, Any]
Returns the device layout information

get_name() → str
Returns the name of the current device

get_power() → bool
Returns the power status of the lights

Returns True if on, False if off

get_saturation() → int
Returns the current saturation value of the lights

identify() → bool
Runs the identify sequence on the lights

Returns True if successful, otherwise False

increment_brightness(*brightness: int*) → bool
Increments the brightness of the lights

Parameters **brightness** – How much to increment the brightness, can also be negative

Returns True if successful, otherwise False

increment_color_temp(*value: int*) → bool
Sets the white colour temperature of the lights

Parameters **value** – How much to increment the colour temperature by, can also be negative.

Returns True if successful, otherwise False

increment_hue(*value: int*) → bool
Increments the hue of the lights

Parameters **value** – How much to increment the hue, can also be negative

Returns True if successful, otherwise False

increment_saturation(*value: int*) → bool
Increments the saturation of the lights

Parameters **brightness** – How much to increment the saturation, can also be negative.

Returns True if successful, otherwise False

list_effects() → List[str]
Returns a list of available effects

power_off() → bool
Powers off the lights

Returns True if successful, otherwise False

power_on() → bool
Powers on the lights

Returns True if successful, otherwise False

pulsate(*rgb: Tuple[int, int, int], speed: float = 1*) → bool
Displays a pulsating effect on the device with two colours

Parameters

- **rgb** – A tuple containing the RGB colour to pulsate in the format (r, g, b).
- **speed** – The speed of the transition between colours in seconds, with a maximum of 1 decimal place.

Raises ***NanoleafEffectCreationError*** – When an invalid rgb value is provided.

Returns True if the effect was created and displayed successfully, otherwise False

register_event (*func: Callable[[Dict[str, Any]], Any], event_types: List[int]*) → None
Starts a thread to register and listen for events

Creates an event listener. This method can only be called once per program run due to API limitations.

Parameters

- **func** – The function to run when an event is recieved (this should be defined by the user with one argument). This function will recieve the event as a dictionary.
- **event_types** – A list containing up to 4 numbers from 1-4 corresponding to the relevant events to be registered for. 1 = state (power/brightness), 2 = layout, 3 = effects, 4 = touch (Canvas only)

set_brightness (*brightness: int*, *duration: int = 0*) → bool
Sets the brightness of the lights

Parameters

- **brightness** – The required brightness (between 0 and 100)
- **duration** – The duration over which to change the brightness

Returns True if successful, otherwise False

set_color (*rgb: Tuple[int, int, int]*) → bool
Sets the colour of the lights

Parameters **rgb** – Tuple in the format (r, g, b)

Returns True if successful, otherwise False

set_color_temp (*value: int*) → bool
Sets the white colour temperature of the lights

Parameters **value** – The required colour temperature (between 0 and 100)

Returns True if successful, otherwise False

set_effect (*effect_name: str*) → bool
Sets the effect of the lights

Parameters **effect_name** – The name of the effect

Returns True if successful, otherwise False

set_hue (*value: int*) → bool
Sets the hue of the lights

Parameters **value** – The required hue (between 0 and 360)

Returns True if successful, otherwise False

set_saturation (*value: int*) → bool
Sets the saturation of the lights

Parameters **value** – The required saturation (between 0 and 100)

Returns True if successful, otherwise False

spectrum (*speed: float = 1*) → bool
Displays a spectrum cycling effect on the device

Parameters **speed** – The speed of the transition between colours in seconds, with a maximum of 1 decimal place.

Returns True if the effect was created and displayed successfully, otherwise False

toggle_power() → bool
Toggles the lights on/off

write_effect(*effect_dict: Dict[str, Any]*) → bool
Writes a user-defined effect to the panels

Parameters **effect_dict** – The effect dictionary in the format described here: https://forum.nanoleaf.me/docs/openapi#_u2t4jzmkp8nt

Raises ***NanoleafEffectCreationError*** – When invalid effect dictionary is provided.

Returns True if successful, otherwise False

exception nanoleaf.**NanoleafConnectionError**
Raised when the connection to the Nanoleaf device fails.

exception nanoleaf.**NanoleafEffectCreationError**
Raised when one of the custom effects creation has incorrect arguments.

exception nanoleaf.**NanoleafRegistrationError**
Raised when an issue during device registration.

5.2 Discovery

discovery

Module to aid with Nanoleaf discovery on a network.

discovery.**discover_devices**(*timeout: int = 30*, *debug: bool = False*) → Dict[Optional[str], str]
Discovers Nanoleaf devices on the network using SSDP

Parameters

- **timeout** – The timeout on the search in seconds (default 30)
- **debug** – Prints each device string for the SSDP discovery

Returns Dictionary of found devices in format {name: ip}

5.3 Digital Twin

NanoleafDigitalTwin

This module allows for the creation of a "digital twin", allowing you to make changes to individual panels and sync them to their real counterparts.

class digital_twin.**NanoleafDigitalTwin**(*nl: nanoleafapi.nanoleaf.Nanoleaf*)
Class for creating and modifying digital twins

Variables

- ***nanoleaf*** – The Nanoleaf object
- **tile_dict** – The dictionary of tiles and their associated colour

get_all_colors() → Dict[int, Tuple[int, int, int]]
Returns a dictionary of all panel IDs and associated colours.

Returns Dictionary with panel IDs as keys and RGB tuples as values.

get_color(*panel_id: int*) → Tuple[int, int, int]
Returns the colour of a specified panel.

Parameters **panel_id** – The panel to get the colour of.

Returns Returns the RGB tuple of the panel with ID panel_id.

get_ids() → List[int]
Returns a list of panel IDs.

Returns List of panel IDs.

set_all_colors(*rgb: Tuple[int, int, int]*) → None
Sets the colour of all the panels.

Parameters **rgb** – A tuple containing the RGB values of the colour to set

set_color(*panel_id: int, rgb: Tuple[int, int, int]*) → None
Sets the colour of an individual panel.

Parameters

- **panel_id** – The ID of the panel to change the colour of
- **rgb** – A tuple containing the RGB values of the colour to set

sync() → bool
Syncs the digital twin's changes to the real Nanoleaf device.

Returns True if success, otherwise False

Python Module Index

d

n

Index

T

W