# LEGO® Education Python API & Web IDE



## Contents


- Introduction
- Connect and Run
- Single Motor
- Double Motor
- Color Sensor
- Controller
- Combine Single Motor and Color Sensor
- Combine Double Motor and Controller
- Constants
- Function Descriptions


---

# Welcome

Welcome to the LEGO® Education Python API! This library allows users to write Python code to control the LEGO Education Hardware found in the LEGO Education Computer Science & AI kits. Some design choices favor simplicity over strict Python conventions to make the API easier to use in education contexts.

# Supported Hardware

You can use the following motors and sensors with the LEGO Education Python API:

- **Single Motor:** Control and monitor direction, speed, absolute/relative position, power, acceleration, deceleration, and end states.
- **Double Motor:** Synchronized or independent control and monitoring of two connected motors (left and right), including direction, speed, absolute/relative position, power, acceleration, deceleration, and end states. Additionally includes a 6-axis IMU for orientation and motion data.
- **Color Sensor:** Access detected color as LEGO Color Number, RGB raw, HSV, and reflection readings.
- **Controller:** Access dual lever input (left and right) with angle and percentage positional readings.

# Updating the LEGO Education Hardware:

To update your LEGO Education hardware, visit the LEGO Education Coding Canvas and connect each hardware device you plan to code with. If an update is available, the hardware will update automatically when connected to a device with internet connectivity.

# Running Programs

Open the LEGO Education Python Web IDE in your browser. You can open your own Python file, or select an existing one from the "Examples" dropdown.

To execute a program, press the green "Run" button on the top left of the IDE. Make sure the LEGO Education motor(s) and sensor(s) you are using are charged, powered on, and broadcasting.

# Troubleshooting

- **Firmware errors:** Ensure your LEGO Education hardware's firmware is on the latest version. See LEGO Education's FAQ for information on updating hardware.
- **Connection issues:** Confirm your Bluetooth setup and verify device compatibility with your OS.

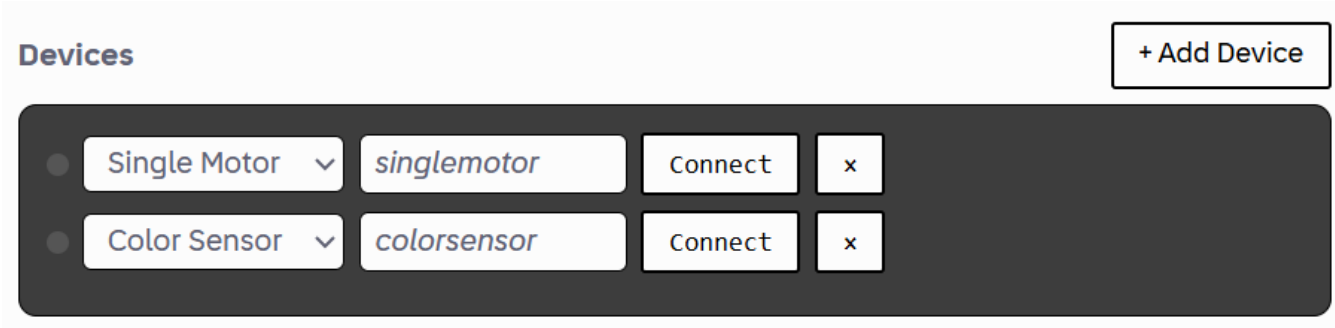
---

## Connect and Run

In the LEGO Education Python Web IDE there are two ways to connect to your hardware.

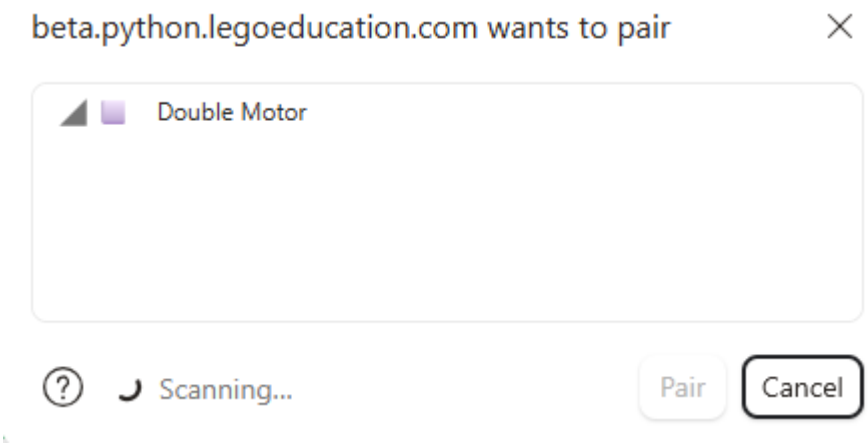
## Method 1

The first is using the "Devices" section in the top right of the IDE.

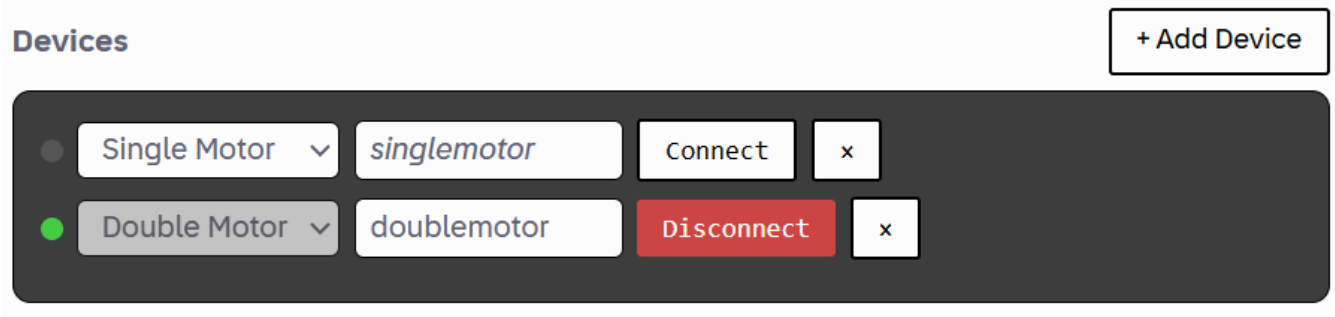



1. Click the "+ Add Device" button in the top right
2. Select your hardware from the dropdown on the left side of the list
3. Click the "Connect" button and your browser should open a pop-up for you to pair your hardware

    - If you do not see your hardware on the list, make sure the light on your hardware is blinking. You need to press the light/button on the hardware twice to allow it to pair.




The "Devices" section should now look something like this, indicating that the Double Motor is connected. You should follow the same steps for each device you are using.




An example of a script where you are meant to use this method can be found by clicking "Examples" and selecting "Supplies from the Skies".

### Troubleshooting

If the hardware quickly disconnects itself after you go through the three steps, it is likely your firmware is not on the latest version.

To update your LEGO Education hardware, visit the LEGO Education Coding Canvas and connect each hardware device you plan to code with. If an update is available, the hardware will update automatically when connected to a device with internet connectivity.

## Method 2

The second method does not use the GUI in the IDE, but will work in both the IDE and in your own local environment (outside the browser).

1. Import the `legoeducation` library
2. Define a variable for the hardware
3. Connect to the hardware
4. Check if connected
5. Interact with the hardware
6. Disconnect from the hardware

An example of the full Connect and Run process, for a Single Motor, can be found by clicking "Examples" and selecting "Single Motor".

### 1. Import the LEGO Education Python API

To use the LEGO Education Python API in your Python code, you need to import the library. Use the `le` alias for easily referencing it later.

```
import legoeducation as le
```

### 2. Define a local variable

Define a local variable for the type of hardware you are using (example: Single Motor).

```
singlemotor = le.SingleMotor()
```

Examples for the four types of LEGO Education hardware:

- ```
  singlemotor = le.SingleMotor()
  ```
- ```
  doublemotor = le.DoubleMotor()
  ```
- ```
  colorsensor = le.ColorSensor()
  ```
- ```
  controller = le.Controller()
  ```

### 3. Connect

Make sure the LEGO Education motor(s) and sensor(s) you are using are charged, powered on, and in Bluetooth broadcasting mode.

By default, the `connect()` function will connect to the first found. In environments with multiple hardware broadcasting simultaneously, to ensure connection to the right one, use a Connection Card and filter by specifying the connection card color and serial number when connecting.

For example, to connect to a Azure connection card with serial number '3683':

```
singlemotor.connect(card_color=le.LEGO_COLOR_AZURE, card_serial="3683")
```




The Connection Card colors, from left-to-right in the image, are:

1. `le.LEGO_COLOR_GREEN`
2. `le.LEGO_COLOR_BLUE`
3. `le.LEGO_COLOR_RED`
4. `le.LEGO_COLOR_ORANGE`
5. `le.LEGO_COLOR_YELLOW`
6. `le.LEGO_COLOR_AZURE`
7. `le.LEGO_COLOR_PURPLE`
8. `le.LEGO_COLOR_MAGENTA`

You can create as many LEGO Education hardware connections as you like, but this may affect the responsiveness.

## 4. Check if connected

The `connected` Boolean can be used to check if the connection attempt was successful or not. For instance, if no hardware was connected, then you could exit the program.

```
if not singlemotor.connected:
    print('Error connecting to Single Motor.')
    exit(1) # error connecting
```

## 5. Interact

Once successfully connected, you can use the LEGO Education Python API to interact with the hardware. See the following examples for using the different LEGO Education hardware:

- Single Motor
- Double Motor
- Color Sensor
- Controller

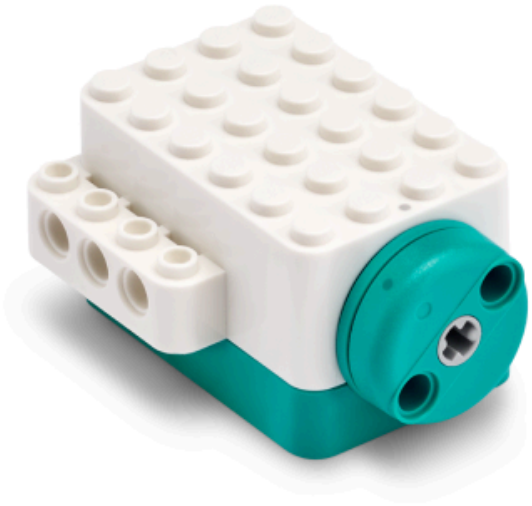
## 6. Disconnect

When done interacting, disconnect from the device with the `disconnect()` function. The hardware will disconnect from your computer and return to broadcast mode (ready to connect again).

```
singlemotor.disconnect()
```

---

## Single Motor



The Single Motor allows precise control and monitoring of a single motor.

# Controlling the Motor

Here is an example for running the single motor for 180-degrees.

```
import legoeducation as le

# update these values to match the Connection Card
card_color = le.LEGO_COLOR_AZURE
card_serial = '3683'

# Connect to the Single Motor
singlemotor = le.SingleMotor()
singlemotor.connect(card_color=card_color, card_serial=card_serial)

# Check connection
if not singlemotor.connected:
    print('Error connecting to Single Motor.')
    exit(1) # error connecting

# Move single motor for 180-degrees
singlemotor.motor_run_for_degrees(180)

# Disconnect
singlemotor.disconnect()
exit(0) # successful execution
```

The `motor_run_for_degrees` also accepts optional parameters for direction and speed. For example, to run full speed counter-clockwise:

```
singlemotor.motor_run_for_degrees(180,
direction=le.MOTOR_MOVE_DIRECTION_COUNTERCLOCKWISE, speed=100)
```

# Reading Data

Reading data from the Single Motor can be done inline within your code or via a callback.

## Inline

```
import legoeducation as le
import time

# update these values to match the Connection Card
card_color = le.LEGO_COLOR_AZURE
card_serial = '3683'

# Connect to the Single Motor
singlemotor = le.SingleMotor()
singlemotor.connect(card_color=card_color, card_serial=card_serial)

# Check connection
if not singlemotor.connected:
    print('Error connecting to Single Motor.')
    exit(1) # error connecting

# Print the single motor position for five seconds
for i in range(50):
    print(f"Position: {singlemotor.motor.position}")
    time.sleep(0.1)

# Disconnect
singlemotor.disconnect()
exit(0) # successful execution
```

## Callback

```
import legoeducation as le
import time

# update these values to match the Connection Card
card_color = le.LEGO_COLOR_AZURE
card_serial = '3683'

# Callback for monitoring position
def notification_callback(data):
    parsed_items = le.device_notification_parser(data)
    for parsed_item in parsed_items:
        if isinstance(parsed_item, le.MotorNotification):
            print(f"Position: {parsed_item.position}")

# Connect to the Single Motor
singlemotor = le.SingleMotor()
singlemotor.connect(card_color=card_color, card_serial=card_serial)
singlemotor.set_notification_callback(notification_callback) # set callback

# Check connection
if not singlemotor.connected:
    print('Error connecting to Single Motor.')
    exit(1) # error connecting
```

```
# Wait for 5 seconds (while data is streaming via callback)
time.sleep(5)

# Disconnect
singlemotor.disconnect()
exit(0) # successful execution
```

# Example

You can find an example of interacting with the single motor by clicking the "Examples" button and selecting "Single Motor".

# Other Functions

There are many functions available for interacting with the Single Motor. Here are a few common ones:

## Motor Control

For controlling the motor:

```
singlemotor.motor_run(direction=le.MOTOR_MOVE_DIRECTION_COUNTERCLOCKWISE, speed=50) #
counter-clockwise at speed 50%
singlemotor.motor_run_for_degrees(degrees=360)
singlemotor.motor_run_for_time(2000) # time in ms, so here: 2 seconds
singlemotor.motor_set_speed(40) # Set motor to 40% speed
singlemotor.motor_stop()
```

## Available Data

Motor data (e.g. `singlemotor.motor`):

```
motorState # compare to Motor State constants
absolutePosition
power
speed
position
gesture # compare to Motor Gesture constants
```

## Hardware Control

For control of the button light color and sound beeps:

```
singlemotor.light_color(le.LEGO_COLOR_BLUE, pattern=le.LIGHT_PATTERN_BREATHE,
intensity=100)
singlemotor.beep(pattern=le.SOUND_PATTERN_BEEP_SINGLE, frequency=440)
```

# For more information

For more information about interacting with the Single Motor through the LEGO® Education

Python API, use the Python `help()` command:

```
help(le.SingleMotor)
```

---

## Double Motor



The Double Motor allows precise control and monitoring of a double motor.

# Controlling the Motor

Here is an example for driving forward for 180-degrees.

```
import legoeducation as le

# update these values to match the Connection Card
card_color = le.LEGO_COLOR_AZURE
card_serial = '3683'

# Connect to the Double Motor
doublemotor = le.DoubleMotor()
doublemotor.connect(card_color=card_color, card_serial=card_serial)

# Check connection
if not doublemotor.connected:
    print('Error connecting to Double Motor.')
    exit(1) # error connecting

# Drive for 180-degrees
doublemotor.movement_move_for_degrees(180)

# Disconnect
doublemotor.disconnect()
exit(0) # successful execution
```

The `movement_move_for_degrees` also accepts optional parameters for direction and speed. For example, to drive backward at 10%:

```
doublemotor.movement_move_for_degrees(180,
direction=le.MOVEMENT_MOVE_DIRECTION_BACKWARD, speed=10)
```

# Reading Data

Reading data from the Double Motor can be done inline within your code or via a callback.

## Inline

```
import legoeducation as le
import time

# update these values to match the Connection Card
card_color = le.LEGO_COLOR_AZURE
card_serial = '3683'

# Connect to the Double Motor
doublemotor = le.DoubleMotor()
doublemotor.connect(card_color=card_color, card_serial=card_serial)

# Check connection
if not doublemotor.connected:
    print('Error connecting to Double Motor.')
    exit(1) # error connecting

# Print both motor positions for five seconds
for i in range(50):
    print(f"Left Position: {doublemotor.motor[le.MOTOR_LEFT].position}, Right Position:
{doublemotor.motor[le.MOTOR_RIGHT].position}")
    time.sleep(0.1)

# Disconnect
doublemotor.disconnect()
exit(0) # successful execution
```

## Callback

```
import legoeducation as le
import time

# update these values to match the Connection Card
card_color = le.LEGO_COLOR_AZURE
card_serial = '3683'

# Callback for monitoring position
def notification_callback(data):
    parsed_items = le.device_notification_parser(data)
    for parsed_item in parsed_items:
        if isinstance(parsed_item, le.MotorNotification):
            if (parsed_item.motorBitMask == le.MOTOR_BITS_LEFT):
                print(f"Left Motor Position: {parsed_item.position}")
            if (parsed_item.motorBitMask == le.MOTOR_BITS_RIGHT):
                print(f"Right Motor Position: {parsed_item.position}")

# Connect to the Double Motor
doublemotor = le.DoubleMotor()
doublemotor.connect(card_color=card_color, card_serial=card_serial)
doublemotor.set_notification_callback(notification_callback) # set callback

# Check connection
```

```
if not doublemotor.connected:
    print('Error connecting to Double Motor.')
    exit(1) # error connecting

# Wait for 5 seconds (while data is streaming via callback)
time.sleep(5)

# Disconnect
doublemotor.disconnect()
exit(0) # successful execution
```

# Example

You can find an example of interacting with the single motor by clicking the "Examples" button and selecting "Double Motor".

# Other Functions

There are many functions available for interacting with the Double Motor. Here are a few common ones:

## Motor Control

Controlling both motors together (Movement Functions):

```
doublemotor.movement_move_for_time(time_ms=1000,
direction=le.MOVEMENT_DIRECTION_BACKWARD, speed=50)
doublemotor.movement_turn_for_degrees(degrees=90,
direction=le.MOVEMENT_TURN_DIRECTION_LEFT)
```

Controlling motors independently (Motor Functions):

```
# Rotate the right side of the Double Motor counterclockwise at 50% speed.
doublemotor.motor_run(direction=le.MOTOR_MOVE_DIRECTION_COUNTERCLOCKWISE,
motor=le.MOTOR_RIGHT, speed=50)
```

Additional examples of controlling one motor:

```
doublemotor.motor_run_for_degrees(360, motor=le.MOTOR_LEFT, speed=30)
doublemotor.motor_run_for_time(2000)
doublemotor.motor_stop()
```

## IMU Settings

```
doublemotor.imu_reset_yaw_axis(0) # reset yaw to new value
doublemotor.imu_set_yaw_face(yaw_face=le.DEVICE_FACE_LEFT) # set to Device Face
constants
```

## Available Data

Motor data (from each motor, e.g. `doublemotor.motor[le.MOTOR_LEFT]`):

```
motorBitMask # compare to Motor Bits constants
motorState # compare to Motor State constants
absolutePos
power
speed
position
gesture # compare to Motor Gesture constants
```

IMU data (from the internal IMU, e.g. `doublemotor.imu_device`).

```
orientation # compare to Device Face constants
yawFace # compare to Device Face constants
yaw
pitch
roll
accelerometerX
accelerometerY
accelerometerZ
gyroscopeX
gyroscopeY
gyroscopeZ
```

IMU gesture (from the internal IMU, e.g. `doublemotor.imu_gesture`).

```
gesture # compare to Motion Gesture constants
```

## Hardware Control

For control of the button light color and sound beeps:

```
doublemotor.light_color(le.LEGO_COLOR_BLUE, pattern=le.LIGHT_PATTERN_BREATHE,
intensity=100)
doublemotor.beep(pattern=le.SOUND_PATTERN_BEEP_SINGLE, frequency=440)
```

# For more information

For more information about interacting with the Double Motor through the LEGO® Education Python API, use the Python `help()` command:

```
help(le.DoubleMotor)
```

---

## Color Sensor



The Color Sensor allows control and monitoring of a color sensor.

# Reading Data

Reading data from the Color Sensor can be done inline within your code or via a callback.

## Inline

```
import legoeducation as le
import time

# update these values to match the Connection Card
card_color = le.LEGO_COLOR_AZURE
card_serial = '3683'

# Connect to the Color Sensor
colorsensor = le.ColorSensor()
colorsensor.connect(card_color=card_color, card_serial=card_serial)

# Check connection
if not colorsensor.connected:
    print('Error connecting to Color Sensor.')
    exit(1) # error connecting

# Print detected color for five seconds
for i in range(50):
    print(f"LEGO Color Detected: {colorsensor.sensor.color}")
    time.sleep(0.1)

# Disconnect
colorsensor.disconnect()
exit(0) # successful execution
```

### Callback

```
import legoeducation as le
import time

# update these values to match the Connection Card
card_color = le.LEGO_COLOR_AZURE
card_serial = '3683'

# Callback for monitoring position
def notification_callback(data):
    parsed_items = le.device_notification_parser(data)
    for parsed_item in parsed_items:
        if isinstance(parsed_item, le.ColorSensorNotification):
            print(f"LEGO Color Detected: {parsed_item.color}")

# Connect to the Color Sensor
colorsensor = le.ColorSensor()
colorsensor.connect(card_color=card_color, card_serial=card_serial)
colorsensor.set_notification_callback(notification_callback) # set callback

# Check connection
if not colorsensor.connected:
    print('Error connecting to Color Sensor.')
    exit(1) # error connecting

# Wait for 5 seconds (while data is streaming via callback)
time.sleep(5)

# Disconnect
colorsensor.disconnect()
exit(0) # successful execution
```

# Example

You can find an example of interacting with the single motor by clicking the "Examples" button and selecting "Color Sensor".

# Other Functions

There are other available ways for interacting with the Color Sensor. Here are a few common ones:

### Available Data

Color data (from the color sensor, e.g. `colorsensor.sensor`):

```
color # compare to LEGO Color constants
reflection
rawRed
rawGreen
rawBlue
hue
saturation
```

```
value
```

## Hardware Control

For control of the button light color and sound beeps:

```
colorsensor.light_color(le.LEGO_COLOR_BLUE, pattern=le.LIGHT_PATTERN_BREATHE,
intensity=100)
colorsensor.beep(pattern=le.SOUND_PATTERN_BEEP_SINGLE, frequency=440)
```

# For more information

For more information about interacting with the Color Sensor through the LEGO® Education Python API, use the Python `help()` command:

```
help(le.ColorSensor)
```

---

## Controller



The Controller allows control and monitoring of a controller.

# Reading Data

Reading data from the Controller can be done inline within your code or via a callback.

## Inline

```
import legoeducation as le
import time

# update these values to match the Connection Card
card_color = le.LEGO_COLOR_AZURE
card_serial = '3683'

# Connect to the Controller
controller = le.Controller()
controller.connect(card_color=card_color, card_serial=card_serial)

# Check connection
if not controller.connected:
    print('Error connecting to Controller.')
    exit(1) # error connecting

# Print lever positions (%) for five seconds
for i in range(50):
    print(f"Left lever: {controller.sensor.leftPercent}, Right lever:
{controller.sensor.rightPercent}")
    time.sleep(0.1)

# Disconnect
controller.disconnect()
```

```
exit(0) # successful execution
```

## Callback

```
import legoeducation as le
import time

# update these values to match the Connection Card
card_color = le.LEGO_COLOR_AZURE
card_serial = '3683'

# Callback for monitoring position
def notification_callback(data):
    parsed_items = le.device_notification_parser(data)
    for parsed_item in parsed_items:
        if isinstance(parsed_item, le.ControllerNotification):
            print(f"Left lever: {parsed_item.leftPercent}, Right lever:
{parsed_item.rightPercent}")

# Connect to the Controller
controller = le.Controller()
controller.connect(card_color=card_color, card_serial=card_serial)
controller.set_notification_callback(notification_callback) # set callback

# Check connection
if not controller.connected:
    print('Error connecting to Controller.')
    exit(1) # error connecting

# Wait for 5 seconds (while data is streaming via callback)
time.sleep(5)

# Disconnect
controller.disconnect()
exit(0) # successful execution
```

# Example

You can find an example of interacting with the single motor by clicking the "Examples" button and selecting "Controller".

# Other Functions

There are other available ways for interacting with the Controller. Here are a few common ones:

## Available Data

Controller data (from the controller, e.g. `controller.sensor`):

```
leftPercent
rightPercent
leftAngle
rightAngle
```

## Hardware Control

For control of the button light color and sound beeps:

```
controller.light_color(le.LEGO_COLOR_BLUE, pattern=le.LIGHT_PATTERN_BREATHE,
intensity=100)
controller.beep(pattern=le.SOUND_PATTERN_BEEP_SINGLE, frequency=440)
```

# For more information

For more information about interacting with the Controller through the LEGO® Education Python API, use the Python `help()` command:

```
help(le.Controller)
```

---

# Combine Single Motor and Color Sensor

You can combine multiple LEGO® Education hardware in one program. Here is an example of a Single Motor and Color Sensor.

```
import legoeducation as le
import time

# update these values to match the Connection Card
card_color = le.LEGO_COLOR_AZURE
card_serial = '3683'

# Connect to the Single Motor
singlemotor = le.SingleMotor()
singlemotor.connect(card_color=card_color, card_serial=card_serial)

# Connect to the Color Sensor
colorsensor = le.ColorSensor()
colorsensor.connect(card_color=card_color, card_serial=card_serial)

# Check connection
if not (singlemotor.connected and colorsensor.connected):
    print('Error connecting to hardware.')
    exit(1) # error connecting

# If color is green, go fast (80%)
# If color is red, go slow (10%)
print('Running for five seconds: green is fast, red is slow.')
for i in range(50):
    if (colorsensor.sensor.color == le.LEGO_COLOR_GREEN):
        singlemotor.motor_run(speed=80)
    elif (colorsensor.sensor.color == le.LEGO_COLOR_RED):
        singlemotor.motor_run(speed=10)
    time.sleep(0.1)
# Done.
singlemotor.motor_stop()

# Disconnect all hardware
singlemotor.disconnect()
colorsensor.disconnect()
exit(0) # successful execution
```

# Combine Double Motor and Controller

You can combine multiple LEGO® Education hardware in one program. Here is an example of a Double Motor and Controller.

```
import legoeducation as le
import time

# update these values to match the Connection Card
card_color = le.LEGO_COLOR_AZURE
card_serial = '3683'

# Connect to the Double Motor
doublemotor = le.DoubleMotor()
doublemotor.connect(card_color=card_color, card_serial=card_serial)

# Connect to the Controller
controller = le.Controller()
controller.connect(card_color=card_color, card_serial=card_serial)

# Check connection
if not (doublemotor.connected and controller.connected):
    print('Error connecting to hardware.')
    exit(1) # error connecting

# Control left-and-right motors (tank movement) based on left-and-right levers
print('Running for five seconds: levers control motors.')
for i in range(50):
    speed_left = controller.sensor.leftPercent
    speed_right = controller.sensor.rightPercent
    doublemotor.movement_move_tank(speed_left=speed_left, speed_right=speed_right)
    time.sleep(0.1)

# Disconnect all hardware
doublemotor.disconnect()
controller.disconnect()
exit(0) # successful execution
```

# Constants

When the LEGO® Education Python API is imported into your Python code with `import legoeducation as le`, then the `le` alias can be used to access all of the following constants (e.g. `le.LEGO_COLOR_NOCOLOR`).

# LEGO Color

The LEGO Color constants are used in a variety of ways (Connection Card colors, Color Sensor detected colors, etc). Note that not all LEGO Color constants are available in each context.

```
LEGO_COLOR_NOCOLOR
LEGO_COLOR_RED
LEGO_COLOR_YELLOW
LEGO_COLOR_BLUE
LEGO_COLOR_TEAL
LEGO_COLOR_GREEN
LEGO_COLOR_PURPLE
LEGO_COLOR_WHITE
LEGO_COLOR_MAGENTA
LEGO_COLOR_ORANGE
LEGO_COLOR_AZURE
```

# Motor End State (Single and Double Motors)

For use in `motor_set_end_state` and `movement_set_end_state` functions.

```
MOTOR_END_STATE_DEFAULT
MOTOR_END_STATE_COAST
MOTOR_END_STATE_BRAKE
MOTOR_END_STATE_HOLD
MOTOR_END_STATE_CONTINUE
MOTOR_END_STATE_SMART_COAST
MOTOR_END_STATE_SMART_BRAKE
```

# Motor Move Direction (Single and Double Motors)

For use in `motor_run`, `motor_run_for_time`, `motor_run_for_degrees`, and `motor_run_to_absolute_position` functions.

```
MOTOR_MOVE_DIRECTION_CLOCKWISE
MOTOR_MOVE_DIRECTION_COUNTERCLOCKWISE
MOTOR_MOVE_DIRECTION_SHORTEST
MOTOR_MOVE_DIRECTION_LONGEST
```

# Motor State (Single and Double Motors)

For use with `motorState` (via `MotorNotification`).

```
MOTOR_STATE_READY
MOTOR_STATE_RUNNING
MOTOR_STATE_STALLED
MOTOR_STATE_CMD_ABORTED
```

```
MOTOR_STATE_REGULATION_ERROR
MOTOR_STATE_MOTOR_DISCONNECTED
MOTOR_STATE_HOLDING
MOTOR_STATE_DC_RUNNING
MOTOR_STATE_NOT_ALLOWED_TO_RUN
```

## Motor Gesture (Single and Double Motors)

For use with `gesture` (via `MotorNotification`).

```
MOTOR_GESTURE_NO_GESTURE
MOTOR_GESTURE_SLOW_CLOCKWISE
MOTOR_GESTURE_FAST_CLOCKWISE
MOTOR_GESTURE_SLOW_COUNTERCLOCKWISE
MOTOR_GESTURE_FAST_COUNTERCLOCKWISE
MOTOR_GESTURE_WIGGLED
```

## Motor Sides (Double Motor)

When specifying motor in `done`, `motor_set_speed`, `motor_run`, `motor_run_for_time`, `motor_run_for_degrees`, `motor_reset_relative_position`, `motor_run_to_relative_position`, `motor_run_to_absolute_position`, `motor_set_duty_cycle`, `motor_stop`, `motor_set_end_state`, and `motor_set_acceleration` functions.

```
MOTOR_LEFT
MOTOR_RIGHT
MOTOR_BOTH
```

## Motor Bits (Double Motor)

For use with `motorBitMask` (via `MotorNotification`).

```
MOTOR_BITS_LEFT
MOTOR_BITS_RIGHT
```

## Movement Direction (Double Motor)

For use in `movement_move` and `movement_move_for_time` functions.

```
MOVEMENT_DIRECTION_FORWARD
MOVEMENT_DIRECTION_BACKWARD
MOVEMENT_DIRECTION_LEFT
MOVEMENT_DIRECTION_RIGHT
```

## Movement Move Direction (Double Motor)

For use in `movement_move_for_degrees` function.

```
MOVEMENT_MOVE_DIRECTION_FORWARD
MOVEMENT_MOVE_DIRECTION_BACKWARD
```

## Movement Turn Direction (Double Motor)

For use in `movement_turn_for_degrees` function.

```
MOVEMENT_TURN_DIRECTION_LEFT
MOVEMENT_TURN_DIRECTION_RIGHT
```

## Device Face (Double Motor)

For use in `imu_set_yaw_face` function and with `orientation` and `yawFace` (via `ImuDeviceNotification`).

```
DEVICE_FACE_TOP
DEVICE_FACE_FRONT
DEVICE_FACE_RIGHT
DEVICE_FACE_BOTTOM
DEVICE_FACE_BACK
DEVICE_FACE_LEFT
```

## Motion Gesture (IMU in Double Motor)

For use with `gesture` (via `ImuGestureNotification`).

```
MOTION_GESTURE_NO_GESTURE
MOTION_GESTURE_TAPPED
MOTION_GESTURE_DOUBLE_TAPPED
MOTION_GESTURE_COLLISION
MOTION_GESTURE_SHAKE
MOTION_GESTURE_FREEFALL
```

## Light Pattern (Button on Hardware)

For use with `light_color` function.

```
LIGHT_PATTERN_SOLID
LIGHT_PATTERN_BREATHE
LIGHT_PATTERN_PULSE
LIGHT_PATTERN_SHORT_BLINK
LIGHT_PATTERN_LONG_BLINK
LIGHT_PATTERN_DOUBLE_BLINK
```

## Button State (Button on Hardware)

For use with `state` (via `ButtonStateNotification`).

```
BUTTON_STATE_RELEASED
BUTTON_STATE_PRESSED
```

## Sound Pattern (Hardware Beep)

For use with `beep` function.

```
SOUND_PATTERN_BEEP_SINGLE
SOUND_PATTERN_BEEP_DOUBLE
SOUND_PATTERN_BEEP_TRIPLE
SOUND_PATTERN_BEEP_UP_MIDDLE_DOWN
```

## USB Powered State

For use with `UsbPowerState` (via `InfoDeviceNotification`).

```
USB_POWER_STATE_USB_NOT_CONNECTED
USB_POWER_STATE_USB_CONNECTED
```

## Program Action

For use with `program_flow_notification` function.

```
PROGRAM_ACTION_START
PROGRAM_ACTION_STOP
```

---

# Motor Commands

## motor_set_speed(self, speed: int, *, motor: int = DEFAULT_MOTOR, blocking: bool = True)

Set the speed of a motor.

### Parameters:

- `speed`: Rotation speed as a percentage, ranging from -100% to 100%. Positive values will rotate the motor clockwise; negative values will rotate counter-clockwise.
- `motor`: Specify which motor will receive the speed setting. For Single Motor, this can be ignored. For Double Motor, specify side using a Motor Sides (Double Motor) constant (e.g. `le.MOTOR_LEFT`).
- `blocking`: Wait for the command to be completed before continuing.

### Simple Example:

```
# Single Motor: Set the speed of the motor 50%.
singlemotor.motor_set_speed(50)

# Double Motor: Set the speed of the right side of the Double Motor to 50%.
doublemotor.motor_set_speed(50, motor=le.MOTOR_RIGHT)
```

### More Examples:

```
# Set the speed of the left side of a Double Motor to -50%. Negative value will rotate
the motor counter-clockwise.
doublemotor.motor_set_speed(-50, motor=le.MOTOR_LEFT)
```

## motor_run(self, *, direction: int = MOTOR_MOVE_DIRECTION_CLOCKWISE, motor: int = DEFAULT_MOTOR, speed: int = UNCHANGED, blocking: bool = True)

Initiate the motor to begin rotating continuously. The motor will continue to rotate until either the motor_stop() function is called or a new motor command is issued.

### Parameters:

- `direction`: Specify direction of motor rotation using a Motor Move Direction (Single and Double Motors) constant.
- `motor`: Specify which motor will rotate. For Single Motor, this can be ignored. For Double Motor, specify side using a Motor Sides (Double Motor) constant (e.g. `le.MOTOR_LEFT`).
- `speed`: Rotation speed as a percentage, ranging from -100% to 100%.
- `blocking`: Wait for the command to be completed before continuing.

### Simple Example:

```
# Single Motor: Motor rotates continuously.
singlemotor.motor_run()

# Double Motor: Both sides of the Double Motor rotate continuously.
doublemotor.motor_run(motor=le.MOTOR_BOTH)
```

### More Examples:

```
# Right side of Double Motor rotates counter-clockwise continuously at a speed of 50%.
doublemotor.motor_run(direction=le.MOTOR_MOVE_DIRECTION_COUNTERCLOCKWISE,
motor=le.MOTOR_RIGHT, speed=50)
```

## motor_run_for_time(self, time_ms: int, *, direction: int = MOTOR_MOVE_DIRECTION_CLOCKWISE, motor: int = DEFAULT_MOTOR, speed: int = UNCHANGED, blocking: bool = True)

Rotate a motor for a number of milliseconds, then stop.

### Parameters:

- `time_ms`: Duration of rotation in milliseconds.
- `direction`: Specify direction of motor rotation using a Motor Move Direction (Single and Double Motors) constant.
- `motor`: Specify which motor will rotate. For Single Motor, this can be ignored. For Double Motor, specify side using a Motor Sides (Double Motor) constant (e.g. `le.MOTOR_LEFT`).
- `speed`: Rotation speed as a percentage, ranging from -100% to 100%.
- `blocking`: Wait for the command to be completed before continuing.

### Simple Example:

```
# Single Motor: Run the motor for 2 seconds.
singlemotor.motor_run_for_time(2000)

# Double Motor: Run the right side of the Double Motor for 1 second.
doublemotor.motor_run_for_time(1000, motor=le.MOTOR_RIGHT)
```

### More Examples:

```
# Run the left side of a Double Motor counterclockwise for 1 second at 60% speed.
doublemotor.motor_run_for_time(1000,
direction=le.MOTOR_MOVE_DIRECTION_COUNTERCLOCKWISE, motor=le.MOTOR_LEFT, speed=60)
```

## motor_run_for_degrees(self, degrees: int, *, direction: int = MOTOR_MOVE_DIRECTION_CLOCKWISE, motor: int = DEFAULT_MOTOR, speed: int = UNCHANGED, blocking: bool = True)

Rotate a motor by a number of degrees, then stop.

### Parameters:

- `degrees`: Angle of rotation in degrees.
- `direction`: Specify direction of motor rotation using a Motor Move Direction (Single and Double Motors) constant.
- `motor`: Specify which motor will rotate. For Single Motor, this can be ignored. For Double Motor, specify side using a Motor Sides (Double Motor) constant (e.g. `le.MOTOR_LEFT`).
- `speed`: Rotation speed as a percentage, ranging from -100% to 100%.
- `blocking`: Wait for the command to be completed before continuing.

### Simple Example:

```
# Single Motor: Rotate the motor 180°.
singlemotor.motor_run_for_degrees(180)

# Double Motor: Rotate the right side of the Double Motor 90°.
doublemotor.motor_run_for_degrees(90, direction=le.MOTOR_MOVE_DIRECTION_CLOCKWISE,
motor=le.MOTOR_RIGHT)
```

### More Examples:

```
# Rotate the left side of the Double Motor 90° at 30% speed.
doublemotor.motor_run_for_degrees(90, motor=le.MOTOR_LEFT, speed=30)
```

## motor_reset_relative_position(self, *, motor: int = DEFAULT_MOTOR, position: int = 0, blocking: bool = True)

Set the current motor relative position to a new value. If no value specified, new position will be 0.

### Parameters:

- `motor`: Specify which motor to reset relative position. For Single Motor, this can be ignored. For Double Motor, specify side using a Motor Sides (Double Motor) constant (e.g. `le.MOTOR_LEFT`).
- `position`: New position value in degrees, ranging from 0 to 360.
- `blocking`: Wait for the command to be completed before continuing.

### Simple Example:

```
# Single Motor: Set the current relative position of the motor to 0 degrees.
singlemotor.motor_reset_relative_position()

# Double Motor: Set the current relative position of the right side of the Double Motor
0 degrees.
doublemotor.motor_reset_relative_position(motor=le.MOTOR_RIGHT)
```

### More Examples:

```
# Set the current position of the left side of the double motor to 180 degrees.
doublemotor.motor_reset_relative_position(motor=le.MOTOR_LEFT, position=180)
```

## motor_run_to_relative_position(self, position: int, *, motor: int = DEFAULT_MOTOR, speed: int = UNCHANGED, blocking: bool = True)

Rotate a motor to a specified position based on the current relative reset point.

### Parameters:

- `position`: Target motor position in degrees based on the current relative reset point.
- `motor`: Specify which motor to rotate. For Single Motor, this can be ignored. For Double Motor, specify side using a Motor Sides (Double Motor) constant (e.g. `le.MOTOR_LEFT`).
- `speed`: Rotation speed as a percentage, ranging from -100% to 100%.
- `blocking`: Wait for the command to be completed before continuing.

### Simple Example:

```
# Single Motor: Rotate the motor to relative position 180 degrees.
singlemotor.motor_run_to_relative_position(180)

# Double Motor: Rotate the right side of Double Motor to relative position 180 degrees.
doublemotor.motor_run_to_relative_position(180, motor=le.MOTOR_RIGHT)
```

### More Examples:

```
# Rotate the left side of the Double Motor to relative position 90 at 30% speed.
# With blocking=False, the program will move onto the next command before this command
is completed.
doublemotor.motor_run_to_relative_position(90, motor=le.MOTOR_LEFT, speed=30,
blocking=False)
```

## motor_run_to_absolute_position(self, position: int, *, direction: int = MOTOR_MOVE_DIRECTION_SHORTEST, motor: int = DEFAULT_MOTOR, speed: int = UNCHANGED, blocking: bool = True)

Rotate motor to a fixed, absolute position.

### Parameters:

- `position`: Target motor position in degree.
- `direction`: Specify direction of motor rotation using a Motor Move Direction (Single and Double Motors) constant.
- `motor`: Specify which motor will rotate. For Single Motor, this can be ignored. For Double Motor, specify side using a Motor Sides (Double Motor) constant (e.g. `le.MOTOR_LEFT`).
- `speed`: Rotation speed as a percentage, ranging from -100% to 100%.
- `blocking`: Wait for the command to be completed before continuing.

### Simple Example:

```
# Single Motor: Rotate the motor to absolute position 90.
singlemotor.motor_run_to_absolute_position(90)

# Double Motor: Rotate the left side of Double Motor to absolute position 180.
doublemotor.motor_run_to_absolute_position(180, motor=le.MOTOR_LEFT)
```

### More Examples:

```
# Rotate the left side of the Double Motor in a clockwise direction at 10% speed until
the motor arrives at position 180.
doublemotor.motor_run_to_absolute_position(180,
direction=le.MOTOR_MOVE_DIRECTION_CLOCKWISE, motor=le.MOTOR_LEFT, speed=10)
```

## motor_set_duty_cycle(self, duty_cycle: int, *, motor: int = DEFAULT_MOTOR, blocking: bool = True)

Directly drive the motor continuously with duty cycle. This uses a raw duty cycle value and overrides the default speed control.

### Parameters:

- `duty_cycle`: Motor duty cycle from -100 to 100. Positive values will rotate the motor clockwise; negative values will rotate counter-clockwise.
- `motor`: Specify which motor will run. For Single Motor, this can be ignored. For Double Motor, specify side using a Motor Sides (Double Motor) constant (e.g. `le.MOTOR_LEFT`).
- `blocking`: Wait for the command to be completed before continuing.

### Simple Example:

```
# Single Motor: Run the motor continuously at duty cycle 50.
singlemotor.motor_set_duty_cycle(50)

# Double Motor: Run the right side of the Double Motor continuously at duty cycle 50.
doublemotor.motor_set_duty_cycle(50, motor=le.MOTOR_RIGHT)
```

## motor_stop(self, *, motor: int = DEFAULT_MOTOR, blocking: bool = True)

Stop the motor. This command uses the current end state as specified by `motor_set_end_state()` to stop the motor.

### Parameters:

- `motor`: Specify which motor will run. For Single Motor, this can be ignored. For Double Motor, specify side using a Motor Sides (Double Motor) constant (e.g. `le.MOTOR_LEFT`).
- `blocking`: Wait for the command to be completed before continuing.

### Simple Example:

```
# Single Motor: Stop the motor.
singlemotor.motor_stop()
```

```
# Double Motor: Stop both sides of the Double Motor.
doublemotor.motor_stop(motor=le.MOTOR_BOTH)
```

## motor_set_end_state()

Set the behavior of a stopped motor with the `end_state` parameter. Options include allowing it to coast, applying a brake, or holding its position.

### Parameters:

- `end_state`: Specify the end state value using a Motor End State (Single and Double Motors) constant.
- `motor`: Specify which motor will receive the end state setting. For Single Motor, this can be ignored. For Double Motor, specify side using a Motor Sides (Double Motor) constant (e.g. `le.MOTOR_LEFT`).
- `blocking`: Wait for the command to be completed before continuing.

### Simple Example:

```
# Single Motor: Set the motor to hold the current position immediately after stopping.
singlemotor.motor_set_end_state(le.MOTOR_END_STATE_HOLD)

# Double Motor: Set the right side of the Double Motor to hold the current position
immediately after stopping.
singlemotor.motor_set_end_state(le.MOTOR_END_STATE_HOLD, motor=le.MOTOR_RIGHT)
```

### More Examples:

```
# Set the right side of the Double Motor to coast, or rotate freely, once stopped.
doublemotor.motor_set_end_state(le.MOTOR_END_STATE_COAST, motor=le.MOTOR_RIGHT)
```

## motor_set_acceleration(self, end_state: int, *, motor: int = DEFAULT_MOTOR, blocking: bool = True)

Control the rate a motor speeds up or slows down.

### Parameters:

- `acceleration`: Rate the motor will speed up, ranging from 0 to 100.
- `deceleration`: Rate the motor will slow down, ranging from 0 to 100.
- `motor`: Specify which motor will receive the acceleration setting. For Single Motor, this can be ignored. For Double Motor, specify side using a Motor Sides (Double Motor) constant (e.g. `le.MOTOR_LEFT`).
- `blocking`: Wait for the command to be completed before continuing.

### Simple Example:

```
# Single Motor: Set the motor to accelerate at 25 and decelerate at 75.
singlemotor.motor_set_acceleration(25, 75)

# Double Motor: Set the right side of the Double Motor to accelerate at 50 and
```

```
# decelerate at 50.
doublemotor.motor_set_acceleration(50, 50, motor=le.MOTOR_RIGHT)
```

# Movement Commands

## movement_move(self, *, direction: int = MOVEMENT_DIRECTION_FORWARD, speed: int = UNCHANGED, blocking: bool = True)

Drive the Double Motor in a specified direction at a given speed.

### Parameters:

- `direction`: Specify which direction to drive using a Movement Direction (Double Motor) constant.
- `speed`: Rotation speed as a percentage, ranging from -100% to 100%.
- `blocking`: Wait for the command to be completed before continuing.

### Simple Example:

```
# Double Motor: Drive forward.
doublemotor.movement_move(direction=le.MOVEMENT_DIRECTION_FORWARD)
```

### More Examples:

```
# Double Motor: Drive backward at 50% speed.
doublemotor.movement_move(direction=le.MOVEMENT_DIRECTION_BACKWARD, speed=50)
```

## movement_move_for_time(self, time_ms: int, *, direction: int = MOVEMENT_DIRECTION_FORWARD, speed = UNCHANGED, blocking: bool = True)

Drive the Double Motor in a specified direction for a number of milliseconds, then stop.

### Parameters:

- `time_ms`: Duration of movement in milliseconds.
- `direction`: Specify which direction to drive using a Movement Direction (Double Motor) constant.
- `speed`: Rotation speed as a percentage, ranging from -100% to 100%.
- `blocking`: Wait for the command to be completed before continuing.

### Simple Example:

```
# Double Motor: Drive the Double Motor forward for 2 seconds.
doublemotor.movement_move_for_time(2000)
```

### More Examples:

```
# Drive backward for 0.5 seconds at 40% speed.
doublemotor.movement_move_for_time(500, direction=le.MOVEMENT_DIRECTION_BACKWARD,
speed=40)
```

## movement_move_for_degrees(self, degrees: int, *, direction: int = MOVEMENT_MOVE_DIRECTION_FORWARD, speed = UNCHANGED, blocking: bool = True)

Drive the Double Motor in a specified direction for a number of motor rotation degrees.

### Parameters:

- `degrees`: Number of degrees the motors will rotate while driving.
- `direction`: Specify which direction to drive using a Movement Move Direction (Double Motor) constant.
- `speed`: Rotation speed as a percentage, ranging from -100% to 100%.
- `blocking`: Wait for the command to be completed before continuing.

### Simple Example:

```
# Double Motor: Drive forward for 360 degrees (one full rotation of the motors).
doublemotor.movement_move_for_degrees(360,
direction=le.MOVEMENT_MOVE_DIRECTION_FORWARD)
```

### More Examples:

```
# Move backward and rotate motors 90 degrees at 40% speed.
doublemotor.movement_move_for_degrees(90,
direction=le.MOVEMENT_MOVE_DIRECTION_BACKWARD, speed=40)
```

## movement_move_tank(self, speed_left: int, speed_right: int, *, blocking: bool = True)

Drive the Double Motor using separate speeds for the left and right sides.

### Parameters:

- `speed_left`: Percentage speed of the left side of the Double Motor, ranging from -100% to 100%.
- `speed_right`: Percentage speed of the right side of the Double Motor, ranging from -100% to 100%.
- `blocking`: Wait for the command to be completed before continuing.

### Simple Example:

```
# Double Motor: Drive straight forward at 40% speed.
doublemotor.movement_move_tank(40, 40)
```

### More Examples:

```
# Control the motor speeds with the positions of the Controller levers.
doublemotor.movement_move_tank(controller.sensor.leftPercent,
controller.sensor.rightPercent)
```

## movement_move_tank_for_degrees(self, degrees: int, *, speed_left: int = 50, speed_right: int = 50, blocking: bool = True)

Drive the Double Motor using separate speeds for the left and right sides. The motors will continue to rotate until one of the motors has turned the specified number of degrees.

### Parameters:

- `degrees`: Number of degrees the motor with the highest speed will rotate while driving.
- `speed_left`: Rotation speed of the left side of the Double Motor as a percentage, ranging from -100% to 100%.
- `speed_right`: Rotation speed of the right side of the Double Motor as a percentage, ranging from -100% to 100%.
- `blocking`: Wait for the command to be completed before continuing.

### Simple Example:

```
# Double Motor: Move straight forward for one full wheel turn.
doublemotor.movement_move_tank_for_degrees(360, speed_left=50, speed_right=50)
```

### More Examples:

```
# Move forward until one of the wheels has completed a full rotation.
doublemotor.movement_move_tank_for_degrees(360, speed_left=30, speed_right=60)
```

## movement_turn_for_degrees(self, degrees: int, *, direction: int = MOVEMENT_TURN_DIRECTION_LEFT, speed = UNCHANGED, blocking: bool = True)

Turn the Double Motor to the left or right for a number of degrees. The Double Motor will continue turning in the chosen direction until the IMU confirms the intended rotation is complete.

### Parameters:

- `degrees`: Number of degrees the Double Motor will turn.
- `direction`: Specify which direction to turn using a Movement Turn Direction (Double Motor) constant.
- `speed`: Rotation speed as a percentage, ranging from -100% to 100%.
- `blocking`: Wait for the command to be completed before continuing.

### Simple Example:

```
# Double Motor: Turn the Double Motor 90 degrees to the left.
doublemotor.movement_turn_for_degrees(90, direction=le.MOVEMENT_TURN_DIRECTION_LEFT)
```

### More Examples:

```
# Turn the Double Motor 30 degrees to the right at 40% speed.
doublemotor.movement_turn_for_degrees(30, direction=le.MOVEMENT_TURN_DIRECTION_RIGHT,
speed=40)
```

## movement_stop(self, *, blocking: bool = True)

Stop the motor. This command uses the current end state as specified by `movement_set_end_state()` to stop the motor.

### Parameters:

- `blocking`: Wait for the command to be completed before continuing.

### Simple Example:

```
# Double Motor: Stop any movement commands running on the Double Motor.
doublemotor.movement_stop()
```

## movement_set_speed(self, speed: int, *, blocking: bool = True)

Set the driving speed of the Double Motor. If the Double Motor is currently moving, the speed changes immediately. If not, the speed is set for future movements.

### Parameters:

- `speed`: Rotation speed as a percentage, ranging from -100% to 100%.
- `blocking`: Wait for the command to be completed before continuing.

### Simple Example:

```
# Double Motor: Set the driving speed of the Double Motor to 50%.
doublemotor.movement_set_speed(50)
```

### More Examples:

```
# Set the driving speed of the Double Motor to -20%. The negative value will drive the
Double Motor backwards.
doublemotor.movement_set_speed(-20)
```

## movement_set_end_state(self, end_state: int, *, blocking: bool = True)

Set the behavior of a stopped Double Motor with the `end_state` parameter. Options include allowing it to coast, applying a brake, or holding its position.

### Parameters:

- `end_state`: Specify the end state value using a Motor End State (Single and Double Motors) constant.

- `blocking`: Wait for the command to be completed before continuing.

### Simple Example:

```
# Double Motor: Set the end state of movement commands to `le.MOTOR_END_STATE_BRAKE`.
doublemotor.movement_set_end_state(le.MOTOR_END_STATE_BRAKE)
```

### More Examples:

```
# Let the Double Motor coast when movement ends.
doublemotor.movement_set_end_state(le.MOTOR_END_STATE_COAST)
```

## movement_set_acceleration(self, acceleration:int, deceleration:int, *, blocking: bool = True)

Control the rate a Double Motor speeds up or slows down when driving.

### Parameters:

- `acceleration`: Rate the Double Motor will speed up, ranging from 0 to 100.
- `deceleration`: Rate the Double Motor will slow down, ranging from 0 to 100.
- `blocking`: Wait for the command to be completed before continuing.

### Simple Example:

```
# Double Motor: Set the acceleration to 50 on both sides of the Double Motor.
doublemotor.movement_set_acceleration(50, 50)
```

### More Examples:

```
# Double Motor will accelerate at 20 and decelerate at 80.
doublemotor.movement_set_acceleration(20, 80)
```

## movement_set_turn_steering(self, steering, *, blocking: bool = True)

Adjust the sharpness of turns by modifying the balance between the left and right sides of the Double Motor.

### Parameters:

- `steering`: Steering value from 0 to 100.
- `blocking`: Wait for the command to be completed before continuing.

### Simple Example:

```
# Double Motor: Set turn steering to 50.
doublemotor.movement_set_turn_steering(50)
```

# IMU Settings

## imu_set_yaw_face(self, yaw_face: int, *, blocking: bool = True)

Choose which side of the Double Motor to set as yaw.

### Parameters:

- `yaw_face`: Specify which side of the Double Motor to set as yaw using a Device Face (Double Motor) constant.
- `blocking`: Wait for the command to be completed before continuing.

### Simple Example:

```
# Double Motor: Set the right face of the Double Motor as yaw.
doublemotor.imu_set_yaw_face(le.DEVICE_FACE_RIGHT)
```

## imu_reset_yaw_axis(self, value: int = 0, *, blocking: bool = True)

Reset the yaw reading to a new value on the Double Motor.

### Parameters:

- `value`: New yaw value.
- `blocking`: Wait for the command to be completed before continuing.

### Simple Example:

```
# Double Motor: Set the yaw value to 0.
doublemotor.imu_reset_yaw_axis()
```

### More Examples:

```
# Double Motor: Reset yaw reading with 10 degree offset.
doublemotor.imu_reset_yaw_axis(10)
```

## done(self, motor = None)

Check if any motor commands are still running.

### Parameters:

- `motor`: Specify which motor to check. For Single Motor, this can be ignored. For Double Motor, specify side using a Motor Sides (Double Motor) constant (e.g. `le.MOTOR_LEFT`). If no motor is given, all motors are checked.

### Simple Example:

```
# Single Motor: Check if any motor commands are still running on the Single Motor.
status = singlemotor.done()
```

```
# Double Motor: Check if any motor commands are still running on the left side of the
Double Motor.
status = doublemotor.done(motor=le.MOTOR_LEFT)
```

### More Examples:

```
# Start moving the Single Motor (without blocking). Wait until motor command finishes.
singlemotor.motor_run_for_degrees(360, speed=10, blocking=False)
while not singlemotor.done():
    time.sleep(0.1)
print('Done.')
```

# Other Functions

## begin_batch(self)

Start batching commands into a single BLE packet. When batch mode is active, motor commands are queued instead of being sent immediately. Call `end_batch()` to send all queued commands in a single packet. This ensures they all start nearly simultaneously. Note: If a previous batch was never ended, it is automatically discarded and a warning is logged.

### Simple Example:

```
# Double Motor: configure Double Motor and start both motors together
doublemotor.begin_batch()
doublemotor.motor_run(motor=le.MOTOR_LEFT, speed=50)
doublemotor.motor_run(motor=le.MOTOR_RIGHT, speed=10)
doublemotor.end_batch() # start together
```

## end_batch(self, blocking=True)

Sends all batched commands in a single packet. To use effectively, use `begin_batch()` to create a queue of motor commands and call `end_batch()` to start all queued commands nearly simultaneously.

### Parameters:

- `blocking`: Wait for the command(s) to be completed before continuing.

### Simple Example:

```
# Double Motor: configure Double Motor and start both motors together
doublemotor.begin_batch()
doublemotor.motor_run(motor=le.MOTOR_LEFT, speed=50)
doublemotor.motor_run(motor=le.MOTOR_RIGHT, speed=10)
doublemotor.end_batch() # start together
```

## batch(self, blocking=True)

Context manager for batch mode operations. This block automatically calls `begin_batch()` on

entry and `end_batch()` on exit. If an error occurs inside the block, it ensures batch mode is properly closed.

### Parameters:

- `blocking`: Wait for the command(s) to be completed before continuing.

### Simple Example:

```
# Double Motor: configure Double Motor and start both motors together
with doublemotor.batch():
    doublemotor.motor_run(motor=le.MOTOR_LEFT, speed=50)
    doublemotor.motor_run(motor=le.MOTOR_RIGHT, speed=10)
```

## cancel_batch(self)

Cancel the current batch without sending any commands. Returns the number of commands that were discarded.

### Simple Example:

```
# Double Motor: configure Double Motor but cancel batch
doublemotor.begin_batch()
doublemotor.motor_run(motor=le.MOTOR_LEFT, speed=50)
doublemotor.motor_run(motor=le.MOTOR_RIGHT, speed=10)
doublemotor.cancel_batch()
```

### More Examples:

```
# Double Motor: configure Double Motor but cancel batch and print number of commands
discarded.
doublemotor.begin_batch()
doublemotor.motor_run(motor=le.MOTOR_LEFT, speed=50)
doublemotor.motor_run(motor=le.MOTOR_RIGHT, speed=10)
num = doublemotor.cancel_batch()
print(f'Cancelled batch. {num} commands canceled.')
```

## search(self, timeout=5, device_name=None, device_mac=None, card_color=None, card_serial=None)

Use Bluetooth to scan for nearby hardware that matches the listed criteria.

### Parameters:

- `timeout`: Duration of scan in seconds.
- `device_name`: Filter for hardware by name contents.
- `device_mac`: Filter for hardware by MAC address contents.
- `card_color`: Filter for hardware by Connection Card of a specific color using a LEGO Color constant.
- `card_serial`: Filter for hardware by Connection Card of a specific serial number. Use a string (e.g., "0049") for serial numbers with leading zeros.

### Simple Example:

```
# Color Sensor: Search for all available Color Sensors.
colorsensor = le.ColorSensor()
sensorlist = colorsensor.search()
```

### More Examples:

```
# Filter for Controllers by Connection Card of color red
controller = le.Controller()
sensorlist = controller.search(card_color=le.LEGO_COLOR_RED)
# Print each Controller found
if (sensorlist):
    for i in range(len(sensorlist)):
        print(sensorlist[i])
```

## connect(self, device=None, *, device_name=None, device_mac=None, card_color=None, card_serial=None, device_notification_delay = DEFAULT_DEVICE_NOTIFICATION_DELAY)

Connect to hardware. This function will use Bluetooth to scan and connect to the first hardware found based on any filters specified.

### Parameters:

- `device`: A device entry returned from `search()`. When `None` (default), a scan is performed automatically.
- `device_name`: Filter for hardware by name contents.
- `device_mac`: Filter for hardware by MAC address contents.
- `card_color`: Filter for hardware by a Connection Card of a specific color using a LEGO Color constant.
- `card_serial`: Filter for hardware by a Connection Card of a specific serial number. Use a string (e.g., "0049") for serials with leading zeros.
- `device_notification_delay`: Duration in milliseconds between automatic device notifications. When this value is greater than 0, the command waits for the first notification before returning. Must be 0 or between 15 and 1000.

### Simple Example:

```
# Single Motor: Scan and connect to the first Single Motor found.
singlemotor = le.SingleMotor()
singlemotor.connect()

# Double Motor: Scan and connect to the first Double Motor found.
doublemotor = le.DoubleMotor()
doublemotor.connect()
```

### More Examples:

```
# Controller: Connect to a Controller indicated by the specified Connection Card.
```

```
controller = le.Controller()
controller.connect(card_color=le.LEGO_COLOR_RED, card_serial="1234")
```

## disconnect(self)

Disconnect from the hardware. This resets the hardware back to Bluetooth broadcast mode.

### Simple Example:

```
# Single Motor: Disconnect from the Single Motor.
singlemotor.disconnect()
```

## set_notification_callback(self, callback: Callable[[Dict[str, Any]], Any])

Register a callback function to handle hardware notifications. This method sets a callback function that will be invoked whenever the hardware sends sensor updates.

### Parameters:

- `callback`: A function that takes one argument (the notification object).

### Simple Example:

```
# Single Motor: set a notification callback
singlemotor.set_notification_callback(notification_callback)
```

### More Examples:

```
# Define a function for handling Single Motor notifications
def notification_callback(data):
    parsed_items = le.device_notification_parser(data)
    for parsed_item in parsed_items:
        if isinstance(parsed_item, le.MotorNotification):
            print(f"Single Motor position: {parsed_item.position}")
# Set Single Motor notification callback
singlemotor.set_notification_callback(notification_callback)
```

## info(self)

Returns technical information about the hardware.

### Simple Example:

```
# Single Motor: Get technical information about the Single Motor
info = singlemotor.info()
```

### More Examples:

```
# Iterate through and print all technical information about the the Color Sensor
for key, value in vars(colorsensor.info()).items():
    if (type(value) is int):
```

```
        print(key, value)
```

## device_uuid(self)

Get the Universally Unique Identifier (UUID) of the hardware. The UUID is linked to the hardware and does not change even if the device is reset or updated.

### Simple Example:

```
# Single Motor: Print the UUID of a Single Motor.
response = singlemotor.device_uuid()
print(response.uuid.hex()) # print byte array as hex
```

## program_flow_notification(self, action: int, *, blocking: bool = True)

Tell the hardware that a program has started or stopped. This resets many device settings depending on the chosen action.

### Parameters:

- `action`: Specify the type of program action using a Program Action constant.
- `blocking`: Wait for the command to be completed before continuing.

### Simple example:

```
# Single Motor: Mark the program as started.
singlemotor.program_flow_notification(le.PROGRAM_ACTION_START)
```

### More examples:

```
# Mark the Single Motor program as stopped.
singlemotor.program_flow_notification(le.PROGRAM_ACTION_STOP)
```

## device_notification_request(self, delay_ms: int, *, blocking: bool = True)

Set how often the hardware sends automatic notifications. The contents of the notification vary based on the type of hardware, and can include button presses, sensor values and other state changes reported by the hardware.

### Parameters:

- `delay_ms`: Duration in milliseconds between notifications. Must be 0 (to turn off notifications) or between 15 and 1000.
- `blocking`: Wait for the command to be completed before continuing.

### Simple Example:

```
# Single Motor: Set duration between notifications to 50 milliseconds.
```

```
singlemotor.device_notification_request(50)
```

### More Examples:

```
# Turn off the automatic notifications on the Single Motor
singlemotor.device_notification_request(0)
```

## light_color(self, color: int, *, pattern: int = 0, intensity: int = 100, blocking: bool = True)

Set the button light color and blink pattern on the hardware.

### Parameters:

- `color`: Set the light color using a LEGO Color constant.
- `pattern`: Set the light behavior using a Light Pattern (Button on Hardware) constant.
- `intensity`: Set the brightness of the light as a percentage, ranging from 0% to 100% (default is 100).
- `blocking`: Wait for the command to be completed before continuing.

### Simple Example:

```
# Single Motor: Set the button light to the color red on the Single Motor.
singlemotor.light_color(le.LEGO_COLOR_RED)
```

### More Examples:

```
# Set the light color on the Single Motor to blue and blink pattern to "breathe"
singlemotor.light_color(le.LEGO_COLOR_BLUE, pattern=le.LIGHT_PATTERN_BREATHE)

# Set the light color on the Single Motor to blue at 50% intensity
singlemotor.light_color(le.LEGO_COLOR_BLUE, intensity=50)

# Set the light color on the Single Motor to green with a "breathe" blink pattern and
75% intensity
singlemotor.light_color(le.LEGO_COLOR_GREEN, pattern=le.LIGHT_PATTERN_BREATHE,
intensity=75)
```

## beep(self, *, pattern: int = SOUND_PATTERN_BEEP_SINGLE, frequency: int = 440, count: int = 1, blocking: bool = True)

Play a beep sound pattern on the hardware.

### Parameters:

- `pattern`: Specify the beep sound pattern to be played on the hardware using a Sound Pattern (Hardware Beep) constant.
- `frequency`: Set the frequency of the beep sound in hertz. Default is 440 Hz.
- `count`: Determine how many times the beep pattern should be repeated. Default is 1 (once).
- `blocking`: Wait for the command to be completed before continuing.

### Simple Example:

```
# Single Motor: play a beep
singlemotor.beep()
```

### More Examples:

```
# Play a high "beep double" sound pattern on a Single Motor.
singlemotor.beep(pattern=le.SOUND_PATTERN_BEEP_DOUBLE, frequency=880)

# Play a low beep three times on a Color Sensor.
colorsensor.beep(frequency=220, count=3)
```

## stop_beep(self, *, blocking: bool = True)

Stop any ongoing beep sound from the hardware.

### Parameters:

- `blocking`: Wait for the command to be completed before continuing.

### Simple Example:

```
# Single Motor: Stop any ongoing beep sounds from the Single Motor.
singlemotor.stop_beep()
```

### More Examples:

```
# Start playing many low beeps on a Color Sensor.
colorsensor.beep(frequency=220, count=10, blocking=False)
time.sleep(3)
colorsensor.stop_beep() # stop beep sounds after three seconds
```

## device_notification_parser(notification_obj)

This function processes a notification object to extract hardware notifications. Each notification is deserialized and added to a list, which is returned by the function.

### Parameters:

- `notification_obj`: A notification object coming from hardware.

### Simple Example:

```
# In a notification handler, parse the notification object to get a list of hardware
notifications.
parsed_items = le.device_notification_parser(data)
```

### More Examples:

```
# Parse the data (and iterate through list) in a notification callback
def notification_callback(data):
    parsed_items = le.device_notification_parser(data)
```

```
    for parsed_item in parsed_items:
        if isinstance(parsed_item, le.MotorNotification):
            print(f"Single Motor position: {parsed_item.position}")
# Set Single Motor notification callback
singlemotor.set_notification_callback(notification_callback)
```