# python-ev3dev Documentation

***Release 2.1.0.post6***

**Ralph Hempel et al**

**Aug 23, 2020**

# Contents

A Python3 library implementing an interface for ev3dev devices, letting you control motors, sensors, hardware buttons, LCD displays and more from Python code.

If you haven't written code in Python before, you can certainly use this library to help you learn the language!

# CHAPTER 1

# Getting Started

This library runs on ev3dev. Before continuing, make sure that you have set up your EV3 or other ev3dev device as explained in the ev3dev Getting Started guide. Make sure you have an ev3dev-stretch version greater than `2.2.0`. You can check the kernel version by selecting "About" in Brickman and scrolling down to the "kernel version". If you don't have a compatible version, upgrade the kernel before continuing.

CHAPTER 2

# Usage

To start out, you'll need a way to work with Python. We recommend the ev3dev Visual Studio Code extension. If you're interested in using that, check out our Python + VSCode introduction tutorial and then come back once you have that set up.

Otherwise, you can can work with files via an SSH connection with an editor such as nano, use the Python interactive REPL (type `python3`), or roll your own solution. If you don't know how to do that, you are probably better off choosing the recommended option above.

## 2.1 The template for a Python script

Every Python program should have a few basic parts. Use this template to get started:

```
#!/usr/bin/env python3

from time import sleep

from ev3dev2.motor import LargeMotor, OUTPUT_A, OUTPUT_B, SpeedPercent, MoveTank
from ev3dev2.sensor import INPUT_1
from ev3dev2.sensor.lego import TouchSensor
from ev3dev2.led import Leds

# TODO: Add code here
```

The first line should be included in every Python program you write for ev3dev. It allows you to run this program from Brickman, the graphical menu that you see on the device screen. The other lines are import statements which give you access to the library functionality. You will need to add additional classes to the import list if you want to use other types of devices or additional utilities.

You should use the `.py` extension for your file, e.g. `my-file.py`.

If you encounter an error such as `/usr/bin/env: 'python3\r': No such file or directory`, you must switch your editor's "line endings" setting for the file from "CRLF" to just "LF". This is usually in the status bar at the bottom. For help, see our FAQ page.

## 2.2 Important: Make your script executable (non-Visual Studio Code only)

To be able to run your Python file, **your program must be executable**. If you are using the ev3dev Visual Studio Code extension, you can skip this step, as it will be automatically performed when you download your code to the brick.

**To mark a program as executable from the command line (often an SSH session), run** `chmod +x my-file.py`.

You can now run `my-file.py` via the Brickman File Browser or you can run it from the command line by preceding the file name with `./`: `./my-file.py`

## 2.3 Controlling the LEDs with a touch sensor

This code will turn the LEDs red whenever the touch sensor is pressed, and back to green when it's released. Plug a touch sensor into any sensor port before trying this out.

```
ts = TouchSensor()
leds = Leds()

print("Press the touch sensor to change the LED color!")

while True:
    if ts.is_pressed:
        leds.set_color("LEFT", "GREEN")
        leds.set_color("RIGHT", "GREEN")
    else:
        leds.set_color("LEFT", "RED")
        leds.set_color("RIGHT", "RED")
    # don't let this loop use 100% CPU
    sleep(0.01)
```

If you'd like to use a sensor on a specific port, specify the port like this:

```
ts = TouchSensor(INPUT_1)
```

*Heads-up:* If you are using a BrickPi instead of an EV3, you will need to manually configure the sensor. See the example here: https://github.com/ev3dev/ev3dev-lang-python-demo/blob/stretch/platform/brickpi3-motor-and-sensor.py

## 2.4 Running a single motor

This will run a LEGO Large Motor at 75% of maximum speed for 5 rotations.

```
m = LargeMotor(OUTPUT_A)
m.on_for_rotations(SpeedPercent(75), 5)
```

You can also run a motor for a number of degrees, an amount of time, or simply start it and let it run until you tell it to stop. Additionally, other units are also available. See the following pages for more information:

- http://python-ev3dev.readthedocs.io/en/ev3dev-stretch/motors.html#ev3dev.motor.Motor.on_for_degrees
- http://python-ev3dev.readthedocs.io/en/ev3dev-stretch/motors.html#units

## 2.5 Driving with two motors

The simplest drive control style is with the *MoveTank* class:

```
tank_drive = MoveTank(OUTPUT_A, OUTPUT_B)

# drive in a turn for 5 rotations of the outer motor
# the first two parameters can be unit classes or percentages.
tank_drive.on_for_rotations(SpeedPercent(50), SpeedPercent(75), 10)

# drive in a different turn for 3 seconds
tank_drive.on_for_seconds(SpeedPercent(60), SpeedPercent(30), 3)
```

There are also *MoveSteering* and *MoveJoystick* classes which provide different styles of control. See the following pages for more information:

- http://python-ev3dev.readthedocs.io/en/ev3dev-stretch/motors.html#multiple-motor-groups
- http://python-ev3dev.readthedocs.io/en/ev3dev-stretch/motors.html#units

## 2.6 Using text-to-speech

If you want to make your robot speak, you can use the `Sound.speak` method:

```
from ev3dev2.sound import Sound

sound = Sound()
sound.speak('Welcome to the E V 3 dev project!')
```

## 2.7 More Demo Code

There are several demo programs that you can run to get acquainted with this language binding. The programs are available at this GitHub site.

You can also copy and run the programs in the *utils* directory to understand some of the code constructs to use the EV3 motors, sensors, LCD console, buttons, sound, and LEDs.

We also highly recommend ev3python.com where one of our community members, @ndward, has put together a great website with detailed guides on using this library which are targeted at beginners. If you are just getting started with programming, we highly recommend that you check it out at ev3python.com!

CHAPTER 3

# Using Micropython

Normal Python too slow? Review Micropython to see if it supports the features your project needs.

CHAPTER 4

# Library Documentation

Class documentation for this library can be found on our Read the Docs page. You can always go there to get information on how you can use this library's functionality.

# CHAPTER 5

# Frequently-Asked Questions

Experiencing an odd error or unsure of how to do something that seems simple? Check our our FAQ to see if there's an existing answer.

**Contents**

## 5.1 Using python-ev3dev with MicroPython

The core modules of this library are shipped as a module for MicroPython, which is faster to load and run on the EV3. If your app only requires functionality supported on MicroPython, we recommend you run your code with it for improved performance.

### 5.1.1 Module support

| Module | Support status |
|---|---|
| *ev3dev2.button* | ✓ |
| *ev3dev2.console* | ✓ |
| *ev3dev2.control*[1] | |
| *ev3dev2.display*[2] | |
| *ev3dev2.fonts*[3] | |
| *ev3dev2.led* | ✓ |
| *ev3dev2.motor* | ✓ |
| *ev3dev2.port* | ✓ |
| *ev3dev2.power* | ✓ |
| *ev3dev2.sensor.** | ✓ |
| *ev3dev2.sound* | ✓ |
| *ev3dev2.unit* | ✓ |
| *ev3dev2.wheel* | ✓ |

### 5.1.2 Differences from standard Python (CPython)

See the MicroPython differences page for language information.

#### Shebang

You should modify the first line of your scripts to replace "python3" with "micropython":

```
#!/usr/bin/env micropython
```

#### Running from the command line

If you previously would have typed `python3 foo.py`, you should now type `micropython foo.py`.

If you are running programs via an SSH shell to your EV3, use the following command line to prevent Brickman from interfering:

```
brickrun -- ./program.py
```

## 5.2 Upgrading from ev3dev-jessie (library v1) to ev3dev-stretch (library v2)

With ev3dev-stretch, we have introduced some breaking changes that you must be aware of to get older scripts running with new features.

**Scripts which worked on ev3dev-jessie are still supported and will continue to work as-is on Stretch.** However, if you want to use any of the new features we have introduced, you will need to switch to using version 2 of the python-ev3dev library. You can switch to version 2 by updating your import statements.

### 5.2.1 Updating import statements

Previously, we recommended using one of the following as your `import` declaration:

```
import ev3dev.ev3 as ev3
import ev3dev.brickpi as ev3
import ev3dev.auto as ev3
```

We have re-arranged the library to provide more control over what gets imported. For all platforms, you will now import from individual modules for things like sensors and motors, like this:

```
from ev3dev2.motor import Motor, OUTPUT_A
from ev3dev2.sensor.lego import TouchSensor, UltrasonicSensor
```

The platform (EV3, BrickPi, etc.) will now be automatically determined.

You can omit import statements for modules you don't need, and add any additional ones that you do require. With this style of import, members are globally available by their name, so you would now refer to the Motor class as simply `Motor` rather than `ev3.Motor`.

[1] Untested/low-priority, but some of it might work.
[2] `ev3dev2.display` isn't implemented. Use `ev3dev2.console` for text-only, using ANSI codes to the EV3 LCD console.
[3] `ev3dev2.console` supports the system fonts, but the fonts for `ev3dev2.display` do not work.

### 5.2.2 Remove references to `connected` attribute

In version 1 of the library, instantiating a device such as a motor or sensor would always succeed without an error. To see if the device connected successfully you would have to check the `connected` attribute. With the new version of the module, the constructor of device classes will throw an `ev3dev2.DeviceNotConnected` exception. You will need to remove any uses of the `connected` attribute.

### 5.2.3 `Screen` class has been renamed to `Display`

To match the name used by LEGO's "EV3-G" graphical programming tools, we have renamed the `Screen` module to `Display`.

### 5.2.4 Reorganization of `RemoteControl`, `BeaconSeeker` and `InfraredSensor`

The `RemoteControl` and `BeaconSeeker` classes have been removed; you will now use `InfraredSensor` for all purposes.

Additionally, we have renamed many of the properties on the `InfraredSensor` class to make the meaning more obvious. Check out the `InfraredSensor` documentation for more info.

### 5.2.5 Re-designed `Sound` class

The names and interfaces of some of the `Sound` class methods have changed. Check out the `Sound` class docs for details.

## 5.3 Once you've adapted to breaking changes, check out the cool new features!

- New classes are available for coordinating motors: *ev3dev2.motor.MotorSet*, *ev3dev2.motor.MoveTank*, *ev3dev2.motor.MoveSteering*, and *ev3dev2.motor.MoveJoystick*.
- Classes representing a variety of motor speed units are available and accepted by many of the motor interfaces: see *Units*.
- Friendlier interfaces for operating motors and sensors: check out *ev3dev2.motor.Motor.on_for_rotations()* and the other `on_for_*` methods on motors.
- Easier interactivity via buttons: each button now has `wait_for_pressed`, `wait_for_released` and `wait_for_bump`
- Improved *ev3dev2.sound.Sound* and *ev3dev2.display.Display* interfaces
- New color conversion methods in *ev3dev2.sensor.lego.ColorSensor*

## 5.4 API reference

### 5.4.1 Device interfaces

**Contents:**

### Motor classes



### Units

Most methods which run motors will accept a `speed` argument. While this can be provided as an integer which will be interpreted as a percentage of max speed, you can also specify an instance of any of the following classes, each of which represents a different unit system:

**class** ev3dev2.motor.**SpeedValue**
A base class for other unit types. Don't use this directly; instead, see *`SpeedPercent`*, *`SpeedRPS`*, *`SpeedRPM`*, *`SpeedDPS`*, and *`SpeedDPM`*.

**class** ev3dev2.motor.**SpeedPercent**(*percent*, *desc=None*)
Speed as a percentage of the motor's maximum rated speed.

**class** ev3dev2.motor.**SpeedNativeUnits**(*native_counts*, *desc=None*)
Speed in tacho counts per second.

**class** ev3dev2.motor.**SpeedRPS**(*rotations_per_second*, *desc=None*)
Speed in rotations-per-second.

**class** ev3dev2.motor.**SpeedRPM**(*rotations_per_minute*, *desc=None*)
Speed in rotations-per-minute.

**class** ev3dev2.motor.**SpeedDPS**(*degrees_per_second*, *desc=None*)
Speed in degrees-per-second.

**class** ev3dev2.motor.**SpeedDPM**(*degrees_per_minute*, *desc=None*)
Speed in degrees-per-minute.

Example:

```
from ev3dev2.motor import SpeedRPM

# later...

# rotates the motor at 200 RPM (rotations-per-minute) for five seconds.
my_motor.on_for_seconds(SpeedRPM(200), 5)
```

### Common motors

#### Tacho Motor (`Motor`)

**class** ev3dev2.motor.**Motor**(*address=None*, *name_pattern='*'*, *name_exact=False*, ***kwargs*)

Bases: *ev3dev2.Device*

The motor class provides a uniform interface for using motors with positional and directional feedback such as the EV3 and NXT motors. This feedback allows for precise control of the motors. This is the most common type of motor, so we just call it `motor`.

**COMMAND_RUN_FOREVER = 'run-forever'**
Run the motor until another command is sent.

**COMMAND_RUN_TO_ABS_POS = 'run-to-abs-pos'**
Run to an absolute position specified by `position_sp` and then stop using the action specified in `stop_action`.

**COMMAND_RUN_TO_REL_POS = 'run-to-rel-pos'**
Run to a position relative to the current `position` value. The new position will be current `position` + `position_sp`. When the new position is reached, the motor will stop using the action specified by `stop_action`.

**COMMAND_RUN_TIMED = 'run-timed'**
Run the motor for the amount of time specified in `time_sp` and then stop the motor using the action specified by `stop_action`.

**COMMAND_RUN_DIRECT = 'run-direct'**
Run the motor at the duty cycle specified by `duty_cycle_sp`. Unlike other run commands, changing `duty_cycle_sp` while running *will* take effect immediately.

**COMMAND_STOP = 'stop'**
Stop any of the run commands before they are complete using the action specified by `stop_action`.

**COMMAND_RESET = 'reset'**
Reset all of the motor parameter attributes to their default value. This will also have the effect of stopping the motor.

**ENCODER_POLARITY_NORMAL = 'normal'**
Sets the normal polarity of the rotary encoder.

**ENCODER_POLARITY_INVERSED = 'inversed'**
Sets the inversed polarity of the rotary encoder.

**POLARITY_NORMAL = 'normal'**
With `normal` polarity, a positive duty cycle will cause the motor to rotate clockwise.

**POLARITY_INVERSED = 'inversed'**
With `inversed` polarity, a positive duty cycle will cause the motor to rotate counter-clockwise.

**STATE_RUNNING = 'running'**
Power is being sent to the motor.

**STATE_RAMPING = 'ramping'**
The motor is ramping up or down and has not yet reached a constant output level.

**STATE_HOLDING = 'holding'**
The motor is not turning, but rather attempting to hold a fixed position.

**STATE_OVERLOADED = 'overloaded'**
The motor is turning, but cannot reach its `speed_sp`.

**STATE_STALLED = 'stalled'**
The motor is not turning when it should be.

**STOP_ACTION_COAST = 'coast'**
Power will be removed from the motor and it will freely coast to a stop.

**STOP_ACTION_BRAKE = 'brake'**
Power will be removed from the motor and a passive electrical load will be placed on the motor. This is usually done by shorting the motor terminals together. This load will absorb the energy from the rotation of the motors and cause the motor to stop more quickly than coasting.

**STOP_ACTION_HOLD = 'hold'**
Does not remove power from the motor. Instead it actively try to hold the motor at the current position. If an external force tries to turn the motor, the motor will `push back` to maintain its position.

**address**
Returns the name of the port that this motor is connected to.

**command**
Sends a command to the motor controller. See `commands` for a list of possible values.

**commands**
Returns a list of commands that are supported by the motor controller. Possible values are `run-forever`, `run-to-abs-pos`, `run-to-rel-pos`, `run-timed`, `run-direct`, `stop` and `reset`. Not all commands may be supported.

- `run-forever` will cause the motor to run until another command is sent.
- `run-to-abs-pos` will run to an absolute position specified by `position_sp` and then stop using the action specified in `stop_action`.
- `run-to-rel-pos` will run to a position relative to the current `position` value. The new position will be current `position` + `position_sp`. When the new position is reached, the motor will stop using the action specified by `stop_action`.
- `run-timed` will run the motor for the amount of time specified in `time_sp` and then stop the motor using the action specified by `stop_action`.
- `run-direct` will run the motor at the duty cycle specified by `duty_cycle_sp`. Unlike other run commands, changing `duty_cycle_sp` while running *will* take effect immediately.
- `stop` will stop any of the run commands before they are complete using the action specified by `stop_action`.
- `reset` will reset all of the motor parameter attributes to their default value. This will also have the effect of stopping the motor.

**count_per_rot**
Returns the number of tacho counts in one rotation of the motor. Tacho counts are used by the position and speed attributes, so you can use this value to convert rotations or degrees to tacho counts. (rotation motors only)

**count_per_m**
Returns the number of tacho counts in one meter of travel of the motor. Tacho counts are used by the position and speed attributes, so you can use this value to convert from distance to tacho counts. (linear motors only)

**driver_name**
Returns the name of the driver that provides this tacho motor device.

**duty_cycle**
Returns the current duty cycle of the motor. Units are percent. Values are -100 to 100.

**duty_cycle_sp**
Writing sets the duty cycle setpoint. Reading returns the current value. Units are in percent. Valid values are -100 to 100. A negative value causes the motor to rotate in reverse.

**full_travel_count**
Returns the number of tacho counts in the full travel of the motor. When combined with the `count_per_m` atribute, you can use this value to calculate the maximum travel distance of the motor. (linear motors only)

**polarity**
Sets the polarity of the motor. With `normal` polarity, a positive duty cycle will cause the motor to rotate clockwise. With `inversed` polarity, a positive duty cycle will cause the motor to rotate counter-clockwise. Valid values are `normal` and `inversed`.

**position**
Returns the current position of the motor in pulses of the rotary encoder. When the motor rotates clockwise, the position will increase. Likewise, rotating counter-clockwise causes the position to decrease. Writing will set the position to that value.

**position_p**
The proportional constant for the position PID.

**position_i**
The integral constant for the position PID.

**position_d**
The derivative constant for the position PID.

**position_sp**
Writing specifies the target position for the `run-to-abs-pos` and `run-to-rel-pos` commands. Reading returns the current value. Units are in tacho counts. You can use the value returned by `count_per_rot` to convert tacho counts to/from rotations or degrees.

**max_speed**
Returns the maximum value that is accepted by the `speed_sp` attribute. This may be slightly different than the maximum speed that a particular motor can reach - it's the maximum theoretical speed.

**speed**
Returns the current motor speed in tacho counts per second. Note, this is not necessarily degrees (although it is for LEGO motors). Use the `count_per_rot` attribute to convert this value to RPM or deg/sec.

**speed_sp**
Writing sets the target speed in tacho counts per second used for all `run-*` commands except `run-direct`. Reading returns the current value. A negative value causes the motor to rotate in reverse with the exception of `run-to-*-pos` commands where the sign is ignored. Use the `count_per_rot` attribute to convert RPM or deg/sec to tacho counts per second. Use the `count_per_m` attribute to convert m/s to tacho counts per second.

**ramp_up_sp**
Writing sets the ramp up setpoint. Reading returns the current value. Units are in milliseconds and must

be positive. When set to a non-zero value, the motor speed will increase from 0 to 100% of `max_speed` over the span of this setpoint. The actual ramp time is the ratio of the difference between the `speed_sp` and the current `speed` and max_speed multiplied by `ramp_up_sp`.

**ramp_down_sp**
Writing sets the ramp down setpoint. Reading returns the current value. Units are in milliseconds and must be positive. When set to a non-zero value, the motor speed will decrease from 0 to 100% of `max_speed` over the span of this setpoint. The actual ramp time is the ratio of the difference between the `speed_sp` and the current `speed` and max_speed multiplied by `ramp_down_sp`.

**speed_p**
The proportional constant for the speed regulation PID.

**speed_i**
The integral constant for the speed regulation PID.

**speed_d**
The derivative constant for the speed regulation PID.

**state**
Reading returns a list of state flags. Possible flags are `running`, `ramping`, `holding`, `overloaded` and `stalled`.

**stop_action**
Reading returns the current stop action. Writing sets the stop action. The value determines the motors behavior when `command` is set to `stop`. Also, it determines the motors behavior when a run command completes. See `stop_actions` for a list of possible values.

**stop_actions**
Returns a list of stop actions supported by the motor controller. Possible values are `coast`, `brake` and `hold`. `coast` means that power will be removed from the motor and it will freely coast to a stop. `brake` means that power will be removed from the motor and a passive electrical load will be placed on the motor. This is usually done by shorting the motor terminals together. This load will absorb the energy from the rotation of the motors and cause the motor to stop more quickly than coasting. `hold` does not remove power from the motor. Instead it actively tries to hold the motor at the current position. If an external force tries to turn the motor, the motor will 'push back' to maintain its position.

**time_sp**
Writing specifies the amount of time the motor will run when using the `run-timed` command. Reading returns the current value. Units are in milliseconds.

**run_forever**(***kwargs*)
Run the motor until another command is sent.

**run_to_abs_pos**(***kwargs*)
Run to an absolute position specified by `position_sp` and then stop using the action specified in `stop_action`.

**run_to_rel_pos**(***kwargs*)
Run to a position relative to the current `position` value. The new position will be current `position` + `position_sp`. When the new position is reached, the motor will stop using the action specified by `stop_action`.

**run_timed**(***kwargs*)
Run the motor for the amount of time specified in `time_sp` and then stop the motor using the action specified by `stop_action`.

**run_direct**(***kwargs*)
Run the motor at the duty cycle specified by `duty_cycle_sp`. Unlike other run commands, changing `duty_cycle_sp` while running *will* take effect immediately.

**stop**(***kwargs*)
Stop any of the run commands before they are complete using the action specified by `stop_action`.

**reset**(***kwargs*)
Reset all of the motor parameter attributes to their default value. This will also have the effect of stopping the motor.

**is_running**
Power is being sent to the motor.

**is_ramping**
The motor is ramping up or down and has not yet reached a constant output level.

**is_holding**
The motor is not turning, but rather attempting to hold a fixed position.

**is_overloaded**
The motor is turning, but cannot reach its `speed_sp`.

**is_stalled**
The motor is not turning when it should be.

**wait**(*cond*, *timeout=None*)
Blocks until `cond(self.state)` is `True`. The condition is checked when there is an I/O event related to the `state` attribute. Exits early when `timeout` (in milliseconds) is reached.

Returns `True` if the condition is met, and `False` if the timeout is reached.

**wait_until_not_moving**(*timeout=None*)
Blocks until `running` is not in `self.state` or `stalled` is in `self.state`. The condition is checked when there is an I/O event related to the `state` attribute. Exits early when `timeout` (in milliseconds) is reached.

Returns `True` if the condition is met, and `False` if the timeout is reached.

Example:

```
m.wait_until_not_moving()
```

**wait_until**(*s*, *timeout=None*)
Blocks until `s` is in `self.state`. The condition is checked when there is an I/O event related to the `state` attribute. Exits early when `timeout` (in milliseconds) is reached.

Returns `True` if the condition is met, and `False` if the timeout is reached.

Example:

```
m.wait_until('stalled')
```

**wait_while**(*s*, *timeout=None*)
Blocks until `s` is not in `self.state`. The condition is checked when there is an I/O event related to the `state` attribute. Exits early when `timeout` (in milliseconds) is reached.

Returns `True` if the condition is met, and `False` if the timeout is reached.

Example:

```
m.wait_while('running')
```

**on_for_rotations**(*speed*, *rotations*, *brake=True*, *block=True*)
Rotate the motor at `speed` for `rotations`

`speed` can be a percentage or a *ev3dev2.motor.SpeedValue* object, enabling use of other units.

**on_for_degrees**(*speed*, *degrees*, *brake=True*, *block=True*)
Rotate the motor at `speed` for `degrees`

`speed` can be a percentage or a *ev3dev2.motor.SpeedValue* object, enabling use of other units.

**on_to_position**(*speed*, *position*, *brake=True*, *block=True*)
Rotate the motor at `speed` to `position`

`speed` can be a percentage or a *ev3dev2.motor.SpeedValue* object, enabling use of other units.

**on_for_seconds**(*speed*, *seconds*, *brake=True*, *block=True*)
Rotate the motor at `speed` for `seconds`

`speed` can be a percentage or a *ev3dev2.motor.SpeedValue* object, enabling use of other units.

**on**(*speed*, *brake=True*, *block=False*)
Rotate the motor at `speed` for forever

`speed` can be a percentage or a *ev3dev2.motor.SpeedValue* object, enabling use of other units.

Note that `block` is False by default, this is different from the other `on_for_XYZ` methods.

### Large EV3 Motor

**class** ev3dev2.motor.**LargeMotor**(*address=None*, *name_pattern='*'*, *name_exact=False*, ***kwargs*)

Bases: *ev3dev2.motor.Motor*

EV3/NXT large servo motor.

Same as *Motor*, except it will only successfully initialize if it finds a "large" motor.

### Medium EV3 Motor

**class** ev3dev2.motor.**MediumMotor**(*address=None*, *name_pattern='*'*, *name_exact=False*, ***kwargs*)

Bases: *ev3dev2.motor.Motor*

EV3 medium servo motor.

Same as *Motor*, except it will only successfully initialize if it finds a "medium" motor.

### Additional motors

### DC Motor

**class** ev3dev2.motor.**DcMotor**(*address=None*, *name_pattern='motor*'*, *name_exact=False*, ***kwargs*)

Bases: *ev3dev2.Device*

The DC motor class provides a uniform interface for using regular DC motors with no fancy controls or feedback. This includes LEGO MINDSTORMS RCX motors and LEGO Power Functions motors.

**address**
Returns the name of the port that this motor is connected to.

**command**
Sets the command for the motor. Possible values are `run-forever`, `run-timed` and `stop`. Not all commands may be supported, so be sure to check the contents of the `commands` attribute.

**commands**
Returns a list of commands supported by the motor controller.

**driver_name**
Returns the name of the motor driver that loaded this device. See the list of [supported devices] for a list of drivers.

**duty_cycle**
Shows the current duty cycle of the PWM signal sent to the motor. Values are -100 to 100 (-100% to 100%).

**duty_cycle_sp**
Writing sets the duty cycle setpoint of the PWM signal sent to the motor. Valid values are -100 to 100 (-100% to 100%). Reading returns the current setpoint.

**polarity**
Sets the polarity of the motor. Valid values are `normal` and `inversed`.

**ramp_down_sp**
Sets the time in milliseconds that it take the motor to ramp down from 100% to 0%. Valid values are 0 to 10000 (10 seconds). Default is 0.

**ramp_up_sp**
Sets the time in milliseconds that it take the motor to up ramp from 0% to 100%. Valid values are 0 to 10000 (10 seconds). Default is 0.

**state**
Gets a list of flags indicating the motor status. Possible flags are `running` and `ramping`. `running` indicates that the motor is powered. `ramping` indicates that the motor has not yet reached the `duty_cycle_sp`.

**stop_action**
Sets the stop action that will be used when the motor stops. Read `stop_actions` to get the list of valid values.

**stop_actions**
Gets a list of stop actions. Valid values are `coast` and `brake`.

**time_sp**
Writing specifies the amount of time the motor will run when using the `run-timed` command. Reading returns the current value. Units are in milliseconds.

**COMMAND_RUN_FOREVER = 'run-forever'**
Run the motor until another command is sent.

**COMMAND_RUN_TIMED = 'run-timed'**
Run the motor for the amount of time specified in `time_sp` and then stop the motor using the action specified by `stop_action`.

**COMMAND_RUN_DIRECT = 'run-direct'**
Run the motor at the duty cycle specified by `duty_cycle_sp`. Unlike other run commands, changing `duty_cycle_sp` while running *will* take effect immediately.

**COMMAND_STOP = 'stop'**
Stop any of the run commands before they are complete using the action specified by `stop_action`.

**POLARITY_NORMAL = 'normal'**
With `normal` polarity, a positive duty cycle will cause the motor to rotate clockwise.

**POLARITY_INVERSED = 'inversed'**
With `inversed` polarity, a positive duty cycle will cause the motor to rotate counter-clockwise.

**STOP_ACTION_COAST = 'coast'**
Power will be removed from the motor and it will freely coast to a stop.

**STOP_ACTION_BRAKE = 'brake'**
Power will be removed from the motor and a passive electrical load will be placed on the motor. This is usually done by shorting the motor terminals together. This load will absorb the energy from the rotation of the motors and cause the motor to stop more quickly than coasting.

**run_forever**(***kwargs*)
Run the motor until another command is sent.

**run_timed**(***kwargs*)
Run the motor for the amount of time specified in `time_sp` and then stop the motor using the action specified by `stop_action`.

**run_direct**(***kwargs*)
Run the motor at the duty cycle specified by `duty_cycle_sp`. Unlike other run commands, changing `duty_cycle_sp` while running *will* take effect immediately.

**stop**(***kwargs*)
Stop any of the run commands before they are complete using the action specified by `stop_action`.

### Servo Motor

**class** ev3dev2.motor.**ServoMotor**(*address=None*, *name_pattern='motor*'*, *name_exact=False*, ***kwargs*)
Bases: *ev3dev2.Device*

The servo motor class provides a uniform interface for using hobby type servo motors.

**address**
Returns the name of the port that this motor is connected to.

**command**
Sets the command for the servo. Valid values are `run` and `float`. Setting to `run` will cause the servo to be driven to the position_sp set in the `position_sp` attribute. Setting to `float` will remove power from the motor.

**driver_name**
Returns the name of the motor driver that loaded this device. See the list of [supported devices] for a list of drivers.

**max_pulse_sp**
Used to set the pulse size in milliseconds for the signal that tells the servo to drive to the maximum (clockwise) position_sp. Default value is 2400. Valid values are 2300 to 2700. You must write to the position_sp attribute for changes to this attribute to take effect.

**mid_pulse_sp**
Used to set the pulse size in milliseconds for the signal that tells the servo to drive to the mid position_sp. Default value is 1500. Valid values are 1300 to 1700. For example, on a 180 degree servo, this would be 90 degrees. On continuous rotation servo, this is the 'neutral' position_sp where the motor does not turn. You must write to the position_sp attribute for changes to this attribute to take effect.

**min_pulse_sp**
Used to set the pulse size in milliseconds for the signal that tells the servo to drive to the miniumum (counter-clockwise) position_sp. Default value is 600. Valid values are 300 to 700. You must write to the position_sp attribute for changes to this attribute to take effect.

**polarity**
Sets the polarity of the servo. Valid values are `normal` and `inversed`. Setting the value to `inversed`

will cause the position_sp value to be inversed. i.e `-100` will correspond to `max_pulse_sp`, and `100` will correspond to `min_pulse_sp`.

**position_sp**
Reading returns the current position_sp of the servo. Writing instructs the servo to move to the specified position_sp. Units are percent. Valid values are -100 to 100 (-100% to 100%) where `-100` corresponds to `min_pulse_sp`, `0` corresponds to `mid_pulse_sp` and `100` corresponds to `max_pulse_sp`.

**rate_sp**
Sets the rate_sp at which the servo travels from 0 to 100.0% (half of the full range of the servo). Units are in milliseconds. Example: Setting the rate_sp to 1000 means that it will take a 180 degree servo 2 second to move from 0 to 180 degrees. Note: Some servo controllers may not support this in which case reading and writing will fail with `-EOPNOTSUPP`. In continuous rotation servos, this value will affect the rate_sp at which the speed ramps up or down.

**state**
Returns a list of flags indicating the state of the servo. Possible values are: * `running`: Indicates that the motor is powered.

**COMMAND_RUN = 'run'**
Drive servo to the position set in the `position_sp` attribute.

**COMMAND_FLOAT = 'float'**
Remove power from the motor.

**POLARITY_NORMAL = 'normal'**
With `normal` polarity, a positive duty cycle will cause the motor to rotate clockwise.

**POLARITY_INVERSED = 'inversed'**
With `inversed` polarity, a positive duty cycle will cause the motor to rotate counter-clockwise.

**run**(***kwargs*)
Drive servo to the position set in the `position_sp` attribute.

**float**(***kwargs*)
Remove power from the motor.

### Actuonix L12 50 Linear Servo Motor

**class** ev3dev2.motor.**ActuonixL1250Motor**(*address=None*, *name_pattern='linear*'*, *name_exact=False*, ***kwargs*)

Bases: *ev3dev2.motor.Motor*

Actuonix L12 50 linear servo motor.

Same as *Motor*, except it will only successfully initialize if it finds an Actuonix L12 50 linear servo motor

### Actuonix L12 100 Linear Servo Motor

**class** ev3dev2.motor.**ActuonixL12100Motor**(*address=None*, *name_pattern='linear*'*, *name_exact=False*, ***kwargs*)

Bases: *ev3dev2.motor.Motor*

Actuonix L12 100 linear servo motor.

Same as *Motor*, except it will only successfully initialize if it finds an Actuonix L12 100linear servo motor

### Multiple-motor groups

### Motor Set

**class** ev3dev2.motor.**MotorSet**(*motor_specs*, *desc=None*)

**off**(*motors=None*, *brake=True*)
Stop motors immediately. Configure motors to brake if `brake` is set.

**stop**(*motors=None*, *brake=True*)
`stop` is an alias of `off`. This is deprecated but helps keep the API for MotorSet somewhat similar to Motor which has both `stop` and `off`.

### Move Tank

**class** ev3dev2.motor.**MoveTank**(*left_motor_port*, *right_motor_port*, *desc=None*, *motor_class=<class 'ev3dev2.motor.LargeMotor'>*)
Bases: *ev3dev2.motor.MotorSet*

Controls a pair of motors simultaneously, via individual speed setpoints for each motor.

Example:

```
tank_drive = MoveTank(OUTPUT_A, OUTPUT_B)
# drive in a turn for 10 rotations of the outer motor
tank_drive.on_for_rotations(50, 75, 10)
```

**on_for_degrees**(*left_speed*, *right_speed*, *degrees*, *brake=True*, *block=True*)
Rotate the motors at 'left_speed & right_speed' for 'degrees'. Speeds can be percentages or any SpeedValue implementation.

If the left speed is not equal to the right speed (i.e., the robot will turn), the motor on the outside of the turn will rotate for the full `degrees` while the motor on the inside will have its requested distance calculated according to the expected turn.

**on_for_rotations**(*left_speed*, *right_speed*, *rotations*, *brake=True*, *block=True*)
Rotate the motors at 'left_speed & right_speed' for 'rotations'. Speeds can be percentages or any SpeedValue implementation.

If the left speed is not equal to the right speed (i.e., the robot will turn), the motor on the outside of the turn will rotate for the full `rotations` while the motor on the inside will have its requested distance calculated according to the expected turn.

**on_for_seconds**(*left_speed*, *right_speed*, *seconds*, *brake=True*, *block=True*)
Rotate the motors at 'left_speed & right_speed' for 'seconds'. Speeds can be percentages or any SpeedValue implementation.

**on**(*left_speed*, *right_speed*)
Start rotating the motors according to `left_speed` and `right_speed` forever. Speeds can be percentages or any SpeedValue implementation.

**follow_line**(*kp*, *ki*, *kd*, *speed*, *target_light_intensity=None*, *follow_left_edge=True*, *white=60*, *off_line_count_max=20*, *sleep_time=0.01*, *follow_for=<function follow_for_forever>*, ***kwargs*)
PID line follower

`kp`, `ki`, and `kd` are the PID constants.

`speed` is the desired speed of the midpoint of the robot

**`target_light_intensity` is the reflected light intensity when the color sensor** is on the edge of the line. If this is None we assume that the color sensor is on the edge of the line and will take a reading to set this variable.

`follow_left_edge` determines if we follow the left or right edge of the line

**`white` is the reflected_light_intensity that is used to determine if we have** lost the line

**`off_line_count_max` is how many consecutive times through the loop the** reflected_light_intensity must be greater than `white` before we declare the line lost and raise an exception

**`sleep_time` is how many seconds we sleep on each pass through** the loop. This is to give the robot a chance to react to the new motor settings. This should be something small such as 0.01 (10ms).

**`follow_for` is called to determine if we should keep following the** line or stop. This function will be passed `self` (the current `MoveTank` object). Current supported options are: - `follow_for_forever` - `follow_for_ms`

`**kwargs` will be passed to the `follow_for` function

Example:

```
from ev3dev2.motor import OUTPUT_A, OUTPUT_B, MoveTank, SpeedPercent, follow_
↪for_ms
from ev3dev2.sensor.lego import ColorSensor

tank = MoveTank(OUTPUT_A, OUTPUT_B)
tank.cs = ColorSensor()

try:
    # Follow the line for 4500ms
    tank.follow_line(
        kp=11.3, ki=0.05, kd=3.2,
        speed=SpeedPercent(30),
        follow_for=follow_for_ms,
        ms=4500
    )
except LineFollowErrorTooFast:
    tank.stop()
    raise
```

**`follow_gyro_angle`**(*kp*, *ki*, *kd*, *speed*, *target_angle=0*, *sleep_time=0.01*, *follow_for=<function follow_for_forever>*, ***kwargs*)

PID gyro angle follower

`kp`, `ki`, and `kd` are the PID constants.

`speed` is the desired speed of the midpoint of the robot

`target_angle` is the angle we want to maintain

**`sleep_time` is how many seconds we sleep on each pass through** the loop. This is to give the robot a chance to react to the new motor settings. This should be something small such as 0.01 (10ms).

**`follow_for` is called to determine if we should keep following the** desired angle or stop. This function will be passed `self` (the current `MoveTank` object). Current supported options are: - `follow_for_forever` - `follow_for_ms`

`**kwargs` will be passed to the `follow_for` function

Example:

```
from ev3dev2.motor import OUTPUT_A, OUTPUT_B, MoveTank, SpeedPercent, follow_
↪for_ms
from ev3dev2.sensor.lego import GyroSensor

# Instantiate the MoveTank object
tank = MoveTank(OUTPUT_A, OUTPUT_B)

# Initialize the tank's gyro sensor
tank.gyro = GyroSensor()

# Calibrate the gyro to eliminate drift, and to initialize the current angle␣
↪as 0
tank.gyro.calibrate()

try:

    # Follow the target_angle for 4500ms
    tank.follow_gyro_angle(
        kp=11.3, ki=0.05, kd=3.2,
        speed=SpeedPercent(30),
        target_angle=0,
        follow_for=follow_for_ms,
        ms=4500
    )
except FollowGyroAngleErrorTooFast:
    tank.stop()
    raise
```

**turn_degrees**(*speed*, *target_angle*, *brake=True*, *error_margin=2*, *sleep_time=0.01*)

Use a GyroSensor to rotate in place for `target_angle`

`speed` is the desired speed of the midpoint of the robot

`target_angle` is the number of degrees we want to rotate

`brake` hit the brakes once we reach `target_angle`

`error_margin` is the +/- angle threshold to control how accurate the turn should be

**`sleep_time` is how many seconds we sleep on each pass through** the loop. This is to give the robot a chance to react to the new motor settings. This should be something small such as 0.01 (10ms).

Rotate in place for `target_degrees` at `speed`

Example:

```
from ev3dev2.motor import OUTPUT_A, OUTPUT_B, MoveTank, SpeedPercent
from ev3dev2.sensor.lego import GyroSensor

# Instantiate the MoveTank object
tank = MoveTank(OUTPUT_A, OUTPUT_B)

# Initialize the tank's gyro sensor
tank.gyro = GyroSensor()

# Calibrate the gyro to eliminate drift, and to initialize the current angle␣
↪as 0
tank.gyro.calibrate()
```

(continues on next page)

(continued from previous page)

```
# Pivot 30 degrees
tank.turn_degrees(
    speed=SpeedPercent(5),
    target_angle=30
)
```

**turn_right**(*speed*, *degrees*, *brake=True*, *error_margin=2*, *sleep_time=0.01*)
Rotate clockwise `degrees` in place

**turn_left**(*speed*, *degrees*, *brake=True*, *error_margin=2*, *sleep_time=0.01*)
Rotate counter-clockwise `degrees` in place

### Move Steering

**class** ev3dev2.motor.**MoveSteering**(*left_motor_port*, *right_motor_port*, *desc=None*, *motor_class=<class 'ev3dev2.motor.LargeMotor'>*)

Bases: *ev3dev2.motor.MoveTank*

Controls a pair of motors simultaneously, via a single "steering" value and a speed.

**steering [-100, 100]:**

- -100 means turn left on the spot (right motor at 100% forward, left motor at 100% backward),
- 0 means drive in a straight line, and
- 100 means turn right on the spot (left motor at 100% forward, right motor at 100% backward).

"steering" can be any number between -100 and 100.

Example:

```
steering_drive = MoveSteering(OUTPUT_A, OUTPUT_B)
# drive in a turn for 10 rotations of the outer motor
steering_drive.on_for_rotations(-20, SpeedPercent(75), 10)
```

**on_for_rotations**(*steering*, *speed*, *rotations*, *brake=True*, *block=True*)
Rotate the motors according to the provided `steering`.

The distance each motor will travel follows the rules of *MoveTank.on_for_rotations()*.

**on_for_degrees**(*steering*, *speed*, *degrees*, *brake=True*, *block=True*)
Rotate the motors according to the provided `steering`.

The distance each motor will travel follows the rules of *MoveTank.on_for_degrees()*.

**on_for_seconds**(*steering*, *speed*, *seconds*, *brake=True*, *block=True*)
Rotate the motors according to the provided `steering` for `seconds`.

**on**(*steering*, *speed*)
Start rotating the motors according to the provided `steering` and `speed` forever.

**get_speed_steering**(*steering*, *speed*)
Calculate the speed_sp for each motor in a pair to achieve the specified steering. Note that calling this function alone will not make the motors move, it only calculates the speed. A run_* function must be called afterwards to make the motors move.

**steering [-100, 100]:**

- -100 means turn left on the spot (right motor at 100% forward, left motor at 100% backward),

- 0 means drive in a straight line, and
- 100 means turn right on the spot (left motor at 100% forward, right motor at 100% backward).

**speed:** The speed that should be applied to the outmost motor (the one rotating faster). The speed of the other motor will be computed automatically.

### Move Joystick

**class** ev3dev2.motor.**MoveJoystick**(*left_motor_port*, *right_motor_port*, *desc=None*, *motor_class=<class 'ev3dev2.motor.LargeMotor'>*)

Bases: *ev3dev2.motor.MoveTank*

Used to control a pair of motors via a single joystick vector.

**on**(*x*, *y*, *radius=100.0*)

Convert x,''y'' joystick coordinates to left/right motor speed percentages and move the motors.

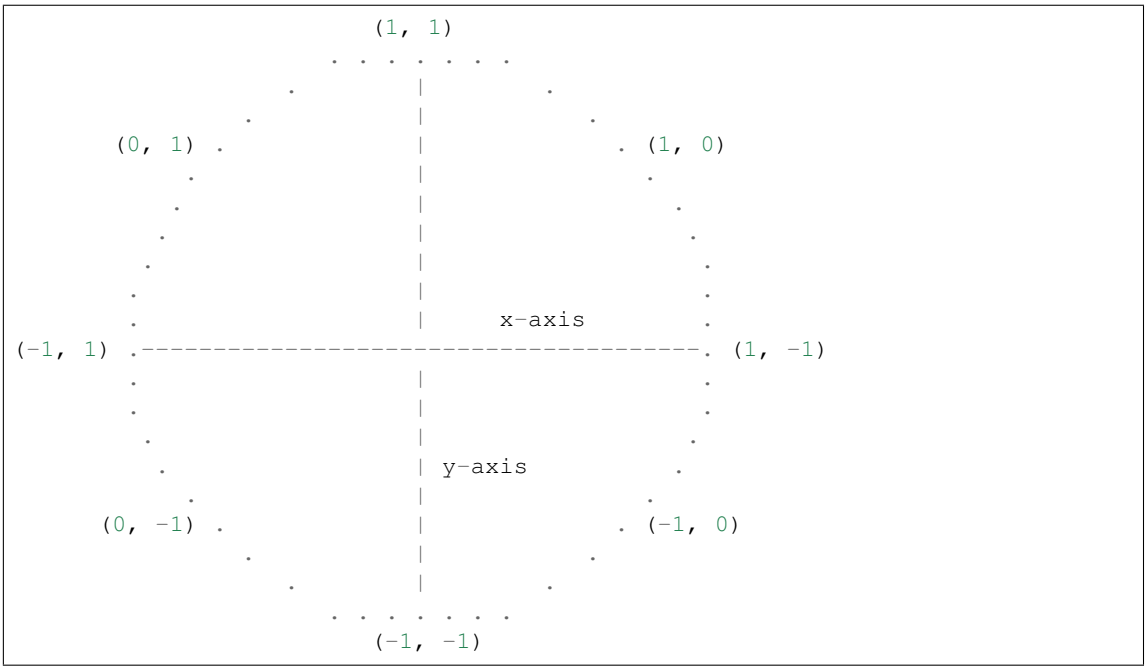
This will use a classic "arcade drive" algorithm: a full-forward joystick goes straight forward and likewise for full-backward. Pushing the joystick all the way to one side will make it turn on the spot in that direction. Positions in the middle will control how fast the vehicle moves and how sharply it turns.

**x, y:** The X and Y coordinates of the joystick's position, with (0,0) representing the center position. X is horizontal and Y is vertical.

**radius (default 100):** The radius of the joystick, controlling the range of the input (x, y) values. e.g. if "x" and "y" can be between -1 and 1, radius should be set to "1".

**static angle_to_speed_percentage**(*angle*)

The following graphic illustrates the **motor power outputs** for the left and right motors based on where the joystick is pointing, of the form `(left power, right power)`:




The joystick is a circle within a circle where the (x, y) coordinates of the joystick form an angle with the x-axis. Our job is to translate this angle into the percentage of power that should be sent to each motor. For instance if the joystick is moved all the way to the top of the circle we want both motors to move forward with 100% power...that is represented above by (1, 1). If the joystick is moved all the way to the right

side of the circle we want to rotate clockwise so we move the left motor forward 100% and the right motor backwards 100%...so (1, -1). If the joystick is at 45 degrees then we move apply (1, 0) to move the left motor forward 100% and the right motor stays still.

The 8 points shown above are pretty easy. For the points in between those 8 we do some math to figure out what the percentages should be. Take 11.25 degrees for example. We look at how the motors transition from 0 degrees to 45 degrees: - the left motor is 1 so that is easy - the right motor moves from -1 to 0

We determine how far we are between 0 and 45 degrees (11.25 is 25% of 45) so we know that the right motor should be 25% of the way from -1 to 0...so -0.75 is the percentage for the right motor at 11.25 degrees.

### Move Differential

**class** ev3dev2.motor.**MoveDifferential**(*left_motor_port*, *right_motor_port*, *wheel_class*, *wheel_distance_mm*, *desc=None*, *motor_class=<class 'ev3dev2.motor.LargeMotor'>*)

Bases: *ev3dev2.motor.MoveTank*

MoveDifferential is a child of MoveTank that adds the following capabilities:

- drive in a straight line for a specified distance
- rotate in place in a circle (clockwise or counter clockwise) for a specified number of degrees
- drive in an arc (clockwise or counter clockwise) of a specified radius for a specified distance

Odometry can be use to enable driving to specific coordinates and rotating to a specific angle.

New arguments:

wheel_class - Typically a child class of *ev3dev2.wheel.Wheel*. This is used to get the circumference of the wheels of the robot. The circumference is needed for several calculations in this class.

wheel_distance_mm - The distance between the mid point of the two wheels of the robot. You may need to do some test drives to find the correct value for your robot. It is not as simple as measuring the distance between the midpoints of the two wheels. The weight of the robot, center of gravity, etc come into play.

You can use utils/move_differential.py to call on_arc_left() to do some test drives of circles with a radius of 200mm. Adjust your wheel_distance_mm until your robot can drive in a perfect circle and stop exactly where it started. It does not have to be a circle with a radius of 200mm, you can test with any size circle but you do not want it to be too small or it will be difficult to test small adjustments to wheel_distance_mm.

Example:

```
from ev3dev2.motor import OUTPUT_A, OUTPUT_B, MoveDifferential, SpeedRPM
from ev3dev2.wheel import EV3Tire

STUD_MM = 8

# test with a robot that:
# - uses the standard wheels known as EV3Tire
# - wheels are 16 studs apart
mdiff = MoveDifferential(OUTPUT_A, OUTPUT_B, EV3Tire, 16 * STUD_MM)

# Rotate 90 degrees clockwise
mdiff.turn_right(SpeedRPM(40), 90)

# Drive forward 500 mm
mdiff.on_for_distance(SpeedRPM(40), 500)
```

(continues on next page)

(continued from previous page)

```
# Drive in arc to the right along an imaginary circle of radius 150 mm.
# Drive for 700 mm around this imaginary circle.
mdiff.on_arc_right(SpeedRPM(80), 150, 700)

# Enable odometry
mdiff.odometry_start()

# Use odometry to drive to specific coordinates
mdiff.on_to_coordinates(SpeedRPM(40), 300, 300)

# Use odometry to go back to where we started
mdiff.on_to_coordinates(SpeedRPM(40), 0, 0)

# Use odometry to rotate in place to 90 degrees
mdiff.turn_to_angle(SpeedRPM(40), 90)

# Disable odometry
mdiff.odometry_stop()
```

**on_for_distance**(*speed*, *distance_mm*, *brake=True*, *block=True*)
Drive in a straight line for `distance_mm`

**on_arc_right**(*speed*, *radius_mm*, *distance_mm*, *brake=True*, *block=True*)
Drive clockwise in a circle with 'radius_mm' for 'distance_mm'

**on_arc_left**(*speed*, *radius_mm*, *distance_mm*, *brake=True*, *block=True*)
Drive counter-clockwise in a circle with 'radius_mm' for 'distance_mm'

**turn_degrees**(*speed*, *degrees*, *brake=True*, *block=True*, *error_margin=2*, *use_gyro=False*)
Rotate in place `degrees`. Both wheels must turn at the same speed for us to rotate in place. If the following conditions are met the GryoSensor will be used to improve the accuracy of our turn: - `use_gyro`, `brake` and `block` are all True - A GyroSensor has been defined via `self.gyro = GyroSensor()`

**turn_right**(*speed*, *degrees*, *brake=True*, *block=True*, *error_margin=2*, *use_gyro=False*)
Rotate clockwise `degrees` in place

**turn_left**(*speed*, *degrees*, *brake=True*, *block=True*, *error_margin=2*, *use_gyro=False*)
Rotate counter-clockwise `degrees` in place

**turn_to_angle**(*speed*, *angle_target_degrees*, *brake=True*, *block=True*, *error_margin=2*, *use_gyro=False*)
Rotate in place to `angle_target_degrees` at `speed`

**odometry_start**(*theta_degrees_start=90.0*, *x_pos_start=0.0*, *y_pos_start=0.0*, *sleep_time=0.005*)
Ported from: http://seattlerobotics.org/encoder/200610/Article3/IMU%20Odometry,%20by%20David%20Anderson.htm

A thread is started that will run until the user calls odometry_stop() which will set odometry_thread_run to False

**odometry_stop**()
Signal the odometry thread to exit

**on_to_coordinates**(*speed*, *x_target_mm*, *y_target_mm*, *brake=True*, *block=True*)
Drive to (`x_target_mm`, `y_target_mm`) coordinates at `speed`

### Sensor classes



*Note:* If you are using a BrickPi rather than an EV3, you will need to manually configure the ports before interacting with your sensors. See the example here.

### Dedicated sensor classes

These classes derive from `ev3dev2.sensor.Sensor` and provide helper functions specific to the corresponding sensor type. Each provides sensible property accessors for the main functionality of the sensor.

### Touch Sensor

**class** ev3dev2.sensor.lego.**TouchSensor**(*address=None*, *name_pattern='sensor*'*, *name_exact=False*, ***kwargs*)

Bases: `ev3dev2.sensor.Sensor`

Touch Sensor

**MODE_TOUCH = 'TOUCH'**
Button state

**is_pressed**
A boolean indicating whether the current touch sensor is being pressed.

**wait_for_pressed**(*timeout_ms=None*, *sleep_ms=10*)
Wait for the touch sensor to be pressed down.

**wait_for_released**(*timeout_ms=None*, *sleep_ms=10*)
Wait for the touch sensor to be released.

**wait_for_bump**(*timeout_ms=None*, *sleep_ms=10*)
Wait for the touch sensor to be pressed down and then released. Both actions must happen within timeout_ms.

### Color Sensor

**class** ev3dev2.sensor.lego.**ColorSensor**(*address=None*, *name_pattern='sensor*'*, *name_exact=False*, ***kwargs*)

Bases: `ev3dev2.sensor.Sensor`

LEGO EV3 color sensor.

**MODE_COL_REFLECT = 'COL-REFLECT'**
Reflected light. Red LED on.

**MODE_COL_AMBIENT = 'COL-AMBIENT'**
Ambient light. Blue LEDs on.

**MODE_COL_COLOR = 'COL-COLOR'**
Color. All LEDs rapidly cycling, appears white.

**MODE_REF_RAW = 'REF-RAW'**
Raw reflected. Red LED on

**MODE_RGB_RAW = 'RGB-RAW'**
Raw Color Components. All LEDs rapidly cycling, appears white.

**COLOR_NOCOLOR = 0**
No color.

**COLOR_BLACK = 1**
Black color.

**COLOR_BLUE = 2**
Blue color.

**COLOR_GREEN = 3**
Green color.

**COLOR_YELLOW = 4**
Yellow color.

**COLOR_RED = 5**
Red color.

**COLOR_WHITE = 6**
White color.

**COLOR_BROWN = 7**
Brown color.

**reflected_light_intensity**
Reflected light intensity as a percentage (0 to 100). Light on sensor is red.

**ambient_light_intensity**
Ambient light intensity, as a percentage (0 to 100). Light on sensor is dimly lit blue.

**color**

**Color detected by the sensor, categorized by overall value.**

- 0: No color
- 1: Black
- 2: Blue
- 3: Green
- 4: Yellow
- 5: Red
- 6: White
- 7: Brown

**color_name**
Returns NoColor, Black, Blue, etc

**raw**
Red, green, and blue components of the detected color, as a tuple.

Officially in the range 0-1020 but the values returned will never be that high. We do not yet know why the values returned are low, but pointing the color sensor at a well lit sheet of white paper will return values in the 250-400 range.

If this is an issue, check out the rgb() and calibrate_white() methods.

**calibrate_white**()
The RGB raw values are on a scale of 0-1020 but you never see a value anywhere close to 1020. This function is designed to be called when the sensor is placed over a white object in order to figure out what are the maximum RGB values the robot can expect to see. We will use these maximum values to scale future raw values to a 0-255 range in rgb().

If you never call this function red_max, green_max, and blue_max will use a default value of 300. This default was selected by measuring the RGB values of a white sheet of paper in a well lit room.

Note that there are several variables that influence the maximum RGB values detected by the color sensor - the distance of the color sensor to the white object - the amount of light in the room - shadows that the robot casts on the sensor

**rgb**
Same as raw() but RGB values are scaled to 0-255

**lab**
Return colors in Lab color space

**hsv**
HSV: Hue, Saturation, Value H: position in the spectrum S: color saturation ("purity") V: color brightness

**hls**
HLS: Hue, Luminance, Saturation H: position in the spectrum L: color lightness S: color saturation

**red**
Red component of the detected color, in the range 0-1020.

**green**
Green component of the detected color, in the range 0-1020.

**blue**
Blue component of the detected color, in the range 0-1020.

### Ultrasonic Sensor

**class** ev3dev2.sensor.lego.**UltrasonicSensor**(*address=None*, *name_pattern='sensor*'*, *name_exact=False*, ***kwargs*)

Bases: *ev3dev2.sensor.Sensor*

LEGO EV3 ultrasonic sensor.

**MODE_US_DIST_CM = 'US-DIST-CM'**
Continuous measurement in centimeters.

**MODE_US_DIST_IN = 'US-DIST-IN'**
Continuous measurement in inches.

**MODE_US_LISTEN = 'US-LISTEN'**
Listen.

**MODE_US_SI_CM = 'US-SI-CM'**
Single measurement in centimeters.

**MODE_US_SI_IN = 'US-SI-IN'**
Single measurement in inches.

**distance_centimeters_continuous**
Measurement of the distance detected by the sensor, in centimeters.

The sensor will continue to take measurements so they are available for future reads.

Prefer using the equivalent *UltrasonicSensor.distance_centimeters()* property.

**distance_centimeters_ping**
Measurement of the distance detected by the sensor, in centimeters.

The sensor will take a single measurement then stop broadcasting.

If you use this property too frequently (e.g. every 100msec), the sensor will sometimes lock up and writing to the mode attribute will return an error. A delay of 250msec between each usage seems sufficient to keep the sensor from locking up.

**distance_centimeters**
Measurement of the distance detected by the sensor, in centimeters.

Equivalent to *UltrasonicSensor.distance_centimeters_continuous()*.

**distance_inches_continuous**
Measurement of the distance detected by the sensor, in inches.

The sensor will continue to take measurements so they are available for future reads.

Prefer using the equivalent *UltrasonicSensor.distance_inches()* property.

**distance_inches_ping**
Measurement of the distance detected by the sensor, in inches.

The sensor will take a single measurement then stop broadcasting.

If you use this property too frequently (e.g. every 100msec), the sensor will sometimes lock up and writing to the mode attribute will return an error. A delay of 250msec between each usage seems sufficient to keep the sensor from locking up.

**distance_inches**
Measurement of the distance detected by the sensor, in inches.

Equivalent to *UltrasonicSensor.distance_inches_continuous()*.

**other_sensor_present**
Boolean indicating whether another ultrasonic sensor could be heard nearby.

### Gyro Sensor

**class** ev3dev2.sensor.lego.**GyroSensor**(*address=None*, *name_pattern='sensor*'*, *name_exact=False*, ***kwargs*)

Bases: *ev3dev2.sensor.Sensor*

LEGO EV3 gyro sensor.

**MODE_GYRO_ANG = 'GYRO-ANG'**
Angle

**MODE_GYRO_RATE = 'GYRO-RATE'**
Rotational speed

**MODE_GYRO_FAS = 'GYRO-FAS'**
Raw sensor value

**MODE_GYRO_G_A = 'GYRO-G&A'**
Angle and rotational speed

**MODE_GYRO_CAL = 'GYRO-CAL'**
Calibration ???

**angle**
The number of degrees that the sensor has been rotated since it was put into this mode.

**rate**
The rate at which the sensor is rotating, in degrees/second.

**angle_and_rate**
Angle (degrees) and Rotational Speed (degrees/second).

**calibrate**()
The robot should be perfectly still when you call this

**reset**()
Resets the angle to 0.

**Caveats:**

- This function only resets the angle to 0, it does not fix drift.
- This function only works on EV3, it does not work on BrickPi, PiStorms, or with any sensor multiplexors.

**wait_until_angle_changed_by**(*delta*, *direction_sensitive=False*)
Wait until angle has changed by specified amount.

If `direction_sensitive` is True we will wait until angle has changed by `delta` and with the correct sign.

If `direction_sensitive` is False (default) we will wait until angle has changed by `delta` in either direction.

**circle_angle**()
As the gryo rotates clockwise the angle increases, it will increase by 360 for each full rotation. As the gyro rotates counter-clockwise the gyro angle will decrease.

The angles on a circle have the opposite behavior though, they start at 0 and increase as you move counter-clockwise around the circle.

Convert the gyro angle to the angle on a circle. We consider the initial position of the gyro to be at 90 degrees on the cirlce.

### Infrared Sensor

**class** ev3dev2.sensor.lego.**InfraredSensor**(*address=None*, *name_pattern='sensor*'*, *name_exact=False*, ***kwargs*)
Bases: *ev3dev2.sensor.Sensor*, ev3dev2.button.ButtonBase

LEGO EV3 infrared sensor.

**MODE_IR_PROX = 'IR-PROX'**
Proximity

**MODE_IR_SEEK = 'IR-SEEK'**
IR Seeker

**MODE_IR_REMOTE = 'IR-REMOTE'**
IR Remote Control

**MODE_IR_REM_A = 'IR-REM-A'**
IR Remote Control. State of the buttons is coded in binary

**MODE_IR_CAL = 'IR-CAL'**
Calibration ???

**on_channel1_top_left = None**
Handler for top-left button events on channel 1. See *InfraredSensor.process()*.

**on_channel1_bottom_left = None**
Handler for bottom-left button events on channel 1. See *InfraredSensor.process()*.

**on_channel1_top_right = None**
Handler for top-right button events on channel 1. See *InfraredSensor.process()*.

**on_channel1_bottom_right = None**
Handler for bottom-right button events on channel 1. See *InfraredSensor.process()*.

**on_channel1_beacon = None**
Handler for beacon button events on channel 1. See *InfraredSensor.process()*.

**on_channel2_top_left = None**
Handler for top-left button events on channel 2. See *InfraredSensor.process()*.

**on_channel2_bottom_left = None**
Handler for bottom-left button events on channel 2. See *InfraredSensor.process()*.

**on_channel2_top_right = None**
Handler for top-right button events on channel 2. See *InfraredSensor.process()*.

**on_channel2_bottom_right = None**
Handler for bottom-right button events on channel 2. See *InfraredSensor.process()*.

**on_channel2_beacon = None**
Handler for beacon button events on channel 2. See *InfraredSensor.process()*.

**on_channel3_top_left = None**
Handler for top-left button events on channel 3. See *InfraredSensor.process()*.

**on_channel3_bottom_left = None**
Handler for bottom-left button events on channel 3. See *InfraredSensor.process()*.

**on_channel3_top_right = None**
Handler for top-right button events on channel 3. See *InfraredSensor.process()*.

**on_channel3_bottom_right = None**
Handler for bottom-right button events on channel 3. See *InfraredSensor.process()*.

**on_channel3_beacon = None**
Handler for beacon button events on channel 3. See *InfraredSensor.process()*.

**on_channel4_top_left = None**
Handler for top-left button events on channel 4. See *InfraredSensor.process()*.

**on_channel4_bottom_left = None**
Handler for bottom-left button events on channel 4. See *InfraredSensor.process()*.

**on_channel4_top_right = None**
Handler for top-right button events on channel 4. See *InfraredSensor.process()*.

**on_channel4_bottom_right = None**
Handler for bottom-right button events on channel 4. See *InfraredSensor.process()*.

**on_channel4_beacon = None**
Handler for beacon button events on channel 4. See *InfraredSensor.process()*.

**proximity**
An estimate of the distance between the sensor and objects in front of it, as a percentage. 100% is approximately 70cm/27in.

**heading**(*channel=1*)
Returns heading (-25, 25) to the beacon on the given channel.

**distance**(*channel=1*)
Returns distance (0, 100) to the beacon on the given channel. Returns None when beacon is not found.

**heading_and_distance**(*channel=1*)
Returns heading and distance to the beacon on the given channel as a tuple.

**top_left**(*channel=1*)
Checks if `top_left` button is pressed.

**bottom_left**(*channel=1*)
Checks if `bottom_left` button is pressed.

**top_right**(*channel=1*)
Checks if `top_right` button is pressed.

**bottom_right**(*channel=1*)
Checks if `bottom_right` button is pressed.

**beacon**(*channel=1*)
Checks if `beacon` button is pressed.

**buttons_pressed**(*channel=1*)
Returns list of currently pressed buttons.

Note that the sensor can only identify up to two buttons pressed at once.

**process**()
Check for currenly pressed buttons. If the new state differs from the old state, call the appropriate button event handlers.

To use the on_channel1_top_left, etc handlers your program would do something like:

```
def top_left_channel_1_action(state):
    print("top left on channel 1: %s" % state)

def bottom_right_channel_4_action(state):
    print("bottom right on channel 4: %s" % state)

ir = InfraredSensor()
ir.on_channel1_top_left = top_left_channel_1_action
ir.on_channel4_bottom_right = bottom_right_channel_4_action

while True:
    ir.process()
    time.sleep(0.01)
```

### Sound Sensor

**class** ev3dev2.sensor.lego.**SoundSensor**(*address=None*, *name_pattern='sensor*'*, *name_exact=False*, ***kwargs*)

Bases: *ev3dev2.sensor.Sensor*

LEGO NXT Sound Sensor

**MODE_DB = 'DB'**
Sound pressure level. Flat weighting

**MODE_DBA = 'DBA'**
Sound pressure level. A weighting

**sound_pressure**
A measurement of the measured sound pressure level, as a percent. Uses a flat weighting.

**sound_pressure_low**
A measurement of the measured sound pressure level, as a percent. Uses A-weighting, which focuses on levels up to 55 dB.

### Light Sensor

**class** ev3dev2.sensor.lego.**LightSensor**(*address=None*, *name_pattern='sensor*'*, *name_exact=False*, ***kwargs*)

Bases: *ev3dev2.sensor.Sensor*

LEGO NXT Light Sensor

**MODE_REFLECT = 'REFLECT'**
Reflected light. LED on

**MODE_AMBIENT = 'AMBIENT'**
Ambient light. LED off

**reflected_light_intensity**
A measurement of the reflected light intensity, as a percentage.

**ambient_light_intensity**
A measurement of the ambient light intensity, as a percentage.

### Base "Sensor"

This is the base class all the other sensor classes are derived from. You generally want to use one of the other classes instead, but if your sensor doesn't have a dedicated class, this is will let you interface with it as a generic device.

**class** ev3dev2.sensor.**Sensor**(*address=None*, *name_pattern='sensor*'*, *name_exact=False*, ***kwargs*)

The sensor class provides a uniform interface for using most of the sensors available for the EV3.

**address**
Returns the name of the port that the sensor is connected to, e.g. `ev3:in1`. I2C sensors also include the I2C address (decimal), e.g. `ev3:in1:i2c8`.

**command**
Sends a command to the sensor.

**commands**
Returns a list of the valid commands for the sensor. Returns -EOPNOTSUPP if no commands are supported.

**decimals**
Returns the number of decimal places for the values in the `value<N>` attributes of the current mode.

**driver_name**
Returns the name of the sensor device/driver. See the list of [supported sensors] for a complete list of drivers.

**mode**
Returns the current mode. Writing one of the values returned by `modes` sets the sensor to that mode.

**modes**
Returns a list of the valid modes for the sensor.

**num_values**
Returns the number of `value<N>` attributes that will return a valid value for the current mode.

**units**
Returns the units of the measured value for the current mode. May return empty string

**value** (*n=0*)
Returns the value or values measured by the sensor. Check num_values to see how many values there are. Values with N >= num_values will return an error. The values are fixed point numbers, so check decimals to see if you need to divide to get the actual value.

**bin_data_format**
Returns the format of the values in `bin_data` for the current mode. Possible values are:

- `u8`: Unsigned 8-bit integer (byte)
- `s8`: Signed 8-bit integer (sbyte)
- `u16`: Unsigned 16-bit integer (ushort)
- `s16`: Signed 16-bit integer (short)
- `s16_be`: Signed 16-bit integer, big endian
- `s32`: Signed 32-bit integer (int)
- `float`: IEEE 754 32-bit floating point (float)

**bin_data** (*fmt=None*)
Returns the unscaled raw values in the `value<N>` attributes as raw byte array. Use `bin_data_format`, `num_values` and the individual sensor documentation to determine how to interpret the data.

Use `fmt` to unpack the raw bytes into a struct.

Example:

```
>>> from ev3dev2.sensor.lego import InfraredSensor
>>> ir = InfraredSensor()
>>> ir.value()
28
>>> ir.bin_data('<b')
(28,)
```

### Button

**class** ev3dev2.button.**Button**
EV3 Buttons

#### Event handlers

These will be called when state of the corresponding button is changed:

**on_up**

**on_down**

**on_left**

**on_right**

**on_enter**

**on_backspace**

#### Member functions and properties

**buttons_pressed**
Returns list of names of pressed buttons.

**any**()
Checks if any button is pressed.

**backspace**
Check if `backspace` button is pressed.

**check_buttons**(*buttons=[]*)
Check if currently pressed buttons exactly match the given list `buttons`.

**down**
Check if `down` button is pressed.

**enter**
Check if `enter` button is pressed.

**evdev_device**
Return our corresponding evdev device object

**left**
Check if `left` button is pressed.

**static on_change**(*changed_buttons*)
This handler is called by `process()` whenever state of any button has changed since last `process()` call. `changed_buttons` is a list of tuples of changed button names and their states.

**process**(*new_state=None*)
Check for currenly pressed buttons. If the `new_state` differs from the old state, call the appropriate button event handlers (on_up, on_down, etc).

**right**
Check if `right` button is pressed.

**up**
Check if `up` button is pressed.

**wait_for_bump**(*buttons*, *timeout_ms=None*)
Wait for `buttons` to be pressed down and then released. Both actions must happen within `timeout_ms`.

**wait_for_pressed**(*buttons*, *timeout_ms=None*)
Wait for `buttons` to be pressed down.

**wait_for_released**(*buttons*, *timeout_ms=None*)
Wait for `buttons` to be released.

### Leds

**class** ev3dev2.led.**Led**(*name_pattern='*'*, *name_exact=False*, *desc=None*, ***kwargs*)
Any device controlled by the generic LED driver. See https://www.kernel.org/doc/Documentation/leds/leds-class.txt for more details.

**max_brightness**
Returns the maximum allowable brightness value.

**brightness**
Sets the brightness level. Possible values are from 0 to `max_brightness`.

**triggers**
Returns a list of available triggers.

**trigger**
Sets the LED trigger. A trigger is a kernel based source of LED events. Triggers can either be simple or complex. A simple trigger isn't configurable and is designed to slot into existing subsystems with minimal additional code. Examples are the `ide-disk` and `nand-disk` triggers.

Complex triggers whilst available to all LEDs have LED specific parameters and work on a per LED basis. The `timer` trigger is an example. The `timer` trigger will periodically change the LED brightness between 0 and the current brightness setting. The `on` and `off` time can be specified via `delay_{on, off}` attributes in milliseconds. You can change the brightness value of a LED independently of the timer trigger. However, if you set the brightness value to 0 it will also disable the `timer` trigger.

**delay_on**
The `timer` trigger will periodically change the LED brightness between 0 and the current brightness setting. The `on` time can be specified via `delay_on` attribute in milliseconds.

**delay_off**
The `timer` trigger will periodically change the LED brightness between 0 and the current brightness setting. The `off` time can be specified via `delay_off` attribute in milliseconds.

**brightness_pct**
Returns LED brightness as a fraction of max_brightness

**class** ev3dev2.led.**Leds**

**set_color**(*group*, *color*, *pct=1*)
Sets brightness of LEDs in the given group to the values specified in color tuple. When percentage is specified, brightness of each LED is reduced proportionally.

Example:

```
my_leds = Leds()
my_leds.set_color('LEFT', 'AMBER')
```

With a custom color:

```
my_leds = Leds()
my_leds.set_color('LEFT', (0.5, 0.3))
```

**set**(*group*, ***kwargs*)
Set attributes for each LED in group.

Example:

```
my_leds = Leds()
my_leds.set_color('LEFT', brightness_pct=0.5, trigger='timer')
```

**all_off**()
Turn all LEDs off

**reset**()
Put all LEDs back to their default color

**animate_stop**()
Signal the current animation thread to exit and wait for it to exit

**animate_police_lights**(*color1*, *color2*, *group1='LEFT'*, *group2='RIGHT'*, *sleeptime=0.5*, *duration=5*, *block=True*)
Cycle the `group1` and `group2` LEDs between `color1` and `color2` to give the effect of police lights. Alternate the `group1` and `group2` LEDs every `sleeptime` seconds.

Animate for `duration` seconds. If `duration` is None animate for forever.

Example:

```
from ev3dev2.led import Leds
leds = Leds()
leds.animate_police_lights('RED', 'GREEN', sleeptime=0.75, duration=10)
```

**animate_flash**(*color*, *groups=('LEFT', 'RIGHT')*, *sleeptime=0.5*, *duration=5*, *block=True*)
Turn all LEDs in `groups` off/on to `color` every `sleeptime` seconds

Animate for `duration` seconds. If `duration` is None animate for forever.

Example:

```
from ev3dev2.led import Leds
leds = Leds()
leds.animate_flash('AMBER', sleeptime=0.75, duration=10)
```

**animate_cycle**(*colors*, *groups=('LEFT', 'RIGHT')*, *sleeptime=0.5*, *duration=5*, *block=True*)
Cycle `groups` LEDs through `colors`. Do this in a loop where we display each color for `sleeptime` seconds.

Animate for `duration` seconds. If `duration` is None animate for forever.

Example:

```
from ev3dev2.led import Leds
leds = Leds()
leds.animate_cycle(('RED', 'GREEN', 'AMBER'))
```

**animate_rainbow**(*group1='LEFT'*, *group2='RIGHT'*, *increment_by=0.1*, *sleeptime=0.1*, *duration=5*, *block=True*)
Gradually fade from one color to the next

Animate for `duration` seconds. If `duration` is None animate for forever.

Example:

```
from ev3dev2.led import Leds
leds = Leds()
leds.animate_rainbow()
```

### LED group and color names

#### EV3 platform

Led groups:

- LEFT
- RIGHT

Colors:

- BLACK
- RED
- GREEN
- AMBER
- ORANGE
- YELLOW

#### BrickPI platform

Led groups:

- LED1
- LED2

Colors:

- BLACK
- BLUE

#### BrickPI3 platform

Led groups:

- LED

Colors:

- BLACK
- BLUE

### PiStorms platform

Led groups:

- `LEFT`
- `RIGHT`

Colors:

- `BLACK`
- `RED`
- `GREEN`
- `BLUE`
- `YELLOW`
- `CYAN`
- `MAGENTA`

### EVB platform

None.

### Power Supply

**class** ev3dev2.power.**PowerSupply**(*address=None*, *name_pattern='*'*, *name_exact=False*, ***kwargs*)

A generic interface to read data from the system's power_supply class. Uses the built-in legoev3-battery if none is specified.

**measured_current**
The measured current that the battery is supplying (in microamps)

**measured_voltage**
The measured voltage that the battery is supplying (in microvolts)

**measured_amps**
The measured current that the battery is supplying (in amps)

**measured_volts**
The measured voltage that the battery is supplying (in volts)

### Sound

**class** ev3dev2.sound.**Sound**

Support beep, play wav files, or convert text to speech.

Examples:

```
from ev3dev2.sound import Sound

spkr = Sound()

# Play 'bark.wav':
```

(continues on next page)

(continued from previous page)

```
spkr.play_file('bark.wav')

# Introduce yourself:
spkr.speak('Hello, I am Robot')

# Play a small song
spkr.play_song((
    ('D4', 'e3'),
    ('D4', 'e3'),
    ('D4', 'e3'),
    ('G4', 'h'),
    ('D5', 'h')
))
```

In order to mimic EV3-G API parameters, durations used in methods exposed as EV3-G blocks for sound related operations are expressed as a float number of seconds.

**PLAY_WAIT_FOR_COMPLETE = 0**
Play the sound and block until it is complete

**PLAY_NO_WAIT_FOR_COMPLETE = 1**
Start playing the sound but return immediately

**PLAY_LOOP = 2**
Never return; start the sound immediately after it completes, until the program is killed

**beep**(*args=''*, *play_type=0*)
Call beep command with the provided arguments (if any). See beep man page and google linux beep music for inspiration.

**Parameters**

- **args** (*string*) – Any additional arguments to be passed to `beep` (see the beep man page for details)
- **play_type** (`Sound.PLAY_WAIT_FOR_COMPLETE` or `Sound.PLAY_NO_WAIT_FOR_COMPLETE`) – The behavior of `beep` once playback has been initiated

**Returns** When python3 is used and `Sound.PLAY_NO_WAIT_FOR_COMPLETE` is specified, returns the spawn subprocess from `subprocess.Popen`; `None` otherwise

**tone**(**args*, *play_type=0*)

#### tone(tone_sequence)

Play tone sequence.

Here is a cheerful example:

```
my_sound = Sound()
my_sound.tone([
    (392, 350, 100), (392, 350, 100), (392, 350, 100), (311.1, 250, 100),
    (466.2, 25, 100), (392, 350, 100), (311.1, 250, 100), (466.2, 25, 100),
    (392, 700, 100), (587.32, 350, 100), (587.32, 350, 100),
    (587.32, 350, 100), (622.26, 250, 100), (466.2, 25, 100),
    (369.99, 350, 100), (311.1, 250, 100), (466.2, 25, 100), (392, 700, 100),
    (784, 350, 100), (392, 250, 100), (392, 25, 100), (784, 350, 100),
```

(continues on next page)

(continued from previous page)

```
    (739.98, 250, 100), (698.46, 25, 100), (659.26, 25, 100),
    (622.26, 25, 100), (659.26, 50, 400), (415.3, 25, 200), (554.36, 350,␣
↪100),
    (523.25, 250, 100), (493.88, 25, 100), (466.16, 25, 100), (440, 25, 100),
    (466.16, 50, 400), (311.13, 25, 200), (369.99, 350, 100),
    (311.13, 250, 100), (392, 25, 100), (466.16, 350, 100), (392, 250, 100),
    (466.16, 25, 100), (587.32, 700, 100), (784, 350, 100), (392, 250, 100),
    (392, 25, 100), (784, 350, 100), (739.98, 250, 100), (698.46, 25, 100),
    (659.26, 25, 100), (622.26, 25, 100), (659.26, 50, 400), (415.3, 25, 200),
    (554.36, 350, 100), (523.25, 250, 100), (493.88, 25, 100),
    (466.16, 25, 100), (440, 25, 100), (466.16, 50, 400), (311.13, 25, 200),
    (392, 350, 100), (311.13, 250, 100), (466.16, 25, 100),
    (392.00, 300, 150), (311.13, 250, 100), (466.16, 25, 100), (392, 700)
    ])
```

Have also a look at *play_song()* for a more musician-friendly way of doing, which uses the conventional notation for notes and durations.

**Parameters**

- **tone_sequence** (*list[tuple(float,float,float)]*) – The sequence of tones to play. The first number of each tuple is frequency in Hz, the second is duration in milliseconds, and the third is delay in milliseconds between this and the next tone in the sequence.
- **play_type** (Sound.PLAY_WAIT_FOR_COMPLETE or Sound.PLAY_NO_WAIT_FOR_COMPLETE) – The behavior of tone once playback has been initiated

**Returns** When python3 is used and Sound.PLAY_NO_WAIT_FOR_COMPLETE is specified, returns the spawn subprocess from subprocess.Popen; None otherwise

### tone(frequency, duration)

Play single tone of given frequency and duration.

**Parameters**

- **frequency** (*float*) – The frequency of the tone in Hz
- **duration** (*float*) – The duration of the tone in milliseconds
- **play_type** (Sound.PLAY_WAIT_FOR_COMPLETE or Sound.PLAY_NO_WAIT_FOR_COMPLETE) – The behavior of tone once playback has been initiated

**Returns** When python3 is used and Sound.PLAY_NO_WAIT_FOR_COMPLETE is specified, returns the spawn subprocess from subprocess.Popen; None otherwise

**play_tone** (*frequency*, *duration*, *delay=0.0*, *volume=100*, *play_type=0*)
Play a single tone, specified by its frequency, duration, volume and final delay.

**Parameters**

- **frequency** (*int*) – the tone frequency, in Hertz
- **duration** (*float*) – Tone duration, in seconds
- **delay** (*float*) – Delay after tone, in seconds (can be useful when chaining calls to play_tone)

- **volume** (*int*) – The play volume, in percent of maximum volume
- **play_type** (Sound.PLAY_WAIT_FOR_COMPLETE, Sound.PLAY_NO_WAIT_FOR_COMPLETE or Sound.PLAY_LOOP) – The behavior of play_tone once playback has been initiated

**Returns** When python3 is used and Sound.PLAY_NO_WAIT_FOR_COMPLETE is specified, returns the PID of the underlying beep command; None otherwise

**Raises** **ValueError** – if invalid parameter

**play_note**(*note*, *duration*, *volume=100*, *play_type=0*)

Plays a note, given by its name as defined in _NOTE_FREQUENCIES.

**Parameters**

- **note** (*string*) – The note symbol with its octave number
- **duration** (*float*) – Tone duration, in seconds
- **volume** (*int*) – The play volume, in percent of maximum volume
- **play_type** (Sound.PLAY_WAIT_FOR_COMPLETE, Sound.PLAY_NO_WAIT_FOR_COMPLETE or Sound.PLAY_LOOP) – The behavior of play_note once playback has been initiated

**Returns** When python3 is used and Sound.PLAY_NO_WAIT_FOR_COMPLETE is specified, returns the PID of the underlying beep command; None otherwise

**Raises** **ValueError** – is invalid parameter (note, duration,...)

**play_file**(*wav_file*, *volume=100*, *play_type=0*)

Play a sound file (wav format) at a given volume. The EV3 audio subsystem will work best if the file is encoded as 16-bit, mono, 22050Hz.

**Parameters**

- **wav_file** (*string*) – The sound file path
- **volume** (*int*) – The play volume, in percent of maximum volume
- **play_type** (Sound.PLAY_WAIT_FOR_COMPLETE, Sound.PLAY_NO_WAIT_FOR_COMPLETE or Sound.PLAY_LOOP) – The behavior of play_file once playback has been initiated

**Returns** When python3 is used and Sound.PLAY_NO_WAIT_FOR_COMPLETE is specified, returns the spawn subprocess from subprocess.Popen; None otherwise

**speak**(*text*, *espeak_opts='-a 200 -s 130'*, *volume=100*, *play_type=0*)

Speak the given text aloud.

Uses the espeak external command.

**Parameters**

- **text** (*string*) – The text to speak
- **espeak_opts** (*string*) – espeak command options (advanced usage)
- **volume** (*int*) – The play volume, in percent of maximum volume
- **play_type** (Sound.PLAY_WAIT_FOR_COMPLETE, Sound.PLAY_NO_WAIT_FOR_COMPLETE or Sound.PLAY_LOOP) – The behavior of speak once playback has been initiated

**Returns** When python3 is used and `Sound.PLAY_NO_WAIT_FOR_COMPLETE` is specified, returns the spawn subprocess from `subprocess.Popen`; `None` otherwise

**set_volume**(*pct*, *channel=None*)

Sets the sound volume to the given percentage [0-100] by calling `amixer -q set <channel> <pct>%`. If the channel is not specified, it tries to determine the default one by running `amixer scontrols`. If that fails as well, it uses the `Playback` channel, as that is the only channel on the EV3.

**get_volume**(*channel=None*)

Gets the current sound volume by parsing the output of `amixer get <channel>`. If the channel is not specified, it tries to determine the default one by running `amixer scontrols`. If that fails as well, it uses the `Playback` channel, as that is the only channel on the EV3.

**play_song**(*song*, *tempo=120*, *delay=0.05*)

Plays a song provided as a list of tuples containing the note name and its value using music conventional notation instead of numerical values for frequency and duration.

It supports symbolic notes (e.g. `A4`, `D#3`, `Gb5`) and durations (e.g. `q`, `h`). You can also specify rests by using `R` instead of note pitch.

For an exhaustive list of accepted note symbols and values, have a look at the `_NOTE_FREQUENCIES` and `_NOTE_VALUES` private dictionaries in the source code.

The value can be suffixed by modifiers:

- a *divider* introduced by a `/` to obtain triplets for instance (e.g. `q/3` for a triplet of eight note)
- a *multiplier* introduced by `*` (e.g. `*1.5` is a dotted note).

Shortcuts exist for common modifiers:

- `3` produces a triplet member note. For instance `e3` gives a triplet of eight notes, i.e. 3 eight notes in the duration of a single quarter. You must ensure that 3 triplets notes are defined in sequence to match the count, otherwise the result will not be the expected one.
- `.` produces a dotted note, i.e. which duration is one and a half the base one. Double dots are not currently supported.

Example:

```
>>> # A long time ago in a galaxy far,
>>> # far away...
>>> from ev3dev2.sound import Sound
>>> spkr = Sound()
>>> spkr.play_song((
>>>     ('D4', 'e3'),      # intro anacrouse
>>>     ('D4', 'e3'),
>>>     ('D4', 'e3'),
>>>     ('G4', 'h'),       # meas 1
>>>     ('D5', 'h'),
>>>     ('C5', 'e3'),      # meas 2
>>>     ('B4', 'e3'),
>>>     ('A4', 'e3'),
>>>     ('G5', 'h'),
>>>     ('D5', 'q'),
>>>     ('C5', 'e3'),      # meas 3
>>>     ('B4', 'e3'),
>>>     ('A4', 'e3'),
>>>     ('G5', 'h'),
>>>     ('D5', 'q'),
```

(continues on next page)

(continued from previous page)

```
>>>     ('C5', 'e3'),       # meas 4
>>>     ('B4', 'e3'),
>>>     ('C5', 'e3'),
>>>     ('A4', 'h.'),
>>> ))
```

**Important:** Only 4/4 signature songs are supported with respect to note durations.

**Parameters**

- **song** (*iterable[tuple(string,string)]*) – the song
- **tempo** (*int*) – the song tempo, given in quarters per minute
- **delay** (*float*) – delay between notes (in seconds)

**Returns** When python3 is used the spawn subprocess from `subprocess.Popen` is returned; `None` otherwise

**Raises** **ValueError** – if invalid note in song or invalid play parameters

### Display

**class** ev3dev2.display.**Display**(*desc='Display'*)

Bases: ev3dev2.display.FbMem

A convenience wrapper for the FbMem class. Provides drawing functions from the python imaging library (PIL).

**xres**
Horizontal screen resolution

**yres**
Vertical screen resolution

**shape**
Dimensions of the screen.

**draw**
Returns a handle to PIL.ImageDraw.Draw class associated with the screen.

Example:

```
screen.draw.rectangle((10,10,60,20), fill='black')
```

**image**
Returns a handle to PIL.Image class that is backing the screen. This can be accessed for blitting images to the screen.

Example:

```
screen.image.paste(picture, (0, 0))
```

**clear**()
Clears the screen

**update**()
Applies pending changes to the screen. Nothing will be drawn on the screen until this function is called.

**line** (*clear_screen=True*, *x1=10*, *y1=10*, *x2=50*, *y2=50*, *line_color='black'*, *width=1*)
Draw a line from (x1, y1) to (x2, y2)

**circle** (*clear_screen=True*, *x=50*, *y=50*, *radius=40*, *fill_color='black'*, *outline_color='black'*)
Draw a circle of 'radius' centered at (x, y)

**rectangle** (*clear_screen=True*, *x1=10*, *y1=10*, *x2=80*, *y2=40*, *fill_color='black'*, *outline_color='black'*)
Draw a rectangle where the top left corner is at (x1, y1) and the bottom right corner is at (x2, y2)

**point** (*clear_screen=True*, *x=10*, *y=10*, *point_color='black'*)
Draw a single pixel at (x, y)

**text_pixels** (*text*, *clear_screen=True*, *x=0*, *y=0*, *text_color='black'*, *font=None*)
Display `text` starting at pixel (x, y).

The EV3 display is 178x128 pixels

- (0, 0) would be the top left corner of the display
- (89, 64) would be right in the middle of the display

`text_color` : PIL says it supports "common HTML color names". There are 140 HTML color names listed here that are supported by all modern browsers. This is probably a good list to start with. https://www.w3schools.com/colors/colors_names.asp

**font** [can be any font displayed here] http://ev3dev-lang.readthedocs.io/projects/python-ev3dev/en/ev3dev-stretch/display.html#bitmap-fonts

- If font is a string, it is the name of a font to be loaded.
- If font is a Font object, returned from *ev3dev2.fonts.load()*, then it is used directly. This is desirable for faster display times.

**text_grid** (*text*, *clear_screen=True*, *x=0*, *y=0*, *text_color='black'*, *font=None*)
Display `text` starting at grid (x, y)

The EV3 display can be broken down in a grid that is 22 columns wide and 12 rows tall. Each column is 8 pixels wide and each row is 10 pixels tall.

`text_color` : PIL says it supports "common HTML color names". There are 140 HTML color names listed here that are supported by all modern browsers. This is probably a good list to start with. https://www.w3schools.com/colors/colors_names.asp

**font** [can be any font displayed here] http://ev3dev-lang.readthedocs.io/projects/python-ev3dev/en/ev3dev-stretch/display.html#bitmap-fonts

- If font is a string, it is the name of a font to be loaded.
- If font is a Font object, returned from *ev3dev2.fonts.load()*, then it is used directly. This is desirable for faster display times.

### Bitmap fonts

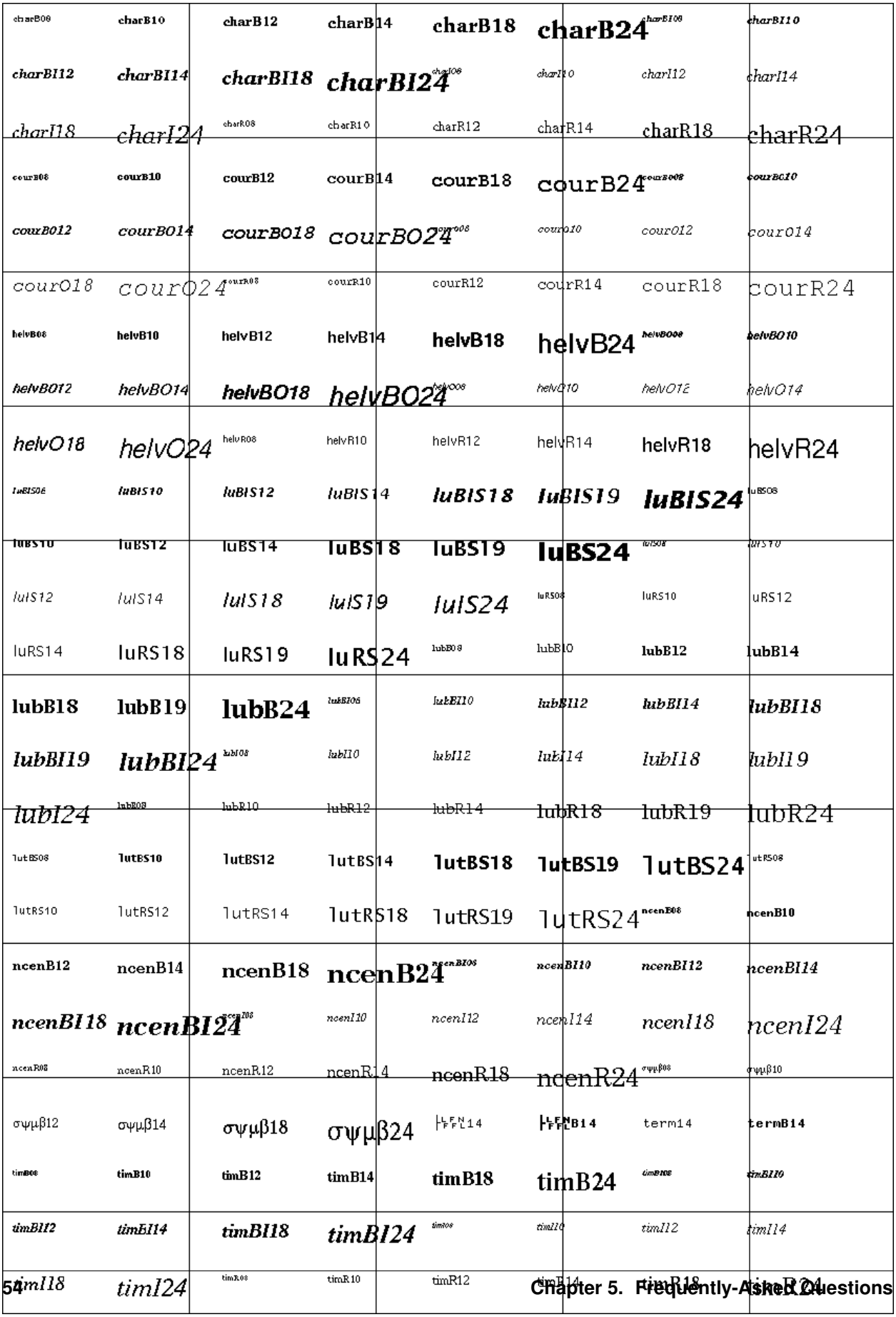
The *ev3dev2.display.Display* class allows to write text on the LCD using python imaging library (PIL) interface (see description of the `text()` method here). The `ev3dev2.fonts` module contains bitmap fonts in PIL format that should look good on a tiny EV3 screen:

```
import ev3dev2.fonts as fonts
display.draw.text((10,10), 'Hello World!', font=fonts.load('luBS14'))
```

ev3dev2.fonts.**available**()
Returns list of available font names.

ev3dev2.fonts.**load**(*name*)
Loads the font specified by name and returns it as an instance of PIL.ImageFont class.

The following image lists all available fonts. The grid lines correspond to EV3 screen size:



### Console

**class** ev3dev2.console.**Console**(*font='Lat15-TerminusBold24x12'*)

A class that represents the EV3 LCD console, which implements ANSI codes for cursor positioning, text color, and resetting the screen. Supports changing the console font using standard system fonts.

**columns**

Return (int) number of columns on the EV3 LCD console supported by the current font.

**rows**

Return (int) number of rows on the EV3 LCD console supported by the current font.

**echo**

Return (bool) whether the console echo mode is enabled.

**cursor**

Return (bool) whether the console cursor is visible.

**text_at**(*text*, *column=1*, *row=1*, *reset_console=False*, *inverse=False*, *alignment='L'*)

Display `text` (string) at grid position (`column`, `row`). Note that the grid locations are 1-based (not 0-based).

Depending on the font, the number of columns and rows supported by the EV3 LCD console can vary. Large fonts support as few as 11 columns and 4 rows, while small fonts support 44 columns and 21 rows. The default font for the Console() class results in a grid that is 14 columns and 5 rows.

Using the `inverse=True` parameter will display the `text` with more emphasis and contrast, as the background of the text will be black, and the foreground is white. Using inverse can help in certain situations, such as to indicate when a color sensor senses black, or the gyro sensor is pointing to zero.

Use the `alignment` parameter to enable the function to align the `text` differently to the column/row values passed-in. Use `L` for left-alignment (default), where the first character in the `text` will show at the column/row position. Use `R` for right-alignment, where the last character will show at the column/row position. Use `C` for center-alignment, where the text string will centered at the column/row position (as close as possible using integer division–odd-length text string will center better than even-length).

Parameters:

- `text` (string): Text to display
- `column` (int): LCD column position to start the text (1 = left column); text will wrap when it reaches the right edge
- `row` (int): LCD row position to start the text (1 = top row)
- `reset_console` (bool): `True` to reset the EV3 LCD console before showing the text; default is `False`
- `inverse` (bool): `True` for white on black, otherwise black on white; default is `False`
- `alignment` (string): Align the `text` horizontally. Use `L` for left-alignment (default), `R` for right-alignment, or `C` for center-alignment

**set_font**(*font='Lat15-TerminusBold24x12'*, *reset_console=True*)

Set the EV3 LCD console font and optionally reset the EV3 LCD console to clear it and turn off the cursor.

Parameters:

- `font` (string): Font name, as found in `/usr/share/consolefonts/`
- `reset_console` (bool): `True` to reset the EV3 LCD console after the font change; default is `True`

**clear_to_eol**(*column=None*, *row=None*)
Clear to the end of line from the `column` and `row` position on the EV3 LCD console. Default to current cursor position.

Parameters:

- `column` (int): LCD column position to move to before clearing
- `row` (int): LCD row position to move to before clearing

**reset_console**()
Clear the EV3 LCD console using ANSI codes, and move the cursor to 1,1

Examples:

```
#!/usr/bin/env micropython
from ev3dev2.console import Console

# create a Console instance, which uses the default font
console = Console()

# reset the console to clear it, home the cursor at 1,1, and then turn off the cursor
console.reset_console()

# display 'Hello World!' at row 5, column 1 in inverse, but reset the EV3 LCD console␣
↪first
console.text_at('Hello World!', column=1, row=5, reset_console=True, inverse=True)
```

```
#!/usr/bin/env micropython
from time import sleep
from ev3dev2.sensor import INPUT_1, INPUT_2, INPUT_3
from ev3dev2.console import Console
from ev3dev2.sensor.lego import GyroSensor, ColorSensor

console = Console()
gyro = GyroSensor(INPUT_1)
gyro.mode = GyroSensor.MODE_GYRO_ANG
color_sensor_left = ColorSensor(INPUT_2)
color_sensor_right = ColorSensor(INPUT_3)

# show the gyro angle and reflected light intensity for both of our color sensors
while True:
    angle = gyro.angle
    left = color_sensor_left.reflected_light_intensity
    right = color_sensor_right.reflected_light_intensity

    # show angle; in inverse color when pointing at 0
    console.text_at("G: %03d" % (angle), column=5, row=1, reset_console=True,␣
↪inverse=(angle == 0))

    # show light intensity values; in inverse when 'dark'
    console.text_at("L: %02d" % (left), column=0, row=3, reset_console=False,␣
↪inverse=(left < 10))
    console.text_at("R: %02d" % (right), column=10, row=3, reset_console=False,␣
↪inverse=(right < 10))

    sleep(0.5)
```

### Console fonts

The *ev3dev2.console.Console* class displays text on the LCD console using ANSI codes in various system console fonts. The system console fonts are located in */usr/share/consolefonts*.

Font filenames consist of the codeset, font face and font size. The codeset specifies the characters supported. The font face determines the look of the font. Each font face is available in multiple sizes.

For Codeset information, see <*https://www.systutorials.com/docs/linux/man/5-console-setup/#lbAP*>.

Note: *Terminus* fonts are "thinner"; *TerminusBold* and *VGA* offer more contrast on the LCD console and are thus more readable; the *TomThumb* font is too small to read!

Depending on the font used, the EV3 LCD console will support various maximum rows and columns, as follows for the *Lat15* fonts. See *utils/console_fonts.py* to discover fonts and their resulting rows/columns. These fonts are listed in larger-to-smaller size order:

| LCD Rows | LCD Columns | Font |
|---|---|---|
| 4 | 11 | Lat15-Terminus32x16.psf.gz |
| 4 | 11 | Lat15-TerminusBold32x16.psf.gz |
| 4 | 11 | Lat15-VGA28x16.psf.gz |
| 4 | 11 | Lat15-VGA32x16.psf.gz |
| 4 | 12 | Lat15-Terminus28x14.psf.gz |
| 4 | 12 | Lat15-TerminusBold28x14.psf.gz |
| 5 | 14 | Lat15-Terminus24x12.psf.gz |
| 5 | 14 | Lat15-TerminusBold24x12.psf.gz |
| 5 | 16 | Lat15-Terminus22x11.psf.gz |
| 5 | 16 | Lat15-TerminusBold22x11.psf.gz |
| 6 | 17 | Lat15-Terminus20x10.psf.gz |
| 6 | 17 | Lat15-TerminusBold20x10.psf.gz |
| 7 | 22 | Lat15-Fixed18.psf.gz |
| 8 | 22 | Lat15-Fixed15.psf.gz |
| 8 | 22 | Lat15-Fixed16.psf.gz |
| 8 | 22 | Lat15-Terminus16.psf.gz |
| 8 | 22 | Lat15-TerminusBold16.psf.gz |
| 8 | 22 | Lat15-TerminusBoldVGA16.psf.gz |
| 8 | 22 | Lat15-VGA16.psf.gz |
| 9 | 22 | Lat15-Fixed13.psf.gz |
| 9 | 22 | Lat15-Fixed14.psf.gz |
| 9 | 22 | Lat15-Terminus14.psf.gz |
| 9 | 22 | Lat15-TerminusBold14.psf.gz |
| 9 | 22 | Lat15-TerminusBoldVGA14.psf.gz |
| 9 | 22 | Lat15-VGA14.psf.gz |
| 10 | 29 | Lat15-Terminus12x6.psf.gz |
| 16 | 22 | Lat15-VGA8.psf.gz |
| 21 | 44 | Lat15-TomThumb4x6.psf.gz |

Example:

```
#!/usr/bin/env micropython
from ev3dev2.console import Console

# create a Console instance, which uses the default font
console = Console()
```

(continues on next page)

(continued from previous page)

```
# change the console font and reset the console to clear it and turn off the cursor
console.set_font('Lat15-TerminusBold16.psf.gz', True)

# compute the middle of the console
mid_col = console.columns // 2
mid_row = console.rows // 2

# display 'Hello World!' in the center of the LCD console
console.text_at('Hello World!', column=mid_col, row=mid_row, alignment="C")
```

### Lego Port

The *LegoPort* class is only needed when manually reconfiguring input/output ports. This is necessary on the BrickPi but not other platforms, such as the EV3. Most users can ignore this page.

**class** ev3dev2.port.**LegoPort**(*address=None*, *name_pattern='*'*, *name_exact=False*, ***kwargs*)

The `lego-port` class provides an interface for working with input and output ports that are compatible with LEGO MINDSTORMS RCX/NXT/EV3, LEGO WeDo and LEGO Power Functions sensors and motors. Supported devices include the LEGO MINDSTORMS EV3 Intelligent Brick, the LEGO WeDo USB hub and various sensor multiplexers from 3rd party manufacturers.

See the following example for using this class to configure sensors: https://github.com/ev3dev/ev3dev-lang-python-demo/blob/stretch/platform/brickpi3-motor-and-sensor.py

Some types of ports may have multiple modes of operation. For example, the input ports on the EV3 brick can communicate with sensors using UART, I2C or analog validate signals - but not all at the same time. Therefore there are multiple modes available to connect to the different types of sensors.

In most cases, ports are able to automatically detect what type of sensor or motor is connected. In some cases though, this must be manually specified using the `mode` and `set_device` attributes. The `mode` attribute affects how the port communicates with the connected device. For example the input ports on the EV3 brick can communicate using UART, I2C or analog voltages, but not all at the same time, so the mode must be set to the one that is appropriate for the connected sensor. The `set_device` attribute is used to specify the exact type of sensor that is connected. Note: the mode must be correctly set before setting the sensor type.

Ports can be found at `/sys/class/lego-port/port<N>` where `<N>` is incremented each time a new port is registered. Note: The number is not related to the actual port at all - use the `address` attribute to find a specific port.

**address**

Returns the name of the port. See individual driver documentation for the name that will be returned.

**driver_name**

Returns the name of the driver that loaded this device. You can find the complete list of drivers in the [list of port drivers].

**modes**

Returns a list of the available modes of the port.

**mode**

Reading returns the currently selected mode. Writing sets the mode. Generally speaking when the mode changes any sensor or motor devices associated with the port will be removed new ones loaded, however this this will depend on the individual driver implementing this class.

**set_device**

For modes that support it, writing the name of a driver will cause a new device to be registered for that

driver and attached to this port. For example, since NXT/Analog sensors cannot be auto-detected, you must use this attribute to load the correct driver. Returns -EOPNOTSUPP if setting a device is not supported.

**status**
In most cases, reading status will return the same value as `mode`. In cases where there is an `auto` mode additional values may be returned, such as `no-device` or `error`. See individual port driver documentation for the full list of possible values.

### Port names

Classes such as *ev3dev2.motor.Motor* and those based on *ev3dev2.sensor.Sensor* accept parameters to specify which port the target device is connected to. This parameter is typically caled `address`.

The following constants are available on all platforms:

### Output

- ev3dev2.motor.OUTPUT_A
- ev3dev2.motor.OUTPUT_B
- ev3dev2.motor.OUTPUT_C
- ev3dev2.motor.OUTPUT_D

### Input

- ev3dev2.sensor.INPUT_1
- ev3dev2.sensor.INPUT_2
- ev3dev2.sensor.INPUT_3
- ev3dev2.sensor.INPUT_4

Additionally, on BrickPi3, the ports of up to four stacked BrickPi's can be referenced as *OUTPUT_E* through *OUTPUT_P* and *INPUT_5* through *INPUT_16*.

### Example

```
from ev3dev2.motor import LargeMotor, OUTPUT_A, OUTPUT_B
from ev3dev2.sensor import INPUT_1
from ev3dev2.sensor.lego import TouchSensor

m = LargeMotor(OUTPUT_A)
s = TouchSensor(INPUT_1)
```

### Wheels

All Wheel class units are in millimeters. The diameter and width for various lego wheels can be found at http://wheels.sariel.pl/

**class** ev3dev2.wheel.**Wheel**(*diameter_mm*, *width_mm*)
A base class for various types of wheels, tires, etc. All units are in mm.

One scenario where one of the child classes below would be used is when the user needs their robot to drive at a specific speed or drive for a specific distance. Both of those calculations require the circumference of the wheel of the robot.

Example:

```
from ev3dev2.wheel import EV3Tire

tire = EV3Tire()

# calculate the number of rotations needed to travel forward 500 mm
rotations_for_500mm = 500 / tire.circumference_mm
```

#### EV3 Rim

**class** ev3dev2.wheel.**EV3Rim**
Bases: *ev3dev2.wheel.Wheel*

part number 56145 comes in set 31313

#### EV3 Tire

**class** ev3dev2.wheel.**EV3Tire**
Bases: *ev3dev2.wheel.Wheel*

part number 44309 comes in set 31313

#### EV3 Education Set Rim

**class** ev3dev2.wheel.**EV3EducationSetRim**
Bases: *ev3dev2.wheel.Wheel*

part number 56908 comes in set 45544

#### EV3 Education Set Tire

**class** ev3dev2.wheel.**EV3EducationSetTire**
Bases: *ev3dev2.wheel.Wheel*

part number 41897 comes in set 45544

### 5.4.2 Other APIs

Each class in ev3dev module inherits from the base *ev3dev2.Device* class.

**class** ev3dev2.**Device**(*class_name*, *name_pattern='*'*, *name_exact=False*, ***kwargs*)
The ev3dev device base class

ev3dev2.**list_device_names**(*class_path*, *name_pattern*, ***kwargs*)
This is a generator function that lists names of all devices matching the provided parameters.

**Parameters:**

**class_path: class path of the device, a subdirectory of /sys/class.** For example, '/sys/class/tacho-motor'.

**name_pattern: pattern that device name should match.** For example, 'sensor*' or 'motor*'. Default value: '*'.

**keyword arguments: used for matching the corresponding device** attributes. For example, address='outA', or driver_name=['lego-ev3-us', 'lego-nxt-us']. When argument value is a list, then a match against any entry of the list is enough.

ev3dev2.**list_devices**(*class_name*, *name_pattern*, ***kwargs*)

This is a generator function that takes same arguments as *Device* class and enumerates all devices present in the system that match the provided arguments.

**Parameters:**

**class_name: class name of the device, a subdirectory of /sys/class.** For example, 'tacho-motor'.

**name_pattern: pattern that device name should match.** For example, 'sensor*' or 'motor*'. Default value: '*'.

**keyword arguments: used for matching the corresponding device** attributes. For example, address='outA', or driver_name=['lego-ev3-us', 'lego-nxt-us']. When argument value is a list, then a match against any entry of the list is enough.

ev3dev2.motor.**list_motors**(*name_pattern='*'*, ***kwargs*)

This is a generator function that enumerates all tacho motors that match the provided arguments.

**Parameters:**

**name_pattern: pattern that device name should match.** For example, 'motor*'. Default value: '*'.

**keyword arguments: used for matching the corresponding device** attributes. For example, driver_name='lego-ev3-l-motor', or address=['outB', 'outC']. When argument value is a list, then a match against any entry of the list is enough.

ev3dev2.sensor.**list_sensors**(*name_pattern='sensor*'*, ***kwargs*)

This is a generator function that enumerates all sensors that match the provided arguments.

**Parameters:**

**name_pattern: pattern that device name should match.** For example, 'sensor*'. Default value: '*'.

**keyword arguments: used for matching the corresponding device** attributes. For example, driver_name='lego-ev3-touch', or address=['in1', 'in3']. When argument value is a list, then a match against any entry of the list is enough.

## 5.5 RPyC on ev3dev

RPyC_ (pronounced as are-pie-see) can be used to: * run a python program on an ev3dev device that controls another ev3dev device. This is more commonly known as daisy chaining. * run a python program on your laptop that controls an ev3dev device. This can be useful if your robot requires CPU intensive code that would be slow to run on the EV3. A good example of this is a Rubik's cube solver, calculating the solution to solve a Rubik's cube can be slow on an EV3.

For both of these scenarios you can use RPyC to control multiple remote ev3dev devices.

### 5.5.1 Networking

You will need IP connectivity between the device where your python code runs (laptop, an ev3dev device, etc) and the remote ev3dev devices. Some common scenarios might be: * Multiple EV3s on the same WiFi network * A laptop and an EV3 on the same WiFi network * A bluetooth connection between two EV3s

The ev3dev networking documentation should get you up and running in terms of networking connectivity.

### 5.5.2 Install

1. RPyC is installed on ev3dev but we need to create a service that launches `rpyc_classic.py` at bootup. SSH to your remote ev3dev devices and cut-n-paste the following commands at the bash prompt.

```
echo "[Unit]
Description=RPyC Classic Service
After=multi-user.target

[Service]
Type=simple
ExecStart=/usr/bin/rpyc_classic.py

[Install]
WantedBy=multi-user.target" > rpyc-classic.service

sudo cp rpyc-classic.service /lib/systemd/system/
sudo systemctl daemon-reload
sudo systemctl enable rpyc-classic.service
sudo systemctl start rpyc-classic.service
```

2. If you will be using an ev3dev device to control another ev3dev device you can skip this step. If you will be using your desktop PC to control an ev3dev device you must install RPyC on your desktop PC. How you install RPyC depends on your operating system. For Linux you should be able to do:

```
sudo apt-get install python3-rpyc
```

   For Windows there is a win32 installer on the project's sourceforge page. Also, have a look at the Download and Install page on their site.

### 5.5.3 Example

We will run code on our laptop to control the remote ev3dev device with IP address X.X.X.X. The goal is to have the LargeMotor connected to `OUTPUT_A` run when the TouchSensor on `INPUT_1` is pressed.

```
import rpyc

# Create a RPyC connection to the remote ev3dev device.
# Use the hostname or IP address of the ev3dev device.
# If this fails, verify your IP connectivty via ``ping X.X.X.X``
conn = rpyc.classic.connect('X.X.X.X')

# import ev3dev2 on the remote ev3dev device
ev3dev2_motor = conn.modules['ev3dev2.motor']
ev3dev2_sensor = conn.modules['ev3dev2.sensor']
ev3dev2_sensor_lego = conn.modules['ev3dev2.sensor.lego']
```

(continues on next page)

(continued from previous page)

```
# Use the LargeMotor and TouchSensor on the remote ev3dev device
motor = ev3dev2_motor.LargeMotor(ev3dev2_motor.OUTPUT_A)
ts = ev3dev2_sensor_lego.TouchSensor(ev3dev2_sensor.INPUT_1)

# If the TouchSensor is pressed, run the motor
while True:
    ts.wait_for_pressed()
    motor.run_forever(speed_sp=200)

    ts.wait_for_released()
    motor.stop()
```

### 5.5.4 Pros

- RPyC is lightweight and only requires an IP connection (no ssh required).
- Some robots may need much more computational power than an EV3 can give you. A notable example is the Rubik's cube solver.

### 5.5.5 Cons

- Latency will be introduced by the network connection. This may be a show stopper for robots where reaction speed is essential.
- RPyC is only supported by python, it is *NOT* supported by micropython

### 5.5.6 References

- RPyC
- sourceforge page
- Download and Install
- connect with SSH

## 5.6 Frequently-Asked Questions

**Q: Why does my Python program exit quickly or immediately throw an error?** A: This may occur if your file includes Windows-style line endings (CRLF–carriage-return line-feed), which are often inserted by editors on Windows. To resolve this issue, open an SSH session and run the following command, replacing `<file>` with the name of the Python file you're using:

```
sed -i 's/\r//g' <file>
```

This will fix it for the copy of the file on the brick, but if you plan to edit it again from Windows, you should configure your editor to use Unix-style line endings (LF–line-feed). For PyCharm, you can find a guide on doing this here. Most other editors have similar options; there may be an option for it in the status bar at the bottom of the window or in the menu bar at the top.

**Q: Where can I learn more about the ev3dev operating system?** A: ev3dev.org is a great resource for finding guides and tutorials on using ev3dev, straight from the maintainers.

**Q: How can I request support on the ev3dev2 Python library?** A: If you are having trouble using this library, please open an issue at our Issues tracker so that we can help you. When opening an issue, make sure to include as much information as possible about what you are trying to do and what you have tried. The issue template is in place to guide you through this process.

**Q: How can I upgrade the library on my EV3?** A: You can upgrade this library from an Internet-connected EV3 with an SSH shell as follows. Make sure to type the password (the default is `maker`) when prompted.

```
sudo apt-get update
sudo apt-get install --only-upgrade python3-ev3dev2 micropython-ev3dev2
```

**Q: Are there other useful Python modules to use on the EV3?** A: The Python language has a package repository where you can find libraries that others have written, including the latest version of this package.

**Q: What compatibility issues are there with the different versions of Python?** A: Some versions of the ev3dev distribution come with Python 2.x, Python 3.x, and micropython installed, but this library is compatible only with Python 3 and micropython.

# Index

## D



## E



## F

## N



## O

## P



## R



## S

## T



## U



## V



## W



## X



## Y