

# Pybricks

## Pybricks Modules and Examples

*Version v3.0.0-rc.1*

Jul 28, 2021

---

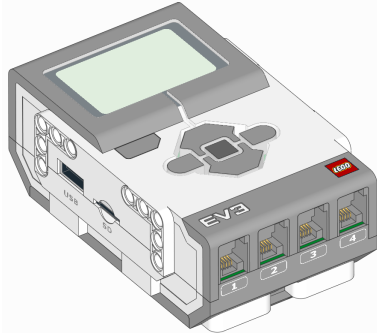
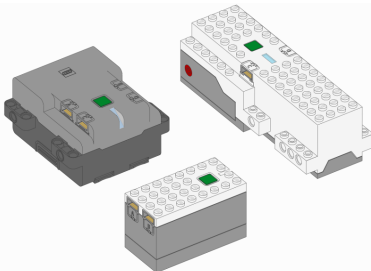
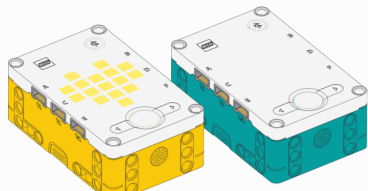
## TABLE OF CONTENTS

<b>1</b>	<b>hubs – Programmable Hubs</b>	<b>2</b>
<b>2</b>	<b>pupdevices – Powered Up Devices</b>	<b>21</b>
<b>3</b>	<b>ev3devices – EV3 Devices</b>	<b>56</b>
<b>4</b>	<b>nxtdevices – NXT Devices</b>	<b>63</b>
<b>5</b>	<b>iodevices – Generic I/O Devices</b>	<b>69</b>
<b>6</b>	<b>parameters – Parameters and Constants</b>	<b>81</b>
<b>7</b>	<b>tools – General purpose tools</b>	<b>86</b>
<b>8</b>	<b>robotics – Robotics</b>	<b>90</b>
<b>9</b>	<b>media – Sounds and Images</b>	<b>93</b>
<b>10</b>	<b>messaging – Messaging</b>	<b>116</b>
<b>11</b>	<b>Signals and Units</b>	<b>122</b>
<b>12</b>	<b>More about Motors</b>	<b>127</b>
	<b>Python Module Index</b>	<b>130</b>
	<b>Index</b>	<b>131</b>

This documentation has everything you need to install Pybricks and run your first scripts.

### Step 1: Install Pybricks

Pybricks scripts are the same across all hubs. The only difference is in how you install Pybricks and how you run scripts. To get started, click one of the platforms below.

MINDSTORMS EV3	Powered Up (Technic/City/BOOST)	SPIKE / MINDSTORMS Inventor
		

### Step 2: Start coding!

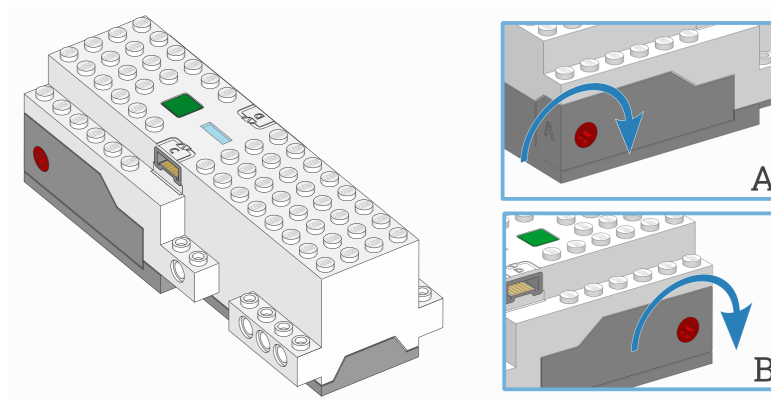
After you've followed the installation steps for your hub, check out the Pybricks modules in the left hand menu to see what you can do.

### Step 3: Share what you made (or ask for help!)

Got questions or issues? Please share your findings on our [support page](#) so we can make Pybricks even better. *Thank you!*

## HUBS – PROGRAMMABLE HUBS

### 1.1 Move Hub



**class MoveHub**  
LEGO® BOOST Move Hub.

#### Using the hub status light

`light.on(color)`  
Turns on the light at the specified color.

**Parameters** `color` (`Color`) – Color of the light.

`light.off()`  
Turns off the light.

`light.blink(color, durations)`  
Blinks the light at a given color by turning it on and off for given durations.

The light keeps blinking indefinitely while the rest of your program keeps running.

This method provides a simple way to make basic but useful patterns. For more generic and multi-color patterns, use `animate()` instead.

#### Parameters

- **color** (`Color`) – Color of the light.
- **durations** (`list`) – List of (*time: ms*) values of the form [`on_1`, `off_1`, `on_2`, `off_2`, ...].

`light.animate(colors, interval)`

Animates the light with a list of colors. The next color in the list is shown after the given interval.

The animation runs in the background while the rest of your program keeps running. When the animation completes, it repeats.

#### Parameters

- **colors** (*list*) – List of *Color* values.
- **interval** (*time: ms*) – Time between color updates.

### Using the battery

`battery.voltage()`

Gets the voltage of the battery.

**Returns** Battery voltage.

**Return type** *voltage: mV*

`battery.current()`

Gets the current supplied by the battery.

**Returns** Battery current.

**Return type** *current: mA*

## 1.1.1 Status light examples

### Turning the light on and off

```
from pybricks.hubs import MoveHub
from pybricks.parameters import Color
from pybricks.tools import wait

# Initialize the hub.
hub = MoveHub()

# Turn the light on and off 5 times.
for i in range(5):

    hub.light.on(Color.RED)
    wait(1000)

    hub.light.off()
    wait(500)
```

## Making the light blink

```
from pybricks.hubs import MoveHub
from pybricks.parameters import Color
from pybricks.tools import wait

# Initialize the hub
hub = MoveHub()

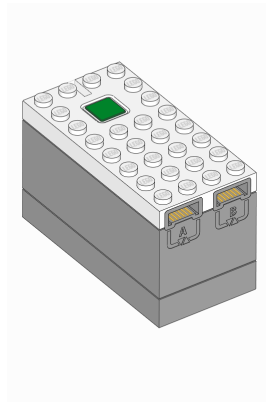
# Keep blinking red on and off.
hub.light.blink(Color.RED, [500, 500])

wait(10000)

# Keep blinking green slowly and then quickly.
hub.light.blink(Color.GREEN, [500, 500, 50, 900])

wait(10000)
```

## 1.2 City Hub



**class CityHub**  
LEGO® City Hub.

### Using the hub status light

`light.on(color)`

Turns on the light at the specified color.

**Parameters** `color` (`Color`) – Color of the light.

`light.off()`

Turns off the light.

`light.blink(color, durations)`

Blinks the light at a given color by turning it on and off for given durations.

The light keeps blinking indefinitely while the rest of your program keeps running.

This method provides a simple way to make basic but useful patterns. For more generic and multi-color patterns, use `animate()` instead.

### Parameters

- **color** (*Color*) – Color of the light.
- **durations** (*list*) – List of (*time: ms*) values of the form [on\_1, off\_1, on\_2, off\_2, ...].

`light.animate(colors, interval)`

Animates the light with a list of colors. The next color in the list is shown after the given interval.

The animation runs in the background while the rest of your program keeps running. When the animation completes, it repeats.

### Parameters

- **colors** (*list*) – List of *Color* values.
- **interval** (*time: ms*) – Time between color updates.

## Using the battery

`battery.voltage()`

Gets the voltage of the battery.

**Returns** Battery voltage.

**Return type** *voltage: mV*

`battery.current()`

Gets the current supplied by the battery.

**Returns** Battery current.

**Return type** *current: mA*

## 1.2.1 Status light examples

### Turning the light on and off

```
from pybricks.hubs import CityHub
from pybricks.parameters import Color
from pybricks.tools import wait

# Initialize the hub.
hub = CityHub()

# Turn the light on and off 5 times.
for i in range(5):

    hub.light.on(Color.RED)
    wait(1000)

    hub.light.off()
    wait(500)
```

## Changing brightness and using custom colors

```
from pybricks.hubs import CityHub
from pybricks.parameters import Color
from pybricks.tools import wait

# Initialize the hub.
hub = CityHub()

# Show the color at 30% brightness.
hub.light.on(Color.RED * 0.3)

wait(2000)

# Use your own custom color.
hub.light.on(Color(h=30, s=100, v=50))

wait(2000)

# Go through all the colors.
for hue in range(360):
    hub.light.on(Color(hue))
    wait(10)
```

## Making the light blink

```
from pybricks.hubs import CityHub
from pybricks.parameters import Color
from pybricks.tools import wait

# Initialize the hub
hub = CityHub()

# Keep blinking red on and off.
hub.light.blink(Color.RED, [500, 500])

wait(10000)

# Keep blinking green slowly and then quickly.
hub.light.blink(Color.GREEN, [500, 500, 50, 900])

wait(10000)
```

## Creating light animations

```
from pybricks.hubs import CityHub
from pybricks.parameters import Color
from pybricks.tools import wait
from math import sin, pi

# Initialize the hub.
hub = CityHub()

# Make an animation with multiple colors.
```

(continues on next page)



(continued from previous page)

```

hub.light.animate([Color.RED, Color.GREEN, Color.NONE], interval=500)

wait(10000)

# Make the color RED grow faint and bright using a sine pattern.
hub.light.animate(
    [Color.RED * (0.5 * sin(i / 15 * pi) + 0.5) for i in range(30)], 40)

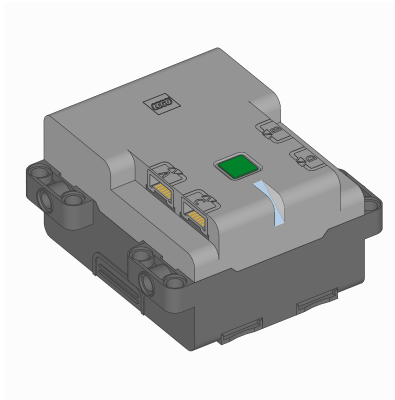
wait(10000)

# Cycle through a rainbow of colors.
hub.light.animate([Color(h=i*8) for i in range(45)], interval=40)

wait(10000)

```

## 1.3 Technic Hub



**class TechnicHub**

### Using the hub status light

`light.on(color)`

Turns on the light at the specified color.

**Parameters** `color` (`Color`) – Color of the light.

`light.off()`

Turns off the light.

`light.blink(color, durations)`

Blinks the light at a given color by turning it on and off for given durations.

The light keeps blinking indefinitely while the rest of your program keeps running.

This method provides a simple way to make basic but useful patterns. For more generic and multi-color patterns, use `animate()` instead.

#### Parameters

- `color` (`Color`) – Color of the light.

- **durations** (*list*) – List of (*time: ms*) values of the form [on\_1, off\_1, on\_2, off\_2, ...].

`light.animate(colors, interval)`

Animates the light with a list of colors. The next color in the list is shown after the given interval.

The animation runs in the background while the rest of your program keeps running. When the animation completes, it repeats.

#### Parameters

- **colors** (*list*) – List of *Color* values.
- **interval** (*time: ms*) – Time between color updates.

### Using the battery

`battery.voltage()`

Gets the voltage of the battery.

**Returns** Battery voltage.

**Return type** *voltage: mV*

`battery.current()`

Gets the current supplied by the battery.

**Returns** Battery current.

**Return type** *current: mA*

## 1.3.1 Status light examples

### Turning the light on and off

```
from pybricks.hubs import TechnicHub
from pybricks.parameters import Color
from pybricks.tools import wait

# Initialize the hub.
hub = TechnicHub()

# Turn the light on and off 5 times.
for i in range(5):

    hub.light.on(Color.RED)
    wait(1000)

    hub.light.off()
    wait(500)
```

## Changing brightness and using custom colors

```
from pybricks.hubs import TechnicHub
from pybricks.parameters import Color
from pybricks.tools import wait

# Initialize the hub.
hub = TechnicHub()

# Show the color at 30% brightness.
hub.light.on(Color.RED * 0.3)

wait(2000)

# Use your own custom color.
hub.light.on(Color(h=30, s=100, v=50))

wait(2000)

# Go through all the colors.
for hue in range(360):
    hub.light.on(Color(hue))
    wait(10)
```

## Making the light blink

```
from pybricks.hubs import TechnicHub
from pybricks.parameters import Color
from pybricks.tools import wait

# Initialize the hub
hub = TechnicHub()

# Keep blinking red on and off.
hub.light.blink(Color.RED, [500, 500])

wait(10000)

# Keep blinking green slowly and then quickly.
hub.light.blink(Color.GREEN, [500, 500, 50, 900])

wait(10000)
```

## Creating light animations

```
from pybricks.hubs import TechnicHub
from pybricks.parameters import Color
from pybricks.tools import wait
from math import sin, pi

# Initialize the hub.
hub = TechnicHub()

# Make an animation with multiple colors.
```

(continues on next page)

(continued from previous page)

```

hub.light.animate([Color.RED, Color.GREEN, Color.NONE], interval=500)

wait(10000)

# Make the color RED grow faint and bright using a sine pattern.
hub.light.animate(
    [Color.RED * (0.5 * sin(i / 15 * pi) + 0.5) for i in range(30)], 40)

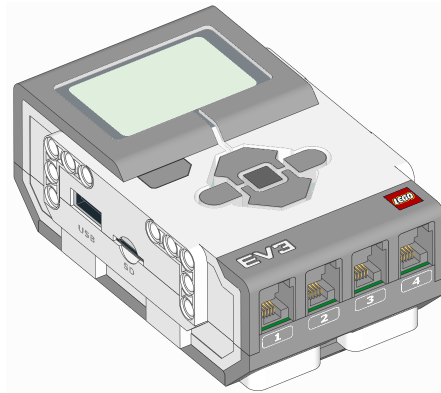
wait(10000)

# Cycle through a rainbow of colors.
hub.light.animate([Color(h=i*8) for i in range(45)], interval=40)

wait(10000)

```

## 1.4 MINDSTORMS EV3 Brick



```

class EV3Brick
    LEGO® MINDSTORMS® EV3 Brick.

```

### Using the buttons

```

buttons.pressed()
    Checks which buttons are currently pressed.

    Returns Tuple of pressed buttons.

    Return type Tuple of Button

```

## Using the brick status light

`light.on(color)`

Turns on the light at the specified color.

**Parameters** `color` (`Color`) – Color of the light.

`light.off()`

Turns off the light.

## Using the speaker

`speaker.beep(frequency=500, duration=100)`

Play a beep/tone.

### Parameters

- **frequency** (*frequency: Hz*) – Frequency of the beep. Frequencies below 100 are treated as 100.
- **duration** (*time: ms*) – Duration of the beep. If the duration is less than 0, then the method returns immediately and the frequency play continues to play indefinitely.

`speaker.play_notes(notes, tempo=120)`

Plays a sequence of musical notes. For example: `['C4/4', 'C4/4', 'G4/4', 'G4/4']`.

Each note is a string with the following format:

- The first character is the name of the note, A to G or R for a rest.
- Note names can also include an accidental # (sharp) or b (flat). B#/Cb and E#/Fb are not allowed.
- The note name is followed by the octave number 2 to 8. For example C4 is middle C. The octave changes to the next number at the note C, for example, B3 is the note below middle C (C4).
- The octave is followed by / and a number that indicates the size of the note. For example /4 is a quarter note, /8 is an eighth note and so on.
- This can optionally followed by a . to make a dotted note. Dotted notes are 1-1/2 times as long as notes without a dot.
- The note can optionally end with a \_ which is a tie or a slur. This causes there to be no pause between this note and the next note.

### Parameters

- **notes** (*iter*) – A sequence of notes to be played.
- **tempo** (*int*) – Beats per minute. A quarter note is one beat.

`speaker.play_file(file)`

Plays a sound file.

**Parameters** `file` (*str*) – Path to the sound file, including the file extension.

`speaker.say(text)`

Says a given text string.

You can configure the language and voice of the text using `set_speech_options()`.

**Parameters** `text` (*str*) – What to say.

`speaker.set_speech_options (language=None, voice=None, speed=None, pitch=None)`

Configures speech settings used by the `say()` method.

Any option that is set to `None` will not be changed. If an option is set to an invalid value `say()` will use the default value instead.

### Parameters

- **language** (*str*) – Language of the text. For example, you can choose 'en' (English) or 'de' (German).<sup>1</sup>

---

<sup>1</sup> You can choose the following languages:

- 'af': Afrikaans
- 'an': Aragonese
- 'bg': Bulgarian
- 'bs': Bosnian
- 'ca': Catalan
- 'cs': Czech
- 'cy': Welsh
- 'da': Danish
- 'de': German
- 'el': Greek
- 'en': English (default)
- 'en-gb': English (United Kingdom)
- 'en-sc': English (Scotland)
- 'en-uk-north': English (United Kingdom, Northern)
- 'en-uk-rp': English (United Kingdom, Received Pronunciation)
- 'en-uk-wmids': English (United Kingdom, West Midlands)
- 'en-us': English (United States)
- 'en-wi': English (West Indies)
- 'eo': Esperanto
- 'es': Spanish
- 'es-la': Spanish (Latin America)
- 'et': Estonian
- 'fa': Persian
- 'fa-pin': Persian
- 'fi': Finnish
- 'fr-be': French (Belgium)
- 'fr-fr': French (France)
- 'ga': Irish
- 'grc': Greek
- 'hi': Hindi
- 'hr': Croatian
- 'hu': Hungarian
- 'hy': Armenian
- 'hy-west': Armenian (Western)

- **voice** (*str*) – The voice to use. For example, you can choose 'f1' (female voice variant 1) or 'm3' (male voice variant 3).<sup>?</sup>
- **speed** (*int*) – Number of words per minute.
- **pitch** (*int*) – Pitch (0 to 99). Higher numbers make the voice higher pitched and lower numbers make the voice lower pitched.

---

```

- 'id': Indonesian
- 'is': Icelandic
- 'it': Italian
- 'jbo': Lojban
- 'ka': Georgian
- 'kn': Kannada
- 'ku': Kurdish
- 'la': Latin
- 'lfn': Lingua Franca Nova
- 'lt': Lithuanian
- 'lv': Latvian
- 'mk': Macedonian
- 'ml': Malayalam
- 'ms': Malay
- 'ne': Nepali
- 'nl': Dutch
- 'no': Norwegian
- 'pa': Punjabi
- 'pl': Polish
- 'pt-br': Portuguese (Brazil)
- 'pt-pt': Portuguese (Portugal)
- 'ro': Romanian
- 'ru': Russian
- 'sk': Slovak
- 'sq': Albanian
- 'sr': Serbian
- 'sv': Swedish
- 'sw': Swahili
- 'ta': Tamil
- 'tr': Turkish
- 'vi': Vietnamese
- 'vi-hue': Vietnamese (Hue)
- 'vi-sgn': Vietnamese (Saigon)
- 'zh': Mandarin Chinese
- 'zh-yue': Cantonese Chinese
  You can choose the following voices:
- 'f1': female variant 1
- 'f2': female variant 2

```

---

```
speaker.set_volume(volume, which='_all_')
```

Sets the speaker volume.

#### Parameters

- **volume** (*percentage: %*) – Volume of the speaker.
- **which** (*str*) – Which volume to set. 'Beep' sets the volume for `beep()` and `play_notes()`. 'PCM' sets the volume for `play_file()` and `say()`. '\_all\_' sets both at the same time.

### Using the screen

```
screen.clear()
```

Clears the screen. All pixels on the screen will be set to `Color.WHITE`.

```
screen.draw_text(x, y, text, text_color=Color(h=0, s=0, v=10), background_color=None)
```

Draws text on the screen.

The most recent font set using `set_font()` will be used or `Font.DEFAULT` if no font has been set yet.

#### Parameters

- **x** (*int*) – The x-axis value where the left side of the text will start.
- **y** (*int*) – The y-axis value where the top of the text will start.
- **text** (*str*) – The text to draw.
- **text\_color** (*Color*) – The color used for drawing the text.
- **background\_color** (*Color*) – The color used to fill the rectangle behind the text or `None` for transparent background.

```
screen.print(*args, sep=' ', end='\n')
```

Prints a line of text on the screen.

This method works like the builtin `print()` function, but it writes on the screen instead.

You can set the font using `set_font()`. If no font has been set, `Font.DEFAULT` will be used. The text is always printed using black text with a white background.

- 
- 'f3': female variant 3
  - 'f4': female variant 4
  - 'f5': female variant 5
  - 'm1': male variant 1
  - 'm2': male variant 2
  - 'm3': male variant 3
  - 'm4': male variant 4
  - 'm5': male variant 5
  - 'm6': male variant 6
  - 'm7': male variant 7
  - 'croak': croak
  - 'whisper': whisper
  - 'whisperf': female whisper



Unlike the builtin `print()`, the text does not wrap if it is too wide to fit on the screen. It just gets cut off. But if the text would go off of the bottom of the screen, the entire image is scrolled up and the text is printed in the new blank area at the bottom of the screen.

#### Parameters

- `*` (*object*) – Zero or more objects to print.
- `sep` (*str*) – Separator that will be placed between each object that is printed.
- `end` (*str*) – End of line that will be printed after the last object.

`screen.set_font(font)`

Sets the font used for writing on the screen.

The font is used for both `draw_text()` and `print()`.

**Parameters** `font` (*Font*) – The font to use.

`screen.load_image(source)`

Clears this image, then draws the `source` image centered in the screen.

**Parameters** `source` (*Image or str*) – The source *Image*. If the argument is a string, then the `source` image is loaded from file.

`screen.draw_image(x, y, source, transparent=None)`

Draws the `source` image on the screen.

#### Parameters

- `x` (*int*) – The x-axis value where the left side of the image will start.
- `y` (*int*) – The y-axis value where the top of the image will start.
- `source` (*Image or str*) – The source *Image*. If the argument is a string, then the `source` image is loaded from file.
- `transparent` (*Color*) – The color of image to treat as transparent or `None` for no transparency.

`screen.draw_pixel(x, y, color=Color(h=0, s=0, v=10))`

Draws a single pixel on the screen.

#### Parameters

- `x` (*int*) – The x coordinate of the pixel.
- `y` (*int*) – The y coordinate of the pixel.
- `color` (*Color*) – The color of the pixel.

`screen.draw_line(x1, y1, x2, y2, width=1, color=Color(h=0, s=0, v=10))`

Draws a line on the screen.

#### Parameters

- `x1` (*int*) – The x coordinate of the starting point of the line.
- `y1` (*int*) – The y coordinate of the starting point of the line.
- `x2` (*int*) – The x coordinate of the ending point of the line.
- `y2` (*int*) – The y coordinate of the ending point of the line.
- `width` (*int*) – The width of the line in pixels.
- `color` (*Color*) – The color of the line.

```
screen.draw_box(x1, y1, x2, y2, r=0, fill=False, color=Color(h=0, s=0, v=10))
```

Draws a box on the screen.

#### Parameters

- **x1** (*int*) – The x coordinate of the left side of the box.
- **y1** (*int*) – The y coordinate of the top of the box.
- **x2** (*int*) – The x coordinate of the right side of the box.
- **y2** (*int*) – The y coordinate of the bottom of the box.
- **r** (*int*) – The radius of the corners of the box.
- **fill** (*bool*) – If `True`, the box will be filled with `color`, otherwise only the outline of the box will be drawn.
- **color** (*Color*) – The color of the box.

```
screen.draw_circle(x, y, r, fill=False, color=Color(h=0, s=0, v=10))
```

Draws a circle on the screen.

#### Parameters

- **x** (*int*) – The x coordinate of the center of the circle.
- **y** (*int*) – The y coordinate of the center of the circle.
- **r** (*int*) – The radius of the circle.
- **fill** (*bool*) – If `True`, the circle will be filled with `color`, otherwise only the circumference will be drawn.
- **color** (*Color*) – The color of the circle.

```
screen.width
```

Gets the width of the screen in pixels.

```
screen.height
```

Gets the height of the screen in pixels.

```
screen.save(filename)
```

Saves the screen as a `.png` file.

**Parameters** **filename** (*str*) – The path to the file to be saved.

#### Raises

- **TypeError** – filename is not a string.
- **OSError** – There was a problem saving the file.

## Using the battery

```
battery.voltage()
```

Gets the voltage of the battery.

**Returns** Battery voltage.

**Return type** *voltage*: *mV*

```
battery.current()
```

Gets the current supplied by the battery.

**Returns** Battery current.

Return type *current: mA*

## 1.4.1 Status light examples

### Turn the light on and change the color

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

from pybricks.hubs import EV3Brick
from pybricks.tools import wait
from pybricks.parameters import Color

# Initialize the EV3
ev3 = EV3Brick()

# Turn on a red light
ev3.light.on(Color.RED)

# Wait
wait(1000)

# Turn the light off
ev3.light.off()
```

## 1.4.2 Screen examples

### Show an image on the screen

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

from pybricks.hubs import EV3Brick
from pybricks.tools import wait
from pybricks.media.ev3dev import Image, ImageFile

# It takes some time to load images from the SD card, so it is best to load
# them once at the beginning of a program like this:
ev3_img = Image(ImageFile.EV3_ICON)

# Initialize the EV3
ev3 = EV3Brick()

# Show an image
ev3.screen.load_image(ev3_img)

# Wait some time to look at the image
wait(5000)
```

## Drawing shapes on the screen

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

from pybricks.hubs import EV3Brick
from pybricks.tools import wait

# Initialize the EV3
ev3 = EV3Brick()

# Draw a rectangle
ev3.screen.draw_box(10, 10, 40, 40)

# Draw a solid rectangle
ev3.screen.draw_box(20, 20, 30, 30, fill=True)

# Draw a rectangle with rounded corners
ev3.screen.draw_box(50, 10, 80, 40, 5)

# Draw a circle
ev3.screen.draw_circle(25, 75, 20)

# Draw a triangle using lines
x1, y1 = 65, 55
x2, y2 = 50, 95
x3, y3 = 80, 95
ev3.screen.draw_line(x1, y1, x2, y2)
ev3.screen.draw_line(x2, y2, x3, y3)
ev3.screen.draw_line(x3, y3, x1, y1)

# Wait some time to look at the shapes
wait(5000)
```

## Using different fonts

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

from pybricks.hubs import EV3Brick
from pybricks.tools import wait
from pybricks.media.ev3dev import Font

# It takes some time for fonts to load from file, so it is best to only
# load them once at the beginning of the program like this:
tiny_font = Font(size=6)
big_font = Font(size=24, bold=True)
chinese_font = Font(size=24, lang='zh-cn')

# Initialize the EV3
ev3 = EV3Brick()

# Say hello
ev3.screen.print('Hello!')
```

(continues on next page)

(continued from previous page)

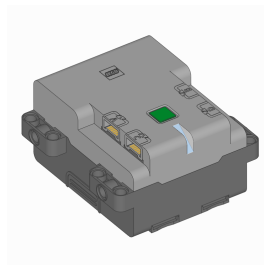
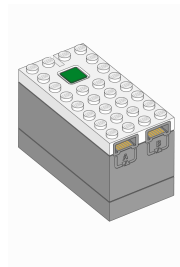
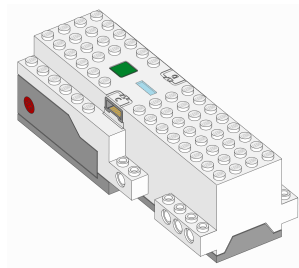
```
# Say tiny hello
ev3.screen.set_font(tiny_font)
ev3.screen.print('hello')

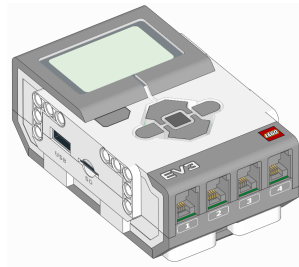
# Say big hello
ev3.screen.set_font(big_font)
ev3.screen.print('HELLO')

# Say Chinese hello
ev3.screen.set_font(chinese_font)
ev3.screen.print('你好')

# Wait some time to look at the screen
wait(5000)
```

## Available languages and voices for speech





## PUPDEVICES – POWERED UP DEVICES

LEGO® Powered Up motor, sensors, and lights.

### 2.1 Motors without Rotation Sensors

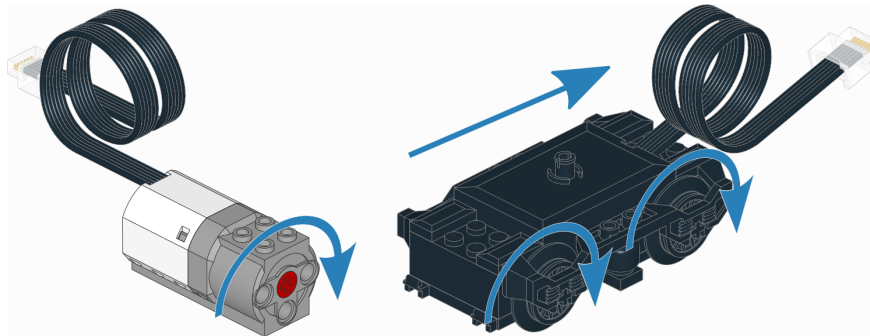


Figure 2.1: Powered Up motors without rotation sensors. The arrows indicate the default positive direction.

```
class DCMotor (port, positive_direction=Direction.CLOCKWISE)
```

Generic class to control simple motors without rotation sensors, such as train motors.

#### Parameters

- **port** (`Port`) – Port to which the motor is connected.
- **positive\_direction** (`Direction`) – Which direction the motor should turn when you give a positive duty cycle value.

#### **dc** (duty)

Rotates the motor at a given duty cycle (also known as “power”).

**Parameters** **duty** (*percentage: %*) – The duty cycle (-100.0 to 100).

#### **stop** ()

Stops the motor and lets it spin freely.

The motor gradually stops due to friction.

#### **brake** ()

Passively brakes the motor.

The motor stops due to friction, plus the voltage that is generated while the motor is still moving.

## 2.1.1 Examples

### Making a train drive forever

```
from pybricks.pupdevices import DCMotor
from pybricks.parameters import Port
from pybricks.tools import wait

# Initialize the motor.
train_motor = DCMotor(Port.A)

# Choose the "power" level for your train. Negative means reverse.
train_motor.dc(50)

# Keep doing nothing. The train just keeps going.
while True:
    wait(1000)
```

### Making the motor move back and forth

```
from pybricks.pupdevices import DCMotor
from pybricks.parameters import Port
from pybricks.tools import wait

# Initialize a motor without rotation sensors on port A.
example_motor = DCMotor(Port.A)

# Make the motor go clockwise (forward) at 70% duty cycle ("70% power").
example_motor.dc(70)

# Wait for three seconds.
wait(3000)

# Make the motor go counterclockwise (backward) at 70% duty cycle.
example_motor.dc(-70)

# Wait for three seconds.
wait(3000)
```

### Changing the positive direction

```
from pybricks.pupdevices import DCMotor
from pybricks.parameters import Port, Direction
from pybricks.tools import wait

# Initialize a motor without rotation sensors on port A,
# with the positive direction as counterclockwise.
example_motor = DCMotor(Port.A, Direction.COUNTERCLOCKWISE)

# When we choose a positive duty cycle, the motor now goes counterclockwise.
example_motor.dc(70)

# This is useful when your (train) motor is mounted in reverse or upside down.
```

(continues on next page)



(continued from previous page)

```
# By changing the positive direction, your script will be easier to read,  
# because a positive value now makes your train/robot go forward.  
  
# Wait for three seconds.  
wait(3000)
```

## Starting and stopping

```
from pybricks.pupdevices import DCMotor  
from pybricks.parameters import Port  
from pybricks.tools import wait  
  
# Initialize a motor without rotation sensors on port A.  
example_motor = DCMotor(Port.A)  
  
# Start and stop 10 times.  
for count in range(10):  
    print("Counter:", count)  
  
    example_motor.dc(70)  
    wait(1000)  
  
    example_motor.stop()  
    wait(1000)
```

## 2.2 Motors with Rotation Sensors

**class Motor** (*port*, *positive\_direction*=*Direction.CLOCKWISE*, *gears*=*None*)

Generic class to control motors with built-in rotation sensors.

### Parameters

- **port** (*Port*) – Port to which the motor is connected.
- **positive\_direction** (*Direction*) – Which direction the motor should turn when you give a positive speed value or angle.
- **gears** (*list*) – List of gears linked to the motor.

For example: `[12, 36]` represents a gear train with a 12-tooth and a 36-tooth gear. Use a list of lists for multiple gear trains, such as `[[12, 36], [20, 16, 40]]`.

When you specify a gear train, all motor commands and settings are automatically adjusted to account for the resulting gear ratio. The motor direction remains unchanged by this.

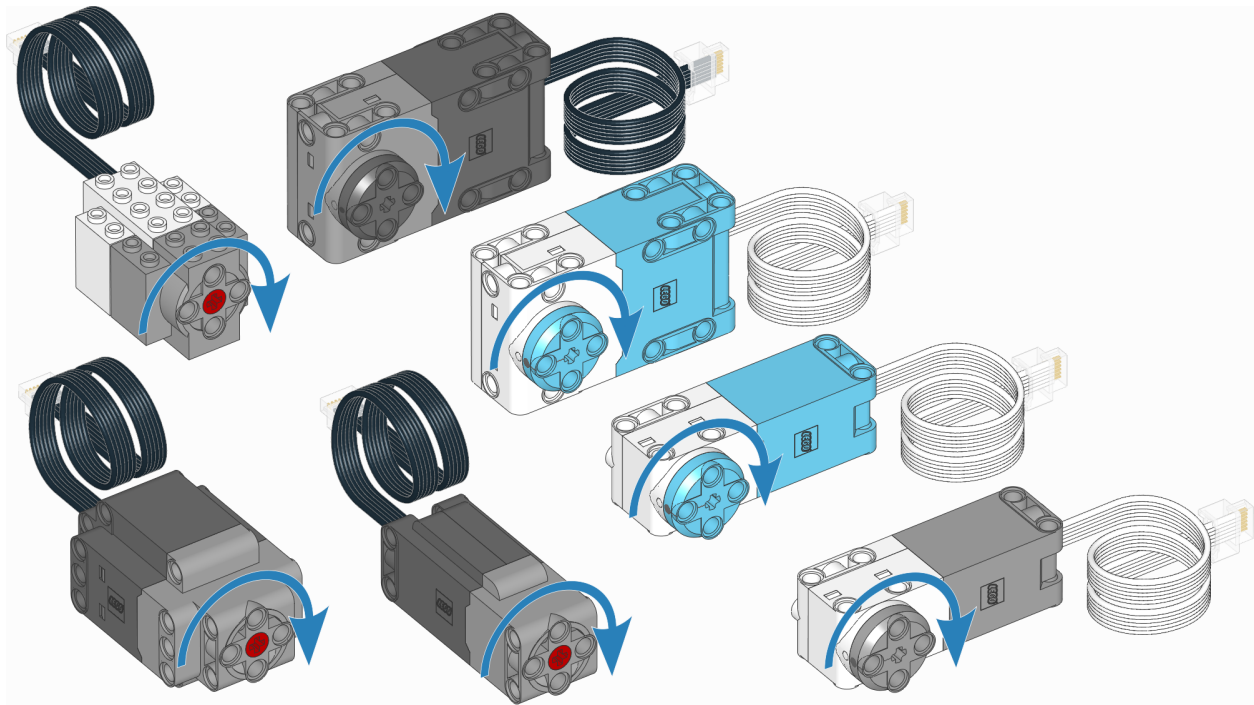


Figure 2.2: Powered Up motors with rotation sensors. The arrows indicate the default positive direction. See the [hubs](#) module for default directions of built-in motors.

## Measuring

### `speed()`

Gets the speed of the motor.

**Returns** Motor speed.

**Return type** *rotational speed: deg/s*

### `angle()`

Gets the rotation angle of the motor.

**Returns** Motor angle.

**Return type** *angle: deg*

### `reset_angle(angle=None)`

Sets the accumulated rotation angle of the motor to a desired value.

If you don't specify an angle, the absolute angle will be used if your motor supports it.

**Parameters** `angle` (*angle: deg*) – Value to which the angle should be reset.

## Stopping

### **stop()**

Stops the motor and lets it spin freely.

The motor gradually stops due to friction.

### **brake()**

Passively brakes the motor.

The motor stops due to friction, plus the voltage that is generated while the motor is still moving.

### **hold()**

Stops the motor and actively holds it at its current angle.

## Action

### **run(speed)**

Runs the motor at a constant speed.

The motor accelerates to the given speed and keeps running at this speed until you give a new command.

**Parameters** **speed** (*rotational speed: deg/s*) – Speed of the motor.

### **run\_time(speed, time, then=Stop.HOLD, wait=True)**

Runs the motor at a constant speed for a given amount of time.

The motor accelerates to the given speed, keeps running at this speed, and then decelerates. The total maneuver lasts for exactly the given amount of `time`.

#### Parameters

- **speed** (*rotational speed: deg/s*) – Speed of the motor.
- **time** (*time: ms*) – Duration of the maneuver.
- **then** (`Stop`) – What to do after coming to a standstill.
- **wait** (`bool`) – Wait for the maneuver to complete before continuing with the rest of the program.

### **run\_angle(speed, rotation\_angle, then=Stop.HOLD, wait=True)**

Runs the motor at a constant speed by a given angle.

#### Parameters

- **speed** (*rotational speed: deg/s*) – Speed of the motor.
- **rotation\_angle** (*angle: deg*) – Angle by which the motor should rotate.
- **then** (`Stop`) – What to do after coming to a standstill.
- **wait** (`bool`) – Wait for the maneuver to complete before continuing with the rest of the program.

### **run\_target(speed, target\_angle, then=Stop.HOLD, wait=True)**

Runs the motor at a constant speed towards a given target angle.

The direction of rotation is automatically selected based on the target angle. It does not matter if `speed` is positive or negative.

#### Parameters

- **speed** (*rotational speed: deg/s*) – Speed of the motor.

- **target\_angle** (*angle: deg*) – Angle that the motor should rotate to.
- **then** (*Stop*) – What to do after coming to a standstill.
- **wait** (*bool*) – Wait for the motor to reach the target before continuing with the rest of the program.

**run\_until\_stalled** (*speed, then=Stop.COAST, duty\_limit=None*)

Runs the motor at a constant speed until it stalls.

#### Parameters

- **speed** (*rotational speed: deg/s*) – Speed of the motor.
- **then** (*Stop*) – What to do after coming to a standstill.
- **duty\_limit** (*percentage: %*) – Duty cycle limit during this command. This is useful to avoid applying the full motor torque to a geared or lever mechanism.

**Returns** Angle at which the motor becomes stalled.

**Return type** *angle: deg*

**dc** (*duty*)

Rotates the motor at a given duty cycle (also known as “power”).

This method lets you use a motor just like a simple DC motor.

**Parameters** **duty** (*percentage: %*) – The duty cycle (-100.0 to 100).

## Advanced motion control

**track\_target** (*target\_angle*)

Tracks a target angle. This is similar to `run_target()`, but the usual smooth acceleration is skipped: it will move to the target angle as fast as possible. This method is useful if you want to continuously change the target angle.

**Parameters** **target\_angle** (*angle: deg*) – Target angle that the motor should rotate to.

### control

The motors use PID control to accurately track the speed and angle targets that you specify. You can change its behavior through the `control` attribute of the motor. See [The Control Class](#) for an overview of available methods.

## 2.2.1 Initialization Examples

### Making the motor move back and forth

```
from pybricks.pupdevices import Motor
from pybricks.parameters import Port
from pybricks.tools import wait

# Initialize a motor on port A.
example_motor = Motor(Port.A)

# Make the motor run clockwise at 500 degrees per second.
example_motor.run(500)

# Wait for three seconds.
```

(continues on next page)

(continued from previous page)

```
wait(3000)

# Make the motor run counterclockwise at 500 degrees per second.
example_motor.run(-500)

# Wait for three seconds.
wait(3000)
```

## Initializing multiple motors

```
from pybricks.pupdevices import Motor
from pybricks.parameters import Port
from pybricks.tools import wait

# Initialize motors on port A and B.
track_motor = Motor(Port.A)
gripper_motor = Motor(Port.B)

# Make both motors run at 500 degrees per second.
track_motor.run(500)
gripper_motor.run(500)

# Wait for three seconds.
wait(3000)
```

## Setting the positive direction as counterclockwise

```
from pybricks.pupdevices import Motor
from pybricks.parameters import Port, Direction
from pybricks.tools import wait

# Initialize a motor on port A with the positive direction as counterclockwise.
example_motor = Motor(Port.A, Direction.COUNTERCLOCKWISE)

# When we choose a positive speed value, the motor now goes counterclockwise.
example_motor.run(500)

# This is useful when your motor is mounted in reverse or upside down.
# By changing the positive direction, your script will be easier to read,
# because a positive value now makes your robot/mechanism go forward.

# Wait for three seconds.
wait(3000)
```

## Using gears

```
from pybricks.pupdevices import Motor
from pybricks.parameters import Port, Direction
from pybricks.tools import wait

# Initialize a motor on port A with the positive direction as counterclockwise.
# Also specify one gear train with a 12-tooth and a 36-tooth gear. The 12-tooth
# gear is attached to the motor axle. The 36-tooth gear is at the output axle.
geared_motor = Motor(Port.A, Direction.COUNTERCLOCKWISE, [12, 36])

# Make the output axle run at 100 degrees per second. The motor speed
# is automatically increased to compensate for the gears.
geared_motor.run(100)

# Wait for three seconds.
wait(3000)
```

## 2.2.2 Measurement Examples

### Measuring the angle and speed

```
from pybricks.pupdevices import Motor
from pybricks.parameters import Port
from pybricks.tools import wait

# Initialize a motor on port A.
example_motor = Motor(Port.A)

# Start moving at 300 degrees per second.
example_motor.run(300)

# Display the angle and speed 50 times.
for i in range(100):

    # Read the angle (degrees) and speed (degrees per second).
    angle = example_motor.angle()
    speed = example_motor.speed()

    # Print the values.
    print(angle, speed)

    # Wait some time so we can read what is displayed.
    wait(200)
```

## Resetting the measured angle

```
from pybricks.pupdevices import Motor
from pybricks.parameters import Port

# Initialize a motor on port A.
example_motor = Motor(Port.A)

# Reset the angle to 0.
example_motor.reset_angle(0)

# Reset the angle to 1234.
example_motor.reset_angle(1234)

# Reset the angle to the absolute angle.
# This is only supported on motors that have
# an absolute encoder. For other motors, this
# will raise an error.
example_motor.reset_angle()
```

## 2.2.3 Movement Examples

### Basic usage of all run methods

```
from pybricks.pupdevices import Motor
from pybricks.parameters import Port
from pybricks.tools import wait

# Initialize a motor on port A.
example_motor = Motor(Port.A)

# Run at 500 deg/s and then stop by coasting.
print("Demo of run")
example_motor.run(500)
wait(1500)
example_motor.stop()
wait(1500)

# Run at 70% duty cycle ("power") and then stop by coasting.
print("Demo of dc")
example_motor.dc(50)
wait(1500)
example_motor.stop()
wait(1500)

# Run at 500 deg/s for two seconds.
print("Demo of run_time")
example_motor.run_time(500, 2000)
wait(1500)

# Run at 500 deg/s for 90 degrees.
print("Demo of run_angle")
example_motor.run_angle(500, 90)
wait(1500)
```

(continues on next page)

(continued from previous page)

```
# Run at 500 deg/s back to the 0 angle
print("Demo of run_target to 0")
example_motor.run_target(500, 0)
wait(1500)

# Run at 500 deg/s back to the -90 angle
print("Demo of run_target to -90")
example_motor.run_target(500, -90)
wait(1500)

# Run at 500 deg/s until the motor stalls
print("Demo of run_until_stalled")
example_motor.run_until_stalled(500)
print("Done")
wait(1500)
```

## Stopping ongoing movements in different ways

```
from pybricks.pupdevices import Motor
from pybricks.parameters import Port
from pybricks.tools import wait

# Initialize a motor on port A.
example_motor = Motor(Port.A)

# Run at 500 deg/s and then stop by coasting.
example_motor.run(500)
wait(1500)
example_motor.stop()
wait(1500)

# Run at 500 deg/s and then stop by braking.
example_motor.run(500)
wait(1500)
example_motor.brake()
wait(1500)

# Run at 500 deg/s and then stop by holding.
example_motor.run(500)
wait(1500)
example_motor.hold()
wait(1500)

# Run at 500 deg/s and then stop by running at 0 speed.
example_motor.run(500)
wait(1500)
example_motor.run(0)
wait(1500)
```



## Using the `then` argument to change how a run command stops

```
from pybricks.pupdevices import Motor
from pybricks.parameters import Port, Stop
from pybricks.tools import wait

# Initialize a motor on port A.
example_motor = Motor(Port.A)

# By default, the motor holds the position. It keeps
# correcting the angle if you move it.
example_motor.run_angle(500, 360)
wait(1000)

# This does exactly the same as above.
example_motor.run_angle(500, 360, then=Stop.HOLD)
wait(1000)

# You can also brake. This applies some resistance
# but the motor does not move back if you move it.
example_motor.run_angle(500, 360, then=Stop.BRAKE)
wait(1000)

# This makes the motor coast freely after it stops.
example_motor.run_angle(500, 360, then=Stop.COAST)
wait(1000)
```

## 2.2.4 Stall Examples

### Running a motor until a mechanical endpoint

```
from pybricks.pupdevices import Motor
from pybricks.parameters import Port

# Initialize a motor on port A.
example_motor = Motor(Port.A)

# We'll use a speed of 200 deg/s in all our commands.
speed = 200

# Run the motor in reverse until it hits a mechanical stop.
# The duty_limit=30 setting means that it will apply only 30%
# of the maximum torque against the mechanical stop. This way,
# you don't push against it with too much force.
example_motor.run_until_stalled(-speed, duty_limit=30)

# Reset the angle to 0. Now whenever the angle is 0, you know
# that it has reached the mechanical endpoint.
example_motor.reset_angle(0)

# Now make the motor go back and forth in a loop.
# This will now work the same regardless of the
# initial motor angle, because we always start
# from the mechanical endpoint.
for count in range(10):
```

(continues on next page)

(continued from previous page)

```
example_motor.run_target(speed, 180)
example_motor.run_target(speed, 90)
```

## Centering a steering mechanism

```
from pybricks.pupdevices import Motor
from pybricks.parameters import Port

# Initialize a motor on port A.
example_motor = Motor(Port.A)

# We'll use a speed of 200 deg/s in all our commands.
speed = 200

# Run the motor in reverse until it hits a mechanical stop.
# The duty_limit=30 setting means that it will apply only 30%
# of the maximum torque against the mechanical stop. This way,
# you don't push against it with too much force.
example_motor.run_until_stalled(-speed, duty_limit=30)

# Reset the angle to 0. Now whenever the angle is 0, you know
# that it has reached the mechanical endpoint.
example_motor.reset_angle(0)

# Now make the motor go back and forth in a loop.
# This will now work the same regardless of the
# initial motor angle, because we always start
# from the mechanical endpoint.
for count in range(10):
    example_motor.run_target(speed, 180)
    example_motor.run_target(speed, 90)
```

## 2.2.5 Parallel Movement Examples

### Using the wait argument to run motors in parallel

```
from pybricks.pupdevices import Motor
from pybricks.parameters import Port

# Initialize motors on port A and B.
track_motor = Motor(Port.A)
gripper_motor = Motor(Port.B)

# Make the track motor start moving,
# but don't wait for it to finish.
track_motor.run_angle(500, 360, wait=False)

# Now make the gripper motor rotate. This
# means they move at the same time.
gripper_motor.run_angle(200, 720)
```

## Waiting for two parallel actions to complete

```
from pybricks.pupdevices import Motor
from pybricks.parameters import Port
from pybricks.tools import wait

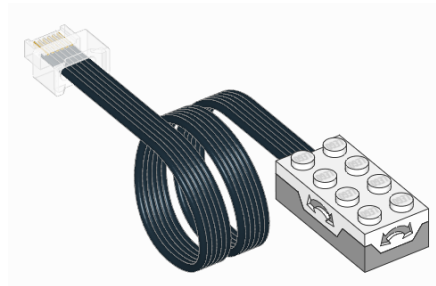
# Initialize motors on port A and B.
track_motor = Motor(Port.A)
gripper_motor = Motor(Port.B)

# Make both motors perform an action with wait=False
track_motor.run_angle(500, 360, wait=False)
gripper_motor.run_angle(200, 720, wait=False)

# While one or both of the motors are not done yet,
# do something else. In this example, just wait.
while not track_motor.control.done() or not gripper_motor.control.done():
    wait(10)

print("Both motors are done!")
```

## 2.3 Tilt Sensor



**class TiltSensor** (*port*)  
LEGO® Powered Up Tilt Sensor.

**Parameters** *port* (*Port*) – Port to which the sensor is connected.

**tilt** ()  
Measures the tilt relative to the horizontal plane.

**Returns** Tuple of pitch and roll angles.

**Return type** (*angle: deg, angle: deg*)

## 2.3.1 Examples

### Measuring pitch and roll

```
from pybricks.pupdevices import TiltSensor
from pybricks.parameters import Port
from pybricks.tools import wait

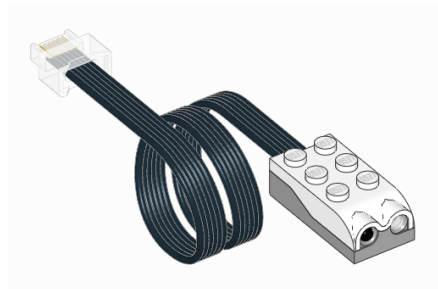
# Initialize the sensor.
accel = TiltSensor(Port.A)

while True:
    # Read the tilt angles relative to the horizontal plane.
    pitch, roll = accel.tilt()

    # Print the values
    print("Pitch:", pitch, "Roll:", roll)

    # Wait some time so we can read what is printed.
    wait(100)
```

## 2.4 Infrared Sensor



**class InfraredSensor** (*port*)

LEGO® Powered Up Infrared Sensor.

**Parameters** *port* (*Port*) – Port to which the sensor is connected.

**distance** ()

Measures the relative distance between the sensor and an object using infrared light.

**Returns** Relative distance ranging from 0 (closest) to 100 (farthest).

**Return type** *relative distance: %*

**reflection** ()

Measures the reflection of a surface using an infrared light.

**Returns** Reflection, ranging from 0.0 (no reflection) to 100.0 (high reflection).

**Return type** *percentage: %*

**count** ()

Counts the number of objects that have passed by the sensor.

**Returns** Number of objects counted.

**Return type** int

## 2.4.1 Examples

### Measuring distance, object count, and reflection

```
from pybricks.pupdevices import InfraredSensor
from pybricks.parameters import Port
from pybricks.tools import wait

# Initialize the sensor.
ir = InfraredSensor(Port.A)

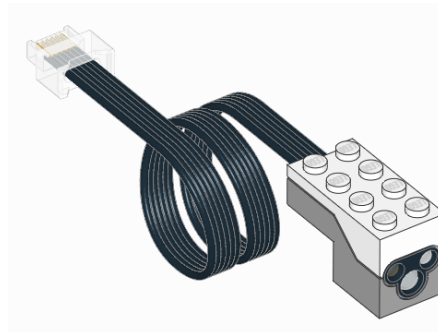
while True:
    # Read all the information we can get from this sensor.
    dist = ir.distance()
    count = ir.count()
    ref = ir.reflection()

    # Print the values
    print("Distance:", dist, "Count:", count, "Reflection:", ref)

    # Move the sensor around and move your hands in front
    # of it to see what happens to the values.

    # Wait some time so we can read what is printed.
    wait(200)
```

## 2.5 Color and Distance Sensor



**class** ColorDistanceSensor(*port*)

LEGO® Powered Up Color and Distance Sensor.

**Parameters** *port* (*Port*) – Port to which the sensor is connected.

**color()**

Scans the color of a surface.

You choose which colors are detected using the *detectable\_colors()* method. By default, it detects *Color.RED*, *Color.YELLOW*, *Color.GREEN*, *Color.BLUE*, *Color.WHITE*, or *Color.NONE*.

**Returns** Detected color.

**Return type** *Color*

**reflection()**

Measures the reflection of a surface.

**Returns** Reflection, ranging from 0.0 (no reflection) to 100.0 (high reflection).

**Return type** *percentage: %*

**ambient()**

Measures the ambient light intensity.

**Returns** Ambient light intensity, ranging from 0 (dark) to 100 (bright).

**Return type** *percentage: %*

**distance()**

Measures the relative distance between the sensor and an object using infrared light.

**Returns** Relative distance ranging from 0 (closest) to 100 (farthest).

**Return type** *relative distance: %*

**hsv()**

Scans the color of a surface.

This method is similar to *color()*, but it gives the full range of hue, saturation and brightness values, instead of rounding it to the nearest detectable color.

**Returns** Measured color. The color is described by a hue (0–359), a saturation (0–100), and a brightness value (0–100).

**Return type** *Color*

**detectable\_colors(colors)**

Configures which colors the *color()* method should detect.

Specify only colors that you wish to detect in your application. This way, the full-color measurements are rounded to the nearest desired color, and other colors are ignored. This improves reliability.

If you give no arguments, the currently chosen colors will be returned as a tuple.

**Parameters** **colors** (*list*) – Tuple of *Color* objects: the colors that you want to detect. You can pick standard colors such as `Color.MAGENTA`, or provide your own colors like `Color(h=348, s=96, v=40)` for even better results. You measure your own colors with the *hsv()* method.

## Built-in light

This sensor has a built-in light. You can make it red, green, blue, or turn it off. If you use the sensor to measure something afterwards, the light automatically turns back on at the default color for that sensing method.

**light.on(color)**

Turns on the light at the specified color.

**Parameters** **color** (*Color*) – Color of the light.

**light.off()**

Turns off the light.

## 2.5.1 Examples

### Measuring color

```
from pybricks.pupdevices import ColorDistanceSensor
from pybricks.parameters import Port
from pybricks.tools import wait

# Initialize the sensor.
sensor = ColorDistanceSensor(Port.A)

while True:
    # Read the color.
    color = sensor.color()

    # Print the measured color.
    print(color)

    # Move the sensor around and see how
    # well you can detect colors.

    # Wait so we can read the value.
    wait(100)
```

### Waiting for a color

```
from pybricks.pupdevices import ColorDistanceSensor
from pybricks.parameters import Port, Color
from pybricks.tools import wait

# Initialize the sensor.
sensor = ColorDistanceSensor(Port.A)

# This is a function that waits for a desired color.
def wait_for_color(desired_color):
    # While the color is not the desired color, we keep waiting.
    while sensor.color() != desired_color:
        wait(20)

# Now we use the function we just created above.
while True:

    # Here you can make your train/vehicle go forward.

    print("Waiting for red ...")
    wait_for_color(Color.RED)

    # Here you can make your train/vehicle go backward.

    print("Waiting for blue ...")
    wait_for_color(Color.BLUE)
```

## Measuring distance

```
from pybricks.pupdevices import ColorDistanceSensor
from pybricks.parameters import Port, Color
from pybricks.tools import wait

# Initialize the sensor.
sensor = ColorDistanceSensor(Port.A)

# Repeat forever.
while True:

    # If the sensor sees an object nearby.
    if sensor.distance() <= 40:

        # Then blink the light red/blue 5 times.
        for i in range(5):
            sensor.light.on(Color.RED)
            wait(30)
            sensor.light.on(Color.BLUE)
            wait(30)
    else:
        # If the sensor sees nothing
        # nearby, just wait briefly.
        wait(10)
```

## Blinking the built-in light

```
from pybricks.pupdevices import ColorDistanceSensor
from pybricks.parameters import Port, Color
from pybricks.tools import wait

# Initialize the sensor.
sensor = ColorDistanceSensor(Port.A)

# Repeat forever.
while True:

    # If the sensor sees an object nearby.
    if sensor.distance() <= 40:

        # Then blink the light red/blue 5 times.
        for i in range(5):
            sensor.light.on(Color.RED)
            wait(30)
            sensor.light.on(Color.BLUE)
            wait(30)
    else:
        # If the sensor sees nothing
        # nearby, just wait briefly.
        wait(10)
```



## Reading hue, saturation, value

```
from pybricks.pupdevices import ColorDistanceSensor
from pybricks.parameters import Port
from pybricks.tools import wait

# Initialize the sensor.
sensor = ColorDistanceSensor(Port.A)

while True:
    # The standard color() method always "rounds" the
    # measurement to the nearest "whole" color.
    # That's useful for most applications.

    # But you can get the original hue, saturation,
    # and value without "rounding", as follows:
    color = sensor.hsv()

    # Print the results.
    print(color)

    # Wait so we can read the value.
    wait(500)
```

## Changing the detectable colors

By default, the sensor is configured to detect red, yellow, green, blue, white, or no color, which suits many applications.

For better results in your application, you can measure your desired colors in advance, and tell the sensor to look only for those colors. Be sure to measure them at the **same distance and light conditions** as in your final application. Then you'll get very accurate results even for colors that are otherwise hard to detect.

```
from pybricks.pupdevices import ColorDistanceSensor
from pybricks.parameters import Port, Color
from pybricks.tools import wait

# Initialize the sensor.
sensor = ColorDistanceSensor(Port.A)

# First, decide which objects you want to detect, and measure their HSV values.
# You can do that with the hsv() method as shown in the previous example.
#
# Use your measurements to override the default colors, or add new colors:
Color.GREEN = Color(h=132, s=94, v=26)
Color.MAGENTA = Color(h=348, s=96, v=40)
Color.BROWN = Color(h=17, s=78, v=15)
Color.RED = Color(h=359, s=97, v=39)

# Put your colors in a list or tuple.
my_colors = (Color.GREEN, Color.MAGENTA, Color.BROWN, Color.RED, Color.NONE)

# Save your colors.
sensor.detectable_colors(my_colors)

# color() works as usual, but now it returns one of your specified colors.
while True:
```

(continues on next page)

(continued from previous page)

```

color = sensor.color()

# Print the color.
print(color)

# Check which one it is.
if color == Color.MAGENTA:
    print("It works!")

# Wait so we can read it.
wait(100)

```

## 2.6 Power Functions

The *ColorDistanceSensor* can send infrared signals to control Power Functions infrared receivers. You can use this technique to control medium, large, extra large, and train motors. The infrared range is limited to about 30 cm, depending on the angle and ambient conditions.

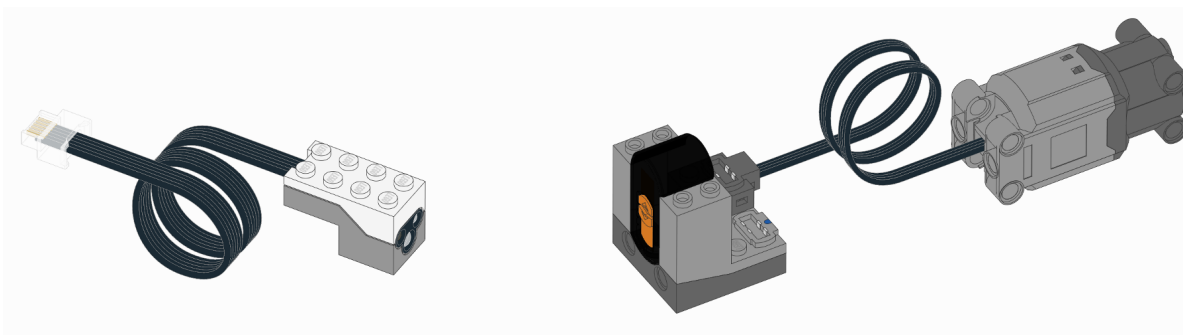


Figure 2.3: Powered Up *ColorDistanceSensor* (left), Power Functions infrared receiver (middle), and a Power Functions motor (right). Here, the receiver uses channel 1 with a motor on the red port.

**class PFMotor** (*sensor, channel, color, positive\_direction=Direction.CLOCKWISE*)  
 Control Power Functions motors with the infrared functionality of the *ColorDistanceSensor*.

### Parameters

- **sensor** (*ColorDistanceSensor*) – Sensor object.
- **channel** (*int*) – Channel number of the receiver: 1, 2, 3, or 4.
- **color** (*Color*) – Color marker on the receiver: *Color.BLUE* or *Color.RED*
- **positive\_direction** (*Direction*) – Which direction the motor should turn when you give a positive duty cycle value.

### dc (*duty*)

Rotates the motor at a given duty cycle (also known as “power”).

**Parameters** **duty** (*percentage: %*) – The duty cycle (-100.0 to 100).

### stop ()

Stops the motor and lets it spin freely.

The motor gradually stops due to friction.

**brake()**

Passively brakes the motor.

The motor stops due to friction, plus the voltage that is generated while the motor is still moving.

## 2.6.1 Examples

### Control a Power Functions motor

```
from pybricks.pupdevices import ColorDistanceSensor, PFMotor
from pybricks.parameters import Port, Color
from pybricks.tools import wait

# Initialize the sensor.
sensor = ColorDistanceSensor(Port.B)

# Initialize a motor on channel 1, on the red output.
motor = PFMotor(sensor, 1, Color.RED)

# Rotate and then stop.
motor.dc(100)
wait(1000)
motor.stop()
wait(1000)

# Rotate the other way at half speed, and then stop.
motor.dc(-50)
wait(1000)
motor.stop()
```

### Controlling multiple Power Functions motors

```
from pybricks.pupdevices import ColorDistanceSensor, PFMotor
from pybricks.parameters import Port, Color, Direction
from pybricks.tools import wait

# Initialize the sensor.
sensor = ColorDistanceSensor(Port.B)

# You can use multiple motors on different channels.
arm = PFMotor(sensor, 1, Color.BLUE)
wheel = PFMotor(sensor, 4, Color.RED, Direction.COUNTERCLOCKWISE)

# Accelerate both motors. Only these values are available.
# Other values will be rounded down to the nearest match.
for duty in [15, 30, 45, 60, 75, 90, 100]:
    arm.dc(duty)
    wheel.dc(duty)
    wait(1000)

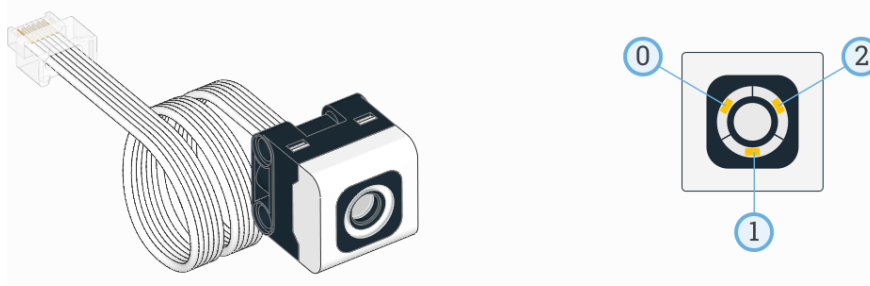
# To make the signal more reliable, there is a short
# pause between commands. So, they change speed and
# stop at a slightly different time.
```

(continues on next page)

(continued from previous page)

```
# Brake both motors.
arm.brake()
wheel.brake()
```

## 2.7 Color Sensor



**class ColorSensor** (*port*)  
LEGO® SPIKE Color Sensor.

**Parameters** *port* (*Port*) – Port to which the sensor is connected.

**color** (*surface=True*)  
Scans the color of a surface or an external light source.

You choose which colors are detected using the *detectable\_colors()* method. By default, it detects *Color.RED*, *Color.YELLOW*, *Color.GREEN*, *Color.BLUE*, *Color.WHITE*, or *Color.NONE*.

**Parameters** *surface* (*bool*) – Choose *true* to scan the color of objects and surfaces.  
Choose *false* to scan the color of screens and other external light sources.

**Returns** Detected color.

**Return type** *Color*

**reflection** ()  
Measures the reflection of a surface.

**Returns** Reflection, ranging from 0.0 (no reflection) to 100.0 (high reflection).

**Return type** *percentage: %*

**ambient** ()  
Measures the ambient light intensity.

**Returns** Ambient light intensity, ranging from 0 (dark) to 100 (bright).

**Return type** *percentage: %*

## Advanced color sensing

**hsv** (*surface=True*)

Scans the color of a surface or an external light source.

This method is similar to `color()`, but it gives the full range of hue, saturation and brightness values, instead of rounding it to the nearest detectable color.

**Parameters** **surface** (*bool*) – Choose `true` to scan the color of objects and surfaces. Choose `false` to scan the color of screens and other external light sources.

**Returns** Measured color. The color is described by a hue (0–359), a saturation (0–100), and a brightness value (0–100).

**Return type** *Color*

**detectable\_colors** (*colors*)

Configures which colors the `color()` method should detect.

Specify only colors that you wish to detect in your application. This way, the full-color measurements are rounded to the nearest desired color, and other colors are ignored. This improves reliability.

If you give no arguments, the currently chosen colors will be returned as a tuple.

**Parameters** **colors** (*list*) – Tuple of *Color* objects: the colors that you want to detect. You can pick standard colors such as `Color.MAGENTA`, or provide your own colors like `Color(h=348, s=96, v=40)` for even better results. You measure your own colors with the `hsv()` method.

## Built-in lights

This sensor has 3 built-in lights. You can adjust the brightness of each light. If you use the sensor to measure something, the lights will be turned on or off as needed for the measurement.

**lights.on** (*brightness*)

Turns on the lights at the specified brightness.

**Parameters** **brightness** (tuple of *brightness: %*) – Brightness of each light, in the order shown above. If you give one brightness value instead of a tuple, all lights get the same brightness.

**lights.off** ()

Turns off all the lights.

## 2.7.1 Examples

### Measuring color and reflection

```
from pybricks.pupdevices import ColorSensor
from pybricks.parameters import Port
from pybricks.tools import wait

# Initialize the sensor.
sensor = ColorSensor(Port.A)

while True:
    # Read the color and reflection
```

(continues on next page)

(continued from previous page)

```
color = sensor.color()
reflection = sensor.reflection()

# Print the measured color and reflection.
print(color, reflection)

# Move the sensor around and see how
# well you can detect colors.

# Wait so we can read the value.
wait(100)
```

## Waiting for a color

```
from pybricks.pupdevices import ColorSensor
from pybricks.parameters import Port, Color
from pybricks.tools import wait

# Initialize the sensor.
sensor = ColorSensor(Port.A)

# This is a function that waits for a desired color.
def wait_for_color(desired_color):
    # While the color is not the desired color, we keep waiting.
    while sensor.color() != desired_color:
        wait(20)

# Now we use the function we just created above.
while True:

    # Here you can make your train/vehicle go forward.

    print("Waiting for red ...")
    wait_for_color(Color.RED)

    # Here you can make your train/vehicle go backward.

    print("Waiting for blue ...")
    wait_for_color(Color.BLUE)
```

## Reading *reflected* hue, saturation, and value

```
from pybricks.pupdevices import ColorSensor
from pybricks.parameters import Port
from pybricks.tools import wait

# Initialize the sensor.
sensor = ColorSensor(Port.A)

while True:
    # The standard color() method always "rounds" the
```

(continues on next page)

(continued from previous page)

```
# measurement to the nearest "whole" color.
# That's useful for most applications.

# But you can get the original hue, saturation,
# and value without "rounding", as follows:
color = sensor.hsv()

# Print the results.
print(color)

# Wait so we can read the value.
wait(500)
```

## Changing the detectable colors

By default, the sensor is configured to detect red, yellow, green, blue, white, or no color, which suits many applications.

For better results in your application, you can measure your desired colors in advance, and tell the sensor to look only for those colors. Be sure to measure them at the **same distance and light conditions** as in your final application. Then you'll get very accurate results even for colors that are otherwise hard to detect.

```
from pybricks.pupdevices import ColorSensor
from pybricks.parameters import Port, Color
from pybricks.tools import wait

# Initialize the sensor.
sensor = ColorSensor(Port.A)

# First, decide which objects you want to detect, and measure their HSV values.
# You can do that with the hsv() method as shown in the previous example.
#
# Use your measurements to override the default colors, or add new colors:
Color.GREEN = Color(h=132, s=94, v=26)
Color.MAGENTA = Color(h=348, s=96, v=40)
Color.BROWN = Color(h=17, s=78, v=15)
Color.RED = Color(h=359, s=97, v=39)

# Put your colors in a list or tuple.
my_colors = (Color.GREEN, Color.MAGENTA, Color.BROWN, Color.RED, Color.NONE)

# Save your colors.
sensor.detectable_colors(my_colors)

# color() works as usual, but now it returns one of your specified colors.
while True:
    color = sensor.color()

    # Print the color.
    print(color)

    # Check which one it is.
    if color == Color.MAGENTA:
        print("It works!")

    # Wait so we can read it.
    wait(100)
```

## Reading *ambient* hue, saturation, value, and color

```
from pybricks.pupdevices import ColorSensor
from pybricks.parameters import Port
from pybricks.tools import wait

# Initialize the sensor.
sensor = ColorSensor(Port.A)

# Repeat forever.
while True:

    # Get the ambient color values. Instead of scanning the color of a surface,
    # this lets you scan the color of light sources like lamps or screens.
    hsv = sensor.hsv(surface=False)
    color = sensor.color(surface=False)

    # Get the ambient light intensity.
    ambient = sensor.ambient()

    # Print the measurements.
    print(hsv, color, ambient)

    # Point the sensor at a computer screen or colored light. Watch the color.
    # Also, cover the sensor with your hands and watch the ambient value.

    # Wait so we can read the printed line
    wait(100)
```

## Blinking the built-in lights

```
from pybricks.pupdevices import ColorSensor
from pybricks.parameters import Port
from pybricks.tools import wait

# Initialize the sensor.
sensor = ColorSensor(Port.A)

# Repeat forever.
while True:

    # Turn on one light at a time, at half the brightness.
    # Do this for all 3 lights and repeat that 5 times.
    for i in range(5):
        sensor.lights.on([50, 0, 0])
        wait(100)
        sensor.lights.on([0, 50, 0])
        wait(100)
        sensor.lights.on([0, 0, 50])
        wait(100)

    # Turn all lights on at maximum brightness.
    sensor.lights.on(100)
    wait(500)

    # Turn all lights off.
```

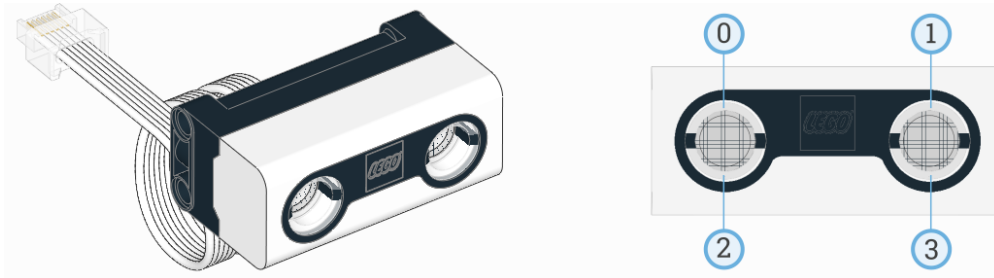
(continues on next page)



(continued from previous page)

```
sensor.lights.off()
wait(500)
```

## 2.8 Ultrasonic Sensor



**class** `UltrasonicSensor` (*port*)

LEGO® SPIKE Color Sensor.

**Parameters** `port` (`Port`) – Port to which the sensor is connected.

**distance** ()

Measures the distance between the sensor and an object using ultrasonic sound waves.

**Returns** Measured distance. If no valid distance was measured, it returns 2000 mm.

**Return type** *distance*: `mm`

**presence** ()

Checks for the presence of other ultrasonic sensors by detecting ultrasonic sounds.

**Returns** `True` if ultrasonic sounds are detected, `False` if not.

**Return type** `bool`

### Built-in lights

This sensor has 4 built-in lights. You can adjust the brightness of each light.

`lights.on(brightness)`

Turns on the lights at the specified brightness.

**Parameters** `brightness` (tuple of *brightness*: `%`) – Brightness of each light, in the order shown above. If you give one brightness value instead of a tuple, all lights get the same brightness.

`lights.off()`

Turns off all the lights.

## 2.8.1 Examples

### Measuring distance and switching on the lights

```
from pybricks.pupdevices import UltrasonicSensor
from pybricks.parameters import Port
from pybricks.tools import wait

# Initialize the sensor.
eyes = UltrasonicSensor(Port.A)

while True:
    # Print the measured distance.
    print(eyes.distance())

    # If an object is detected closer than 500mm:
    if eyes.distance() < 500:
        # Turn the lights on.
        eyes.lights.on(100)
    else:
        # Turn the lights off.
        eyes.lights.off()

    # Wait some time so we can read what is printed.
    wait(100)
```

### Gradually change the brightness of the lights

```
from pybricks.pupdevices import UltrasonicSensor
from pybricks.parameters import Port
from pybricks.tools import wait, Stopwatch

# The math module is part of standard MicroPython:
# https://docs.micropython.org/en/latest/library/math.html
from math import pi, sin

# Initialize the sensor.
eyes = UltrasonicSensor(Port.A)

# Initialize a timer.
watch = Stopwatch()

# We want one full light cycle to last three seconds.
PERIOD = 3000

while True:
    # The phase is where we are in the unit circle now.
    phase = watch.time()/PERIOD*2*pi

    # Each light follows a sine wave with a mean of 50, with an amplitude of 50.
    # We offset this sine wave by 90 degrees for each light, so that all the
    # lights do something different.
    brightness = [sin(phase + offset*pi/2) * 50 + 50 for offset in range(4)]

    # Set the brightness values for all lights.
```

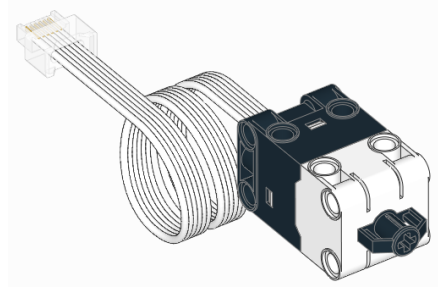
(continues on next page)

(continued from previous page)

```
eyes.lights.on(brightness)

# Wait some time.
wait(50)
```

## 2.9 Force Sensor



**class ForceSensor** (*port*)

LEGO® SPIKE Force Sensor.

**Parameters** *port* (*Port*) – Port to which the sensor is connected.

**force** ()

Measures the force exerted on the sensor.

**Returns** Measured force (up to approximately 10.00 N).

**Return type** *force*: *N*

**distance** ()

Measures by how much the sensor button has moved.

**Returns** How much the sensor button has moved (up to approximately 8.00 mm).

**Return type** *distance*: *mm*

**pressed** (*force=3*)

Checks if the sensor button is pressed.

**Parameters** *force* (*force*: *N*) – Minimum force to be considered pressed.

**Returns** `True` if the sensor is pressed, `False` if it is not.

**Return type** `bool`

**touched** ()

Checks if the sensor is touched.

This is similar to `pressed()`, but it detects slight movements of the button even when the measured force is still considered zero.

**Returns** `True` if the sensor is touched or pressed, `False` if it is not.

**Return type** `bool`

## 2.9.1 Examples

### Measuring force and movement

```
from pybricks.pupdevices import ForceSensor
from pybricks.parameters import Port
from pybricks.tools import wait

# Initialize the sensor.
button = ForceSensor(Port.A)

while True:
    # Read all the information we can get from this sensor.
    force = button.force()
    dist = button.distance()
    press = button.pressed()
    touch = button.touched()

    # Print the values
    print("Force", force, "Dist:", dist, "Pressed:", press, "Touched:", touch)

    # Push the sensor button see what happens to the values.

    # Wait some time so we can read what is printed.
    wait(200)
```

### Measuring peak force

```
from pybricks.pupdevices import ForceSensor
from pybricks.parameters import Port
from pybricks.tools import wait

# Initialize the sensor.
button = ForceSensor(Port.A)

# This function waits until the button is pushed. It keeps track of the maximum
# detected force until the button is released. Then it returns the maximum.
def wait_for_force():

    # Wait for a force, by doing nothing for as long the force is nearly zero.
    print("Waiting for force.")
    while button.force() <= 0.1:
        wait(10)

    # Now we wait for the release, by waiting for the force to be zero again.
    print("Waiting for release.")

    # While we wait for that to happen, we keep reading the force and remember
    # the maximum force. We do this by initializing the maximum at 0, and
    # updating it each time we detect a bigger force.
    maximum = 0
    force = 10
    while force > 0.1:
        # Read the force.
```

(continues on next page)

(continued from previous page)

```

    force = button.force()

    # Update the maximum if the measured force is larger.
    if force > maximum:
        maximum = force

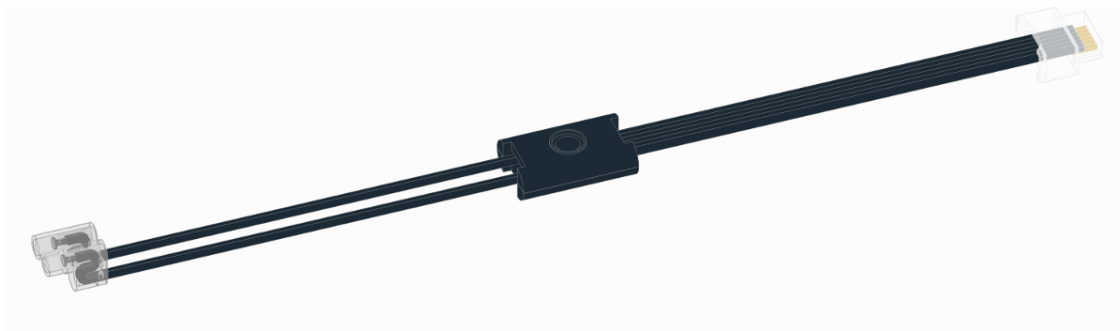
    # Wait and then measure again.
    wait(10)

    # Return the maximum force.
    return maximum

# Keep waiting for the sensor button to be pushed. When it is, display
# the peak force and repeat.
while True:
    peak = wait_for_force()
    print("Released. Peak force: {0} N\n".format(peak))

```

## 2.10 Light



**class Light** (*port*)

LEGO® Powered Up Light.

**Parameters** **port** (*Port*) – Port to which the device is connected.

**on** (*brightness=100*)

Turns on the light at the specified brightness.

**Parameters** **brightness** (*brightness: %*) – Brightness of the light.

**off** ()

Turns off the light.

## 2.10.1 Examples

### Making the light blink

```
from pybricks.pupdevices import Light
from pybricks.parameters import Port
from pybricks.tools import wait

# Initialize the light.
light = Light(Port.A)

# Blink the light forever.
while True:
    # Turn the light on at 100% brightness.
    light.on(100)
    wait(500)

    # Turn the light off.
    light.off()
    wait(500)
```

### Gradually change the brightness

```
from pybricks.pupdevices import Light
from pybricks.parameters import Port
from pybricks.tools import wait, Stopwatch

# The math module is part of standard MicroPython:
# https://docs.micropython.org/en/latest/library/math.html
from math import pi, cos

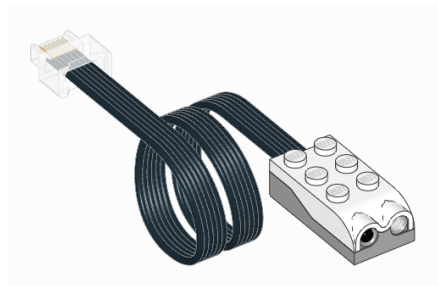
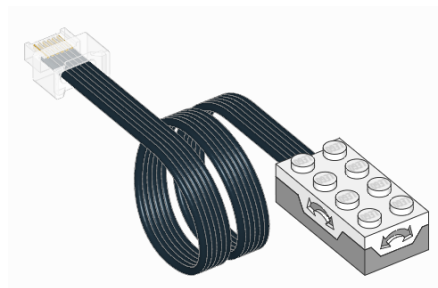
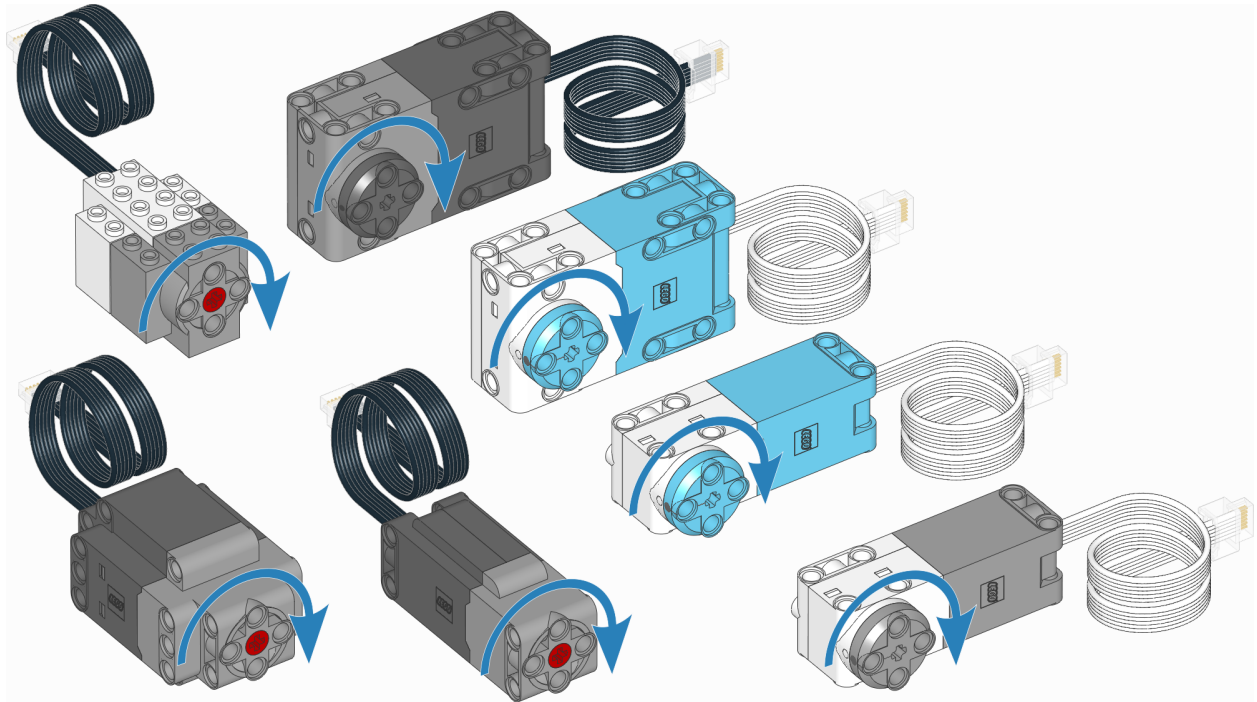
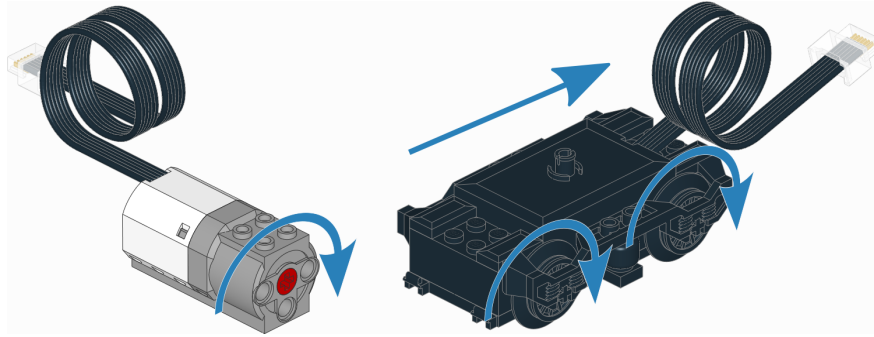
# Initialize the light and a Stopwatch.
light = Light(Port.A)
watch = Stopwatch()

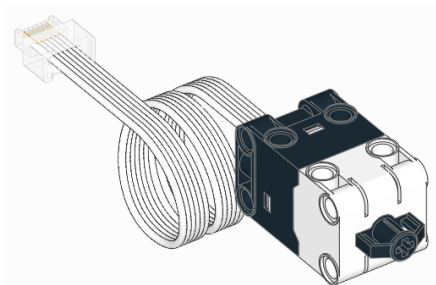
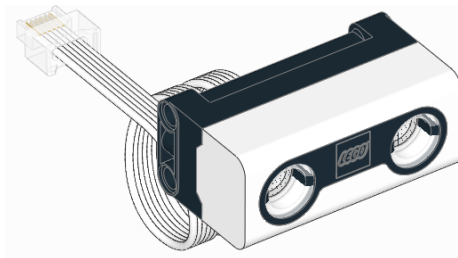
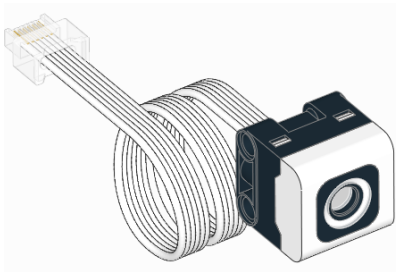
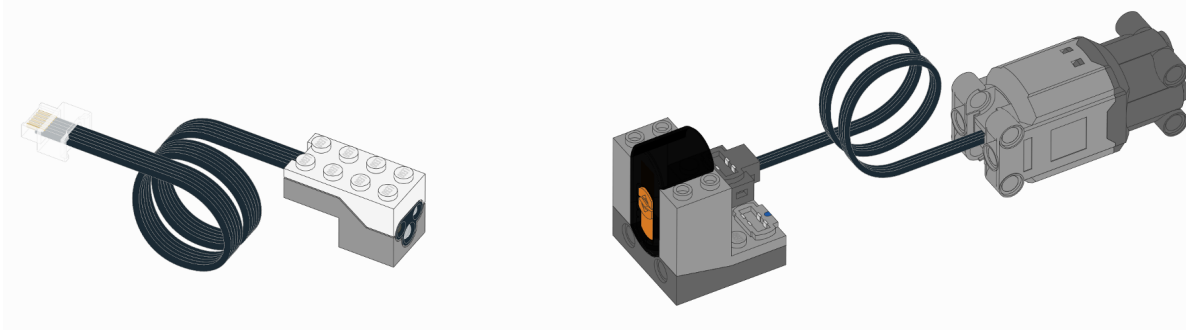
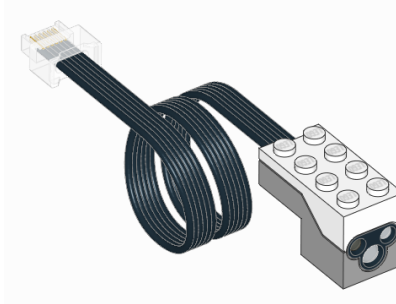
# Cosine pattern properties.
PERIOD = 2000
MAX = 100

# Make the brightness fade in and out.
while True:
    # Get phase of the cosine.
    phase = watch.time()/PERIOD*2*pi

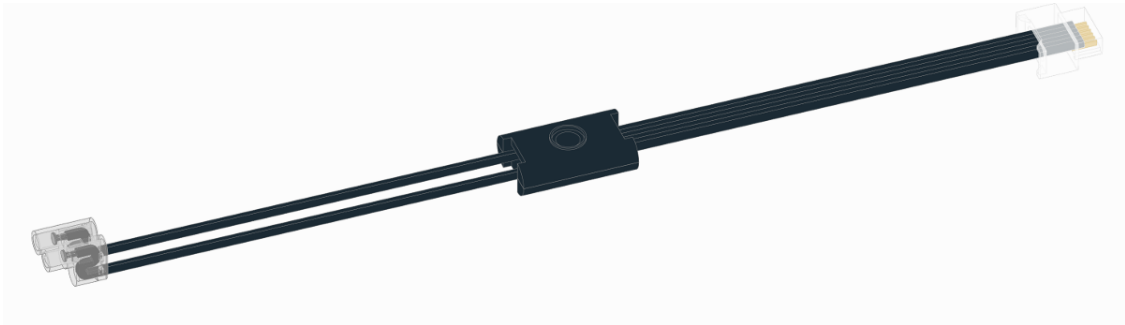
    # Evaluate the brightness.
    brightness = (0.5 - 0.5*cos(phase))*MAX

    # Set light brightness and wait a bit.
    light.on(brightness)
    wait(10)
```









## EV3DEVICES – EV3 DEVICES

LEGO® MINDSTORMS® EV3 motors and sensors.

### 3.1 Motors

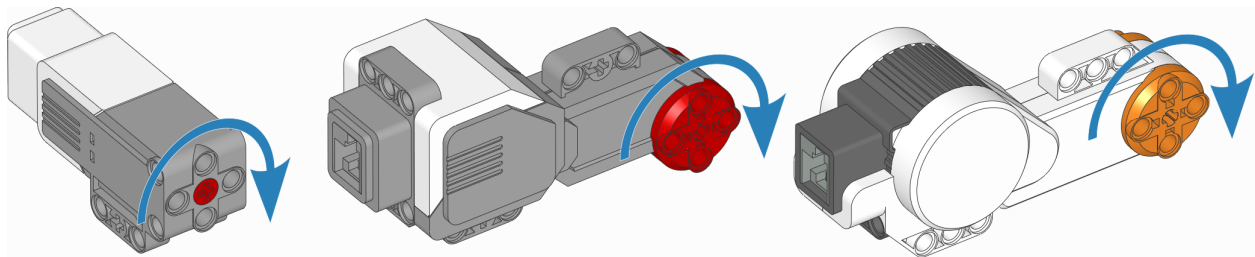


Figure 3.1: EV3-compatible motors. The arrows indicate the default positive direction.

**class Motor** (*port*, *positive\_direction*=*Direction.CLOCKWISE*, *gears*=*None*)  
Generic class to control motors with built-in rotation sensors.

#### Parameters

- **port** (*Port*) – Port to which the motor is connected.
- **positive\_direction** (*Direction*) – Which direction the motor should turn when you give a positive speed value or angle.
- **gears** (*list*) – List of gears linked to the motor.

For example: `[12, 36]` represents a gear train with a 12-tooth and a 36-tooth gear. Use a list of lists for multiple gear trains, such as `[[12, 36], [20, 16, 40]]`.

When you specify a gear train, all motor commands and settings are automatically adjusted to account for the resulting gear ratio. The motor direction remains unchanged by this.

## Measuring

### **speed** ()

Gets the speed of the motor.

**Returns** Motor speed.

**Return type** *rotational speed: deg/s*

### **angle** ()

Gets the rotation angle of the motor.

**Returns** Motor angle.

**Return type** *angle: deg*

### **reset\_angle** (*angle*)

Sets the accumulated rotation angle of the motor to a desired value.

**Parameters** **angle** (*angle: deg*) – Value to which the angle should be reset.

## Stopping

### **stop** ()

Stops the motor and lets it spin freely.

The motor gradually stops due to friction.

### **brake** ()

Passively brakes the motor.

The motor stops due to friction, plus the voltage that is generated while the motor is still moving.

### **hold** ()

Stops the motor and actively holds it at its current angle.

## Action

### **run** (*speed*)

Runs the motor at a constant speed.

The motor accelerates to the given speed and keeps running at this speed until you give a new command.

**Parameters** **speed** (*rotational speed: deg/s*) – Speed of the motor.

### **run\_time** (*speed, time, then=Stop.HOLD, wait=True*)

Runs the motor at a constant speed for a given amount of time.

The motor accelerates to the given speed, keeps running at this speed, and then decelerates. The total maneuver lasts for exactly the given amount of `time`.

#### **Parameters**

- **speed** (*rotational speed: deg/s*) – Speed of the motor.
- **time** (*time: ms*) – Duration of the maneuver.
- **then** (`Stop`) – What to do after coming to a standstill.
- **wait** (*bool*) – Wait for the maneuver to complete before continuing with the rest of the program.

**run\_angle** (*speed*, *rotation\_angle*, *then=Stop.HOLD*, *wait=True*)

Runs the motor at a constant speed by a given angle.

#### Parameters

- **speed** (*rotational speed: deg/s*) – Speed of the motor.
- **rotation\_angle** (*angle: deg*) – Angle by which the motor should rotate.
- **then** (*Stop*) – What to do after coming to a standstill.
- **wait** (*bool*) – Wait for the maneuver to complete before continuing with the rest of the program.

**run\_target** (*speed*, *target\_angle*, *then=Stop.HOLD*, *wait=True*)

Runs the motor at a constant speed towards a given target angle.

The direction of rotation is automatically selected based on the target angle. It does not matter if *speed* is positive or negative.

#### Parameters

- **speed** (*rotational speed: deg/s*) – Speed of the motor.
- **target\_angle** (*angle: deg*) – Angle that the motor should rotate to.
- **then** (*Stop*) – What to do after coming to a standstill.
- **wait** (*bool*) – Wait for the motor to reach the target before continuing with the rest of the program.

**run\_until\_stalled** (*speed*, *then=Stop.COAST*, *duty\_limit=None*)

Runs the motor at a constant speed until it stalls.

#### Parameters

- **speed** (*rotational speed: deg/s*) – Speed of the motor.
- **then** (*Stop*) – What to do after coming to a standstill.
- **duty\_limit** (*percentage: %*) – Duty cycle limit during this command. This is useful to avoid applying the full motor torque to a geared or lever mechanism.

**Returns** Angle at which the motor becomes stalled.

**Return type** *angle: deg*

**dc** (*duty*)

Rotates the motor at a given duty cycle (also known as “power”).

This method lets you use a motor just like a simple DC motor.

**Parameters** **duty** (*percentage: %*) – The duty cycle (-100.0 to 100).

## Advanced motion control

**track\_target** (*target\_angle*)

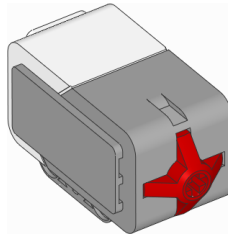
Tracks a target angle. This is similar to *run\_target()*, but the usual smooth acceleration is skipped: it will move to the target angle as fast as possible. This method is useful if you want to continuously change the target angle.

**Parameters** **target\_angle** (*angle: deg*) – Target angle that the motor should rotate to.

**control**

The motors use PID control to accurately track the speed and angle targets that you specify. You can change its behavior through the `control` attribute of the motor. See [The Control Class](#) for an overview of available methods.

## 3.2 Touch Sensor



```
class TouchSensor(port)
```

LEGO® MINDSTORMS® EV3 Touch Sensor.

**Parameters** `port` ([Port](#)) – Port to which the sensor is connected.

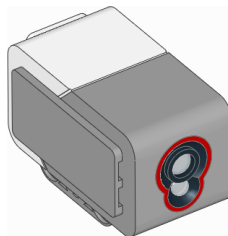
```
pressed()
```

Checks if the sensor is pressed.

**Returns** `True` if the sensor is pressed, `False` if it is not pressed.

**Return type** `bool`

## 3.3 Color Sensor



```
class ColorSensor(port)
```

LEGO® MINDSTORMS® EV3 Color Sensor.

**Parameters** `port` ([Port](#)) – Port to which the sensor is connected.

```
color()
```

Measures the color of a surface.

**Returns** `Color.BLACK`, `Color.BLUE`, `Color.GREEN`, `Color.YELLOW`, `Color.RED`, `Color.WHITE`, `Color.BROWN` or `None`.

**Return type** [Color](#), or `None` if no color is detected.

```
ambient()
```

Measures the ambient light intensity.

**Returns** Ambient light intensity, ranging from 0 (dark) to 100 (bright).

**Return type** *percentage: %*

**reflection()**

Measures the reflection of a surface using a red light.

**Returns** Reflection, ranging from 0 (no reflection) to 100 (high reflection).

**Return type** *percentage: %*

**rgb()**

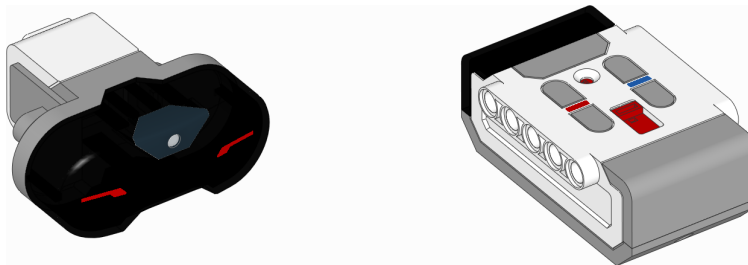
Measures the reflection of a surface using a red, green, and then a blue light.

**Returns** Tuple of reflections for red, green, and blue light, each ranging from 0.0 (no reflection) to 100.0 (high reflection).

**Return type** *(percentage: %, percentage: %, percentage: %)*

## 3.4 Infrared Sensor and Beacon

Each method of this class puts the sensor in a different *mode*. Switching modes takes about one second on this sensor. To make sure that your program runs quickly, use only of these methods in your program.



**class InfraredSensor** (*port*)

LEGO® MINDSTORMS® EV3 Infrared Sensor and Beacon.

**Parameters** **port** (*Port*) – Port to which the sensor is connected.

**distance()**

Measures the relative distance between the sensor and an object using infrared light.

**Returns** Relative distance ranging from 0 (closest) to 100 (farthest).

**Return type** *relative distance: %*

**beacon** (*channel*)

Measures the relative distance and angle between the remote and the infrared sensor.

**Parameters** **channel** (*int*) – Channel number of the remote.

**Returns** Tuple of relative distance (0 to 100) and approximate angle (-75 to 75 degrees) between remote and infrared sensor.

**Return type** *(relative distance: %, angle: deg)* or (None, None) if no remote is detected.

**buttons** (*channel*)

Checks which buttons on the infrared remote are pressed.

This method can detect up to two buttons at once. If you press more buttons, you may not get useful data.

**Parameters** **channel** (*int*) – Channel number of the remote.

**Returns** List of pressed buttons on the remote on selected channel.

**Return type** List of *Button*

**keypad()**

Checks which buttons on the infrared remote are pressed.

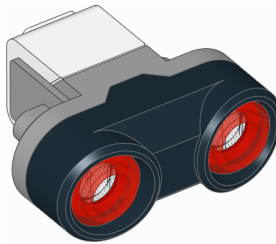
This method can independently detect all 4 up/down buttons, but it cannot detect the beacon button.

This method only works with the remote in channel 1.

**Returns** List of pressed buttons on the remote on selected channel.

**Return type** List of *Button*

## 3.5 Ultrasonic Sensor



**class UltrasonicSensor** (*port*)

LEGO® MINDSTORMS® EV3 Ultrasonic Sensor.

**Parameters** *port* (*Port*) – Port to which the sensor is connected.

**distance** (*silent=False*)

Measures the distance between the sensor and an object using ultrasonic sound waves.

**Parameters** *silent* (*bool*) – Choose `True` to turn the sensor off after measuring the distance.

This reduces interference with other ultrasonic sensors. If you do this too frequently, the sensor can freeze. If this happens, unplug it and plug it back in.

**Returns** Distance.

**Return type** *distance: mm*

**presence** ()

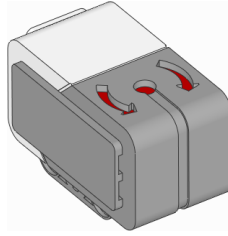
Checks for the presence of other ultrasonic sensors by detecting ultrasonic sounds.

If the other ultrasonic sensor is operating in silent mode, you can only detect the presence of that sensor while it is taking a measurement.

**Returns** `True` if ultrasonic sounds are detected, `False` if not.

**Return type** `bool`

## 3.6 Gyroscopic Sensor



```
class GyroSensor (port, positive_direction=Direction.CLOCKWISE)  
    LEGO® MINDSTORMS® EV3 Gyro Sensor.
```

### Parameters

- **port** (`Port`) – Port to which the sensor is connected.
- **positive\_direction** (`Direction`) – Positive rotation direction when looking at the red dot on top of the sensor.

### `speed()`

Gets the speed (angular velocity) of the sensor.

**Returns** Sensor angular velocity.

**Return type** *rotational speed: deg/s*

### `angle()`

Gets the accumulated angle of the sensor.

**Returns** Rotation angle.

**Return type** *angle: deg*

If you use the `angle()` method, you cannot use the `speed()` method in the same program. Doing so would reset the sensor angle to zero every time you read the speed.

### `reset_angle(angle)`

Sets the rotation angle of the sensor to a desired value.

**Parameters** **angle** (*angle: deg*) – Value to which the angle should be reset.



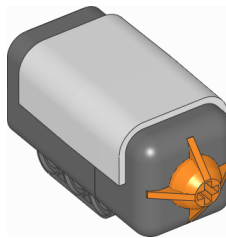
## NXTDEVICES – NXT DEVICES

Use LEGO® MINDSTORMS® NXT motors and sensors with the EV3 brick.

### 4.1 NXT Motor

This motor works just like a LEGO MINDSTORMS EV3 Large Motor. You can use it in your programs using the *Motor* class.

### 4.2 NXT Touch Sensor



```
class TouchSensor (port)
    LEGO® MINDSTORMS® NXT Touch Sensor.

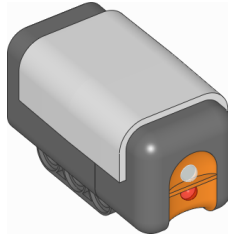
    Parameters port (Port) – Port to which the sensor is connected.

    pressed ()
        Checks if the sensor is pressed.

        Returns True if the sensor is pressed, False if it is not pressed.

        Return type bool
```

## 4.3 NXT Light Sensor



```
class LightSensor(port)
```

LEGO® MINDSTORMS® NXT Color Sensor.

**Parameters** `port` (`Port`) – Port to which the sensor is connected.

**ambient** ()

Measures the ambient light intensity.

**Returns** Ambient light intensity, ranging from 0 (dark) to 100 (bright).

**Return type** *percentage: %*

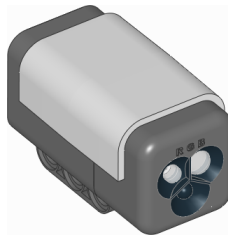
**reflection** ()

Measures the reflection of a surface using a red light.

**Returns** Reflection, ranging from 0 (no reflection) to 100 (high reflection).

**Return type** *percentage: %*

## 4.4 NXT Color Sensor



```
class ColorSensor(port)
```

LEGO® MINDSTORMS® NXT Color Sensor.

**Parameters** `port` (`Port`) – Port to which the sensor is connected.

**color** ()

Measures the color of a surface.

**Returns** `Color.BLACK`, `Color.BLUE`, `Color.GREEN`, `Color.YELLOW`, `Color.RED`, `Color.WHITE` or `Color.NONE`.

**Return type** *Color*

**ambient** ()

Measures the ambient light intensity.

**Returns** Ambient light intensity, ranging from 0 (dark) to 100 (bright).

**Return type** *percentage: %*

**reflection()**

Measures the reflection of a surface.

**Returns** Reflection, ranging from 0 (no reflection) to 100 (high reflection).

**Return type** *percentage: %*

**rgb()**

Measures the reflection of a surface using a red, green, and then a blue light.

**Returns** Tuple of reflections for red, green, and blue light, each ranging from 0.0 (no reflection) to 100.0 (high reflection).

**Return type** (*percentage: %*, *percentage: %*, *percentage: %*)

### Built-in light

This sensor has a built-in light. You can make it red, green, blue, or turn it off.

**light.on(*color*)**

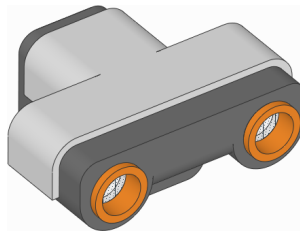
Turns on the light at the specified color.

**Parameters** **color** (*Color*) – Color of the light.

**light.off()**

Turns off the light.

## 4.5 NXT Ultrasonic Sensor



**class UltrasonicSensor(*port*)**

LEGO® MINDSTORMS® NXT Ultrasonic Sensor.

**Parameters** **port** (*Port*) – Port to which the sensor is connected.

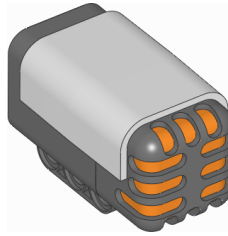
**distance()**

Measures the distance between the sensor and an object using ultrasonic sound waves.

**Returns** Distance.

**Return type** *distance: mm*

## 4.6 NXT Sound Sensor



**class** `SoundSensor` (*port*)

LEGO® MINDSTORMS® NXT Sound Sensor.

**Parameters** `port` (`Port`) – Port to which the sensor is connected.

**intensity** (*audible\_only=True*)

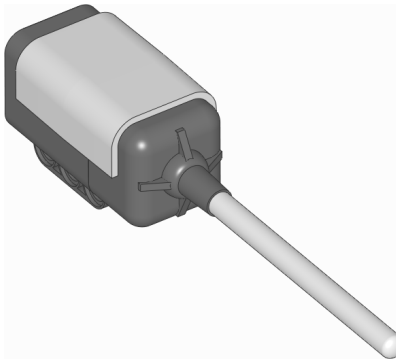
Measures the ambient sound intensity (loudness).

**Parameters** `audible_only` (*bool*) – Detect only audible sounds. This tries to filter out frequencies that cannot be heard by the human ear.

**Returns** Sound intensity.

**Return type** *percentage: %*

## 4.7 NXT Temperature Sensor



**class** `TemperatureSensor` (*port*)

LEGO® MINDSTORMS® NXT Temperature Sensor.

**Parameters** `port` (`Port`) – Port to which the sensor is connected.

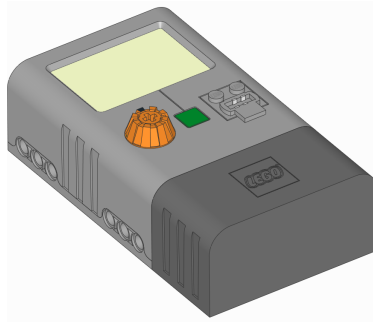
**temperature** ()

Measures the temperature.

**Returns** Measured temperature.

**Return type** *temperature: °C*

## 4.8 NXT Energy Meter



**class EnergyMeter** (*port*)

LEGO® MINDSTORMS® Education NXT Energy Meter.

**Parameters** **port** (*Port*) – Port to which the sensor is connected.

**storage** ()

Gets the total available energy stored in the battery.

**Returns** Remaining stored energy.

**Return type** *energy: J*

**input** ()

Measures the electrical signals at the input (bottom) side of the energy meter. It measures the voltage applied to it and the current passing through it. The product of these two values is power. This power value is the rate at which the stored energy increases. This power is supplied by an energy source such as the provided solar panel or an externally driven motor.

**Returns** Voltage, current, and power measured at the input port.

**Return type** (*voltage: mV, current: mA, power: mW*)

**output** ()

Measures the electrical signals at the output (top) side of the energy meter. It measures the voltage applied to the external load and the current passing to it. The product of these two values is power. This power value is the rate at which the stored energy decreases. This power is consumed by the load, such as a light or a motor.

**Returns** Voltage, current, and power measured at the output port.

**Return type** (*voltage: mV, current: mA, power: mW*)

## 4.9 Vernier Adapter

**class VernierAdapter** (*port, conversion=None*)

LEGO® MINDSTORMS® Education NXT/EV3 Adapter for Vernier Sensors.

**Parameters**

- **port** (*Port*) – Port to which the sensor is connected.
- **conversion** (*callable*) – Function of the format *conversion*. This function is used to convert the raw analog voltage to the sensor-specific output value. Each Vernier Sensor has its own conversion function. The example given below demonstrates the conversion for the Surface Temperature Sensor.

**voltage()**

Measures the raw analog sensor voltage.

**Returns** Analog voltage.

**Return type** *voltage: mV*

**conversion(voltage)**

Converts the raw voltage (mV) to a sensor value.

If you did not provide a `conversion` function earlier, no conversion will be applied.

**Parameters** **voltage** (*voltage: mV*) – Analog sensor voltage

**Returns** Converted sensor value.

**Return type** float

**value()**

Measures the sensor *voltage()* and then applies your *conversion()* to give you the sensor value.

**Returns** Converted sensor value.

**Return type** float

**Example: Using the Surface Temperature Sensor.**

```
#!/usr/bin/env pybricks-micropython
from pybricks.parameters import Port
from pybricks.nxtdevices import VernierAdapter

from math import log

# Conversion formula for Surface Temperature Sensor
def convert_raw_to_temperature(voltage):

    # Convert the raw voltage to the NTC resistance
    # according to the Vernier Adapter EV3 block.
    counts = voltage/5000*4096
    ntc = 15000*(counts)/(4130-counts)

    # Handle log(0) safely: make sure that ntc value is positive.
    if ntc <= 0:
        ntc = 1

    # Apply Steinhart-Hart equation as given in the sensor documentation.
    K0 = 1.02119e-3
    K1 = 2.22468e-4
    K2 = 1.33342e-7
    return 1/(K0 + K1*log(ntc) + K2*log(ntc)**3)

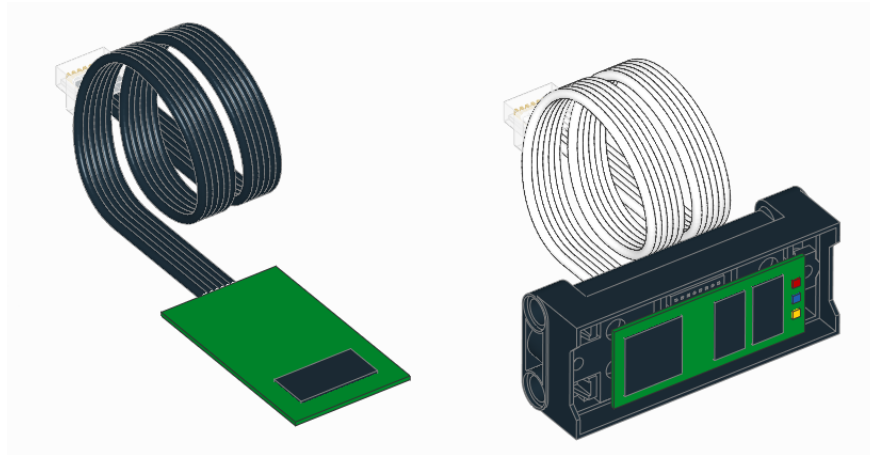
# Initialize the adapter on port 1
thermometer = VernierAdapter(Port.S1, convert_raw_to_temperature)

# Get the measured value and print it
temp = thermometer.value()
print(temp)
```

## IODEVICES – GENERIC I/O DEVICES

Generic input/output devices.

### 5.1 Powered Up Device



```
class PUPDevice (port)
    Powered Up motor or sensor.

    Parameters port (Port) – Port to which the device is connected.

    info ()
        Returns information about the device.

        Returns Dictionary with information, such as the device id.

        Return type dict

    read (mode)
        Reads values from a given mode.

        Parameters mode (int) – Device mode.

        Returns Values read from the sensor.

        Return type tuple

    write (mode, data)
        Writes values to the sensor. Only selected sensors and modes support this.

        Parameters
```

- **mode** (int) – Device mode.
- **data** (tuple) – Values to be written.

## 5.1.1 Examples

### Detecting devices

```
from pybricks.iodevices import PUPDevice
from pybricks.parameters import Port
from uerrno import ENODEV

# Dictionary of device identifiers along with their name.
device_names = {
    34: "Wedo 2.0 Tilt Sensor",
    35: "Wedo 2.0 Infrared Sensor",
    37: "BOOST Color Distance Sensor",
    38: "BOOST Interactive Motor",
    46: "Technic Large Motor",
    47: "Technic Extra Large Motor",
    48: "SPIKE Medium Angular Motor",
    49: "SPIKE Large Angular Motor",
    61: "SPIKE Color Sensor",
    62: "SPIKE Ultrasonic Sensor",
    63: "SPIKE Force Sensor",
    75: "Technic Medium Angular Motor",
    76: "Technic Large Angular Motor",
}

# Make a list of known ports.
ports = [Port.A, Port.B]

# On hubs that support it, add more ports.
try:
    ports.append(Port.C)
    ports.append(Port.D)
except AttributeError:
    pass

# On hubs that support it, add more ports.
try:
    ports.append(Port.E)
    ports.append(Port.F)
except AttributeError:
    pass

# Go through all available ports.
for port in ports:

    # Try to get the device, if it is attached.
    try:
        device = PUPDevice(port)
    except OSError as ex:
        if ex.args[0] == ENODEV:
            # No device found on this port.
            print(port, ": ---")
```

(continues on next page)



(continued from previous page)

```

        continue
    else:
        raise

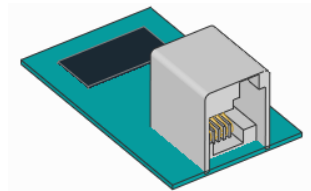
# Get the device id
id = device.info()['id']

# Look up the name.
try:
    print(port, ":", device_names[id])
except KeyError:
    print(port, ":", "Unknown device with ID", id)

```

## 5.2 Generic I2C Device

**Note:** This class is **only supported on the EV3** at this time. It could be added to Powered Up hubs in a future release. If you'd like to see this happen, be sure to ask us on our [support page](#).



**class** `I2CDevice` (*port, address*)  
Generic or custom I2C device.

### Parameters

- **port** (`Port`) – Port to which the device is connected.
- **address** (`int`) – I2C address of the client device. See *I2C Addresses*.

**read** (*reg, length=1*)

Reads bytes, starting at a given register.

### Parameters

- **reg** (`int`) – Register at which to begin reading: 0–255 or 0x00–0xFF.
- **length** (`int`) – How many bytes to read.

**Returns** Bytes returned from the device.

**Return type** `bytes`

**write** (*reg, data=None*)

Writes bytes, starting at a given register.

### Parameters

- **reg** (`int`) – Register at which to begin writing: 0–255 or 0x00–0xFF.
- **data** (`bytes`) – Bytes to be written.

**Example: Read and write to an I2C device**

```
#!/usr/bin/env pybricks-micropython
from pybricks.hubs import EV3Brick
from pybricks.iodevices import I2CDevice
from pybricks.parameters import Port

# Initialize the EV3
ev3 = EV3Brick()

# Initialize I2C Sensor
device = I2CDevice(Port.S2, 0xD2 >> 1)

# Read one byte from the device.
# For this device, we can read the Who Am I
# register (0x0F) for the expected value: 211.
if 211 not in device.read(0x0F):
    raise ValueError("Unexpected I2C device ID")

# To write data, create a bytes object of one
# or more bytes. For example:
# data = bytes((1, 2, 3))

# Write one byte (value 0x08) to register 0x22
device.write(0x22, bytes((0x08,)))
```

## 5.2.1 I2C Addresses

I2C addresses are 7-bit values. However, most vendors who make LEGO compatible sensors provide an 8-bit address in their documentation. To use those addresses, you must shift them by 1 bit. For example, if the documented address is 0xD2, you can do `address = 0xD2 >> 1`.

## 5.2.2 Advanced I2C Commands

Some rudimentary I2C devices do not require a register argument or even any data. You can achieve this behavior as shown in the examples below.

### Example: Advanced I2C read and write techniques

```
#!/usr/bin/env pybricks-micropython
from pybricks.hubs import EV3Brick
from pybricks.iodevices import I2CDevice
from pybricks.parameters import Port

# Initialize the EV3
ev3 = EV3Brick()

# Initialize I2C Sensor
device = I2CDevice(Port.S2, 0xD2 >> 1)

# Recommended for reading
result, = device.read(reg=0x0F, length=1)

# Read 1 byte from no particular register:
device.read(reg=None, length=1)
```

(continues on next page)

(continued from previous page)

```
# Read 0 bytes from no particular register:
device.read(reg=None, length=0)

# I2C write operations consist of a register byte followed
# by a series of data bytes. Depending on your device, you
# can choose to skip the register or data as follows:

# Recommended for writing:
device.write(reg=0x22, data=b'\x08')

# Write 1 byte to no particular register:
device.write(reg=None, data=b'\x08')

# Write 0 bytes to a particular register:
device.write(reg=0x08, data=None)

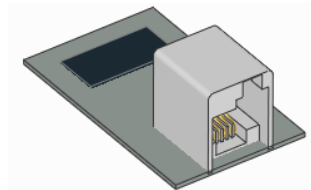
# Write 0 bytes to no particular register:
device.write(reg=None, data=None)
```

### Additional technical resources

The `I2CDevice` class methods call functions from the Linux SMBus driver. To find out which commands are called under the hood, check the [Pybricks source code](#). More details about using I2C without MicroPython can be found on the [ev3dev I2C](#) page.

## 5.3 Generic UART Device

**Note:** This class is **only supported on the EV3** at this time. It could be added to Powered Up hubs in a future release. If you'd like to see this happen, be sure to ask us on our [support page](#).



**class** `UARTDevice` (*port*, *baudrate*, *timeout=None*)  
Generic UART device.

#### Parameters

- **port** (`Port`) – Port to which the device is connected.
- **baudrate** (*int*) – Baudrate of the UART device.
- **timeout** (*time: ms*) – How long to wait during `read` before giving up. If you choose `None`, it will wait forever.

#### **read** (*length=1*)

Reads a given number of bytes from the buffer.

Your program will wait until the requested number of bytes are received. If this takes longer than `timeout`, the `ETIMEDOUT` exception is raised.

**Parameters** `length` (int) – How many bytes to read.

**Returns** Bytes returned from the device.

**Return type** bytes

**`read_all()`**

Reads all bytes from the buffer.

**Returns** Bytes returned from the device.

**Return type** bytes

**`write(data)`**

Writes bytes.

**Parameters** `data` (bytes) – Bytes to be written.

**`waiting()`**

Gets how many bytes are still waiting to be read.

**Returns** Number of bytes in the buffer.

**Return type** int

**`clear()`**

Empties the buffer.

#### Example: Read and write to a UART device

```
#!/usr/bin/env pybricks-micropython
from pybricks.hubs import EV3Brick
from pybricks.iodevices import UARTDevice
from pybricks.parameters import Port
from pybricks.media.ev3dev import SoundFile

# Initialize the EV3
ev3 = EV3Brick()

# Initialize sensor port 2 as a uart device
ser = UARTDevice(Port.S2, baudrate=115200)

# Write some data
ser.write(b'\r\nHello, world!\r\n')

# Play a sound while we wait for some data
for i in range(3):
    ev3.speaker.play_file(SoundFile.HELLO)
    ev3.speaker.play_file(SoundFile.GOOD)
    ev3.speaker.play_file(SoundFile.MORNING)
    print("Bytes waiting to be read:", ser.waiting())

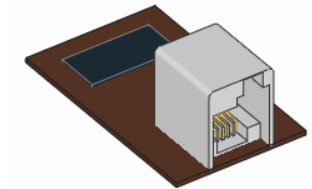
# Read all data received while the sound was playing
data = ser.read_all()
print(data)
```

## 5.4 EV3 Analog Sensor

---

**Note:** This class is only available on EV3.

---



**class** `AnalogSensor` (*port*)

Generic or custom analog sensor.

**Parameters** `port` (`Port`) – Port to which the sensor is connected.

**voltage** ()

Measures analog voltage.

**Returns** Analog voltage.

**Return type** *voltage: mV*

**resistance** ()

Measures resistance.

This value is only meaningful if the analog device is a passive load such as a resistor or thermistor.

**Returns** Resistance of the analog device.

**Return type** *resistance:*

**active** ()

Sets sensor to active mode. This sets pin 5 of the sensor port to *high*.

This is used in some analog sensors to control a switch. For example, if you use the NXT Light Sensor as a custom analog sensor, this method will turn the light on. From then on, `voltage()` returns the raw reflected light value.

**passive** ()

Sets sensor to passive mode. This sets pin 5 of the sensor port to *low*.

This is used in some analog sensors to control a switch. For example, if you use the NXT Light Sensor as a custom analog sensor, this method will turn the light off. From then on, `voltage()` returns the raw ambient light value.

## 5.5 EV3 UART Device

---

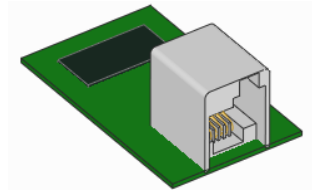
**Note:** This class is only available on EV3.

---

**class** `LUMPDevice` (*port*)

Devices using the LEGO UART Messaging Protocol.

**Parameters** `port` (`Port`) – Port to which the device is connected.



**read** (*mode*)

Reads values from a given mode.

**Parameters** *mode* (*int*) – Device mode.

**Returns** Values read from the sensor.

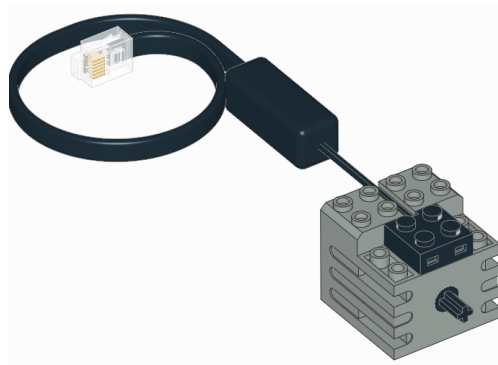
**Return type** *tuple*

## 5.6 EV3 DC Motor

---

**Note:** This class is specifically for on EV3. For Powered Up DC Motors, just use the *DCMotor* class.

---



**class** *DCMotor* (*port*, *positive\_direction*=*Direction.CLOCKWISE*)

Generic class to control simple motors without rotation sensors, such as train motors.

**Parameters**

- **port** (*Port*) – Port to which the motor is connected.
- **positive\_direction** (*Direction*) – Which direction the motor should turn when you give a positive duty cycle value.

**dc** (*duty*)

Rotates the motor at a given duty cycle (also known as “power”).

**Parameters** *duty* (*percentage: %*) – The duty cycle (-100.0 to 100).

**stop** ()

Stops the motor and lets it spin freely.

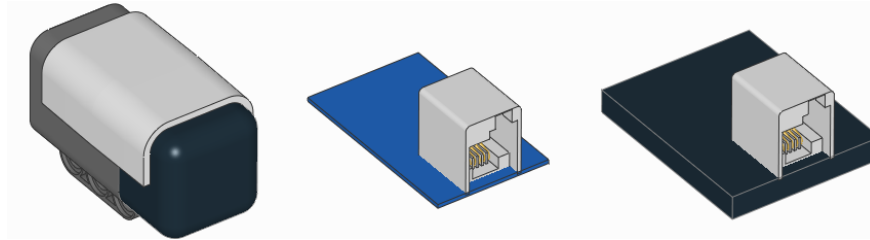
The motor gradually stops due to friction.

## 5.7 Ev3dev sensors

---

**Note:** This class is only available on EV3.

---



EV3 MicroPython is built on top of ev3dev, which means that a sensor may be supported even if it is not listed in this documentation. If so, you can use it with the `Ev3devSensor` class. This is easier and faster than using the custom device classes given above.

To check whether you can use the `Ev3devSensor` class:

- Plug the sensor into your EV3 Brick.
- Go to the main menu of the EV3 Brick.
- Select *Device Browser* and then *Sensors*.
- If your sensor shows up, you can use it.

Now select your sensor from the menu and choose *set mode*. This shows all available modes for this sensor. You can use these mode names as the `mode` setting below.

To learn more about compatible devices and what each mode does, visit the [ev3dev sensors](#) page.

**class** `Ev3devSensor` (`port`)

Read values of an ev3dev-compatible sensor.

**Parameters** `port` (`Port`) – Port to which the device is connected.

**sensor\_index**

Index of the ev3dev sysfs [lego-sensor](#) class.

**port\_index**

Index of the ev3dev sysfs [lego-port](#) class.

**read** (`mode`)

Reads values at a given mode.

**Parameters** `mode` (`str`) – [Mode name](#).

**Returns** Values read from the sensor.

**Return type** `tuple`

### Example: Reading values with the `Ev3devSensor` class

In this example we use the LEGO MINDSTORMS EV3 Color Sensor with the raw RGB mode. This gives uncalibrated red, green, and blue reflection values.

```
#!/usr/bin/env pybricks-micropython
from pybricks.parameters import Port
from pybricks.tools import wait
```

(continues on next page)

(continued from previous page)

```

from pybricks.iodevices import Ev3devSensor

# Initialize an Ev3devSensor.
# In this example we use the
# LEGO MINDSTORMS EV3 Color Sensor.
sensor = Ev3devSensor(Port.S3)

while True:
    # Read the raw RGB values
    r, g, b = sensor.read('RGB-RAW')

    # Print results
    print('R: {0}\t G: {1}\t B: {2}'.format(r, g, b))

    # Wait
    wait(200)

```

**Example: Extending the Ev3devSensor class**

This example shows how to extend the Ev3devSensor class by accessing additional features found in the Linux system folder for this device.

```

#!/usr/bin/env pybricks-micropython
from pybricks.parameters import Port
from pybricks.iodevices import Ev3devSensor

class MySensor(Ev3devSensor):
    """Example of extending the Ev3devSensor class."""

    def __init__(self, port):
        """Initialize the sensor."""

        # Initialize the parent class.
        super().__init__(port)

        # Get the sysfs path.
        self.path = '/sys/class/lego-sensor/sensor' + str(self.sensor_index)

    def get_modes(self):
        """Get a list of mode strings so we don't have to look them up."""

        # The path of the modes file.
        modes_path = self.path + '/modes'

        # Open the modes file.
        with open(modes_path, 'r') as m:

            # Read the contents.
            contents = m.read()

            # Strip the newline symbol, and split at every space symbol.
            return contents.strip().split(' ')

# Initialize the sensor
sensor = MySensor(Port.S3)

```

(continues on next page)



(continued from previous page)

```
# Show where this sensor can be found
print(sensor.path)

# Print the available modes
modes = sensor.get_modes()
print(modes)

# Read mode 0 of this sensor
val = sensor.read(modes[0])
print(val)
```

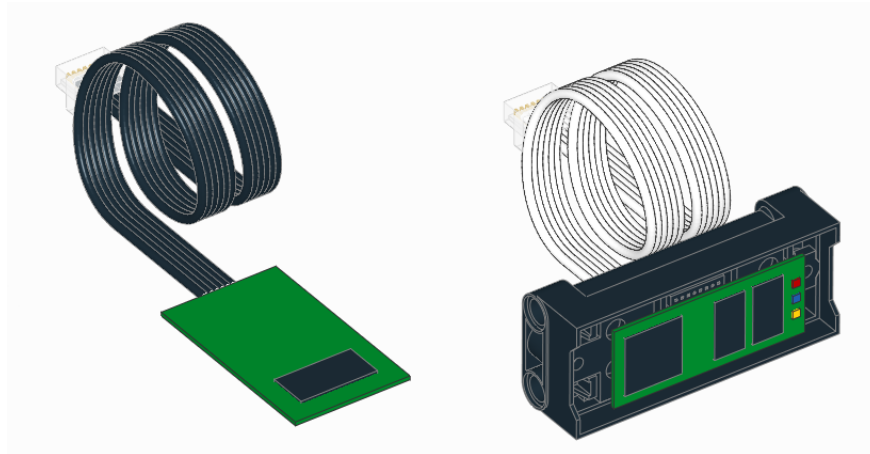
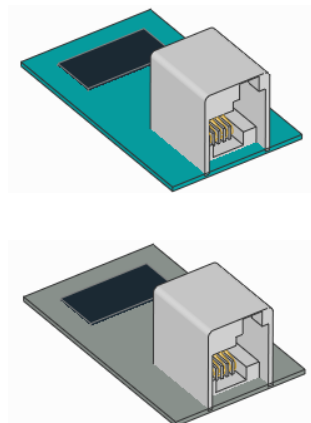
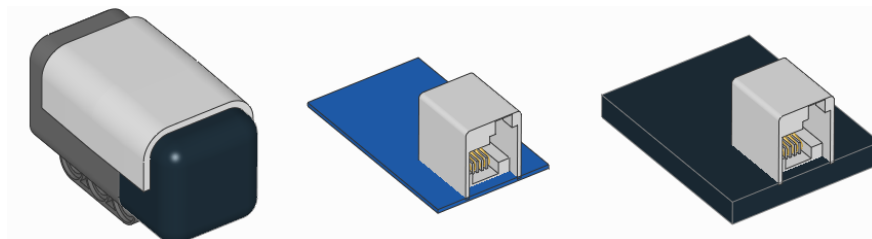
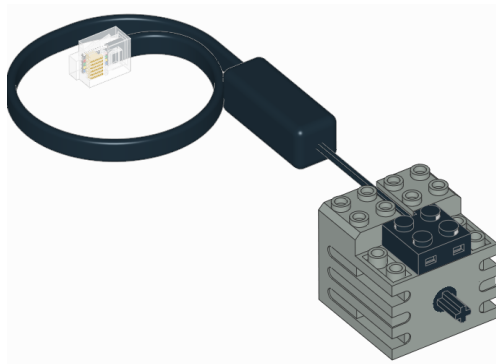
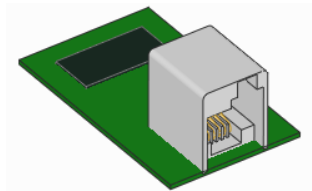
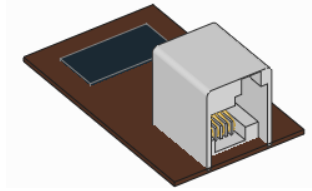


Figure 5.1: This class is only available on Powered Up hubs.





## PARAMETERS – PARAMETERS AND CONSTANTS

Constant parameters/arguments for the Pybricks API.

### 6.1 Button

```
class Button
    Buttons on a hub or remote.

    LEFT_DOWN
    LEFT_MINUS
    DOWN
    RIGHT_DOWN
    RIGHT_MINUS
    LEFT
    CENTER
    RIGHT
    LEFT_UP
    LEFT_PLUS
    UP
    BEACON
    RIGHT_UP
    RIGHT_PLUS
```

### 6.2 Color

```
class Color(h, s=100, v=100)
    Light or surface color.
```

## Saturated colors

These colors have maximum saturation and brightness value. They differ only in hue.

```
RED = Color(h=0, s=100, v=100)
```

```
ORANGE = Color(h=30, s=100, v=100)
```

```
YELLOW = Color(h=60, s=100, v=100)
```

```
GREEN = Color(h=120, s=100, v=100)
```

```
CYAN = Color(h=180, s=100, v=100)
```

```
BLUE = Color(h=240, s=100, v=100)
```

```
VIOLET = Color(h=270, s=100, v=100)
```

```
MAGENTA = Color(h=300, s=100, v=100)
```

## Unsaturated colors

These colors have zero hue and saturation. They differ only in brightness value.

When detecting these colors using sensors, their values depend a lot on the distance to the object. If the distance between the sensor and the object is not constant in your robot, it is better to use only one of these colors in your programs.

```
WHITE = Color(h=0, s=0, v=100)
```

```
GRAY = Color(h=0, s=0, v=50)
```

```
BLACK = Color(h=0, s=0, v=10)
```

This represents dark objects that still reflect a very small amount of light.

```
NONE = Color(h=0, s=0, v=0)
```

This is total darkness, with no reflection or light at all.

## Making your own colors

This example shows the basics of color properties, and how to define new colors.

```
from pybricks.parameters import Color

# You can print colors. Colors may be obtained from the Color class, or
# from sensors that return color measurements.
print(Color.RED)

# You can read hue, saturation, and value properties.
```

(continues on next page)

(continued from previous page)

```
print(Color.RED.h, Color.RED.s, Color.RED.v)

# You can make your own colors. Saturation and value are 100 by default.
my_green = Color(h=125)
my_dark_green = Color(h=125, s=80, v=30)

# When you print custom colors, you see exactly how they were defined.
print(my_dark_green)

# You can also add colors to the builtin colors.
Color.MY_DARK_BLUE = Color(h=235, s=80, v=30)

# When you add them like this, printing them only shows its name. But you can
# still read h, s, v by reading its attributes.
print(Color.MY_DARK_BLUE)
print(Color.MY_DARK_BLUE.h, Color.MY_DARK_BLUE.s, Color.MY_DARK_BLUE.v)
```

This example shows more advanced use cases of the Color class.

```
from pybricks.parameters import Color

# Two colors are equal if their h, s, and v attributes are equal.
if Color.BLUE == Color(240, 100, 100):
    print("Yes, these colors are the same.")

# You can scale colors to change their brightness value.
red_dark = Color.RED * 0.5

# You can shift colors to change their hue.
red_shifted = Color.RED >> 30

# Colors are immutable, so you can't change h, s, or v of an existing object.
try:
    Color.GREEN.h = 125
except AttributeError:
    print("Sorry, can't change the hue of an existing color object!")

# But you can override builtin colors by defining a whole new color.
Color.GREEN = Color(h=125)

# You can access and store colors as class attributes, or as a dictionary.
print(Color.BLUE)
print(Color["BLUE"])
print(Color["BLUE"] is Color.BLUE)
print(Color)
print([c for c in Color])

# This allows you to update existing colors in a loop.
for name in ("BLUE", "RED", "GREEN"):
    Color[name] = Color(1, 2, 3)
```

## 6.3 Direction

### **class Direction**

Rotational direction for positive speed or angle values.

#### **CLOCKWISE**

A positive speed value should make the motor move clockwise.

#### **COUNTERCLOCKWISE**

A positive speed value should make the motor move counterclockwise.

positive_direction =	Positive speed:	Negative speed:
Direction.CLOCKWISE	clockwise	counterclockwise
Direction.COUNTERCLOCKWISE	counterclockwise	clockwise

In general, clockwise is defined by **looking at the motor shaft, just like looking at a clock**. Some motors have two shafts. If in doubt, refer to the diagram in the `Motor` class documentation.

## 6.4 Port

### **class Port**

Input and output ports:

**A**

**B**

**C**

**D**

**E**

**F**

EV3 Sensor ports:

**S1**

**S2**

**S3**

**S4**

## 6.5 Stop

### **class Stop**

Action after the motor stops.

#### **COAST**

Let the motor move freely.

#### **BRAKE**

Passively resist small external forces.

**HOLD**

Keep controlling the motor to hold it at the commanded angle. This is only available on motors with encoders.

The following table shows how each stop type adds an extra level of resistance to motion. In these examples, *m* is a *Motor* and *d* is a *DriveBase*. The examples also show how running at zero speed compares to these stop types.

Type	Friction	Back EMF	Speed kept at 0	Angle kept at target	Examples
Coast	•				<pre>m.stop() m.run_target(500, 90, Stop.COAST)</pre>
Brake	•	•			<pre>m.brake() m.run_target(500, 90, Stop.BRAKE)</pre>
	•	•	•		<pre>m.run(0) d.drive(0, 0)</pre>
Hold	•	•	•	•	<pre>m.hold() m.run_target(500, 90, Stop.HOLD) d.straight(0) d.straight(100)</pre>

## TOOLS – GENERAL PURPOSE TOOLS

Common tools for timing and data logging.

### 7.1 Data logging

At the moment, this class is only available on EV3.

**class DataLog** (*\*headers, name='log', timestamp=True, extension='csv', append=False*)  
Create a file and log data.

#### Parameters

- **headers** (*coll, col2, ...*) – Column headers. These are the names of the data columns. For example, choose 'time' and 'angle'.
- **name** (*str*) – Name of the file.
- **timestamp** (*bool*) – Choose `True` to add the date and time to the file name. This way, your file has a unique name. Choose `False` to omit the timestamp.
- **extension** (*str*) – File extension.
- **append** (*bool*) – Choose `True` to reopen an existing data log file and append data to it. Choose `False` to clear existing data. If the file does not exist yet, an empty file will be created either way.

**log** (*\*values*)

Saves one or more values on a new line in the file.

**Parameters values** (*object, object, ...*) – One or more objects or values.

By default, this class creates a `csv` file on the EV3 brick with the name `log` and the current date and time. For example, if you use this class on 13 February 2020 on 10:07 and 44.431260 seconds, the file is called `log_2020_02_13_10_07_44_431260.csv`.

See [managing files on the EV3](#) to learn how to upload the log file back to your computer.



## 7.1.1 Examples

### Logging and visualizing measurements

This example shows how to log the angle of a rotating wheel as time passes.

```
#!/usr/bin/env pybricks-micropython
from pybricks.ev3devices import Motor
from pybricks.parameters import Port
from pybricks.tools import DataLog, StopWatch, wait

# Create a data log file in the project folder on the EV3 Brick.
# * By default, the file name contains the current date and time, for example:
#   log_2020_02_13_10_07_44_431260.csv
# * You can optionally specify the titles of your data columns. For example,
#   if you want to record the motor angles at a given time, you could do:
data = DataLog('time', 'angle')

# Initialize a motor and make it move
wheel = Motor(Port.B)
wheel.run(500)

# Start a stopwatch to measure elapsed time
watch = StopWatch()

# Log the time and the motor angle 10 times
for i in range(10):
    # Read angle and time
    angle = wheel.angle()
    time = watch.time()

    # Each time you use the log() method, a new line with data is added to
    # the file. You can add as many values as you like.
    # In this example, we save the current time and motor angle:
    data.log(time, angle)

    # Wait some time so the motor can move a bit
    wait(100)

# You can now upload your file to your computer
```

In this example, the generated file has the following contents:

```
time, angle
3, 0
108, 6
212, 30
316, 71
419, 124
523, 176
628, 228
734, 281
838, 333
942, 385
```

When you upload the file to your computer as shown above, you can open it in a spreadsheet editor. You can then generate a graph of the data, as shown in [Figure 7.1](#).

In this example, we see that the motor angle changes slowly at first. Then the angle begins to change faster, and the graph becomes a straight line. This means that the motor has reached a constant speed. You can verify that the angle increases by 500 degrees per second.

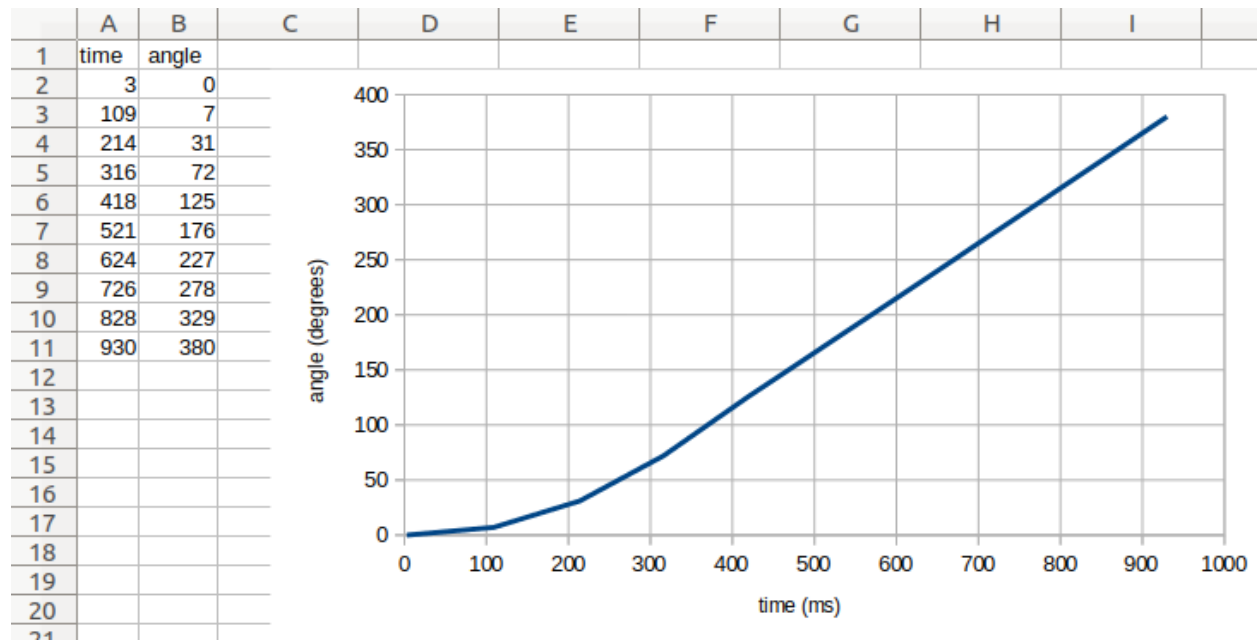


Figure 7.1: Original file contents (left) and a generated graph (right).

## Using the optional arguments

This example shows how to log data beyond just numbers. It also shows how you can use the optional arguments of the DataLog class to choose the file name and extension.

In this example, `timestamp=False`, which means that the date and time are not added to the file name. This can be convenient because the file name will always be the same. However, this means that the contents of `my_file.txt` will be overwritten every time you run this script.

```
#!/usr/bin/env pybricks-micropython
from pybricks.parameters import Color
from pybricks.tools import DataLog

# Create a data log file called my_file.txt
data = DataLog('time', 'angle', name='my_file', timestamp=False, extension='txt')

# The log method uses the print() method to add a line of text.
# So, you can do much more than saving numbers. For example:
data.log('Temperature', 25)
data.log('Sunday', 'Monday', 'Tuesday')
data.log({'Kiwi': Color.GREEN}, {'Banana': Color.YELLOW})

# You can upload the file to your computer, but you can also print the data:
print(data)
```

## 7.2 Timing

**wait** (*time*)

Pauses the user program for a specified amount of time.

**Parameters** **time** (*time: ms*) – How long to wait.

**class Stopwatch**

A stopwatch to measure time intervals. Similar to the stopwatch feature on your phone.

**time** ()

Gets the current time of the stopwatch.

**Returns** Elapsed time.

**Return type** *time: ms*

**pause** ()

Pauses the stopwatch.

**resume** ()

Resumes the stopwatch.

**reset** ()

Resets the stopwatch time to 0.

The run state is unaffected:

- If it was paused, it stays paused (but now at 0).
- If it was running, it stays running (but starting again from 0).

## ROBOTICS – ROBOTICS

Robotics module for the Pybricks API.

**class DriveBase** (*left\_motor*, *right\_motor*, *wheel\_diameter*, *axle\_track*)

A robotic vehicle with two powered wheels and an optional support wheel or caster.

By specifying the dimensions of your robot, this class makes it easy to drive a given distance in millimeters or turn by a given number of degrees.

**Positive** distances and drive speeds mean driving **forward**. **Negative** means **backward**.

**Positive** angles and turn rates mean turning **right**. **Negative** means **left**. So when viewed from the top, positive means clockwise and negative means counterclockwise.

### Parameters

- **left\_motor** (*Motor*) – The motor that drives the left wheel.
- **right\_motor** (*Motor*) – The motor that drives the right wheel.
- **wheel\_diameter** (*dimension: mm*) – Diameter of the wheels.
- **axle\_track** (*dimension: mm*) – Distance between the points where both wheels touch the ground.

### Driving for a given distance or by an angle

Use the following commands to drive a given distance, or turn by a given angle.

This is measured using the internal rotation sensors. Because wheels may slip while moving, the traveled distance and angle are only estimates.

**straight** (*distance*)

Drives straight for a given distance and then stops.

**Parameters** **distance** (*distance: mm*) – Distance to travel.

**turn** (*angle*)

Turns in place by a given angle and then stops.

**Parameters** **angle** (*angle: deg*) – Angle of the turn.

**settings** (*straight\_speed*, *straight\_acceleration*, *turn\_rate*, *turn\_acceleration*)

Configures the speed and acceleration used by *straight()* and *turn()*.

If you give no arguments, this returns the current values as a tuple.

You can only change the settings while the robot is stopped. This is either before you begin driving or after you call *stop()*.

### Parameters

- **straight\_speed** (*speed: mm/s*) – Speed of the robot during `straight()`.
- **straight\_acceleration** (*linear acceleration: mm/s/s*) – Acceleration and deceleration of the robot at the start and end of `straight()`.
- **turn\_rate** (*rotational speed: deg/s*) – Turn rate of the robot during `turn()`.
- **turn\_acceleration** (*rotational acceleration: deg/s/s*) – Angular acceleration and deceleration of the robot at the start and end of `turn()`.

## Drive forever

Use `drive()` to begin driving at a desired speed and steering.

It keeps going until you use `stop()` or change course by using `drive()` again. For example, you can drive until a sensor is triggered and then stop or turn around.

**drive** (*speed, turn\_rate*)

Starts driving at the specified speed and turn rate. Both values are measured at the center point between the wheels of the robot.

### Parameters

- **speed** (*speed: mm/s*) – Speed of the robot.
- **turn\_rate** (*rotational speed: deg/s*) – Turn rate of the robot.

**stop** ()

Stops the robot by letting the motors spin freely.

## Measuring

**distance** ()

Gets the estimated driven distance.

**Returns** Driven distance since last reset.

**Return type** *distance: mm*

**angle** ()

Gets the estimated rotation angle of the drive base.

**Returns** Accumulated angle since last reset.

**Return type** *angle: deg*

**state** ()

Gets the state of the robot.

This returns the current `distance()`, the drive speed, the `angle()`, and the turn rate.

**Returns** Distance, drive speed, angle, turn rate

**Return type** (*distance: mm, speed: mm/s, angle: deg, rotational speed: deg/s*)

**reset** ()

Resets the estimated driven distance and angle to 0.

## Measuring and validating the robot dimensions

As a first estimate, you can measure the `wheel_diameter` and the `axle_track` with a ruler. Because it is hard to see where the wheels effectively touch the ground, you can estimate the `axle_track` as the distance between the midpoint of the wheels.

In practice, most wheels compress slightly under the weight of your robot. To verify, make your robot drive 1000 mm using `my_robot.straight(1000)` and measure how far it really traveled. Compensate as follows:

- If your robot drives **not far enough**, **decrease** the `wheel_diameter` value slightly.
- If your robot drives **too far**, **increase** the `wheel_diameter` value slightly.

Motor shafts and axles bend slightly under the load of the robot, causing the ground contact point of the wheels to be closer to the midpoint of your robot. To verify, make your robot turn 360 degrees using `my_robot.turn(360)` and check that it is back in the same place:

- If your robot turns **not far enough**, **increase** the `axle_track` value slightly.
- If your robot turns **too far**, **decrease** the `axle_track` value slightly.

When making these adjustments, always adjust the `wheel_diameter` first, as done above. Be sure to test both turning and driving straight after you are done.

## Using the DriveBase motors individually

Suppose you make a *DriveBase* object using two *Motor* objects called `left_motor` and `right_motor`. You **cannot** use these motors individually while the *DriveBase* is **active**.

The *DriveBase* is active if it is driving, but also when it is actively holding the wheels in place after a `straight()` or `turn()` command. To deactivate the *DriveBase*, call `stop()`.

## Advanced Settings

The `settings()` method is used to adjust commonly used settings like the default speed and acceleration for straight maneuvers and turns. Use the following attributes to adjust more advanced control settings.

You can only change the settings while the robot is stopped. This is either before you begin driving or after you call `stop()`.

### **distance\_control**

The traveled distance and drive speed are controlled by a PID controller. You can use this attribute to change its settings. See *The Control Class* for an overview of available methods.

### **heading\_control**

The robot turn angle and turn rate are controlled by a PID controller. You can use this attribute to change its settings. See *The Control Class* for an overview of available methods.

## MEDIA – SOUNDS AND IMAGES

This module describes media such as sound and images that you can use in your projects. Media are divided into submodules that indicate on which platform they are available.

### 9.1 `media.ev3dev` – Sounds and Images

EV3 MicroPython is built on top of ev3dev, which comes with a variety of image and sound files. You can access them using the classes below.

You can also use your own sound and image files by placing them in your project folder.

#### 9.1.1 Image Files

**class** `ImageFile`  
Paths to standard EV3 images.

##### Information

**ACCEPT**



**BACKWARD**



**DECLINE**



**FORWARD**



LEFT



NO\_GO



QUESTION\_MARK



RIGHT



STOP\_1



STOP\_2



THUMBS\_DOWN



THUMBS\_UP



WARNING



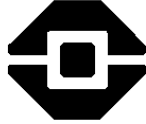


## LEGO

EV3

**EV3**

EV3\_ICON



## Objects

TARGET



## Eyes

ANGRY



AWAKE



BOTTOM\_LEFT



BOTTOM\_RIGHT



CRAZY\_1



CRAZY\_2



DIZZY



DOWN



EVIL



KNOCKED\_OUT



MIDDLE\_LEFT



MIDDLE\_RIGHT



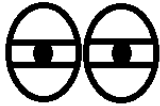
NEUTRAL



PINCHED\_LEFT



PINCHED\_MIDDLE



PINCHED\_RIGHT



SLEEPING



TIRED\_LEFT



TIRED\_MIDDLE



TIRED\_RIGHT



UP



WINKING



## 9.1.2 Sound Files

**class SoundFile**

Paths to standard EV3 sounds.

### Expressions

**BOING**

[Download](#)

**BOO**

[Download](#)

**CHEERING**

[Download](#)

**CRUNCHING**

[Download](#)

**CRYING**

[Download](#)

**FANFARE**

[Download](#)

**KUNG\_FU**

[Download](#)

**LAUGHING\_1**

[Download](#)

**LAUGHING\_2**

[Download](#)

**MAGIC\_WAND**

[Download](#)

**OUCH**

[Download](#)

**SHOUTING**

[Download](#)

**SMACK**

[Download](#)

**SNEEZING**

[Download](#)

**SNORING**

[Download](#)

**UH\_OH**

[Download](#)

**Information****ACTIVATE**

[Download](#)

**ANALYZE**

[Download](#)

**BACKWARDS**

[Download](#)

**COLOR**

[Download](#)

**DETECTED**

[Download](#)

**DOWN**

[Download](#)

**ERROR**

[Download](#)

**ERROR\_ALARM**

Download

### **FLASHING**

Download

### **FORWARD**

Download

### **LEFT**

Download

### **OBJECT**

Download

### **RIGHT**

Download

### **SEARCHING**

Download

### **START**

Download

### **STOP**

Download

### **TOUCH**

Download

### **TURN**

Download

### **UP**

Download

## Communication

### **BRAVO**

[Download](#)

### **EV3**

[Download](#)

### **FANTASTIC**

[Download](#)

### **GAME\_OVER**

[Download](#)

### **GO**

[Download](#)

### **GOOD\_JOB**

[Download](#)

### **GOOD**

[Download](#)

### **GOODBYE**

[Download](#)

### **HELLO**

[Download](#)

### **HI**

[Download](#)

### **LEGO**

[Download](#)

### **MINDSTORMS**

[Download](#)

### **MORNING**

[Download](#)

**NO**[Download](#)**OKAY**[Download](#)**OKEY\_DOKEY**[Download](#)**SORRY**[Download](#)**THANK\_YOU**[Download](#)**YES**[Download](#)**Movement sounds****SPEED\_DOWN**[Download](#)**SPEED\_IDLE**[Download](#)**SPEED\_UP**[Download](#)**Colors****BLACK**[Download](#)**BLUE**[Download](#)**BROWN**[Download](#)



**GREEN**

[Download](#)

**RED**

[Download](#)

**WHITE**

[Download](#)

**YELLOW**

[Download](#)

**Mechanical****AIR\_RELEASE**

[Download](#)

**AIRBRAKE**

[Download](#)

**BACKING\_ALERT**

[Download](#)

**HORN\_1**

[Download](#)

**HORN\_2**

[Download](#)

**LASER**

[Download](#)

**MOTOR\_IDLE**

[Download](#)

**MOTOR\_START**

[Download](#)

**MOTOR\_STOP**

[Download](#)

### **RATCHET**

[Download](#)

### **SONAR**

[Download](#)

### **TICK\_TACK**

[Download](#)

## **Animal sounds**

### **CAT\_PURR**

[Download](#)

### **DOG\_BARK\_1**

[Download](#)

### **DOG\_BARK\_2**

[Download](#)

### **DOG\_GROWL**

[Download](#)

### **DOG\_SNIFF**

[Download](#)

### **DOG\_WHINE**

[Download](#)

### **ELEPHANT\_CALL**

[Download](#)

### **INSECT\_BUZZ\_1**

[Download](#)

### **INSECT\_BUZZ\_2**

[Download](#)

**INSECT\_CHIRP**[Download](#)**SNAKE\_HISS**[Download](#)**SNAKE\_RATTLE**[Download](#)**T\_REX\_ROAR**[Download](#)**Numbers****ZERO**[Download](#)**ONE**[Download](#)**TWO**[Download](#)**THREE**[Download](#)**FOUR**[Download](#)**FIVE**[Download](#)**SIX**[Download](#)**SEVEN**[Download](#)**EIGHT**

Download

**NINE**

Download

**TEN**

Download

## System sounds

**CLICK**

Download

**CONFIRM**

Download

**GENERAL\_ALERT**

Download

**OVERPOWER**

Download

**READY**

Download

### 9.1.3 Fonts

**class Font** (*family=None, size=12, bold=False, monospace=False, lang=None, script=None*)

Object that represents a font for writing text.

The font object will be a font that is the “best” match based on the parameters given and available fonts installed.

#### Parameters

- **family** (*str*) – The preferred font family or `None` to use the default value.
- **size** (*int*) – The preferred font size. Most fonts have sizes between 6 and 24. This is the “point” size and not the same as *height*.
- **bold** (*bool*) – When `True`, prefer bold fonts.
- **monospace** (*bool*) – When `True` prefer monospaced fonts. This is useful for aligning multiple rows of text.
- **lang** (*str*) – A language code, such as `'en'` or `'zh-cn'` or `None` to use the default language.<sup>1</sup>

<sup>1</sup> Note: Languages depend on installed fonts. Additional language codes are possible and some listed language codes may not have a satisfactory font.

- **script** (*str*) – A unicode script identifier such as 'Runr' or None.

**DEFAULT = Font('Lucida', 12)**

The default font.

**family**

Gets the family name of the font.

---

- 'aa': Afar
- 'af': Afrikaans
- 'an': Aragonese
- 'av': Avaric
- 'ay': Aymara
- 'az-az': Azerbaijani
- 'be': Belarusian
- 'bg': Bulgarian
- 'bi': Bislama
- 'bm': Bambara
- 'br': Breton
- 'bs': Bosnian
- 'bua': Buriat
- 'ca': Catalan
- 'ce': Chechen
- 'ch': Chamorro
- 'co': Corsican
- 'crh': Crimean
- 'cs': Czech
- 'csb': Kashubian
- 'cy': Welsh
- 'da': Danish
- 'de': German
- 'ee': Ewe
- 'el': Greek
- 'en': English
- 'eo': Esperanto
- 'es': Spanish
- 'et': Estonian
- 'eu': Basque
- 'ff': Fulah
- 'fi': Finnish
- 'fil': Filipino
- 'fj': Fijian
- 'fo': Faroese
- 'fr': French
- 'fur': Friulian

---

**style**

Gets a string describing the font style.

Can be “Regular” or “Bold”.

**width**

Gets the width of the widest character of the font.

- 
- 'fy': Western Frisian
  - 'ga': Irish
  - 'gd': Gaelic
  - 'gl': Galician
  - 'gv': Manx
  - 'ha': Hausa
  - 'haw': Hawaiian
  - 'he': Hebrew
  - 'ho': Hiri Motu
  - 'hr': Croatian
  - 'hsb': Upper Sorbian
  - 'ht': Haitian
  - 'hu': Hungarian
  - 'ia': Interlingua
  - 'id': Indonesian
  - 'ie': Interlingue
  - 'ik': Inupiaq
  - 'io': Ido
  - 'is': Icelandic
  - 'it': Italian
  - 'ja': Japanese
  - 'jv': Javanese
  - 'ki': Kikuyu
  - 'kj': Kuanyama
  - 'kl': Kalaallisut
  - 'ko': Korean
  - 'ku-tr': Kurdish
  - 'kum': Kumyk
  - 'kw': Cornish
  - 'kwm': Kwambi
  - 'la': Latin
  - 'lb': Luxembourgish
  - 'lez': Lezghian
  - 'lg': Ganda
  - 'li': Limburgan
  - 'ln': Lingala
  - 'lt': Lithuanian

**height**

Gets the height of the font.

**text\_width** (*text*)

Gets the width of the text when the text is drawn using this font.

**Parameters** **text** (*str*) – The text.

- 
- 'lv': Latvian
  - 'mg': Malagasy
  - 'mh': Marshallese
  - 'mi': Maori
  - 'mk': Macedonian
  - 'mn-mn': Mongolian
  - 'mo': Moldavian
  - 'ms': Malay
  - 'mt': Maltese
  - 'na': Nauru
  - 'nb': Norwegian Bokmål
  - 'nds': Low German
  - 'ng': Ndonga
  - 'nl': Dutch
  - 'nn': Norwegian Nynorsk
  - 'no': Norwegian
  - 'nr': South Ndebele
  - 'nso': Northern Sotho
  - 'nv': Navajo
  - 'ny': Chichewa
  - 'oc': Occitan
  - 'om': Oromo
  - 'os': Ossetian
  - 'pap-an': Papiamentu, Netherlands Antilles
  - 'pap-aw': Papiamentu, Aruba
  - 'pl': Polish
  - 'pt': Portuguese
  - 'qu': Quechua
  - 'quz': Cusco Quechua
  - 'rm': Romansh
  - 'rn': Rundi
  - 'ro': Romanian
  - 'ru': Russian
  - 'rw': Kinyarwanda
  - 'sc': Sardinian
  - 'sco': Scots
  - 'se': Northern Sami
-

**Returns** The width in pixels.

**Return type** int

**text\_height** (*text*)

Gets the height of the text when the text is drawn using this font.

**Parameters** **text** (*str*) – The text.

- 
- 'sel': Selkup
  - 'sg': Sango
  - 'sk': Slovak
  - 'sl': Slovenian
  - 'sm': Samoan
  - 'sma': Southern Sami
  - 'smj': Lule Sami
  - 'smn': Inari Sami
  - 'sms': Skolt Sami
  - 'sn': Shona
  - 'so': Somali
  - 'sq': Albanian
  - 'sr': Serbian
  - 'ss': Swati
  - 'st': Southern Sotho
  - 'su': Sundanese
  - 'sv': Swedish
  - 'sw': Swahili
  - 'tk': Turkmen
  - 'tl': Tagalog
  - 'tn': Tswana
  - 'to': Tonga
  - 'tr': Turkish
  - 'ts': Tsonga
  - 'ty': Tahitian
  - 'uk': Ukrainian
  - 'uz': Uzbek
  - 'vo': Volapük
  - 'vot': Votic
  - 'wa': Walloon
  - 'wen': Sorbian
  - 'wo': Wolof
  - 'xh': Xhosa
  - 'yap': Yapese
  - 'yi': Yiddish
  - 'za': Zhuang
  - 'zh-cn': Chinese, China
-



**Returns** The height in pixels.

**Return type** int

### Exploring more fonts

Behind the scenes, Pybricks uses `Fontconfig` for fonts. The `Fontconfig` command line tools can be used to explore available fonts in more detail. To do so, go to the `ev3dev` device browser, right click on your EV3 brick, and click *Open SSH Terminal*. Then you can enter one of these commands:

```
# List available font families.
fc-list :scalable=false family
# Perform lookup similar to Font.DEFAULT
fc-match :scalable=false:dpi=119:family=Lucida:size=12
# Perform lookup similar to Font(size=24,lang=zh-cn)
fc-match :scalable=false:dpi=119:size=24:lang=zh-cn
```

Pybricks only allows the use of bitmap fonts (`scalable=false`) and the screen on the EV3 has 119 pixels per inch (`dpi=119`).

## 9.1.4 Image Manipulation

Instead of drawing directly on the EV3 screen, you can make and interact with image files using the `Image` class given below.

**class** `Image` (*source*, *sub=False*)

Object representing a graphics image. This can either be an in-memory copy of an image or the image displayed on a screen.

### Parameters

- **source** (*str* or *Image*) – The source of the image.  
If *source* is a string, then the image will be loaded from the file path given by the string. Only `.png` files are supported. As a special case, if the string is `_screen_`, the image will be configured to draw directly on the screen.  
If an *Image* is given, the new object will contain a copy of the *source* image object.
- **sub** (*bool*) – If *sub* is `True`, then the image object will act as a sub-image of the *source* image (this only works if the type of *source* is *Image* and not when it is a *str*).  
Additional keyword arguments *x1*, *y1*, *x2*, *y2* are needed when *sub=True*. These specify the top-left and bottom-right coordinates in the *source* image that will be used as the bounds for the sub-image.

**static empty** (*width=<screen width>*, *height=<screen height>*)

Creates a new empty *Image* object.

### Parameters

- **width** (*int*) – The width of the image in pixels.
- **height** (*int*) – The height of the image in pixels.

**Returns** A new image with all pixels set to `Color.WHITE`.

- 
- `'zh-sg'`: Chinese, Singapore
  - `'zh-tw'`: Chinese, Taiwan
  - `'zu'`: Zulu

Return type *Image*

Raises

- **TypeError** – width or height is not a number.
- **ValueError** – width or height is less than 1.
- **RuntimeError** – There was a problem allocating a new image.

## Drawing text

There are two ways to draw text on images. `draw_text()` lets text be placed precisely on the image or `print()` can be used to automatically print text on a new line.

**draw\_text** (*x*, *y*, *text*, *text\_color*=*Color*(*h*=0, *s*=0, *v*=10), *background\_color*=*None*)

Draws text on this image.

The most recent font set using `set_font()` will be used or `Font.DEFAULT` if no font has been set yet.

Parameters

- **x** (*int*) – The x-axis value where the left side of the text will start.
- **y** (*int*) – The y-axis value where the top of the text will start.
- **text** (*str*) – The text to draw.
- **text\_color** (*Color*) – The color used for drawing the text.
- **background\_color** (*Color*) – The color used to fill the rectangle behind the text or *None* for transparent background.

**print** (*\*args*, *sep*=' ', *end*='\n')

Prints a line of text on this image.

This method works like the builtin `print()` function, but it writes on this image instead.

You can set the font using `set_font()`. If no font has been set, `Font.DEFAULT` will be used. The text is always printed using black text with a white background.

Unlike the builtin `print()`, the text does not wrap if it is too wide to fit on this image. It just gets cut off. But if the text would go off of the bottom of this image, the entire image is scrolled up and the text is printed in the new blank area at the bottom of this image.

Parameters

- **\*** (*object*) – Zero or more objects to print.
- **sep** (*str*) – Separator that will be placed between each object that is printed.
- **end** (*str*) – End of line that will be printed after the last object.

**set\_font** (*font*)

Sets the font used for writing on this image.

The font is used for both `draw_text()` and `print()`.

Parameters **font** (*Font*) – The font to use.

## Drawing images

A copy of another image can be drawn on an image. Also consider using sub-images to copy part of an image.

**draw\_image** (*x*, *y*, *source*, *transparent=None*)

Draws the *source* image on this image.

### Parameters

- **x** (*int*) – The x-axis value where the left side of the image will start.
- **y** (*int*) – The y-axis value where the top of the image will start.
- **source** (*Image* or *str*) – The source *Image*. If the argument is a string, then the *source* image is loaded from file.
- **transparent** (*Color*) – The color of *image* to treat as transparent or *None* for no transparency.

## Drawing shapes

These are the methods to draw basic shapes, including points, lines, rectangles and circles.

**draw\_pixel** (*x*, *y*, *color=Color(h=0, s=0, v=10)*)

Draws a single pixel on this image.

### Parameters

- **x** (*int*) – The x coordinate of the pixel.
- **y** (*int*) – The y coordinate of the pixel.
- **color** (*Color*) – The color of the pixel.

**draw\_line** (*x1*, *y1*, *x2*, *y2*, *width=1*, *color=Color(h=0, s=0, v=10)*)

Draws a line on this image.

### Parameters

- **x1** (*int*) – The x coordinate of the starting point of the line.
- **y1** (*int*) – The y coordinate of the starting point of the line.
- **x2** (*int*) – The x coordinate of the ending point of the line.
- **y2** (*int*) – The y coordinate of the ending point of the line.
- **width** (*int*) – The width of the line in pixels.
- **color** (*Color*) – The color of the line.

**draw\_box** (*x1*, *y1*, *x2*, *y2*, *r=0*, *fill=False*, *color=Color(h=0, s=0, v=10)*)

Draws a box on this image.

### Parameters

- **x1** (*int*) – The x coordinate of the left side of the box.
- **y1** (*int*) – The y coordinate of the top of the box.
- **x2** (*int*) – The x coordinate of the right side of the box.
- **y2** (*int*) – The y coordinate of the bottom of the box.
- **r** (*int*) – The radius of the corners of the box.

- **fill** (*bool*) – If `True`, the box will be filled with `color`, otherwise only the outline of the box will be drawn.
- **color** (*Color*) – The color of the box.

**draw\_circle** (*x*, *y*, *r*, *fill=False*, *color=Color(h=0, s=0, v=10)*)

Draws a circle on this image.

#### Parameters

- **x** (*int*) – The x coordinate of the center of the circle.
- **y** (*int*) – The y coordinate of the center of the circle.
- **r** (*int*) – The radius of the circle.
- **fill** (*bool*) – If `True`, the circle will be filled with `color`, otherwise only the circumference will be drawn.
- **color** (*Color*) – The color of the circle.

## Image properties

### **width**

Gets the width of this image in pixels.

### **height**

Gets the height of this image in pixels.

## Replacing the entire image

### **clear** ()

Clears this image. All pixels on this image will be set to `Color.WHITE`.

### **load\_image** (*source*)

Clears this image, then draws the `source` image centered in this image.

**Parameters** **source** (*Image* or *str*) – The source *Image*. If the argument is a string, then the `source` image is loaded from file.

## Saving the image

### **save** (*filename*)

Saves this image as a `.png` file.

**Parameters** **filename** (*str*) – The path to the file to be saved.

#### **Raises**

- **TypeError** – `filename` is not a string.
- **OSError** – There was a problem saving the file.

## Available languages for fonts

## MESSAGING – MESSAGING

An EV3 Brick can send information to another EV3 Brick using Bluetooth. This page shows you how to connect multiple bricks and how to write scripts to send messages between them.

### 10.1 Pairing two EV3 Bricks

Before two EV3 bricks can exchange messages, they must be *paired*. You'll need to do this only the first time. First, activate bluetooth on all EV3 bricks as shown in Figure 10.1.

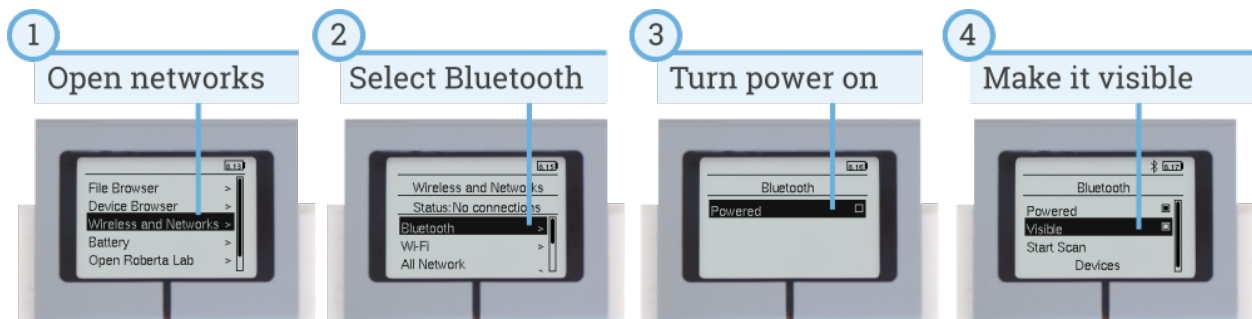


Figure 10.1: Turn on Bluetooth and make Bluetooth visible.

Now you can make one EV3 Brick search for the other and pair with it, as shown in Figure 10.2.

Once they are paired, do *not* click *connect* in the menu that appears. The connection will be made when you run your programs, as described below.

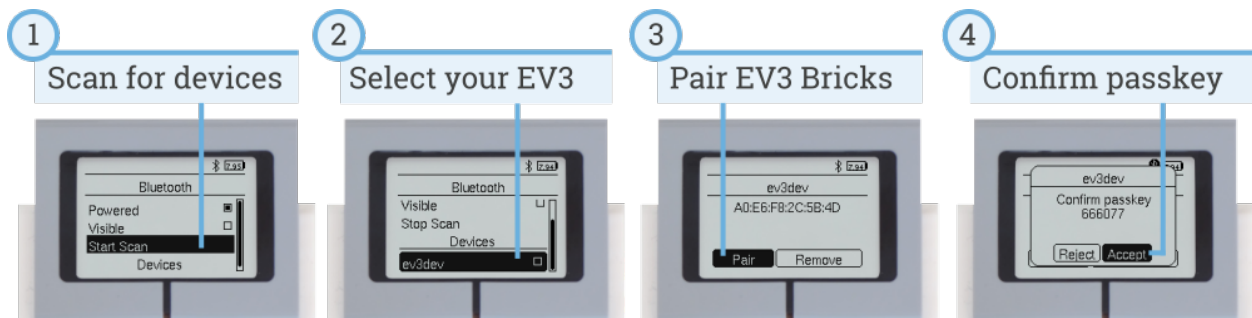


Figure 10.2: Pairing one EV3 Brick to another EV3 Brick.

When you scan for Bluetooth devices, you'll see a list of device names. By default, all EV3 Bricks are named *ev3dev*. Click [here](#) to learn how to change that name. This makes it easy to tell them apart.

Repeat the steps in [Figure 10.2](#) if you want to pair more than two EV3 Bricks.

## 10.2 Server and Client

A wireless network consists of EV3 Bricks acting as servers or clients. An example with one server and one client is shown in [Figure 10.3](#). Messages can be sent in both ways: the server can send a message to the client, and the client can send a message to the server.

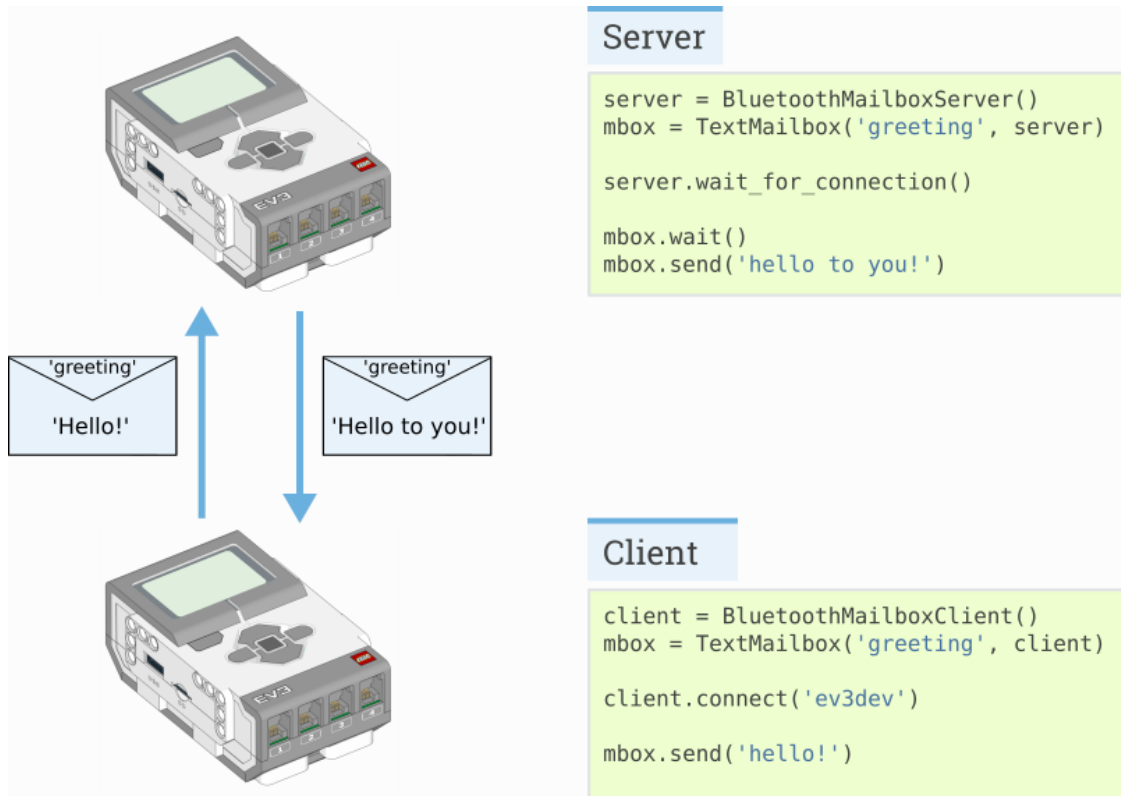


Figure 10.3: An example network with one server and one clients.

The only difference between the client and the server is which one initiates the connection at the beginning of the program:

- The **server** must always be started first. It uses the `BluetoothMailboxServer` class. Then it waits for clients using the `wait_for_connection` method.
- The **client** uses the `BluetoothMailboxClient` class. It connects to the server using the `connect` method.
- After that, sending and receiving messages is done in the same way on both EV3 Bricks.

### **class** `BluetoothMailboxServer`

Object that represents a Bluetooth connection from one or more remote EV3s.

The remote EV3s can either be running MicroPython or the standard EV3 firmware.

A “server” waits for a “client” to connect to it.

#### **wait\_for\_connection** (*count=1*)

Waits for a `BluetoothMailboxClient` on a remote device to connect.

**Parameters** `count` (*int*) – The number of remote connections to wait for.

**Raises** `OSError` – There was a problem establishing the connection.

`close()`

Closes all connections.

**class** `BluetoothMailboxClient`

Object that represents a Bluetooth connection to one or more remote EV3s.

The remote EV3s can either be running MicroPython or the standard EV3 firmware.

A “client” initiates a connection to a waiting “server”.

**connect** (*brick*)

Connects to an `BluetoothMailboxServer` on another device.

The remote device must be paired and waiting for a connection. See `BluetoothMailboxServer.wait_for_connection()`.

**Parameters** `brick` (*str*) – The name or Bluetooth address of the remote EV3 to connect to.

**Raises** `OSError` – There was a problem establishing the connection.

**server\_close** ()

Closes all connections.

## 10.3 Mailboxes

Mailboxes are used to send data to and from other EV3 Bricks.

A Mailbox has a `name`, similar to the “subject” of an email. If two EV3 Bricks have a Mailbox with the same name, they can send messages between them. Each EV3 Brick can read its own Mailbox, and send messages to the Mailbox on the other EV3 Brick.

Depending on the type of messages you would like to exchange (bytes, booleans, numbers, or text), you can choose one of the Mailboxes below.

**class** `Mailbox` (*name, connection, encode=None, decode=None*)

Object that represents a mailbox containing data.

You can read data that is delivered by other EV3 bricks, or send data to other bricks that have the same mailbox.

By default, the mailbox reads and send only bytes. To send other data, you can provide an `encode` function that encodes your Python object into bytes, and a `decode` function to convert bytes back to a Python object.

### Parameters

- **name** (*str*) – The name of this mailbox.
- **connection** – A connection object such as `BluetoothMailboxClient`.
- **encode** (*callable*) – Function that encodes a Python object to bytes.
- **decode** (*callable*) – Function that creates a new Python object from bytes.

**read** ()

Gets the current value of the mailbox.

**Returns** The current value or `None` if the mailbox is empty.

**send** (*value, brick=None*)

Sends a value to this mailbox on connected devices.



**Parameters**

- **value** – The value that will be delivered to the mailbox.
- **brick** (*str*) – The name or Bluetooth address of the brick or `None` to broadcast to all connected devices.

**Raises** `OSError` – There is a problem with the connection.

**wait** ()

Waits for the mailbox to be updated by remote device.

**wait\_new** ()

Waits for a new value to be delivered to the mailbox that is not equal to the current value in the mailbox.

**Returns** The new value.

**class LogicMailbox** (*name, connection*)

Object that represents a mailbox containing boolean data.

This works just like a regular *Mailbox*, but values must be `True` or `False`.

This is compatible with the “logic” mailbox type in EV3-G.

**Parameters**

- **name** (*str*) – The name of this mailbox.
- **connection** – A connection object such as *BluetoothMailboxClient*.

**class NumericMailbox** (*name, connection*)

Object that represents a mailbox containing numeric data.

This works just like a regular *Mailbox*, but values must be a number, such as 15 or 12.345

This is compatible with the “numeric” mailbox type in EV3-G.

**Parameters**

- **name** (*str*) – The name of this mailbox.
- **connection** – A connection object such as *BluetoothMailboxClient*.

**class TextMailbox** (*name, connection*)

Object that represents a mailbox containing text data.

This works just like a regular *Mailbox*, but data must be a string, such as 'hello!' or 'My name is EV3'.

This is compatible with the “text” mailbox type in EV3-G.

**Parameters**

- **name** (*str*) – The name of this mailbox.
- **connection** – A connection object such as *BluetoothMailboxClient*.

## 10.4 Examples

### EV3 Bluetooth Server

This is the full version of the excerpt shown in [Figure 10.3](#).

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

# Before running this program, make sure the client and server EV3 bricks are
# paired using Bluetooth, but do NOT connect them. The program will take care
# of establishing the connection.

# The server must be started before the client!

from pybricks.messaging import BluetoothMailboxServer, TextMailbox

server = BluetoothMailboxServer()
mbox = TextMailbox('greeting', server)

# The server must be started before the client!
print('waiting for connection...')
server.wait_for_connection()
print('connected!')

# In this program, the server waits for the client to send the first message
# and then sends a reply.
mbox.wait()
print(mbox.read())
mbox.send('hello to you!')
```

### EV3 Bluetooth Client

This is the full version of the excerpt shown in [Figure 10.3](#).

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

# Before running this program, make sure the client and server EV3 bricks are
# paired using Bluetooth, but do NOT connect them. The program will take care
# of establishing the connection.

# The server must be started before the client!

from pybricks.messaging import BluetoothMailboxClient, TextMailbox

# This is the name of the remote EV3 or PC we are connecting to.
SERVER = 'ev3dev'

client = BluetoothMailboxClient()
mbox = TextMailbox('greeting', client)

print('establishing connection...')
client.connect(SERVER)
print('connected!')

# In this program, the client sends the first message and then waits for the
# server to reply.
mbox.send('hello!')
mbox.wait()
```

(continues on next page)

(continued from previous page)

```
print(mbox.read())
```

## 10.5 Making bigger networks

The classes in this module are not limited to just two EV3 Bricks. for example, you can add more clients to your network. An example with pseudo-code is shown in [Figure 10.4](#).

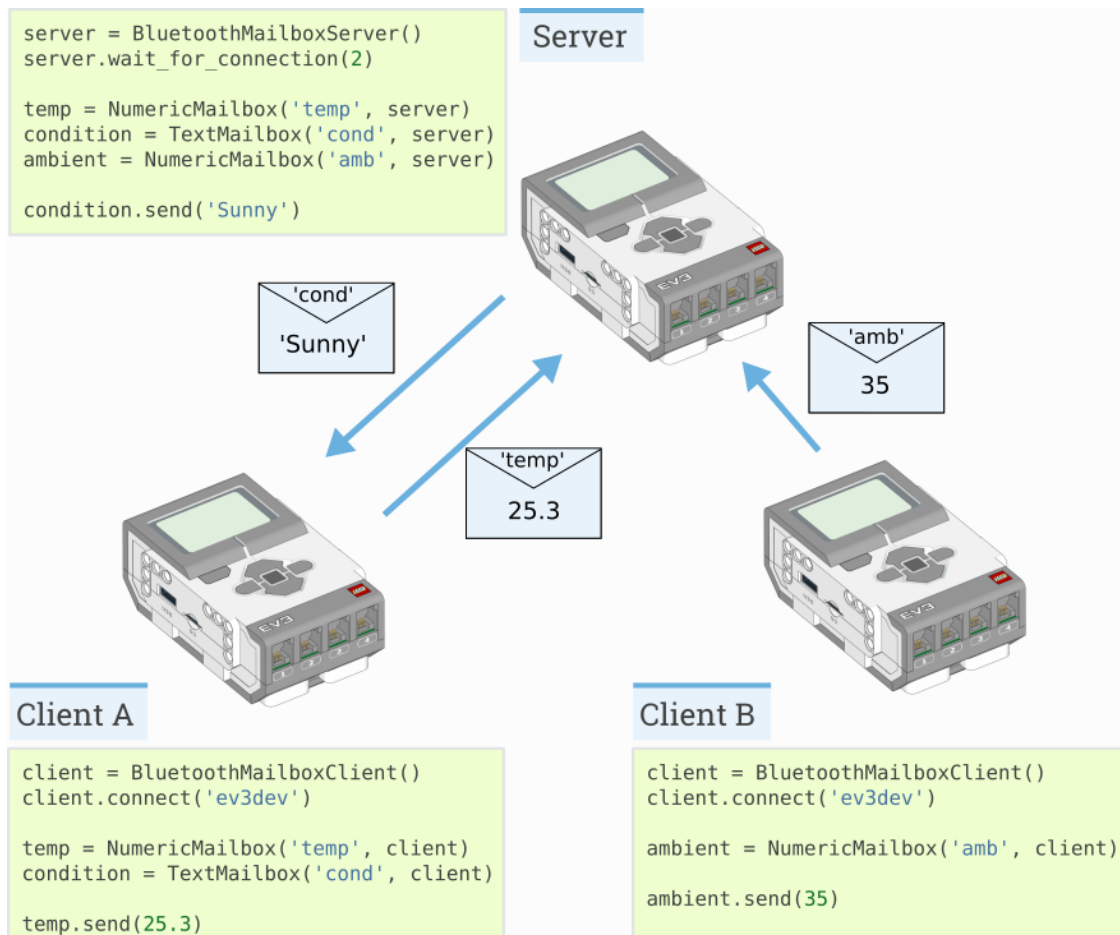


Figure 10.4: An example network with one server and two clients.

## SIGNALS AND UNITS

Many commands allow you to specify arguments in terms of well-known physical quantities. This page gives an overview of each quantity and its unit.

### 11.1 Time

#### 11.1.1 time: ms

All time and duration values are measured in milliseconds (ms).

For example, the duration of motion with `run_time`, and the duration of `wait` are specified in milliseconds.

### 11.2 Angles and angular motion

#### 11.2.1 angle: deg

All angles are measured in degrees (deg). One full rotation corresponds to 360 degrees.

For example, the angle values of a `Motor` or the `GyroSensor` are expressed in degrees.

#### 11.2.2 rotational speed: deg/s

Rotational speed, or *angular velocity* describes how fast something rotates, expressed as the number of degrees per second (deg/s).

For example, the rotational speed values of a `Motor` or the `GyroSensor` are expressed in degrees per second.

While we recommend working with degrees per second in your programs, you can use the following table to convert between commonly used units.

	deg/s	rpm
1 deg/s =	1	1/6=0.167
1 rpm =	6	1

### 11.2.3 rotational acceleration: deg/s/s

Rotational acceleration, or *angular acceleration* describes how fast the rotational speed changes. This is expressed as the change of the number of degrees per second, during one second (deg/s/s). This is also commonly written as  $\text{deg/s}^2$ .

For example, you can adjust the rotational acceleration setting of a `Motor` to change how smoothly or how quickly it reaches the constant speed set point.

## 11.3 Distance and linear motion

### 11.3.1 distance: mm

Distances are expressed in millimeters (mm) whenever possible.

For example, the distance value of the `UltrasonicSensor` is measured in millimeters.

While we recommend working with millimeters in your programs, you can use the following table to convert between commonly used units.

	mm	cm	inch
1 mm =	1	0.1	0.0394
1 cm =	10	1	0.394
1 inch =	25.4	2.54	1

### 11.3.2 dimension: mm

Dimensions are expressed in millimeters (mm), just like distances.

For example, the diameter of a wheel is measured in millimeters.

### 11.3.3 speed: mm/s

Linear speeds are expressed as millimeters per second (mm/s).

For example, the speed of a robotic vehicle is expressed in mm/s.

### 11.3.4 linear acceleration: mm/s/s

Linear acceleration describes how fast the speed changes. This is expressed as the change of the millimeters per second, during one second (deg/s/s). This is also commonly written as  $\text{mm/s}^2$ .

For example, you can adjust the acceleration setting of a `DriveBase` to change how smoothly or how quickly it reaches the constant speed set point.

### 11.3.5 linear acceleration: m/s/s

As above, but expressed in meters per second squared:  $m/s^2$ . This is a more practical unit for large values such as those given by an accelerometer.

## 11.4 Approximate and relative units

### 11.4.1 percentage: %

Some signals do not have specific units. They range from a minimum (0%) to a maximum (100%). Specifics type of percentages are *relative distances* or *brightness*.

Another example is the sound volume, which ranges from 0% (silent) to 100% (loudest).

### 11.4.2 relative distance: %

Some distance measurements do not provide an accurate value with a specific unit, but they range from very close (0%) to very far (100%). These are referred to as relative distances.

For example, the distance value of the `InfraredSensor` is a relative distance.

### 11.4.3 brightness: %

The perceived brightness of a light is expressed as a percentage. It is 0% when the light is off and 100% when the light is fully on. When you choose 50%, this means that the light is perceived as approximately half as bright to the human eye.

## 11.5 Force and torque

### 11.5.1 force: N

Force values are expressed in newtons (N).

While we recommend working with newtons in your programs, you can use the following table to convert to and from other units.

	mN	N	lbf
1 mN =	1	0.001	$2.248 \cdot 10^{-4}$
1 N =	1000	1	0.2248
1 lbf =	4448	4.448	1

## 11.5.2 torque: mNm

Torque values are expressed in millinewtonmeter (mNm) unless stated otherwise.

# 11.6 Electricity

## 11.6.1 voltage: mV

Voltages are expressed in millivolt (mV).

For example, you can check the voltage of the battery.

## 11.6.2 current: mA

Electrical currents are expressed in milliampere (mA).

For example, you can check the current supplied by the battery.

## 11.6.3 energy: J

Stored energy or energy consumption can be expressed in Joules (J).

## 11.6.4 power: mW

Power is the rate at which energy is stored or consumed. It is expressed in milliwatt (mW).

# 11.7 Ambient environment

## 11.7.1 frequency: Hz

Sound frequencies are expressed in Hertz (Hz).

For example, you can choose the frequency of a beep to change the pitch.

## 11.7.2 temperature: °C

Temperature is measured in degrees Celcius (°C). To convert to degrees Fahrenheit (°F) or Kelvin (K), you can use the following conversion formulas:

$$^{\circ}F = ^{\circ}C \cdot \frac{9}{5} + 32.$$

$$K = ^{\circ}C + 273.15.$$

### 11.7.3 hue: deg

Hue of a color (0-359 degrees).

TODO: diagram



## MORE ABOUT MOTORS

### 12.1 The Control Class

The `Motor` class uses PID control to accurately track your commanded target angles. Similarly, the `DriveBase` class uses two of such controllers: one to control the heading and one to control the traveled distance.

You can change the control settings through the following attributes, which are instances of the `Control` class given below.:

- `Motor.control`
- `DriveBase.heading_control`
- `DriveBase.distance_control`

You can only change the settings while the controller is stopped. For example, you can set the settings at the beginning of your program. Alternatively, first call `stop()` to make your `Motor` or `DriveBase` stop, and then change the settings.

#### **class Control**

Class to interact with PID controller and settings.

#### **scale**

Scaling factor between the controlled integer variable and the physical output. For example, for a single motor this is the number of encoder pulses per degree of rotation.

#### **Status**

#### **done()**

Checks if an ongoing command or maneuver is done.

**Returns** `True` if the command is done, `False` if not.

**Return type** `bool`

#### **stalled()**

Checks if the controller is currently stalled.

A controller is stalled when it cannot reach the target speed or position, even with the maximum actuation signal.

**Returns** `True` if the controller is stalled, `False` if not.

**Return type** `bool`

**load()**

Gets the load acting on the Motor or DriveBase.

This value is determined from the feedback torque that is needed to track the speed or position command given by the user.

When coasting, braking, or controlling the duty cycle manually, the load cannot be estimated in this way. Then this method returns zero.

**Returns** The load torque. It returns 0 if control is not active.

**Return type** *torque: mNm*

**Settings****limits** (*speed, acceleration, duty, torque*)

Configures the maximum speed, acceleration, duty, and torque.

If no arguments are given, this will return the current values.

**Parameters**

- **speed** (*rotational speed: deg/s or speed: mm/s*) – Maximum speed. All speed commands will be capped to this value.
- **acceleration** (*rotational acceleration: deg/s/s or linear acceleration: mm/s/s*) – Maximum acceleration.
- **duty** (*percentage: %*) – Maximum duty cycle during control.
- **torque** (*torque: mNm*) – Maximum feedback torque during control.

**pid** (*kp, ki, kd, integral\_range, integral\_rate, feed\_forward*)

Gets or sets the PID values for position and speed control.

If no arguments are given, this will return the current values.

**Parameters**

- **kp** (*int*) – Proportional position control constant. It is the feedback torque per degree of error:  $\mu\text{Nm/deg}$ .
- **ki** (*int*) – Integral position control constant. It is the feedback torque per accumulated degree of error:  $\mu\text{Nm}/(\text{deg s})$ .
- **kd** (*int*) – Derivative position (or proportional speed) control constant. It is the feedback torque per unit of speed:  $\mu\text{Nm}/(\text{deg/s})$ .
- **integral\_range** (*angle: deg or distance: mm*) – Region around the target angle or distance, in which integral control errors are accumulated.
- **integral\_rate** (*rotational speed: deg/s or speed: mm/s*) – Maximum rate at which the error integral is allowed to grow.
- **feed\_forward** (*percentage: %*) – This adds a feed forward signal to the PID feedback signal, in the direction of the speed reference. This value is expressed as a percentage of the absolute maximum duty cycle.

**target\_tolerances** (*speed, position*)

Gets or sets the tolerances that say when a maneuver is done.

If no arguments are given, this will return the current values.

**Parameters**

- **speed** (*rotational speed: deg/s* or *speed: mm/s*) – Allowed deviation from zero speed before motion is considered complete.
- **position** (*angle: deg* or *distance: mm*) – Allowed deviation from the target before motion is considered complete.

**stall\_tolerances** (*speed, time*)

Gets or sets stalling tolerances.

If no arguments are given, this will return the current values.

#### Parameters

- **speed** (*rotational speed: deg/s* or *speed: mm/s*) – If the controller cannot reach this speed for some `time` even with maximum actuation, it is stalled.
- **time** (*time: ms*) – How long the controller has to be below this minimum `speed` before we say it is stalled.

---

## PYTHON MODULE INDEX

### p

- `pybricks.ev3devices`, [56](#)
- `pybricks.hubs`, [2](#)
- `pybricks.iodevices`, [69](#)
- `pybricks.media`, [93](#)
- `pybricks.media.ev3dev`, [93](#)
- `pybricks.messaging`, [116](#)
- `pybricks.nxtdevices`, [63](#)
- `pybricks.parameters`, [81](#)
- `pybricks.pupdevices`, [21](#)
- `pybricks.robotics`, [90](#)
- `pybricks.tools`, [86](#)

## A

A (*Port attribute*), 84  
 active() (*AnalogSensor method*), 75  
 ambient() (*ColorDistanceSensor method*), 36  
 ambient() (*ColorSensor method*), 42, 59, 64  
 ambient() (*LightSensor method*), 64  
 AnalogSensor (*class in pybricks.iodevices*), 75  
 angle() (*DriveBase method*), 91  
 angle() (*GyroSensor method*), 62  
 angle() (*Motor method*), 24, 57  
 animate() (*CityHub.light method*), 5  
 animate() (*MoveHub.light method*), 2  
 animate() (*TechnicHub.light method*), 8

## B

B (*Port attribute*), 84  
 BEACON (*Button attribute*), 81  
 beacon() (*InfraredSensor method*), 60  
 beep() (*EV3Brick.speaker method*), 11  
 BLACK (*Color attribute*), 82  
 blink() (*CityHub.light method*), 4  
 blink() (*MoveHub.light method*), 2  
 blink() (*TechnicHub.light method*), 7  
 BLUE (*Color attribute*), 82  
 BluetoothMailboxClient (*class in pybricks.messaging*), 118  
 BluetoothMailboxServer (*class in pybricks.messaging*), 117  
 BRAKE (*Stop attribute*), 84  
 brake() (*Motor method*), 25, 57  
 Button (*built-in class*), 81  
 buttons() (*InfraredSensor method*), 60

## C

C (*Port attribute*), 84  
 CENTER (*Button attribute*), 81  
 CityHub (*class in pybricks.hubs*), 4  
 clear() (*EV3Brick.screen method*), 14  
 clear() (*Image method*), 114  
 clear() (*UARTDevice method*), 74  
 CLOCKWISE (*Direction attribute*), 84  
 close() (*BluetoothMailboxServer method*), 118

COAST (*Stop attribute*), 84  
 Color (*class in pybricks.parameters*), 81  
 color() (*ColorDistanceSensor method*), 35  
 color() (*ColorSensor method*), 42, 59, 64  
 ColorDistanceSensor (*class in pybricks.pupdevices*), 35  
 ColorSensor (*class in pybricks.ev3devices*), 59  
 ColorSensor (*class in pybricks.nxtdevices*), 64  
 ColorSensor (*class in pybricks.pupdevices*), 42  
 connect() (*BluetoothMailboxClient method*), 118  
 Control (*class in pybricks.\_common*), 127  
 control (*Motor attribute*), 58  
 Control.scale (*in module pybricks.\_common*), 127  
 conversion() (*VernierAdapter method*), 68  
 count() (*InfraredSensor method*), 34  
 COUNTERCLOCKWISE (*Direction attribute*), 84  
 current() (*CityHub.battery method*), 5  
 current() (*EV3Brick.battery method*), 16  
 current() (*MoveHub.battery method*), 3  
 current() (*TechnicHub.battery method*), 8  
 CYAN (*Color attribute*), 82

## D

D (*Port attribute*), 84  
 DataLog (*class in pybricks.tools*), 86  
 dc() (*Motor method*), 26, 58  
 DCMotor (*class in pybricks.pupdevices*), 21  
 DEFAULT (*Font attribute*), 107  
 detectable\_colors() (*ColorDistanceSensor method*), 36  
 detectable\_colors() (*ColorSensor method*), 43  
 Direction (*built-in class*), 84  
 distance() (*ColorDistanceSensor method*), 36  
 distance() (*DriveBase method*), 91  
 distance() (*ForceSensor method*), 49  
 distance() (*InfraredSensor method*), 34, 60  
 distance() (*UltrasonicSensor method*), 47, 61, 65  
 distance\_control (*DriveBase attribute*), 92  
 done() (*Control method*), 127  
 DOWN (*Button attribute*), 81  
 draw\_box() (*EV3Brick.screen method*), 15  
 draw\_box() (*Image method*), 113

[draw\\_circle\(\)](#) (*EV3Brick.screen method*), 16  
[draw\\_circle\(\)](#) (*Image method*), 114  
[draw\\_image\(\)](#) (*EV3Brick.screen method*), 15  
[draw\\_image\(\)](#) (*Image method*), 113  
[draw\\_line\(\)](#) (*EV3Brick.screen method*), 15  
[draw\\_line\(\)](#) (*Image method*), 113  
[draw\\_pixel\(\)](#) (*EV3Brick.screen method*), 15  
[draw\\_pixel\(\)](#) (*Image method*), 113  
[draw\\_text\(\)](#) (*EV3Brick.screen method*), 14  
[draw\\_text\(\)](#) (*Image method*), 112  
[drive\(\)](#) (*DriveBase method*), 91  
[DriveBase](#) (*class in pybricks.robotics*), 90

## E

[E](#) (*Port attribute*), 84  
[empty\(\)](#) (*Image static method*), 111  
[EnergyMeter](#) (*class in pybricks.nxtdevices*), 67  
[EV3Brick](#) (*class in pybricks.hubs*), 10  
[Ev3devSensor](#) (*class in pybricks.iodevices*), 77

## F

[F](#) (*Port attribute*), 84  
[family](#) (*Font attribute*), 107  
[Font](#) (*class in pybricks.media.ev3dev*), 106  
[force\(\)](#) (*ForceSensor method*), 49  
[ForceSensor](#) (*class in pybricks.pupdevices*), 49

## G

[GRAY](#) (*Color attribute*), 82  
[GREEN](#) (*Color attribute*), 82  
[GyroSensor](#) (*class in pybricks.ev3devices*), 62

## H

[heading\\_control](#) (*DriveBase attribute*), 92  
[height](#) (*EV3Brick.screen attribute*), 16  
[height](#) (*Font attribute*), 108  
[height](#) (*Image attribute*), 114  
[HOLD](#) (*Stop attribute*), 84  
[hold\(\)](#) (*Motor method*), 25, 57  
[hsv\(\)](#) (*ColorDistanceSensor method*), 36  
[hsv\(\)](#) (*ColorSensor method*), 43

## I

[I2CDevice](#) (*class in pybricks.iodevices*), 71  
[Image](#) (*class in pybricks.media.ev3dev*), 111  
[ImageFile](#) (*class in pybricks.media.ev3dev*), 93  
[ImageFile.ACCEPT](#) (*in module py-bricks.media.ev3dev*), 93  
[ImageFile.ANGRY](#) (*in module py-bricks.media.ev3dev*), 95  
[ImageFile.AWAKE](#) (*in module py-bricks.media.ev3dev*), 95  
[ImageFile.BACKWARD](#) (*in module py-bricks.media.ev3dev*), 93

[ImageFile.BOTTOM\\_LEFT](#) (*in module py-bricks.media.ev3dev*), 95  
[ImageFile.BOTTOM\\_RIGHT](#) (*in module py-bricks.media.ev3dev*), 95  
[ImageFile.CRAZY\\_1](#) (*in module py-bricks.media.ev3dev*), 95  
[ImageFile.CRAZY\\_2](#) (*in module py-bricks.media.ev3dev*), 95  
[ImageFile.DECLINE](#) (*in module py-bricks.media.ev3dev*), 93  
[ImageFile.DIZZY](#) (*in module py-bricks.media.ev3dev*), 96  
[ImageFile.DOWN](#) (*in module pybricks.media.ev3dev*), 96  
[ImageFile.EV3](#) (*in module pybricks.media.ev3dev*), 95  
[ImageFile.EV3\\_ICON](#) (*in module py-bricks.media.ev3dev*), 95  
[ImageFile.EVIL](#) (*in module pybricks.media.ev3dev*), 96  
[ImageFile.FORWARD](#) (*in module py-bricks.media.ev3dev*), 93  
[ImageFile.KNOCKED\\_OUT](#) (*in module py-bricks.media.ev3dev*), 96  
[ImageFile.LEFT](#) (*in module pybricks.media.ev3dev*), 93  
[ImageFile.MIDDLE\\_LEFT](#) (*in module py-bricks.media.ev3dev*), 96  
[ImageFile.MIDDLE\\_RIGHT](#) (*in module py-bricks.media.ev3dev*), 96  
[ImageFile.NEUTRAL](#) (*in module py-bricks.media.ev3dev*), 96  
[ImageFile.NO\\_GO](#) (*in module py-bricks.media.ev3dev*), 94  
[ImageFile.PINCHED\\_LEFT](#) (*in module py-bricks.media.ev3dev*), 96  
[ImageFile.PINCHED\\_MIDDLE](#) (*in module py-bricks.media.ev3dev*), 96  
[ImageFile.PINCHED\\_RIGHT](#) (*in module py-bricks.media.ev3dev*), 97  
[ImageFile.QUESTION\\_MARK](#) (*in module py-bricks.media.ev3dev*), 94  
[ImageFile.RIGHT](#) (*in module py-bricks.media.ev3dev*), 94  
[ImageFile.SLEEPING](#) (*in module py-bricks.media.ev3dev*), 97  
[ImageFile.STOP\\_1](#) (*in module py-bricks.media.ev3dev*), 94  
[ImageFile.STOP\\_2](#) (*in module py-bricks.media.ev3dev*), 94  
[ImageFile.TARGET](#) (*in module py-bricks.media.ev3dev*), 95  
[ImageFile.THUMBS\\_DOWN](#) (*in module py-bricks.media.ev3dev*), 94

ImageFile.THUMBS\_UP (in module *pybricks.media.ev3dev*), 94  
 ImageFile.TIRED\_LEFT (in module *pybricks.media.ev3dev*), 97  
 ImageFile.TIRED\_MIDDLE (in module *pybricks.media.ev3dev*), 97  
 ImageFile.TIRED\_RIGHT (in module *pybricks.media.ev3dev*), 97  
 ImageFile.UP (in module *pybricks.media.ev3dev*), 97  
 ImageFile.WARNING (in module *pybricks.media.ev3dev*), 94  
 ImageFile.WINKING (in module *pybricks.media.ev3dev*), 97  
 info() (*PUPDevice* method), 69  
 InfraredSensor (class in *pybricks.ev3devices*), 60  
 InfraredSensor (class in *pybricks.pupdevices*), 34  
 input() (*EnergyMeter* method), 67  
 intensity() (*SoundSensor* method), 66

## K

keypad() (*InfraredSensor* method), 61

## L

LEFT (*Button* attribute), 81  
 LEFT\_DOWN (*Button* attribute), 81  
 LEFT\_MINUS (*Button* attribute), 81  
 LEFT\_PLUS (*Button* attribute), 81  
 LEFT\_UP (*Button* attribute), 81  
 Light (class in *pybricks.pupdevices*), 51  
 LightSensor (class in *pybricks.nxtdevices*), 64  
 limits() (*Control* method), 128  
 load() (*Control* method), 127  
 load\_image() (*EV3Brick.screen* method), 15  
 load\_image() (*Image* method), 114  
 log() (*DataLog* method), 86  
 LogicMailbox (class in *pybricks.messaging*), 119  
 LUMPDevice (class in *pybricks.iodevices*), 75

## M

MAGENTA (*Color* attribute), 82  
 Mailbox (class in *pybricks.messaging*), 118  
 module  
     *pybricks.ev3devices*, 56  
     *pybricks.hubs*, 2  
     *pybricks.iodevices*, 69  
     *pybricks.media*, 93  
     *pybricks.media.ev3dev*, 93  
     *pybricks.messaging*, 116  
     *pybricks.nxtdevices*, 63  
     *pybricks.parameters*, 81  
     *pybricks.pupdevices*, 21  
     *pybricks.robotics*, 90  
     *pybricks.tools*, 86  
 Motor (class in *pybricks.pupdevices*), 23

MoveHub (class in *pybricks.hubs*), 2

## N

NONE (*Color* attribute), 82  
 NumericMailbox (class in *pybricks.messaging*), 119

## O

off() (*CityHub.light* method), 4  
 off() (*ColorDistanceSensor.light* method), 36  
 off() (*ColorSensor.light* method), 65  
 off() (*ColorSensor.lights* method), 43  
 off() (*EV3Brick.light* method), 11  
 off() (*Light* method), 51  
 off() (*MoveHub.light* method), 2  
 off() (*TechnicHub.light* method), 7  
 off() (*UltrasonicSensor.lights* method), 47  
 on() (*CityHub.light* method), 4  
 on() (*ColorDistanceSensor.light* method), 36  
 on() (*ColorSensor.light* method), 65  
 on() (*ColorSensor.lights* method), 43  
 on() (*EV3Brick.light* method), 11  
 on() (*Light* method), 51  
 on() (*MoveHub.light* method), 2  
 on() (*TechnicHub.light* method), 7  
 on() (*UltrasonicSensor.lights* method), 47  
 ORANGE (*Color* attribute), 82  
 output() (*EnergyMeter* method), 67

## P

passive() (*AnalogSensor* method), 75  
 pause() (*StopWatch* method), 89  
 pid() (*Control* method), 128  
 play\_file() (*EV3Brick.speaker* method), 11  
 play\_notes() (*EV3Brick.speaker* method), 11  
 Port (built-in class), 84  
 port\_index (*Ev3devSensor* attribute), 77  
 presence() (*UltrasonicSensor* method), 47, 61  
 pressed() (*EV3Brick.buttons* method), 10  
 pressed() (*ForceSensor* method), 49  
 pressed() (*TouchSensor* method), 59, 63  
 print() (*EV3Brick.screen* method), 14  
 print() (*Image* method), 112  
 PUPDevice (class in *pybricks.iodevices*), 69  
*pybricks.ev3devices*  
     module, 56  
*pybricks.hubs*  
     module, 2  
*pybricks.iodevices*  
     module, 69  
*pybricks.media*  
     module, 93  
*pybricks.media.ev3dev*  
     module, 93  
*pybricks.messaging*



module, 116  
 pybricks.nxtdevices  
   module, 63  
 pybricks.parameters  
   module, 81  
 pybricks.pupdevices  
   module, 21  
 pybricks.robotics  
   module, 90  
 pybricks.tools  
   module, 86

## R

read() (*Ev3devSensor method*), 77  
 read() (*I2CDevice method*), 71  
 read() (*LUMPDevice method*), 75  
 read() (*Mailbox method*), 118  
 read() (*PUPDevice method*), 69  
 read() (*UARTDevice method*), 73  
 read\_all() (*UARTDevice method*), 74  
 RED (*Color attribute*), 82  
 reflection() (*ColorDistanceSensor method*), 36  
 reflection() (*ColorSensor method*), 42, 60, 65  
 reflection() (*InfraredSensor method*), 34  
 reflection() (*LightSensor method*), 64  
 reset() (*DriveBase method*), 91  
 reset() (*StopWatch method*), 89  
 reset\_angle() (*GyroSensor method*), 62  
 reset\_angle() (*Motor method*), 24, 57  
 resistance() (*AnalogSensor method*), 75  
 resume() (*StopWatch method*), 89  
 rgb() (*ColorSensor method*), 60, 65  
 RIGHT (*Button attribute*), 81  
 RIGHT\_DOWN (*Button attribute*), 81  
 RIGHT\_MINUS (*Button attribute*), 81  
 RIGHT\_PLUS (*Button attribute*), 81  
 RIGHT\_UP (*Button attribute*), 81  
 run() (*Motor method*), 25, 57  
 run\_angle() (*Motor method*), 25, 57  
 run\_target() (*Motor method*), 25, 58  
 run\_time() (*Motor method*), 25, 57  
 run\_until\_stalled() (*Motor method*), 26, 58

## S

S1 (*Port attribute*), 84  
 S2 (*Port attribute*), 84  
 S3 (*Port attribute*), 84  
 S4 (*Port attribute*), 84  
 save() (*EV3Brick.screen method*), 16  
 save() (*Image method*), 114  
 say() (*EV3Brick.speaker method*), 11  
 send() (*Mailbox method*), 118  
 sensor\_index (*Ev3devSensor attribute*), 77

server\_close() (*BluetoothMailboxClient method*), 118  
 set\_font() (*EV3Brick.screen method*), 15  
 set\_font() (*Image method*), 112  
 set\_speech\_options() (*EV3Brick.speaker method*), 11  
 set\_volume() (*EV3Brick.speaker method*), 13  
 settings() (*DriveBase method*), 90  
 SoundFile (*class in pybricks.media.ev3dev*), 98  
 SoundFile.ACTIVATE (*in module pybricks.media.ev3dev*), 99  
 SoundFile.AIR\_RELEASE (*in module pybricks.media.ev3dev*), 103  
 SoundFile.AIRBRAKE (*in module pybricks.media.ev3dev*), 103  
 SoundFile.ANALYZE (*in module pybricks.media.ev3dev*), 99  
 SoundFile.BACKING\_ALERT (*in module pybricks.media.ev3dev*), 103  
 SoundFile.BACKWARDS (*in module pybricks.media.ev3dev*), 99  
 SoundFile.BLACK (*in module pybricks.media.ev3dev*), 102  
 SoundFile.BLUE (*in module pybricks.media.ev3dev*), 102  
 SoundFile.BOING (*in module pybricks.media.ev3dev*), 98  
 SoundFile.BOO (*in module pybricks.media.ev3dev*), 98  
 SoundFile.BRAVO (*in module pybricks.media.ev3dev*), 101  
 SoundFile.BROWN (*in module pybricks.media.ev3dev*), 102  
 SoundFile.CAT\_PURR (*in module pybricks.media.ev3dev*), 104  
 SoundFile.CHEERING (*in module pybricks.media.ev3dev*), 98  
 SoundFile.CLICK (*in module pybricks.media.ev3dev*), 106  
 SoundFile.COLOR (*in module pybricks.media.ev3dev*), 99  
 SoundFile.CONFIRM (*in module pybricks.media.ev3dev*), 106  
 SoundFile.CRUNCHING (*in module pybricks.media.ev3dev*), 98  
 SoundFile.CRYING (*in module pybricks.media.ev3dev*), 98  
 SoundFile.DETECTED (*in module pybricks.media.ev3dev*), 99  
 SoundFile.DOG\_BARK\_1 (*in module pybricks.media.ev3dev*), 104  
 SoundFile.DOG\_BARK\_2 (*in module pybricks.media.ev3dev*), 104  
 SoundFile.DOG\_GROWL (*in module pybricks.media.ev3dev*), 104



<i>bricks.media.ev3dev</i> ), 104				<i>bricks.media.ev3dev</i> ), 104			
SoundFile.DOG_SNIFF (in module py-				SoundFile.INSECT_CHIRP (in module py-			
<i>bricks.media.ev3dev</i> ), 104				<i>bricks.media.ev3dev</i> ), 104			
SoundFile.DOG_WHINE (in module py-				SoundFile.KUNG_FU (in module py-			
<i>bricks.media.ev3dev</i> ), 104				<i>bricks.media.ev3dev</i> ), 98			
SoundFile.DOWN (in module <i>pybricks.media.ev3dev</i> ),				SoundFile.LASER (in module py-			
99				<i>bricks.media.ev3dev</i> ), 103			
SoundFile.EIGHT (in module py-				SoundFile.LAUGHING_1 (in module py-			
<i>bricks.media.ev3dev</i> ), 105				<i>bricks.media.ev3dev</i> ), 98			
SoundFile.ELEPHANT_CALL (in module py-				SoundFile.LAUGHING_2 (in module py-			
<i>bricks.media.ev3dev</i> ), 104				<i>bricks.media.ev3dev</i> ), 98			
SoundFile.ERROR (in module py-				SoundFile.LEFT (in module <i>pybricks.media.ev3dev</i> ),			
<i>bricks.media.ev3dev</i> ), 99				100			
SoundFile.ERROR_ALARM (in module py-				SoundFile.LEGO (in module <i>pybricks.media.ev3dev</i> ),			
<i>bricks.media.ev3dev</i> ), 99				101			
SoundFile.EV3 (in module <i>pybricks.media.ev3dev</i> ),				SoundFile.MAGIC_WAND (in module py-			
101				<i>bricks.media.ev3dev</i> ), 98			
SoundFile.FANFARE (in module py-				SoundFile.MINDSTORMS (in module py-			
<i>bricks.media.ev3dev</i> ), 98				<i>bricks.media.ev3dev</i> ), 101			
SoundFile.FANTASTIC (in module py-				SoundFile.MORNING (in module py-			
<i>bricks.media.ev3dev</i> ), 101				<i>bricks.media.ev3dev</i> ), 101			
SoundFile.FIVE (in module <i>pybricks.media.ev3dev</i> ),				SoundFile.MOTOR_IDLE (in module py-			
105				<i>bricks.media.ev3dev</i> ), 103			
SoundFile.FLASHING (in module py-				SoundFile.MOTOR_START (in module py-			
<i>bricks.media.ev3dev</i> ), 100				<i>bricks.media.ev3dev</i> ), 103			
SoundFile.FORWARD (in module py-				SoundFile.MOTOR_STOP (in module py-			
<i>bricks.media.ev3dev</i> ), 100				<i>bricks.media.ev3dev</i> ), 103			
SoundFile.FOUR (in module <i>pybricks.media.ev3dev</i> ),				SoundFile.NINE (in module <i>pybricks.media.ev3dev</i> ),			
105				106			
SoundFile.GAME_OVER (in module py-				SoundFile.NO (in module <i>pybricks.media.ev3dev</i> ),			
<i>bricks.media.ev3dev</i> ), 101				101			
SoundFile.GENERAL_ALERT (in module py-				SoundFile.OBJECT (in module py-			
<i>bricks.media.ev3dev</i> ), 106				<i>bricks.media.ev3dev</i> ), 100			
SoundFile.GO (in module <i>pybricks.media.ev3dev</i> ),				SoundFile.OKAY (in module <i>pybricks.media.ev3dev</i> ),			
101				102			
SoundFile.GOOD (in module <i>pybricks.media.ev3dev</i> ),				SoundFile.OKEY_DOKEY (in module py-			
101				<i>bricks.media.ev3dev</i> ), 102			
SoundFile.GOOD_JOB (in module py-				SoundFile.ONE (in module <i>pybricks.media.ev3dev</i> ),			
<i>bricks.media.ev3dev</i> ), 101				105			
SoundFile.GOODBYE (in module py-				SoundFile.OUCH (in module <i>pybricks.media.ev3dev</i> ),			
<i>bricks.media.ev3dev</i> ), 101				98			
SoundFile.GREEN (in module py-				SoundFile.OVERPOWER (in module py-			
<i>bricks.media.ev3dev</i> ), 102				<i>bricks.media.ev3dev</i> ), 106			
SoundFile.HELLO (in module py-				SoundFile.RATCHET (in module py-			
<i>bricks.media.ev3dev</i> ), 101				<i>bricks.media.ev3dev</i> ), 104			
SoundFile.HI (in module <i>pybricks.media.ev3dev</i> ),				SoundFile.READY (in module py-			
101				<i>bricks.media.ev3dev</i> ), 106			
SoundFile.HORN_1 (in module py-				SoundFile.RED (in module <i>pybricks.media.ev3dev</i> ),			
<i>bricks.media.ev3dev</i> ), 103				103			
SoundFile.HORN_2 (in module py-				SoundFile.RIGHT (in module py-			
<i>bricks.media.ev3dev</i> ), 103				<i>bricks.media.ev3dev</i> ), 100			
SoundFile.INSECT_BUZZ_1 (in module py-				SoundFile.SEARCHING (in module py-			
<i>bricks.media.ev3dev</i> ), 104				<i>bricks.media.ev3dev</i> ), 100			
SoundFile.INSECT_BUZZ_2 (in module py-				SoundFile.SEVEN (in module py-			

- bricks.media.ev3dev*), 105
- SoundFile.SHOUTING (in module *pybricks.media.ev3dev*), 98
- SoundFile.SIX (in module *pybricks.media.ev3dev*), 105
- SoundFile.SMACK (in module *pybricks.media.ev3dev*), 99
- SoundFile.SNAKE\_HISS (in module *pybricks.media.ev3dev*), 105
- SoundFile.SNAKE\_RATTLE (in module *pybricks.media.ev3dev*), 105
- SoundFile.SNEEZING (in module *pybricks.media.ev3dev*), 99
- SoundFile.SNORING (in module *pybricks.media.ev3dev*), 99
- SoundFile.SONAR (in module *pybricks.media.ev3dev*), 104
- SoundFile.SORRY (in module *pybricks.media.ev3dev*), 102
- SoundFile.SPEED\_DOWN (in module *pybricks.media.ev3dev*), 102
- SoundFile.SPEED\_IDLE (in module *pybricks.media.ev3dev*), 102
- SoundFile.SPEED\_UP (in module *pybricks.media.ev3dev*), 102
- SoundFile.START (in module *pybricks.media.ev3dev*), 100
- SoundFile.STOP (in module *pybricks.media.ev3dev*), 100
- SoundFile.T\_REX\_ROAR (in module *pybricks.media.ev3dev*), 105
- SoundFile.TEN (in module *pybricks.media.ev3dev*), 106
- SoundFile.THANK\_YOU (in module *pybricks.media.ev3dev*), 102
- SoundFile.THREE (in module *pybricks.media.ev3dev*), 105
- SoundFile.TICK\_TACK (in module *pybricks.media.ev3dev*), 104
- SoundFile.TOUCH (in module *pybricks.media.ev3dev*), 100
- SoundFile.TURN (in module *pybricks.media.ev3dev*), 100
- SoundFile.TWO (in module *pybricks.media.ev3dev*), 105
- SoundFile.UH\_OH (in module *pybricks.media.ev3dev*), 99
- SoundFile.UP (in module *pybricks.media.ev3dev*), 100
- SoundFile.WHITE (in module *pybricks.media.ev3dev*), 103
- SoundFile.YELLOW (in module *pybricks.media.ev3dev*), 103
- SoundFile.YES (in module *pybricks.media.ev3dev*), 102
- SoundFile.ZERO (in module *pybricks.media.ev3dev*), 105
- SoundSensor (class in *pybricks.nxtdevices*), 66
- speed() (*GyroSensor* method), 62
- speed() (*Motor* method), 24, 57
- stall\_tolerances() (*Control* method), 129
- stalled() (*Control* method), 127
- state() (*DriveBase* method), 91
- Stop (built-in class), 84
- stop() (*DriveBase* method), 91
- stop() (*Motor* method), 25, 57
- StopWatch (class in *pybricks.tools*), 89
- storage() (*EnergyMeter* method), 67
- straight() (*DriveBase* method), 90
- style (*Font* attribute), 107
- ## T
- target\_tolerances() (*Control* method), 128
- TechnicHub (built-in class), 7
- temperature() (*TemperatureSensor* method), 66
- TemperatureSensor (class in *pybricks.nxtdevices*), 66
- text\_height() (*Font* method), 110
- text\_width() (*Font* method), 109
- TextMailbox (class in *pybricks.messaging*), 119
- tilt() (*TiltSensor* method), 33
- TiltSensor (class in *pybricks.pupdevices*), 33
- time() (*StopWatch* method), 89
- touched() (*ForceSensor* method), 49
- TouchSensor (class in *pybricks.ev3devices*), 59
- TouchSensor (class in *pybricks.nxtdevices*), 63
- track\_target() (*Motor* method), 26, 58
- turn() (*DriveBase* method), 90
- ## U
- UARTDevice (class in *pybricks.iodevices*), 73
- UltrasonicSensor (class in *pybricks.ev3devices*), 61
- UltrasonicSensor (class in *pybricks.nxtdevices*), 65
- UltrasonicSensor (class in *pybricks.pupdevices*), 47
- UP (*Button* attribute), 81
- ## V
- value() (*VernierAdapter* method), 68
- VernierAdapter (class in *pybricks.nxtdevices*), 67
- VIOLET (*Color* attribute), 82
- voltage() (*AnalogSensor* method), 75
- voltage() (*CityHub.battery* method), 5
- voltage() (*EV3Brick.battery* method), 16
- voltage() (*MoveHub.battery* method), 3
- voltage() (*TechnicHub.battery* method), 8
- voltage() (*VernierAdapter* method), 68

## W

`wait()` (in module *pybricks.tools*), 89

`wait()` (*Mailbox* method), 119

`wait_for_connection()` (*BluetoothMailboxServer* method), 117

`wait_new()` (*Mailbox* method), 119

`waiting()` (*UARTDevice* method), 74

`WHITE` (*Color* attribute), 82

`width` (*EV3Brick.screen* attribute), 16

`width` (*Font* attribute), 108

`width` (*Image* attribute), 114

`write()` (*I2CDevice* method), 71

`write()` (*PUPDevice* method), 69

`write()` (*UARTDevice* method), 74

## Y

`YELLOW` (*Color* attribute), 82