# CONTENTS:

1 About GoPiGo3
   1.1 Who are we and what we do. .................................................. 3
   1.2 What’s this documentation about. .......................................... 3

2 Getting Started
   2.1 Buying a GoPiGo3 ................................................................. 5
   2.2 Assembling GoPiGo3 ............................................................ 5
   2.3 Connecting to GoPiGo3 ......................................................... 6
   2.4 Program your GoPiGo3 .......................................................... 6
   2.5 Connect More Sensors .......................................................... 6

3 Tutorials - Basic
   3.1 Driving Around ................................................................. 7
   3.2 Flashing an LED ................................................................. 10
   3.3 Pushing a Button ............................................................... 12
   3.4 Ringing a Buzzer ............................................................... 14
   3.5 Detecting Light ................................................................. 17
   3.6 Measuring Distance ........................................................... 19

4 Tutorials - Advanced ................................................................. 23

5 API GoPiGo3 - Basic
   5.1 About Library & Hardware .................................................... 25
   5.2 API - Library - EasyGoPiGo3 ................................................ 32
   5.3 API - Library - Sensors ....................................................... 44
   5.4 API - Deprecated ............................................................... 59

6 API GoPiGo3 - Advanced ............................................................ 61
   6.1 Requirements ................................................................. 61
   6.2 Sensor ........................................................... 61
   6.3 DigitalSensor ............................................................... 64
   6.4 AnalogSensor .............................................................. 65

7 Developer’s Guide
   7.1 Our collaborators .......................................................... 67

8 Frequently Asked Questions .................................................. 69

9 Indices and tables ................................................................. 71

Index ................................................................. 73
ABOUT GOPIGO3

1.1 Who are we and what we do.

Dexter Industries is an American educational robotics company that develops robot kits that make programming accessible for everyone.

1.2 What’s this documentation about.

This documentation is all about the GoPiGo3 robot. Within this, you will find instructions on:

- How to get started with the GoPiGo3 robot - assembling, setting up the environment, etc.
- How to get started with the example programs found in our repo.
- How to operate the GoPiGo3 with our API. The user has a comprehensive documentation of all the modules/functions/classes that are needed for controlling the robot.
- How to troubleshoot the GoPiGo3 in case of unsuspected situations.
2.1 Buying a GoPiGo3

To buy a GoPiGo3 robot, please head over to our online shop and search for the GoPiGo3 robot. From our shop, you can get sensors for your robot such as the Distance Sensor, the Grove Light Sensor, etc.

2.2 Assembling GoPiGo3

For assembling your GoPiGo3 robot, read the instructions from the following page: assembling instructions.
2.3 Connecting to GoPiGo3

To connect to your GoPiGo3 robot with a computer or laptop, read the instructions on the following page: connecting to robot.

2.4 Program your GoPiGo3

To program your GoPiGo3 to do what you want, you can follow the instructions found here (programming your robot) or follow the rest of instructions found in this section.

To install or update the GoPiGo3 library on your RaspberryPi, open a terminal or the command line and type the following command:

```
# follow any given instructions given through this command
curl -kL dexterindustries.com/update_gopigo3 | bash
```

Also, in order to be able to use the `easygopigo3.EasyGoPiGo3.init_distance_sensor()` and `easygopigo3.EasyGoPiGo3.init_line_follower()` methods, the DI-Sensors package is required. You can install it or update it with the following command in the terminal:

```
# follow any given instructions given through this command
curl -kL dexterindustries.com/update_sensors | bash
```

2.5 Connect More Sensors

The GoPiGo3 can also be paired with our in-house sensors. There are a number of digital and analog sensors that can be connected to the GoPiGo3. We have further in-depth documentation on the following 4 sensors:

- The DI IMU Sensor.
- The DI Light and Color Sensor.
- The DI Temperature Humidity Pressure Sensor.
- The DI Distance Sensor.
- The DI Line Follower Sensor (black board)/Line Follower Sensor (red board).

For more on getting started with these sensors, please check the DI-Sensors documentation.
This chapter revolves around the easygopigo3 module and just a bit around di_sensors.

Please make sure you have followed all the instructions found in Getting Started before jumping into tutorials. In all these tutorials, you will need:

1. A GoPiGo3 robot.
2. A sensor/actuator if requested in the tutorial i.e.: a Grove Buzzer, a Black Line Follower, etc.

### 3.1 Driving Around

Driving the GoPiGo3 around is only a matter of choosing the right methods from the EasyGoPiGo3 class. In all of our examples, the easygopigo3 is required and needs to be imported and a EasyGoPiGo3 object can be instantiated this way:

```python
from easygopigo3 import EasyGoPiGo3
# importing the EasyGoPiGo3 class
gpg = EasyGoPiGo3()
# instantiating a EasyGoPiGo3 object
```

Instantiating `EasyGoPiGo3` object can fail if the GoPiGo3 is not present or there’s a firmware mismatch. Check the `__init__` constructor of this class to see what exceptions can be raised.

#### 3.1.1 Forward, R/L, Backward

The most basic commands don’t offer control over how much distance the GoPiGo3 travels. They only get called once and then it’s the responsibility of the user to stop the GoPiGo3 from moving, regularly by using the `stop()` command.

We first need to import the `easygopigo3` and the `time` modules and instantiate a EasyGoPiGo3 object.

```python
# import the time library for the sleep function
import time
from easygopigo3 import EasyGoPiGo3

# create an instance of the GoPiGo3 class.
# gpg will be the GoPiGo3 object.
gpg = EasyGoPiGo3()
```

Next, move the robot forward for exactly one second and right after that continue to the next block of code.
print("Move the motors forward freely for 1 second.")
gpg.forward()
time.sleep(1)
gpg.stop()

Stop the motors for a second.

print("Stop the motors for 1 second.")
time.sleep(1)

And then move forward again, but this time for 50 centimeters and once the moving function ends, wait a second until the next block of code gets executed. Notice that $drive\_cm()$ is a blocking function in this case.

print("Drive the motors 50 cm and then stop.")
gpg.drive_cm(50, True)
time.sleep(1)

Then right for a second.

print("Turn right 1 second.")
gpg.right()
time.sleep(1)

And likewise to the left.

print("Turn left 1 second.")
gpg.left()
time.sleep(1)

Finally, stop the robot.

print("Stop!")
gpg.stop()
print("Done!")

If you want to run this by yourself, here’s the script on github.

3.1.2 Describing a Square

To make the GoPiGo3 describe a square by moving itself, you need to run the following script. To do this, $drive\_cm()$ and $turn\_degrees()$ methods are required. A square with the side length of 30cm is drawn. The square is drawn clockwise.

```
from easygopigo3 import EasyGoPiGo3
gpg = EasyGoPiGo3()
length = 30

for i in range(4):
    gpg.drive_cm(length) # drive forward for length cm
    gpg.turn_degrees(90) # rotate 90 degrees to the right
```
3.1.3 Making Circular Moves

Driving straight in one direction is one thing, but rotating around a center axis at a specific radius is something entirely different. In this example, the GoPiGo3 draws half a circle and then returns on the same track by spinning itself on the spot.

The radius of the circle is set at 50 centimeters and the robot will move for half of the circle (aka 180 degrees).

```python
from easygopigo3 import EasyGoPiGo3
gpg = EasyGoPiGo3()
gpg.orbit(180, 50) # draw half a circle
gpg.turn_degrees(180) # rotate the GoPiGo3 around
gpg.orbit(-180, 50) # return on the initial path
gpg.turn_degrees(180) # and put it in the initial position
```

3.1.4 Drawing an 8 Shape

Let’s say we want to draw an 8 shape with the GoPiGo3 and at the end have the GoPiGo3 reach the same position it initially left from.

To do this, we have to use `orbit()` and `drive_cm()` methods.

```python
from easygopigo3 import EasyGoPiGo3
gpg = EasyGoPiGo3()
radius = 30

gpg.orbit(-270, radius) # to rotate to the left
gpg.drive_cm(radius * 2) # move forward

gpg.orbit(270, radius) # to rotate to the right
gpg.drive_cm(radius * 2) # move forward
```

3.1.5 Going Forward at Increasing Speed

In this example, we make the GoPiGo3 go forward at an ever increasing speed. We start off with a speed of 50 and end up going at 300. `forward()`, `set_speed()` and `stop()` methods are used.

```python
Warning: This example will not work with versions released before November 2018. Do an update before running it.

from easygopigo3 import EasyGoPiGo3
from time import time, sleep

gpg = EasyGoPiGo3()

# setting speed to lowest value and
# calculating the step increase in speed
```

(continues on next page)
current_speed = 50
end_speed = 400
step = (end_speed - current_speed) / 20
gpg.set_speed(current_speed)

# start moving the robot at an ever increasing speed
gpg.forward()
while current_speed <= end_speed:
    sleep(0.1)
    gpg.set_speed(current_speed)
    current_speed += step

# and then stop it
gpg.stop()

3.1.6 Other Examples

There are also other examples you can look at, namely the projects in the GoPiGo3 repository. Also, to see all methods for moving around the GoPiGo3, check the GoPiGo3 movement API.

3.2 Flashing an LED

3.2.1 Our goal

In this tutorial, we are making a Grove Led flash continuously, while it’s being connected to a GoPiGo3 robot.

3.2.2 The code we analyse

The code we’re analyzing in this tutorial is the following one.

```python
# import the EasyGoPiGo3 drivers
import time
import easygopigo3 as easy

# Create an instance of the GoPiGo3 class.
# GPG will be the GoPiGo3 object.
gpg = easy.EasyGoPiGo3()

# create the LED instance, passing the port and GPG
my_led = gpg.init_led("AD1")

# loop 100 times
for i in range(100):
    my_led.light_max() # turn LED at max power
time.sleep(0.5)

    my_led.light_on(30) # 30% power
time.sleep(0.5)
```

(continues on next page)
The source code for this example program can be found here on github.

### 3.2.3 The modules

Start by importing 2 important modules:

```python
import time
import easygopigo3 as easy
```

The `easygopigo3` module is used for interacting with the GoPiGo robot, whilst the `time` module is generally used for delaying actions, commands, setting timers etc.

### 3.2.4 The objects

After this, we need to instantiate an `easygopigo3.EasyGoPiGo3` object. We are using the `EasyGoPiGo3` object for creating an instance of `Led` class, which is necessary for controlling the Grove Led device.

```python
gpg = easy.EasyGoPiGo3()
```

Now that we have an `EasyGoPiGo3` object, we can instantiate a `Led` object. The argument of the initializer method is the port to which we connect the Grove Led and it’s set to "AD1".

```python
my_led = gpg.init_led("AD1")
```

**Note:** See the following *graphical representation* as a reference to where the ports are.

### 3.2.5 Main part

In this section of the tutorial we are focusing on 3 methods of the `easysensors.Led` class.

- The `light_max()` method - which turns the LED at the maximum brightness.
- The `light_on()` method - used for turning the LED at a certain percent of the maximum brightness.
- The `light_off()` method - used for turning off the LED.

All in all, the following code snippet turns on the LED to the maximum brightness, then it sets the LED’s brightness at 30% and in the last it turns off the LED. The delay between all these 3 commands is set at half a second.

```python
for i in range(100):
    my_led.light_max()  # turn LED at max power
    time.sleep(0.5)

my_led.light_on(30)  # 30% power
    time.sleep(0.5)

my_led.light_off()  # turn LED off
    time.sleep(0.5)
```
3.2.6 Running it

Connect the Grove Led to your GoPiGo3 robot to port "AD1" and then let’s crank up the Raspberry Pi. For running the analyzed example program, within a terminal on your Raspberry Pi, type the following 2 commands:

```bash
cd ~/Desktop/GoPiGo3/Software/Python/Examples
python easy_LED.py
```

3.3 Pushing a Button

3.3.1 Our goal

In this tutorial, we are going to control GoPiGo3 Dex’s eyes with a Grove Button.

- When the Grove Button is pressed, Dex’s eyes turn on.
- When the Grove Button is released, Dex’s eyes turn off.

3.3.2 The code we analyse

In the end the code should look like this.

```python
# import the time library for the sleep function
import time

# import the GoPiGo3 drivers
import easygopigo3 as easy

# Create an instance of the GoPiGo3 class.
# GPG will be the GoPiGo3 object.
gpg = easy.EasyGoPiGo3()

# Put a grove button in port AD1
my_button = gpg.init_button_sensor("AD1")

print("Ensure there's a button in port AD1")
print("Press and release the button as often as you want")
print("the program will run for 2 minutes or")
print("Ctrl-C to interrupt it")

start = time.time()  
RELEASED = 0  
PRESSED = 1  
state = RELEASED

while time.time() - start < 120:
    if state == RELEASED and my_button.read() == 1:
        print("PRESSED")
        gpg.open_eyes()
        state = PRESSED
```
if state == PRESSED and my_button.read() == 0:
    print("RELEASED")
    gpg.close_eyes()
    state = RELEASED
    time.sleep(0.05)
print("All done!")

The source code for this example program can be found here on github.

3.3.3 The modules

Start by importing 2 important modules:

```python
import time
import easygopigo3 as easy
```

The `easygopigo3` module is used for interacting with the GoPiGo robot, whilst the `time` module is generally used for delaying actions, commands, setting timers etc.

3.3.4 The objects

After this, we need to instantiate an `easygopigo3.EasyGoPiGo3` object. The `EasyGoPiGo3` object is used for 2 things:

- For turning ON and OFF the GoPiGo3 Dex's eyes.
- For instantiating a `ButtonSensor` object for reading the Grove Button's state.

```python
gpg = easy.EasyGoPiGo3()
```

Now that we have an `EasyGoPiGo3` object, we can instantiate a `ButtonSensor` object. The argument of the initializer method is the port to which we connect the Grove Button and it's set to "AD1".

```python
my_button = gpg.init_button_sensor("AD1")
```

**Note:** See the following `graphical representation` as a reference to where the ports are.

3.3.5 Setting variables

Define 2 states for the button we're using. We are setting the default state to "RELEASED".

```python
start = time.time()
RELEASED = 0
PRESSED = 1
state = RELEASED
```

There's also a variable called `start` to which we assign the clock time of that moment. We use it to limit for how long the script runs.
3.3.6 Main part

The main part is basically a while loop that’s going to run for 120 seconds. Within the while loop, we have 2 if / else blocks that define a simple algorithm: whenever the previous state is different from the current one, we either turn on or close Dex’s eyes. Here’s the logic:

- If in the previous iteration of the while loop the button was released and now the button is 1 (aka pressed), then we turn on the LEDs and save the new state in state variable.
- If in the previous iteration of the while loop the button was pressed and now the button is 0 (aka released), then we turn off the LEDs and save the new state in state variable.

This way, we don’t call gpg.open_eyes() all the time when the button is pressed or gpg.close_eyes() when the button is released. It only needs to call one of these 2 functions once.

```python
while time.time() - start < 120:
    if state == RELEASED and my_button.read() == 1:
        print("PRESSED")
        gpg.open_eyes()
        state = PRESSED
    if state == PRESSED and my_button.read() == 0:
        print("RELEASED")
        gpg.close_eyes()
        state = RELEASED
    time.sleep(0.05)
```

time.sleep(0.05) was added to limit the CPU time. 50 mS is more than enough.

3.3.7 Running it

Make sure you have connected the Grove Button to your GoPiGo3 robot to port "AD1". Then, on the Rasperry Pi, from within a terminal, type the following commands.

```bash
cd ~/Desktop/GoPiGo3/Software/Python/Examples
python easy_Button.py
```

3.4 Ringing a Buzzer

3.4.1 Our goal

In this tutorial, we are making a Grove Buzzer play different musical tones on our GoPiGo3 robot. We start off with 3 musical notes and finish by playing the well-known “Twinkle Twinkle Little Star” song.

3.4.2 The code we analyse

The code we’re analyzing in this tutorial is this.
# import the time library for the sleep function
import time

# import the GoPiGo3 drivers
import easygopigo3 as easy

# Create an instance of the GoPiGo3 class.
# GPG will be the GoPiGo3 object.
gpg = easy.EasyGoPiGo3()

# Create an instance of the Buzzer
# connect a buzzer to port AD2
my_buzzer = gpg.init_buzzer("AD2")

# Twinkle notes
twinkle = ["C4","C4","G4","G4","A4","A4","G4"]

print("Expecting a buzzer on Port AD2")
print("A4")
my_buzzer.sound(440)
time.sleep(1)
print("A5")
my_buzzer.sound(880)
time.sleep(1)
print("A3")
my_buzzer.sound(220)
time.sleep(1)

for note in twinkle:
    print(note)
    my_buzzer.sound(my_buzzer.scale[note])
    time.sleep(0.5)
    my_buzzer.sound_off()
    time.sleep(0.25)

my_buzzer.sound_off()

The source code for this example program can be found [here on github](https://github.com).

## 3.4.3 The modules

Start by importing 2 important modules:

```python
import time
import easygopigo3 as easy
```

The `easygopigo3` module is used for interacting with the GoPiGo3 robot, whilst the `time` module is generally used for delaying actions, commands, setting timers etc.

## 3.4.4 The objects

After this, we need to instantiate an `easygopigo3.EasyGoPiGo3` object. We will be using the `EasyGoPiGo3` object for creating an instance of `Buzzer` class, which is necessary for controlling the Grove Buzzer device.

```python
gpg = easy.EasyGoPiGo3()
```

## 3.4. Ringing a Buzzer
Now that we have an EasyGoPiGo3 object, we can instantiate a Buzzer object. The argument of the initializer method is the port to which we connect the Grove Buzzer and it’s set to "AD2".

```python
my_buzzer = gpg.init_buzzer("AD2")
```

**Note:** See the following *graphical representation* as a reference to where the ports are.

### 3.4.5 Setting variables

To play the “Twinkle Twinkle Little Star” song, we need to have a sequence of musical notes that describe this song. We’re encoding the musical notes into a list (called `twinkle`) of strings, where each string represents a musical note.

```python
twinkle = ["C4","C4","G4","G4","A4","A4","G4"]
```

### 3.4.6 Main part

The main zone of the code is divided into 2 sections:

1. The 1st section, where we only play 3 musical notes with a 1 second delay.
2. The 2nd section, where we play the lovely “Twinkle Twinkle Little Start” song.

In the 1st section, we use the `easysensors.Buzzer.sound()` method, which takes as a paramater, an integer that represents the frequency of the emitted sound. As you can see in the following code snippet, each musical note corresponds to a certain frequency:

- The frequency of `A4` musical note is 440Hz.
- The frequency of `A5` musical note is 880Hz.
- The frequency of `A3` musical note is 220Hz.

```python
print("A4")
my_buzzer.sound(440)
time.sleep(1)

print("A5")
my_buzzer.sound(880)
time.sleep(1)

print("A3")
my_buzzer.sound(220)
time.sleep(1)
```

In the 2nd section we are using the `scale` dictionary. In this dictionary there are stored the frequencies of each musical note. So, when using the `twinkle` list in conjuction with `scale` attribute, we’re basically retrieving the frequency of a musical note (found in `twinkle` attribute) from the `scale` dictionary.

```python
for note in twinkle:
    print(note)
    my_buzzer.sound(buzzer.scale[note])
time.sleep(0.5)
my_buzzer.sound_off()
time.sleep(0.25)
```
3.4.7 Running it

The only thing left to do is to connect the Grove Buzzer to your GoPiGo3 robot to port "AD2". Then, on your Raspberry Pi, from within a terminal, type the following commands:

```
cd ~/Desktop/GoPiGo3/Software/Python/Examples
python easy_Buzzer.py
```

**Tip:** Please don’t expect to hear a symphony, because the buzzer wasn’t made for playing tones. We use the buzzer within this context to only demonstrate that it’s a nice feature.

3.5 Detecting Light

3.5.1 Our goal

In this tutorial, we are making a Grove Light Sensor light up a Grove Led depending on how strong the intensity of the light is. The Grove Light Sensor and the Grove Led are both connected to the GoPiGo3 and use the following ports.

- Port "AD1" for the light sensor.
- Port "AD2" for the LED.

**Important:** Since this tutorial is based on Led tutorial, we recommend following that one before going through the current one.

3.5.2 The code we analyse

The code we’re analyzing in this tutorial is the following one.

```
# import the time library for the sleep function
import time

# import the GoPiGo3 drivers
import easygopigo3 as easy

# Create an instance of the GoPiGo3 class.
# GPG will be the GoPiGo3 object.
gpg = easy.EasyGoPiGo3()

# Create an instance of the Light sensor
my_light_sensor = gpg.init_light_sensor("AD1")
my_led = gpg.init_led("AD2")

# loop forever while polling the sensor
while(True):
    # get absolute value
    reading = my_light_sensor.read()
    # scale the reading to a 0-100 scale
    percent_reading = my_light_sensor.percent_read() (continues on next page)
```
GoPiGo3 Documentation

# check if the light's intensity is above 50%
if percent_reading >= 50:
    my_led.light_off()
else:
    my_led.light_max()
print("{}, {:.1f}%".format(reading, percent_reading))
time.sleep(0.05)

The source code for this tutorial can also be found here on github.

3.5.3 The modules

Start by importing 2 important modules:

```python
import time
import easygopigo3 as easy
```

The easygopigo3 module is used for interacting with the GoPiGo3 robot, whilst the time module is generally used for delaying actions, commands, setting timers etc.

3.5.4 The objects

After this, we need to instantiate an easygopigo3.EasyGoPiGo3 object. We are using the EasyGoPiGo3 object for creating an instance of Led class, which is necessary for controlling the Grove Led and for reading off of the Grove Light Sensor.

```python
gpg = easy.EasyGoPiGo3()
```

Now that we have an EasyGoPiGo3 object, we can instantiate a LightSensor and Led objects. The argument of each of the 2 initializer methods represents the port to which a device is connected.

```python
my_light_sensor = gpg.init_light_sensor("AD1")
my_led = gpg.init_led("AD2")
```

Note: See the following graphical representation as a reference to where the ports are.

3.5.5 Main part

Let’s make the LED behave in the following way.

- When the light’s intensity is below 50%, turn on the LED.
- When the light's intensity is above 50%, turn off the LED.

To do this, we need to read the percentage value off of the light sensor - the variable responsible for holding the value is called percent_reading. Depending on the determined percentage, we turn the LED on or off.

To do all this, check out the following code snippet.
while(True):
    # get absolute value
    reading = my_light_sensor.read()
    # scale the reading to a 0-100 scale
    percent_reading = my_light_sensor.percent_read()

    # check if the light's intensity is above 50%
    if percent_read >= 50:
        my_led.light_off()
    else:
        my_led.light_max()

    print("{}", {:.1f}%%.format(reading, percent_reading))

    time.sleep(0.05)

3.5.6 Running it

Here’s the fun part. Let’s run the python script.

Connect the Grove Light Sensor to your GoPiGo3 robot to port "AD1" and Grove Led to port "AD2". Within a terminal on your Raspberry Pi, type the following 2 commands:

```bash
cd ~/Desktop/GoPiGo3/Software/Python/Examples
python easy_Light_Sensor.py
```

3.6 Measuring Distance

3.6.1 Our goal

In this tutorial, we are using a Distance Sensor for measuring the distance to a target with the GoPiGo3 robot. We are going to print the values on a terminal.

3.6.2 The code we analyse

The code we’re analyzing in this tutorial is the following one.

```python
# import the GoPiGo3 drivers
import time
import easygopigo3 as easy

# This example shows how to read values from the Distance Sensor

# Create an instance of the GoPiGo3 class.
# GPG will be the GoPiGo3 object.
gpg = easy.EasyGoPiGo3()

# Create an instance of the Distance Sensor class.
# I2C1 and I2C2 are just labels used for identifying the port on the GoPiGo3 board.
# But technically, I2C1 and I2C2 are the same thing, so we don't have to pass any
# port to the constructor.
```
my_distance_sensor = gpg.init_distance_sensor()

while True:
    # Directly print the values of the sensor.
    print("Distance Sensor Reading (mm): " + str(my_distance_sensor.read_mm()))

The source code for this example program can be found here on github.

3.6.3 The modules

Start by importing 2 important modules:

import time
import easygopigo3 as easy

The easygopigo3 module is used for interacting with the GoPiGo3 robot, whilst the time module is generally used for delaying actions, commands, setting timers etc.

3.6.4 The objects

For interfacing with the Distance Sensor we need to instantiate an object of the easygopigo3. EasyGoPiGo3 class so in return, we can instantiate an object of the di_sensors.easy_distance_sensor. EasyDistanceSensor class. We do it like in the following code snippet.

gpg = easy.EasyGoPiGo3()  # this is an EasyGoPiGo3 object
my_distance_sensor = gpg.init_distance_sensor()  # this is a DistanceSensor object

Important: Notice that in order for this to work, we also need to have the DI-Sensors library installed. Follow the instructions on that documentation on how to install the required package.

3.6.5 Main part

There’s a single while loop in the entire script. The loop is for printing the values that we’re reading repeatedly. We will be using the read_mm() method for reading the distance in millimeters to the target.

while True:
    # Directly print the values of the sensor.
    print("Distance Sensor Reading (mm): " + str(my_distance_sensor.read_mm()))

See also:
Check out di_sensors.easy_distance_sensor.EasyDistanceSensor’s API for more details.

3.6.6 Running it

Connect the Distance Sensor to any of the 2 "I2C" ports on the GoPiGo3 robot. After the sensor is connected, on your Raspberry Pi, open up a terminal and type in the following 2 commands.
cd ~/Desktop/GoPiGo3/Software/Python/Examples
python easy_Distance_Sensor.py

**Note:** See the following *graphical representation* as a reference to where the ports are.
Note: Coming soon!
5.1 About Library & Hardware

5.1.1 Requirements

Before using this chapter’s classes, you need to be able to import the following modules.

```python
import easygopigo3
import easysensors
```

If you have issues importing these two modules, then make sure:

- You have followed the steps found in *Getting Started* guide.
- You have installed either Raspbian For Robots or the GoPiGo3 repository’s python package.
- You have the `gopigo3` package installed by typing the command `pip freeze | grep gopigo3` on your Raspberry Pi’s terminal. If the package is installed, then a string with the `gopigo3==[x.y.z]` format will show up.

If you encounter issues that aren’t covered by our *Getting Started* guide or *FAQ* chapter, please head over to our forum.

5.1.2 Hardware Ports

In this graphical representation, the GoPiGo3 board has the following ports available for use. The quoted literals are to be used as pin identifiers inside the python scripts.
These ports have the following functionalities:

- Ports "AD1" and "AD2" - general purpose input/output ports.
- Ports "SERVO1" and "SERVO2" - servo controller ports.
- Ports "I2C" - ports to which you can connect I2C-enabled devices.
- Port "SERIAL" - port to which you can connect UART-enabled device.

Note: Use the quoted port names when referencing them inside a python script like in the following example.

```python
# we need an EasyGoPiGo3 object for instantiating sensor / actuator objects
gpg3_obj = EasyGoPiGo3()

# we're using the quoted port names from the above graphical representation

# here's a LightSensor object bindsed on port AD2
light_obj = gpg3_obj.init_light_sensor("AD2")

# here's a UltraSonicSensor object bindsed on port AD1
us_obj = gpg3_obj.init_ultrasonic_sensor("AD1")

# here's a LineFollower object bindsed on port I2C
line_follower_obj = gpg3_obj.init_line_follower("I2C")

# and so on
```

See also:

For more technical details on the GoPiGo3 robot, please check our technical specs page.
5.1.3 Library Structure

Classes Short-List

Each of the following classes are organized in 3 modules: gopigo3, easygopigo3 and easysensors. Out of these 3 modules, we’re only interested in easygopigo3 and easysensors as gopigo3 is meant for more advanced users.

Note: The following graphic isn’t just a photo (PNG, JPEG, etc), but it’s an interactive graphic on which if you click on a class, you’ll land on the documentation of a given class.

For initializing Sensor-derived objects, there are 2 ways to do it:

1. Either by using the init methods of the EasyGoPiGo3 class. This is the preferred way.
2. Or by using the constructor of each class (DHTSensor, Led, etc), which is harder because it also requires passing a EasyGoPiGo3 object for instantiation.

Note: Class easysensors.AnalogSensor does not only deal with sensors, but outputs as well like easysensors.Led and easysensors.Buzzer. While we agree this isn’t the best name for this class, we’re
**GoPiGo3 Documentation**

going to keep it this way due to compatibility reasons we don’t want to break (for now).

---

**Functions Short-List**

Here’s a short summary of the methods of all classes that are documented. We start off with a couple of lists for the methods/functions/methods of the easygopigo3 module and then we finish with another few for the easysensors module, so that we can create a visual distinction one from another and thus make it more easy to search around.

**GoPiGo3 Movement**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>easygopigo3.EasyGoPiGo3([config_file_path, ...])</td>
<td>This class is used for controlling a GoPiGo3 robot.</td>
</tr>
<tr>
<td>easygopigo3.EasyGoPiGo3.volt()</td>
<td>This method returns the battery voltage of the GoPiGo3.</td>
</tr>
<tr>
<td>easygopigo3.EasyGoPiGo3.set_speed(in_speed)</td>
<td>This method sets the speed of the GoPiGo3 specified by in_speed argument.</td>
</tr>
<tr>
<td>easygopigo3.EasyGoPiGo3.get_speed()</td>
<td>Use this method for getting the speed of your GoPiGo3.</td>
</tr>
<tr>
<td>easygopigo3.EasyGoPiGo3.reset_speed()</td>
<td>This method resets the speed to its original value.</td>
</tr>
<tr>
<td>easygopigo3.EasyGoPiGo3.backward()</td>
<td>Move the GoPiGo3 backward.</td>
</tr>
<tr>
<td>easygopigo3.EasyGoPiGo3.right()</td>
<td>Move the GoPiGo3 to the right.</td>
</tr>
<tr>
<td>easygopigo3.EasyGoPiGo3.spin_right()</td>
<td>Rotate the GoPiGo towards the right while staying on the same spot.</td>
</tr>
<tr>
<td>easygopigo3.EasyGoPiGo3.left()</td>
<td>Move the GoPiGo3 to the left.</td>
</tr>
<tr>
<td>easygopigo3.EasyGoPiGo3.spin_left()</td>
<td>Rotate the GoPiGo towards the left while staying on the same spot.</td>
</tr>
<tr>
<td>easygopigo3.EasyGoPiGo3.steer(left_percent,...)</td>
<td>Control each motor in order to get a variety of turning movements.</td>
</tr>
<tr>
<td>easygopigo3.EasyGoPiGo3.orbit(degrees[,...])</td>
<td>Control the GoPiGo so it will orbit around an object.</td>
</tr>
<tr>
<td>easygopigo3.EasyGoPiGo3.forward()</td>
<td>Move the GoPiGo3 forward.</td>
</tr>
<tr>
<td>easygopigo3.EasyGoPiGo3.drive_cm(dist[, ...])</td>
<td>Move the GoPiGo3 forward / backward for dist amount of centimeters.</td>
</tr>
<tr>
<td>easygopigo3.EasyGoPiGo3.drive_inches(dist[, ...])</td>
<td>Move the GoPiGo3 forward / backward for dist amount of inches.</td>
</tr>
<tr>
<td>easygopigo3.EasyGoPiGo3.drive_degrees(degrees)</td>
<td>Move the GoPiGo3 forward / backward for degrees / 360 wheel rotations.</td>
</tr>
<tr>
<td>easygopigo3.EasyGoPiGo3.target_reached(...)</td>
<td>Checks if (wheels have rotated for a given number of degrees):</td>
</tr>
<tr>
<td>easygopigo3.EasyGoPiGo3.reset_encoders(...)</td>
<td>Resets both the encoders back to 0.</td>
</tr>
<tr>
<td>easygopigo3.EasyGoPiGo3.read_encoders()</td>
<td>Reads the encoders’ position in degrees.</td>
</tr>
<tr>
<td>easygopigo3.EasyGoPiGo3.turn_degrees(degrees)</td>
<td>Makes the GoPiGo3 robot turn at a specific angle while staying in the same spot.</td>
</tr>
</tbody>
</table>

**GoPiGo3 LEDs**
### easygopigo3.EasyGoPiGo3.blinker_on(id)
Turns ON one of the 2 red blinkers that GoPiGo3 has.

### easygopigo3.EasyGoPiGo3.blinker_off(id)
Turns OFF one of the 2 red blinkers that GoPiGo3 has.

### easygopigo3.EasyGoPiGo3.set_left_eye_color(color)
Sets the LED color for Dexter mascot’s left eye.

### easygopigo3.EasyGoPiGo3.set_right_eye_color(color)
Sets the LED color for Dexter mascot’s right eye.

### easygopigo3.EasyGoPiGo3.set_eye_color(color)
Sets the LED color for Dexter mascot’s eyes.

### easygopigo3.EasyGoPiGo3.open_left_eye()
Turns ON Dexter mascot’s left eye.

### easygopigo3.EasyGoPiGo3.open_right_eye()
Turns ON Dexter mascot’s right eye.

### easygopigo3.EasyGoPiGo3.open_eyes()
Turns ON Dexter mascot’s eyes.

### easygopigo3.EasyGoPiGo3.close_left_eye()
Turns OFF Dexter mascot’s left eye.

### easygopigo3.EasyGoPiGo3.close_right_eye()
Turns OFF Dexter mascot’s right eye.

### easygopigo3.EasyGoPiGo3.close_eyes()
Turns OFF Dexter mascot’s eyes.

### GoPiGo3 Constants

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>easygopigo3.EasyGoPiGo3.load_robot_constants(...)</td>
<td>Load wheel diameter and wheel base width constants for the GoPiGo3 from file.</td>
</tr>
<tr>
<td>easygopigo3.EasyGoPiGo3.save_robot_constants(...)</td>
<td>Save the current wheel diameter and wheel base width constants (from within this object’s context) for the GoPiGo3 to file for future use.</td>
</tr>
<tr>
<td>easygopigo3.EasyGoPiGo3.set_robot_constants(...)</td>
<td>Set new wheel diameter and wheel base width values for the GoPiGo3.</td>
</tr>
</tbody>
</table>

### GoPiGo3 Init Methods

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>easygopigo3.EasyGoPiGo3.init_light_sensor([port])</td>
<td>Initialises a LightSensor object and then returns it.</td>
</tr>
<tr>
<td>easygopigo3.EasyGoPiGo3.init_sound_sensor([port])</td>
<td>Initialises a SoundSensor object and then returns it.</td>
</tr>
<tr>
<td>easygopigo3.EasyGoPiGo3.init_loudness_sensor([port])</td>
<td>Initialises a LoudnessSensor object and then returns it.</td>
</tr>
<tr>
<td>easygopigo3.EasyGoPiGo3.init_ultrasonic_sensor([port])</td>
<td>Initialises a UltraSonicSensor object and then returns it.</td>
</tr>
<tr>
<td>easygopigo3.EasyGoPiGo3.init_buzzer([port])</td>
<td>Initialises a Buzzer object and then returns it.</td>
</tr>
<tr>
<td>easygopigo3.EasyGoPiGo3.init_led([port])</td>
<td>Initialises a Led object and then returns it.</td>
</tr>
<tr>
<td>easygopigo3.EasyGoPiGo3.init_button_sensor([port])</td>
<td>Initialises a ButtonSensor object and then returns it.</td>
</tr>
<tr>
<td>easygopigo3.EasyGoPiGo3.init_line_follower([port])</td>
<td>Initialises a EasyLineFollower object and then returns it.</td>
</tr>
<tr>
<td>easygopigo3.EasyGoPiGo3.init_servo([port])</td>
<td>Initialises a Servo object and then returns it.</td>
</tr>
</tbody>
</table>

Continued on next page
Table 4 – continued from previous page

GoPiGo3 Documentation

easygopigo3.EasyGoPiGo3.
init_distance_sensor([port])
Initialises a EasyDistanceSensor object and then returns it.
easygopigo3.EasyGoPiGo3.
init_dht_sensor([...])
Initialises a DHTSensor object and then returns it.
easygopigo3.EasyGoPiGo3.
init_remote([port])
Initialises a Remote object and then returns it.
easygopigo3.EasyGoPiGo3.
init_motion_sensor([port])
Initialises a MotionSensor object and then returns it.

Sensor/Actuator

easysensors.LoudnessSensor([port, gpg, ...])
Class for the Grove Loudness Sensor.
easysensors.Buzzer([port, gpg, use_mutex])
Class for the Grove Buzzer.
easysensors.Buzzer.sound(freq)
Sets a musical note for the Grove Buzzer.
easysensors.Buzzer.sound_off()
Turns off the Grove Buzzer.
easysensors.Buzzer.sound_on()
Turns on the Grove Buzzer at the set frequency.
easysensors.Led([port, gpg, use_mutex])
Class for the Grove LED.
easysensors.Led.light_on(power)
Sets the duty cycle for the Grove LED.
easysensors.Led.light_max()
Turns on the Grove LED at full power.
easysensors.Led.light_off()
Turns off the Grove LED.
easysensors.Led.is_on()
Checks if the Grove LED is turned on.
easysensors.Led.is_off()
Checks if the Grove LED is turned off.
easysensors.MotionSensor([port, gpg, use_mutex])
Class for the Grove Motion Sensor.
easysensors.MotionSensor.
motion_detected([port])
Checks if the Grove Motion Sensor detects a motion.
easysensors.ButtonSensor([port, gpg, use_mutex])
Class for the Grove Button.
is_button_pressed()
Checks if the Grove Button is pressed.
easysensors.Servo([port, gpg, use_mutex])
Class for controlling servo motors with the GoPiGo3 robot.
easysensors.Servo.
rotate_servo(servoposition)
Rotates the servo at a specific angle.
easysensors.Servo.reset_servo()
Resets the servo straight ahead, in the middle position.
easysensors.Servo.disable_servo()
Disable (or “float”) the servo.
easysensors.Remote([port, gpg, use_mutex])
Class for interfacing with the Infrared Receiver.
easysensors.Remote.read()
Reads the numeric code received by the Infrared Receiver.
easysensors.Remote.get_remote_code()
Returns the symbol of the pressed key in a string format.

Retired Sensor/Actuator

easysensors.LightSensor([port, gpg, use_mutex])
Class for the Grove Light Sensor.
easysensors.SoundSensor([port, gpg, use_mutex])
Class for the Grove Sound Sensor.

Continued on next page
Table 6 – continued from previous page

| easysensors.UltraSonicSensor((port, gpg, ...)) | Class for the Grove Ultrasonic Sensor. |
| easysensors.UltraSonicSensor.is_too_close() | Checks whether the Grove Ultrasonic Sensor measures a distance that’s too close to a target than what we consider a safe distance. |
| easysensors.UltraSonicSensor.get_safe_distance() | Gets what we call the safe distance for the Grove Ultrasonic Sensor. |
| easysensors.UltraSonicSensor.read_mm() | Measures the distance from a target in millimeters. |
| easysensors.UltraSonicSensor.read_inches() | Measures the distance from a target in inches. |
| easysensors.UltraSonicSensor.read() | Measures the distance from a target in centimeters. |
| easysensors.DHTSensor((gpg, sensor_type, ...)) | Class for interfacing with the Grove DHT Sensor. |
| easysensors.DHTSensor.read_temperature() | Return the temperature in Celsius degrees. |
| easysensors.DHTSensor.read_humidity() | Return the humidity as a percentage. |
| easysensors.DHTSensor.read() | Return the temperature and humidity. |

Base Sensor

| easysensors.Sensor(port, pinmode, gpg[...]) | Base class for all sensors. |
| easysensors.Sensor.__str__() | Prints out a short summary of the class-instantiated object’s attributes. |
| easysensors.Sensor.set_pin(pin) | Selects one of the 2 available pins of the grove connector. |
| easysensors.Sensor.get_pin() | Tells us which pin of the grove connector is used. |
| easysensors.Sensor.set_port(port) | Sets the port that’s going to be used by our new device. |
| easysensors.Sensor.get_port() | Gets the current port our device is connected to. |
| easysensors.Sensor.get_port_ID() | Gets the ID of the port we’re set to. |
| easysensors.Sensor.set_pin_mode(pinmode) | Sets the pin mode of the port we’re set to. |
| easysensors.Sensor.get_pin_mode() | Gets the pin mode of the port we’re set to. |
| easysensors.Sensor.set_descriptor(descriptor) | Sets the object’s description. |
| easysensors.AnalogSensor(port, pinmode, gpg[...]) | Class for analog devices with input/output capabilities on the GoPiGo3 robot. |
| easysensors.AnalogSensor.read() | Reads analog value of the sensor that’s connected to our GoPiGo3 robot. |
| easysensors.AnalogSensor.percent_read() | Reads analog value of the sensor that’s connected to our GoPiGo3 robot as a percentage. |
| easysensors.AnalogSensor.write_freq(freq) | Sets the frequency of the PWM signal. |
5.2 API - Library - EasyGoPiGo3

5.2.1 EasyGoPiGo3

class easygopigo3.EasyGoPiGo3(config_file_path='/home/pi/Dexter/gpg3_config.json', use_mutex=False)
Bases: gopigo3.GoPiGo3

This class is used for controlling a GoPiGo3 robot.

With this class you can do the following things with your GoPiGo3:

• Drive your robot in any number of directions.
• Have precise control over the direction of the robot.
• Set the speed of the robot.
• Turn on or off the blinker LEDs.
• Control the GoPiGo3’ Dex’s eyes, color and so on …

Warning: Without a battery pack connected to the GoPiGo3, the robot won’t move.

__init__(config_file_path='/home/pi/Dexter/gpg3_config.json', use_mutex=False)
This constructor sets the variables to the following values:

Parameters

• config_file_path = "/home/pi/Dexter/gpg3_config.json" (str) – Path to JSON config file that stores the wheel diameter and wheel base width for the GoPiGo3.
• use_mutex = False (bool) – When using multiple threads/processes that access the same resource/device, mutex has to be enabled.

Variables

• speed = 300 (int) – The speed of the motors should go between 0-1000 DPS.
• left_eye_color = (0,255,255) (tuple(int,int,int)) – Set Dex’s left eye color to turquoise.
• right_eye_color = (0,255,255) (tuple(int,int,int)) – Set Dex’s right eye color to turquoise.
• DEFAULT_SPEED = 300 (int) – Starting speed value: not too fast, not too slow.

Raises

• IOError – When the GoPiGo3 is not detected. It also debugs a message in the terminal.
• gopigo3.FirmwareVersionError – If the GoPiGo3 firmware needs to be updated. It also debugs a message in the terminal.
• Exception – For any other kind of exceptions.

The config_file_path parameter represents the path to a JSON file. The presence of this configuration file is optional and is only required in cases where the GoPiGo3 has a skewed trajectory due to minor differences in these two constants: the wheel diameter and the wheel base width. In most cases, this won’t be the case.
By-default, the constructor tries to read the `config_file_path` file and silently fails if something goes wrong: wrong permissions, non-existent file, improper key values and so on. To set custom values to these 2 constants, use `set_robot_constants()` method and for saving the constants to a file call `save_robot_constants()` method.

**load_robot_constants(config_file_path='/home/pi/Dexter/gpg3_config.json')**

Load wheel diameter and wheel base width constants for the GoPiGo3 from file.

This method gets called by the constructor.

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>config_file_path</td>
<td>Path to JSON config file that stores the wheel diameter and wheel base width for the GoPiGo3.</td>
</tr>
</tbody>
</table>

**Raises**

- `FileNotFoundException` – When the file is non-existent.
- `KeyError` – If one of the keys is not part of the dictionary.
- `ValueError` – If the saved values are not positive numbers (floats or ints).
- `TypeError` – If the saved values are not numbers.
- `IOError` – When the file cannot be accessed.
- `PermissionError` – When there are not enough permissions to access the file.
- `json.JSONDecodeError` – When the config file fails at parsing.

Here’s how the JSON config file must look like before reading it. Obviously, the supported format is JSON so that anyone can come in and edit their own config file if they don’t want to go through saving the values by using the API.

```json
{
    "wheel-diameter": 66.5,
    "wheel-base-width": 117
}
```

**save_robot_constants(config_file_path='/home/pi/Dexter/gpg3_config.json')**

Save the current wheel diameter and wheel base width constants (from within this object’s context) for the GoPiGo3 to file for future use.

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>config_file_path</td>
<td>Path to JSON config file that stores the wheel diameter and wheel base width for the GoPiGo3.</td>
</tr>
</tbody>
</table>

**Raises**

- `IOError` – When the file cannot be accessed.
- `PermissionError` – When there are not enough permissions to create the file.

Here’s how the JSON config file will end up looking like. The values can differ from case to case.

```json
{
    "wheel-diameter": 66.5,
    "wheel-base-width": 117
}
```

**set_robot_constants(wheel_diameter, wheel_base_width)**

Set new wheel diameter and wheel base width values for the GoPiGo3.

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>wheel_diameter</td>
<td>New wheel diameter value.</td>
</tr>
<tr>
<td>wheel_base_width</td>
<td>New wheel base width value.</td>
</tr>
</tbody>
</table>
GoPiGo3 Documentation

- **wheel_diameter** *(float)* – Diameter of the GoPiGo3 wheels as measured in millimeters.
- **wheel_base_width** *(float)* – The distance between the 2 centers of the 2 wheels as measured in millimeters.

This should only be required in rare cases when the GoPiGo3’s trajectory is skewed due to minor differences in the wheel-to-body measurements.

The GoPiGo3 class instantiates itself with default values for both constants:

1. **wheel_diameter** is by-default set to **66.5 mm**.
2. **wheel_base_width** is by-default set to **117 mm**.

**volt()**

This method returns the battery voltage of the GoPiGo3.

**Returns** The battery voltage of the GoPiGo3.

**Return type** float

**set_speed**(in_speed)

This method sets the speed of the GoPiGo3 specified by in_speed argument.

The speed is measured in **DPS = degrees per second** of the robot’s wheel(s).

**Parameters in_speed** *(int)* – The speed at which the robot is set to run - speed between 0-1000 DPS.

**Warning:** **0-1000** DPS are the preferred speeds for the GoPiGo3 robot. The speed variable can be basically set to any positive value, but factors like **voltage, load, battery amp rating**, etc, will determine the effective speed of the motors.

Experiments should be run by every user, as each case is unique.

**get_speed()**

Use this method for getting the speed of your GoPiGo3.

**Returns** The speed of the robot measured between **0-1000** DPS.

**Return type** int

**reset_speed()**

This method resets the speed to its original value.

**stop()**

This method stops the GoPiGo3 from moving. It brings the GoPiGo3 to a full stop.

**Note:** This method is used in conjunction with the following methods:

- **backward()**
- **right()**
- **left()**
- **forward()**

**forward()**

Move the GoPiGo3 forward.
For setting the motor speed, use `set_speed()`. Default speed is set to 300 - see `__init__()`.

**drive_cm** *(dist, blocking=True)*
Move the GoPiGo3 forward / backward for `dist` amount of centimeters.

For moving the GoPiGo3 robot forward, the `dist` parameter has to be *positive*.
For moving the GoPiGo3 robot backward, the `dist` parameter has to be *negative*.

**Parameters**
- `dist` *(float)* – The distance in cm the GoPiGo3 has to move.
- `blocking = True` *(boolean)* – Set it as a blocking or non-blocking method.

`blocking` parameter can take the following values:
- *True* so that the method will wait for the GoPiGo3 robot to finish moving.
- *False* so that the method will exit immediately while the GoPiGo3 robot will continue moving.

**drive_inches** *(dist, blocking=True)*
Move the GoPiGo3 forward / backward for `dist` amount of inches.

For moving the GoPiGo3 robot forward, the `dist` parameter has to be *positive*.
For moving the GoPiGo3 robot backward, the `dist` parameter has to be *negative*.

**Parameters**
- `dist` *(float)* – The distance in inches the GoPiGo3 has to move.
- `blocking = True` *(boolean)* – Set it as a blocking or non-blocking method.

`blocking` parameter can take the following values:
- *True* so that the method will wait for the GoPiGo3 robot to finish moving.
- *False* so that the method will exit immediately while the GoPiGo3 robot will continue moving.

**drive_degrees** *(degrees, blocking=True)*
Move the GoPiGo3 forward / backward for `degrees / 360` wheel rotations.

For moving the GoPiGo3 robot forward, the `degrees` parameter has to be *positive*.
For moving the GoPiGo3 robot backward, the `degrees` parameter has to be *negative*.

**Parameters**
- `degrees` *(float)* – Distance based on how many wheel rotations are made. Calculated by `degrees / 360`.
- `blocking = True` *(boolean)* – Set it as a blocking or non-blocking method.

`blocking` parameter can take the following values:
- *True* so that the method will wait for the GoPiGo3 robot to finish rotating.
- *False* so that the method will exit immediately while the GoPiGo3 robot will continue rotating.
For instance, the following function call is going to drive the GoPiGo3 robot forward for 310 / 360 wheel rotations, which equates to approximately 86% of the GoPiGo3’s wheel circumference.

```python
gpg3_obj.drive_degrees(310)
```

On the other hand, changing the polarity of the argument we’re passing, is going to make the GoPiGo3 robot move backward.

```python
gpg3_obj.drive_degrees(-30.5)
```

This line of code makes the GoPiGo3 robot go backward for 30.5 / 360 rotations, which is roughly 8.5% of the GoPiGo3’s wheel circumference.

**backward()**
Move the GoPiGo3 backward.
For setting the motor speed, use `set_speed()`. Default speed is set to 300 - see `__init__()`.

**right()**
Move the GoPiGo3 to the right.
For setting the motor speed, use `set_speed()`. Default speed is set to 300 - see `__init__()`.

**Important:**
The robot will activate only the left motor, whilst the right motor will be completely stopped. This causes the robot to rotate in very short circles.

**spin_right()**
Rotate the GoPiGo3 towards the right while staying on the same spot.
This differs from the `right()` method as both wheels will be rotating but in different directions. This causes the robot to spin in place, as if doing a pirouette.
For setting the motor speed, use `set_speed()`. Default speed is set to 300 - see `__init__()`.

**Important:** You can achieve the same effect by calling `steer(100, -100)` (method `steer()`).

**left()**
Move the GoPiGo3 to the left.
For setting the motor speed, use `set_speed()`. Default speed is set to 300 - see `__init__()`.

**Important:**
The robot will activate only the right motor, whilst the left motor will be completely stopped. This causes the robot to rotate in very short circles.

**spin_left()**
Rotate the GoPiGo3 towards the left while staying on the same spot.
This differs from the `left()` method as both wheels will be rotating but in different directions. This causes the robot to spin in place, as if doing a pirouette.
For setting the motor speed, use `set_speed()`. Default speed is set to 300 - see `__init__()`.
**Important:** You can achieve the same effect by calling `steer(-100, 100)` (method `steer()`).

**steer** *(left_percent, right_percent)*

Control each motor in order to get a variety of turning movements.

Each motor is assigned a percentage of the current speed value. While there is no limit on the values of `left_percent` and `right_percent` parameters, they are expected to be between -100 and 100.

**Parameters**

- **left_percent** *(int)* – Percentage of current speed value that gets applied to left motor. The range is between -100 (for backward rotation) and 100 (for forward rotation).
- **right_percent** *(int)* – Percentage of current speed value that gets applied to right motor. The range is between -100 (for backward rotation) and 100 (for forward rotation).

For setting the motor speed, use `set_speed()`. Default speed is set to 300 - see `__init__()`

**Important:** Setting both `left_percent` and `right_percent` parameters to 100 will result in the same behavior as the `forward()` method. The other behavior for `backward()` method will be experienced if both parameters are set to -100.

Setting both `left_percent` and `right_percent` to 0 will stop the GoPiGo from moving.

**orbit** *(degrees, radius_cm=0, blocking=True)*

Control the GoPiGo so it will orbit around an object.

**Parameters**

- **degrees** *(int)* – Degrees to steer. 360 for full rotation. Negative for left turn.
- **radius_cm** *(int)* – Radius in cm of the circle to drive. Default is 0 (turn in place).
- **blocking = True** *(boolean)* – Set it as a blocking or non-blocking method.

**Important:** Note that while in non-blocking mode the speed cannot be changed before the end of the orbit as it would negate all orbit calculations. After a non-blocking call, `set_speed()` has to be called before any other movement.

**target_reached** *(left_target_degrees, right_target_degrees)*

Checks if *(wheels have rotated for a given number of degrees)*:

- The left wheel has rotated for `left_target_degrees` degrees.
- The right wheel has rotated for `right_target_degrees` degrees.

If both conditions are met, it returns `True`, otherwise it’s `False`.

**Parameters**

- **left_target_degrees** *(int)* – Target degrees for the left wheel.
- **right_target_degrees** *(int)* – Target degrees for the right wheel.

**Returns** Whether both wheels have reached their target.

**Return type** `boolean`.

For checking if the GoPiGo3 robot has moved `forward` for 360 / 360 wheel rotations, we’d use the following code snippet.
# both variables are measured in degrees
left_motor_target = 360
right_motor_target = 360

# reset the encoders
gpg3_obj.reset_encoders()
# and make the robot move forward
gpg3_obj.forward()

while gpg3_obj.target_reached(left_motor_target, right_motor_target):
    # give the robot some time to move
    sleep(0.05)

# now lets stop the robot
# otherwise it would keep on going
# reset_encoders(blocking=True)
gpg3_obj.stop()

On the other hand, for moving the GoPiGo3 robot to the right for \( \frac{187}{360} \) wheel rotations of the left wheel, we’d use the following code snippet.

# both variables are measured in degrees
left_motor_target = 187
right_motor_target = 0

# reset the encoders
gpg3_obj.reset_encoders()
# and make the robot move to the right
gpg3_obj.right()

while gpg3_obj.target_reached(left_motor_target, right_motor_target):
    # give the robot some time to move
    sleep(0.05)

# now lets stop the robot
# otherwise it would keep on going
# reset_encoders(blocking=True)
gpg3_obj.stop()

**Note:** You *can* use this method in conjunction with the following methods:

- `drive_cm()`
- `drive_inches()`
- `drive_degrees()`

when they are used in *non-blocking* mode.

And almost *everytime* with the following ones:

- `backward()`
- `right()`
- `left()`
- `forward()`

**reset_encoders** *(blocking=True)*

Resets both the encoders back to 0.
Parameters blocking = True (boolean) – Set it as a blocking or non-blocking method.

blocking parameter can take the following values:

• True so that the method will wait for the GoPiGo3 motors to finish resetting.

• False so that the method will exit immediately while the GoPiGo3 motors will continue to reset. Sending another motor command to the motors while the reset is under way might lead to confusing behavior.

read_encoders ()

Reads the encoders’ position in degrees. 360 degrees represent 1 full rotation (or 360 degrees) of a wheel.

Returns A tuple containing the position in degrees of each encoder. The 1st element is for the left motor and the 2nd is for the right motor.

Return type tuple(int, int)

read_encoders_average (units=’cm’)

Reads the encoders’ position in degrees. 360 degrees represent 1 full rotation (or 360 degrees) of a wheel.

Parameters units (string) – By default it’s “cm”, but it could also be “in” for inches. Anything else will return raw encoder average.

Returns The average of the two wheel encoder values.

Return type int

turn_degrees (degrees, blocking=True)

Makes the GoPiGo3 robot turn at a specific angle while staying in the same spot.

Parameters

• degrees (float) – The angle in degress at which the GoPiGo3 has to turn. For rotating the robot to the left, degrees has to be negative, and make it turn to the right, degrees has to be positive.

• blocking = True (boolean) – Set it as a blocking or non-blocking method.

blocking parameter can take the following values:

• True so that the method will wait for the GoPiGo3 robot to finish moving.

• False so that the method will exit immediately while the GoPiGo3 robot will continue moving.

In order to better understand what does this method do, let’s take a look at the following graphical representation.

In the image, we have multiple identifiers:

• The “heading”: it represents the robot’s heading. By default, “rotating” the robot by 0 degrees is going to make the robot stay in place.

• The “wheel circle circumference”: this is the circle that’s described by the 2 motors moving in opposite direction.

• The “GoPiGo3”: the robot we’re playing with. The robot’s body isn’t draw in this representation as it’s not the main focus here.

• The “wheels”: these are just the GoPiGo3’s wheels - selfexplanatory.

The effect of this class method is that the GoPiGo3 will rotate in the same spot (depending on degrees parameter), while the wheels will be describing a perfect circle.
So, in order to calculate how much the motors have to spin, we divide the angle (at which we want to rotate the robot) by 360 degrees and we get a float number between 0 and 1 (think of it as a percentage). We then multiply this value with the wheel circle circumference (which is the circumference of the circle the robot’s wheels describe when rotating in the same place).

At the end we get the distance each wheel has to travel in order to rotate the robot by degrees.

**blinker_on** *(id)*

Turns ON one of the 2 red blinkers that GoPiGo3 has.

**Parameters**

id *(int/str)* – 0 / 1 for the right / left led or string literals can be used: "right" and "left".

**blinker_off** *(id)*

Turns OFF one of the 2 red blinkers that GoPiGo3 has.

**Parameters**

id *(int/str)* – 0 / 1 for the right / left led or string literals can be used: "right" and "left".

**led_on** *(id)*

Turns ON one of the 2 red blinkers that GoPiGo3 has. The same as **blinker_on()**.

**Parameters**

id *(int/str)* – 0 / 1 for the right / left led or string literals can be used: "right" and "left".

**led_off** *(id)*

Turns OFF one of the 2 red blinkers that GoPiGo3 has. The same as **blinker_off()**.

**Parameters**

id *(int/str)* – 0 / 1 for the right / left led or string literals can be used: "right" and "left".

**set_left_eye_color** *(color)*

Sets the LED color for Dexter mascot’s left eye.

**Parameters**

color *(tuple(int, int, int)) – 8-bit RGB tuple that represents the left eye’s color.

**Raises** **TypeError** – When color parameter is not valid.

**Important:** After setting the eye’s color, call **open_left_eye()** or **open_eyes()** to update the color, or otherwise the left eye’s color won’t change.

**set_right_eye_color** *(color)*

Sets the LED color for Dexter mascot’s right eye.

**Parameters**

color *(tuple(int, int, int)) – 8-bit RGB tuple that represents the right eye’s color.

**Raises** **TypeError** – When color parameter is not valid.

**Important:** After setting the eye’s color, call **open_right_eye()** or **open_eyes()** to update the color, or otherwise the right eye’s color won’t change.

**set_eye_color** *(color)*

Sets the LED color for Dexter mascot’s eyes.

**Parameters**

color *(tuple(int, int, int)) – 8-bit RGB tuple that represents the eyes’ color.

**Raises** **TypeError** – When color parameter is not valid.
Important: After setting the eyes’ color, call `open_eyes()` to update the color of both eyes, or otherwise the color won’t change.

```python
open_left_eye()
    Turns ON Dexter mascot’s left eye.

open_right_eye()
    Turns ON Dexter mascot’s right eye.

open_eyes()
    Turns ON Dexter mascot’s eyes.

close_left_eye()
    Turns OFF Dexter mascot’s left eye.

close_right_eye()
    Turns OFF Dexter mascot’s right eye.

close_eyes()
    Turns OFF Dexter mascot’s eyes.

init_light_sensor(port='AD1')
    Initialises a LightSensor object and then returns it.

    Parameters port = "AD1" (str) – Can be either "AD1" or "AD2". By default it’s set to be "AD1".

    Returns An instance of the LightSensor class and with the port set to port’s value.

    The "AD1" and "AD2" ports are mapped to the following Hardware Ports.

    The use_mutex parameter of the __init__() constructor is passed down to the constructor of LightSensor class.

init_sound_sensor(port='AD1')
    Initialises a SoundSensor object and then returns it.

    Parameters port = "AD1" (str) – Can be either "AD1" or "AD2". By default it’s set to be "AD1".

    Returns An instance of the SoundSensor class and with the port set to port’s value.

    The "AD1" and "AD2" ports are mapped to the following Hardware Ports.

    The use_mutex parameter of the __init__() constructor is passed down to the constructor of SoundSensor class.

init_loudness_sensor(port='AD1')
    Initialises a LoudnessSensor object and then returns it.

    Parameters port = "AD1" (str) – Can be either "AD1" or "AD2". By default it’s set to be "AD1".

    Returns An instance of the LoudnessSensor class and with the port set to port’s value.

    The "AD1" and "AD2" ports are mapped to the following Hardware Ports.

    The use_mutex parameter of the __init__() constructor is passed down to the constructor of LoudnessSensor class.

init_ultrasonic_sensor(port='AD1')
    Initialises a UltraSonicSensor object and then returns it.
Parameters `port` = "AD1" (`str`) – Can be either "AD1" or "AD2". By default it’s set to be "AD1".

Returns An instance of the `UltraSonicSensor` class and with the port set to `port`’s value.

The "AD1" and "AD2" ports are mapped to the following `Hardware Ports`.

The `use_mutex` parameter of the `__init__()` constructor is passed down to the constructor of `UltraSonicSensor` class.

`init_buzzer` (`port='AD1'`)  
Initialises a `Buzzer` object and then returns it.

Parameters `port` = "AD1" (`str`) – Can be either "AD1" or "AD2". By default it’s set to be "AD1".

Returns An instance of the `Buzzer` class and with the port set to `port`’s value.

The "AD1" and "AD2" ports are mapped to the following `Hardware Ports`.

The `use_mutex` parameter of the `__init__()` constructor is passed down to the constructor of `Buzzer` class.

`init_led` (`port='AD1'`)  
Initialises a `Led` object and then returns it.

Parameters `port` = "AD1" (`str`) – Can be either "AD1" or "AD2". By default it’s set to be "AD1".

Returns An instance of the `Led` class and with the port set to `port`’s value.

The "AD1" and "AD2" ports are mapped to the following `Hardware Ports`.

The `use_mutex` parameter of the `__init__()` constructor is passed down to the constructor of `Led` class.

`init_button_sensor` (`port='AD1'`)  
Initialises a `ButtonSensor` object and then returns it.

Parameters `port` = "AD1" (`str`) – Can be either "AD1" or "AD2". By default it’s set to be "AD1".

Returns An instance of the `ButtonSensor` class and with the port set to `port`’s value.

The "AD1" and "AD2" ports are mapped to the following `Hardware Ports`.

The `use_mutex` parameter of the `__init__()` constructor is passed down to the constructor of `ButtonSensor` class.

`init_line_follower` (`port='I2C'`)  
Initialises a `EasyLineFollower` object and then returns it.

Parameters `port` = "I2C" (`str`) – The only option for this parameter for the Line Follower Sensor (red board) is "I2C" and for the Line Follower Sensor (black board) it can also be "AD1"/"AD2". The default value for this parameter is already set to "I2C".

Returns An instance of the `EasyLineFollower` class and with the port set to `port`’s value.

The "I2C", "AD1" and "AD2" ports are mapped to the following `Hardware Ports`.

Tip:

The sensor can be connected to any of the I2C ports.

You can connect different I2C devices simultaneously provided that:
• The I2C devices have different addresses.

• The I2C devices are recognizable by the GoPiGo3 platform.

The `use_mutex` parameter of the `__init__()` constructor is passed down to the constructor of `EasyLineFollower` class.

`init_servo (port='SERVO1')`
Initialises a `Servo` object and then returns it.

Parameters

port = "SERVO1" (str) – Can be either "SERVO1" or "SERVO2". By default it’s set to be "SERVO1".

Returns
An instance of the `Servo` class and with the port set to `port`’s value.

The "SERVO1" and "SERVO2" ports are mapped to the following Hardware Ports.

The `use_mutex` parameter of the `__init__()` constructor is passed down to the constructor of `Servo` class.

`init_distance_sensor (port='I2C')`
Initialises a `EasyDistanceSensor` object and then returns it.

Parameters

port = "I2C" (str) – The options for this parameter are "I2C" by default, "AD1", or "AD2".

Returns
An instance of the `EasyDistanceSensor` class and with the port set to `port`’s value.

 Raises `ImportError` – When the `di_sensors` module can’t be found. Check the DI-Sensors documentation on how to install the libraries.

The ports are mapped to the following Hardware Ports.

The `use_mutex` parameter of the `__init__()` constructor is passed down to the constructor of `EasyDistanceSensor` class.

Tip:
The sensor can be connected to any of the I2C ports.
You can connect different I2C devices simultaneously provided that:

• The I2C devices have different addresses.

• The I2C devices are recognizable by the GoPiGo3 platform.

If the devices share the same address, like two distance sensors for example, you can still use them with the GoPiGo3 provided at least one is connected via the "AD1", or "AD2", port.

`init_dht_sensor (sensor_type=0)`
Initialises a `DHTSensor` object and then returns it.

Parameters

sensor_type = 0 (int) – Choose `sensor_type = 0` when you have the blue-coloured DHT sensor or `sensor_type = 1` when it’s white.

Returns
An instance of the `DHTSensor` class and with the port set to `port`’s value.
The `use_mutex` parameter of the `__init__()` constructor is passed down to the constructor of `DHTSensor` class.

**Important:** The only port to which this device can be connected is the "SERIAL" port, so therefore, there's no need for a parameter which specifies the port of device because we've only got one available. The "SERIAL" port is mapped to the following *Hardware Ports*.

```python
init_remote(port='AD1')
```
Initialises a `Remote` object and then returns it.

- **Parameters**
  - `port = "AD1" (str)` – Can be set to either "AD1" or "AD2". Set by default to "AD1".
  - **Returns**
    - An instance of the `Remote` class and with the port set to `port`'s value.

  The "AD1" port is mapped to the following *Hardware Ports*.

  The `use_mutex` parameter of the `__init__()` constructor is passed down to the constructor of `Remote` class.

```python
init_motion_sensor(port='AD1')
```
Initialises a `MotionSensor` object and then returns it

- **Parameters**
  - `port = "AD1" (str)` – Can be set to either "AD1" or "AD2". Set by default to "AD1".
  - **Returns**
    - An instance of the `MotionSensor` class and with the port set to `port`'s value.

  The "AD1" port is mapped to the following *Hardware Ports*.

  The `use_mutex` parameter of the `__init__()` constructor is passed down to the constructor of `MotionSensor` class.

### 5.3 API - Library - Sensors

#### 5.3.1 LightSensor

```python
class easysensors.LightSensor(port='AD1', gpg=None, use_mutex=False)

Bases: easysensors.AnalogSensor
```

Class for the Grove Light Sensor.

This class derives from `AnalogSensor` class, so all of its attributes and methods are inherited. For creating a `LightSensor` object we need to call `init_light_sensor()` method like in the following examples.

```python
# create an EasyGoPiGo3 object
gpg3_obj = EasyGoPiGo3()

# and now instantiate a LightSensor object through the gpg3_obj object
light_sensor = gpg3_obj.init_light_sensor()

# do the usual stuff, like read the data of the sensor
value = light_sensor.read()
value_percentage = light_sensor.percent_read()

# take a look at AnalogSensor class for more methods and attributes
```
Or if we need to specify the port we want to use, we might do it like in the following example.

```python
# create an EasyGoPiGo3 object
gpg3_obj = EasyGoPiGo3()

# variable for holding the port to which we have the Light Sensor connected to
port = "AD2"

light_sensor = gpg3_obj.init_light_sensor(port)

# read the sensor the same way as in the previous example
```

See also:

For more sensors, please see our Dexter Industries shop.

```python
__init__(port='AD1', gpg=None, use_mutex=False)

Constructor for initializing a LightSensor object for the Grove Light Sensor.

Parameters

- **port** = "AD1" *(str)* – Port to which we have the Grove Light Sensor connected to.
- **gpg** = None *(easygopigo3.EasyGoPiGo3)* – EasyGoPiGo3 object used for instantiating a LightSensor object.
- **use_mutex** = False *(bool)* – When using multiple threads/processes that access the same resource/device, mutexes should be enabled.

Raises **TypeError** – If the gpg parameter is not a EasyGoPiGo3 object.

The port parameter can take the following values:

- "AD1" - general purpose input/output port.
- "AD2" - general purpose input/output port.

The ports’ locations can be seen in the following graphical representation: Hardware Ports.

### 5.3.2 SoundSensor

```python
class easysensors.SoundSensor(port='AD1', gpg=None, use_mutex=False)

Bases: easysensors.AnalogSensor

Class for the Grove Sound Sensor.

This class derives from AnalogSensor class, so all of its attributes and methods are inherited. For creating a SoundSensor object we need to call `init_sound_sensor()` method like in the following examples.

```python
# create an EasyGoPiGo3 object
gpg3_obj = EasyGoPiGo3()

# and now instantiate a SoundSensor object through the gpg3_obj object
sound_sensor = gpg3_obj.init_sound_sensor()

# do the usual stuff, like read the data of the sensor
value = sound_sensor.read()
value_percentage = sound_sensor.percent_read()

# take a look at AnalogSensor class for more methods and attributes
```
Or if we need to specify the port we want to use, we might do it like in the following example.

```python
# create an EasyGoPiGo3 object
gpg3_obj = EasyGoPiGo3()

# variable for holding the port to which we have the sound sensor connected to
port = "AD1"

sound_sensor = gpg3_obj.init_sound_sensor(port)

# read the sensor the same way as in the previous example
```

See also:

For more sensors, please see our Dexter Industries shop.

```python
__init__(port='AD1', gpg=None, use_mutex=False)
```

Constructor for initializing a `SoundSensor` object for the Grove Sound Sensor.

**Parameters**

- **port** = "AD1" (str) – Port to which we have the Grove Sound Sensor connected to.
- **gpg** = None (easygopigo3.EasyGoPiGo3) – `EasyGoPiGo3` object used for instantiating a `SoundSensor` object.
- **use_mutex** = False (bool) – When using multiple threads/processes that access the same resource/device, mutexes should be enabled.

**Raises** `TypeError` – If the `gpg` parameter is not a `EasyGoPiGo3` object.

The `port` parameter can take the following values:

- "AD1" - general purpose input/output port.
- "AD2" - general purpose input/output port.

The ports’ locations can be seen in the following graphical representation: Hardware Ports.

### 5.3.3 LoudnessSensor

```python
class easysensors.LoudnessSensor (port='AD1', gpg=None, use_mutex=False)
```

Bases: easysensors.AnalogSensor

Class for the Grove Loudness Sensor.

This class derives from `AnalogSensor` class, so all of their attributes and methods are inherited. For creating a `LoudnessSensor` object we need to call `init_loudness_sensor()` method like in the following examples.

```python
# create an EasyGoPiGo3 object
gpg3_obj = EasyGoPiGo3()

# and now instantiate a LoudnessSensor object through the gpg3_obj object
loudness_sensor = gpg3_obj.init_loudness_sensor()

# do the usual stuff, like read the data of the sensor
value = loudness_sensor.read()
value_percentage = loudness_sensor.percent_read()
```

(continues on next page)
Or if we need to specify the port we want to use, we might do it like in the following example.

```python
# create an EasyGoPiGo3 object
gpg3_obj = EasyGoPiGo3()

# variable for holding the port to which we have the sound sensor connected to
port = "AD1"

loudness_sensor = gpg3_obj.init_loudness_sensor(port)

# read the sensor the same way as in the previous example
```

See also:

For more sensors, please see our Dexter Industries shop.

### `__init__` (port='AD1', gpg=None, use_mutex=False)

Constructor for initializing a `LoudnessSensor` object for the Grove Loudness Sensor.

**Parameters**

- **port** = "AD1" *(str)* – Port to which we have the Grove Loudness Sensor connected to.
- **gpg** = None *(easygopigo3.EasyGoPiGo3)* – EasyGoPiGo3 object used for instantiating a `LoudnessSensor` object.
- **use_mutex** = False *(bool)* – When using multiple threads/processes that access the same resource/device, mutexes should be enabled.

**Raises** `TypeError` – If the `gpg` parameter is not a `EasyGoPiGo3` object.

The `port` parameter can take the following values:

- "AD1" - general purpose input/output port.
- "AD2" - general purpose input/output port.

The ports’ locations can be seen in the following graphical representation: *Hardware Ports*.

#### 5.3.4 UltrasonicSensor

```python
class easysensors.UltraSonicSensor (port='AD1', gpg=None, use_mutex=False)
Bases: easysensors.AnalogSensor
```

Class for the Grove Ultrasonic Sensor.

This class derives from `AnalogSensor` class, so all of its attributes and methods are inherited. For creating a `UltraSonicSensor` object we need to call `init_ultrasonic_sensor()` method like in the following examples.

```python
# create an EasyGoPiGo3 object
gpg3_obj = EasyGoPiGo3()

# and now instantiate a UltraSonicSensor object through the gpg3_obj object
```
ultrasonic_sensor = gpg3_obj.init_ultrasonic_sensor()

# do the usual stuff, like read the distance the sensor is measuring
distance_cm = ultrasonic_sensor.read()
distance_inches = ultrasonic_sensor.read_inches()

# take a look at AnalogSensor class for more methods and attributes

Or if we need to specify the port we want to use, we might do it like in the following example.

# create an EasyGoPiGo3 object
gpg3_obj = EasyGoPiGo3()

# variable for holding the port to which we have the ultrasonic sensor connected
port = "AD1"

ultrasonic_sensor = gpg3_obj.init_ultrasonic_sensor(port)

# read the sensor’s measured distance as in the previous example

See also:

For more sensors, please see our Dexter Industries shop.

__init__(port='AD1', gpg=None, use_mutex=False)

    Constructor for initializing a UltraSonicSensor object for the Grove Ultrasonic Sensor.

    Parameters

    • port = "AD1" (str) – Port to which we have the Grove Ultrasonic Sensor connected to.
    • gpg = None (easygopigo3.EasyGoPiGo3) – EasyGoPiGo3 object used for instantiating a UltraSonicSensor object.
    • use_mutex = False (bool) – When using multiple threads/processes that access the same resource/device, mutexes should be enabled.

    Raises TypeError – If the gpg parameter is not a EasyGoPiGo3 object.

The port parameter can take the following values:

• "AD1" - general purpose input/output port.
• "AD2" - general purpose input/output port.

The ports’ locations can be seen in the following graphical representation: Hardware Ports.

is_too_close()

    Checks whether the Grove Ultrasonic Sensor measures a distance that’s too close to a target than what we consider a safe distance.

    Returns Whether the Grove Ultrasonic Sensor is too close from a target.

    Return type bool

    Raises gopigo3.SensorError – If a sensor is not yet configured when trying to read it.

A safe distance can be set with the set_safe_distance() method.
Note: The default safe distance is set at 50 cm.

```python
set_safe_distance(dist)
```
Sets a safe distance for the Grove Ultrasonic Sensor.

**Parameters**
- `dist (int)`: Minimum distance from a target that we can call a safe distance.

To check whether the robot is too close from a target, please check the `is_too_close()` method.

Note: The default safe distance is set at 50 cm.

```python
get_safe_distance()
```
Gets what we call the safe distance for the Grove Ultrasonic Sensor.

**Returns**
The minimum distance from a target that can be considered a safe distance.

**Return type**
`int`

Note: The default safe distance is set at 50 cm.

```python
read_mm()
```
Measures the distance from a target in millimeters.

**Returns**
The distance from a target in millimeters.

**Return type**
`int`

**Raises**
- `gopigo3.ValueError`: If trying to read an invalid value.
- `Exception`: If any other error occurs.

**Important:**
- This method can read distances between 15-4300 millimeters.
- This method will read the data for 3 times and it’ll discard anything that’s smaller than 15 millimeters and bigger than 4300 millimeters.
- If data is discarded 5 times (due to a communication error with the sensor), then the method returns 5010.

```python
read()
```
Measures the distance from a target in centimeters.

**Returns**
The distance from a target in centimeters.

**Return type**
`int`

**Important:**
- This method can read distances between 2-430 centimeters.
- If data is discarded 5 times (due to a communication error with the sensor), then the method returns 501.
read_inches()

Measures the distance from a target in inches.

Returns  The distance from a target in inches.

Return type  float (one decimal)

Important:

• This method can read distances of up to 169 inches.

• If data is discarded 5 times (due to a communication error with the sensor), then the method returns 501.

5.3.5 Buzzer

class easysensors.Buzzer (port='AD1', gpg=None, use_mutex=False)

Bases: easysensors.AnalogSensor

Class for the Grove Buzzer.

This class derives from AnalogSensor class, so all of its attributes and methods are inherited. For creating a Buzzer object we need to call init_buzzer() method like in the following examples.

```python
# create an EasyGoPiGo3 object
gpg3_obj = EasyGoPiGo3()

# and now instantiate a UltraSonicSensor object through the gpg3_obj object
buzzer = gpg3_obj.init_buzzer()

# turn on and off the buzzer
buzzer.sound_on()
sleep(1)
buzzer.sound_off()

# take a look at AnalogSensor class for more methods and attributes
```

If we need to specify the port we want to use, we might do it like in the following example.

```python
# create an EasyGoPiGo3 object

gpg3_obj = EasyGoPiGo3()

# variable for holding the port to which we have the ultrasonic sensor connected.

port = "AD1"

buzzer = gpg3_obj.init_buzzer(port)
```

See also:

For more sensors, please see our Dexter Industries shop.

```makefile
```

Dictionary of frequencies for each musical note. For instance, scale["A3"] instruction is equal to 220 Hz (that’s the A3 musical note’s frequency). This dictionary is useful when we want to make the buzzer ring at certain frequencies (aka musical notes).
__init__ (port='AD1', gpg=None, use_mutex=False)
Constructor for initializing a Buzzer object for the Grove Buzzer.

Parameters

- **port** = "AD1" *(str)* – Port to which we have the Grove Buzzer connected to.
- **gpg** = None *(easygopigo3.EasyGoPiGo3)* – EasyGoPiGo3 object used for instantiating a Buzzer object.
- **use_mutex** = False *(bool)* – When using multiple threads/processes that access the same resource/device, mutexes should be enabled.

Variables

- **power** = 50 *(int)* – Duty cycle of the signal that’s put on the buzzer.
- **freq** = 329 *(int)* – Frequency of the signal that’s put on the buzzer. 329Hz is synonymous to E4 musical note. See scale for more musical notes.

Raises **TypeError** – If the **gpg** parameter is not a EasyGoPiGo3 object.

The **port** parameter can take the following values:

- "AD1" - general purpose input/output port.
- "AD2" - general purpose input/output port.

The ports’ locations can be seen in the following graphical representation: Hardware Ports.

**sound** *(freq)*
Sets a musical note for the Grove Buzzer.

**Parameters** **freq** *(int)* – The frequency of the signal that’s put on the Grove Buzzer.

For a list of musical notes, please see scale. See this example script on how to play musical notes.

```python
# initialize all the required objects and connect the sensor to the GoPiGo3

musical_notes = buzzer.scale
notes_i_want_to_play = {"F4#", "F4#", "C5#", "B3", "B3", "B3"}
wait_time = 1.0

for note in notes_i_want_to_play:
    buzzer.sound(musical_notes[note])
    sleep(wait_time)
# enjoy the musical notes
```

**sound_off** ()
Turns off the Grove Buzzer.

**sound_on** ()
Turns on the Grove Buzzer at the set frequency.

For changing the frequency, please check the **sound()** method.

### 5.3.6 Led

class easysensors.Led *(port='AD1', gpg=None, use_mutex=False)*
Bases: easysensors.AnalogSensor

Class for the Grove LED.
With this class the following things can be done:

- Turn ON/OFF an LED.
- Set a level of brightness for the LED.
- Check if an LED is turned ON or OFF.

This class derives from AnalogSensor class, so all of its attributes and methods are inherited. For creating a Led object we need to call init_led() method like in the following examples.

```python
# create an EasyGoPiGo3 object
gpg3_obj = EasyGoPiGo3()

# and now instantiate a Led object through the gpg3_obj object
led = gpg3_obj.init_led()

# turn on and off the buzzer
led.light_max()
sleep(1)
led.light_off()

# take a look at AnalogSensor class for more methods and attributes
```

If we need to specify the port we want to use, we might do it like in the following example.

```python
# create an EasyGoPiGo3 object
gpg3_obj = EasyGoPiGo3()

# variable for holding the port to which we have the led connected to
port = "AD1"

led = gpg3_obj.init_led(port)

# call some Led-specific methods
```

See also:

For more sensors, please see our Dexter Industries shop.

```python
__init__ (port='AD1', gpg=None, use_mutex=False)
Constructor for initializing a Led object for the Grove LED.

Parameters

- **port** = "AD1" (str) – Port to which we have the Grove LED connected to.
- **gpg** = None (easygopigo3.EasyGoPiGo3) – EasyGoPiGo3 object used for instantiating a Led object.
- **use_mutex** = False (bool) – When using multiple threads/processes that access the same resource/device, mutexes should be enabled.

Raises **TypeError** – If the gpg parameter is not a EasyGoPiGo3 object.
```

The `port` parameter can take the following values:

- "AD1" - general purpose input/output port.
- "AD2" - general purpose input/output port.

The ports' locations can be seen in the following graphical representation: Hardware Ports.
**light_on** (*power*)
Sets the duty cycle for the Grove LED.

**Parameters**
- **power** (*int*) – Number between 0 and 100 that represents the duty cycle of PWM signal.

**light_max** ()
Turns on the Grove LED at full power.

**light_off** ()
Turns off the Grove LED.

**is_on** ()
Checks if the Grove LED is turned on.

**Returns**
If the Grove LED is on.

**Return type**
bool

**is_off** ()
Checks if the Grove LED is turned off.

**Returns**
If the Grove LED is off.

**Return type**
bool

### 5.3.7 MotionSensor

**class** easysensors.MotionSensor (*port='AD1', gpg=None, use_mutex=False*)
**Bases:** easysensors.DigitalSensor

Class for the Grove Motion Sensor.

This class derives from Sensor (check for throwable exceptions) and DigitalSensor classes, so all attributes and methods are inherited. For creating a MotionSensor object we need to call `init_motion_sensor()` method like in the following examples.

```python
# create an EasyGoPiGo3 object
gpg3_obj = EasyGoPiGo3()

# and now instantiate a Motion Sensor object through the gpg3_obj object on default port AD1
motion_sensor = gpg3_obj.init_motion_sensor()

while True:
    if motion_sensor.motion_detected():
        print("motion detected")
    else:
        print("no motion")

# take a look at DigitalSensor & Sensor class for more methods and attributes
```

If we need to specify the port we want to use, we might do it like in the following example.

```python
# create an EasyGoPiGo3 object
gpg3_obj = EasyGoPiGo3()

# variable for holding the port to which we have the motion sensor connected to port = "AD2"
```
motion_sensor = gpg3_obj.init_motion_sensor(port)

See also:
For more sensors, please see our Dexter Industries shop.

__init__ (port='AD1', gpg=None, use_mutex=False)
Constructor for initializing a MotionSensor object for the Grove Motion Sensor.

Parameters

• port = "AD1" (str) – Port to which we have the Grove Motion Sensor connected to.
• gpg = None (easygopigo3.EasyGoPiGo3) – EasyGoPiGo3 object used for instantiating a MotionSensor object.
• use_mutex = False (bool) – When using multiple threads/processes that access the same resource/device, mutexes should be enabled.

Raises TypeError – If the gpg parameter is not a EasyGoPiGo3 object.

The port parameter can take the following values:

• "AD1" - general purpose input/output port.
• "AD2" - general purpose input/output port.

The ports’ locations can be seen in the following graphical representation: Hardware Ports.

motion_detected (port='AD1')
Checks if the Grove Motion Sensor detects a motion.

Returns True or False, if the Grove Motion Sensor detects a motion or not.

Return type bool

5.3.8 ButtonSensor

class easysensors.ButtonSensor (port='AD1', gpg=None, use_mutex=False)
Bases: easysensors.DigitalSensor

Class for the Grove Button.

This class derives from Sensor (check for throwable exceptions) and DigitalSensor classes, so all attributes and methods are inherited. For creating a ButtonSensor object we need to call init_button_sensor() method like in the following examples.

# create an EasyGoPiGo3 object
gpg3_obj = EasyGoPiGo3()

# and now instantiate a Button object through the gpg3_obj object
button = gpg3_obj.init_button_sensor()

while True:
    if button.is_button_pressed():
        print("button pressed")
    else:
        print("button released")

# take a look at DigitalSensor & Sensor class for more methods and attributes
If we need to specify the port we want to use, we might do it like in the following example.

```python
# create an EasyGoPiGo3 object
gpg3_obj = EasyGoPiGo3()

# variable for holding the port to which we have the button connected to
port = "AD1"

button = gpg3_obj.init_button_sensor(port)

# call some button-specific methods
```

See also:

For more sensors, please see our Dexter Industries shop.

```python
__init__(self, port='AD1', gpg=None, use_mutex=False)

Constructor for initializing a ButtonSensor object for the Grove Button.

Parameters

- **port** = "AD1" (str) – Port to which we have the Grove Button connected to.
- **gpg** = None (easygopigo3.EasyGoPiGo3) – EasyGoPiGo3 object used for instantiating a Button object.
- **use_mutex** = False (bool) – When using multiple threads/processes that access the same resource/device, mutexes should be enabled.

Raises TypeError – If the gpg parameter is not a EasyGoPiGo3 object.

The port parameter can take the following values:

- "AD1" - general purpose input/output port.
- "AD2" - general purpose input/output port.

The ports’ locations can be seen in the following graphical representation: Hardware Ports.

```python
is_button_pressed()

Checks if the Grove Button is pressed.

Returns True or False, if the Grove Button is pressed.

Return type bool
```

5.3.9 Servo

```python
class easysensors.Servo(port='SERVO1', gpg=None, use_mutex=False)
    Bases: easysensors.Sensor

Class for controlling servo motors with the GoPiGo robot. Allows you to rotate the servo by serving the angle of rotation.

This class is derived from Sensor class and because of this, it inherits all the attributes and methods.

For creating a Servo object we need to call init_servo() method like in the following examples.

```python
# create an EasyGoPiGo3 object
gpg3_obj = EasyGoPiGo3()

# and now let's instantiate a Servo object through the gpg3_obj object
```
# this will bind a servo to port "SERVO1"
servo = gpg3_obj.init_servo()

# rotate the servo at 160 degrees
servo.rotate_servo(160)

Or if we want to specify the port to which we connect the servo, we need to call `init_servo()` the following way.

```python
servo = gpg3_obj.init_servo("SERVO2")
```

See also:

For more sensors, please see our Dexter Industries shop.

```
__init__ (port='SERVO1', gpg=None, use_mutex=False)
    Constructor for instantiating a Servo object for a (or multiple) servo (servos).

    Parameters
    • port = "SERVO1" (str) – The port to which we have connected the servo.
    • gpg = None (easygopigo3.EasyGoPiGo3) – EasyGoPiGo3 object that we need for instantiation.
    • use_mutex = False (bool) – When using multiple threads/processes that access the same resource/device, mutexes should be enabled.

    Raises TypeError – If the gpg parameter is not a EasyGoPiGo3 object.
```

The available ports that can be used for a servo are:

- "SERVO1" - servo controller port.
- "SERVO2" - servo controller port.

To see where these 2 ports are located, please take a look at the following graphical representation: Hardware Ports.

```
rotate_servo (servo_position)
    Rotates the servo at a specific angle.

    Parameters servo_position (int) – Angle at which the servo has to rotate. The values can be anywhere from 0 to 180 degrees.

    The pulse width varies the following way:
    • 575 uS for 0 degrees - the servo’s default position.
    • 24250 uS for 180 degrees - where the servo is rotated at its maximum position.

    Each rotation of 1 degree requires an increase of the pulse width by 10.27 uS.
```

**Warning:**

We use PWM signals (Pulse Width Modulation), so the angle at which a servo will rotate will be case-dependent.

This means a servo’s 180 degrees position won’t be the same as with another servo.

```
reset_servo ()
    Resets the servo straight ahead, in the middle position.
```
Tip:
Same as calling `rotate_servo(90)`.
Read more about `rotate_servo()` method.

def disable_servo()
    Disable (or “float”) the servo.
    The effect of this command is that if you then try to rotate the servo manually, it won’t resist you, thus meaning that it’s not trying to hold a target position.

5.3.10 DHTSensor

class easysensors.DHTSensor (gpg=None, sensor_type=0, use_mutex=False)
    Bases: easysensors.Sensor

    Class for interfacing with the Grove DHT Sensor. This class derives from Sensor class, so all of its attributes and methods are inherited.

    We can create a DHTSensor object similar to how we create it in the following template.

    ```python
    # create an EasyGoPiGo3 object
gpg3_obj = EasyGoPiGo3()

    # and now let’s instantiate a DHTSensor object through the gpg3_obj object
dht_sensor = gpg3_obj.init_dht_sensor()

    # read values continuously and print them in the terminal
    while True:
        temp, hum = dht_sensor.read()
        print("temp = {:.1f} hum = {:.1f}".format(temp, hum))
    ```

    __init__(gpg=None, sensor_type=0, use_mutex=False)
    Constructor for creating a DHTSensor object which can be used for interfacing with the Grove DHT Sensor.

    Parameters

    - **gpg** = *None* (easygopigo3.EasyGoPiGo3) – Object that’s required for instantiating a DHTSensor object.

    - **sensor_type** = *0* (*int*) – Choose sensor_type = 0 when you have the blue-coloured DHT sensor or sensor_type = 1 when it’s white.

    - **use_mutex** = *False* (*bool*) – When using multiple threads/processes that access the same resource/device, mutexes have to be used.

    Raises

    Any of the Sensor constructor’s exceptions in case of error.

    read_temperature()
    Return the temperature in Celsius degrees.

    Returns

    The temperature in Celsius degrees.

    Return type

    float

    If the sensor isn’t plugged in, a "Bad reading, try again" message is returned. If there is a runtime error, then a "Runtime error" message is returned.

5.3. API - Library - Sensors
read_humidity()
    Return the humidity as a percentage.

    Returns  Return the humidity as a percentage number from 0% to 100%.
    Return type  float

    If the sensor isn’t plugged in, a "Bad reading, try again" message is returned. If there is a runtime error, then a "Runtime error" message is returned.

read()
    Return the temperature and humidity.

    Returns  The temperature and humidity as a tuple, where the temperature is the 1st element of the tuple and the humidity the 2nd.
    Return type  (float, float)

    If the sensor isn’t plugged in, a "Bad reading, try again" message is returned. If there is a runtime error, then a "Runtime error" message is returned.

5.3.11 Remote

class easysensors.Remote (port='AD1', gpg=None, use_mutex=False)
    Bases: easysensors.Sensor

    Class for interfacing with the Infrared Receiver.

    With this sensor, you can command your GoPiGo3 with an Infrared Remote.

    In order to create an object of this class, we would do it like in the following example.

```python
# initialize an EasyGoPiGo3 object
gpg3_obj = EasyGoPiGo3()

# now initialize a Remote object
remote_control = gpg3_obj.init_remote()

# read whatever command you want from the remote by using # the [remote_control] object

keycodes = ['up', 'left', 'ok', 'right', 'down', '1', '2', '3', '4', '5', '6', '7', '8', '9', '*', '0', '#']
    List for mapping the codes we get with the read() method to the actual symbols we see on the Infrared Remote.

    __init__ (port='AD1', gpg=None, use_mutex=False)
        Constructor for initializing a Remote object.

        Parameters

        • port = "AD1" (str) – The port to which we connect the Infrared Receiver.

        • gpg = None (easygopigo3.EasyGoPiGo3) – The EasyGoPiGo3 object that we need for instantiating this object.

        • use_mutex = False (bool) – When using multiple threads/processes that access the same resource/device, mutexes should be enabled.

        Raises TypeError – If the gpg parameter is not a EasyGoPiGo3 object.

    The "AD1" and "AD2" ports' location on the GoPiGo3 robot can be seen in the following graphical representation: Hardware Ports.
read()
   Reads the numeric code received by the Infrared Receiver.

   Returns  The numeric code of the symbol that was pressed on the Infrared Remote.
   Return type  int

The numeric code represents the index of the keycodes list. By accessing the keycodes elements with the numeric code, you get the symbol that was pressed on the Infrared Remote.

For only getting the symbol that was pressed on the Infrared Remote, please check the get_remote_code() method.

Warning: On SensorError exception:
   • "Invalid Reading" string is printed in the console.
   • The value of -1 is returned.

get_remote_code()
   Returns the symbol of the pressed key in a string format.

   Returns  The symbol that was pressed on the Infrared Remote.
   Return type  str

Check the keycodes list for seeing what strings this method can return. On error or when nothing is read, an empty string is returned.

5.3.12 LineFollower

easysensors.LineFollower (port='I2C', gpg=None, use_mutex=False)
   Returns an instantiated object of di_sensors.easy_line_follower.EasyLineFollower.

   This is a replacement function for the line follower class that used to be found here - it has not moved to di_sensors library.

   Parameters
      • port = "I2C" (str) – The port to which we have connected the line follower sensor. Can also be "AD1"/"AD2" for the ‘Line Follower’_sensor.
      • gpg = None (easygopigo3.EasyGoPiGo3) – This is no longer required. Just skip this parameter.
      • use_mutex = False (bool) – When using multiple threads/processes that access the same resource/device, mutexes should be enabled.

   Raises
      • ImportError – If the di_sensors library couldn’t be found.
      • IOError – If the line follower is not responding.

5.4 API - Deprecated

In earlier versions of this library, all classes of the easysensors module belonged to easygopigo3 module. This can potentially make scripts written on a older version of this not work anymore. In order to prevent this problem, we have created dummy functions of each class of the easysensors module and placed them in easygopigo3.
Since these functions take the same arguments as the classes that were in previously and because they return an object, the scripts that were built on that long-gone API still work.

We intend to support these “dummy” functions for a while before we retire them.

In summary, the classes that are now being instantiated from the easysensors can still be instantiated from easygopigo3 for a while. These classes are:

```python
easygopigo3.LightSensor([port, gpg, use_mutex])
Use easysensors.LightSensor instead

easygopigo3.SoundSensor([port, gpg, use_mutex])
Use easysensors.SoundSensor instead

easygopigo3.LoudnessSensor([port, gpg, ...])
Use easysensors.LoudnessSensor instead

easygopigo3.UltraSonicSensor([port, gpg, ...])
Use easysensors.UltraSonicSensor instead

easygopigo3.Buzzer([port, gpg, use_mutex])
Use easysensors.Buzzer instead

easygopigo3.Led([port, gpg, use_mutex])
Use easysensors.Led instead

easygopigo3.MotionSensor([port, gpg, use_mutex])
Use easysensors.MotionSensor instead

easygopigo3.ButtonSensor([port, gpg, use_mutex])
Use easysensors.ButtonSensor instead

easygopigo3.Remote([port, gpg, use_mutex])
Use easysensors.Remote instead

easygopigo3.LineFollower([port, gpg, use_mutex])
Use di_sensors.easy_line_follower.EasyLineFollower instead

easygopigo3.Servo([port, gpg, use_mutex])
Use easysensors.Servo instead

easygopigo3.DHTSensor([gpg, sensor_type, ...])
Use easysensors.DHTSensor instead
```

The only exception is the easygopigo3.DistanceSensor() which class no longer exists in either module. This surrogate function has the same arguments as the other classes and the only port available to use is the "I2C". If you wish to initialize an object for the Distance Sensor, a better way is just to use the DistanceSensor class of DI-Sensors library on which this library is based upon.
CHAPTER SIX

API GOPIGO3 - ADVANCED

6.1 Requirements

Before using this chapter’s classes, you need to be able to import the following module.

```python
import easygopigo3
```

If you have issues importing this module, then make sure that:

- You’ve followed the steps found in Getting Started guide.
- You have installed either Raspbian For Robots, the GoPiGo3 repository or the GoPiGo3 package (the pip package).
- You have the gopigo3 package installed by typing the command `pip freeze | grep gopigo3` on your Raspberry Pi’s terminal. If the package is installed, then a string with the `gopigo3==[x.y.z]` format will show up.

If you encounter issues that aren’t covered by our Getting Started guide or FAQ chapter, please head over to our forum. Take note that the easygopigo3 module depends on the underlying gopigo3 module. The base gopigo3 module contains lower level calls that would allow you finer control over your GoPiGo3 and its functionality is available to you once easygopigo3 is imported.

6.2 Sensor

```python
class easysensors.Sensor(port, pinmode, gpg, use_mutex=False)
```

Base class for all sensors. Can only be instantiated through the use of an EasyGoPiGo3 object.

It should only be used as a base class for any type of sensor. Since it contains methods for setting / getting the ports or the pinmode, it’s basically useless unless a derived class comes in and adds functionalities.

**Variables**

- **port (str)** – There’re 4 types of ports - analog, digital, I2C and serial ports. The string identifiers are mapped in the following graphical representation - Hardware Ports.
- **pinmode (str)** – Represents the mode of operation of a pin - can be a digital input/output, an analog input/output or even a custom mode which must defined in the GoPiGo3’s firmware.
- **pin (int)** – Each grove connector has 4 pins: GND, VCC and 2 signal pins that can be user-defined. This variable specifies which pin of these 2 signal pins is used.
• **portID** (*int*) – Depending on *port*s’s value, an ID is given to each port. This variable is not important to us.

• **descriptor** (*str*) – Represents the “informal” string representation of an instantiated object of this class.

• **gpg** (*EasyGoPiGo3*) – Object instance of the *EasyGoPiGo3* class.

**Note:** The classes which derive from this class are the following:

• DigitalSensor
• AnalogSensor
• Servo
• DHTSensor

And the classes which are found at 2nd level of inheritance from this class are:

• LightSensor
• SoundSensor
• LoudnessSensor
• UltraSonicSensor
• Buzzer
• Led
• ButtonSensor

---

**Warning:**

1. This class should only be used by the developers of the *GoPiGo3* platform.

2. The name of this class isn’t representative of the devices we connect to the *GoPiGo3* robot - we don’t only use this class for sensors, but for any kind of device that we can connect to the *GoPiGo3* robot.

---

**__init__** (*port, pinmode, gpg, use_mutex=False*)

Constructor for creating a connection to one of the available grove ports on the *GoPiGo3*.  

**Parameters**

• **port** (*str*) – Specifies the port with which we want to communicate / interact with. The string literals we can use for identifying a port are found in the following graphical drawing: *Hardware Ports*.

• **pinmode** (*str*) – The mode of operation of the pin we’re selecting.

• **gpg** (*easygopigo3.EasyGoPiGo3*) – An instantiated object of the *EasyGoPiGo3* class. We need this *EasyGoPiGo3* class for setting up the *GoPiGo3* robot’s pins.

**Raises** **TypeError** – If the gpg parameter is not a *EasyGoPiGo3* object.

The port parameter can take the following string values:

• "AD1" - for digital and analog pinmode’s.
• "AD2" - for digital and analog pinmode’s.
• "SERVO1" - for "OUTPUT" pinmode.
• "SERVO2" - for "OUTPUT" pinmode.
• "I2C" - the pinmode is irrelevant here.
• "SERIAL" - the pinmode is irrelevant here.

These ports’ locations can be seen in the following graphical representation - *Hardware Ports*.

The pinmode parameter can take the following string values:
• "INPUT" - for general purpose inputs. The GoPiGo3 has 12-bit ADCs.
• "DIGITAL_INPUT" - for digital inputs. The port can detect either 0 or 1.
• "OUTPUT" - for general purpose outputs.
• "DIGITAL_OUTPUT" - for digital outputs. The port can only be set to 0 or 1.
• "US" - that’s for the *UltraSonicSensor* which can be bought from our shop. Can only be used with ports "AD1" and "AD2".
• "IR" - that’s for the infrared receiver. Can only be used with ports "AD1" and "AD2".

**Warning:** At the moment, there’s no class for interfacing with the infrared receiver.

### reconfig_bus()
Sets the bus properly. Sometimes this needs to be done even after instantiation when two processes are trying to connect to the GoPiGo and one re-initialises the ports.

### __str__()
Prints out a short summary of the class-instantiated object’s attributes.

**Returns** A string with a short summary of the object’s attributes.

**Return type** `str`

The returned string is made of the following components:
• the descriptor’s description
• the port name
• the pin identifier
• the portID - see *set_port()* method.

Sample of returned string as shown in a terminal:

```
$ ultrasonic sensor on port AD1
$ pinmode OUTPUT
$ portID 3
```

### set_pin(pin)
Selects one of the 2 available pins of the grove connector.

**Parameters** `pin (int) – 1 for the exterior pin of the grove connector (aka SIG) or anything else for the interior one.**

### get_pin()
Tells us which pin of the grove connector is used.

**Returns** For exterior pins (aka SIG) it returns `gopigo3.GROVE_2_1` or `gopigo3.GROVE_2_2` and for interior pins (aka NC) it returns `gopigo3.GROVE_1_2` or `gopigo3.GROVE_1_2`.

6.2. Sensor
Return type int

set_port (port)
Sets the port that’s going to be used by our new device. Again, we can’t communicate with our device, because the class doesn’t have any methods for interfacing with it, so we need to create a derived class that does this.

Parameters port (str) – The port we’re connecting the device to. Take a look at the Hardware Ports’ locations.

Apart from this graphical representation of the ports’ locations (Hardware Ports locations), take a look at the list of ports in __init__()’s description.

get_port ()
Gets the current port our device is connected to.

Returns The current set port.

Return type str

Apart from this graphical representation of the ports’ locations (Hardware Ports locations), take a look at the list of ports in __init__()’s description.

get_port_ID ()
Gets the ID of the port we’re set to.

Returns The ID of the port we’re set to.

Return type int

See more about port IDs in Sensor’s description.

set_pin_mode (pinmode)
Sets the pin mode of the port we’re set to.

Parameters pinmode (str) – The pin mode of the port.

See more about pin modes in __init__()’s description.

get_pin_mode ()
Gets the pin mode of the port we’re set to.

Returns The pin mode of the port.

Return type str

See more about pin modes in __init__()’s description.

set_descriptor (descriptor)
Sets the object’s description.

Parameters descriptor (str) – The object’s description.

See more about class descriptors in Sensor’s description.

6.3 DigitalSensor

Note: Coming soon!
6.4 AnalogSensor

class easysensors.AnalogSensor (port, pinmode, gpg, use_mutex=False)
    Bases: easysensors.Sensor

Class for analog devices with input/output capabilities on the GoPiGo3 robot. This class is derived from Sensor class, so this means this class inherits all attributes and methods.

For creating an AnalogSensor object an EasyGoPiGo3 object is needed like in the following example.

```python
# initialize an EasyGoPiGo3 object first
gpg3_obj = EasyGoPiGo3()

# let's have an analog input sensor on "AD1" port
port = "AD1"
pinmode = "INPUT"

# instantiate an AnalogSensor object
# pay attention that we need the gpg3_obj
anlogsensor_obj = AnalogSensor(port, pinmode, gpg3_obj)

# for example
# read the sensor’s value as we have an analog sensor connected to "AD1" port
anlogsensor_obj.read()
```

**Warning:** The name of this class isn’t representative of the type of devices we connect to the GoPiGo3 robot. With this class, both analog sensors and actuators (output devices such as LEDs which may require controlled output voltages) can be connected.

**__init__** (port, pinmode, gpg, use_mutex=False)

Binds an analog device to the specified port with the appropriate pinmode mode.

**Parameters**

- **port (str)** – The port to which the sensor/actuator is connected.
- **pinmode (str)** – The pin mode of the device that’s connected to the GoPiGo3.
- **gpg (easygopigo3.EasyGoPiGo3)** – Required object for instantiating an AnalogSensor object.
- **use_mutex = False (bool)** – When using multiple threads/processes that access the same resource/device, mutexes should be enabled.

**Raises** **TypeError** – If the gpg parameter is not a EasyGoPiGo3 object.

The available port’s for use are the following:

- "AD1" - general purpose input/output port.
- "AD2" - general purpose input/output port.

The ports’ locations can be seen in the following graphical representation: Hardware Ports.

**Important:** Since the grove connector allows 2 signals to pass through 2 pins (not concurrently), we can select which pin to go with by using the set_pin() method. By default, we’re using pin 1, which corresponds to the exterior pin of the grove connector (aka SIG) and the wire is yellow.
read()
Reads analog value of the sensor that’s connected to our GoPiGo3 robot.

Returns 12-bit number representing the voltage we get from the sensor. Range goes from 0V-5V.

Return type int

Raises

• gopigo3.ValueError – If an invalid value was read.
• Exception – If any other errors happens.

percent_read()
Reads analog value of the sensor that’s connected to our GoPiGo3 robot as a percentage.

Returns Percentage mapped to 0V-5V range.

Return type int

write(power)
Generates a PWM signal on the selected port. Good for simulating an DAC convertor - for instance an LED is a good candidate.

Parameters power (int) – Number from 0 to 100 that represents the duty cycle as a percentage of the frequency’s period.

Tip: If the power parameter is out of the given range, the most close and valid value will be selected.

write_freq(freq)
Sets the frequency of the PWM signal.

The frequency range goes from 3Hz up to 48000Hz. Default value is set to 24000Hz (24kHz).

Parameters freq (int) – Frequency of the PWM signal.

See also:

Read more about this in gopigo3.GoPiGo3.set_grove_pwm_frequency()’s description.
7.1 Our collaborators

The following collaborators are ordered alphabetically:

- Matt Richardson - Github Account.
- Nicole Parrot - Github Account.
- Robert Lucian Chiriac - Github Account.
- Shoban Narayan - Github account.
FREQUENTLY ASKED QUESTIONS

For more questions, please head over to our Dexter Industries forum.
CHAPTER

NINE

INDICES AND TABLES

• genindex
• modindex
• search
<table>
<thead>
<tr>
<th>Symbols</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>init</strong>() (easygopigo3.EasyGoPiGo3 method)</td>
<td>32</td>
</tr>
<tr>
<td><strong>init</strong>() (easysensors.AnalogSensor method)</td>
<td>65</td>
</tr>
<tr>
<td><strong>init</strong>() (easysensors.ButtonSensor method)</td>
<td>55</td>
</tr>
<tr>
<td><strong>init</strong>() (easysensors.Buzzer method)</td>
<td>50</td>
</tr>
<tr>
<td><strong>init</strong>() (easysensors.DHTSensor method)</td>
<td>57</td>
</tr>
<tr>
<td><strong>init</strong>() (easysensors.Led method)</td>
<td>52</td>
</tr>
<tr>
<td><strong>init</strong>() (easysensors.LightSensor method)</td>
<td>45</td>
</tr>
<tr>
<td><strong>init</strong>() (easysensors.LoudnessSensor method)</td>
<td>47</td>
</tr>
<tr>
<td><strong>init</strong>() (easysensors.MotionSensor method)</td>
<td>54</td>
</tr>
<tr>
<td><strong>init</strong>() (easysensors.Remote method)</td>
<td>58</td>
</tr>
<tr>
<td><strong>init</strong>() (easysensors.Sensor method)</td>
<td>64</td>
</tr>
<tr>
<td><strong>init</strong>() (easysensors.Servo method)</td>
<td>56</td>
</tr>
<tr>
<td><strong>init</strong>() (easysensors.SoundSensor method)</td>
<td>46</td>
</tr>
<tr>
<td><strong>init</strong>() (easysensors.UltraSonicSensor method)</td>
<td>49</td>
</tr>
<tr>
<td><strong>str</strong>() (easysensors.Sensor method)</td>
<td>63</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>AnalogSensor (class in easysensors)</td>
<td>65</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>B</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>backward() (easygopigo3.EasyGoPiGo3 method)</td>
<td>36</td>
</tr>
<tr>
<td>blinker_off() (easygopigo3.EasyGoPiGo3 method)</td>
<td>40</td>
</tr>
<tr>
<td>blinker_on() (easygopigo3.EasyGoPiGo3 method)</td>
<td>40</td>
</tr>
<tr>
<td>ButtonSensor (class in easysensors)</td>
<td>54</td>
</tr>
<tr>
<td>Buzzer (class in easysensors)</td>
<td>50</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>close_eyes() (easygopigo3.EasyGoPiGo3 method)</td>
<td>41</td>
</tr>
<tr>
<td>close_left_eye() (easygopigo3.EasyGoPiGo3 method)</td>
<td>41</td>
</tr>
<tr>
<td>close_right_eye() (easygopigo3.EasyGoPiGo3 method)</td>
<td>41</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>D</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>DHTSensor (class in easysensors)</td>
<td>57</td>
</tr>
<tr>
<td>disable_servo() (easysensors.Servo method)</td>
<td>57</td>
</tr>
<tr>
<td>drive_cm() (easygopigo3.EasyGoPiGo3 method)</td>
<td>35</td>
</tr>
<tr>
<td>drive_degrees() (easygopigo3.EasyGoPiGo3 method)</td>
<td>35</td>
</tr>
<tr>
<td>drive_inches() (easygopigo3.EasyGoPiGo3 method)</td>
<td>35</td>
</tr>
<tr>
<td>E</td>
<td></td>
</tr>
<tr>
<td>EasyGoPiGo3 (class in easygopigo3)</td>
<td>32</td>
</tr>
<tr>
<td>F</td>
<td></td>
</tr>
<tr>
<td>forward() (easygopigo3.EasyGoPiGo3 method)</td>
<td>34</td>
</tr>
<tr>
<td>G</td>
<td></td>
</tr>
<tr>
<td>get_pin() (easysensors.Sensor method)</td>
<td>63</td>
</tr>
<tr>
<td>get_pin_mode() (easysensors.Sensor method)</td>
<td>64</td>
</tr>
<tr>
<td>get_port() (easysensors.Sensor method)</td>
<td>64</td>
</tr>
<tr>
<td>get_port_ID() (easysensors.Sensor method)</td>
<td>64</td>
</tr>
<tr>
<td>get_remote_code() (easysensors.Remote method)</td>
<td>59</td>
</tr>
<tr>
<td>get_safe_distance() (easysensors.UltraSonicSensor method)</td>
<td>49</td>
</tr>
<tr>
<td>get_speed() (easygopigo3.EasyGoPiGo3 method)</td>
<td>34</td>
</tr>
<tr>
<td>I</td>
<td></td>
</tr>
<tr>
<td>init_button_sensor() (easygopigo3.EasyGoPiGo3 method)</td>
<td>42</td>
</tr>
<tr>
<td>init_buzzer() (easygopigo3.EasyGoPiGo3 method)</td>
<td>42</td>
</tr>
<tr>
<td>init_dht_sensor() (easygopigo3.EasyGoPiGo3 method)</td>
<td>42</td>
</tr>
<tr>
<td>init_distance_sensor() (easygopigo3.EasyGoPiGo3 method)</td>
<td>43</td>
</tr>
<tr>
<td>init_led() (easygopigo3.EasyGoPiGo3 method)</td>
<td>42</td>
</tr>
<tr>
<td>init_light_sensor() (easygopigo3.EasyGoPiGo3 method)</td>
<td>41</td>
</tr>
<tr>
<td>init_line_follower() (easygopigo3.EasyGoPiGo3 method)</td>
<td>42</td>
</tr>
<tr>
<td>init_loudness_sensor() (easygopigo3.EasyGoPiGo3 method)</td>
<td>42</td>
</tr>
<tr>
<td>init_motion_sensor() (easygopigo3.EasyGoPiGo3 method)</td>
<td>41</td>
</tr>
</tbody>
</table>
init_remote() (easygopigo3.EasyGoPiGo3 method), 44
init_servo() (easygopigo3.EasyGoPiGo3 method), 43
init_sound_sensor() (easygopigo3.EasyGoPiGo3 method), 41
init_ultrasonic_sensor() (easygopigo3.EasyGoPiGo3 method), 41
is_button_pressed() (easysensors.ButtonSensor method), 55
is_off() (easysensors.Led method), 53
is_on() (easysensors.Led method), 53
is_too_close() (easysensors.UltraSonicSensor method), 48
K
keycodes (easysensors.Remote attribute), 58
L
Led (class in easysensors), 51
led_off() (easygopigo3.EasyGoPiGo3 method), 40
led_on() (easygopigo3.EasyGoPiGo3 method), 40
left() (easygopigo3.EasyGoPiGo3 method), 36
light_max() (easysensors.Led method), 53
light_off() (easysensors.Led method), 53
LightSensor (class in easysensors), 44
LineFollower() (in module easysensors), 59
load_robot_constants() (easygopigo3.EasyGoPiGo3 method), 33
LoudnessSensor (class in easysensors), 46
M
motion_detected() (easysensors.MotionSensor method), 54
MotionSensor (class in easysensors), 53
O
open_eyes() (easygopigo3.EasyGoPiGo3 method), 41
open_left_eye() (easygopigo3.EasyGoPiGo3 method), 41
open_right_eye() (easygopigo3.EasyGoPiGo3 method), 41
orbit() (easygopigo3.EasyGoPiGo3 method), 37
P
percent_read() (easysensors.AnalogSensor method), 66
read() (easysensors.AnalogSensor method), 65
read() (easysensors.DHTSensor method), 58
R
read() (easysensors.Remote method), 58
read() (easysensors.UltraSonicSensor method), 49
read_encoders() (easygopigo3.EasyGoPiGo3 method), 39
read_encoders_average() (easygopigo3.EasyGoPiGo3 method), 39
read_humidity() (easysensors.DHTSensor method), 57
read_inches() (easysensors.UltraSonicSensor method), 49
read_mm() (easysensors.UltraSonicSensor method), 49
read_temperature() (easysensors.DHTSensor method), 57
reconfig_bus() (easysensors.Sensor method), 63
Remote (class in easysensors), 58
reset_encoders() (easygopigo3.EasyGoPiGo3 method), 38
reset_servo() (easysensors.Servo method), 56
reset_speed() (easygopigo3.EasyGoPiGo3 method), 34
right() (easygopigo3.EasyGoPiGo3 method), 36
rotate_servo() (easysensors.Servo method), 56
S
save_robot_constants() (easygopigo3.EasyGoPiGo3 method), 33
scale (easysensors.Buzzer attribute), 50
Sensor (class in easysensors), 61
Servo (class in easysensors), 55
set_descriptor() (easysensors.Sensor method), 64
set_left_eye_color() (easygopigo3.EasyGoPiGo3 method), 40
set_left_eye_color() (easygopigo3.EasyGoPiGo3 method), 40
set_pin() (easysensors.Sensor method), 63
set_pin_mode() (easysensors.Sensor method), 64
set_port() (easysensors.Sensor method), 64
set_right_eye_color() (easygopigo3.EasyGoPiGo3 method), 40
set_robot_constants() (easygopigo3.EasyGoPiGo3 method), 33
set_safe_distance() (easysensors.UltraSonicSensor method), 49
set_speed() (easygopigo3.EasyGoPiGo3 method), 34
sound() (easysensors.Buzzer method), 51
sound_off() (easysensors.Buzzer method), 51
sound_on() (easysensors.Buzzer method), 51
SoundSensor (class in easysensors), 45
spin_left() (easygopigo3.EasyGoPiGo3 method), 36
spin_right() (easygopigo3.EasyGoPiGo3 method), 36
steer() (easygopigo3.EasyGoPiGo3 method), 37
stop() (easygopigo3.EasyGoPiGo3 method), 34

T
target_reached() (easygopigo3.EasyGoPiGo3 method), 37
turn_degrees() (easygopigo3.EasyGoPiGo3 method), 39

U
UltraSonicSensor (class in easysensors), 47

V
volt() (easygopigo3.EasyGoPiGo3 method), 34

W
write() (easysensors.AnalogSensor method), 66
write_freq() (easysensors.AnalogSensor method), 66