GoPiGo3 Documentation

Release 1.0.0

Robert Lucian Chiriac

Dec 21, 2017

Contents:

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

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

3 Tutorials - Basic **7**
- 3.1 Flashing an LED . . . 7
- 3.2 Pushing a Button . . . 9
- 3.3 Ringing a Buzzer . . . 12
- 3.4 Detecting Light . . . 14
- 3.5 Measuring with the Distance Sensor . . . 16

4 Tutorials - Advanced **19**

5 API GoPiGo3 - Basic **21**
- 5.1 Requirements . . . 21
- 5.2 Hardware Ports . . . 21
- 5.3 EasyGoPiGo3 . . . 23
- 5.4 LightSensor . . . 34
- 5.5 SoundSensor . . . 35
- 5.6 LoudnessSensor . . . 36
- 5.7 UltrasonicSensor . . . 38
- 5.8 Buzzer . . . 40
- 5.9 Led . . . 42
- 5.10 MotionSensor . . . 44
- 5.11 ButtonSensor . . . 45
- 5.12 LineFollower . . . 46
- 5.13 Servo . . . 48
- 5.14 DistanceSensor . . . 50
- 5.15 DHTSensor . . . 51
- 5.16 Remote . . . 52

6 API GoPiGo3 - Advanced **55**
6.1 Requirements . . . 55
6.2 Sensor . . . 55
6.3 DigitalSensor . . . 59
6.4 AnalogSensor . . . 59

7 Developer's Guide **61**
7.1 Our collaborators . . . 61

8 Frequently Asked Questions **63**

9 Indices and tables **65**

DEXTER INDUSTRIES

CHAPTER 1

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.

CHAPTER 2

Getting Started

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

sudo sh -c "curl -kL dexterindustries.com/update_gopigo3 | bash"
```

Also, in order to be able to use the *easygopigo3.EasyGoPiGo3.init_distance_sensor()* method and the *easygopigo3.DistanceSensor* class, 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

sudo sh -c "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.

For more on getting started with these sensors, please check the DI-Sensors documentation.

CHAPTER 3

Tutorials - Basic

This chapter revolves around the `easygopigo3` module.

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. Sensor/Actuator specific for the tutorial : i.e.: a Grove Buzzer, a Line Follower, etc.

3.1 Flashing an LED

3.1.1 Our goal

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

3.1.2 The code we analyse

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

```
# 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")
# or
# my_LED = easy.Led("AD1", GPG)
```

```
# 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)

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

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

3.1.3 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.

3.1.4 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.

```
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"`.

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

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

3.1.5 Main part

In this section of the tutorial we are focusing on 3 methods of the *easygopigo3.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.

```
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)
```

3.1.6 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:

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

3.2 Pushing a Button

3.2.1 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.

3.2.2 The code we analyse

In the end the code should look like 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()

# 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.

3.2.3 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.

3.2.4 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.

```
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"`.

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

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

3.2.5 Setting variables

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

```
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.

3.2.6 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.

```
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.

3.2.7 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.

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

3.3 Ringing a Buzzer

3.3.1 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 Twinklle Little Star"* song.

3.3.2 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 = ["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.

3.3.3 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.

3.3.4 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.

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

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"`.

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

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

3.3.5 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.

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

3.3.6 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 *easygopigo3.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*.

```
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.

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

3.3.7 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.

3.4 Detecting Light

3.4.1 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.

3.4.2 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()

    # 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.

3.4.3 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.

3.4.4 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.

```
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.

```
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.

3.4.5 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)
```

3.4.6 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:

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

3.5 Measuring with the Distance Sensor

3.5.1 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.

3.5.2 The code we analyse

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

```
# 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 identifyng 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.

3.5.3 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.

3.5.4 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 *easygopigo3.DistanceSensor* 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
```

3.5.5 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 *easygopigo3.DistanceSensor*'s API for more details.

3.5.6 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.

CHAPTER 4

Tutorials - Advanced

Note: Coming soon!

CHAPTER 5

API GoPiGo3 - Basic

5.1 Requirements

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

```
import easygopigo3
```

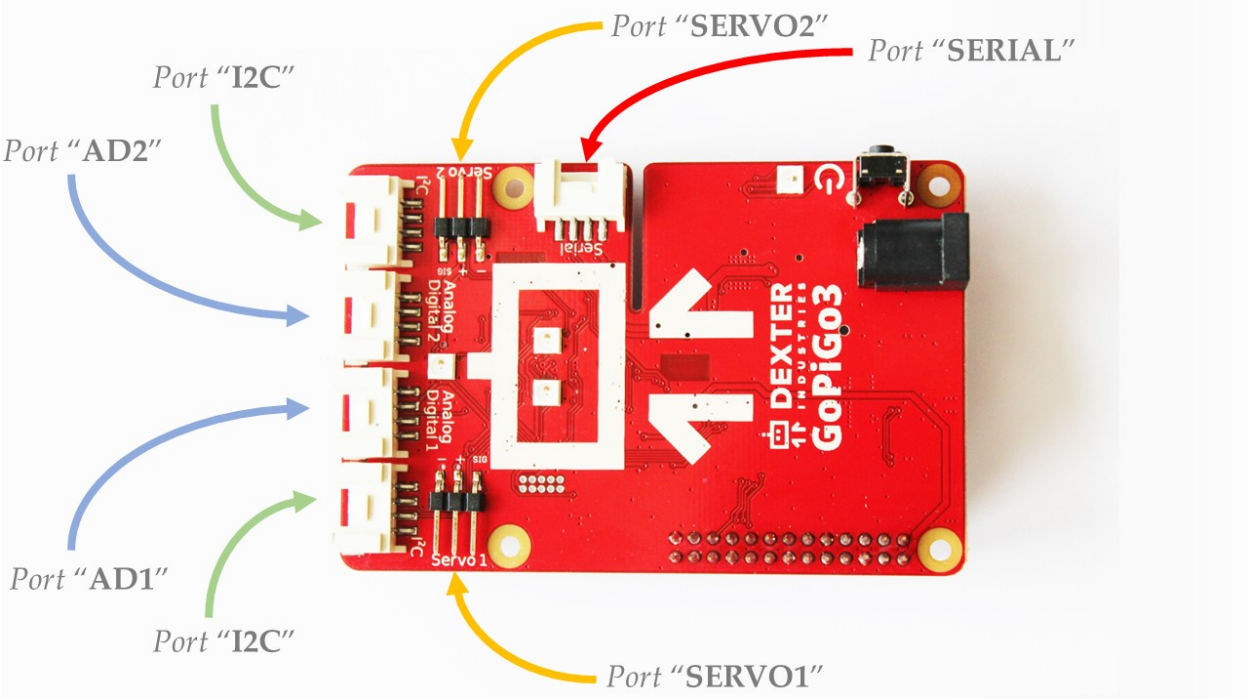
If you have issues importing these two modules, then make sure:

- You have followed the steps found in *Getting Started* guide.
- You have installed either Raspbian For Robots, 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.

5.2 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.

```
# 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 binded on port AD2
light_obj = gpg3_obj.init_light_sensor("AD2")

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

# here's a LineFollower object binded 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.

5.3 EasyGoPiGo3

class easygopigo3.**EasyGoPiGo3**(*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__(*use_mutex=False*)

This constructor sets the variables to the following values:

Parameters **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 **turqoise**.
- **right_eye_color = (0,255,255)** (*tuple(int,int,int)*) – Set Dex's right eye color to **turqoise**.
- **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.

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 conjuction with the following methods:

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

---

backward()

Move the GoPiGo3 backward.

For setting the motor speed, use *set_speed()*.
Default `speed` is set `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.

---

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.

---

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 aproximately *86%* of the GoPiGo3's wheel circumference.

```
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.

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

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

target_reached(*left_target_degrees*, *right_target_degrees*)

Checks if :

- 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
gpg3_obj.stop()
```

On the other hand, for moving the GoPiGo3 robot to the **right** for `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
gpg3_obj.stop()
```

Note: You *can* use this method in conjuction 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()

Resets both the encoders back to **0**.

When keeping track of the GoPiGo3 movements, this method is exclusively being required by the following methods:

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

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)

turn_degrees(*degrees*, *blocking=False*)

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 negative, and make it turn to the right, `degrees` has to be positive.

- **blocking = False** (*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` 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.

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*.

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*.

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*.

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*.

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*.

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*.

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*.

init_line_follower(*port='I2C'*)

Initialises a *LineFollower* object and then returns it.

Parameters **port = "I2C"** (*str*) – The only option for this parameter is "I2C". The default value for this parameter is already set to "I2C".

Returns An instance of the *LineFollower* class and with the port set to port's value.

The "I2C" 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 recognizeable by the GoPiGo3 platform.

---

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*.

init_distance_sensor(*port='I2C'*)

Initialises a *DistanceSensor* object and then returns it.

Parameters **port = "I2C"** (*str*) – The only option for this parameter is "I2C". The parameter has "I2C" as a default value.

Returns An instance of the *DistanceSensor* class and with the port set to port's value.

The "I2C" 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 recognizeable by the GoPiGo3 platform.

---

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.

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*.

---

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*.

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*.

5.4 LightSensor

class easygopigo3.**LightSensor**(*port='AD1'*, *gpg=None*, *use_mutex=False*)

Bases: *easygopigo3.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.

```
# 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.

```
# 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.

__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*.

5.5 SoundSensor

class easygopigo3.**SoundSensor**(*port='AD1'*, *gpg=None*, *use_mutex=False*)

Bases: *easygopigo3.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.

```
# 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.

```
# 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.

__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*.

5.6 LoudnessSensor

class easygopigo3.**LoudnessSensor**(*port='AD1'*, *gpg=None*, *use_mutex=False*)

Bases: *easygopigo3.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.

```
# 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()

# take a look at AnalogSensor class and Sensor 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 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*.

5.7 UltrasonicSensor

class easygopigo3.**UltraSonicSensor**(*port='AD1'*, *gpg=None*, *use_mutex=False*)

Bases: *easygopigo3.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.

```
# 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␣
↪to
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 boolean

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.

---

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.

---

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.

---

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**.

---

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**.

5.8 Buzzer

class easygopigo3.**Buzzer**(*port='AD1'*, *gpg=None*, *use_mutex=False*)
Bases: *easygopigo3.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.

```
# 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.

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

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

buzzer = gpg3_obj.init_buzzer(port)
```

See also:

For more sensors, please see our Dexter Industries shop.

scale = {'G4#': 415, 'D5#': 622, 'F4#': 370, 'C5#': 554, 'G5': 784, 'G4': 392, '

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*.

Example on how to play musical notes.

```
# 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.

5.9 Led

class easygopigo3.**Led**(*port='AD1'*, *gpg=None*, *use_mutex=False*)
Bases: *easygopigo3.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.

```
# 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.

```
# 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.

__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 boolean

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

Returns If the Grove LED is off.

Return type boolean

5.10 MotionSensor

class easygopigo3.**MotionSensor**(*port='AD1'*, *gpg=None*, *use_mutex=False*)
Bases: easygopigo3.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.

```
# 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.

```
# 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 boolean

5.11 ButtonSensor

class easygopigo3.**ButtonSensor**(*port='AD1'*, *gpg=None*, *use_mutex=False*)
Bases: easygopigo3.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.

```
# 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.

__init__(*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*.

is_button_pressed()

Checks if the Grove Button is pressed.

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

Return type boolean

5.12 LineFollower

class easygopigo3.**LineFollower**(*port='I2C'*, *gpg=None*, *use_mutex=False*)

Bases: *easygopigo3.Sensor*

Class for interacting with the Line Follower sensor. With this sensor, you can make your robot follow a black line on a white background.

The Line Follower sensor has 5 IR sensors. Each IR sensor is capable of diferentiating a black surface from a white one.

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

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

# and then initialize the LineFollower object
line_follower = gpg3_obj.init_line_follower()

# use it however you want it
line_follower.read_raw_sensors()
```

Warning: This class requires the `line_sensor` library.

__init__(*port='I2C'*, *gpg=None*, *use_mutex=False*)
Constructor for initalizing a *LineFollower* object.

Parameters

- **port = "I2C"** (*str*) – The port to which we have connected the Line Follower sensor.
- **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

- **ImportError** – If the `line_follower` module couldn't be found.
- **TypeError** – If the `gpg` parameter is not a *EasyGoPiGo3* object.
- **IOError** – If the line follower is not responding.

The only value the `port` parameter can take is `"I2C"`.

The I2C ports' location on the GoPiGo3 robot can be seen in the following graphical representation: *Hardware Ports*.

read_raw_sensors()
Read the 5 IR sensors of the Line Follower sensor.

Returns A list with 5 10-bit numbers that represent the readings from the line follower device.

Return type list[int]

Raises **IOError** – If the line follower is not responding.

get_white_calibration()
Place the GoPiGo3 robot on top of a white-colored surface. After that, call this method for calibrating the robot on a white surface.

Returns A list with 5 10-bit numbers that represent the readings of line follower sensor.

Return type int

Also, for fully calibrating the sensor, the *get_black_calibration* method also needs to be called.

get_black_calibration()
Place the GoPiGo3 robot on top of a black-colored surface. After that, call this method for calibrating the robot on a black surface.

Returns A list with 5 10-bit numbers that represent the readings of line follower sensor.

Return type int

Also, for fully calibrating the sensor, the *get_white_calibration* method also needs to be called.

read()
Reads the 5 IR sensors of the Line Follower sensor.

Returns A list with 5 numbers that represent the readings of the line follower device. The values are either **0** (for black) or **1** (for white).

Return type list[int]

> **Warning:** If an error occurs, a list of **5 numbers** with values set to **-1** will be returned. This may be caused by bad calibration values.
>
> Please use *get_black_calibration()* or *get_white_calibration()* methods before calling this method.

read_position()
Returns a string telling to which side the black line that we're following is located.

Returns String that's indicating the location of the black line.

Return type str

The strings this method can return are the following:

- `"center"` - when the line is found in the middle.
- `"black"` - when the line follower sensor only detects black surfaces.
- `"white"` - when the line follower sensor only detects white surfaces.
- `"left"` - when the black line is located on the left of the sensor.
- `"right"` - when the black line is located on the right of the sensor.

Note: This isn't the most "intelligent" algorithm for following a black line, but it proves the point and it works.

5.13 Servo

class easygopigo3.**Servo**(*port='SERVO1'*, *gpg=None*, *use_mutex=False*)
Bases: *easygopigo3.Sensor*

Class for controlling servo motors with the GoPiGo3 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.

```
# 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.

```
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.

5.14 DistanceSensor

Important: The *easygopigo3.DistanceSensor* class requires the `DI_Sensors` package installed. Please check the documentation of the DI-Sensors's package on how to install it.

class easygopigo3.**DistanceSensor**(*port='I2C'*, *gpg=None*, *use_mutex=False*)
Bases: *easygopigo3.Sensor*, `mock_package.distance_sensor.DistanceSensor`

Class for the Distance Sensor device.

We can create this *DistanceSensor* object similar to how we create it in the following template.

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

# and now let's instantiate a DistanceSensor object through the gpg3_obj object
distance_sensor = gpg3_obj.init_distance_sensor()

# read values continuously and print them in the terminal
while True:
    distance = distance_sensor.read()

    print(distance)
```

__init__(*port='I2C'*, *gpg=None*, *use_mutex=False*)
Creates a *DistanceSensor* object which can be used for interfacing with a distance sensor.

Parameters

- **port = "I2C"** (*str*) – Port to which the distance sensor is connected.
- **gpg = None** (`easygopigo3.EasyGoPiGo3`) – Object that's required for instantianting a *DistanceSensor* object.
- **use_mutex = False** (*bool*) – When using multiple threads/processes that access the same resource/device, mutexes should be enabled.

Raises

- **IOError** – If `di_sensors.distance_sensor.DistanceSensor` can't be found. Probably the `di_sensors` module isn't installed.
- **TypeError** – If the `gpg` parameter is not a *EasyGoPiGo3* object.

To see where the ports are located on the GoPiGo3 robot, please take a look at the following diagram: *Hardware Ports*.

read_mm()
Reads the distance in millimeters.

Returns Distance from target in millimeters.

Return type int

Note:

1. Sensor's range is **5-8,000** millimeters.
2. When the values are out of the range, it returns **8190**.

---

read()
Reads the distance in centimeters.

Returns Distance from target in centimeters.

Return type int

Note:

1. Sensor's range is **0-800** centimeters.
2. When the values are out of the range, it returns **819**.

---

read_inches()
Reads the distance in inches.

Returns Distance from target in inches.

Return type float with one decimal

Note:

1. Sensor's range is **0-314** inches.
2. Anything that's bigger than **314** inches is returned when the sensor can't detect any target/surface.

5.15 DHTSensor

class easygopigo3.**DHTSensor**(*gpg=None*, *sensor_type=0*, *use_mutex=False*)
Bases: *easygopigo3.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.

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

# and now let's instantiate a DistanceSensor 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 instantianting a *DistanceSensor* 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.

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.

5.16 Remote

class easygopigo3.**Remote**(*port='AD1'*, *gpg=None*, *use_mutex=False*)
Bases: *easygopigo3.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.

```
# 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
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.

CHAPTER 6

API GoPiGo3 - Advanced

6.1 Requirements

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

```
import easygopigo3
import gopigo3
```

If you have issues importing these 2 modules, 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.

6.2 Sensor

class easygopigo3.**Sensor**(*port*, *pinmode*, *gpg*, *use_mutex=False*)

Bases: object

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 `ports`'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*
- *LineFollower*
- *Servo*
- *DistanceSensor*
- *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.

__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`.

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 easygopigo3.**AnalogSensor**(*port*, *pinmode*, *gpg*, *use_mutex=False*)

Bases: *easygopigo3.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.

```
# 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
analogsensor_obj = AnalogSensor(port, pinmode, gpg3_obj)

# for example
# read the sensor's value as we have an analog sensor connected to "AD1" port
analogsensor_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 concurently), 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.

CHAPTER 7

Developer's Guide

7.1 Our collaborators

The following collaborators are ordered alphabetically:

- John Cole - Github Account.
- Matt Richardson - Github Account.
- Nicole Parrot - Github Account.
- Robert Lucian Chiriac - Github Account.
- Shoban Narayan - Github account.

CHAPTER 8

Frequently Asked Questions

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

CHAPTER 9

Indices and tables

- genindex
- modindex
- search

Index

Symbols

__init__() (easygopigo3.AnalogSensor method), 59
__init__() (easygopigo3.ButtonSensor method), 46
__init__() (easygopigo3.Buzzer method), 41
__init__() (easygopigo3.DHTSensor method), 52
__init__() (easygopigo3.DistanceSensor method), 50
__init__() (easygopigo3.EasyGoPiGo3 method), 23
__init__() (easygopigo3.Led method), 43
__init__() (easygopigo3.LightSensor method), 35
__init__() (easygopigo3.LineFollower method), 47
__init__() (easygopigo3.LoudnessSensor method), 37
__init__() (easygopigo3.MotionSensor method), 45
__init__() (easygopigo3.Remote method), 53
__init__() (easygopigo3.Sensor method), 56
__init__() (easygopigo3.Servo method), 49
__init__() (easygopigo3.SoundSensor method), 36
__init__() (easygopigo3.UltraSonicSensor method), 38
__str__() (easygopigo3.Sensor method), 57

A

AnalogSensor (class in easygopigo3), 59

B

backward() (easygopigo3.EasyGoPiGo3 method), 24
blinker_off() (easygopigo3.EasyGoPiGo3 method), 29
blinker_on() (easygopigo3.EasyGoPiGo3 method), 29
ButtonSensor (class in easygopigo3), 45
Buzzer (class in easygopigo3), 40

C

close_eyes() (easygopigo3.EasyGoPiGo3 method), 31
close_left_eye() (easygopigo3.EasyGoPiGo3 method), 31
close_right_eye() (easygopigo3.EasyGoPiGo3 method), 31

D

DHTSensor (class in easygopigo3), 51
DistanceSensor (class in easygopigo3), 50
drive_cm() (easygopigo3.EasyGoPiGo3 method), 25
drive_degrees() (easygopigo3.EasyGoPiGo3 method), 26
drive_inches() (easygopigo3.EasyGoPiGo3 method), 26

E

EasyGoPiGo3 (class in easygopigo3), 23

F

forward() (easygopigo3.EasyGoPiGo3 method), 25

G

get_black_calibration() (easygopigo3.LineFollower method), 47
get_pin() (easygopigo3.Sensor method), 58
get_pin_mode() (easygopigo3.Sensor method), 58
get_port() (easygopigo3.Sensor method), 58
get_port_ID() (easygopigo3.Sensor method), 58
get_remote_code() (easygopigo3.Remote method), 53
get_safe_distance() (easygopigo3.UltraSonicSensor method), 39
get_speed() (easygopigo3.EasyGoPiGo3 method), 24
get_white_calibration() (easygopigo3.LineFollower method), 47

I

init_button_sensor() (easygopigo3.EasyGoPiGo3 method), 32
init_buzzer() (easygopigo3.EasyGoPiGo3 method), 32
init_dht_sensor() (easygopigo3.EasyGoPiGo3 method), 33
init_distance_sensor() (easygopigo3.EasyGoPiGo3 method), 33
init_led() (easygopigo3.EasyGoPiGo3 method), 32
init_light_sensor() (easygopigo3.EasyGoPiGo3 method), 31
init_line_follower() (easygopigo3.EasyGoPiGo3 method), 32
init_loudness_sensor() (easygopigo3.EasyGoPiGo3 method), 31

init_motion_sensor() (easygopigo3.EasyGoPiGo3 method), 34
init_remote() (easygopigo3.EasyGoPiGo3 method), 33
init_servo() (easygopigo3.EasyGoPiGo3 method), 33
init_sound_sensor() (easygopigo3.EasyGoPiGo3 method), 31
init_ultrasonic_sensor() (easygopigo3.EasyGoPiGo3 method), 31
is_button_pressed() (easygopigo3.ButtonSensor method), 46
is_off() (easygopigo3.Led method), 44
is_on() (easygopigo3.Led method), 43
is_too_close() (easygopigo3.UltraSonicSensor method), 39

K

keycodes (easygopigo3.Remote attribute), 53

L

Led (class in easygopigo3), 42
led_off() (easygopigo3.EasyGoPiGo3 method), 29
led_on() (easygopigo3.EasyGoPiGo3 method), 29
left() (easygopigo3.EasyGoPiGo3 method), 25
light_max() (easygopigo3.Led method), 43
light_off() (easygopigo3.Led method), 43
light_on() (easygopigo3.Led method), 43
LightSensor (class in easygopigo3), 34
LineFollower (class in easygopigo3), 46
LoudnessSensor (class in easygopigo3), 36

M

motion_detected() (easygopigo3.MotionSensor method), 45
MotionSensor (class in easygopigo3), 44

O

open_eyes() (easygopigo3.EasyGoPiGo3 method), 31
open_left_eye() (easygopigo3.EasyGoPiGo3 method), 30
open_right_eye() (easygopigo3.EasyGoPiGo3 method), 30

P

percent_read() (easygopigo3.AnalogSensor method), 60

R

read() (easygopigo3.AnalogSensor method), 60
read() (easygopigo3.DHTSensor method), 52
read() (easygopigo3.DistanceSensor method), 51
read() (easygopigo3.LineFollower method), 48
read() (easygopigo3.Remote method), 53
read() (easygopigo3.UltraSonicSensor method), 40
read_encoders() (easygopigo3.EasyGoPiGo3 method), 28
read_humidity() (easygopigo3.DHTSensor method), 52
read_inches() (easygopigo3.DistanceSensor method), 51
read_inches() (easygopigo3.UltraSonicSensor method), 40
read_mm() (easygopigo3.DistanceSensor method), 50
read_mm() (easygopigo3.UltraSonicSensor method), 39
read_position() (easygopigo3.LineFollower method), 48
read_raw_sensors() (easygopigo3.LineFollower method), 47
read_temperature() (easygopigo3.DHTSensor method), 52
Remote (class in easygopigo3), 52
reset_encoders() (easygopigo3.EasyGoPiGo3 method), 28
reset_servo() (easygopigo3.Servo method), 50
reset_speed() (easygopigo3.EasyGoPiGo3 method), 24
right() (easygopigo3.EasyGoPiGo3 method), 24
rotate_servo() (easygopigo3.Servo method), 49

S

scale (easygopigo3.Buzzer attribute), 41
Sensor (class in easygopigo3), 55
Servo (class in easygopigo3), 48
set_descriptor() (easygopigo3.Sensor method), 58
set_eye_color() (easygopigo3.EasyGoPiGo3 method), 30
set_left_eye_color() (easygopigo3.EasyGoPiGo3 method), 30
set_pin() (easygopigo3.Sensor method), 57
set_pin_mode() (easygopigo3.Sensor method), 58
set_port() (easygopigo3.Sensor method), 58
set_right_eye_color() (easygopigo3.EasyGoPiGo3 method), 30
set_safe_distance() (easygopigo3.UltraSonicSensor method), 39
set_speed() (easygopigo3.EasyGoPiGo3 method), 23
sound() (easygopigo3.Buzzer method), 41
sound_off() (easygopigo3.Buzzer method), 42
sound_on() (easygopigo3.Buzzer method), 42
SoundSensor (class in easygopigo3), 35
stop() (easygopigo3.EasyGoPiGo3 method), 24

T

target_reached() (easygopigo3.EasyGoPiGo3 method), 26
turn_degrees() (easygopigo3.EasyGoPiGo3 method), 28

U

UltraSonicSensor (class in easygopigo3), 38

V

volt() (easygopigo3.EasyGoPiGo3 method), 23

W

write() (easygopigo3.AnalogSensor method), 60
write_freq() (easygopigo3.AnalogSensor method), 60