python-ev3dev Documentation

Release 0.4.1.post44

Ralph Hempel et al

February 28, 2017

Contents

This is a python library implementing unified interface for ev3dev devices.

CHAPTER 1

Module interface

An assortment of classes modeling specific features of the EV3 brick.

Device	The ev3dev device base class
Motor	The motor class provides a uniform interface for using motors with positional and directional feedback suc
DcMotor	The DC motor class provides a uniform interface for using regular DC motors with no fancy controls or fee
ServoMotor	The servo motor class provides a uniform interface for using hobby type servo motors.
MediumMotor	EV3 medium servo motor
LargeMotor	EV3 large servo motor
Sensor	The sensor class provides a uniform interface for using most of the sensors available for the EV3.
I2cSensor	A generic interface to control I2C-type EV3 sensors.
TouchSensor	Touch Sensor
ColorSensor	LEGO EV3 color sensor.
UltrasonicSensor	LEGO EV3 ultrasonic sensor.
GyroSensor	LEGO EV3 gyro sensor.
SoundSensor	LEGO NXT Sound Sensor
LightSensor	LEGO NXT Light Sensor
InfraredSensor	LEGO EV3 infrared sensor.
RemoteControl	EV3 Remote Controller
Led	Any device controlled by the generic LED driver.
PowerSupply	A generic interface to read data from the system's power_supply class.
Button	EV3 Buttons
Sound	Sound-related functions.
Screen	A convenience wrapper for the FbMem class.

Generic device

class ev3dev.ev3.**Device**(*class_name*, *name='*'*, ***kwargs*)
The ev3dev device base class

Motors

class ev3dev.ev3.**Motor**(*port=None*, *name='motor*'*, ***kwargs*)
The motor class provides a uniform interface for using motors with positional and directional feedback such as the EV3 and NXT motors. This feedback allows for precise control of the motors. This is the most common type of motor, so we just call it *motor*.

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

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

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

count_per_rot
Returns the number of tacho counts in one rotation of the motor. Tacho counts are used by the position and speed attributes, so you can use this value to convert rotations or degrees to tacho counts. In the case of linear actuators, the units here will be counts per centimeter.

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

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

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

encoder_polarity
Sets the polarity of the rotary encoder. This is an advanced feature to all use of motors that send inversed encoder signals to the EV3. This should be set correctly by the driver of a device. It You only need to change this value if you are using a unsupported device. Valid values are *normal* and *inversed*.

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

port_name
Returns the name of the port that the motor is connected to.

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

position_d
The derivative constant for the position PID.

position_i
The integral constant for the position PID.

position_p
The proportional constant for the position PID.

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

ramp_down_sp
Writing sets the ramp down setpoint. Reading returns the current value. Units are in milliseconds. When set to a value > 0, the motor will ramp the power sent to the motor from 100% duty cycle down to 0 over the span of this setpoint when stopping the motor. If the starting duty cycle is less than 100%, the ramp time duration will be less than the full span of the setpoint.

ramp_up_sp
Writing sets the ramp up setpoint. Reading returns the current value. Units are in milliseconds. When set to a value > 0, the motor will ramp the power sent to the motor from 0 to 100% duty cycle over the span of this setpoint when starting the motor. If the maximum duty cycle is limited by *duty_cycle_sp* or speed regulation, the actual ramp time duration will be less than the setpoint.

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

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

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

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

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

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

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

speed_regulation_d
The derivative constant for the speed regulation PID.

speed_regulation_enabled
Turns speed regulation on or off. If speed regulation is on, the motor controller will vary the power supplied to the motor to try to maintain the speed specified in *speed_sp*. If speed regulation is off, the controller will use the power specified in *duty_cycle_sp*. Valid values are *on* and *off*.

speed_regulation_i
The integral constant for the speed regulation PID.

speed_regulation_p
The proportional constant for the speed regulation PID.

speed_sp
Writing sets the target speed in tacho counts per second used when *speed_regulation* is on. Reading returns the current value. Use the *count_per_rot* attribute to convert RPM or deg/sec to tacho counts per second.

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

stop(***kwargs*)
Stop any of the run commands before they are complete using the command specified by *stop_command*.

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

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

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

class ev3dev.ev3.**MediumMotor**(*port=None*, *name='motor*'*, ***kwargs*)
Bases: ev3dev.core.Motor

EV3 medium servo motor

class ev3dev.ev3.**LargeMotor**(*port=None*, *name='motor*'*, ***kwargs*)
Bases: ev3dev.core.Motor

EV3 large servo motor

class ev3dev.ev3.**DcMotor**(*port=None*, *name='motor*'*, ***kwargs*)
The DC motor class provides a uniform interface for using regular DC motors with no fancy controls or feedback. This includes LEGO MINDSTORMS RCX motors and LEGO Power Functions motors.

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

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

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

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

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

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

port_name
Returns the name of the port that the motor is connected to.

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

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

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

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

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

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

stop(***kwargs*)
Stop any of the run commands before they are complete using the command specified by *stop_command*.

stop_command
Sets the stop command that will be used when the motor stops. Read *stop_commands* to get the list of valid values.

stop_commands
Gets a list of stop commands. Valid values are *coast* and *brake*.

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

class ev3dev.ev3.**ServoMotor**(*port=None*, *name='motor*'*, ***kwargs*)
The servo motor class provides a uniform interface for using hobby type servo motors.

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

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

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

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

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

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

polarity
Sets the polarity of the servo. Valid values are *normal* and *inversed*. Setting the value to *inversed* will cause the position_sp value to be inversed. i.e *-100* will correspond to *max_pulse_sp*, and *100* will correspond to *min_pulse_sp*.

port_name
Returns the name of the port that the motor is connected to.

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

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

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

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

Sensors

class ev3dev.ev3.**Sensor**(*port=None*, *name='sensor*'*, ***kwargs*)
The sensor class provides a uniform interface for using most of the sensors available for the EV3. The various underlying device drivers will create a *lego-sensor* device for interacting with the sensors.

Sensors are primarily controlled by setting the *mode* and monitored by reading the *value<N>* attributes. Values can be converted to floating point if needed by *value<N>* / 10.0 ^ *decimals*.

Since the name of the *sensor<N>* device node does not correspond to the port that a sensor is plugged in to, you must look at the *port_name* attribute if you need to know which port a sensor is plugged in to. However, if you don't have more than one sensor of each type, you can just look for a matching *driver_name*. Then it will not matter which port a sensor is plugged in to - your program will still work.

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

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

Example:

```
>>> from ev3dev import *
>>> ir = InfraredSensor()
>>> ir.value()
28
>>> ir.bin_data('<b')
(28,)
```

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

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

command
Sends a command to the sensor.

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

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

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

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

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

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

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

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

class ev3dev.ev3.**I2cSensor**(*port=None*, *name='sensor*'*, ***kwargs*)
Bases: ev3dev.core.Sensor

A generic interface to control I2C-type EV3 sensors.

fw_version
Returns the firmware version of the sensor if available. Currently only I2C/NXT sensors support this.

poll_ms
Returns the polling period of the sensor in milliseconds. Writing sets the polling period. Setting to 0 disables polling. Minimum value is hard coded as 50 msec. Returns -EOPNOTSUPP if changing polling is not supported. Currently only I2C/NXT sensors support changing the polling period.

class ev3dev.ev3.**TouchSensor**(*port=None*, *name='sensor*'*, ***kwargs*)
Bases: ev3dev.core.Sensor

Touch Sensor

class ev3dev.ev3.**ColorSensor**(*port=None*, *name='sensor*'*, ***kwargs*)
Bases: ev3dev.core.Sensor

LEGO EV3 color sensor.

class ev3dev.ev3.**UltrasonicSensor**(*port=None*, *name='sensor*'*, ***kwargs*)
Bases: ev3dev.core.Sensor

LEGO EV3 ultrasonic sensor.

class ev3dev.ev3.**GyroSensor**(*port=None*, *name='sensor*'*, ***kwargs*)
Bases: ev3dev.core.Sensor

LEGO EV3 gyro sensor.

class ev3dev.ev3.**SoundSensor**(*port=None*, *name='sensor*'*, ***kwargs*)
Bases: ev3dev.core.Sensor

LEGO NXT Sound Sensor

class ev3dev.ev3.**LightSensor**(*port=None*, *name='sensor*'*, ***kwargs*)
Bases: ev3dev.core.Sensor

LEGO NXT Light Sensor

class ev3dev.ev3.**InfraredSensor**(*port=None*, *name='sensor*'*, ***kwargs*)
Bases: ev3dev.core.Sensor

LEGO EV3 infrared sensor.

class ev3dev.ev3.**RemoteControl**(*sensor=None*, *channel=1*)
EV3 Remote Controller

any()
Checks if any button is pressed.

beacon
Checks if *beacon* button is pressed.

blue_down
Checks if *blue_down* button is pressed.

blue_up
Checks if *blue_up* button is pressed.

buttons_pressed
Returns list of currently pressed buttons.

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

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

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

red_down
Checks if *red_down* button is pressed.

red_up
Checks if *red_up* button is pressed.

Other

class ev3dev.ev3.**Led**(*port=None*, *name='*'*, ***kwargs*)
Any device controlled by the generic LED driver. See https://www.kernel.org/doc/Documentation/leds/leds-class.txt for more details.

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

brightness_pct
Returns led brightness as a fraction of max_brightness

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

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

max_brightness
Returns the maximum allowable brightness value.

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

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

triggers
Returns a list of available triggers.

class ev3dev.ev3.**PowerSupply**(*port=None*, *name='*'*, ***kwargs*)
A generic interface to read data from the system's power_supply class. Uses the built-in legoev3-battery if none is specified.

max_voltage

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

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

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

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

min_voltage

technology

type

class ev3dev.ev3.**Button**
EV3 Buttons

any()
Checks if any button is pressed.

backspace
Check if 'backspace' button is pressed.

buttons_pressed
Returns list of names of pressed buttons.

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

down
Check if 'down' button is pressed.

enter
Check if 'enter' button is pressed.

left
Check if 'left' button is pressed.

static on_backspace(*state*)
This handler is called by *process()* whenever state of 'backspace' button has changed since last *process()* call. *state* parameter is the new state of the button.

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

static on_down(*state*)
This handler is called by *process()* whenever state of 'down' button has changed since last *process()* call. *state* parameter is the new state of the button.

static on_enter(*state*)
This handler is called by *process()* whenever state of 'enter' button has changed since last *process()* call. *state* parameter is the new state of the button.

static on_left(*state*)
This handler is called by *process()* whenever state of 'left' button has changed since last *process()* call. *state* parameter is the new state of the button.

static on_right(*state*)
This handler is called by *process()* whenever state of 'right' button has changed since last *process()* call. *state* parameter is the new state of the button.

static on_up(*state*)
This handler is called by *process()* whenever state of 'up' button has changed since last *process()* call. *state* parameter is the new state of the button.

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

right
Check if 'right' button is pressed.

up
Check if 'up' button is pressed.

class ev3dev.ev3.**Sound**
Sound-related functions. The class has only static methods and is not intended for instantiation. It can beep, play wav files, or convert text to speech.

Note that all methods of the class spawn system processes and return subprocess.Popen objects. The methods are asynchronous (they return immediately after child process was spawned, without waiting for its completion), but you can call wait() on the returned result.

Examples:

```
# Play 'bark.wav', return immediately:
Sound.play('bark.wav')

# Introduce yourself, wait for completion:
Sound.speak('Hello, I am Robot').wait()
```

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

static play(*wav_file*)
Play wav file.

static speak(*text*)
Speak the given text aloud.

static tone(**args*)
tone(tone_sequence):

Play tone sequence. The tone_sequence parameter is a list of tuples, where each tuple contains up to three numbers. The first number is frequency in Hz, the second is duration in milliseconds, and the third is delay in milliseconds between this and the next tone in the sequence.

Here is a cheerful example:

```
Sound.tone([
    (392, 350, 100), (392, 350, 100), (392, 350, 100), (311.1, 250, 100),
    (466.2, 25, 100), (392, 350, 100), (311.1, 250, 100), (466.2, 25, 100),
    (392, 700, 100), (587.32, 350, 100), (587.32, 350, 100),
    (587.32, 350, 100), (622.26, 250, 100), (466.2, 25, 100),
    (369.99, 350, 100), (311.1, 250, 100), (466.2, 25, 100), (392, 700, 100),
    (784, 350, 100), (392, 250, 100), (392, 25, 100), (784, 350, 100),
    (739.98, 250, 100), (698.46, 25, 100), (659.26, 25, 100),
    (622.26, 25, 100), (659.26, 50, 400), (415.3, 25, 200), (554.36, 350, 100),
    (523.25, 250, 100), (493.88, 25, 100), (466.16, 25, 100), (440, 25, 100),
    (466.16, 50, 400), (311.13, 25, 200), (369.99, 350, 100),
    (311.13, 250, 100), (392, 25, 100), (466.16, 350, 100), (392, 250, 100),
    (466.16, 25, 100), (587.32, 700, 100), (784, 350, 100), (392, 250, 100),
```

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

tone(frequency, duration):

Play single tone of given frequency (Hz) and duration (milliseconds).

class ev3dev.ev3.**Screen**

Bases: ev3dev.core.FbMem

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

clear()

Clears the screen

draw

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

Example:

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

shape

Dimensions of the screen.

update()

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

xres

Horizontal screen resolution

yres

Vertical screen resolution

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

e

N

O

P

R

S

T

U

X

Y