SimulRPi

Release 0.1.0a1.dev1

Raul C.

Sep 25, 2020

CONTENTS

SimulRPi (0.1.0a1.dev1) is a Python library that partly fakes RPi.GPIO and simulates some I/O devices on a Raspberry Pi (RPi).

Each dot represents a blinking LED connected to an RPi and the number between brackets is the associated GPIO channel number. Here the LED on channel 22 toggles between on and off when a key is pressed.

See the README for more info about the library.

CHAPTER

ONE

README

SimulRPi (0.1.0a1.dev1) is a Python library that partly fakes RPi.GPIO and simulates some I/O devices on a Raspberry Pi (RPi).

1.1 Introduction

In addition to partly faking RPi.GPIO, **SimulRPi** also simulates these I/O devices connected to an RPi:

- push buttons by listening to pressed keyboard keys and
- LEDs by blinking dots in the terminal along with their GPIO pin numbers.

When a LED is turned on, it is shown as a red dot in the terminal. The pynput package is used to monitor the keyboard for any pressed key. Thus, the **SimulRPi** library can be useful in the case that you want to try your RPi.GPIO-based script by running it on your computer when no RPi is available at the moment.

Example: terminal output

Each dot represents a blinking LED connected to an RPi and the number between brackets is the associated GPIO channel number. Here the LED on channel 22 toggles between on and off when a key is pressed.

Also, the color of the LEDs can be customized as you can see here where the LED on channel 22 is colored differently from the others.

Important: This library is not a Raspberry Pi emulator nor a complete mock-up of RPi.GPIO, only the most important functions that I needed for my Darth-Vader-RPi project were added.

If there is enough interest in this library, I will eventually mock more functions from RPi.GPIO.

1.2 Dependencies

- **Platforms:** Linux, macOS
- **Python**: 3.5, 3.6, 3.7, 3.8
- `pynput` >=1.6.8: for monitoring the keyboard for any pressed key

1.3 Installation instructions

1. It is highly recommended to install `SimulRPi` in a virtual environment using for example `venv` or conda.
2. Make sure to update *pip*:

   ```
   $ pip install --upgrade pip
   ```

3. Install the package `SimulRPi` (**v0.1.0a1**) with *pip*:

   ```
   $ pip install SimulRPi
   ```

   It will install the dependency `pynput` if it is not already found in your system.

Important: Make sure that *pip* is working with the correct Python version. It might be the case that *pip* is using Python 2.x You can find what Python version *pip* uses with the following:

```
$ pip -V
```

If *pip* is working with the wrong Python version, then try to use *pip3* which works with Python 3.x

Note: To install the **bleeding-edge version** (0.1.0a1.dev1) of the `SimulRPi` package, install it from its github repository:

```
$ pip install git+https://github.com/raul23/SimulRPi#egg=SimulRPi
```

However, this latest version is not as stable as the one from PyPI but you get the latest features being implemented.

Warning message

If you get the warning message from *pip* that the *run_examples* script is not defined in your *PATH*:

```
WARNING: The script run_examples is installed in '/home/pi/.local/bin' which is not␣
↪on PATH.
```

Add the directory mentioned in the warning to your *PATH* by editing your configuration file (e.g. *.bashrc*). See this article on how to set *PATH* on Linux and macOS.

Test installation

Test your installation by importing `SimulRPi` and printing its version:

```
$ python -c "import SimulRPi; print(SimulRPi.__version__)"
```

1.4 Usage

1.4.1 Use the library in your own code

Case 1: with a `try` and `except` blocks

You can try importing `RPi.GPIO` first and if it is not found, then fallback on the *SimulRPi.GPIO* module.

Listing 1: **Case 1:** with a `try` and `except` blocks

```
try:
    import RPi.GPIO as GPIO
except ImportError:
    import SimulRPi.GPIO as GPIO

# Rest of your code
```

The code from the previous example would be put at the beginning of your file with the other imports.

Case 2: with a simulation flag

Or maybe you have a flag to tell whether you want to work with the simulation module or the real one.

Listing 2: **Case 2:** with a simulation flag

```
if simulation:
    import SimulRPi.GPIO as GPIO
else:
    import RPi.GPIO as GPIO

# Rest of your code
```

1.4.2 Script `run_examples`

The *run_examples* script which you have access to once you *install* the `SimulRPi` package allows you to run different code examples on your RPi or computer. If it is run on your computer, it will make use of the *SimulRPi.GPIO* module which partly fakes `RPi.GPIO`.

The different code examples are those presented in *Examples* and show the capability of *SimulRPi.GPIO* for simulating I/O devices on an RPi such as push buttons and LEDs.

Here is a list of the functions that implement each code example:

- Example 1: *ex1_turn_on_led()*
- Example 2: *ex2_turn_on_many_leds()*
- Example 3: *ex3_detect_button()*
- Example 4: *ex4_blink_led()*
- Example 5: *ex5_blink_led_if_button()*

List of options

To display the script's list of options and their descriptions:

```
$ run_examples -h
```

-e	The number of the code example you want to run. It is required. (default: None)
-m	Set the numbering system (BCM or BOARD) used to identify the I/O pins on an RPi. (default: BCM)

-s	Enable simulation mode, i.e. `SimulRPi.GPIO` will be used for simulating `RPi.GPIO`. (default: False)
-l	The channel numbers to be used for LEDs. If an example only requires 1 channel, the first channel from the provided list will be used. (default: [9, 10, 11])
-b	The channel number to be used for a push button. The default value is channel 17 which is associated by default with the keyboard key *cmd_r*. (default: 17)
-k	The name of the key associated with the button channel. The name must be one of those recognized by the *pynput* package. See the *SimulRPi* documentation for a list of valid key names: https://bit.ly/2Pw1OBe. Example: *alt*, *ctrl_r* (default: *cmd_r*)
-t	Total time in seconds the LEDs will be blinking. (default: 4)
--on	Time in seconds the LEDs will stay turned ON at a time. (default: 1)
--off	Time in seconds the LEDs will stay turned OFF at a time. (default: 1)
-a	Use ASCII-based LED symbols. Useful if you are having problems displaying the default LED signs that make use of special characters. However, it is recommended to fix your display problems which might be caused by locale settings not set correctly. Check the article 'Display problems' @ https://bit.ly/35B8bfs for more info about solutions to display problems (default: False)

How to run the script

Once you *install* the `SimulRPi` package, you should have access to the *`run_examples`* script which can be called from the terminal by providing some arguments.

For example:

```
$ run_examples -e 1 -s
```

Let's run the code example 5 which blinks a LED if a specified key is pressed:

```
$ run_examples -s -e 5 -l 22 -t 5 -k ctrl_r
```

Explanation of the previous command-line:

- `-s`: we run the code example as a **simulation**, i.e. on our computer instead of an RPi
- `-e 5`: we run code example **5** which blinks a LED if a key is pressed
- `-l 22`: we blink a LED on channel **22**
- `-t 5`: we blink a LED for a total of **5** seconds
- `-k ctrl_r`: a LED is blinked if the key `ctrl_r` is pressed

Output:

Important: Don't forget the *-s* flag when running the *run_examples* script as simulation, if you want to run a code example on your computer, and not on your RPi.

1.5 Examples

The examples presented thereafter will show you how to use `SimulRPi` to simulate LEDs and push buttons.

The code for the examples shown here can be also found as a script in run_examples.

Note: Since we are showing how to use the `SimulRPi` package, the presented code examples are to be executed on your computer. However, the *run_examples* script which runs the following code examples can be executed on a Raspberry Pi or your computer.

1.5.1 Example 1: display 1 LED

Example 1 consists in displaying one LED on the GPIO channel 10. Here is the code along with the output from the terminal:

```
import SimulRPi.GPIO as GPIO

led_channel = 10
GPIO.setmode(GPIO.BCM)
GPIO.setup(led_channel, GPIO.OUT)
GPIO.output(led_channel, GPIO.HIGH)
GPIO.cleanup()
```

Output:

The command line for reproducing the same results for example 1 with the *run_examples* script is the following:

```
$ run_examples -s -e 1 -l 10
```

Warning: Always call *cleanup()* at the end of your program to free up any resources such as stopping threads.

1.5.2 Example 2: display 3 LEDs

Example 2 consists in displaying three LEDs on channels 9, 10, and 11, respectively. Here is the code along with the output from the terminal:

```
import SimulRPi.GPIO as GPIO

led_channels = [9, 10, 11]
GPIO.setmode(GPIO.BCM)
GPIO.setup(led_channels, GPIO.OUT)
GPIO.output(led_channels, GPIO.HIGH)
GPIO.cleanup()
```

Output:

The command line for reproducing the same results for example 2 with the *run_examples* script is the following:

```
$ run_examples -s -e 2
```

Note: In example 2, we could have also used a `for` loop to setup the output channels and set their states (but more cumbersome):

```
import SimulRPi.GPIO as GPIO

led_channels = [9, 10, 11]
GPIO.setmode(GPIO.BCM)
for ch in led_channels:
    GPIO.setup(ch, GPIO.OUT)
    GPIO.output(ch, GPIO.HIGH)
GPIO.cleanup()
```

The *setup()* function accepts channel numbers as `int`, `list`, and `tuple`. Same with the *output()* function which also accepts channel numbers and output states as `int`, `list`, and `tuple`.

1.5.3 Example 3: detect a pressed key

Example 3 consists in detecting if the key `cmd_r` is pressed and then printing a message. Here is the code along with the output from the terminal:

```
import SimulRPi.GPIO as GPIO

channel = 17
GPIO.setmode(GPIO.BCM)
GPIO.setup(channel, GPIO.IN, pull_up_down=GPIO.PUD_UP)
print("Press key 'cmd_r' to exit\n")
while True:
    if not GPIO.input(channel):
        print("Key pressed!")
        break
GPIO.cleanup()
```

Output:

```
Press key 'cmd_r' to exit


Key pressed!
```

The command line for reproducing the same results for example 3 with the *run_examples* script is the following:

```
$ run_examples -s -e 3 -k cmd_r
```

Note: By default, `SimulRPi` maps the key `cmd_r` to channel 17 as can be seen from the default key-to-channel map.

See also the documentation for SimulRPi.mapping where the default keymap is defined.

1.5.4 Example 4: blink a LED

Example 4 consists in blinking a LED on channel 22 for 4 seconds (or until you press `ctrl` + `c`). Here is the code along with the output from the terminal:

```
import time
import SimulRPi.GPIO as GPIO

channel = 22
GPIO.setmode(GPIO.BCM)
GPIO.setup(channel, GPIO.OUT)
start = time.time()
print("Ex 4: blink a LED for 4.0 seconds\n")
```

(continues on next page)

(continued from previous page)

```
while (time.time() - start) < 4:
    try:
        GPIO.output(channel, GPIO.HIGH)
        time.sleep(0.5)
        GPIO.output(channel, GPIO.LOW)
        time.sleep(0.5)
    except KeyboardInterrupt:
        break
GPIO.cleanup()
```

Output:

The command line for reproducing the same results for example 4 with the *run_examples* script is the following:

```
$ run_examples -s -e 4 -t 4 -l 22
```

1.5.5 Example 5: blink a LED if a key is pressed

Example 5 consists in blinking a LED on channel 10 for 3 seconds if the key `shift_r` is pressed. And then exiting from the program. The program can also be terminated at anytime by pressing `ctrl` + `c`. Here is the code along with the output from the terminal:

```
import time
import SimulRPi.GPIO as GPIO

led_channel = 10
key_channel = 27
GPIO.setmode(GPIO.BCM)
GPIO.setup(led_channel, GPIO.OUT)
GPIO.setup(key_channel, GPIO.IN, pull_up_down=GPIO.PUD_UP)
print("Press the key 'shift_r' to turn on light ...\n")
while True:
    try:
        if not GPIO.input(key_channel):
            print("The key 'shift_r' was pressed!")
            start = time.time()
            while (time.time() - start) < 3:
                GPIO.output(led_channel, GPIO.HIGH)
                time.sleep(0.5)
                GPIO.output(led_channel, GPIO.LOW)
                time.sleep(0.5)
            break
    except KeyboardInterrupt:
        break
GPIO.cleanup()
```

Output:

The command line for reproducing the same results for example 5 with the *run_examples* script is the following:

```
$ run_examples -s -e 5 -t 3 -l 10 -b 27
```

Note: By default, `SimulRPi` maps the key `shift_r` to channel 27 as can be seen from the default key-to-channel map.

See also the documentation for SimulRPi.mapping where the default keymap is defined.

1.6 How to uninstall

To uninstall **only** the package `SimulRPi`:

```
$ pip uninstall simulrpi
```

To uninstall the package `SimulRPi` and its dependency:

```
$ pip uninstall simulrpi pynput
```

1.7 Resources

- SimulRPi GitHub: source code
- SimulRPi PyPI
- Darth-Vader-RPi: personal project using `RPi.GPIO` for activating a Darth Vader action figure with light and sounds and *SimulRPi.GPIO* as fallback if testing on a computer when no RPi is available.

1.8 References

- pynput: a package to control and monitor input devices.
- RPi.GPIO: a module to control RPi GPIO channels.

CHAPTER

TWO

EXAMPLE: HOW TO USE `SIMULRPI`

We will show a code example that makes use of both SimulRPi.GPIO and RPi.GPIO so you can run the script on a Raspberry Pi (RPi) or computer.

2.1 Code example

The following code blinks a LED for 3 seconds after a user presses a push button. The code can be run on an RPi or computer. In the latter case, the simulation package `SimulRPi` is used for displaying a LED in the terminal and monitoring the keyboard.

Listing 1: Script that blinks a LED for 3 seconds when a button (or the key `cmd_r`) is pressed

```
import sys
import time

if len(sys.argv) > 1 and sys.argv[1] == '-s':
    import SimulRPi.GPIO as GPIO
    msg1 = "\nPress key 'cmd_r' to blink a LED"
    msg2 = "Key 'cmd_r' pressed!"
else:
    import RPi.GPIO as GPIO
    msg1 = "\nPress button to blink a LED"
    msg2 = "Button pressed!"

led_channel = 10
button_channel = 17
GPIO.setmode(GPIO.BCM)
GPIO.setup(led_channel, GPIO.OUT)
GPIO.setup(button_channel, GPIO.IN, pull_up_down=GPIO.PUD_UP)
print(msg1)
while True:
    try:
        if not GPIO.input(button_channel):
            print(msg2)
            start = time.time()
```

(continues on next page)

(continued from previous page)

```
            while (time.time() - start) < 3:
                GPIO.output(led_channel, GPIO.HIGH)
                time.sleep(0.5)
                GPIO.output(led_channel, GPIO.LOW)
                time.sleep(0.5)
            break
    except KeyboardInterrupt:
        break
GPIO.cleanup()
```

Add the previous code in a script named for example *script.py*. To run it on your **computer**, use the `-s` option like this:

```
$ python script.py -s
```

If you run it on your **RPi**, connect a LED to the GPIO channel 10 and a push button to the GPIO channel 17. You don't have to add the `-s` option when running the script on the RPi:

```
$ python script.py
```

On your **computer**, you get the following:

Listing 2: Output for the script when it is run on a **computer** (blinking of the LED not shown)

```
$ python script.py -s

Press key 'cmd_r' to blink a LED
Key 'cmd_r' pressed!

    [10]
```

On your **RPi**, you get almost the same result without the LED shown in the terminal:

Listing 3: Output for the script when it is run on an **RPi** (the LED will blink for 3 seconds)

```
$ python script.py

Press button to blink a LED
Button pressed!
```

Note: The script can be stopped at any moment if the keys `ctrl` + `c` are pressed.

2.2 Code explanation

At the beginning of the *script*, we check if the `-s` flag was used. If it is the case, then the simulation module *SimulRPi.GPIO* is imported. Otherwise, the module `RPi.GPIO` is used:

```
if len(sys.argv) > 1 and sys.argv[1] == '-s':
    import SimulRPi.GPIO as GPIO
    msg1 = "\nPress key 'cmd_r' to blink a LED"
    msg2 = "Key 'cmd_r' pressed!"
else:
    import RPi.GPIO as GPIO
    msg1 = "\nPress button to blink a LED"
    msg2 = "Button pressed!"
```

Then, we setup the LED and button channels using the *BCM* mode:

```
led_channel = 10
button_channel = 17
GPIO.setmode(GPIO.BCM)
GPIO.setup(led_channel, GPIO.OUT)
GPIO.setup(button_channel, GPIO.IN, pull_up_down=GPIO.PUD_UP)
```

Finally, we enter the infinite loop where we wait for the push button (or the key `cmd_r`) to be pressed or `ctrl` + `c` which terminates the script immediately. If the push button (or the key `cmd_r`) is pressed, we blink a LED for 3 seconds, then do a cleanup of GPIO channels (very important), and terminate the script:

```
while True:
    try:
        if not GPIO.input(button_channel):
            print(msg2)
            start = time.time()
            while (time.time() - start) < 3:
                GPIO.output(led_channel, GPIO.HIGH)
                time.sleep(0.5)
                GPIO.output(led_channel, GPIO.LOW)
                time.sleep(0.5)
            break
    except KeyboardInterrupt:
        break
GPIO.cleanup()
```

CHAPTER

THREE

USEFUL FUNCTIONS FROM THE API

We present some useful functions from the SimulRPi API along with code examples.

Important: These are functions that are available when working with the simulation module *SimulRPi.GPIO*. Thus, you will always see the following import at the beginning of each code example presented:

```
import SimulRPi.GPIO as GPIO
```

The code examples are to be executed on your computer, not on an RPi since the main reason for these examples is to show how to use the SimulRPi API.

See also:

Example: How to use SimulRPi: It shows you how to integrate the simulation module *SimulRPi.GPIO* with `RPi.GPIO`

Contents

3.1 GPIO.cleanup

cleanup() cleans up any resources at the end of your program. Very importantly, when running in simulation, the threads responsible for displaying "LEDs" in the terminal and listening to the keyboard are stopped. Hence, we avoid the program hanging at the end of its execution.

Here is a simple example on how to use *cleanup()* which should be called at the end of your program:

```
import SimulRPi.GPIO as GPIO

led_channel = 11
GPIO.setmode(GPIO.BCM)
GPIO.setup(led_channel, GPIO.OUT)
GPIO.output(led_channel, GPIO.HIGH)
GPIO.cleanup()
```

Output:

```
  [11]
```

3.2 GPIO.setchannelnames

setchannelnames() sets the channel names for multiple GPIO channels. The channel name will be shown in the terminal along with the LED symbol for each output channel:

```
  [LED 1]          [LED 2]          [LED 3]          [lightsaber]
```

If no channel name is provided for a GPIO channel, its channel number will be shown instead in the terminal.

setchannelnames() takes as argument a dictionary that maps channel numbers (`int`) to channel names (`str`):

```
channel_names = {
    1: "The Channel 1",
    2: "The Channel 2"
}
```

Listing 1: **Example:** updating channel names for two output channels

```
import SimulRPi.GPIO as GPIO

GPIO.setchannelnames({
   10: "led 10",
   11: "led 11"
})
GPIO.setmode(GPIO.BCM)
for ch in [10, 11]:
   GPIO.setup(ch, GPIO.OUT)
   GPIO.output(ch, GPIO.HIGH)
GPIO.cleanup()
```

Output:

```
  [led 10]          [led 11]
```

3.3 GPIO.setchannels

setchannels() sets the attributes for multiple GPIO channels. These attributes are:

- channel_id: unique identifier
- channel_name: will be shown along the LED symbol in the terminal
- channel_number: GPIO channel number based on the numbering system you have specified (*BOARD* or *BCM*).
- led_symbols: should only be defined for output channels. It is a dictionary defining the symbols to be used when the LED is turned ON and turned OFF.
- key: should only be defined for input channels. The names of keyboard keys that you can use are those specified in the SimulRPi's API documentation, e.g. *media_play_pause*, *shift*, and *shift_r*.

setchannels() accepts as argument a list where each item is a dictionary defining the attributes for a given GPIO channel.

Example: updating attributes for an input and output channels. Then when the user presses cmd_r, we blink a LED for 3 seconds

```
import time
import SimulRPi.GPIO as GPIO

key_channel = 23
led_channel = 10
gpio_channels = [
    {
        "channel_id": "button",
        "channel_name": "The button",
        "channel_number": key_channel,
        "key": "cmd_r"
    },
    {
        "channel_id": "led",
        "channel_name": "The LED",
        "channel_number": led_channel,
        "led_symbols": {
            "ON": "",
            "OFF": " "
        }
    }
]
GPIO.setchannels(gpio_channels)
GPIO.setmode(GPIO.BCM)
GPIO.setup(key_channel, GPIO.IN, pull_up_down=GPIO.PUD_UP)
GPIO.setup(led_channel, GPIO.OUT)
print("Press key 'cmd_r' to blink a LED")
while True:
   try:
        if not GPIO.input(key_channel):
            print("Key 'cmd_r' pressed")
            start = time.time()
            while (time.time() - start) < 3:
                GPIO.output(led_channel, GPIO.HIGH)
                time.sleep(0.5)
                GPIO.output(led_channel, GPIO.LOW)
```

(continues on next page)

(continued from previous page)

```
                time.sleep(0.5)
            break
        except KeyboardInterrupt:
            break
    GPIO.cleanup()
```

Output: blinking not shown

```
Press key 'cmd_r' to blink a LED
Key 'cmd_r' pressed

    [The LED]
```

Note: In the previous example, we changed the default keyboard key associated with the GPIO channel 23 from `media_volume_mute` to `cmd_r`.

```
    key_channel = 23
    led_channel = 10
    gpio_channels = [
        {
            "channel_id": "button",
            "channel_name": "The button",
            "channel_number": key_channel,
            "key": "cmd_r"
        },
      ...
```

3.4 GPIO.setdefaultsymbols

setdefaultsymbols() sets the default LED symbols used by **all output** channels. It accepts as argument a dictionary that maps an output state ('*ON*', '*OFF*') to a LED symbol (`str`).

By default, these are the LED symbols used by all output channels:

```
default_led_symbols = {
    'ON': '',
    'OFF': ''
}
```

The next example shows you how to change these default LED symbols with the function *setdefaultsymbols()*

Listing 2: **Example:** updating the default LED symbols and toggling a LED

```
    import time
    import SimulRPi.GPIO as GPIO

    GPIO.setdefaultsymbols(
        {
            'ON': '',
            'OFF': ' '
        }
```

(continues on next page)

(continued from previous page)

```
    )
    led_channel = 11
    GPIO.setmode(GPIO.BCM)
    GPIO.setup(led_channel, GPIO.OUT)
    GPIO.output(led_channel, GPIO.HIGH)
    time.sleep(0.5)
    GPIO.output(led_channel, GPIO.LOW)
    time.sleep(0.5)
    GPIO.cleanup()
```

Output: blinking not shown

```
    [11]
```

3.5 GPIO.setkeymap

setkeymap() sets the default keymap dictionary with a new mapping between keyboard keys and channel numbers.

It takes as argument a dictionary mapping keyboard keys (`str`) to GPIO channel numbers (`int`):

```
key_to_channel_map = {
    "cmd": 23,
    "alt_r": 24,
    "ctrl_r": 25
}
```

Listing 3: **Example:** by default, `cmd_r` is mapped to channel 17. We change this mapping by associating `ctrl r` to channel 17.

```
import SimulRPi.GPIO as GPIO

channel = 17
GPIO.setkeymap({
    'ctrl_r': channel
})
GPIO.setmode(GPIO.BCM)
GPIO.setup(channel, GPIO.IN, pull_up_down=GPIO.PUD_UP)
print("Press key 'ctrl_r' to exit")
while True:
    if not GPIO.input(channel):
        print("Key 'ctrl_r' pressed!")
        break
GPIO.cleanup()
```

Output:

```
Press key 'ctrl_r' to exit
Key 'ctrl_r' pressed!
```

3.6 GPIO.setprinting

setprinting() enables or disables printing the LED symbols and channel names/numbers to the terminal.

Listing 4: **Example:** disable printing to the terminal

```
import SimulRPi.GPIO as GPIO

GPIO.setprinting(False)
led_channel = 11
GPIO.setmode(GPIO.BCM)
GPIO.setup(led_channel, GPIO.OUT)
GPIO.output(led_channel, GPIO.HIGH)
GPIO.cleanup()
```

3.7 GPIO.setsymbols

setsymbols() sets the LED symbols for multiple **output** channels. It takes as argument a dictionary mapping channel numbers (`int`) to LED symbols (`dict`):

```
led_symbols = {
    1: {
        'ON': '',
        'OFF': ' '
    },
    2: {
        'ON': '',
        'OFF': ' '
    }
}
```

There is a LED symbol for each output state (*ON* and *OFF*) for a given output channel.

Listing 5: **Example:** set the LED symbols for a GPIO channel

```
import time
import SimulRPi.GPIO as GPIO

GPIO.setsymbols({
    11: {
        'ON': '',
        'OFF': ' '
    }
})
led_channel = 11
GPIO.setmode(GPIO.BCM)
GPIO.setup(led_channel, GPIO.OUT)
GPIO.output(led_channel, GPIO.HIGH)
time.sleep(0.5)
GPIO.output(led_channel, GPIO.LOW)
time.sleep(0.5)
GPIO.cleanup()
```

Output: blinking not shown

```
[11]
```

3.8 GPIO.wait

wait() waits for the threads to do their tasks. If there was an exception caught by one of the threads, then it is raised by *wait()*.

Thus it is ideal for *wait()* to be called within a `try` block after you are done with the *SimulRPi.GPIO* API:

```
try:
    do_something_with_gpio_api()
    GPIO.wait()
except Exception as e:
    # Do something with error
finally:
   GPIO.cleanup()
```

wait() takes as argument the number of seconds you want to wait at most for the threads to accomplish their tasks.

Example: wait for the threads to do their jobs and if there is an exception in one of the threads' target function or callback, it will be caught in our `except` block.

```
import time
import SimulRPi.GPIO as GPIO

try:
   led_channel = 11
   GPIO.setmode(GPIO.BCM)
   GPIO.setup(led_channel, GPIO.OUT)
   GPIO.output(led_channel, GPIO.HIGH)
   GPIO.wait(1)
except Exception as e:
   # Could be an exception raised in a thread's target function or callback
   # from SimulRPi library
   print(e)
finally:
   GPIO.cleanup()
```

Important: If we don't use *wait()* in the previous example, we won't be able to catch any exception occurring in a thread's target function or callback since the threads simply catch and save the exceptions but don't raise them. *wait()* takes care of raising an exception if it was already caught and saved by a thread.

Also, the reason for not raising the exception within a thread's `run` method or its callback is because the main program will not be able to catch it. The thread's exception needs to be raised outside of the thread's `run` method or callback so the main program can further catch it. And this is what *input()*, *output()*, and *wait()* do: they raise the thread's exception so the main program can catch it and process it down the line.

See Test threads raising exceptions about some tests done to check what happens when a thread raises an exception within its `run` method or callback (**spoiler:** not good!).

CHAPTER

FOUR

DISPLAY PROBLEMS

4.1 Non-ASCII characters can't be displayed

When running the *`SimulRPi.run_examples`* script or using the *`SimulRPi.GPIO`* module in your own code, your terminal might have difficulties printing the default LED symbols based on special characters:

```
UnicodeEncodeError: 'ascii' codec can't encode character '\U0001f6d1' in position 2:␣
↪ordinal not in range(128)
```

This is mainly a problem with your **locale** settings used by your terminal.

4.1.1 Solution #1: change your locale settings (best solution)

The best solution consists in fixing your **locale** settings since it is permanent and you don't have to change any Python code.

1. Append `~/.bashrc` or `~/.bash_profile` with:

   ```
   export LANG="en_US.UTF-8"
   export LANGUAGE="en_US:en"
   ```

   You should provide your own **UTF-8** based locale settings. The example uses the English (US) locale with the encoding **UTF-8**. The `locale -a` command gives you all the available locales on your Linux or Unix-like system.

2. Reload the `.bashrc`:

   ```
   $ source .bashrc
   ```

3. Run the `locale` command to make sure that your locale settings were set correctly:

```
$ locale

LANG="en_US.UTF-8"
LC_COLLATE="en_US.UTF-8"
LC_CTYPE="en_US.UTF-8"
LC_MESSAGES="en_US.UTF-8"
LC_MONETARY="en_US.UTF-8"
LC_NUMERIC="en_US.UTF-8"
LC_TIME="en_US.UTF-8"
LC_ALL=
```

4. Run the *SimulRPi.run_examples* script to test if you can display the LED symbols fine using the correct encoding **UTF-8**:

   ```
   $ run_examples -s -e 1
   ```

   Output:

See also:

- How to Set Locales (i18n) On a Linux or Unix: detailed article
- How can I change the locale?: from *raspberrypi.stackexchange.com*, provides answers to set the locale user and system-wide

4.1.2 Solution #2: `export PYTHONIOENCODING=utf8` (temporary solution)

Before running the *SimulRPi.run_examples* script, export the environment variable `PYTHONIOENCODING` with the correct encoding:

```
$ export PYTHONIOENCODING=utf8
$ run_examples -s -e 1
```

Output:

However, this is **not a permanent solution** because if you use another terminal, you will have to export `PYTHONIOENCODING` again before running the script.

4.1.3 Use ASCII-based LED symbols

If you tried the *previous two solutions*, and you still can't display the LED symbols that use special characters (UTF-8 encoding), you can instead opt for ASCII-based LED symbols.

Method #1: use the `SimulRPi.GPIO` API

If you are using the *`SimulRPi.GPIO`* module in your code, you can change the default LED symbols used by all output channels with the function *`setdefaultsymbols()`*. Hence, you can provide your own ASCII-based LED symbols using ANSI codes to color them:

Listing 1: **Example:** updating the default LED symbols with ASCII characters and ANSI codes

```
import time
import SimulRPi.GPIO as GPIO

GPIO.setdefaultsymbols(
    {
        'ON': '\033[91m(0)\033[0m',
        'OFF': '(0)'
    }
)
led_channel = 11
GPIO.setmode(GPIO.BCM)
GPIO.setup(led_channel, GPIO.OUT)
GPIO.output(led_channel, GPIO.HIGH)
GPIO.cleanup()
```

Or you can provide the argument `"default_ascii"` to the function *`setdefaultsymbols()`* which will provide default ASCII-based LED symbols for you:

```
GPIO.setdefaultsymbols("default_ascii")
```

Output:

Note: If working with the Darth-Vader-RPi library, you can use ASCII LED symbols when running the `start_dv` script by assigning the value *"default_ascii"* to the `default_led_symbols` setting in the main configuration file:

```
"default_led_symbols": "default_ascii",
```

See also:

- Build your own Command Line with ANSI escape codes : more info about using ANSI escape codes (e.g. color text, move the cursor up)

- How to print colored text in Python? : from *stackoverflow*, lots of Python examples using built-in modules or third-party libraries to color text in the terminal.

Method #2: use the command-line option `-a`

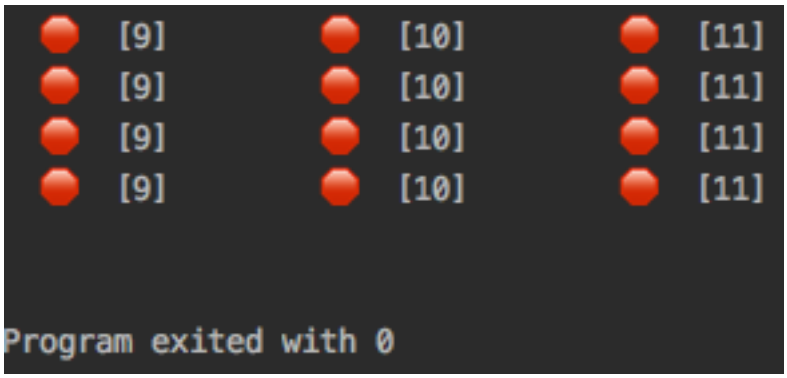
When running the *SimulRPi.run_examples* script, you can use the command-line option `-a` which will make use of ASCII-based LED symbols:

```
$ run_examples -s -e -1 -a
```

Output:

4.2 Multiple lines of LED symbols

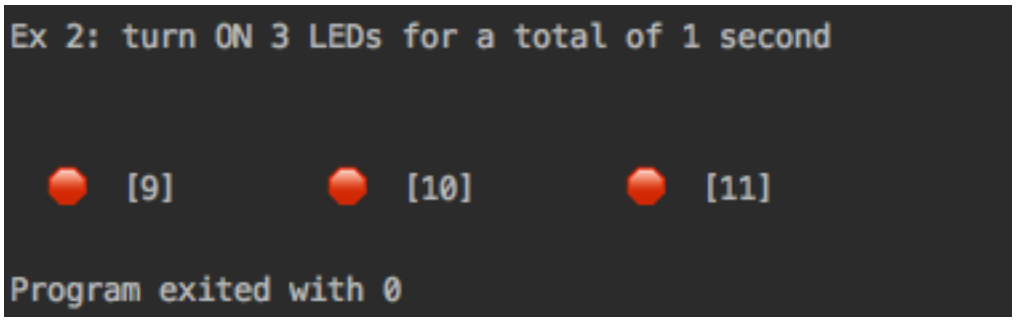
When running the *SimulRPi.run_examples* script, if you get the following:



It means that you are running the script within a too small terminal window, less than the length of a displayed line.

4.2.1 Solution: enlarge the window

The solution is to simply **enlarge** your terminal window a little bit:



Technical explanation: the script is supposed to display the LEDs turning ON and OFF always on the same line. That is, when a line of LEDs is displayed, the script goes to the beginning of the line to display the next state of LEDs by printing over the previous LEDs.

However, when the window is too small, the first line of LEDs that gets printed overflows on the second line since there is not enough space to print everything on the first line. Then, the script won't be able to overwrite the first line of LEDs because it will be positioned on the second line instead. So you get this display of multiple lines of LEDs.

CHAPTER

FIVE

API REFERENCE

5.1 SimulRPi.GPIO

Module that partly fakes RPi.GPIO and simulates some I/O devices.

It simulates these I/O devices connected to a Raspberry Pi:

- push buttons by listening to pressed keyboard keys and
- LEDs by displaying red dots blinking in the terminal along with their GPIO channel number.

When a LED is turned on, it is shown as a red dot in the terminal. The pynput package is used to monitor the keyboard for any pressed key.

Example: terminal output

```
 [9]    [10]    [11]
```

where each dot represents a LED and the number between brackets is the associated GPIO channel number.

Important: This library is not a Raspberry Pi emulator nor a complete mock-up of RPi.GPIO, only the most important functions that I needed for my Darth-Vader-RPi project were added.

If there is enough interest in this library, I will eventually mock more functions from RPi.GPIO.

SimulRPi.GPIO.**cleanup**()
Clean up any resources (e.g. GPIO channels).

At the end of any program, it is good practice to clean up any resources you might have used. This is no different with RPi.GPIO. By returning all channels you have used back to inputs with no pull up/down, you can avoid accidental damage to your RPi by shorting out the pins. [**Ref:** RPi.GPIO wiki]

Also, the two threads responsible for displaying LEDs in the terminal and listening for pressed/released keys are stopped.

Note: On an RPi, `cleanup()` will:

- only clean up GPIO channels that your script has used
- also clear the pin numbering system in use (*BOARD* or *BCM*)

Ref.: RPi.GPIO wiki

When using the `SimulRPi` package, *`cleanup()`* will:

- stop the displaying thread `Manager.th_display_leds`
- stop the listening thread `Manager.th_listener`
- show the cursor again which was hidden in *`display_leds()`*
- reset the `GPIO.manager`'s attributes (an instance of *`Manager`*)

`SimulRPi.GPIO.`**`input`**(*channel*)

Read the value of a GPIO pin.

The listening thread is also started if possible.

Parameters **`channel`** (*`int`*) – Input channel number based on the numbering system you have specified (*BOARD* or *BCM*).

Returns **state** – If no *`Pin`* could be retrieved based on the given channel number, then `None` is returned. Otherwise, the *`Pin`*'s state is returned: 1 (*HIGH*) or 0 (*LOW*).

Return type `int` or `None`

Raises **`Exception`** – If the listening thread caught an exception that occurred in *`on_press()`* or *`on_release()`*, the said exception will be raised here.

Note: The listening thread (for monitoring pressed keys) is started if there is no exception caught by the thread and if it is not alive, i.e. it is not already running.

Important: The reason for checking if there is no exception already caught by a thread, i.e. `if not manager.th_listener.exc`, is to avoid having another thread calling this function and re-starting the failed thread. Hence, we avoid raising a `RuntimeError` on top of the thread's already caught exception.

`SimulRPi.GPIO.`**`output`**(*channel*, *state*)

Set the output state of a GPIO pin.

The displaying thread is also started if possible.

Parameters

- **`channel`** (*`int or list or tuple`*) – Output channel number based on the numbering system you have specified (*BOARD* or *BCM*).

  You can also provide a list or tuple of channel numbers:

```
chan_list = [11,12]
```

- **state** (*int or list or tuple*) – State of the GPIO channel: 1 (*HIGH*) or 0 (*LOW*).

  You can also provide a list of states:

```
chan_list = [11,12]
GPIO.output(chan_list, GPIO.LOW)                # sets all to LOW
GPIO.output(chan_list, (GPIO.HIGH, GPIO.LOW))   # sets 1st HIGH and
↪2nd LOW.
```

Raises **Exception** – If the displaying thread caught an exception that occurred in its target function *display_leds()*, the said exception will be raised here.

Note: The displaying thread (for showing "LEDs" on the terminal) is started if there is no exception caught by the thread and if it is not alive, i.e. it is not already running.

See also:

input() Read the **Important** message about why we need to check if there is an exception caught by the thread when trying to start it.

SimulRPi.GPIO.**setchannelnames**(*channel_names*)

Set the channel names for multiple channels

The channel names will be displayed in the terminal along each LED symbol. If no channel name is given, then the channel number will be shown.

Parameters **channel_names** (*dict*) – Dictionary that maps channel numbers (int) to channel names (str).

Example:

```
channel_names = {
    1: "The Channel 1",
    2: "The Channel 2"
}
```

SimulRPi.GPIO.**setchannels**(*gpio_channels*)

Set the attributes (e.g. *channel_name* and *led_symbols*) for multiple channels.

The attributes that can be updated for a given GPIO channel are:

- channel_id: unique identifier
- channel_name: will be shown along the LED symbol in the terminal
- channel_number: GPIO channel number based on the numbering system you have specified (*BOARD* or *BCM*).
- led_symbols: should only be defined for output channels. It is a dictionary defining the symbols to be used when the LED is turned ON and OFF.
- key: keyboard key associated with a channel, e.g. "*cmd_r*".

Parameters **gpio_channels** (*list*) – A list where each item is a dictionary defining the attributes for a given GPIO channel.

Example:

```
gpio_channels = [
    {
        "channel_id": "lightsaber_button",
        "channel_name": "lightsaber_button",
        "channel_number": 23,
        "key": "cmd"
    },
    {
        "channel_id": "lightsaber_led",
        "channel_name": "lightsaber",
        "channel_number": 22,
        "led_symbols": {
            "ON": "\033[1;31;48m\033[1;37;0m",
            "OFF": ""
        }
    }
]
```

Raises **KeyError** – Raised if two channels are using the same channel number.

SimulRPi.GPIO.**setdefaultsymbols**(*default_led_symbols*)

Set the default LED symbols used by all output channels.

Parameters **default_led_symbols** (*str or dict*) – Dictionary that maps each output state (str, {'*ON*', '*OFF*'}) to the LED symbol (str).

Example:

```
default_led_symbols = {
    'ON': '',
    'OFF': ' '
}
```

You can also provide the string default_ascii to make use of ASCII-based LED symbols for all output channels. Useful if you are still having problems displaying the default LED signs (which make use of special characters) after you have tried the solutions shown here:

```
default_led_symbols = "default_ascii"
```

SimulRPi.GPIO.**setkeymap**(*key_to_channel_map*)

Set the default keymap dictionary with new keys and channels.

The default dictionary default_key_to_channel_map that maps keyboard keys to GPIO channels can be modified by providing your own mapping key_to_channel_map containing only the keys and channels that you want to be modified.

Parameters **key_to_channel_map** (*dict*) – A dictionary mapping keys (str) to GPIO channel numbers (int) that will be used to update the default keymap.

For example:

```
key_to_channel_map = {
    "q": 23,
    "w": 24,
    "e": 25
}
```

SimulRPi.GPIO.**setmode**(*mode*)

Set the numbering system used to identify the I/O pins on an RPi within RPi.GPIO.

There are two ways of numbering the I/O pins on a Raspberry Pi within `RPi.GPIO`:

1. The *BOARD* numbering system: refers to the pin numbers on the P1 header of the Raspberry Pi board
2. The *BCM* numbers: refers to the channel numbers on the Broadcom SOC.

Parameters **mode** (*int*) – Numbering system used to identify the I/O pins on an RPi: *BOARD* or *BCM*.

References

Function description and more info from RPi.GPIO wiki.

`SimulRPi.GPIO.`**`setprinting`**(*enable_printing*)
Enable or disable printing to the terminal.

If printing is enabled, blinking red dots will be shown in the terminal, simulating LEDs connected to a Raspberry Pi. Otherwise, nothing will be printed in the terminal.

Parameters **enable_printing** (*bool*) – If *True*. printing to the terminal is enabled. Otherwise, printing will be disabled.

`SimulRPi.GPIO.`**`setsymbols`**(*led_symbols*)
Set the LED symbols for multiple output channels.

Parameters **led_symbols** (*dict*) – Dictionary that maps channel numbers (`int`) to LED symbols (`dict`).

Example:

```
led_symbols = {
    1: {
        'ON': '',
        'OFF': ' '
    },
    2: {
        'ON': '',
        'OFF': ' '
    }
}
```

`SimulRPi.GPIO.`**`setup`**(*channel*, *channel_type*, *pull_up_down=None*, *initial=None*)
Setup a GPIO channel as an input or output.

To configure a channel as an input:

```
GPIO.setup(channel, GPIO.IN)
```

To configure a channel as an output:

```
GPIO.setup(channel, GPIO.OUT)
```

You can also specify an initial value for your output channel:

```
GPIO.setup(channel, GPIO.OUT, initial=GPIO.HIGH)
```

Parameters

- **channel** (*int or list or tuple*) – GPIO channel number based on the numbering system you have specified (*BOARD* or *BCM*).

  You can also provide a list or tuple of channel numbers. All channels will take the same values for the other parameters.

- **channel_type** (*int*) – Type of a GPIO channel: e.g. 1 (*GPIO.IN*) or 0 (*GPIO.OUT*).
- **pull_up_down** (*int or None, optional*) – Initial value of an input channel, e.g. *GPIO.PUP_UP*. Default value is `None`.
- **initial** (*int or None, optional*) – Initial value of an output channel, e.g. *GPIO.HIGH*. Default value is `None`.

References

RPi.GPIO wiki

`SimulRPi.GPIO.`**setwarnings**(*show_warnings*)

Set warnings when configuring a GPIO pin other than the default (input).

It is possible that you have more than one script/circuit on the GPIO of your Raspberry Pi. As a result of this, if `RPi.GPIO` detects that a pin has been configured to something other than the default (input), you get a warning when you try to configure a script. [**Ref:** RPi.GPIO wiki]

Parameters **show_warnings** (*bool*) – Whether to show warnings when using a pin other than the default GPIO function (input).

`SimulRPi.GPIO.`**wait**(*timeout=2*)

Wait for certain events to complete.

Wait for the displaying and listening threads to do their tasks. If there was an exception caught and saved by one thread, then it is raised here.

If more than `timeout` seconds elapsed without any of the events described previously happening, the function exits.

Parameters **timeout** (*float*) – How long to wait (in seconds) before exiting from this function. By default, we wait for 2 seconds.

Raises **Exception** – If the displaying or listening thread caught an exception, it will be raised here.

Important: This function is not called in *cleanup()* because if a thread exception is raised, it will not be caught in the main program because *cleanup()* should be found in a `finally` block:

```
try:
    do_something_with_gpio_api()
    GPIO.wait()
except Exception as e:
    # Do something with error
    print(e)
finally:
    GPIO.cleanup()
```

5.2 SimulRPi.manager

Module that manages the *PinDB* database, threads, and default keymap.

The threads are responsible for displaying LEDs in the terminal and listening to the keyboard.

The default keymap maps keyboard keys to GPIO channel numbers and is defined in default_key_to_channel_map.

class SimulRPi.manager.**DisplayExceptionThread**(**args*, ***kwargs*)

Bases: threading.Thread

A subclass from threading.Thread that defines threads that can catch errors if their target functions raise an exception.

Variables

- **exception_raised** (*bool*) – When the exception is raised, it should be set to *True*. By default, it is *False*.
- **exc** (Exception) – Represents the exception raised by the target function.

References

- stackoverflow

run()

Method representing the thread's activity.

Overridden from the base class threading.Thread. This method invokes the callable object passed to the object's constructor as the target argument, if any, with sequential and keyword arguments taken from the args and kwargs arguments, respectively.

It also catches and saves any error that the target function might raise.

Important: The exception is only caught here, not raised. The exception is further raised in *SimulRPi.GPIO.output()* or *SimulRPi.GPIO.wait()*. The reason for not raising it here is because the main program won't catch it. The exception must be raised outside the thread's run method so that the thread's exception can be caught by the main program.

The same reasoning applies to the listening thread's callbacks *Manager.on_press()* and *Manager.on_release()*.

---

class SimulRPi.manager.**Manager**

Bases: object

Class that manages the pin database (*SimulRPi.pindb.PinDB*), the threads responsible for displaying "LEDs" in the terminal and listening for pressed/released keys, and the default keymap.

The threads are not started right away in __init__() but in *SimulRPi.GPIO.input()* for the listening thread and *SimulRPi.GPIO.output()* for the displaying thread.

They are eventually stopped in *SimulRPi.GPIO.cleanup()*.

The default keymap maps keyboard keys to GPIO channel numbers and is defined in default_key_to_channel_map.

Variables

- **mode** (*int*) – Numbering system used to identify the I/O pins on an RPi: *BOARD* or *BCM*. Default value is None.

- **warnings** (*bool*) – Whether to show warnings when using a pin other than the default GPIO function (input). Default value is *True*.
- **enable_printing** (*bool*) – Whether to enable printing on the terminal. Default value is *True*.
- **pin_db** (PinDB) – A database of *Pin*s. See *PinDB* on how to access it.
- **default_led_symbols** (*dict*) – A dictionary that maps each output channel's state ('ON' and 'OFF') to a LED symbol. By default, it is set to these LED symbols:

```
default_led_symbols = {
    "ON": "",
    "OFF": ""
}
```

- **key_to_channel_map** (*dict*) – A dictionary that maps keyboard keys (string) to GPIO channel numbers (int). By default, it takes the keys and values defined in the keymap default_key_to_channel_map.
- **channel_to_key_map** (*dict*) – The reverse dictionary of key_to_channel_map. It maps channels to keys.
- **th_display_leds** (manager.DisplayExceptionThread) – Thread responsible for displaying blinking red dots in the terminal as to simulate LEDs connected to an RPi.
- **th_listener** (*manager.KeyboardExceptionThread*) – Thread responsible for listening on any pressed or released keyboard key as to simulate push buttons connected to an RPi.

  If pynput couldn't be imported, th_listener is None. Otherwise, it is instantiated from manager.KeyboardExceptionThread.

  Note: A keyboard listener is a subclass of threading.Thread, and all callbacks will be invoked from the thread.

  Ref.: https://pynput.readthedocs.io/en/latest/keyboard.html#monitoring-the-keyboard

---

Important: If the pynput.keyboard module couldn't be imported, the listening thread th_listener will not be created and the parts of the SimulRPi library that monitors the keyboard for any pressed or released key will be ignored. Only the thread th_display_leds that displays "LEDs" in the terminal will be created.

This is necessary for example in the case we are running tests on travis and we don't want travis to install pynput in a headless setup because the following exception will get raised:

```
Xlib.error.DisplayNameError: Bad display name ""
```

The tests involving pynput will be performed with a mock version of pynput.

---

add_pin(*channel_number*, *channel_type*, *pull_up_down=None*, *initial=None*)

Add an input or output pin to the pin database.

An instance of *Pin* is created with the given arguments and added to the pin database *PinDB*.

Parameters

- **channel_number** (*int*) – GPIO channel number associated with the *Pin* to be added in the pin database.

- **channel_type** (*int*) – Type of a GPIO channel: e.g. 1 (*GPIO.IN*) or 0 (*GPIO.OUT*).
- **pull_up_down** (*int or None, optional*) – Initial value of an input channel, e.g. *GPIO.PUP_UP*. Default value is None.
- **initial** (*int or None, optional*) – Initial value of an output channel, e.g. *GPIO.HIGH*. Default value is None.

bulk_channel_update(*new_channels_attributes*)

Update the attributes (e.g. *channel_name* and *led_symbols*) for multiple channels.

If a channel number is associated with a not yet created *Pin*, the corresponding attributes will be temporary saved for later when the pin object will be created with *add_pin()*.

Parameters **new_channels_attributes** (*dict*) – A dictionary mapping channel numbers (int) with channels' attributes (dict). The accepted attributes are those specified in *SimulRPi.GPIO.setchannels()*.

Example:

```
new_channels_attributes = {
    1: {
        'channel_id': 'channel1',
        'channel_name': 'The Channel 1',
        'led_symbols': {
            'ON': '',
            'OFF': ' '
        }
    }.
    2: {
        'channel_id': 'channel2',
        'channel_name': 'The Channel 2',
        'key': 'cmd_r'
    }
}
```

display_leds()

Displaying thread's **target function** that simulates LEDs connected to an RPi by blinking red dots in a terminal.

Example: terminal output

```
 [9]    [10]    [11]
```

where each dot represents a LED and the number between brackets is the associated GPIO channel number.

Important: *display_leds()* should be run by a thread and eventually stopped from the main program by setting its do_run attribute to *False* to let the thread exit from its target function.

For example:

```
th = DisplayExceptionThread(target=self.display_leds, args=())
th.start()

# Your other code ...

# Time to stop thread
th.do_run = False
th.join()
```

Note: If `enable_printing` is set to *True*, the terminal's cursor will be hidden. It will be eventually shown again in *SimulRPi.GPIO.cleanup()* which is called by the main program when it is exiting.

The reason is to avoid messing with the display of LEDs done by the displaying thread `th_display_leds`.

Note: Since the displaying thread `th_display_leds` is an *DisplayExceptionThread* object, it has an attribute `exc` which stores the exception raised by this target function.

static get_key_name(*key*)
Get the name of a keyboard key as a string.

The name of the special or alphanumeric key is given by the pynput package.

Parameters **key** (*pynput.keyboard.Key or pynput.keyboard.KeyCode*) – The keyboard key (from `pynput.keyboard`) whose name will be returned.

Returns **key_name** – Returns the name of the given keyboard key if one was found by pynput. Otherwise, it returns `None`.

Return type str or None

on_press(*key*)
When a valid keyboard key is pressed, set the associated pin's state to *GPIO.LOW*.

Callback invoked from the thread `th_listener`.

This thread is used to monitor the keyboard for any valid pressed key. Only keys defined in the pin database are treated, i.e. keys that were configured with *SimulRPi.GPIO.setup()* are further processed.

Once a valid key is detected as pressed, the associated pin's state is changed to *GPIO.LOW*.

Parameters **key** (*pynput.keyboard.Key, pynput.keyboard.KeyCode, or None*) – The key parameter passed to callbacks is

- a `pynput.keyboard.Key` for special keys,
- a `pynput.keyboard.KeyCode` for normal alphanumeric keys, or
- `None` for unknown keys.

Ref.: https://bit.ly/3k4whEs

Note: If an exception is raised, it is caught to be further raised in *SimulRPi.GPIO.input()* or *SimulRPi.GPIO.wait()*.

See also:

DisplayExceptionThread() Read the **Important** message that explains why an exception is not raised in a thread's callback or target function.

on_release(*key*)
When a valid keyboard key is released, set the associated pin's state to *GPIO.HIGH*.

Callback invoked from the thread `th_listener`.

This thread is used to monitor the keyboard for any valid released key. Only keys defined in the pin database are treated, i.e. keys that were configured with *SimulRPi.GPIO.setup()* are further processed.

Once a valid key is detected as released, the associated pin's state is changed to *GPIO.HIGH*.

Parameters **key** (*pynput.keyboard.Key*, *pynput.keyboard.KeyCode*, *or None*) – The key parameter passed to callbacks is

- a pynput.keyboard.Key for special keys,
- a pynput.keyboard.KeyCode for normal alphanumeric keys, or
- None for unknown keys.

Ref.: https://bit.ly/3k4whEs

Note: If an exception is raised, it is caught to be further raised in *SimulRPi.GPIO.input()* or *SimulRPi.GPIO.wait()*.

See also:

DisplayExceptionThread() Read the **Important** message that explains why an exception is not raised in a thread's callback or target function.

update_channel_names(*new_channel_names*)

Update the channels names for multiple channels.

If a channel number is associated with a not yet created *Pin*, the corresponding *channel_name* will be temporary saved for later when the pin object will be created with *add_pin()*.

Parameters **new_channel_names** (*dict*) – Dictionary that maps channel numbers (int) to channel names (str).

Example:

```
new_channel_names = {
    1: "The Channel 1",
    2: "The Channel 2"
}
```

update_default_led_symbols(*new_default_led_symbols*)

Update the default LED symbols used by all output channels.

Parameters **new_default_led_symbols** (*dict*) – Dictionary that maps each output state (str, {'*ON*', '*OFF*'}) to a LED symbol (str).

Example:

```
new_default_led_symbols = {
    'ON': '',
    'OFF': ' '
}
```

update_keymap(*new_keymap*)

Update the default dictionary mapping keys and GPIO channels.

new_keymap is a dictionary mapping some keys to their new GPIO channels, and will be used to update the default keymap default_key_to_channel_map.

Parameters **new_keymap** (*dict*) – Dictionary that maps keys (str) to their new GPIO channels (int).

Example:

```
new_keymap = {
    "f": 24,
    "g": 25,
    "h": 23
}
```

Raises **TypeError** – Raised if a given key is invalid: only special and alphanumeric keys recognized by pynput are accepted.

See the documentation for *SimulRPi.mapping* for a list of accepted keys.

Note: If the key to be updated is associated to a channel that is already taken by another key, both keys' channels will be swapped. However, if a key is being linked to a None channel, then it will take on the maximum channel number available + 1.

---

update_led_symbols(*new_led_symbols*)

Update the LED symbols for multiple channels.

If a channel number is associated with a not yet created *Pin*, the corresponding LED symbols will be temporary saved for later when the pin object will be created with *add_pin()*.

Parameters **new_led_symbols** (*dict*) – Dictionary that maps channel numbers (int) to LED symbols (dict).

Example:

```
new_led_symbols = {
    1: {
        'ON': '',
        'OFF': ' '
    },
    2: {
        'ON': '',
        'OFF': ' '
    }
}
```

static validate_key(*key*)

Validate if a key is recognized by pynput

A valid key can either be:

- a pynput.keyboard.Key for special keys (e.g. tab or up), or
- a pynput.keyboard.KeyCode for normal alphanumeric keys.

Parameters **key** (*str*) – The key (e.g. '*tab*') that will be validated.

Returns **retval** – Returns *True* if it's a valid key. Otherwise, it returns *False*.

Return type bool

References

pynput

See also:

SimulRPi.mapping for a list of special keys supported by pynput.

5.3 SimulRPi.mapping

Module that defines the *dictionary* that maps keys to GPIO channels.

This module defines the default mapping between keyboard keys and GPIO channels. It is used by *SimulRPi.manager* when monitoring the keyboard with the package pynput for any pressed/released key as to simulate a push button connected to a Raspberry Pi.

Notes

In early RPi models, there are 17 GPIO channels and in late RPi models, there are 28 GPIO channels.

By default, 28 GPIO channels (from 0 to 27) are mapped to alphanumeric and special keys. See the *content of the default keymap*.

Here is the full list of special keys you can use with info about some of them (taken from pynput reference):

- alt
- alt_gr
- alt_l
- alt_r
- backspace
- caps_lock
- cmd: A generic command button. On PC platforms, this corresponds to the Super key or Windows key, and on Mac it corresponds to the Command key.
- cmd_l: The left command button. On PC platforms, this corresponds to the Super key or Windows key, and on Mac it corresponds to the Command key.
- cmd_r: The right command button. On PC platforms, this corresponds to the Super key or Windows key, and on Mac it corresponds to the Command key.
- ctrl: A generic Ctrl key.
- ctrl_l
- ctrl_r
- delete
- down
- end
- enter
- esc

- `f1`: The function keys. F1 to F20 are defined.
- `home`
- `insert`: The Insert key. This may be undefined for some platforms.
- `left`
- `media_next`
- `media_play_pause`
- `media_previous`
- `media_volume_down`
- `media_volume_mute`
- `media_volume_up`
- `menu`: The Menu key. This may be undefined for some platforms.
- `num_lock`: The NumLock key. This may be undefined for some platforms.
- `page_down`
- `page_up`
- `pause`: The Pause/Break key. This may be undefined for some platforms.
- `print_screen`: The PrintScreen key. This may be undefined for some platforms.
- `right`
- `scroll_lock`
- `shift`
- `shift_l`
- `shift_r`
- `space`
- `tab`
- `up`

References

- **RPi Header**: https://bit.ly/30ZM2Uj
- **pynput**: https://pynput.readthedocs.io/

Important: *`SimulRPi.GPIO.setkeymap()`* allows you to modify the default keymap.

Content of the default keymap dictionary (*key:* keyboard key as `string`, *value:* GPIO channel as `int`):

```
default_key_to_channel_map = {
    "0": 0,  # sudo on mac
    "1": 1,  # sudo on mac
    "2": 2,  # sudo on mac
    "3": 3,  # sudo on mac
    "4": 4,  # sudo on mac
```

(continues on next page)

(continued from previous page)

```
    "5": 5,  # sudo on mac
    "6": 6,  # sudo on mac
    "7": 7,  # sudo on mac
    "8": 8,  # sudo on mac
    "9": 9,  # sudo on mac
    "q": 10,  # sudo on mac
    "alt": 11,  # left alt on mac
    "alt_l": 12,  # not recognized on mac
    "alt_r": 13,
    "alt_gr": 14,
    "cmd": 15,  # left cmd on mac
    "cmd_l": 16,  # not recognized on mac
    "cmd_r": 17,
    "ctrl": 18,  # left ctrl on mac
    "ctrl_l": 19,  # not recognized on mac
    "ctrl_r": 20,
    "media_play_pause": 21,
    "media_volume_down": 22,
    "media_volume_mute": 23,
    "media_volume_up": 24,
    "shift": 25,  # left shift on mac
    "shift_l": 26,  # not recognized on mac
    "shift_r": 27,
}
```

Important: There are some platform limitations on using some of the keyboard keys with pynput which is used for monitoring the keyboard.

For instance, on macOS, some keyboard keys may require that you run your script with *sudo*. All alphanumeric keys and some special keys (e.g. `backspace` and `right`) require *sudo*. In the content of *default_key_to_channel_map* shown previously, I commented those keyboard keys that need *sudo* on macOS. The others don't need *sudo* on macOS such as `cmd_r` and `shift`.

For more information about those platform limitations, see pynput documentation.

Warning: If you want to be able to run your python script with *sudo* in order to use some keys that require it, you might need to edit **/etc/sudoers** to add your `PYTHONPATH` if your script makes use of your `PYTHONPATH` as configured in your *~/.bashrc* file. However, I don't recommend editing **/etc/sudoers** since you might break your *sudo* command (e.g. `sudo:  /etc/sudoers is owned by uid 501, should be 0`).

Instead, use the keys that don't requre *sudo* such as `cmd_r` and `shift` on macOS.

Note: On macOS, if the left keys `alt_l`, `ctrl_l`, `cmd_l`, and `shift_l` are not recognized, use their generic counterparts instead: `alt`, `ctrl`, `cmd`, and `shift`.

5.4 SimulRPi.pinbdb

Module that defines a database for storing information about GPIO pins.

The database is created as a dictionary mapping channel numbers to objects representing GPIO pins.

The *PinDB* class provides an API for accessing this database with such functions as retrieving or setting pins' attributes.

class SimulRPi.pindb.**Pin**(*channel_number*, *channel_id*, *channel_type*, *channel_name=None*, *key=None*, *led_symbols=None*, *pull_up_down=None*, *initial=None*)

Bases: object

Class that represents a GPIO pin.

Parameters

- **channel_number** (*int*) – GPIO channel number based on the numbering system you have specified (*BOARD* or *BCM*).
- **channel_id** (*str*) – Unique identifier.
- **gpio_type** (*int*) – Type of a GPIO channel: e.g. 1 (*GPIO.IN*) or 0 (*GPIO.OUT*).
- **channel_name** (*str, optional*) – It will be displayed in the terminal along with the LED symbol if it is available. Otherwise, the channel_number is shown. By default, its value is None.
- **key** (*str or None, optional*) – Keyboard key associated with the GPIO channel, e.g. *cmd_r*.
- **led_symbols** (*dict, optional*) – It should only be defined for output channels. It is a dictionary defining the symbols to be used when the LED is turned ON and OFF. If not found for an ouput channel, then the default LED symbols will be used as specified in *SimulRPi.manager.Manager*.

  Example:

```
{
    "ON": "",
    "OFF": " "
}
```

- **pull_up_down** (*int or None, optional*) – Initial value of an input channel, e.g. *GPIO.PUP_UP*. Default value is None.
- **initial** (*int or None, optional*) – Initial value of an output channel, e.g. *GPIO.HIGH*. Default value is None.

Variables **state** (*int*) – State of the GPIO channel: 1 (*HIGH*) or 0 (*LOW*).

class SimulRPi.pindb.**PinDB**

Bases: object

Class for storing and modifying *Pin*s.

Each instance of *Pin* is saved in a dictionary that maps its channel number to the *Pin* object.

Variables **output_pins** (*list*) – List containing *Pin* objects that are **output** channels.

Note: The dictionary (a "database" of *Pin*s) must be accessed through the different methods available in *PinDB*, e.g. *get_pin_from_channel()*.

---

create_pin(*channel_number*, *channel_id*, *channel_type*, ***kwargs*)
Create an instance of *Pin* and save it in a dictionary.

Based on the given arguments, an instance of *Pin* is created and added to a dictionary that acts like a database of pins with the key being the pin's channel number and the value is an instance of *Pin*.

Parameters

- **channel_number** (*int*) – GPIO channel number based on the numbering system you have specified (*BOARD* or *BCM*).
- **channel_id** (*str*) – Unique identifier.
- **channel_type** (*int*) – Type of a GPIO channel: e.g. 1 (*GPIO.IN*) or 0 (*GPIO.OUT*).
- **kwargs** (*dict, optional*) – These are the (optional) keyword arguments for `Pin.__init__()`. See *Pin* for a list of its parameters which can be included in `kwargs`.

Raises **KeyError** – Raised if two channels are using the same channel number.

get_pin_from_channel(*channel_number*)
Get a *Pin* from a given channel.

Parameters **channel_number** (*int*) – GPIO channel number associated with the *Pin* to be retrieved.

Returns **Pin** – If no *Pin* could be retrieved based on the given channel, `None` is returned. Otherwise, a *Pin* object is returned.

Return type *Pin* or `None`

get_pin_from_key(*key*)
Get a *Pin* from a given pressed/released key.

Parameters **key** (*str*) – The pressed/released key that is associated with the *Pin* to be retrieved.

Returns **Pin** – If no *Pin* could be retrieved based on the given key, `None` is returned. Otherwise, a *Pin* object is returned.

Return type *Pin* or `None`

get_pin_state(*channel_number*)
Get a *Pin*'s state from a given channel.

The state associated with a *Pin* can either be 1 (*HIGH*) or 0 (*LOW*).

Parameters **channel_number** (*int*) – GPIO channel number associated with the *Pin* whose state is to be returned.

Returns **state** – If no *Pin* could be retrieved based on the given channel number, then `None` is returned. Otherwise, the *Pin*'s state is returned: 1 (*HIGH*) or 0 (*LOW*).

Return type `int` or `None`

set_pin_id_from_channel(*channel_number*, *channel_id*)
Set a *Pin*'s channel id from a given channel number.

A *Pin* is retrieved based on a given channel, then its `channel_id` is set.

Parameters

- **channel_number** (*int*) – GPIO channel number associated with the *Pin* whose channel id will be set.
- **channel_id** (*str*) – The new channel id that a *Pin* will be updated with.

Returns **retval** – Returns *True* if the `Pin` was successfully set with *channel_id*. Otherwise, it returns *False*.

Return type bool

set_pin_key_from_channel(*channel_number*, *key*)
Set a `Pin`'s key from a given channel.

A `Pin` is retrieved based on a given channel, then its `key` is set.

Parameters

- **channel_number** (`int`) – GPIO channel number associated with the `Pin` whose key will be set.
- **key** (`str`) – The new keyboard key that a `Pin` will be updated with.

Returns **retval** – Returns *True* if the `Pin` was successfully set with *key*. Otherwise, it returns *False*.

Return type bool

set_pin_name_from_channel(*channel_number*, *channel_name*)
Set a `Pin`'s channel name from a given channel number.

A `Pin` is retrieved based on a given channel, then its `channel_name` is set.

Parameters

- **channel_number** (`int`) – GPIO channel number associated with the `Pin` whose channel name will be set.
- **channel_name** (`str`) – The new channel name that a `Pin` will be updated with.

Returns **retval** – Returns *True* if the `Pin` was successfully set with *channel_name*. Otherwise, it returns *False*.

Return type bool

set_pin_state_from_channel(*channel_number*, *state*)
Set a `Pin`'s state from a given channel.

A `Pin` is retrieved based on a given channel, then its `state` is set.

Parameters

- **channel_number** (`int`) – GPIO channel number associated with the `Pin` whose state will be set.
- **state** (`int`) – State the GPIO channel should take: 1 (*HIGH*) or 0 (*LOW*).

Returns **retval** – Returns *True* if the `Pin` was successfully set with *state*. Otherwise, it returns *False*.

Return type bool

set_pin_state_from_key(*key*, *state*)
Set a `Pin`'s state from a given key.

A `Pin` is retrieved based on a given key, then its `state` is set.

Parameters

- **key** (`str`) – The keyboard key associated with the `Pin` whose state will be set.
- **state** (`int`) – State the GPIO channel should take: 1 (*HIGH*) or 0 (*LOW*).

Returns **retval** – Returns *True* if the *Pin* was successfully set with *state*. Otherwise, it returns *False*.

Return type bool

set_pin_symbols_from_channel(*channel_number*, *led_symbols*)
Set a *Pin*'s led symbols from a given channel.

A *Pin* is retrieved based on a given key, then its `led_symbols` is set.

Parameters

- **channel_number** (*int*) – GPIO channel number associated with the *Pin* whose state will be set.
- **led_symbols** (*dict*) – It is a dictionary defining the symbols to be used when the LED is turned ON and OFF. See *Pin* for more info about this attribute.

Returns **retval** – Returns *True* if the *Pin* was successfully set with *led_symbols*. Otherwise, it returns *False*.

Return type bool

5.5 SimulRPi.run_examples

Script for executing code examples on a Raspberry Pi or computer (simulation).

This script allows you to run different code examples on your Raspberry Pi (RPi) or computer in which case it will make use of the SimulRPi library which partly fakes RPi.GPIO.

The code examples test different parts of the `SimulRPi` library in order to show what it is capable of simulating from I/O devices connected to an RPi:

- Turn on/off LEDs: blink LED symbols in the terminal
- Detect pressed button: monitor keyboard with pynput

5.5.1 Usage

Once the **SimulRPi** package is installed, you should have access to the *run_examples* script:

```
$ run_examples -h

run_examples [-h] [-v] -e EXAMPLE_NUMBER [-m {BOARD,BCM}] [-s]
             [-l [LED_CHANNEL [LED_CHANNEL ...]]]
             [-b BUTTON_CHANNEL] [-k KEY_NAME]
             [-t TOTAL_TIME_BLINKING] [--on TIME_LED_ON]
             [--off TIME_LED_OFF] [-a]
```

Run the code for example 1 on the **RPi** with default values for the options `-l` (channel 10) and `--on` (1 second):

```
$ run_examples -e 1
```

Run the code for example 1 on your **computer** using the simulation module *SimulRPi.GPIO*:

```
$ run_examples -s -e 1
```

SimulRPi.run_examples.**ex1_turn_on_led**(*channel*, *time_led_on=3*)

Example 1: Turn ON a LED for some specified time.

A LED will be turned on for `time_led_on` seconds.

Parameters

- **channel** (*int*) – Output channel number based on the numbering system you have specified (*BOARD* or *BCM*).
- **time_led_on** (*float, optional*) – Time in seconds the LED will stay turned ON. The default value is 3 seconds.

SimulRPi.run_examples.**ex2_turn_on_many_leds**(*channels*, *time_leds_on=3*)

Example 2: Turn ON multiple LEDs for some specified time.

All LEDs will be turned on for `time_leds_on` seconds.

Parameters

- **channels** (*list*) – List of output channel numbers based on the numbering system you have specified (*BOARD* or *BCM*).
- **time_leds_on** (*float, optional*) – Time in seconds the LEDs will stay turned ON. The default value is 3 seconds.

SimulRPi.run_examples.**ex3_detect_button**(*channel*)

Example 3: Detect if a button is pressed.

The function waits for the button to be pressed associated with the given `channel`. As soon as the button is pressed, a message is printed and the function exits.

Parameters **channel** (*int*) – Input channel number based on the numbering system you have specified (*BOARD* or *BCM*).

Note: If the simulation mode is enabled (`-s`), the specified keyboard key will be detected if pressed. The keyboard key can be specified through the command line option `-b` (button channel) or `-k` (the key name, e.g. '*ctrl*'). See *script's usage*.

SimulRPi.run_examples.**ex4_blink_led**(*channel*, *total_time_blinking=4*, *time_led_on=0.5*, *time_led_off=0.5*)

Example 4: Blink a LED for some specified time.

The led will blink for a total of `total_time_blinking` seconds. The LED will stay turned on for `time_led_on` seconds before turning off for `time_led_off` seconds, and so on until `total_time_blinking` seconds elapse.

Press `ctrl` + `c` to stop the blinking completely and exit from the function.

Parameters

- **channel** (*int*) – Output channel number based on the numbering system you have specified (*BOARD* or *BCM*).
- **total_time_blinking** (*float, optional*) – Total time in seconds the LED will be blinking. The default value is 4 seconds.
- **time_led_on** (*float, optional*) – Time in seconds the LED will stay turned ON at a time. The default value is 0.5 second.
- **time_led_off** (*float, optional*) – Time in seconds the LED will stay turned OFF at a time. The default value is 0.5 second.

SimulRPi.run_examples.**ex5_blink_led_if_button**(*led_channel*, *button_channel*, *total_time_blinking=4*, *time_led_on=0.5*, *time_led_off=0.5*)

Example 5: If a button is pressed, blink a LED for some specified time.

As soon as the button from the given `button_channel` is pressed, the LED will blink for a total of `total_time_blinking` seconds.

The LED will stay turned on for `time_led_on` seconds before turning off for `time_led_off` seconds, and so on until `total_time_blinking` seconds elapse.

Press `ctrl` + `c` to stop the blinking completely and exit from the function.

Parameters

- **led_channel** (*int*) – Output channel number based on the numbering system you have specified (*BOARD* or *BCM*).
- **button_channel** (*int*) – Input channel number based on the numbering system you have specified (*BOARD* or *BCM*).
- **total_time_blinking** (*float, optional*) – Total time in seconds the LED will be blinking. The default value is 4 seconds.
- **time_led_on** (*float, optional*) – Time in seconds the LED will stay turned ON at a time. The default value is 0.5 second.
- **time_led_off** (*float, optional*) – Time in seconds the LED will stay turned OFF at a time. The default value is 0.5 second.

Note: If the simulation mode is enabled (`-s`), the specified keyboard key will be detected if pressed. The keyboard key can be specified through the command line option `-b` (button channel) or `-k` (the key name, e.g. 'ctrl'). See *script's usage*.

SimulRPi.run_examples.**main**()

Main entry-point to the script.

According to the user's choice of action, the script might run one of the specified code examples.

If the simulation flag (*-s*) is used, then the SimulRPi.GPIO module will be used which partly fakes RPi.GPIO.

Notes

Only one action at a time can be performed.

SimulRPi.run_examples.**setup_argparser**()

Setup the argument parser for the command-line script.

The script allows you to run a code example on your RPi or on your computer. In the latter case, it will make use of the SimulRPi.GPIO module which partly fakes RPi.GPIO.

Returns **args** – Simple class used by default by `parse_args()` to create an object holding attributes and return it[1].

Return type argparse.Namespace

[1] argparse.Namespace.

References

5.6 SimulRPi.utils

Collection of utility functions used for the SimulRPi library.

SimulRPi.utils.**blink_led**(*channel*, *time_led_on*, *time_led_off*)
Blink LEDs from the given channels.

LEDs on the given `channel` will be turned ON and OFF for `time_led_on` seconds and `time_led_off` seconds, respectively.

Parameters

- **channel** (*int or list or tuple*) – Channel numbers associated with the LEDs which will blink.
- **time_led_on** (*float*) – Time in seconds the LEDs will stay turned ON at a time.
- **time_led_off** (*float*) – Time in seconds the LEDs will stay turned OFF at a time.

SimulRPi.utils.**turn_off_led**(*channel*)
Turn off LEDs from the given channels.

Parameters **channel** (*int or list or tuple*) – Channel numbers associated with LEDs which will be turned off.

SimulRPi.utils.**turn_on_led**(*channel*)
Turn on LEDs from the given channels.

Parameters **channel** (*int or list or tuple*) – Channel numbers associated with LEDs which will be turned on.

CHAPTER

SIX

CHANGELOG

6.1 Version 0.1.0a1

September 20, 2020

- Remove *Work-In-Progress* from documentation
- Version 0.1.0a0.post1 was yanked for a clearer version number

6.2 Version 0.1.0a0

September 15, 2020

- The default LED symbols are now big non-ASCII signs:

```
: LED turned ON
: LED turned OFF
```

  NOTE: the default symbols used by all GPIO channels can be modified with *SimulRPi.GPIO.setdefaultsymbols()*

- LED symbols for each channel can be modified with *SimulRPi.GPIO.setsymbols()*
- Channel names can now be displayed instead of channel numbers in the terminal:

```
  [LED 1]          [LED 2]          [LED 3]          [lightsaber]
```

- New modules: *SimulRPi.manager* and *SimulRPi.pindb*
  - *Manager* is now in its own module: *SimulRPi.manager*
  - *Pin* and *PinDB* are now in their own module: *SimulRPi.pindb*

  NOTE: these classes used to be in *SimulRPi.GPIO*

- New attributes in *SimulRPi.pindb.Pin* and *SimulRPi.manager.Manager*:
  - `Pin.channel_id`: unique identifier
  - `Pin.channel_name`: displayed in the terminal along each LED symbol
  - `Pin.channel_number`: used to be called `channel`
  - `Pin.channel_type`: used to be called `gpio_function` and refers to the type of GPIO channel, e.g. 1 (*GPIO.IN*) or 0 (*GPIO.OUT*).
  - `Pin.led_symbols`: each pin (aka channel) is represented by LED symbols if it is an output channel
  - `Manager.default_led_symbols`: defines the *default LED symbols* used to represent each GPIO channel in the terminal
- New functions in *SimulRPi.GPIO*:
  - *setchannelnames()*: sets channels names for multiple channels
  - *setchannels()*: sets the attributes (e.g. `channel_name` and `led_symbols`) for multiple channels
  - *setdefaultsymbols()*: changes the default LED symbols used by all output channels
  - *setsymbols()*: sets the LED symbols for multiple channels
  - *wait()*: waits for the threads to do their tasks and raises an exception if there was an error in a thread's target function. Hence, the main program can catch these thread exceptions.
- *SimulRPi.GPIO.output()* accepts *channel* and *state* as `int`, `list` or `tuple`
- *SimulRPi.GPIO.setup()* accepts *channel* as `int`, `list` or `tuple`
- The displaying thread in *SimulRPi.manager* is now an instance of *DisplayExceptionThread*. Thus, if there is an exception raised in *display_leds()*, it is now possible to catch it in the main program
- The keyboard listener thread in *SimulRPi.manager* is now an instance of `KeyboardExceptionThread` (a subclass of `pynput.keyboard.Listener`). Thus, if there is an exception raised in *on_press()* or *on_release()*, it is now possible to catch it in the main program
- *SimulRPi.GPIO.input()* and *SimulRPi.GPIO.output()* now raise an exception caught by the listening and displaying threads, respectively.
- If two channels use the same channel numbers, an exception is now raised.
- *SimulRPi.run_examples*:
  - accepts the new option `-a` which will make use of ASCII-based LED symbols in case that you are having problems displaying the *default LED symbols* which use special characters (based on the **UTF-8** encoding). See Display problems.
  - all simulation-based examples involving "LEDs" and pressing keyboard keys worked on the RPi OS (Debian-based)

See also:

The SimulRPi API reference.

6.3 Version 0.0.1a0

August 14, 2020

- In `SimulRPi.GPIO`, the package `pynput` is not required anymore. If it is not found, all keyboard-related functionalities from the `SimulRPi` library will be skipped. Thus, no keyboard keys will be detected if pressed or released when `pynput` is not installed.

  This was necessary because *Travis* was raising an exception when I was running a unit test: Xlib.error.DisplayNameError. It was due to `pynput` not working well in a headless setup. Thus, `pynput` is now removed from *requirements_travis.txt*.

  Eventually, I will mock `pynput` when doing unit tests on parts of the library that make use of `pynput`.

- Started writing unit tests

6.4 Version 0.0.0a0

August 9, 2020

- Initial release
- Tested code examples on different platforms and here are the results
  - On an RPi with `RPi.GPIO`: all examples involving LEDs and pressing buttons worked
  - On a computer with `SimulRPi.GPIO`
    - macOS: all examples involving "LEDs" and keyboard keys worked
    - RPi OS [Debian-based]: all examples involving "LEDs" only worked

      NOTE: I was running the script *`run_examples`* with `ssh` but `pynput` doesn't detect any pressed keyboard key even though I set my environment variable `Display`, added `PYTHONPATH` to *etc/sudoers* and ran the script with `sudo`. To be further investigated.

[*NOTE:* tested the code examples with *`run_examples`*] [*EDIT:* use *Initial release*]

CHAPTER

SEVEN

LICENSE: GPL3

```
GNU GENERAL PUBLIC LICENSE
                       Version 3, 29 June 2007

 Copyright (C) 2007 Free Software Foundation, Inc. <https://fsf.org/>
 Everyone is permitted to copy and distribute verbatim copies
 of this license document, but changing it is not allowed.

                            Preamble

  The GNU General Public License is a free, copyleft license for
software and other kinds of works.

  The licenses for most software and other practical works are designed
to take away your freedom to share and change the works.  By contrast,
the GNU General Public License is intended to guarantee your freedom to
share and change all versions of a program--to make sure it remains free
software for all its users.  We, the Free Software Foundation, use the
GNU General Public License for most of our software; it applies also to
any other work released this way by its authors.  You can apply it to
your programs, too.

  When we speak of free software, we are referring to freedom, not
price.  Our General Public Licenses are designed to make sure that you
have the freedom to distribute copies of free software (and charge for
them if you wish), that you receive source code or can get it if you
want it, that you can change the software or use pieces of it in new
free programs, and that you know you can do these things.

  To protect your rights, we need to prevent others from denying you
these rights or asking you to surrender the rights.  Therefore, you have
certain responsibilities if you distribute copies of the software, or if
you modify it: responsibilities to respect the freedom of others.

  For example, if you distribute copies of such a program, whether
gratis or for a fee, you must pass on to the recipients the same
freedoms that you received.  You must make sure that they, too, receive
or can get the source code.  And you must show them these terms so they
know their rights.

  Developers that use the GNU GPL protect your rights with two steps:
(1) assert copyright on the software, and (2) offer you this License
giving you legal permission to copy, distribute and/or modify it.

  For the developers' and authors' protection, the GPL clearly explains
```

(continues on next page)

(continued from previous page)

```
that there is no warranty for this free software.  For both users' and
authors' sake, the GPL requires that modified versions be marked as
changed, so that their problems will not be attributed erroneously to
authors of previous versions.

  Some devices are designed to deny users access to install or run
modified versions of the software inside them, although the manufacturer
can do so.  This is fundamentally incompatible with the aim of
protecting users' freedom to change the software.  The systematic
pattern of such abuse occurs in the area of products for individuals to
use, which is precisely where it is most unacceptable.  Therefore, we
have designed this version of the GPL to prohibit the practice for those
products.  If such problems arise substantially in other domains, we
stand ready to extend this provision to those domains in future versions
of the GPL, as needed to protect the freedom of users.

  Finally, every program is threatened constantly by software patents.
States should not allow patents to restrict development and use of
software on general-purpose computers, but in those that do, we wish to
avoid the special danger that patents applied to a free program could
make it effectively proprietary.  To prevent this, the GPL assures that
patents cannot be used to render the program non-free.

  The precise terms and conditions for copying, distribution and
modification follow.

                       TERMS AND CONDITIONS

  0. Definitions.

  "This License" refers to version 3 of the GNU General Public License.

  "Copyright" also means copyright-like laws that apply to other kinds of
works, such as semiconductor masks.

  "The Program" refers to any copyrightable work licensed under this
License.  Each licensee is addressed as "you".  "Licensees" and
"recipients" may be individuals or organizations.

  To "modify" a work means to copy from or adapt all or part of the work
in a fashion requiring copyright permission, other than the making of an
exact copy.  The resulting work is called a "modified version" of the
earlier work or a work "based on" the earlier work.

  A "covered work" means either the unmodified Program or a work based
on the Program.

  To "propagate" a work means to do anything with it that, without
permission, would make you directly or secondarily liable for
infringement under applicable copyright law, except executing it on a
computer or modifying a private copy.  Propagation includes copying,
distribution (with or without modification), making available to the
public, and in some countries other activities as well.

  To "convey" a work means any kind of propagation that enables other
parties to make or receive copies.  Mere interaction with a user through
a computer network, with no transfer of a copy, is not conveying.
```

(continues on next page)

(continued from previous page)

```
  An interactive user interface displays "Appropriate Legal Notices"
to the extent that it includes a convenient and prominently visible
feature that (1) displays an appropriate copyright notice, and (2)
tells the user that there is no warranty for the work (except to the
extent that warranties are provided), that licensees may convey the
work under this License, and how to view a copy of this License.  If
the interface presents a list of user commands or options, such as a
menu, a prominent item in the list meets this criterion.

  1. Source Code.

  The "source code" for a work means the preferred form of the work
for making modifications to it.  "Object code" means any non-source
form of a work.

  A "Standard Interface" means an interface that either is an official
standard defined by a recognized standards body, or, in the case of
interfaces specified for a particular programming language, one that
is widely used among developers working in that language.

  The "System Libraries" of an executable work include anything, other
than the work as a whole, that (a) is included in the normal form of
packaging a Major Component, but which is not part of that Major
Component, and (b) serves only to enable use of the work with that
Major Component, or to implement a Standard Interface for which an
implementation is available to the public in source code form.  A
"Major Component", in this context, means a major essential component
(kernel, window system, and so on) of the specific operating system
(if any) on which the executable work runs, or a compiler used to
produce the work, or an object code interpreter used to run it.

  The "Corresponding Source" for a work in object code form means all
the source code needed to generate, install, and (for an executable
work) run the object code and to modify the work, including scripts to
control those activities.  However, it does not include the work's
System Libraries, or general-purpose tools or generally available free
programs which are used unmodified in performing those activities but
which are not part of the work.  For example, Corresponding Source
includes interface definition files associated with source files for
the work, and the source code for shared libraries and dynamically
linked subprograms that the work is specifically designed to require,
such as by intimate data communication or control flow between those
subprograms and other parts of the work.

  The Corresponding Source need not include anything that users
can regenerate automatically from other parts of the Corresponding
Source.

  The Corresponding Source for a work in source code form is that
same work.

  2. Basic Permissions.

  All rights granted under this License are granted for the term of
copyright on the Program, and are irrevocable provided the stated
conditions are met.  This License explicitly affirms your unlimited
```

(continues on next page)

(continued from previous page)

```
permission to run the unmodified Program.  The output from running a
covered work is covered by this License only if the output, given its
content, constitutes a covered work.  This License acknowledges your
rights of fair use or other equivalent, as provided by copyright law.

  You may make, run and propagate covered works that you do not
convey, without conditions so long as your license otherwise remains
in force.  You may convey covered works to others for the sole purpose
of having them make modifications exclusively for you, or provide you
with facilities for running those works, provided that you comply with
the terms of this License in conveying all material for which you do
not control copyright.  Those thus making or running the covered works
for you must do so exclusively on your behalf, under your direction
and control, on terms that prohibit them from making any copies of
your copyrighted material outside their relationship with you.

  Conveying under any other circumstances is permitted solely under
the conditions stated below.  Sublicensing is not allowed; section 10
makes it unnecessary.

  3. Protecting Users' Legal Rights From Anti-Circumvention Law.

  No covered work shall be deemed part of an effective technological
measure under any applicable law fulfilling obligations under article
11 of the WIPO copyright treaty adopted on 20 December 1996, or
similar laws prohibiting or restricting circumvention of such
measures.

  When you convey a covered work, you waive any legal power to forbid
circumvention of technological measures to the extent such circumvention
is effected by exercising rights under this License with respect to
the covered work, and you disclaim any intention to limit operation or
modification of the work as a means of enforcing, against the work's
users, your or third parties' legal rights to forbid circumvention of
technological measures.

  4. Conveying Verbatim Copies.

  You may convey verbatim copies of the Program's source code as you
receive it, in any medium, provided that you conspicuously and
appropriately publish on each copy an appropriate copyright notice;
keep intact all notices stating that this License and any
non-permissive terms added in accord with section 7 apply to the code;
keep intact all notices of the absence of any warranty; and give all
recipients a copy of this License along with the Program.

  You may charge any price or no price for each copy that you convey,
and you may offer support or warranty protection for a fee.

  5. Conveying Modified Source Versions.

  You may convey a work based on the Program, or the modifications to
produce it from the Program, in the form of source code under the
terms of section 4, provided that you also meet all of these conditions:

    a) The work must carry prominent notices stating that you modified
    it, and giving a relevant date.
```

(continues on next page)

(continued from previous page)

```
    b) The work must carry prominent notices stating that it is
    released under this License and any conditions added under section
    7.  This requirement modifies the requirement in section 4 to
    "keep intact all notices".

    c) You must license the entire work, as a whole, under this
    License to anyone who comes into possession of a copy.  This
    License will therefore apply, along with any applicable section 7
    additional terms, to the whole of the work, and all its parts,
    regardless of how they are packaged.  This License gives no
    permission to license the work in any other way, but it does not
    invalidate such permission if you have separately received it.

    d) If the work has interactive user interfaces, each must display
    Appropriate Legal Notices; however, if the Program has interactive
    interfaces that do not display Appropriate Legal Notices, your
    work need not make them do so.

  A compilation of a covered work with other separate and independent
works, which are not by their nature extensions of the covered work,
and which are not combined with it such as to form a larger program,
in or on a volume of a storage or distribution medium, is called an
"aggregate" if the compilation and its resulting copyright are not
used to limit the access or legal rights of the compilation's users
beyond what the individual works permit.  Inclusion of a covered work
in an aggregate does not cause this License to apply to the other
parts of the aggregate.

  6. Conveying Non-Source Forms.

  You may convey a covered work in object code form under the terms
of sections 4 and 5, provided that you also convey the
machine-readable Corresponding Source under the terms of this License,
in one of these ways:

    a) Convey the object code in, or embodied in, a physical product
    (including a physical distribution medium), accompanied by the
    Corresponding Source fixed on a durable physical medium
    customarily used for software interchange.

    b) Convey the object code in, or embodied in, a physical product
    (including a physical distribution medium), accompanied by a
    written offer, valid for at least three years and valid for as
    long as you offer spare parts or customer support for that product
    model, to give anyone who possesses the object code either (1) a
    copy of the Corresponding Source for all the software in the
    product that is covered by this License, on a durable physical
    medium customarily used for software interchange, for a price no
    more than your reasonable cost of physically performing this
    conveying of source, or (2) access to copy the
    Corresponding Source from a network server at no charge.

    c) Convey individual copies of the object code with a copy of the
    written offer to provide the Corresponding Source.  This
    alternative is allowed only occasionally and noncommercially, and
    only if you received the object code with such an offer, in accord
```

(continues on next page)

(continued from previous page)

```
    with subsection 6b.

    d) Convey the object code by offering access from a designated
    place (gratis or for a charge), and offer equivalent access to the
    Corresponding Source in the same way through the same place at no
    further charge.  You need not require recipients to copy the
    Corresponding Source along with the object code.  If the place to
    copy the object code is a network server, the Corresponding Source
    may be on a different server (operated by you or a third party)
    that supports equivalent copying facilities, provided you maintain
    clear directions next to the object code saying where to find the
    Corresponding Source.  Regardless of what server hosts the
    Corresponding Source, you remain obligated to ensure that it is
    available for as long as needed to satisfy these requirements.

    e) Convey the object code using peer-to-peer transmission, provided
    you inform other peers where the object code and Corresponding
    Source of the work are being offered to the general public at no
    charge under subsection 6d.

  A separable portion of the object code, whose source code is excluded
from the Corresponding Source as a System Library, need not be
included in conveying the object code work.

  A "User Product" is either (1) a "consumer product", which means any
tangible personal property which is normally used for personal, family,
or household purposes, or (2) anything designed or sold for incorporation
into a dwelling.  In determining whether a product is a consumer product,
doubtful cases shall be resolved in favor of coverage.  For a particular
product received by a particular user, "normally used" refers to a
typical or common use of that class of product, regardless of the status
of the particular user or of the way in which the particular user
actually uses, or expects or is expected to use, the product.  A product
is a consumer product regardless of whether the product has substantial
commercial, industrial or non-consumer uses, unless such uses represent
the only significant mode of use of the product.

  "Installation Information" for a User Product means any methods,
procedures, authorization keys, or other information required to install
and execute modified versions of a covered work in that User Product from
a modified version of its Corresponding Source.  The information must
suffice to ensure that the continued functioning of the modified object
code is in no case prevented or interfered with solely because
modification has been made.

  If you convey an object code work under this section in, or with, or
specifically for use in, a User Product, and the conveying occurs as
part of a transaction in which the right of possession and use of the
User Product is transferred to the recipient in perpetuity or for a
fixed term (regardless of how the transaction is characterized), the
Corresponding Source conveyed under this section must be accompanied
by the Installation Information.  But this requirement does not apply
if neither you nor any third party retains the ability to install
modified object code on the User Product (for example, the work has
been installed in ROM).

  The requirement to provide Installation Information does not include a
```

(continues on next page)

(continued from previous page)

```
requirement to continue to provide support service, warranty, or updates
for a work that has been modified or installed by the recipient, or for
the User Product in which it has been modified or installed.  Access to a
network may be denied when the modification itself materially and
adversely affects the operation of the network or violates the rules and
protocols for communication across the network.

  Corresponding Source conveyed, and Installation Information provided,
in accord with this section must be in a format that is publicly
documented (and with an implementation available to the public in
source code form), and must require no special password or key for
unpacking, reading or copying.

  7. Additional Terms.

  "Additional permissions" are terms that supplement the terms of this
License by making exceptions from one or more of its conditions.
Additional permissions that are applicable to the entire Program shall
be treated as though they were included in this License, to the extent
that they are valid under applicable law.  If additional permissions
apply only to part of the Program, that part may be used separately
under those permissions, but the entire Program remains governed by
this License without regard to the additional permissions.

  When you convey a copy of a covered work, you may at your option
remove any additional permissions from that copy, or from any part of
it.  (Additional permissions may be written to require their own
removal in certain cases when you modify the work.)  You may place
additional permissions on material, added by you to a covered work,
for which you have or can give appropriate copyright permission.

  Notwithstanding any other provision of this License, for material you
add to a covered work, you may (if authorized by the copyright holders of
that material) supplement the terms of this License with terms:

    a) Disclaiming warranty or limiting liability differently from the
    terms of sections 15 and 16 of this License; or

    b) Requiring preservation of specified reasonable legal notices or
    author attributions in that material or in the Appropriate Legal
    Notices displayed by works containing it; or

    c) Prohibiting misrepresentation of the origin of that material, or
    requiring that modified versions of such material be marked in
    reasonable ways as different from the original version; or

    d) Limiting the use for publicity purposes of names of licensors or
    authors of the material; or

    e) Declining to grant rights under trademark law for use of some
    trade names, trademarks, or service marks; or

    f) Requiring indemnification of licensors and authors of that
    material by anyone who conveys the material (or modified versions of
    it) with contractual assumptions of liability to the recipient, for
    any liability that these contractual assumptions directly impose on
    those licensors and authors.
```

(continues on next page)

(continued from previous page)

```
  All other non-permissive additional terms are considered "further
restrictions" within the meaning of section 10.  If the Program as you
received it, or any part of it, contains a notice stating that it is
governed by this License along with a term that is a further
restriction, you may remove that term.  If a license document contains
a further restriction but permits relicensing or conveying under this
License, you may add to a covered work material governed by the terms
of that license document, provided that the further restriction does
not survive such relicensing or conveying.

  If you add terms to a covered work in accord with this section, you
must place, in the relevant source files, a statement of the
additional terms that apply to those files, or a notice indicating
where to find the applicable terms.

  Additional terms, permissive or non-permissive, may be stated in the
form of a separately written license, or stated as exceptions;
the above requirements apply either way.

  8. Termination.

  You may not propagate or modify a covered work except as expressly
provided under this License.  Any attempt otherwise to propagate or
modify it is void, and will automatically terminate your rights under
this License (including any patent licenses granted under the third
paragraph of section 11).

  However, if you cease all violation of this License, then your
license from a particular copyright holder is reinstated (a)
provisionally, unless and until the copyright holder explicitly and
finally terminates your license, and (b) permanently, if the copyright
holder fails to notify you of the violation by some reasonable means
prior to 60 days after the cessation.

  Moreover, your license from a particular copyright holder is
reinstated permanently if the copyright holder notifies you of the
violation by some reasonable means, this is the first time you have
received notice of violation of this License (for any work) from that
copyright holder, and you cure the violation prior to 30 days after
your receipt of the notice.

  Termination of your rights under this section does not terminate the
licenses of parties who have received copies or rights from you under
this License.  If your rights have been terminated and not permanently
reinstated, you do not qualify to receive new licenses for the same
material under section 10.

  9. Acceptance Not Required for Having Copies.

  You are not required to accept this License in order to receive or
run a copy of the Program.  Ancillary propagation of a covered work
occurring solely as a consequence of using peer-to-peer transmission
to receive a copy likewise does not require acceptance.  However,
nothing other than this License grants you permission to propagate or
modify any covered work.  These actions infringe copyright if you do
not accept this License.  Therefore, by modifying or propagating a
```

(continues on next page)

(continued from previous page)

```
covered work, you indicate your acceptance of this License to do so.

  10. Automatic Licensing of Downstream Recipients.

  Each time you convey a covered work, the recipient automatically
receives a license from the original licensors, to run, modify and
propagate that work, subject to this License.  You are not responsible
for enforcing compliance by third parties with this License.

  An "entity transaction" is a transaction transferring control of an
organization, or substantially all assets of one, or subdividing an
organization, or merging organizations.  If propagation of a covered
work results from an entity transaction, each party to that
transaction who receives a copy of the work also receives whatever
licenses to the work the party's predecessor in interest had or could
give under the previous paragraph, plus a right to possession of the
Corresponding Source of the work from the predecessor in interest, if
the predecessor has it or can get it with reasonable efforts.

  You may not impose any further restrictions on the exercise of the
rights granted or affirmed under this License.  For example, you may
not impose a license fee, royalty, or other charge for exercise of
rights granted under this License, and you may not initiate litigation
(including a cross-claim or counterclaim in a lawsuit) alleging that
any patent claim is infringed by making, using, selling, offering for
sale, or importing the Program or any portion of it.

  11. Patents.

  A "contributor" is a copyright holder who authorizes use under this
License of the Program or a work on which the Program is based.  The
work thus licensed is called the contributor's "contributor version".

  A contributor's "essential patent claims" are all patent claims
owned or controlled by the contributor, whether already acquired or
hereafter acquired, that would be infringed by some manner, permitted
by this License, of making, using, or selling its contributor version,
but do not include claims that would be infringed only as a
consequence of further modification of the contributor version.  For
purposes of this definition, "control" includes the right to grant
patent sublicenses in a manner consistent with the requirements of
this License.

  Each contributor grants you a non-exclusive, worldwide, royalty-free
patent license under the contributor's essential patent claims, to
make, use, sell, offer for sale, import and otherwise run, modify and
propagate the contents of its contributor version.

  In the following three paragraphs, a "patent license" is any express
agreement or commitment, however denominated, not to enforce a patent
(such as an express permission to practice a patent or covenant not to
sue for patent infringement).  To "grant" such a patent license to a
party means to make such an agreement or commitment not to enforce a
patent against the party.

  If you convey a covered work, knowingly relying on a patent license,
and the Corresponding Source of the work is not available for anyone
```

(continues on next page)

(continued from previous page)

```
to copy, free of charge and under the terms of this License, through a
publicly available network server or other readily accessible means,
then you must either (1) cause the Corresponding Source to be so
available, or (2) arrange to deprive yourself of the benefit of the
patent license for this particular work, or (3) arrange, in a manner
consistent with the requirements of this License, to extend the patent
license to downstream recipients.  "Knowingly relying" means you have
actual knowledge that, but for the patent license, your conveying the
covered work in a country, or your recipient's use of the covered work
in a country, would infringe one or more identifiable patents in that
country that you have reason to believe are valid.

  If, pursuant to or in connection with a single transaction or
arrangement, you convey, or propagate by procuring conveyance of, a
covered work, and grant a patent license to some of the parties
receiving the covered work authorizing them to use, propagate, modify
or convey a specific copy of the covered work, then the patent license
you grant is automatically extended to all recipients of the covered
work and works based on it.

  A patent license is "discriminatory" if it does not include within
the scope of its coverage, prohibits the exercise of, or is
conditioned on the non-exercise of one or more of the rights that are
specifically granted under this License.  You may not convey a covered
work if you are a party to an arrangement with a third party that is
in the business of distributing software, under which you make payment
to the third party based on the extent of your activity of conveying
the work, and under which the third party grants, to any of the
parties who would receive the covered work from you, a discriminatory
patent license (a) in connection with copies of the covered work
conveyed by you (or copies made from those copies), or (b) primarily
for and in connection with specific products or compilations that
contain the covered work, unless you entered into that arrangement,
or that patent license was granted, prior to 28 March 2007.

  Nothing in this License shall be construed as excluding or limiting
any implied license or other defenses to infringement that may
otherwise be available to you under applicable patent law.

  12. No Surrender of Others' Freedom.

  If conditions are imposed on you (whether by court order, agreement or
otherwise) that contradict the conditions of this License, they do not
excuse you from the conditions of this License.  If you cannot convey a
covered work so as to satisfy simultaneously your obligations under this
License and any other pertinent obligations, then as a consequence you may
not convey it at all.  For example, if you agree to terms that obligate you
to collect a royalty for further conveying from those to whom you convey
the Program, the only way you could satisfy both those terms and this
License would be to refrain entirely from conveying the Program.

  13. Use with the GNU Affero General Public License.

  Notwithstanding any other provision of this License, you have
permission to link or combine any covered work with a work licensed
under version 3 of the GNU Affero General Public License into a single
combined work, and to convey the resulting work.  The terms of this
```

(continues on next page)

(continued from previous page)

```
License will continue to apply to the part which is the covered work,
but the special requirements of the GNU Affero General Public License,
section 13, concerning interaction through a network will apply to the
combination as such.

  14. Revised Versions of this License.

  The Free Software Foundation may publish revised and/or new versions of
the GNU General Public License from time to time.  Such new versions will
be similar in spirit to the present version, but may differ in detail to
address new problems or concerns.

  Each version is given a distinguishing version number.  If the
Program specifies that a certain numbered version of the GNU General
Public License "or any later version" applies to it, you have the
option of following the terms and conditions either of that numbered
version or of any later version published by the Free Software
Foundation.  If the Program does not specify a version number of the
GNU General Public License, you may choose any version ever published
by the Free Software Foundation.

  If the Program specifies that a proxy can decide which future
versions of the GNU General Public License can be used, that proxy's
public statement of acceptance of a version permanently authorizes you
to choose that version for the Program.

  Later license versions may give you additional or different
permissions.  However, no additional obligations are imposed on any
author or copyright holder as a result of your choosing to follow a
later version.

  15. Disclaimer of Warranty.

  THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY
APPLICABLE LAW.  EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT
HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY
OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO,
THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
PURPOSE.  THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM
IS WITH YOU.  SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF
ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

  16. Limitation of Liability.

  IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING
WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS
THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY
GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE
USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF
DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD
PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS),
EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF
SUCH DAMAGES.

  17. Interpretation of Sections 15 and 16.

  If the disclaimer of warranty and limitation of liability provided
```

(continues on next page)

(continued from previous page)

```
above cannot be given local legal effect according to their terms,
reviewing courts shall apply local law that most closely approximates
an absolute waiver of all civil liability in connection with the
Program, unless a warranty or assumption of liability accompanies a
copy of the Program in return for a fee.

                     END OF TERMS AND CONDITIONS

            How to Apply These Terms to Your New Programs

  If you develop a new program, and you want it to be of the greatest
possible use to the public, the best way to achieve this is to make it
free software which everyone can redistribute and change under these terms.

  To do so, attach the following notices to the program.  It is safest
to attach them to the start of each source file to most effectively
state the exclusion of warranty; and each file should have at least
the "copyright" line and a pointer to where the full notice is found.

    <one line to give the program's name and a brief idea of what it does.>
    Copyright (C) <year>  <name of author>

    This program is free software: you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with this program.  If not, see <https://www.gnu.org/licenses/>.

Also add information on how to contact you by electronic and paper mail.

  If the program does terminal interaction, make it output a short
notice like this when it starts in an interactive mode:

    <program>  Copyright (C) <year>  <name of author>
    This program comes with ABSOLUTELY NO WARRANTY; for details type `show w'.
    This is free software, and you are welcome to redistribute it
    under certain conditions; type `show c' for details.

The hypothetical commands `show w' and `show c' should show the appropriate
parts of the General Public License.  Of course, your program's commands
might be different; for a GUI interface, you would use an "about box".

  You should also get your employer (if you work as a programmer) or school,
if any, to sign a "copyright disclaimer" for the program, if necessary.
For more information on this, and how to apply and follow the GNU GPL, see
<https://www.gnu.org/licenses/>.

  The GNU General Public License does not permit incorporating your program
into proprietary programs.  If your program is a subroutine library, you
may consider it more useful to permit linking proprietary applications with
```

(continues on next page)

(continued from previous page)

```
the library.  If this is what you want to do, use the GNU Lesser General
Public License instead of this License.  But first, please read
<https://www.gnu.org/licenses/why-not-lgpl.html>.
```

CHAPTER

EIGHT

INDICES AND TABLES

- genindex
- modindex

PYTHON MODULE INDEX

S

INDEX

T

U

V

W