pitop

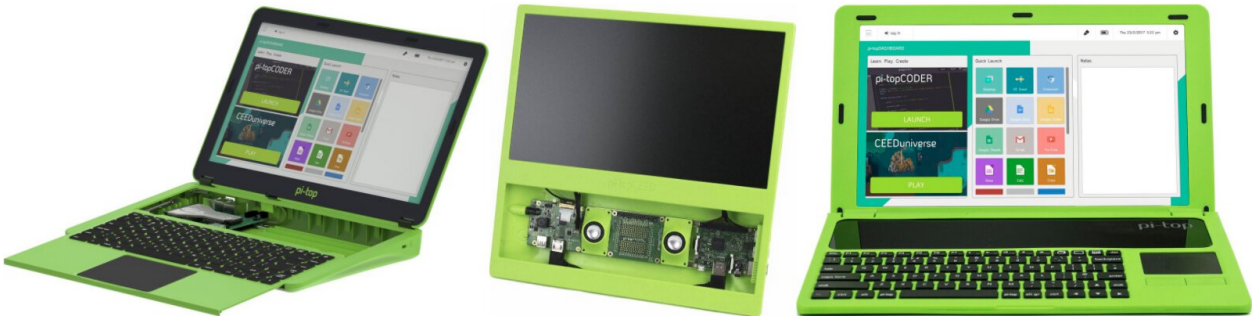
Release 0.0.1.dev1

Jul 05, 2023

Contents

A simple, modular interface for interacting with a pi-top and its related accessories and components.

Supports all pi-top devices:



Supports pi-top Maker Architecture (PMA):

Supports all pi-top peripherals:

CHAPTER 1

Status: Active Development

This SDK is currently in active development. Please be patient while we work towards v1.0.0!

1.1 Backwards Compatibility

When this library reaches v1.0.0, we will aim to maintain backwards-compatibility thereafter. Until then, every effort will be made to ensure stable support, but it cannot be guaranteed. Breaking changes will be clearly documented.

CHAPTER 2

About

This SDK aims to provide an easy-to-use framework for managing a pi-top. It includes a Python 3 package (*pitop*), with several custom modules and classes for interfacing with a range of pi-top devices and peripherals. It also contains CLI utilities, to interact with your pi-top using the terminal.

The SDK is included out-of-the-box with pi-topOS.

Ensure that you keep your system up-to-date to enjoy the latest features and bug fixes.

This library is installed as a Python 3 module called *pitop*. It includes several submodules that allow you to easily interact with most of the hardware inside a pi-top.

You can easily connect different components of the system using the modules available in the library:

```
from time import sleep
from pitop import UltrasonicSensor, Miniscreen

utrasonic = UltrasonicSensor("D1")
miniscreen = Miniscreen()

while True:
    miniscreen.display_text(utrasonic.distance)
    sleep(0.1)
```

Check out the *Overview* chapter for more information on what you can do.

The SDK also contains a Command Line Interface (CLI). See the *'pi-top' command* for more information.

CHAPTER 3

Table of Contents

3.1 Getting Started

3.1.1 Installing the SDK

pi-topOS

This SDK is pre-installed on pi-topOS, so you don't need to install it manually!

Using apt

The recommended way of getting the latest version is through *apt*.

Check out Using pi-top Hardware with Raspberry Pi OS in the pi-top knowledge base for how to do this.

Note: If you only want to install the SDK, then you can replace the "Install software packages" step:

```
sudo apt install -y python3-pitop
```

This will also install additional packages onto your system that the SDK requires in order to work.

Using PyPI

In general, this is not recommended.

You can also install the latest version of the SDK through PyPI in your pi-top with:

```
pip3 install pitop
```

You'll need to install one extra dependency for the SDK to work when using pip:

```
sudo apt install libatlas-base-dev -y
```

Note: This will not install the system packages required for all areas of the SDK to work. This may be useful if you wish to use a virtualenv with a different version dependency to the system.

Building from source

In general, this is not recommended.

Building from source is simple:

```
git clone https://github.com/pi-top/pi-top-Python-SDK.git
cd pi-top-Python-SDK
pip3 install -e .
```

You'll need to install one extra dependency for the SDK to work when using pip:

```
sudo apt install libatlas-base-dev -y
```

Note: This will not install the system packages required for all areas of the SDK to work. This may be useful if you wish to use a virtualenv with a different version dependency to the system.

3.1.2 Checking that the SDK is installed and working

Try and run the following:

```
pi-top devices hub
```

If this works, then you should be good to go! Go and check out the Examples section!

3.1.3 What next!?

Now that you're ready to go, check out the *Overview* chapter for more information on what you can do.

3.2 Overview

This API provides features that are selectively available, depending on the pi-top device that you are using. To find out what is available for your pi-top device, see the relevant section below.

Choose your pi-top device to go to the relevant section:

- *pi-top [4]*
- *pi-top [3]*
- *pi-topCEED*
- *Original pi-top*

This API provides some convenience classes for *common System Peripheral Devices*, such as:

- Camera
- Keyboard

3.2.1 pi-top [4]

Interacting with onboard pi-top [4] hardware

pi-top [4] supports the following API devices/components for its onboard hardware:

- *pi-top Battery*
- *pi-top [4] Miniscreen*

pi-top [4] does not support the following API devices/components:

- *pi-top Display*

This is due to the fact that pi-top [4] has no attached display, and the pi-top [4] official display's brightness is handled in hardware with physical brightness buttons, and the backlight is handled by DPMS (the operating system's internal screen blanking functionality).

Physical computing with pi-top [4]

pi-top [4] supports the following API devices/components for physical computing:

- *pi-topPULSE*
- *pi-top Maker Architecture (PMA) Components*

The pi-topPULSE can be used as a Raspberry Pi HAT with a pi-top [4]. The USB camera library can be used with any USB camera, and - whilst technically can be used with any Raspberry Pi/pi-top, was designed with the pi-top [4] and PMA in mind.

pi-top [4] does not support the following API devices/components:

- *pi-topPROTO+*

This is due to the fact that pi-topPROTO+ makes use of the legacy 'modular rail', which has no way of connecting to a pi-top [4].

Check out the *key concepts for pi-top Maker Architecture* for more information.

3.2.2 pi-top laptops

Interacting with onboard pi-top laptop hardware

pi-top laptops (Original pi-top and pi-top [3]) support the following API devices/components for their onboard hardware:

- *pi-top Battery*
- *pi-top Display*

pi-top laptops does not support the following API devices/components:

- *pi-top [4] Miniscreen*

This is due to the fact that pi-top laptops do not include the pi-top [4]'s miniscreen.

Using peripherals with a pi-top laptop

pi-top laptops (Original pi-top and pi-top [3]) support the following API devices/components for use with peripherals:

- *pi-topPROTO+*
- *pi-topPULSE*

Note that the USB camera library works with any pi-top with a USB camera connected. This was designed for pi-top [4] usage, but due to its general purpose functionality, it can technically be used if desired.

pi-topSPEAKER support is provided automagically by pi-topd, and so there is no exposed API for this.

pi-top laptops does not support the following API devices/components:

- *pi-top Maker Architecture (PMA) Components*

This is due to the fact that PMA is only available for pi-top [4].

3.2.3 pi-topCEED

Interacting with onboard pi-topCEED hardware

pi-top laptops (Original pi-top and pi-top [3]) support the following API devices/components for their onboard hardware:

- *pi-top Display*

pi-top laptops does not support the following API devices/components:

- *pi-top Battery*
- *pi-top [4] Miniscreen*

This is due to the fact that pi-topCEED does not include a battery or the pi-top [4]'s miniscreen.

Using peripherals with a pi-topCEED

pi-topCEED supports the following API devices/components for use with peripherals:

- *pi-topPROTO+*
- *pi-topPULSE*

Note that the USB camera library works with any pi-top with a USB camera connected. This was designed for pi-top [4] usage, but due to its general purpose functionality, it can technically be used if desired.

pi-topSPEAKER support is provided automagically by pi-topd, and so there is no exposed API for this.

pi-topCEED does not support the following API devices/components:

- *pi-top Maker Architecture (PMA) Components*

This is due to the fact that PMA is only available for pi-top [4].

3.3 Key Concepts

3.3.1 pi-top Maker Architecture

This section aims to clarify the various components of PMA, and the terminology that is required to get the most out of it.

Inputs and Outputs

A component can be classified as an Input or Output, according to how it behaves.

An Input component generates electric signals that can be interpreted as information when read. For example, when a button is clicked, the electric signal it produces lets you know that it's state changed.

An Output component receives electric signals and performs an action based on them. For example, a buzzer; when no signal is applied it will be silent; however when an electric signal is applied, it will generate sound.

Digital and Analog

Components can also be classified according to the type of electric signals they use.

Digital components only use digital electric signals; digital signals are discrete and carry information in binary form, most of the times consisting in different voltage values. This change in voltage can be read by a Raspberry Pi directly.

Analog components use analog electric signals; analog signals are continuous and can have infinite values in a determined range. Raspberry Pi can't directly read these signals since it's a digital component. To be able to manage analog signals, the Foundation and Expansion plates include an Analog to Digital Converter (ADC). This device converts the analog signal from the component into a digital signal that can be interpreted by the Raspberry Pi.

Ports and Pins

The pi-top Maker Architecture (PMA) connector on the pi-top [4] makes available all GPIO from the Raspberry Pi to the Foundation and Expansion Plates.

This means that the ports located in these plates are mapped to the GPIO header on the Raspberry Pi, providing easy and standard access through Grove connectors to these pins.

Foundation and Expansion Plates have multiple connectors that can be used to interface with different kind components.

Digital Ports

Used to communicate with digital devices.

These ports are labeled from *D0* to *D7*.

Analog Ports

Used to communicate with analog devices.

These ports are labeled from *A0* to *A3*.

Motor Ports

Communicates a motor encoder component with the motor controller, located inside the Expansion Plate.

These ports are labeled from *M0* to *M3*

ServoMotor Ports

Communicates a servo motor component with the servomotor controller, located inside the Expansion Plate.

These ports are labeled from *S0* to *S3*.

I2C Ports

Used to communicate with generic I2C devices.

These ports are labeled as *I2C*.

Identifying PMA port for a component

The components included in the Foundation Kit & Robotics Kit can be classified according to how they operate and communicate.

Digital component

These components should be connected to a *Digital Port* on the Foundation/Expansion Plates.

The Digital components included in the Foundation & Robotics Kits are:

- *Button*
- *Buzzer*
- *LED*

Analog component

These components should be connected to a *Analog Port* on the Foundation/Expansion Plates, labeled from *A0* to *A3*.

The Analog components included in the Foundation & Robotics Kits are:

- *LightSensor*
- *Potentiometer*
- *SoundSensor*
- *UltrasonicSensor*

Motor component

An electromechanical component that is controlled by communicating with a microprocessor located inside the Expansion Plate.

These components should be connected to a *Motor Port* or to *ServoMotor Port* on the Expansion Plate, depending on the component used.

The *Motor* component included in the Robotics Kits are:

- *MotorEncoder* (connects to a *Motor Port*)
- *ServoMotor* (connects to a *ServoMotor Port*)

More Information

For more information about pi-top Maker Architecture, check out the pi-top Knowledge Base.

3.3.2 pi-top [4] Miniscreen

The miniscreen of the pi-top [4] can be found on the front, comprised of an 128x64 pixel OLED screen and 4 programmable buttons.

The *pt-miniscreen* package (*pt-sys-oled* in earlier versions of pi-topOS), provided out-of-the-box with pi-topOS (and available for Raspberry Pi OS), provides a convenient interactive menu interface, using the pi-top [4]'s miniscreen OLED display and buttons for navigation and actions. This menu includes useful information and options about the system state and configuration.

When a user program creates an instance of the miniscreen, the system menu will clear itself and start to ignore button press events until the user program exits. This is true, regardless of whether or not the OLED display or the buttons were intended to be used.

Warning: When you write a program that interacts with the pi-top [4] miniscreen, the miniscreen display will clear itself, ready to be controlled by user code.

The system menu cannot be accessed until the program exits, at which point the system menu is automatically restored.

Note: For convenience, it is recommended that you provide yourself with an easy method of being able to exit your program. It is recommended that you configure an input (such as the miniscreen's 'cancel' button) to trigger an exit. This is particularly helpful if you wish to start/stop your project headlessly (that is, without requiring a display or keyboard/mouse).

Here is one way of achieving this:

```
from time import sleep

from pitop import Pitop

pitop = Pitop()
miniscreen = pitop.miniscreen
miniscreen.display_multiline_text("Press cancel to exit!", font_size=22)

while not miniscreen.cancel_button.is_pressed:
    sleep(0.1)

miniscreen.display_multiline_text("Bye!")
sleep(2)
```

If you wish to make use of any of the functionality in system menu, have a go at implementing it yourself in your own project!

3.4 Recipes

In addition to the examples provided for each component/device in the API reference section of this documentation, the following recipes demonstrate some of the more advanced capabilities of the pi-top Python SDK. In particular, these recipes focus on practical use-cases that make use of multiple components/devices within the pi-top Python SDK.

Be sure to check out each component/device separately for simple examples of how to use them.

3.4.1 PMA: Using a Button to Control an LED

```
from time import sleep

from pitop import LED, Button

button = Button("D1")
led = LED("D2")

# Connect button to LED
button.when_pressed = led.on
button.when_released = led.off
```

(continues on next page)

(continued from previous page)

```
# Wait for Ctrl+C to exit
try:
    while True:
        sleep(1)
except KeyboardInterrupt:
    pass
```

3.4.2 Robotics Kit: DIY Rover

```
from threading import Thread
from time import sleep

from pitop import BrakingType, EncoderMotor, ForwardDirection

# Setup the motors for the rover configuration

motor_left = EncoderMotor("M3", ForwardDirection.CLOCKWISE)
motor_right = EncoderMotor("M0", ForwardDirection.COUNTER_CLOCKWISE)

motor_left.braking_type = BrakingType.COAST
motor_right.braking_type = BrakingType.COAST


# Define some functions for easily controlling the rover


def drive(target_rpm: float):
    print("Start driving at target", target_rpm, "rpm...")
    motor_left.set_target_rpm(target_rpm)
    motor_right.set_target_rpm(target_rpm)


def stop_rover():
    print("Stopping rover...")
    motor_left.stop()
    motor_right.stop()


def turn_left(rotation_speed: float):
    print("Turning left...")
    motor_left.stop()
    motor_right.set_target_rpm(rotation_speed)


def turn_right(rotation_speed: float):
    print("Turning right...")
    motor_right.stop()
    motor_left.set_target_rpm(rotation_speed)


# Start a thread to monitor the rover


def monitor_rover():
```

(continues on next page)

(continued from previous page)

```
    while True:
        print(
            "> Rover motor RPM's (L,R):",
            round(motor_left.current_rpm, 2),
            round(motor_right.current_rpm, 2),
        )
        sleep(1)


monitor_thread = Thread(target=monitor_rover, daemon=True)
monitor_thread.start()

# Go!

rpm_speed = 100
for _ in range(4):
    drive(rpm_speed)
    sleep(5)

    turn_left(rpm_speed)
    sleep(5)

stop_rover()
```

3.4.3 Robotics Kit: Robot - Moving Randomly

```
from random import randint
from time import sleep

from pitop import Pitop
from pitop.robotics.drive_controller import DriveController

# Create a basic robot
robot = Pitop()
drive = DriveController(left_motor_port="M3", right_motor_port="M0")
robot.add_component(drive)


# Use miniscreen display
robot.miniscreen.display_multiline_text("hey there!")


def random_speed_factor():
    # 0.01 - 1, 0.01 resolution
    return randint(1, 100) / 100


def random_sleep():
    # 0.5 - 2, 0.5 resolution
    return randint(1, 4) / 2


# Move around randomly
robot.drive.forward(speed_factor=random_speed_factor())
sleep(random_sleep())
```

(continues on next page)

(continued from previous page)

```
robot.drive.left(speed_factor=random_speed_factor())
sleep(random_sleep())

robot.drive.backward(speed_factor=random_speed_factor())
sleep(random_sleep())

robot.drive.right(speed_factor=random_speed_factor())
sleep(random_sleep())
```

3.4.4 Robotics Kit: Robot - Line Detection

```
from signal import pause

from pitop import Camera, DriveController, Pitop
from pitop.processing.algorithms.line_detect import process_frame_for_line

# Assemble a robot
robot = Pitop()
robot.add_component(DriveController(left_motor_port="M3", right_motor_port="M0"))
robot.add_component(Camera())


# Set up logic based on line detection
def drive_based_on_frame(frame):
    processed_frame = process_frame_for_line(frame)

    if processed_frame.line_center is None:
        print("Line is lost!", end="\r")
        robot.drive.stop()
    else:
        print(f"Target angle: {processed_frame.angle:.2f} deg ", end="\r")
        robot.drive.forward(0.25, hold=True)
        robot.drive.target_lock_drive_angle(processed_frame.angle)
        robot.miniscreen.display_image(processed_frame.robot_view)


# On each camera frame, detect a line
robot.camera.on_frame = drive_based_on_frame


pause()
```

3.4.5 Displaying camera stream in pi-top [4]'s miniscreen

```
from pitop import Camera, Pitop

camera = Camera()
pitop = Pitop()
camera.on_frame = pitop.miniscreen.display_image
```

3.4.6 Robotics Kit: Robot - Control using Bluedot

Note: BlueDot is a Python library that allows you to control Raspberry Pi projects remotely. This example demonstrates a way to control a robot with a virtual joystick.

```
from signal import pause
from threading import Lock

from bluedot import BlueDot

from pitop import DriveController

bd = BlueDot()
bd.color = "#00B2A2"
lock = Lock()

drive = DriveController(left_motor_port="M3", right_motor_port="M0")


def move(pos):
    if lock.locked():
        return

    if any(
        [
            pos.angle > 0 and pos.angle < 20,
            pos.angle < 0 and pos.angle > -20,
        ]
    ):
        drive.forward(pos.distance, hold=True)
    elif pos.angle > 0 and 20 <= pos.angle <= 160:
        turn_radius = 0 if 70 < pos.angle < 110 else pos.distance
        speed_factor = -pos.distance if pos.angle > 110 else pos.distance
        drive.right(speed_factor, turn_radius)
    elif pos.angle < 0 and -160 <= pos.angle <= -20:
        turn_radius = 0 if -110 < pos.angle < -70 else pos.distance
        speed_factor = -pos.distance if pos.angle < -110 else pos.distance
        drive.left(speed_factor, turn_radius)
    elif any(
        [
            pos.angle > 0 and pos.angle > 160,
            pos.angle < 0 and pos.angle < -160,
        ]
    ):
        drive.backward(pos.distance, hold=True)


def stop(pos):
    lock.acquire()
    drive.stop()


def start(pos):
    if lock.locked():
        lock.release()
    move(pos)
```

(continues on next page)

(continued from previous page)

```
bd.when_pressed = start
bd.when_moved = move
bd.when_released = stop

pause()
```

3.4.7 Using the pi-topPULSE's LED matrix to show the battery level

```
from time import sleep

from pitop import Pitop
from pitop.pulse import ledmatrix


def draw_battery_outline():  # Draw the naked battery
    for y in range(0, 6):
        ledmatrix.set_pixel(1, y, 64, 64, 255)
        ledmatrix.set_pixel(5, y, 64, 64, 255)
    for x in range(2, 5):
        ledmatrix.set_pixel(x, 0, 64, 64, 255)
        ledmatrix.set_pixel(x, 6, 192, 192, 192)
    ledmatrix.show()


def update_battery_state(charging_state, capacity):
    r = 0
    g = 0
    b = 0
    if charging_state == 0:
        if capacity < 11:
            r = 255
        else:
            g = 255
    elif charging_state == 1:
        r = 255
        g = 225

    cap = int(capacity / 20) + 1
    if cap < 0:
        cap = 0
    if cap > 5:
        cap = 5

    if cap > 0:
        for y in range(1, cap + 1):
            ledmatrix.set_pixel(2, y, r, g, b)
            ledmatrix.set_pixel(3, y, r, g, b)
            ledmatrix.set_pixel(4, y, r, g, b)
    if cap == 0:
        cap = 1
    if cap < 6:
        if (capacity < 50) and (charging_state == 0):
            # blinking warning
```

(continues on next page)

(continued from previous page)

```
            for i in range(1, 3):
                for y in range(cap + 1, 6):
                    ledmatrix.set_pixel(2, y, 0, 0, 0)
                    ledmatrix.set_pixel(3, y, 0, 0, 0)
                    ledmatrix.set_pixel(4, y, 0, 0, 0)
                ledmatrix.show()
                sleep(0.4)
                for y in range(cap + 1, 6):
                    ledmatrix.set_pixel(2, y, 255, 0, 0)
                    ledmatrix.set_pixel(3, y, 255, 0, 0)
                    ledmatrix.set_pixel(4, y, 255, 0, 0)
                ledmatrix.show()
                sleep(0.4)

        else:
            for y in range(cap + 1, 6):
                ledmatrix.set_pixel(2, y, 0, 0, 0)
                ledmatrix.set_pixel(3, y, 0, 0, 0)
                ledmatrix.set_pixel(4, y, 0, 0, 0)
            ledmatrix.show()
            sleep(5)
    return 0


def main():
    ledmatrix.rotation(0)
    ledmatrix.clear()  # Clear the display
    draw_battery_outline()  # Draw the battery outline

    battery = Pitop().battery

    while True:
        try:
            charging_state, capacity, _, _ = battery.get_full_state()
            update_battery_state(charging_state, capacity)  # Fill battery with␣
↪capacity

        except Exception as e:
            print("Error getting battery info: " + str(e))


if __name__ == "__main__":
    main()
```

3.4.8 Choose a pi-top [4] miniscreen startup animation

Note: This code makes use of the GIPHY SDK. Follow the instructions here to find out how to apply for an API Key to use with this project.

Replace <*MY GIPHY KEY*> with the key provided (keep the quotes).

You can change the type of images that you get by changing *SEARCH_TERM = "Monochrome"* to whatever you want.

```
import json
from configparser import ConfigParser
from os import geteuid
from random import randint
from signal import pause
from sys import exit
from time import sleep
from urllib.parse import urlencode
from urllib.request import urlopen

from PIL import Image
from requests.models import PreparedRequest

from pitop.miniscreen import Miniscreen


def is_root():
    return geteuid() == 0


if not is_root():
    print("Admin access required - please run this script with 'sudo'.")
    exit()

# Define Giphy parameters
SEARCH_LIMIT = 10
SEARCH_TERM = "Monochrome"

CONFIG_FILE_PATH = "/etc/pt-miniscreen/settings.ini"
STARTUP_GIF_PATH = "/home/pi/miniscreen-startup.gif"


API_KEY = "<MY GIPHY KEY>"

# Define global variables
gif = None
miniscreen = Miniscreen()
req = PreparedRequest()
req.prepare_url(
    "http://api.giphy.com/v1/gifs/search",
    urlencode({"q": SEARCH_TERM, "api_key": API_KEY, "limit": f"{SEARCH_LIMIT}"}),
)


def display_instructions_dialog():
    miniscreen.select_button.when_pressed = play_random_gif
    miniscreen.cancel_button.when_pressed = None
    miniscreen.display_multiline_text(
        "Press SELECT to load a random GIF!", font_size=18
    )


def display_user_action_select_dialog():
    miniscreen.select_button.when_pressed = save_gif_as_startup
    miniscreen.cancel_button.when_pressed = play_random_gif
    miniscreen.display_multiline_text(
        "SELECT: save GIF as default startup animation. CANCEL: load new GIF",
```

(continues on next page)

(continued from previous page)

```
        font_size=12,
    )


def display_loading_dialog():
    miniscreen.select_button.when_pressed = None
    miniscreen.cancel_button.when_pressed = display_instructions_dialog
    miniscreen.display_multiline_text("Loading random GIF...", font_size=18)


def display_saving_dialog():
    miniscreen.select_button.when_pressed = None
    miniscreen.cancel_button.when_pressed = None
    miniscreen.display_multiline_text(
        "GIF saved as default startup animation!", font_size=18
    )
    # Saving is fast, so we need to wait a short while for the message to be seen on␣
↪the display
    sleep(1)


def play_random_gif():
    global gif

    # Show "Loading..." while processing for a GIF
    display_loading_dialog()

    # Get GIF data from Giphy
    with urlopen(req.url) as response:
        data = json.loads(response.read())

    # Extract random GIF URL from JSON response
    gif_url = data["data"][randint(0, SEARCH_LIMIT - 1)]["images"]["fixed_height"][
        "url"
    ]

    # Load GIF from URL
    gif = Image.open(urlopen(gif_url))

    # Play one loop of GIF animation
    miniscreen.play_animated_image(gif)

    # Ask user if they want to save it
    display_user_action_select_dialog()


def save_gif_as_startup():
    # Display "saving" dialog
    display_saving_dialog()

    # Save file to home directory
    gif.save(STARTUP_GIF_PATH, save_all=True)

    config = ConfigParser()
    cfg_section = "Bootsplash"

    if not config.has_section(cfg_section):
```

(continues on next page)

(continued from previous page)

```
        config.add_section(cfg_section)

    config.set(cfg_section, "Path", STARTUP_GIF_PATH)

    with open(CONFIG_FILE_PATH, "w") as f:
        config.write(f)

    # Go back to the start
    display_instructions_dialog()


# Display initial dialog
display_instructions_dialog()

# Wait indefinitely for user input
pause()
```

3.5 API - pi-top Device

3.5.1 Pitop

This class represents a pi-top device. Each of the on-board features of pi-tops can be accessed from this object.

Note: This class has been built with pi-top [4] in mind, as is in early development. You may notice that some features do not behave as expected on other platforms.

If you would like to help us with development, please refer to the Contributing document in this repository for information!

Here is some sample code demonstrating how the various subsystems of a pi-top [4] can be accessed and used:

```
from time import sleep

from PIL import Image

from pitop import Pitop

# Set up pi-top
pitop = Pitop()

# Say hi!
pitop.miniscreen.display_text("Hello!")
sleep(2)

# Display battery info
battery_capacity = pitop.battery.capacity
battery_charging = pitop.battery.is_charging

pitop.miniscreen.display_multiline_text(
    "Battery Status:\n"
    f"-Capacity: {battery_capacity}%\n"
    f"-Charging: {battery_charging}",
```

(continues on next page)

(continued from previous page)

```
    font_size=15,
)
sleep(2)


# Configure buttons to do something
keep_running = True


def display_gif_and_exit():
    image = Image.open(
        "/usr/lib/python3/dist-packages/pitop/miniscreen/images/rocket.gif"
    )
    pitop.miniscreen.play_animated_image(image)
    pitop.miniscreen.display_text("Bye!")
    sleep(2)
    global keep_running
    keep_running = False


pitop.miniscreen.select_button.when_pressed = display_gif_and_exit
pitop.miniscreen.cancel_button.when_pressed = display_gif_and_exit
pitop.miniscreen.up_button.when_pressed = display_gif_and_exit
pitop.miniscreen.down_button.when_pressed = display_gif_and_exit

pitop.miniscreen.display_multiline_text("Press any button...", font_size=25)

# Sleep until `display_gif_and_exit` runs
while keep_running:
    sleep(0.3)
```

Although it is possible to access pi-top subsystems individually, it is recommended to access them via this class.

Class Reference: Pitop

class pitop.**Pitop**

Represents a pi-top Device.

When creating a *Pitop* object, multiple properties will be set, depending on the pi-top device that it's running the code. For example, if run on a pi-top [4], a *miniscreen* attribute will be created as an interface to control the miniscreen OLED display, but that won't be available for other pi-top devices.

The Pitop class is a Singleton. This means that only one instance per process will be created. In practice, this means that if in a particular project you instance a Pitop class in 2 different files, they will share the internal state.

***property* miniscreen** If using a pi-top [4], this property returns a *pitop.miniscreen.Miniscreen* object, to interact with the device's Miniscreen.

***property* oled** Refer to *miniscreen*.

***property* battery** If using a pi-top with a battery, this property returns a *pitop.battery.Battery* object, to interact with the device's battery.

own_state

Representation of an object state that will be used to determine the current state of an object.

All pi-tops come with some software-controllable onboard hardware. These sections of the API make it easy to access and change the state of your pi-top hardware.

Using the Pitop object

Attaching objects and saving configuration to a file

```
from time import sleep

from pitop import LED, Pitop
from pitop.robotics.drive_controller import DriveController

pitop = Pitop()
drive_controller = DriveController()
led = LED("D0", name="red_led")

# Add components to the Pitop object
pitop.add_component(drive_controller)
pitop.add_component(led)

# Do something with the object
pitop.red_led.on()
pitop.drive.forward(0.5)
sleep(2)
pitop.red_led.off()
pitop.drive.stop()

# Store configuration to a file
pitop.save_config("/home/pi/my_custom_config.json")
```

Loading an existing configuration

```
from time import sleep

from pitop import Pitop

# Load configuration from a previous session
pitop = Pitop.from_file("/home/pi/my_custom_config.json")

# Check the loaded configuration
print(pitop.config)

# Do something with your device
pitop.red_led.on()
pitop.drive.forward(0.5)
sleep(2)
pitop.red_led.off()
pitop.drive.stop()

# Check the state of all the components attached to the Pitop object
pitop.print_state()
```

3.5.2 pi-top Battery

This class provides a simple way to check the current onboard pi-top battery state, and handle some state change events.

This class will work with original pi-top, pi-top [3] and pi-top [4]. pi-topCEED has no onboard battery, and so will not work.

```
from pitop import Pitop

battery = Pitop().battery

print(f"Battery capacity: {battery.capacity}")
print(f"Battery time remaining: {battery.time_remaining}")
print(f"Battery is charging: {battery.is_charging}")
print(f"Battery is full: {battery.is_full}")
print(f"Battery wattage: {battery.wattage}")


def do_low_battery_thing():
    print("Battery is low!")


def do_critical_battery_thing():
    print("Battery is critically low!")


def do_full_battery_thing():
    print("Battery is full!")


def do_charging_battery_thing():
    print("Battery is charging!")


def do_discharging_battery_thing():
    print("Battery is discharging!")


# To invoke a function when the battery changes state, you can assign the function to␣
↪various 'when_' data members
battery.when_low = do_low_battery_thing
battery.when_critical = do_critical_battery_thing
battery.when_full = do_full_battery_thing
battery.when_charging = do_charging_battery_thing
battery.when_discharging = do_discharging_battery_thing


# Another way to react to battery events is to poll
while True:
    if battery.is_full:
        do_full_battery_thing()
```

Class Reference: pi-top Battery

class pitop.battery.**Battery**

capacity

classmethod get_full_state()

is_charging

is_full

time_remaining

wattage

3.5.3 pi-top Display

This class provides a simple way to check the current onboard pi-top display state, and handle state change events.

This class will work with original pi-top, pi-topCEED and pi-top [3].

Note: Not compatible with pi-top [4].

pi-top [4] has no onboard display, and the official pi-top [4] FHD Display is not software-controllable.

```
from signal import pause
from time import sleep

from pitop import Pitop

pitop = Pitop()
display = pitop.display


# Get display information
print(f"Display brightness: {display.brightness}")
print(f"Display blanking timeout: {display.blanking_timeout}")
print(f"Display backlight is on: {display.backlight}")
print(f"Display lid is open: {display.lid_is_open}")

# Change the brightness levels incrementally
display.increment_brightness()
display.decrement_brightness()

# Set brightness explicitly
display.brightness = 7

# Set screen blank state
display.blank()
display.unblank()

# Set screen blanking timeout (s)
display.blanking_timeout = 60


# Define some functions to call on events
def do_brightness_changed_thing(new_brightness):
    print(new_brightness)
    print("Display brightness has changed!")
```

(continues on next page)

(continued from previous page)

```
def do_screen_blanked_thing():
    print("Display is blanked!")


def do_screen_unblanked_thing():
    print("Display is unblanked!")


def do_lid_closed_thing():
    print("Display lid is closed!")


def do_lid_opened_thing():
    print("Display lid is open!")


# 'Wire up' functions to display events
display.when_brightness_changed = do_brightness_changed_thing
display.when_screen_blanked = do_screen_blanked_thing
display.when_screen_unblanked = do_screen_unblanked_thing
display.when_lid_closed = do_lid_closed_thing
display.when_lid_opened = do_lid_opened_thing


# Wait indefinitely for events to be handled in the background
pause()


# Or alternatively poll
print("Polling for if lid is open (Original pi-top/pi-top [3] only)")
while True:
    if display.lid_is_open:
        do_lid_opened_thing()
    sleep(0.1)
```

Class Reference: pi-top Display

class pitop.display.**Display**

backlight

blank()

blanking_timeout

brightness

decrement_brightness()

increment_brightness()

lid_is_open

unblank()

3.5.4 pi-top [4] Miniscreen

The miniscreen of the pi-top [4] can be found on the front, comprised of an 128x64 pixel OLED screen and 4 programmable buttons.

Check out *Key Concepts: pi-top [4] Miniscreen* for useful information about how this class works.

Using the Miniscreen's OLED Display

The OLED display is an array of pixels that can be either on or off. Unlike the pixels in a more advanced display, such as the monitor you are most likely reading this on, the display is a "1-bit monochromatic" display. Text and images can be displayed by directly manipulating the pixels.

The *pitop.miniscreen.Miniscreen* class directly provides display functions for the OLED.

Displaying text

```
from time import sleep

from pitop import Pitop

pitop = Pitop()
miniscreen = pitop.miniscreen
miniscreen.display_multiline_text("Hello, world!", font_size=20)
sleep(5)
```

Showing an image

```
from time import sleep

from pitop import Pitop

pitop = Pitop()
miniscreen = pitop.miniscreen

miniscreen.display_image_file(
    "/usr/lib/python3/dist-packages/pitop/miniscreen/images/rocket.gif"
)
```

(continues on next page)

(continued from previous page)

```
sleep(2)
```

Loop a GIF

```
from PIL import Image, ImageSequence

from pitop import Pitop

pitop = Pitop()
miniscreen = pitop.miniscreen

rocket = Image.open("/usr/lib/python3/dist-packages/pitop/miniscreen/images/rocket.gif
↪")

while True:
    for frame in ImageSequence.Iterator(rocket):
        miniscreen.display_image(frame)
```

Displaying an GIF once

```
from PIL import Image

from pitop import Pitop

pitop = Pitop()
miniscreen = pitop.miniscreen

rocket = Image.open("/usr/lib/python3/dist-packages/pitop/miniscreen/images/rocket.gif
↪")

miniscreen.play_animated_image(rocket)
```

Displaying an GIF once through frame by frame

```
from PIL import Image, ImageSequence

from pitop import Pitop

pitop = Pitop()
miniscreen = pitop.miniscreen

rocket = Image.open("/usr/lib/python3/dist-packages/pitop/miniscreen/images/rocket.gif
↪")

for frame in ImageSequence.Iterator(rocket):
    miniscreen.display_image(frame)
```

Displaying an GIF looping in background

```
from time import sleep

from PIL import Image

from pitop import Pitop

pitop = Pitop()
miniscreen = pitop.miniscreen

image = Image.open("/usr/lib/python3/dist-packages/pitop/miniscreen/images/rocket.gif
↪")

# Run animation loop in background by setting `background` to True
miniscreen.play_animated_image(image, background=True, loop=True)


# Do stuff while showing image
print("Counting to 100 while showing animated image on miniscreen...")

for i in range(100):
    print("\r{}".format(i), end="", flush=True)
    sleep(0.2)

print("\rFinished!")

# Stop animation
miniscreen.stop_animated_image()
```

Handling basic 2D graphics drawing and displaying

```
from PIL import Image, ImageDraw, ImageFont

from pitop import Pitop

pitop = Pitop()
miniscreen = pitop.miniscreen
image = Image.new(
    miniscreen.mode,
    miniscreen.size,
)
canvas = ImageDraw.Draw(image)
miniscreen.set_max_fps(1)


def clear():
    canvas.rectangle(miniscreen.bounding_box, fill=0)


print("Drawing an arc")
canvas.arc(miniscreen.bounding_box, 0, 180, fill=1, width=1)
miniscreen.display_image(image)

clear()
```

(continues on next page)

(continued from previous page)

```
print("Drawing an image")
# Note: this is an animated file, but this approach will only show the first frame
demo_image = Image.open(
    "/usr/lib/python3/dist-packages/pitop/miniscreen/images/rocket.gif"
).convert("1")
canvas.bitmap((0, 0), demo_image, fill=1)
miniscreen.display_image(image)

clear()

print("Drawing a chord")
canvas.chord(miniscreen.bounding_box, 0, 180, fill=1)
miniscreen.display_image(image)

clear()

print("Drawing an ellipse")
canvas.ellipse(miniscreen.bounding_box, fill=1)
miniscreen.display_image(image)

clear()

print("Drawing a line")
canvas.line(miniscreen.bounding_box, fill=1)
miniscreen.display_image(image)

clear()

print("Drawing a pieslice")
canvas.pieslice(miniscreen.bounding_box, 0, 180, fill=1)
miniscreen.display_image(image)

clear()

print("Drawing a point")
canvas.point(miniscreen.bounding_box, fill=1)
miniscreen.display_image(image)

clear()

print("Drawing a polygon")
canvas.polygon(miniscreen.bounding_box, fill=1)
miniscreen.display_image(image)

clear()

print("Drawing a rectangle")
canvas.rectangle(miniscreen.bounding_box, fill=1)
miniscreen.display_image(image)

clear()

print("Drawing some text")
canvas.text((0, 0), "Hello\nWorld!", font=ImageFont.load_default(), fill=1)
miniscreen.display_image(image)
```

Displaying a clock

```
from datetime import datetime

from PIL import Image, ImageDraw

from pitop import Pitop

pitop = Pitop()
miniscreen = pitop.miniscreen
miniscreen.set_max_fps(1)

image = Image.new(
    miniscreen.mode,
    miniscreen.size,
)
canvas = ImageDraw.Draw(image)

bounding_box = (32, 0, 95, 63)

big_hand_box = (
    bounding_box[0] + 5,
    bounding_box[1] + 5,
    bounding_box[2] - 5,
    bounding_box[3] - 5,
)

little_hand_box = (
    bounding_box[0] + 15,
    bounding_box[1] + 15,
    bounding_box[2] - 15,
    bounding_box[3] - 15,
)

while True:
    current_time = datetime.now()

    # Clear
    canvas.rectangle(bounding_box, fill=0)

    # Draw face
    canvas.ellipse(bounding_box, fill=1)

    # Draw hands
    angle_second = (current_time.second * 360 / 60) - 90
    canvas.pieslice(big_hand_box, angle_second, angle_second + 2, fill=0)

    angle_minute = (current_time.minute * 360 / 60) - 90
    canvas.pieslice(big_hand_box, angle_minute, angle_minute + 5, fill=0)

    angle_hour = (
        (current_time.hour * 360 / 12) + (current_time.minute * 360 / 12 / 60)
    ) - 90
    canvas.pieslice(little_hand_box, angle_hour, angle_hour + 5, fill=0)

    # Display to screen
    miniscreen.display_image(image)
```

Display a particle-based screensaver

```
from random import randint

from PIL import Image, ImageDraw

from pitop import Pitop

pitop = Pitop()
miniscreen = pitop.miniscreen
image = Image.new(
    miniscreen.mode,
    miniscreen.size,
)
canvas = ImageDraw.Draw(image)

speed_factor = 15
particles = []


class Particle:
    def __init__(self, x, y):
        self.x = x
        self.y = y
        self.update()

    def get_position(self):
        return (self.x, self.y)

    def update(self):
        dx = (
            (self.x - (miniscreen.width / 2)) / speed_factor
            if self.x < (miniscreen.width / 2)
            else (self.x - (miniscreen.width / 2)) / speed_factor
        )
        dy = (
            (self.y - (miniscreen.height / 2)) / speed_factor
            if self.y < (miniscreen.height / 2)
            else (self.y - (miniscreen.height / 2)) / speed_factor
        )
        self.x += dx
        self.y += dy


def add_new_particle():
    x = randint(0, miniscreen.width)
    y = randint(0, miniscreen.height)
    particles.append(Particle(x, y))


while True:
    # Clear display
    canvas.rectangle(miniscreen.bounding_box, fill=0)
    particles.clear()

    speed_factor = randint(5, 30)
    particle_count = randint(5, 50)
```

(continues on next page)

(continued from previous page)

```
    for count in range(particle_count):
        add_new_particle()

    for _ in range(100):
        for particle in particles:
            x, y = particle.get_position()

            if (x < 0 or x > miniscreen.width) or (y < 0 or y > miniscreen.height):
                particles.remove(particle)
                add_new_particle()
            else:
                canvas.point((x, y), fill=1)
                particle.update()

        miniscreen.display_image(image)
```

Prim's algorithm

```
from random import randint, random
from time import sleep

from PIL import Image, ImageDraw

from pitop import Pitop

# https://en.wikipedia.org/wiki/Maze_generation_algorithm

pitop = Pitop()
miniscreen = pitop.miniscreen
image = Image.new(
    miniscreen.mode,
    miniscreen.size,
)
canvas = ImageDraw.Draw(image)
miniscreen.set_max_fps(50)


def draw_pixel(pos):
    canvas.point(pos, fill=1)
    miniscreen.display_image(image)
    drawn_pixels.append(pos)


width = (miniscreen.width // 2) * 2 - 1
height = (miniscreen.height // 2) * 2 - 1

while True:
    print("Initialising...")
    canvas.rectangle(miniscreen.bounding_box, fill=0)

    drawn_pixels = list()
    complexity = int(random() * (5 * (width + height)))
    density = int(random() * ((width // 2) * (height // 2)))
```

(continues on next page)

(continued from previous page)

```
    print("Drawing the borders...")

    for x in range(width):
        draw_pixel((x, 0))
        draw_pixel((x, (height // 2) * 2))

    for y in range(height):
        draw_pixel((0, y))
        draw_pixel(((width // 2) * 2, y))

    print("Filling the maze...")

    for i in range(density):
        x, y = randint(0, width // 2) * 2, randint(0, height // 2) * 2
        if (x, y) not in drawn_pixels:
            draw_pixel((x, y))

            for j in range(complexity):
                neighbours = []
                if x > 1:
                    neighbours.append((x - 2, y))
                if x < width - 3:
                    neighbours.append((x + 2, y))
                if y > 1:
                    neighbours.append((x, y - 2))
                if y < height - 3:
                    neighbours.append((x, y + 2))
                if len(neighbours):
                    x_, y_ = neighbours[randint(0, len(neighbours) - 1)]
                    if (x_, y_) not in drawn_pixels:
                        draw_pixel((x_, y_))
                        draw_pixel((x_ + (x - x_) // 2, y_ + (y - y_) // 2))
                        x, y = x_, y_

    print("Done!")

    sleep(10)
```

2-Player Pong Game

```
from random import randrange
from time import sleep

from PIL import Image, ImageDraw, ImageFont

from pitop import Pitop

# Game variables
BALL_RADIUS = 2
PADDLE_SIZE = (2, 20)
PADDLE_CTRL_VEL = 4
```

(continues on next page)

(continued from previous page)

```
class Ball:
    def __init__(self):
        self.pos = [0, 0]
        self.vel = [0, 0]

        # 50/50 chance of direction
        self.init(move_right=randrange(0, 2) == 0)

    def init(self, move_right):
        self.pos = [miniscreen.width // 2, miniscreen.height // 2]

        horz = randrange(1, 3)
        vert = randrange(1, 3)

        if move_right is False:
            horz = -horz

        self.vel = [horz, -vert]

    @property
    def x_pos(self):
        return self.pos[0]

    @property
    def y_pos(self):
        return self.pos[1]

    def is_aligned_with_paddle_horizontally(self, paddle):
        return abs(self.x_pos - paddle.x_pos) <= BALL_RADIUS + PADDLE_SIZE[0] // 2

    def is_aligned_with_paddle_vertically(self, paddle):
        return abs(self.y_pos - paddle.y_pos) <= BALL_RADIUS + PADDLE_SIZE[1] // 2

    def is_touching_paddle(self, paddle):
        hor = self.is_aligned_with_paddle_horizontally(paddle)
        ver = self.is_aligned_with_paddle_vertically(paddle)
        return hor and ver

    @property
    def is_touching_vertical_walls(self):
        return (
            self.y_pos <= BALL_RADIUS
            or self.y_pos >= miniscreen.height + 1 - BALL_RADIUS
        )

    def change_direction(self, change_x=False, change_y=False, speed_factor=1.0):
        x_vel = -self.vel[0] if change_x else self.vel[0]
        self.vel[0] = speed_factor * x_vel

        y_vel = -self.vel[1] if change_y else self.vel[1]
        self.vel[1] = speed_factor * y_vel

    def update(self):
        self.pos = [x + y for x, y in zip(self.pos, self.vel)]

        if self.is_touching_vertical_walls:
            self.change_direction(change_y=True, speed_factor=1.0)
```

(continues on next page)

(continued from previous page)

```
    @property
    def bounding_box(self):
        def get_circle_bounds(center, radius):
            x0 = center[0] - radius
            y0 = center[1] - radius
            x1 = center[0] + radius
            y1 = center[1] + radius
            return (x0, y0, x1, y1)

        return get_circle_bounds(self.pos, BALL_RADIUS)


class Paddle:
    def __init__(self, start_pos=[0, 0]):
        self.pos = start_pos
        self.vel = 0
        self.score = 0

    def increase_score(self):
        self.score += 1

    @property
    def x_pos(self):
        return self.pos[0]

    @property
    def y_pos(self):
        return self.pos[1]

    @y_pos.setter
    def y_pos(self, new_y):
        self.pos[1] = new_y

    @property
    def touching_top(self):
        return self.y_pos - PADDLE_SIZE[1] // 2 <= 0

    @property
    def touching_bottom(self):
        return self.y_pos + PADDLE_SIZE[1] // 2 >= miniscreen.height - 1

    def update(self):
        moving_down = self.vel > 0

        if self.touching_top and not moving_down:
            return

        if self.touching_bottom and moving_down:
            return

        self.y_pos += self.vel

        if self.touching_top:
            self.y_pos = PADDLE_SIZE[1] // 2

        if self.touching_bottom:
```

(continues on next page)

(continued from previous page)

```
            self.y_pos = miniscreen.height - PADDLE_SIZE[1] // 2 - 1

    @property
    def bounding_box(self):
        return (
            self.x_pos,
            self.y_pos - PADDLE_SIZE[1] // 2,
            self.x_pos,
            self.y_pos + PADDLE_SIZE[1] // 2,
        )


def update_button_state():
    down_pressed = miniscreen.down_button.is_pressed
    up_pressed = miniscreen.up_button.is_pressed
    select_pressed = miniscreen.select_button.is_pressed
    cancel_pressed = miniscreen.cancel_button.is_pressed

    if down_pressed == up_pressed:
        l_paddle.vel = 0
    elif down_pressed:
        l_paddle.vel = PADDLE_CTRL_VEL
    elif up_pressed:
        l_paddle.vel = -PADDLE_CTRL_VEL

    if select_pressed == cancel_pressed:
        r_paddle.vel = 0
    elif select_pressed:
        r_paddle.vel = PADDLE_CTRL_VEL
    elif cancel_pressed:
        r_paddle.vel = -PADDLE_CTRL_VEL


def update_positions():
    round_finished = False

    l_paddle.update()
    r_paddle.update()
    ball.update()

    paddles = {l_paddle, r_paddle}
    for paddle in paddles:
        if ball.is_aligned_with_paddle_horizontally(paddle):
            if ball.is_touching_paddle(paddle):
                ball.change_direction(change_x=True, speed_factor=1.1)
            else:
                other_paddle = paddles - {paddle}
                other_paddle = other_paddle.pop()
                other_paddle.increase_score()

                ball.init(move_right=other_paddle == r_paddle)
                paddle.y_pos = miniscreen.height // 2
                other_paddle.y_pos = miniscreen.height // 2

                round_finished = True

            break
```

(continues on next page)

(continued from previous page)

```
    return round_finished


def draw(wait=False):
    canvas = ImageDraw.Draw(image)

    # Clear screen
    canvas.rectangle(miniscreen.bounding_box, fill=0)

    # Draw ball
    canvas.ellipse(ball.bounding_box, fill=1)

    # Draw paddles
    canvas.line(l_paddle.bounding_box, fill=1, width=PADDLE_SIZE[0])

    canvas.line(r_paddle.bounding_box, fill=1, width=PADDLE_SIZE[0])

    # Draw score
    font = ImageFont.truetype("VeraMono.ttf", size=12)
    canvas.multiline_text(
        (1 * miniscreen.width // 3, 2),
        str(l_paddle.score),
        fill=1,
        font=font,
        align="center",
    )
    canvas.multiline_text(
        (2 * miniscreen.width // 3, 2),
        str(r_paddle.score),
        fill=1,
        font=font,
        align="center",
    )

    # Display image
    miniscreen.display_image(image)

    if wait:
        sleep(1.5)


# Internal variables
pitop = Pitop()
miniscreen = pitop.miniscreen
miniscreen.set_max_fps(30)

ball = Ball()

l_paddle = Paddle([PADDLE_SIZE[0] // 2 - 1, miniscreen.height // 2])
r_paddle = Paddle([miniscreen.width - 1 - PADDLE_SIZE[0] // 2, miniscreen.height //␣
↪2])

image = Image.new(
    miniscreen.mode,
    miniscreen.size,
)
```

(continues on next page)

(continued from previous page)

```
def main():
    while True:
        update_button_state()
        draw(update_positions())


if __name__ == "__main__":
    main()
```

Class Reference: pi-top [4] Miniscreen

class pitop.miniscreen.**Miniscreen**

Represents a pi-top [4]'s miniscreen display.

Also owns the surrounding 4 buttons as properties (*up_button*, *down_button*, *select_button*, *cancel_button*). See *pitop.miniscreen.miniscreen.MiniscreenButton* for how to use these buttons.

bottom_left

Gets the bottom-left corner of the miniscreen display.

Returns The coordinates of the bottom left of the display's bounding box as an (x,y) tuple.

Return type tuple

bottom_right

Gets the bottom-right corner of the miniscreen display.

Returns The coordinates of the bottom right of the display's bounding box as an (x,y) tuple.

Return type tuple

bounding_box

Gets the bounding box of the miniscreen display.

Returns The device's bounding box as an (top-left x, top-left y, bottom-right x, bottom-right y) tuple.

Return type tuple

cancel_button

Gets the cancel button of the pi-top [4] miniscreen.

Returns A gpiozero-like button instance representing the cancel button of the pi-top [4] miniscreen.

Return type *pitop.miniscreen.miniscreen.MiniscreenButton*

center

Gets the center of the miniscreen display.

Returns The coordinates of the center of the display's bounding box as an (x,y) tuple.

Return type tuple

clear()

Clears any content displayed in the miniscreen display.

contrast(*new_contrast_value*)
Sets the contrast value of the miniscreen display to the provided value.

Parameters **new_contrast_value** (*int*) – contrast value to set, between 0 and 255.

device
Gets the miniscreen display device instance.

Return type pitop.miniscreen.oled.core.contrib.luma.oled.device.sh1106

display(*force=False*)
Displays what is on the current canvas to the screen as a single frame.

Warning: This method is deprecated and will be deleted on the next major release of the SDK.

This method does not need to be called when using the other *draw* functions in this class, but is used when the caller wants to use the *canvas* object to draw composite objects and then render them to screen in a single frame.

display_image(*image*, *xy=None*, *invert=False*)
Render a static image to the screen from a file or URL at a given position.

The image should be provided as a PIL Image object.

Parameters

- **image** (*Image*) – A PIL Image object to be rendered
- **xy** (*tuple*) – The position on the screen to render the image. If not provided or passed as *None* the image will be drawn in the top-left of the screen.
- **invert** (*bool*) – Set to True to flip the on/off state of each pixel in the image

display_image_file(*file_path_or_url*, *xy=None*, *invert=False*)
Render a static image to the screen from a file or URL at a given position.

The display's positional properties (e.g. *top_left*, *top_right*) can be used to assist with specifying the *xy* position parameter.

Parameters

- **file_path_or_url** (*str*) – A file path or URL to the image
- **xy** (*tuple*) – The position on the screen to render the image. If not provided or passed as *None* the image will be drawn in the top-left of the screen.
- **invert** (*bool*) – Set to True to flip the on/off state of each pixel in the image

display_multiline_text(*text*, *xy=None*, *font_size=None*, *font=None*, *invert=False*, *anchor=None*, *align=None*)
Renders multi-lined text to the screen at a given position and size. Text that is too long for the screen will automatically wrap to the next line.

The display's positional properties (e.g. *top_left*, *top_right*) can be used to assist with specifying the *xy* position parameter.

Parameters

- **text** (*string*) – The text to render
- **xy** (*tuple*) – The position on the screen to render the image. If not provided or passed as *None* the image will be drawn in the top-left of the screen.

- **font_size** (*int*) – The font size in pixels. If not provided or passed as *None*, the default font size will be used
- **font** (*string*) – A filename or path of a TrueType or OpenType font. If not provided or passed as *None*, the default font will be used
- **invert** (*bool*) – Set to True to flip the on/off state of each pixel in the image
- **align** (*str*) – PIL ImageDraw alignment to use
- **anchor** (*str*) – PIL ImageDraw text anchor to use

display_text(*text*, *xy=None*, *font_size=None*, *font=None*, *invert=False*, *align=None*, *anchor=None*)

Renders a single line of text to the screen at a given position and size.

The display's positional properties (e.g. *top_left*, *top_right*) can be used to assist with specifying the *xy* position parameter.

Parameters

- **text** (*string*) – The text to render
- **xy** (*tuple*) – The position on the screen to render the image. If not provided or passed as *None* the image will be drawn in the top-left of the screen.
- **font_size** (*int*) – The font size in pixels. If not provided or passed as *None*, the default font size will be used
- **font** (*string*) – A filename or path of a TrueType or OpenType font. If not provided or passed as *None*, the default font will be used
- **invert** (*bool*) – Set to True to flip the on/off state of each pixel in the image
- **align** (*str*) – PIL ImageDraw alignment to use
- **anchor** (*str*) – PIL ImageDraw text anchor to use

down_button

Gets the down button of the pi-top [4] miniscreen.

Returns A gpiozero-like button instance representing the down button of the pi-top [4] miniscreen.

Return type *pitop.miniscreen.miniscreen.MiniscreenButton*

draw()

warning:: This method is deprecated in favor of *display_image()* and *display_text()*, and will be deleted on the next major release of the SDK.

draw_image(*image*, *xy=None*)

warning:: This method is deprecated in favor of *display_image()*, and will be deleted on the next major release of the SDK.

draw_image_file(*file_path_or_url*, *xy=None*)

warning:: This method is deprecated in favor of *display_image_file()*, and will be deleted on the next major release of the SDK.

draw_multiline_text(*text*, *xy=None*, *font_size=None*)

warning:: This method is deprecated in favor of *display_multiline_text()*, and will be deleted on the next major release of the SDK.

draw_text(*text*, *xy=None*, *font_size=None*)

warning:: This method is deprecated in favor of *display_text()*, and will be deleted on the next major release of the SDK.

height
Gets the height of the miniscreen display.

Return type int

hide()
The miniscreen display is put into low power mode.

The previously shown image will re-appear when show() is given, even if the internal frame buffer has been changed (so long as display() has not been called).

is_active
Determine if the current miniscreen instance is in control of the miniscreen hardware.

Returns whether the miniscreen instance is in control of the miniscreen hardware.

Return type bool

mode

play_animated_image(*image*, *background=False*, *loop=False*)
Render an animation or a image to the screen.

Use stop_animated_image() to end a background animation

Parameters

- **image** (*Image*) – A PIL Image object to be rendered
- **background** (*bool*) – Set whether the image should be in a background thread or in the main thread.
- **loop** (*bool*) – Set whether the image animation should start again when it has finished

play_animated_image_file(*file_path_or_url*, *background=False*, *loop=False*)
Render an animated image to the screen from a file or URL.

Parameters

- **file_path_or_url** (*str*) – A file path or URL to the image
- **background** (*bool*) – Set whether the image should be in a background thread or in the main thread.
- **loop** (*bool*) – Set whether the image animation should start again when it has finished

prepare_image(*image_to_prepare*)
Formats the given image into one that can be used directly by the OLED.

Parameters **image_to_prepare** (PIL.Image.Image) – Image to be formatted.

Return type PIL.Image.Image

refresh()

reset(*force=True*)
Gives the caller access to the miniscreen display (i.e. in the case the system is currently rendering information to the screen) and clears the screen.

select_button
Gets the select button of the pi-top [4] miniscreen.

Returns A gpiozero-like button instance representing the select button of the pi-top [4] miniscreen.

Return type *pitop.miniscreen.miniscreen.MiniscreenButton*

set_control_to_hub()
Signals the pi-top hub to take control of the miniscreen display.

set_control_to_pi()
Signals the pi-top hub to give control of the miniscreen display to the Raspberry Pi.

set_max_fps(*max_fps*)
Set the maximum frames per second that the miniscreen display can display. This method can be useful to control or limit the speed of animations.

This works by blocking on the OLED's display methods if called before the amount of time that a frame should last is not exceeded.

Parameters **max_fps** (*int*) – The maximum frames that can be rendered per second

should_redisplay(*image_to_display=None*)
Determines if the miniscreen display needs to be refreshed, based on the provided image. If no image is provided, the content of the display's deprecated internal canvas property will be used.

Parameters **image_to_display** (PIL.Image.Image or None) – Image to be displayed.

Return type bool

show()
The miniscreen display comes out of low power mode showing the previous image shown before hide() was called (so long as display() has not been called)

size
Gets the size of the miniscreen display as a (width, height) tuple.

Return type tuple

sleep()
The miniscreen display in set to low contrast mode, without modifying the content of the screen.

spi_bus
Gets the SPI bus used by the miniscreen display to receive data as an integer. Setting this property will modify the SPI bus that the OLED uses. You might notice a flicker in the screen.

Parameters **bus** (*int*) – Number of the SPI bus for the OLED to use. Accepted values are *0* or *1*.

stop_animated_image()
Stop background animation started using *start()*, if currently running.

top_left
Gets the top left corner of the miniscreen display.

Returns The coordinates of the center of the display's bounding box as an (x,y) tuple.

Return type tuple

top_right
Gets the top-right corner of the miniscreen display.

Returns The coordinates of the top right of the display's bounding box as an (x,y) tuple.

Return type tuple

up_button
Gets the up button of the pi-top [4] miniscreen.

Returns A gpiozero-like button instance representing the up button of the pi-top [4] miniscreen.

Return type *pitop.miniscreen.miniscreen.MiniscreenButton*

visible
Gets whether the device is currently in low power state.

Returns whether the screen is in low power mode

Return type bool

wake()
The miniscreen display is set to high contrast mode, without modifying the content of the screen.

when_system_controlled
Function to call when user gives back control of the miniscreen to the system.

This is used by pt-miniscreen to update its 'user-controlled' application state.

when_user_controlled
Function to call when user takes control of the miniscreen.

This is used by pt-miniscreen to update its 'user-controlled' application state.

width
Gets the width of the miniscreen display.

Return type int

Using the Miniscreen's Buttons

The miniscreen's buttons are simple, and behave in a similar way to the other button-style components in this SDK. Each miniscreen button can be queried for their "is pressed" state, and also invoke callback functions for when pressed and released.

The *pitop.miniscreen.Miniscreen* class provides these buttons as properties:

```
>>> from pitop import Pitop
>>> pitop = Pitop()
>>> miniscreen = pitop.miniscreen
>>> miniscreen.up_button
```

(continues on next page)

(continued from previous page)

```
<pitop.miniscreen.miniscreen.MiniscreenButton object at 0xb3e44e50>
>>> miniscreen.down_button
<pitop.miniscreen.miniscreen.MiniscreenButton object at 0xb3e44d30>
>>> miniscreen.select_button
<pitop.miniscreen.miniscreen.MiniscreenButton object at 0xb3e44e90>
>>> miniscreen.cancel_button
<pitop.miniscreen.miniscreen.MiniscreenButton object at 0xb3e44e70>
```

Here is an example demonstrating 2 ways to make use of these buttons:

```
from time import sleep

from pitop import Pitop

pitop = Pitop()
miniscreen = pitop.miniscreen
up = miniscreen.up_button
down = miniscreen.down_button


def do_up_thing():
    print("Up button was pressed")


def do_down_thing():
    print("Down button was pressed")


def do_another_thing():
    print("do_another_thing invoked")


def select_something():
    print("select_something called")


# To invoke a function when the button is pressed/released,
# you can assign the function to the 'when_pressed' or 'when_released' data member of␣
↪a button
print("Configuring miniscreen's up and down button events...")
up.when_pressed = do_up_thing
down.when_pressed = do_down_thing
down.when_released = do_another_thing


# Another way to react to button events is to poll the is_pressed data member
print("Polling for if select button is pressed...")
while True:
    if miniscreen.select_button.is_pressed:
        select_something()
        sleep(0.1)
```

Class Reference: pi-top [4] Miniscreen Button

class pitop.miniscreen.miniscreen.**MiniscreenButton**

Represents one of the 4 buttons around the miniscreen display on a pi- top [4].

Should not be created directly - instead, use *pitop.miniscreen.Miniscreen*.

is_pressed
Get or set the button state as a boolean value.

Return type bool

when_pressed
Get or set the 'when pressed' button state callback function. When set, this callback function will be invoked when this event happens.

Parameters **callback** (*Function*) – Callback function to run when a button is pressed.

when_released
Get or set the 'when released' button state callback function. When set, this callback function will be invoked when this event happens.

Parameters **callback** (*Function*) – Callback function to run when a button is released.

3.6 API - pi-top Maker Architecture (PMA) Components

The Foundation & Expansion Plates and all the parts included in the Foundation & Robotics Kit are known as the pi-top Maker Architecture (PMA).

Each PMA component has a Python class provided for it.

Check out *Key Concepts: pi-top Maker Architecture* for useful information to get started with using PMA.

3.6.1 Button

Note: This is a *Digital Component* which connects to a *Digital Port [D0-D7]*.

```
from time import sleep

from pitop import Button

button = Button("D5")


def on_button_pressed():
    print("Pressed!")


def on_button_released():
    print("Released!")


button.when_pressed = on_button_pressed
button.when_released = on_button_released

while True:
    if button.is_pressed is True:  # When button is pressed it will return True
        print(button.value)
    sleep(1)
```

class pitop.pma.**Button**(*port_name*, *name='button'*)
Encapsulates the behaviour of a push-button.

A push-button is a simple switch mechanism for controlling some aspect of a circuit.

Parameters

- **port_name** (*str*) – The ID for the port to which this component is connected
- **name** (*str*) – Component name, defaults to *button*. Used to access this component when added to a *pitop.Pitop* object.

active_time
The length of time (in seconds) that the device has been active for. When the device is inactive, this is None.

close()
Shut down the device and release all associated resources. This method can be called on an already closed device without raising an exception.

This method is primarily intended for interactive use at the command line. It disables the device and releases its pin(s) for use by another device.

You can attempt to do this simply by deleting an object, but unless you've cleaned up all references to the object this may not work (even if you've cleaned up all references, there's still no guarantee the garbage collector will actually delete the object at that point). By contrast, the close method provides a means of ensuring that the object is shut down.

For example, if you have a buzzer connected to port D0, but then wish to attach an LED instead:

```
>>> from pitop import Buzzer, LED
>>> bz = Buzzer("D0")
>>> bz.on()
>>> bz.off()
>>> bz.close()
>>> led = LED("D0")
>>> led.blink()
```

Device descendents can also be used as context managers using the with statement. For example:

```
>>> from pitop import Buzzer, LED
>>> with Buzzer("D0") as bz:
...     bz.on()
...
>>> with LED("D0") as led:
...     led.on()
...
```

closed
Returns True if the device is closed (see the *close()* method). Once a device is closed you can no longer use any other methods or properties to control or query the device.

config
Returns a dictionary with the set of parameters that can be used to recreate an object.

classmethod from_config(*config_dict*)
Creates an instance of a Recreatable object with parameters in the provided dictionary.

classmethod from_file(*path*)
Creates an instance of an object using the JSON file from the provided path.

held_time
The length of time (in seconds) that the device has been held for. This is counted from the first execution of the *when_held* event rather than when the device activated, in contrast to active_time. If the device is not currently held, this is None.

hold_repeat
If True, *when_held* will be executed repeatedly with *hold_time* seconds between each invocation.

hold_time
The length of time (in seconds) to wait after the device is activated, until executing the *when_held* handler. If *hold_repeat* is True, this is also the length of time between invocations of *when_held*.

static import_class(*module_name*, *class_name*)
Imports a class given a module and a class name.

inactive_time
The length of time (in seconds) that the device has been inactive for. When the device is active, this is None.

is_active
Returns True if the device is currently active and False otherwise. This property is usually derived from *value*. Unlike *value*, this is *always* a boolean.

is_held
When True, the device has been active for at least *hold_time* seconds.

is_pressed
Returns True if the device is currently active and False otherwise. This property is usually derived from *value*. Unlike *value*, this is *always* a boolean.

own_state
Representation of an object state that will be used to determine the current state of an object.

pin
The Pin that the device is connected to. This will be None if the device has been closed (see the close() method). When dealing with GPIO pins, query pin.number to discover the GPIO pin (in BCM numbering) that the device is connected to.

pressed_time
The length of time (in seconds) that the device has been active for. When the device is inactive, this is None.

print_config()

print_state()

pull_up
If True, the device uses a pull-up resistor to set the GPIO pin "high" by default.

save_config(*path*)
Stores the set of parameters to recreate an object in a JSON file.

state
Returns a dictionary with the state of the current object and all of its children.

value
Returns 1 if the button is currently pressed, and 0 if it is not.

values
An infinite iterator of values read from *value*.

wait_for_active(*timeout=None*)
Pause the script until the device is activated, or the timeout is reached.

Parameters **timeout** (*float or None*) – Number of seconds to wait before proceeding. If this is None (the default), then wait indefinitely until the device is active.

wait_for_inactive(*timeout=None*)
Pause the script until the device is deactivated, or the timeout is reached.

Parameters **timeout** (*float or None*) – Number of seconds to wait before proceeding. If this is None (the default), then wait indefinitely until the device is inactive.

wait_for_press(*timeout=None*)
Pause the script until the device is activated, or the timeout is reached.

Parameters **timeout** (*float or None*) – Number of seconds to wait before proceeding. If this is None (the default), then wait indefinitely until the device is active.

wait_for_release(*timeout=None*)
Pause the script until the device is deactivated, or the timeout is reached.

Parameters **timeout** (*float or None*) – Number of seconds to wait before proceeding. If this is None (the default), then wait indefinitely until the device is inactive.

when_activated
The function to run when the device changes state from inactive to active.

This can be set to a function which accepts no (mandatory) parameters, or a Python function which accepts a single mandatory parameter (with as many optional parameters as you like). If the function accepts a single mandatory parameter, the device that activated it will be passed as that parameter.

Set this property to None (the default) to disable the event.

when_deactivated
The function to run when the device changes state from active to inactive.

This can be set to a function which accepts no (mandatory) parameters, or a Python function which accepts a single mandatory parameter (with as many optional parameters as you like). If the function accepts a single mandatory parameter, the device that deactivated it will be passed as that parameter.

Set this property to None (the default) to disable the event.

when_held
The function to run when the device has remained active for *hold_time* seconds.

This can be set to a function which accepts no (mandatory) parameters, or a Python function which accepts a single mandatory parameter (with as many optional parameters as you like). If the function accepts a single mandatory parameter, the device that activated will be passed as that parameter.

Set this property to None (the default) to disable the event.

when_pressed
The function to run when the device changes state from inactive to active.

This can be set to a function which accepts no (mandatory) parameters, or a Python function which accepts a single mandatory parameter (with as many optional parameters as you like). If the function accepts a single mandatory parameter, the device that activated it will be passed as that parameter.

Set this property to None (the default) to disable the event.

when_released
The function to run when the device changes state from active to inactive.

This can be set to a function which accepts no (mandatory) parameters, or a Python function which accepts a single mandatory parameter (with as many optional parameters as you like). If the function accepts a single mandatory parameter, the device that deactivated it will be passed as that parameter.

Set this property to None (the default) to disable the event.

3.6.2 Buzzer

Note: This is a *Digital Component* which connects to a *Digital Port [D0-D7]*.

```
from time import sleep

from pitop import Buzzer

buzzer = Buzzer("D0")

buzzer.on()  # Set buzzer sound on
print(buzzer.value)  # Return 1 while the buzzer is on
sleep(2)

buzzer.off()  # Set buzzer sound off
print(buzzer.value)  # Return 0 while the buzzer is off
sleep(2)

buzzer.toggle()  # Swap between on and off states
print(buzzer.value)  # Return the current state of the buzzer

sleep(2)

buzzer.off()
```

class pitop.pma.**Buzzer**(*port_name*, *name='buzzer'*)

Encapsulates the behaviour of a simple buzzer that can be turned on and off.

Parameters

- **port_name** (*str*) – The ID for the port to which this component is connected
- **name** (*str*) – Component name, defaults to *buzzer*. Used to access this component when added to a *pitop.Pitop* object.

active_high
When True, the *value* property is True when the device's pin is high. When False the *value* property is True when the device's pin is low (i.e. the value is inverted).

This property can be set after construction; be warned that changing it will invert *value* (i.e. changing this property doesn't change the device's pin state - it just changes how that state is interpreted).

beep(*on_time=1*, *off_time=1*, *n=None*, *background=True*)
Make the device turn on and off repeatedly.

Parameters

- **on_time** (*float*) – Number of seconds on. Defaults to 1 second.
- **off_time** (*float*) – Number of seconds off. Defaults to 1 second.
- **n** (*int or None*) – Number of times to blink; None (the default) means forever.
- **background** (*bool*) – If True (the default), start a background thread to continue blinking and return immediately. If False, only return when the blink is finished (warning: the default value of *n* will result in this method never returning).

blink(*on_time=1*, *off_time=1*, *n=None*, *background=True*)
Make the device turn on and off repeatedly.

Parameters

- **on_time** (*float*) – Number of seconds on. Defaults to 1 second.
- **off_time** (*float*) – Number of seconds off. Defaults to 1 second.
- **n** (*int or None*) – Number of times to blink; None (the default) means forever.
- **background** (*bool*) – If True (the default), start a background thread to continue blinking and return immediately. If False, only return when the blink is finished (warning: the default value of *n* will result in this method never returning).

close()
Shut down the device and release all associated resources. This method can be called on an already closed device without raising an exception.

This method is primarily intended for interactive use at the command line. It disables the device and releases its pin(s) for use by another device.

You can attempt to do this simply by deleting an object, but unless you've cleaned up all references to the object this may not work (even if you've cleaned up all references, there's still no guarantee the garbage collector will actually delete the object at that point). By contrast, the close method provides a means of ensuring that the object is shut down.

For example, if you have a buzzer connected to port D0, but then wish to attach an LED instead:

```
>>> from pitop import Buzzer, LED
>>> bz = Buzzer("D0")
>>> bz.on()
>>> bz.off()
>>> bz.close()
>>> led = LED("D0")
>>> led.blink()
```

Device descendents can also be used as context managers using the with statement. For example:

```
>>> from pitop import Buzzer, LED
>>> with Buzzer("D0") as bz:
...     bz.on()
...
>>> with LED("D0") as led:
...     led.on()
...
```

closed
Returns True if the device is closed (see the *close()* method). Once a device is closed you can no longer use any other methods or properties to control or query the device.

config
Returns a dictionary with the set of parameters that can be used to recreate an object.

classmethod from_config(*config_dict*)
Creates an instance of a Recreatable object with parameters in the provided dictionary.

classmethod from_file(*path*)
Creates an instance of an object using the JSON file from the provided path.

static import_class(*module_name*, *class_name*)
Imports a class given a module and a class name.

is_active
Returns True if the device is currently active and False otherwise. This property is usually derived from *value*. Unlike *value*, this is *always* a boolean.

off()
Turns the device off.

on()
Turns the device on.

own_state
Representation of an object state that will be used to determine the current state of an object.

pin
The Pin that the device is connected to. This will be None if the device has been closed (see the close() method). When dealing with GPIO pins, query pin.number to discover the GPIO pin (in BCM numbering) that the device is connected to.

print_config()

print_state()

save_config(*path*)
Stores the set of parameters to recreate an object in a JSON file.

source
The iterable to use as a source of values for *value*.

source_delay
The delay (measured in seconds) in the loop used to read values from *source*. Defaults to 0.01 seconds which is generally sufficient to keep CPU usage to a minimum while providing adequate responsiveness.

state
Returns a dictionary with the state of the current object and all of its children.

toggle()
Reverse the state of the device. If it's on, turn it off; if it's off, turn it on.

value
Returns 1 if the device is currently active and 0 otherwise. Setting this property changes the state of the device.

values
An infinite iterator of values read from *value*.

3.6.3 Encoder Motor

Note: This is a *Motor Component* which connects to a *MotorEncoder Port [M0-M3]*.

```
from time import sleep

from pitop import BrakingType, EncoderMotor, ForwardDirection

# Setup the motor

motor = EncoderMotor("M0", ForwardDirection.COUNTER_CLOCKWISE)
motor.braking_type = BrakingType.COAST


# Move in both directions

rpm_speed = 100
for _ in range(4):
    motor.set_target_rpm(rpm_speed)
    sleep(2)
    motor.set_target_rpm(-rpm_speed)
    sleep(2)

motor.stop()
```

class pitop.pma.**EncoderMotor**(*port_name*, *forward_direction*, *braking_type=<BrakingType.BRAKE: 1>*, *wheel_diameter=0.075*, *name='encoder_motor'*)

Represents a pi-top motor encoder component.

Note that pi-top motor encoders use a built-in closed-loop control system, that feeds the readings from an encoder sensor to an PID controller. This controller will actively modify the motor's current to move at the desired speed or position, even if a load is applied to the shaft.

This internal controller is used when moving the motor through *set_target_rpm* or *set_target_speed* methods, while using the *set_power* method will make the motor work in open-loop, not using the controller.

Note: Note that some methods allow to use distance and speed settings in meters and meters per second. These will only make sense when using a wheel attached to the shaft of the motor.

The conversions between angle, rotations and RPM used by the motor to meters and meters/second are performed considering the *wheel_diameter* parameter. This parameter defaults to the diameter of the wheel included with MMK. If a wheel of different dimmensions is attached to the motor, you'll need to measure it's diameter, in order for these methods to work properly.

Parameters

- **port_name** (*str*) – The ID for the port to which this component is connected.
- **forward_direction** (ForwardDirection) – The type of rotation of the motor shaft that corresponds to forward motion.
- **braking_type** (BrakingType) – The braking type of the motor. Defaults to coast.
- **wheel_diameter** (*int or float*) – The diameter of the wheel attached to the motor.
- **name** (*str*) – Component name, defaults to *encoder_motor*. Used to access this component when added to a *pitop.Pitop* object.

backward(*target_speed*, *distance=0.0*)

Run the wheel backwards at the desired speed in meters per second.

This method is a simple interface to move the wheel that wraps a call to *set_target_speed*, specifying the back direction.

If desired, a *distance* to travel can also be specified in meters, after which the motor will stop. Setting *distance* to 0 will set the motor to run indefinitely until stopped.

Note: Note that for this method to move the wheel the expected *distance*, the correct *wheel_circumference* value needs to be used.

Parameters

- **target_speed** (*int or float*) – Desired speed in m/s
- **distance** (*int or float*) – Total distance to travel in m. Set to 0 to run indefinitely.

braking_type

Returns the type of braking used by the motor when it's stopping after a movement.

Setting this property will change the way the motor stops a movement:

- BrakingType.COAST will make the motor coast to a halt when stopped.
- BrakingType.BRAKE will cause the motor to actively brake when stopped.

Parameters **braking_type** (BrakingType) – The braking type of the motor.

current_rpm

Returns the actual RPM currently being achieved at the output shaft, measured by the encoder sensor.

This value might differ from the target RPM set through *set_target_rpm*.

current_speed

Returns the speed currently being achieved by the motor in meters per second.

This value may differ from the target speed set through *set_target_speed*.

distance

Returns the distance the wheel has travelled in meters.

This value depends on the correct *wheel_circumference* value being set.

forward(*target_speed*, *distance=0.0*)

Run the wheel forward at the desired speed in meters per second.

This method is a simple interface to move the motor that wraps a call to *set_target_speed*, specifying the forward direction.

If desired, a *distance* to travel can also be specified in meters, after which the motor will stop. Setting *distance* to 0 will set the motor to run indefinitely until stopped.

Note: Note that for this method to move the wheel the expected *distance*, the correct *wheel_circumference* value needs to be used.

Parameters

- **target_speed** (*int or float*) – Desired speed in m/s
- **distance** (*int or float*) – Total distance to travel in m. Set to 0 to run indefinitely.

forward_direction

Represents the forward direction setting used by the motor.

Setting this property will determine on which direction the motor will turn whenever a movement in a particular direction is requested.

Parameters **forward_direction** (ForwardDirection) – The direction that corresponds to forward motion.

max_rpm

Returns the approximate maximum RPM capable given the motor and gear ratio.

max_speed

The approximate maximum speed possible for the wheel attached to the motor shaft, given the motor specs, gear ratio and wheel circumference.

This value depends on the correct *wheel_circumference* value being set.

own_state

Representation of an object state that will be used to determine the current state of an object.

power()

Get the current power of the motor.

Returns a value from -1.0 to +1.0, assuming the user is controlling the motor using the *set_power* method (motor is in control mode 0). If this is not the case, returns None.

rotation_counter

Returns the total or partial number of rotations performed by the motor shaft.

Rotations will increment when moving forward, and decrement when moving backward. This value is a float with many decimal points of accuracy, so can be used to monitor even very small turns of the output shaft.

set_power(*power*, *direction=<Direction.FORWARD: 1>*)

Turn the motor on at the power level provided, in the range -1.0 to.

+1.0, where:

- 1.0: motor will turn with full power in the `direction` provided as argument.
- 0.0: motor will not move.
- -1.0: motor will turn with full power in the direction contrary to `direction`.

Warning: Setting a `power` value out of range will cause the method to raise an exception.

Parameters

- **power** (*int or float*) – Motor power, in the range -1.0 to +1.0
- **direction** (Direction) – Direction to rotate the motor

set_target_rpm(*target_rpm*, *direction=<Direction.FORWARD: 1>*, *total_rotations=0.0*)

Run the motor at the specified target_rpm RPM.

If desired, a number of full or partial rotations can also be set through the total_rotations parameter. Once reached, the motor will stop. Setting total_rotations to 0 will set the motor to run indefinitely until stopped.

If the desired RPM setting cannot be achieved, *torque_limited* will be set to True and the motor will run at the maximum possible RPM it is capable of for the instantaneous torque. This means that if the torque lowers, then the RPM will continue to rise until it meets the desired level.

Care needs to be taken here if you want to drive a vehicle forward in a straight line, as the motors are not guaranteed to spin at the same rate if they are torque-limited.

Warning: Setting a target_rpm higher than the maximum allowed will cause the method to throw an exception. To determine what the maximum possible target RPM for the motor is, use the *max_rpm* method.

Parameters

- **target_rpm** (*int or float*) – Desired RPM of output shaft
- **direction** (Direction) – Direction to rotate the motor. Defaults to forward.
- **total_rotations** (*int or float*) – Total number of rotations to be execute. Set to 0 to run indefinitely.

set_target_speed(*target_speed*, *direction=<Direction.FORWARD: 1>*, *distance=0.0*)

Run the wheel at the specified target speed in meters per second.

If desired, a *distance* to travel can also be specified in meters, after which the motor will stop. Setting *distance* to 0 will set the motor to run indefinitely until stopped.

Warning: Setting a target_speed higher than the maximum allowed will cause the method to throw an exception. To determine what the maximum possible target speed for the motor is, use the *max_speed* method.

Note: Note that for this method to move the wheel the expected *distance*, the correct *wheel_diameter* value needs to be used.

Parameters

- **target_speed** (*int or float*) – Desired speed in m/s
- **direction** (Direction) – Direction to rotate the motor. Defaults to forward.
- **distance** (*int or float*) – Total distance to travel in m. Set to 0 to run indefinitely.

stop()
Stop the motor in all circumstances.

target_rpm()
Get the desired RPM of the motor output shaft, assuming the user is controlling the motor using *set_target_rpm* (motor is in control mode 1).

If this is not the case, returns None.

torque_limited
Check if the actual motor speed or RPM does not match the target speed or RPM.

Returns a boolean value, `True` if the motor is torque- limited and `False` if it is not.

wheel_circumference

wheel_diameter
Represents the diameter of the wheel attached to the motor in meters.

This parameter is important if using library functions to measure speed or distance, as these rely on knowing the diameter of the wheel in order to function correctly. Use one of the predefined pi-top wheel and tyre types, or define your own wheel size.

Note: Note the following diameters:

- pi-top MMK Standard Wheel: 0.060.0m
- pi-top MMK Standard Wheel with tank track: 0.070m

Parameters **wheel_diameter** (*int or float*) – Wheel diameter in meters.

Parameters

class pitop.pma.parameters.**BrakingType**
Braking types.

BRAKE = 1

COAST = 0

class pitop.pma.parameters.**ForwardDirection**
Forward directions.

CLOCKWISE = 1

COUNTER_CLOCKWISE = -1

class pitop.pma.parameters.**Direction**
Directions.

BACK = -1

FORWARD = 1

3.6.4 LED

Note: This is a *Digital Component* which connects to a *Digital Port [D0-D7]*.

```
from time import sleep

from pitop import LED

led = LED("D2")

led.on()
print(led.is_lit)
sleep(1)

led.off()
print(led.is_lit)
sleep(1)

led.toggle()
print(led.is_lit)
sleep(1)

print(led.value)  # Returns 1 is the led is on or 0 if the led is off
```

class pitop.pma.**LED**(*port_name*, *name='led'*, *color=None*)

Encapsulates the behaviour of an LED.

An LED (Light Emitting Diode) is a simple light source that can be controlled directly.

Parameters

- **port_name** (*str*) – The ID for the port to which this component is connected
- **name** (*str*) – Component name, defaults to *led*. Used to access this component when added to a *pitop.Pitop* object.

active_high
When True, the *value* property is True when the device's pin is high. When False the *value* property is True when the device's pin is low (i.e. the value is inverted).

This property can be set after construction; be warned that changing it will invert *value* (i.e. changing this property doesn't change the device's pin state - it just changes how that state is interpreted).

blink(*on_time=1*, *off_time=1*, *n=None*, *background=True*)
Make the device turn on and off repeatedly.

Parameters

- **on_time** (*float*) – Number of seconds on. Defaults to 1 second.
- **off_time** (*float*) – Number of seconds off. Defaults to 1 second.
- **n** (*int or None*) – Number of times to blink; None (the default) means forever.
- **background** (*bool*) – If True (the default), start a background thread to continue blinking and return immediately. If False, only return when the blink is finished (warning: the default value of *n* will result in this method never returning).

close()
Shut down the device and release all associated resources. This method can be called on an already closed device without raising an exception.

This method is primarily intended for interactive use at the command line. It disables the device and releases its pin(s) for use by another device.

You can attempt to do this simply by deleting an object, but unless you've cleaned up all references to the object this may not work (even if you've cleaned up all references, there's still no guarantee the garbage collector will actually delete the object at that point). By contrast, the close method provides a means of ensuring that the object is shut down.

For example, if you have a buzzer connected to port D0, but then wish to attach an LED instead:

```
>>> from pitop import Buzzer, LED
>>> bz = Buzzer("D0")
>>> bz.on()
>>> bz.off()
>>> bz.close()
>>> led = LED("D0")
>>> led.blink()
```

Device descendents can also be used as context managers using the with statement. For example:

```
>>> from pitop import Buzzer, LED
>>> with Buzzer("D0") as bz:
...     bz.on()
...
>>> with LED("D0") as led:
...     led.on()
...
```

closed
Returns True if the device is closed (see the *close()* method). Once a device is closed you can no longer use any other methods or properties to control or query the device.

config
Returns a dictionary with the set of parameters that can be used to recreate an object.

classmethod from_config(*config_dict*)
Creates an instance of a Recreatable object with parameters in the provided dictionary.

classmethod from_file(*path*)
Creates an instance of an object using the JSON file from the provided path.

static import_class(*module_name*, *class_name*)
Imports a class given a module and a class name.

is_active
Returns `True` if the device is currently active and `False` otherwise. This property is usually derived from *value*. Unlike *value*, this is *always* a boolean.

is_lit
Returns `True` if the device is currently active and `False` otherwise. This property is usually derived from *value*. Unlike *value*, this is *always* a boolean.

off()
Turns the device off.

on()
Turns the device on.

own_state
Representation of an object state that will be used to determine the current state of an object.

pin
The `Pin` that the device is connected to. This will be `None` if the device has been closed (see the `close()` method). When dealing with GPIO pins, query `pin.number` to discover the GPIO pin (in BCM numbering) that the device is connected to.

print_config()

print_state()

save_config(*path*)
Stores the set of parameters to recreate an object in a JSON file.

source
The iterable to use as a source of values for *value*.

source_delay
The delay (measured in seconds) in the loop used to read values from *source*. Defaults to 0.01 seconds which is generally sufficient to keep CPU usage to a minimum while providing adequate responsiveness.

state
Returns a dictionary with the state of the current object and all of its children.

toggle()
Reverse the state of the device. If it's on, turn it off; if it's off, turn it on.

value
Returns 1 if the device is currently active and 0 otherwise. Setting this property changes the state of the device.

values
An infinite iterator of values read from *value*.

3.6.5 Light Sensor

Note: This is a *Analog Component* which connects to a *Analog Port [A0-A3]*.

```
from time import sleep

from pitop import LightSensor

light_sensor = LightSensor("A1")

while True:
    # Returns a value depending on the amount of light
    print(light_sensor.reading)
    sleep(0.1)
```

class pitop.pma.**LightSensor**(*port_name*, *pin_number=1*, *name='light_sensor'*, *number_of_samples=3*)

Encapsulates the behaviour of a light sensor module.

A simple analogue photo transistor is used to detect the intensity of the light striking the sensor. The component contains a photoresistor which detects light intensity. The resistance decreases as light intensity increases; thus the brighter the light, the higher the voltage.

Uses an Analog-to-Digital Converter (ADC) to turn the analog reading from the sensor into a digital value.

By default, the sensor uses 3 samples to report a *reading*, which takes around 0.5s. This can be changed by modifying the parameter number_of_samples in the constructor.

Parameters

- **port_name** (*str*) – The ID for the port to which this component is connected

- **number_of_samples** (*str*) – Amount of sensor samples used to report a *reading*. Defaults to 3.
- **name** (*str*) – Component name, defaults to *light_sensor*. Used to access this component when added to a *pitop.Pitop* object.

own_state
Representation of an object state that will be used to determine the current state of an object.

reading
Take a reading from the sensor.

Returns A value representing the amount of light striking the sensor at the current time from 0 to 999.

Return type float

value
Get a simple binary value based on a reading from the device.

Returns 1 if the sensor is detecting any light, 0 otherwise

Return type integer

3.6.6 Potentiometer

Note: This is a *Analog Component* which connects to a *Analog Port [A0-A3]*.

```
from time import sleep
```

(continues on next page)

(continued from previous page)

```
from pitop import Potentiometer

potentiometer = Potentiometer("A3")

while True:
    # Returns the current position of the Potentiometer
    print(potentiometer.position)
    sleep(0.1)
```

class pitop.pma.**Potentiometer**(*port_name*, *pin_number=1*, *name='potentiometer'*, *number_of_samples=1*)

Encapsulates the behaviour of a potentiometer.

A potentiometer is a three-terminal resistor with a sliding or rotating contact that forms an adjustable voltage divider. The component is used for measuring the electric potential (voltage) between the two 'end' terminals. If only two of the terminals are used, one end and the wiper, it acts as a variable resistor or rheostat. Potentiometers are commonly used to control electrical devices such as volume controls on audio equipment.

Uses an Analog-to-Digital Converter (ADC) to turn the analog reading from the sensor into a digital value.

Parameters

- **port_name** (*str*) – The ID for the port to which this component is connected
- **number_of_samples** (*str*) – Amount of sensor samples used to report a *position*. Defaults to 1.
- **name** (*str*) – Component name, defaults to *potentiometer*. Used to access this component when added to a *pitop.Pitop* object.

own_state

Representation of an object state that will be used to determine the current state of an object.

position

Get the current reading from the sensor.

Returns A value representing the potential difference (voltage) from 0 to 999.

Return type float

value

Get a simple binary value based on a reading from the device.

Returns 1 if the sensor is detecting a potential difference (voltage), 0 otherwise

Return type integer

3.6.7 Servo Motor

Note: This is a *Motor Component* which connects to a *ServoMotor Port [S0-S3]*.

```
from time import sleep

from pitop import ServoMotor, ServoMotorSetting

servo = ServoMotor("S0")
```

(continues on next page)

(continued from previous page)

```
# Scan back and forward across a 180 degree angle range in 30 degree hops using␣
↪default servo speed
for angle in range(90, -100, -30):
    print("Setting angle to", angle)
    servo.target_angle = angle
    sleep(0.5)

# you can also set angle with a different speed than the default
servo_settings = ServoMotorSetting()
servo_settings.speed = 25

for angle in range(-90, 100, 30):
    print("Setting angle to", angle)
    servo_settings.angle = angle
    servo.setting = servo_settings
    sleep(0.5)

sleep(1)

# Scan back and forward displaying current angle and speed
STOP_ANGLE = 80
TARGET_SPEED = 40

print("Sweeping using speed ", -TARGET_SPEED)
servo.target_speed = -TARGET_SPEED

current_state = servo.setting
current_angle = current_state.angle

# sweep using the already set servo speed
servo.sweep()
while current_angle > -STOP_ANGLE:
    current_state = servo.setting
    current_angle = current_state.angle
    current_speed = current_state.speed
    print(f"current_angle: {current_angle} | current_speed: {current_speed}")
    sleep(0.05)

print("Sweeping using speed ", TARGET_SPEED)

# you can also sweep specifying the speed when calling the sweep method
servo.sweep(speed=TARGET_SPEED)
while current_angle < STOP_ANGLE:
    current_state = servo.setting
    current_angle = current_state.angle
    current_speed = current_state.speed
    print(f"current_angle: {current_angle} | current_speed: {current_speed}")
    sleep(0.05)
```

class pitop.pma.**ServoMotor**(*port_name*, *zero_point=0*, *name='servo'*)

Represents a pi-top servo motor component.

Note that pi-top servo motors use an open-loop control system. As such, the output of the device (e.g. the angle and speed of the servo horn) cannot be measured directly. This means that you can set a target angle or speed for the servo, but you cannot read the current angle or speed.

Parameters

- **port_name** (*str*) – The ID for the port to which this component is connected.
- **zero_point** (*int*) – A user-defined offset from 'true' zero.
- **name** (*str*) – Component name, defaults to *servo*. Used to access this component when added to a *pitop.Pitop* object.

angle_range
Returns a tuple with minimum and maximum possible angles where the servo horn can be moved to.

If *zero_point* is set to 0 (default), the angle range will be (-90, 90).

current_angle
Returns the current angle that the servo motor is at.

Note: If you need synchronized angle and speed values, use `ServoMotor.state()` instead, this will return both current angle and current speed at the same time.

Returns float value of the current angle of the servo motor in degrees.

current_speed
Returns the current speed the servo motor is at.

Note: If you need synchronized angle and speed values, use `ServoMotor.state()` instead, this will return both current angle and current speed at the same time.

Returns float value of the current speed of the servo motor in deg/s.

own_state
Representation of an object state that will be used to determine the current state of an object.

setting
Returns the current state of the servo motor, giving current angle and current speed.

Returns :class:'ServoMotorSetting' object that has angle and speed attributes.

smooth_acceleration
Gets whether or not the servo is configured to use a linear acceleration profile to ramp speed at start and end of cycle.

Returns boolean value of the acceleration mode

stop()
Stop servo at its current position.

Returns None

sweep(*speed=None*)
Moves the servo horn from the current position to one of the servo motor limits (maximum/minimum possible angle), moving at the specified speed. The speed value must be a number from -100.0 to 100.0 deg/s.

The sweep direction is given by the speed.

Setting a `speed` value higher than zero will move the horn to the maximum angle (90 degrees by default), while a value less than zero will move it to the minimum angle (-90 degress by default).

> **Warning:** Using a `speed` out of the valid speed range will cause the method to raise an exception.

Parameters **speed** (*int or float*) – The target speed at which to move the servo horn, from -100 to 100 deg/s.

target_angle
Returns the last target angle that has been set.

Returns float value of the target angle of the servo motor in deg.

target_speed
Returns the last target speed that has been set.

Returns float value of the target speed of the servo motor in deg/s.

zero_point
Represents the servo motor angle that the library treats as 'zero'. This value can be anywhere in the range of -90 to +90.

For example, if the zero_point were set to be -30, then the valid range of values for setting the angle would be -60 to +120.

> **Warning:** Setting a zero point out of the range of -90 to 90 will cause the method to raise an exception.

3.6.8 Sound Sensor

Note: This is a *Analog Component* which connects to a *Analog Port [A0-A3]*.

```
from time import sleep

from pitop import SoundSensor

sound_sensor = SoundSensor("A2")

while True:
    # Returns reading the amount of sound in the room
    print(sound_sensor.reading)
    sleep(0.1)
```

class pitop.pma.**SoundSensor**(*port_name*, *pin_number=1*, *name='sound_sensor'*, *number_of_samples=1*)

Encapsulates the behaviour of a sound sensor.

A sound sensor component is typically a simple microphone that detects the vibrations of the air entering the sensor and produces an analog reading based on the amplitude of these vibrations.

Uses an Analog-to-Digital Converter (ADC) to turn the analog reading from the sensor into a digital value.

Parameters

- **port_name** (*str*) – The ID for the port to which this component is connected
- **number_of_samples** (*str*) – Amount of sensor samples used to report a *reading*. Defaults to 1.
- **name** (*str*) – Component name, defaults to *sound_sensor*. Used to access this component when added to a *pitop.Pitop* object.

own_state

Representation of an object state that will be used to determine the current state of an object.

reading

Take a reading from the sensor. Uses a builtin peak detection system to retrieve the sound level.

Returns A value representing the volume of sound detected by the sensor at the current time from 0 to 500.

Return type float

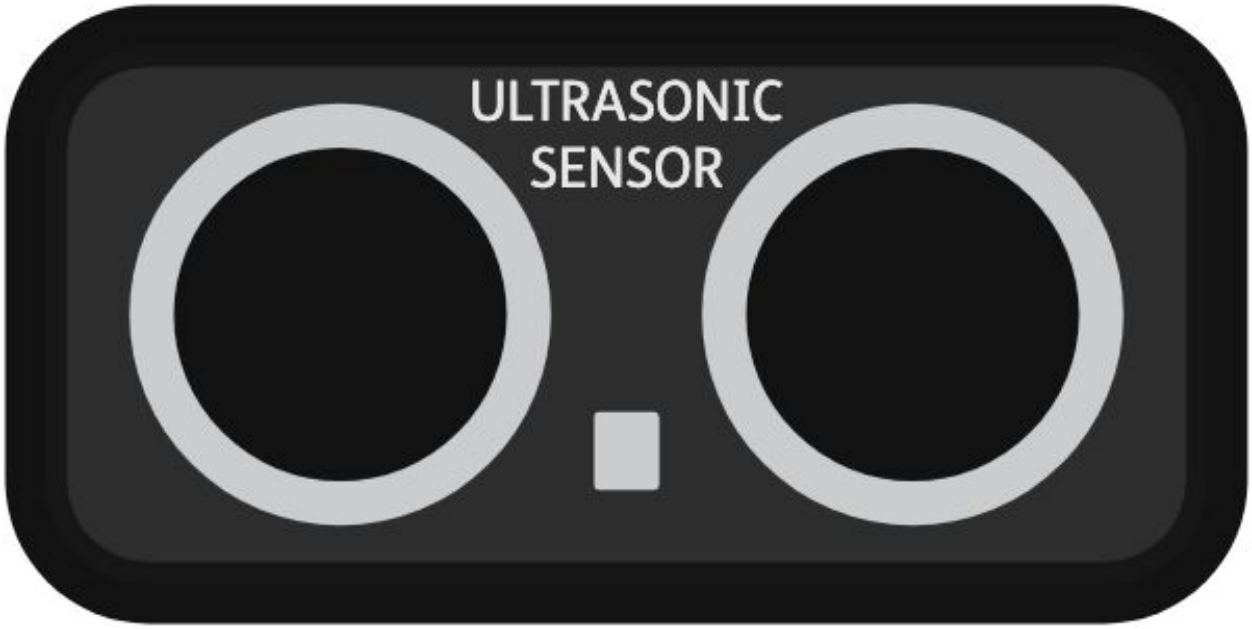
value

Get a simple binary value based on a reading from the device.

Returns 1 if the sensor is detecting any sound, 0 otherwise

Return type integer

3.6.9 Ultrasonic Sensor



Note: This is a *Digital Component* which connects to a *Digital Port [D0-D7]*.

```
from time import sleep

from pitop import UltrasonicSensor

distance_sensor = UltrasonicSensor("D3", threshold_distance=0.2)

# Set up functions to print when an object crosses 'threshold_distance'
distance_sensor.when_in_range = lambda: print("in range")
distance_sensor.when_out_of_range = lambda: print("out of range")

while True:
    # Print the distance (in meters) to an object in front of the sensor
    print(distance_sensor.distance)
    sleep(0.1)
```

class pitop.pma.**UltrasonicSensor**(*port_name*, *queue_len=5*, *max_distance=3*, *threshold_distance=0.3*, *partial=False*, *name='ultrasonic'*)

close()

Shut down the device and release all associated resources. This method can be called on an already closed device without raising an exception.

This method is primarily intended for interactive use at the command line. It disables the device and releases its pin(s) for use by another device.

You can attempt to do this simply by deleting an object, but unless you've cleaned up all references to the object this may not work (even if you've cleaned up all references, there's still no guarantee the garbage collector will actually delete the object at that point). By contrast, the close method provides a means of ensuring that the object is shut down.

For example, if you have a buzzer connected to port D0, but then wish to attach an LED instead:

```
>>> from pitop import Buzzer, LED
>>> bz = Buzzer("D0")
>>> bz.on()
>>> bz.off()
>>> bz.close()
>>> led = LED("D0")
>>> led.blink()
```

Device descendents can also be used as context managers using the with statement. For example:

```
>>> from pitop import Buzzer, LED
>>> with Buzzer("D0") as bz:
...     bz.on()
...
>>> with LED("D0") as led:
...     led.on()
...
```

distance
Returns the current distance measured by the sensor in meters.

Note that this property will have a value between 0 and *max_distance*.

in_range

max_distance
The maximum distance that the sensor will measure in meters.

This value is specified in the constructor and is used to provide the scaling for the value attribute. When *distance* is equal to *max_distance*, value will be 1.

own_state
Representation of an object state that will be used to determine the current state of an object.

pin

threshold_distance
The distance, measured in meters, that will trigger the *when_in_range* and *when_out_of_range* events when crossed. This is simply a meter-scaled variant of the usual threshold attribute.

value
Returns a value between 0, indicating that something is either touching the sensor or is sufficiently near that the sensor can't tell the difference, and 1, indicating that something is at or beyond the specified *max_distance*.

wait_for_in_range(*timeout=None*)

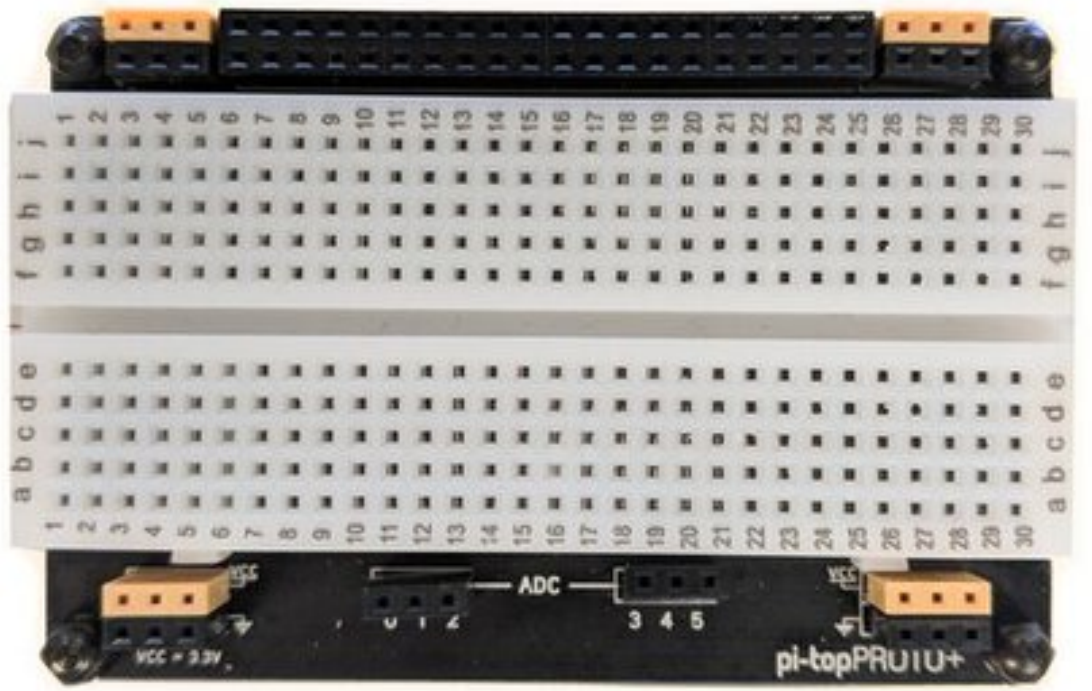
wait_for_out_of_range(*timeout=None*)

when_in_range

when_out_of_range

3.7 API - pi-top Peripheral Devices

3.7.1 pi-topPROTO+



This module provides 2 classes - a simple way to use a pi-topPROTO+'s onboard ADC (analog-to-digital converter), and another to use it as a distance sensor.

These classes will work with original pi-top, pi-topCEED and pi-top [3]. pi-top [4] does not support the pi-topPROTO+'s modular rail connector, and so will not work.

Using the pi-topPROTO+ as a Distance Sensor

```
from time import sleep

from pitop.protoplus import DistanceSensor

ultrasonic = DistanceSensor()

while True:
    print(ultrasonic.distance)
    sleep(1)
```

Class Reference: pi-topPROTO+ Distance Sensor

class pitop.protoplus.sensors.**DistanceSensor**(*trigger_gpio_pin=23*, *echo_gpio_pin=27*)

Encapsulates the behaviour of a simple DistanceSensor that can be turned on and off.

Parameters

- **trigger_gpio_pin** (*str*) – GPIO pin for trigger input
- **echo_gpio_pin** (*str*) – GPIO pin for echo response

close()

Shut down the device and release all associated resources. This method can be called on an already closed device without raising an exception.

This method is primarily intended for interactive use at the command line. It disables the device and releases its pin(s) for use by another device.

You can attempt to do this simply by deleting an object, but unless you've cleaned up all references to the object this may not work (even if you've cleaned up all references, there's still no guarantee the garbage collector will actually delete the object at that point). By contrast, the close method provides a means of ensuring that the object is shut down.

For example, if you have a buzzer connected to port D0, but then wish to attach an LED instead:

```
>>> from pitop import Buzzer, LED
>>> bz = Buzzer("D0")
>>> bz.on()
>>> bz.off()
>>> bz.close()
>>> led = LED("D0")
>>> led.blink()
```

Device descendents can also be used as context managers using the with statement. For example:

```
>>> from pitop import Buzzer, LED
>>> with Buzzer("D0") as bz:
...     bz.on()
...
>>> with LED("D0") as led:
...     led.on()
...
```

get_distance()

get_raw_distance()

raw_distance

Using the pi-topPROTO+'s onboard ADC

```
from time import sleep

from pitop.protoplus import ADCProbe

temp_sensor = ADCProbe()

while True:
    print(temp_sensor.read_value(1))
    sleep(0.5)
```

Class Reference: pi-topPROTO+ ADC Probe

class pitop.protoplus.adc.**ADCProbe**(*i2c_device_name='/dev/i2c-1'*)

poll(*delay=0.5*)

read_all()

read_value(*channel*)

3.7.2 pi-topPULSE

This module provides a simple way to use a pi-topPULSE, and will work with any Raspberry Pi and/or pi-top.

The hardware representation of each color is 5 bits (i.e. only 32 different values). Without gamma correction, this would mean the actual color value changes only every 8th color intensity value. This module applies gamma correction, which means that pixels with seemingly different intensities actually have the same.

Using the pi-topPULSE's microphone

```
from time import sleep

from pitop.pulse import ledmatrix, microphone


def set_bit_rate_to_unsigned_8():
    print("Setting bit rate to 8...")
    microphone.set_bit_rate_to_unsigned_8()


def set_bit_rate_to_signed_16():
    print("Setting bit rate to 16...")
```

(continues on next page)

(continued from previous page)

```
    microphone.set_bit_rate_to_signed_16()


def set_sample_rate_to_16khz():
    print("Setting sample rate to 16KHz...")
    microphone.set_sample_rate_to_16khz()


def set_sample_rate_to_22khz():
    print("Setting sample rate to 22KHz...")
    microphone.set_sample_rate_to_22khz()


def pause(length):
    ledmatrix.off()
    sleep(length)


def record(record_time, output_file, pause_time=1):
    print("Recording audio for " + str(record_time) + "s...")
    ledmatrix.set_all(255, 0, 0)
    ledmatrix.show()
    microphone.record()
    sleep(record_time)
    microphone.stop()
    ledmatrix.off()
    microphone.save(output_file, True)
    print("Saved to " + output_file)
    print("")
    pause(pause_time)


set_sample_rate_to_22khz()

set_bit_rate_to_unsigned_8()
record(5, "/tmp/test22-8.wav")

set_bit_rate_to_signed_16()
record(5, "/tmp/test22-16.wav")


set_sample_rate_to_16khz()

set_bit_rate_to_unsigned_8()
record(5, "/tmp/test16-8.wav")

set_bit_rate_to_signed_16()
record(5, "/tmp/test16-16.wav")
```

Using the pi-topPULSE's LED matrix: Test colors

```
import time

from pitop.pulse import ledmatrix

```

(continues on next page)

(continued from previous page)

```
def show_map(r, g, b):
    for x in range(0, 7):
        for y in range(0, 7):
            z = (float(y) + 7.0 * float(x)) / 49.0
            rr = int(z * r)
            gg = int(z * g)
            bb = int(z * b)
            ledmatrix.set_pixel(x, y, rr, gg, bb)
    ledmatrix.show()


ledmatrix.rotation(0)
ledmatrix.clear()

# Display 49 different color intensities
for r in range(0, 2):
    for g in range(0, 2):
        for b in range(2):
            if r + g + b > 0:
                rr = 255 * r
                gg = 255 * g
                bb = 255 * b
                print(rr, gg, bb)
                show_map(rr, gg, bb)
                time.sleep(5)

ledmatrix.clear()
ledmatrix.show()
```

Using the pi-topPULSE's LED matrix: Fancy Light Show!

```
import colorsys
import math

from pitop.pulse import ledmatrix

s_width, s_height = ledmatrix.get_shape()


# twisty swirly goodness
def swirl(x, y, step):
    x -= s_width / 2
    y -= s_height / 2

    dist = math.sqrt(pow(x, 2) + pow(y, 2)) / 2.0
    angle = (step / 10.0) + (dist * 1.5)
    s = math.sin(angle)
    c = math.cos(angle)

    xs = x * c - y * s
    ys = x * s + y * c

    r = abs(xs + ys)
    r = r * 64.0
```

(continues on next page)

(continued from previous page)

```
    r -= 20

    return (r, r + (s * 130), r + (c * 130))


# roto-zooming checker board


def checker(x, y, step):
    x -= s_width / 2
    y -= s_height / 2

    angle = step / 10.0
    s = math.sin(angle)
    c = math.cos(angle)

    xs = x * c - y * s
    ys = x * s + y * c

    xs -= math.sin(step / 200.0) * 40.0
    ys -= math.cos(step / 200.0) * 40.0

    scale = step % 20
    scale /= 20
    scale = (math.sin(step / 50.0) / 8.0) + 0.25

    xs *= scale
    ys *= scale

    xo = abs(xs) - int(abs(xs))
    yo = abs(ys) - int(abs(ys))
    val = (
        0
        if (math.floor(xs) + math.floor(ys)) % 2
        else 1
        if xo > 0.1 and yo > 0.1
        else 0.5
    )

    r, g, b = colorsys.hsv_to_rgb((step % 255) / 255.0, 1, val)

    return (r * 255, g * 255, b * 255)


# weeee waaaah


def blues_and_twos(x, y, step):
    x -= s_width / 2
    y -= s_height / 2

    scale = math.sin(step / 6.0) / 1.5
    r = math.sin((x * scale) / 1.0) + math.cos((y * scale) / 1.0)
    b = math.sin(x * scale / 2.0) + math.cos(y * scale / 2.0)
    g = r - 0.8
    g = 0 if g < 0 else g
```

(continues on next page)

(continued from previous page)

```
    b -= r
    b /= 1.4

    return (r * 255, (b + g) * 255, g * 255)


# rainbow search spotlights


def rainbow_search(x, y, step):
    xs = math.sin((step) / 100.0) * 20.0
    ys = math.cos((step) / 100.0) * 20.0

    scale = ((math.sin(step / 60.0) + 1.0) / 5.0) + 0.2
    r = math.sin((x + xs) * scale) + math.cos((y + xs) * scale)
    g = math.sin((x + xs) * scale) + math.cos((y + ys) * scale)
    b = math.sin((x + ys) * scale) + math.cos((y + ys) * scale)

    return (r * 255, g * 255, b * 255)


# zoom tunnel


def tunnel(x, y, step):
    speed = step / 100.0
    x -= s_width / 2
    y -= s_height / 2

    xo = math.sin(step / 27.0) * 2
    yo = math.cos(step / 18.0) * 2

    x += xo
    y += yo

    if y == 0:
        if x < 0:
            angle = -(math.pi / 2)
        else:
            angle = math.pi / 2
    else:
        angle = math.atan(x / y)

    if y > 0:
        angle += math.pi

    angle /= 2 * math.pi  # convert angle to 0...1 range

    shade = math.sqrt(math.pow(x, 2) + math.pow(y, 2)) / 2.1
    shade = 1 if shade > 1 else shade

    angle += speed
    depth = speed + (math.sqrt(math.pow(x, 2) + math.pow(y, 2)) / 10)

    col1 = colorsys.hsv_to_rgb((step % 255) / 255.0, 1, 0.8)
    col2 = colorsys.hsv_to_rgb((step % 255) / 255.0, 1, 0.3)
```

(continues on next page)

(continued from previous page)

```
    col = col1 if int(abs(angle * 6.0)) % 2 == 0 else col2

    td = 0.3 if int(abs(depth * 3.0)) % 2 == 0 else 0

    col = (col[0] + td, col[1] + td, col[2] + td)

    col = (col[0] * shade, col[1] * shade, col[2] * shade)

    return (col[0] * 255, col[1] * 255, col[2] * 255)


effects = [tunnel, rainbow_search, checker, swirl]

step = 0
while True:
    for i in range(500):
        for y in range(s_height):
            for x in range(s_width):
                r, g, b = effects[0](x, y, step)
                if i > 400:
                    r2, g2, b2 = effects[-1](x, y, step)

                    ratio = (500.00 - i) / 100.0
                    r = r * ratio + r2 * (1.0 - ratio)
                    g = g * ratio + g2 * (1.0 - ratio)
                    b = b * ratio + b2 * (1.0 - ratio)
                r = int(max(0, min(255, r)))
                g = int(max(0, min(255, g)))
                b = int(max(0, min(255, b)))
                ledmatrix.set_pixel(x, y, r, g, b)
        step += 1

        ledmatrix.show()

    effect = effects.pop()
    effects.insert(0, effect)
```

Using the pi-topPULSE's LED matrix: Showing CPU temperature

```
import time

from pitop.pulse import ledmatrix


def getCpuTemperature():
    tempFile = open("/sys/class/thermal/thermal_zone0/temp")
    cpu_temp = tempFile.read()
    tempFile.close()
    return int(int(cpu_temp) / 1000)


OFFSET_LEFT = 0
OFFSET_TOP = 2
```

(continues on next page)

(continued from previous page)

```
# fmt: off
NUMS = [1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1,  # 0
        0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0,  # 1
        1, 1, 1, 0, 0, 1, 0, 1, 0, 1, 0, 0, 1, 1, 1,  # 2
        1, 1, 1, 0, 0, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1,  # 3
        1, 0, 0, 1, 0, 1, 1, 1, 1, 0, 0, 1, 0, 0, 1,  # 4
        1, 1, 1, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 1,  # 5
        1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 0, 1, 1, 1, 1,  # 6
        1, 1, 1, 0, 0, 1, 0, 1, 0, 1, 0, 0, 1, 0, 0,  # 7
        1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1,  # 8
        1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 0, 1, 0, 0, 1]  # 9
# fmt: on


# Displays a single digit (0-9)
def show_digit(val, xd, yd, r, g, b):
    offset = val * 15
    for p in range(offset, offset + 15):
        xt = p % 3
        yt = (p - offset) // 3
        ledmatrix.set_pixel(xt + xd, 7 - yt - yd, r * NUMS[p], g * NUMS[p], b *␣
↪NUMS[p])
    ledmatrix.show()


# Displays a two-digits positive number (0-99)
def show_number(val, r, g, b):
    abs_val = abs(val)
    tens = abs_val // 10
    units = abs_val % 10
    if abs_val > 9:
        show_digit(tens, OFFSET_LEFT, OFFSET_TOP, r, g, b)
    show_digit(units, OFFSET_LEFT + 4, OFFSET_TOP, r, g, b)


#############################################################
# MAIN
#############################################################

ledmatrix.rotation(0)
ledmatrix.clear()

lastTemperature = -1

try:
    while True:
        temperature = getCpuTemperature()
        if temperature != lastTemperature:
            if temperature < 60:
                show_number(temperature, 0, 255, 0)
            elif temperature < 70:
                show_number(temperature, 255, 255, 0)
            else:
                show_number(temperature, 255, 0, 0)
            lastemperature = temperature
        time.sleep(2)
```

(continues on next page)

(continued from previous page)

```
except KeyboardInterrupt:
    ledmatrix.clear()
    ledmatrix.show()
```

Using the pi-topPULSE's LED matrix: Showing CPU usage

```
import time

from pitop.pulse import ledmatrix

last_work = [0, 0, 0, 0]
last_idle = [0, 0, 0, 0]


def get_cpu_rates():
    global last_work, last_idle
    rate = [0, 0, 0, 0]
    f = open("/proc/stat", "r")
    line = ""
    for i in range(0, 4):
        while not "cpu" + str(i) in line:
            line = f.readline()
        # print(line)
        splitline = line.split()
        work = int(splitline[1]) + int(splitline[2]) + int(splitline[3])
        idle = int(splitline[4])
        diff_work = work - last_work[i]
        diff_idle = idle - last_idle[i]
        rate[i] = float(diff_work) / float(diff_idle + diff_work)
        last_work[i] = work
        last_idle[i] = idle
    f.close()
    return rate


ledmatrix.rotation(0)

try:
    while True:
        rate = get_cpu_rates()
        ledmatrix.clear()
        for i in range(0, 4):
            level = int(6.99 * rate[i])
            if level < 4:
                r = 0
                g = 255
                b = 0
            elif level < 6:
                r = 255
                g = 255
                b = 6
            else:
                r = 255
                g = 0
                b = 0
```

(continues on next page)

(continued from previous page)

```
            for y in range(0, level + 1):
                ledmatrix.set_pixel(2 * i, y, r, g, b)

        ledmatrix.show()
        time.sleep(1)

except KeyboardInterrupt:
    ledmatrix.clear()
    ledmatrix.show()
```

Module Reference: pi-topPULSE Configuration

pitop.pulse.configuration.**disable_device**()

pitop.pulse.configuration.**eeprom_enabled**()
Get whether the eeprom is enabled.

pitop.pulse.configuration.**enable_device**()

pitop.pulse.configuration.**mcu_enabled**()
Get whether the onboard MCU is enabled.

pitop.pulse.configuration.**microphone_sample_rate_is_16khz**()
Get whether the microphone is set to record at a sample rate of 16,000Hz.

pitop.pulse.configuration.**microphone_sample_rate_is_22khz**()
Get whether the microphone is set to record at a sample rate of 22,050Hz.

pitop.pulse.configuration.**reset_device_state**(*enable*)
reset_device_state: Deprecated

pitop.pulse.configuration.**set_microphone_sample_rate_to_16khz**()
Set the appropriate I2C bits to enable 16,000Hz recording on the microphone.

pitop.pulse.configuration.**set_microphone_sample_rate_to_22khz**()
Set the appropriate I2C bits to enable 22,050Hz recording on the microphone.

pitop.pulse.configuration.**speaker_enabled**()
Get whether the speaker is enabled.

Module Reference: pi-topPULSE LED Matrix

pitop.pulse.ledmatrix.**brightness**(*new_brightness*)
Set the display brightness between 0.0 and 1.0.

Parameters **new_brightness** – Brightness from 0.0 to 1.0 (default 1.0)

pitop.pulse.ledmatrix.**clear**()
Clear the buffer.

pitop.pulse.ledmatrix.**flip_h**()
Flips the grid horizontally.

pitop.pulse.ledmatrix.**flip_v**()
Flips the grid vertically.

pitop.pulse.ledmatrix.**get_brightness**()
Get the display brightness value.

Returns a float between 0.0 and 1.0.

pitop.pulse.ledmatrix.**get_pixel**(*x*, *y*)
Get the RGB value of a single pixel.

Parameters

- **x** – Horizontal position from 0 to 7
- **y** – Veritcal position from 0 to 7

pitop.pulse.ledmatrix.**get_shape**()
Returns the shape (width, height) of the display.

pitop.pulse.ledmatrix.**off**()
Clear the buffer and immediately update pi-topPULSE.

pitop.pulse.ledmatrix.**rotation**(*new_rotation=0*)
Set the display rotation.

Parameters **new_rotation** – Specify the rotation in degrees: 0, 90, 180 or 270

pitop.pulse.ledmatrix.**run_tests**()
Runs a series of tests to check the LED board is working as expected.

pitop.pulse.ledmatrix.**set_all**(*r*, *g*, *b*)
Set all pixels to a specific color.

pitop.pulse.ledmatrix.**set_debug_print_state**(*debug_enable*)
Enable/disable debug prints.

pitop.pulse.ledmatrix.**set_pixel**(*x*, *y*, *r*, *g*, *b*)
Set a single pixel to RGB color.

Parameters

- **x** – Horizontal position from 0 to 7
- **y** – Veritcal position from 0 to 7
- **r** – Amount of red from 0 to 255
- **g** – Amount of green from 0 to 255
- **b** – Amount of blue from 0 to 255

pitop.pulse.ledmatrix.**show**()
Update pi-topPULSE with the contents of the display buffer.

pitop.pulse.ledmatrix.**start**(*new_update_rate=0.1*)
Starts a timer to automatically refresh the LEDs.

pitop.pulse.ledmatrix.**stop**()
Stops the timer that automatically refreshes the LEDs.

Module Reference: pi-topPULSE Microphone

pitop.pulse.microphone.**is_recording**()
Returns recording state of the pi-topPULSE microphone.

pitop.pulse.microphone.**record**()
Start recording on the pi-topPULSE microphone.

pitop.pulse.microphone.**save**(*file_path*, *overwrite=False*)
Saves recorded audio to a file.

pitop.pulse.microphone.**set_bit_rate_to_signed_16**()
Set bitrate to double that of device default by scaling the signal.

pitop.pulse.microphone.**set_bit_rate_to_unsigned_8**()
Set bitrate to device default.

pitop.pulse.microphone.**set_sample_rate_to_16khz**()
Set the appropriate I2C bits to enable 16,000Hz recording on the microphone.

pitop.pulse.microphone.**set_sample_rate_to_22khz**()
Set the appropriate I2C bits to enable 22,050Hz recording on the microphone.

pitop.pulse.microphone.**stop**()
Stops recording audio.

Advanced: EEPROM

The pi-topPULSE contains an EEPROM which was programmed using this settings file. during factory production.

See the Raspberry Pi Foundation's HAT Github repository for more information.

3.8 API - System Peripheral Devices

The pi-top Python SDK provides classes which represent devices, including some that can be used by generic devices, such as USB cameras. These classes are intended to simplify using these common system peripheral devices.

3.8.1 USB Camera

This class provides an easy way to:

- save image and video files
- directly access camera frames
- process frames in the background (via callback)

It is easy to make use of some pre-written video processors, such as motion detection.

It is also possible to make use of this class to read frames from a directory of images, removing the need for a stream of images from physical hardware. This can be useful for testing, or simulating a real camera.

```
from time import sleep

from pitop import Camera

# Record a 10s video to ~/Camera/

cam = Camera()

cam.start_video_capture()
sleep(10)
cam.stop_video_capture()
```

By default, camera frames are of `PIL.Image.Image` type (using the Pillow module), which provides a standardized way of working with the image. These Image objects use raw, RGB-ordered pixels.

It is also possible to use OpenCV standard format, if desired. This may be useful if you are intending to do your own image processing with OpenCV. The OpenCV format uses raw, BGR-ordered pixels in a NumPy `numpy.ndarray` object. This can be done by setting the camera's format property to "OpenCV":

```
from pitop import Camera

c = Camera()
c.format = "OpenCV"
```

This can be also be done by passing the format to the camera's constructor:

```
from pitop import Camera

c = Camera(format="OpenCV")
```

Using a USB Camera to Access Image Data

```
from pitop import Camera

cam = Camera()

while True:
    image = cam.get_frame()
    print(image.getpixel((0, 0)))
```

Using a USB Camera to Capture Video

```
from time import sleep

from pitop import Camera

# Record a 10s video to ~/Camera/

cam = Camera()

cam.start_video_capture()
sleep(10)
cam.stop_video_capture()
```

Adding Motion Detection to a USB Camera

```
from datetime import datetime
from time import localtime, sleep, strftime

from pitop import Camera

# Example code for Camera
# Records videos of any motion captured by the camera

cam = Camera()

last_motion_detected = None
```

(continues on next page)

(continued from previous page)

```
def motion_detected():
    global last_motion_detected

    last_motion_detected = datetime.now().timestamp()

    if cam.is_recording() is False:
        print("Motion detected! Starting recording...")
        output_file_name = f"/home/pi/Desktop/My Motion Recording {strftime('%Y-%m-%d
↪%H:%M:%S', localtime(last_motion_detected))}.avi"
        cam.start_video_capture(output_file_name=output_file_name)

        while (datetime.now().timestamp() - last_motion_detected) < 3:
            sleep(1)

        cam.stop_video_capture()
        print(f"Recording completed - saved to {output_file_name}")


print("Motion detector starting...")
cam.start_detecting_motion(
    callback_on_motion=motion_detected, moving_object_minimum_area=350
)

sleep(60)

cam.stop_detecting_motion()
print("Motion detector stopped")
```

Processing Camera Frame

```
from PIL import ImageDraw

from pitop import Camera

cam = Camera()


def draw_red_cross_over_image(im):
    # Use Pillow to draw a red cross over the image
    draw = ImageDraw.Draw(im)
    draw.line((0, 0) + im.size, fill=128, width=5)
    draw.line((0, im.size[1], im.size[0], 0), fill=128, width=5)
    return im


im = draw_red_cross_over_image(cam.get_frame())
im.show()
```

Processing Camera Frame Stream with OpenCV (Convert to grayscale)

```
from time import sleep

import cv2

from pitop import Camera

cam = Camera(format="OpenCV")


def show_gray_image(image):
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    cv2.imshow("frame", gray)
    cv2.waitKey(1)  # Necessary to show image


# Use callback function for 60s
cam.on_frame = show_gray_image
sleep(60)


# Use get_frame indefinitely
try:
    while True:
        show_gray_image(cam.get_frame())

except KeyboardInterrupt:
    cv2.destroyAllWindows()
```

Ball Color Detection with OpenCV

```
from signal import pause

import cv2

from pitop.camera import Camera
from pitop.processing.algorithms import BallDetector


def process_frame(frame):
    detected_balls = ball_detector(frame, color=["red", "green", "blue"])

    red_ball = detected_balls.red
    if red_ball.found:
        print(f"Red ball center: {red_ball.center}")
        print(f"Red ball radius: {red_ball.radius}")
        print(f"Red ball angle: {red_ball.angle}")
        print()

    green_ball = detected_balls.green
    if green_ball.found:
        print(f"Green ball center: {green_ball.center}")
        print(f"Green ball radius: {green_ball.radius}")
        print(f"Green ball angle: {green_ball.angle}")
        print()
```

(continues on next page)

(continued from previous page)

```
    blue_ball = detected_balls.blue
    if blue_ball.found:
        print(f"Blue ball center: {blue_ball.center}")
        print(f"Blue ball radius: {blue_ball.radius}")
        print(f"Blue ball angle: {blue_ball.angle}")
        print()

    cv2.imshow("Image", detected_balls.robot_view)
    cv2.waitKey(1)


ball_detector = BallDetector()
camera = Camera(resolution=(640, 480))
camera.on_frame = process_frame

pause()
```

Class Reference: USB Camera

class pitop.camera.**Camera**(*index=None*, *resolution=(640, 480)*, *camera_type=<CameraTypes.USB_CAMERA: 0>*, *path_to_images=''*, *format='PIL'*, *flip_top_bottom: bool = False*, *flip_left_right: bool = False*, *rotate_angle=0*, *name='camera'*)

Provides a variety of high-level functionality for using the PMA USB Camera, including capturing images and video, and processing image data from the camera.

Parameters **index** (*int*) – ID of the video capturing device to open. Passing *None* will cause the backend to autodetect the available video capture devices and attempt to use them.

capture_image(*output_file_name=''*)

Capture a single frame image to file.

Note: If no `output_file_name` argument is provided, images will be stored in *~/Camera*.

Parameters **output_file_name** (*str*) – The filename into which to write the image.

current_frame(*format=None*)

Returns the latest frame captured by the camera. This method is non- blocking and can return the same frame multiple times.

By default the returned image is formatted as a `PIL.Image.Image`.

Parameters **format** (*string*) – DEPRECATED. Set 'camera.format' directly, and call this function directly instead.

format

classmethod from_file_system(*path_to_images: str*)

Alternative classmethod to create an instance of a *Camera* object using a `FileSystemCamera`

classmethod from_usb(*index=None*)

Alternative classmethod to create an instance of a *Camera* object using a `UsbCamera`

get_frame(*format=None*)

Returns the next frame captured by the camera. This method blocks until a new frame is available.

Parameters **format** (*string*) – DEPRECATED. Set 'camera.format' directly, and call this function directly instead.

is_detecting_motion()
Returns True if motion detection mode is enabled.

is_recording()
Returns True if recording mode is enabled.

own_state
Representation of an object state that will be used to determine the current state of an object.

start_detecting_motion(*callback_on_motion*, *moving_object_minimum_area=300*)
Begin processing image data from the camera, attempting to detect motion. When motion is detected, call the function passed in.

> **Warning:** The callback function can take either no arguments or only one, which will be used to provide the image back to the user when motion is detected. If a callback with another signature is received, the method will raise an exception.

Parameters

- **callback_on_motion** (*function*) – A callback function that will be called when motion is detected.
- **moving_object_minimum_area** (*int*) – The sensitivity of the motion detection, measured as the area of pixels changing between frames that constitutes motion.

start_handling_frames(*callback_on_frame*, *frame_interval=1*, *format=None*)
Begin calling the passed callback with each new frame, allowing for custom processing.

> **Warning:** The callback function can take either no arguments or only one, which will be used to provide the image back to the user. If a callback with another signature is received, the method will raise an exception.

Parameters

- **callback_on_frame** (*function*) – A callback function that will be called every `frame_interval` camera frames.
- **frame_interval** (*int*) – The callback will run every frame_interval frames, decreasing the frame rate of processing. Defaults to 1.
- **format** (*string*) – DEPRECATED. Set 'camera.format' directly, and call this function directly instead.

start_video_capture(*output_file_name=''*, *fps=20.0*, *resolution=None*)
Begin capturing video from the camera.

> **Note:** If no `output_file_name` argument is provided, video will be stored in *~/Camera*.

Parameters

- **output_file_name** (*str*) – The filename into which to write the video.

- **fps** (*int or float*) – The framerate to use for the captured video. Defaults to 20.0 fps
- **resolution** (*tuple*) – The resolution to use for the captured video. Defaults to (640, 368)

stop_detecting_motion()
Stop running the motion detection processing.

Does nothing unless *start_detecting_motion* has been called.

stop_handling_frames()
Stops handling camera frames.

Does nothing unless *start_handling_frames* has been called.

stop_video_capture()
Stop capturing video from the camera.

Does nothing unless *start_video_capture* has been called.

3.8.2 Keyboard Button

This class makes it easy to handle a keyboard button in the same way as a GPIO-based button.

You can listen for any standard keyboard key input. For example, using `a` or `A` will provide the ability to 'listen' for the A-key being pressed - with or without shift.

Warning: This class depends on pynput, which interfaces with Xorg to handle key press events. This means that this component cannot be used via SSH, or in a headless environment (that is, without a desktop environment).

Note: The DISPLAY environment variable is required to be set in order for this component to work.

Note: If your code is being run from a terminal window, then the key presses will be captured in the terminal output. This can cause confusion and issues around reading output.

```
from time import sleep

from pitop import KeyboardButton


def on_up_pressed():
    print("up pressed")


def on_up_released():
    print("up released")


def on_down_pressed():
    print("down pressed")
```

(continues on next page)

(continued from previous page)

```
def on_down_released():
    print("down released")


def on_left_pressed():
    print("left pressed")


def on_left_released():
    print("left released")


def on_right_pressed():
    print("right pressed")


def on_right_released():
    print("right released")


keyboard_btn_up = KeyboardButton("up")
keyboard_btn_down = KeyboardButton("down")
keyboard_btn_left = KeyboardButton("left")
keyboard_btn_right = KeyboardButton("right")
keyboard_btn_uppercase_z = KeyboardButton("Z")

# Methods will be called when key is pressed:

keyboard_btn_up.when_pressed = on_up_pressed
keyboard_btn_up.when_released = on_up_released
keyboard_btn_down.when_pressed = on_down_pressed
keyboard_btn_down.when_released = on_down_released
keyboard_btn_left.when_pressed = on_left_pressed
keyboard_btn_left.when_released = on_left_released
keyboard_btn_right.when_pressed = on_right_pressed
keyboard_btn_right.when_released = on_right_released

# Or alternatively you can "poll" for key presses:

while True:
    if keyboard_btn_uppercase_z.is_pressed is True:
        print("Z pressed!")

    sleep(0.1)
```

Class Reference: KeyboardButton

class pitop.keyboard.**KeyboardButton**(*key*)

is_pressed
Get or set the button state as a boolean value.

Return type bool

when_pressed

Get or set the 'when pressed' button state callback function. When set, this callback function will be invoked when this event happens.

Parameters **callback** (*Function*) – Callback function to run when a button is pressed.

when_released

Get or set the 'when released' button state callback function. When set, this callback function will be invoked when this event happens.

Parameters **callback** (*Function*) – Callback function to run when a button is released.

Special Key Names

You can listen for the following special keys by passing their names when creating an instance of KeyboardButton.

Identifier	Description
alt	A generic Alt key. This is a modifier.
alt_l	The left Alt key. This is a modifier.
alt_r	The right Alt key. This is a modifier.
alt_gr	The AltGr key. This is a modifier.
backspace	The Backspace key.
caps_lock	The CapsLock key.
cmd	A generic command button.
cmd_l	The left command button. On PC keyboards, this corresponds to the Super key or Windows key, and on Mac key
cmd_r	The right command button. On PC keyboards, this corresponds to the Super key or Windows key, and on Mac ke
ctrl	A generic Ctrl key. This is a modifier.
ctrl_l	The left Ctrl key. This is a modifier.
ctrl_r	The right Ctrl key. This is a modifier.
delete	The Delete key.
down	A down arrow key.
up	An up arrow key.
left	A left arrow key.
right	A right arrow key.
end	The End key.
enter	The Enter or Return key.
esc	The Esc key.
home	The Home key.
page_down	The PageDown key.
page_up	The PageUp key.
shift	A generic Shift key. This is a modifier.
shift_l	The left Shift key. This is a modifier.
shift_r	The right Shift key. This is a modifier.
space	The Space key.
tab	The Tab key.
insert	The Insert key. This may be undefined for some platforms.
menu	The Menu key. This may be undefined for some platforms.
num_lock	The NumLock key. This may be undefined for some platforms.
pause	The Pause/Break key. This may be undefined for some platforms.
print_screen	The PrintScreen key. This may be undefined for some platforms.
scroll_lock	The ScrollLock key. This may be undefined for some platforms.
f1	The F1 key
f2	The F2 key

Table 1 – continued from previous page

Identifier	Description
f3	The F3 key
f4	The F4 key
f5	The F5 key
f6	The F6 key
f7	The F7 key
f8	The F8 key
f9	The F9 key
f10	The F10 key
f11	The F11 key
f12	The F12 key
f13	The F13 key
f14	The F14 key
f15	The F15 key
f16	The F16 key
f17	The F17 key
f18	The F18 key
f19	The F19 key
f20	The F20 key

3.9 Command-Line Tools (CLI)

3.9.1 'pi-top' Command

Utility to interact with pi-top hardware.

```
pi-top [-h] {battery,devices,display,support,imu,oled} ...
```

Where:

-h, --help Show a help message and exits

{battery,devices,display,help,imu,oled}

battery: Get battery information from a pi-top

devices: Get information about device and attached pi-top hardware

display: Communicate and control the device's display

support: Find support resources

imu: Expansion Plate IMU utilities

oled: Quickly display text in pi-top [4]'s miniscreen OLED display

pi-top battery

If the pi-top device has an internal battery, it will report its status.

```
pi-top battery [-h] [-s] [-c] [-t] [-w] [-v]
```

Where:

-h, --help Show a help message and exits

-s, --charging-state Optional. Return the charging state of the battery as an number, where:

- -1: No pi-top battery detected
- 0: Discharging
- 1: Charging
- 2: Full battery

-c, --capacity Optional. Get battery capacity percentage %

-t, --time-remaining Optional. Get the time (in minutes) to full or time to empty based on the charging state

-w, --wattage Optional. Get the wattage (mAh) of the battery

-v, --verbose If no argument is provided, this option will be used by default.

Report all the information available about the battery (charging state, capacity, time remaining and wattage)

Example:

```
pi@pi-top:~ $ pi-top battery
Charging State: 0
Capacity: 42
Time Remaining: 104
Wattage: -41
```

pi-top display

This command provides a way to control different display settings on pi-top devices with a built-in screen.

```
pi-top display [-h] {brightness,backlight,timeout}
```

Where:

-h, --help Show a help message and exits

brightness Control display brightness

backlight Control display backlight

timeout Set the timeout before the screen blanks in seconds (0 to disable)

pi-top display brightness

Request or change the value of the display's brightness.

Note: This only works for the original pi-top, pi-topCEED and pi-top [3]. The pi-top [4] Full HD Touch Display uses hardware buttons to control the brightness, and is not controllable via this SDK.

```
pi-top display brightness [-h] [-v] [-i] [-d]
                          [brightness_value]
```

Where:

-h, --help Show a help message and exits

-v, --verbose Increase verbosity of output

-i, --increment_brightness Increment screen brightness level

-d, --decrement_brightness Decrement screen brightness level

brightness_value Set screen brightness level; [1-10] on pi-top [1] and pi-topCEED, [1-16] for pi-top [3]

Using *pi-top display brightness* without arguments will return the current brightness value.

Note: The *brightness_value* range differs for different devices: for pi-top [3] is from 0-16; pi-top [1] and CEED is 0-10.

Example:

```
pi@pi-top:~ $ pi-top display brightness
16
```

pi-top display backlight

Using *pi-top display backlight* without arguments will return the current backlight status.

```
pi-top display backlight [-h] [-v] [{0,1}]
```

Where:

-h, --help Show a help message and exits

-v, --verbose Increase verbosity of output

{0,1} Set the screen backlight state [0-1]

pi-top display blank_time

Set the time before the screen goes blank on inactivity periods.

Using *pi-top display blank_time* without arguments will return the screen's timeout value.

```
pi-top display timeout [-h] [-v] [timeout_value]
```

Where:

-h, --help Show a help message and exits

-v, --verbose Increase verbosity of output

timeout_value Timeout value in seconds. Set to 0 to disable.

pi-top devices

Finds useful information about the system and the attached devices that are being managed by *pi-topd*.

Running *pi-top devices* on its own will report back the current brightness value.

```
pi-top devices [-h] [--quiet] [--name-only] {hub,peripherals}
```

Where:

-h, --help Show a help message and exits

--quiet, -q Display only the connected devices

--name-only, -n Display only the name of the devices, without further information

hub Get the name of the active pi-top device

peripherals Get information about attached pi-top peripherals

Example:

pi@pi-top:~ $ pi-top devices HUB ==========
pi-top [4] (v5.4) PERIPHERALS ========== [✓] pi-top [4] Expansion Plate (v21.5) [ ] pi-top Touchscreen [ ] pi-top Keyboard [ ] pi-topPULSE [ ] pi-topSPEAKER (v1) - Left channel [ ] pi-topSPEAKER (v1) - Right channel [ ] pi-topSPEAKER (v1) - Mono [ ] pi-topSPEAKER (v2)

pi@pi-top:~ $ pt devices peripherals [✓] pi-top [4] Expansion Plate (v21.5) [ ] pi-top Touchscreen [ ] pi-top Keyboard [ ] pi-topPULSE [ ] pi-topSPEAKER (v1) - Left channel [ ] pi-topSPEAKER (v1) - Right channel [ ] pi-topSPEAKER (v1) - Mono [ ] pi-topSPEAKER (v2)

pi@pi-top:~ $ pt devices hub –name-only pi-top [4]

pi-top imu

Utility to calibrate the IMU included in the Expansion Plate.

```
pi-top imu calibrate [-h] [-p PATH]
```

Where:

-h, --help Show a help message and exits

-p PATH, --path PATH Directory for storing calibration graph data

Example:

```
pi-top imu calibrate --path /tmp
```

pi-top oled

Configure and display text/images directly onto pi-top [4]'s miniscreen OLED display.

```
pi-top oled [-h] {display,spi}
```

Where:

-h, --help Show a help message and exits

display Display text and images into the OLED

spi Control the SPI bus used by OLED

pi-top oled display

Display text and images directly onto pi-top [4]'s miniscreen OLED display.

```
pi-top oled display [-h] [--timeout TIMEOUT] [--font-size FONT_SIZE] text
```

Where:

-h, --help Show a help message and exits

-t, --timeout TIMEOUT set the timeout in seconds

--font-size FONT_SIZE set the font size

text set the text to write to screen

Example:

```
pi@pi-top:~ $ pi-top oled display "hey!" -t 5
```

pi-top oled spi

Control the SPI bus used by the OLED. When using *pi-top oled spi* without arguments, the SPI bus currently used by the OLED will be returned.

```
pi-top oled spi [-h] {0,1}
```

Where:

-h, --help Show a help message and exits

{0,1} Optional. Set the SPI bus to be used by OLED. Valid options: 0 or 1

Example:

```
pi@pi-top:~ $ pi-top oled spi
1

pi@pi-top:~ $ pi-top oled spi 0

pi@pi-top:~ $ pi-top oled spi
0
```

pi-top support

Find information about support topics for your device.

```
pi-top support [-h] {links,health_check} ...
```

Where:

-h, --help Show a help message and exits

{links,health_check} Subcommands, please refer to the next sections.

pi-top support links

Find resources to learn how to use your device and get help if needed.

```
pi-top support links [-h] {docs,help}
```

Where:

-h, --help Show a help message and exits

{docs,help} docs: Print links to pi-top documentation

help: Print links to places where to look for help

Example:

```
$ pi-top support links docs
 ================================================
 DOCS
 ================================================
 [ ✓ ] pi-top Python SDK documentation: online version, recommended
   https://docs.pi-top.com/python-sdk/
 [ ✓ ] pi-top Python SDK documentation: offline version
   /usr/share/doc/python3-pitop/html/index.html

 pi@pi-top:~ $ pi-top support links
 ================================================
 DOCS
 ================================================
 [ ✓ ] pi-top Python SDK documentation: online version, recommended
   https://docs.pi-top.com/python-sdk/
 [ ✓ ] pi-top Python SDK documentation: offline version
   /usr/share/doc/python3-pitop/html/index.html
 ================================================
 OTHER
 ================================================
 [ ✓ ] Knowledge Base: Find answers to commonly asked questions
   https://knowledgebase.pi-top.com/
 [ ✓ ] Forum: Discuss and search through support topics.
   https://forum.pi-top.com/
```

pi-top support health_check

Perform a system wide check to help troubleshooting any problems with pi-top software and hardware.

```
pi-top support health_check
```

3.10 Labs - Experimental APIs

Note: The pi-top Python SDK Labs are a set of classes which are being provided as experiments in exciting new ways to interact with your device.

Warning: Everything in Labs is subject to change - so use at your own risk!

3.10.1 Web

This Web API has been created with the goal of giving users the ability to easily create a web application that runs directly on the pi-top that can easily offer a dynamic, interactive interface for controlling the pi-top.

The Web API provides a selection of *web server interfaces*, as well as a selection of prebuilt features known as *Blueprints* to be used with these servers.

For examples of how to use this, check out the labs examples directory on GitHub.

Servers

For simple static web apps or ground-up customisation, use *WebServer*.

If you would like a 'batteries included' WebServer that makes it easy to interact with your pi-top, use *WebController*.

For a quick way to control your pi-top [4] Robotics Kit, use *RoverWebController*, which offers a preconfigured but customisable WebController for rover-style robots.

WebServer

The WebServer class is used to create a zero-config server that can:

- serve static files and templates
- handle requests
- handle WebSocket connections

WebServer is a preconfigured gevent WSGIServer, due to this it can be started and stopped just like a gevent BaseServer:

```
from pitop.labs import WebServer

server = WebServer()

# start server in the background
server.start()

# stop server that has been started in the background
server.stop()

# start server and wait until interrupted
server.serve_forever()
```

WebServer serves static files and templates found in the working directory automatically. The entrypoint file is always `index.html`. All html files found are considered to be Jinja templates, this means that if you have a file `layout.html` in the same directory as your WebServer:

```
<html>
  <head>
      <title>My Web App</title>
  </head>
  <body>
    {% block body %}
    {% endblock %}
  </body>
</html>
```

It is possible to use it as template for other html files. For example `index.html` can extend `layout.html`:

```
{% extends 'layout.html' %}

{% block body %}
  <h1>My Custom Body</h1>
{% endblock %}
```

To add routes you can use the underlying Flask app's route decorator:

```
from pitop.labs import WebServer

server = WebServer()

@server.app.route('/ping')
def ping():
    return 'pong'

server.serve_forever()
```

WebSocket routes can be added by using the route decorator provided by Flask Sockets:

```
from pitop.labs import WebServer

server = WebServer()

@server.sockets.route('/ws')
def ws(socket):
    while not socket.closed:
        message = socket.receive()
        socket.send(message)

server.serve_forever()
```

The server `port` defaults to 8070 but can be customised:

```
from pitop.labs import WebServer

server = WebServer(port=8071)
```

It is also possible to customise the Flask app by passing your own into the `app` keyword argument:

```
from pitop.labs import WebServer
from flask import Flask

server = WebServer(app=Flask(__name__))
```

WebServer is fully compatible with Flask blueprints, which can be passed to the `blueprints` keyword argument:

```
from pitop.labs import WebServer
from flask import Blueprint

WebServer(blueprints=[
    Blueprint('custom', __name__)
])
```

We provide a number of premade blueprints:

- *BaseBlueprint*

- *WebComponentsBlueprint*
- *MessagingBlueprint*
- *VideoBlueprint*

By default WebServer uses the *BaseBlueprint*

Warning: When using WebServer in a multithreaded project you must use gevent threading. This is because using Python standard library threading while using a gevent server can result in unexpected behaviour, or may not work at all. See the dashboard example for a basic idea of how gevent threading can be used.

WebController

The WebController class is subclass of *WebServer* that uses the *ControllerBlueprint*. It exists as a convenience class so that blueprints are not required to be able to build simple web controllers.

```
from pitop import Camera
from pitop.labs import WebController

camera = Camera()

def on_dinner_change(data):
    print(f'dinner is now {data}')

server = WebController(
  get_frame=camera.get_frame,
  message_handlers={'dinner_changed': on_dinner_change}
)

server.serve_forever()
```

See the *ControllerBlueprint* reference for more detail.

RoverWebController

The RoverWebController class is subclass of *WebServer* that uses the *RoverControllerBlueprint*. It exists as a convenience class so that blueprints are not required to build simple rover web controllers.

```
from pitop import Pitop, Camera, DriveController, PanTiltController
from pitop.labs import RoverWebController

rover = Pitop()
rover.add_component(Camera())
rover.add_component(DriveController())
rover.add_component(PanTiltController())

server = RoverWebController(
  get_frame=rover.camera.get_frame,
  drive=rover.drive,
  pan_tilt=rover.pan_tilt
)

server.serve_forever()
```

See the *RoverControllerBlueprint* reference for more detail.

Blueprints

BaseBlueprint

BaseBlueprint provides a layout and styles that are the base of the templates found in other blueprints. It adds a `base.html` template which has the following structure:

```
<html>
  <head>
    <title>{% block title %}{% endblock %}</title>
    {% block head %}
      <link rel="stylesheet" href="/base/index.css"></link>
    {% endblock %}
  </head>

  <body>
    {% block body %}
      <header> {% block header %}{% endblock %} </header>
      <main> {% block main %}{% endblock %} </main>
      <footer> {% block footer %}{% endblock %} </footer>
    {% endblock %}
  </body>
</html>
```

The `base.html` adds some basic styles and variables to the page by linking the `index.css` static file.

```
:root {
  --background-color: #00B2A2
}

body {
  background-color: var(--background-color);
  margin: 0;
  padding: 0;
}
```

Adding the BaseBlueprint to a WebServer is done as follows:

```
from pitop.labs import WebServer, BaseBlueprint

server = WebServer(blueprints=[
    BaseBlueprint()
])

server.serve_forever()
```

Note: WebServer uses BaseBlueprint by default, so the above is only necessary if you are using BaseBlueprint with other blueprints.

Then you are able to extend the `base.html` in your other html files:

```
{% extends 'base.html' %}

{% block title %}Custom Page{% endblock %}
```

(continues on next page)

(continued from previous page)

```
{% block head %}
  <!-- call super() to add index.css -->
  {{ super() }}
  <link rel="styles" href="custom-styles.css"></link>
{% endblock %}

{% block header %}
  <img src="logo.png"></img>
{% endblock %}

{% block main %}
  <section>Section One</section>
  <section>Section Two</section>
{% endblock %}

{% block footer %}
  Contact Info: 123456789
{% endblock %}
```

If you want to use the static files provided without extending the `base.html` template you can do so by adding them to the page yourself:

```
<html>
  <head>
    <link rel="stylesheet" href="/base/index.css"></link>
  </head>
  <body>
  </body>
</html>
```

WebComponentsBlueprint

WebComponentsBlueprint provides a set of Web Components for adding complex elements to the page.

Adding the WebComponentsBlueprint to a WebServer is done as follows:

```
from pitop.labs import WebServer, WebComponentsBlueprint

server = WebServer(blueprints=[
    WebComponentsBlueprint()
])

server.serve_forever()
```

To add the components to the page WebComponentsBlueprint provides a setup template `setup-components.html` that can be included in the `head` of your page

```
<head>
  {% include "setup-webcomponents.html" %}
</head>
```

Currently the only component included is the `joystick-component`, which acts a wrapper around nippleJS.

```
<joystick-component
  mode="static"
  size="200"
  position="relative"
  positionTop="100"
  positionLeft="100"
  positionRight=""
  positionBottom=""
  onmove="console.log(data)"
  onend="console.log(data)"
></joystick-component>
```

To add the joystick-component to the page without using templates you can add it to the page by adding the `nipplejs.min.js` and `joystick-component.js` scripts to the `head` of your page:

```
<head>
  <script type="text/javascript" src="/webcomponents/vendor/nipplejs.min.js"></script>
  <script type="text/javascript" src="/webcomponents/joystick-component.js"></script>
</head>
```

MessagingBlueprint

MessagingBlueprint is used to communicate between your python code and the page.

Adding the MessagingBlueprint to a WebServer is done as follows:

```
from pitop.labs import WebServer, MessagingBlueprint

server = WebServer(blueprints=[
    MessagingBlueprint()
])

server.serve_forever()
```

To add messaging to the page MessagingBlueprint provides a setup template `setup-messaging.html` that can be included in the `head` of your page:

```
<head>
  {% include "setup-messaging.html" %}
</head>
```

This adds a JavaScript function `publish` to the page, which you can use to send JavaScript Objects to your WebServer. The messages must have a `type`, and can optionally have some `data`.

```
<select
  id="dinner-select"
  onchange="publish({ type: 'dinner_changed', data: this.value })"
>
  <option value="tacos">Tacos</option>
  <option value="spaghetti">Spaghetti</option>
</select>
```

To receive the messages sent by `publish` you can pass a `message_handlers` dictionary to MessagingBlueprint. The keys of `message_handlers` correspond to the `type` of the message and the value must be a function that handles the message, a 'message handler'. The message handler is passed the message's `data` value as it's first argument.

```
from pitop.labs import WebServer, MessagingBlueprint

def on_dinner_change(data):
    print(f'dinner is now {data}')

messaging = MessagingBlueprint(message_handlers={
    'dinner_changed': on_dinner_change
})

server = WebServer(blueprints=[messaging])
server.serve_forever()
```

The second argument of a message handler is a `send` function which can send a message back to the page:

```
def on_dinner_change(data, send):
    print(f'dinner is now {data}')
    send({ 'type': 'dinner_received' })
```

To receive messages sent from a message handler the MessagingBlueprint also adds a JavaScript function `subscribe` to the page:

```
<script>
  subscribe((message) => {
    if (message.type === 'dinner_received') {
      console.log('Dinner Received!')
    }
  })
</script>
```

Another way of sending messages to the page is to use the MessagingBlueprint's `broadcast` method:

```
from pitop import Button
from pitop.labs import WebServer, MessagingBlueprint

button = Button('D1')

def on_dinner_change(data):
    print(f'dinner is now {data}')

messaging = MessagingBlueprint(message_handlers={
    'dinner_changed': on_dinner_change
})

def reset():
  messaging.broadcast({ 'type': 'reset' })

button.on_press = reset

server = WebServer(blueprints=[messaging])
server.serve_forever()
```

This is received by the same subscribe function as before:

```
<script>
  subscribe((message) => {
    if (message.type === 'reset') {
      console.log('Reset')
```

(continues on next page)

(continued from previous page)

```
    }
  })
</script>
```

There is one difference between `broadcast` and `send`: `broadcast` sends the message to every client whereas `send` only responds to the client that sent the message being handled.

VideoBlueprint

VideoBlueprint adds the ability to add a video feed from your python code to the page.

Adding the VideoBlueprint to a WebServer is done as follows:

```
from pitop import Camera
from pitop.labs import WebServer, VideoBlueprint

camera = Camera()

server = WebServer(blueprints=[
    VideoBlueprint(get_frame=camera.get_frame)
])

server.serve_forever()
```

To add video styles to the page VideoBlueprint provides a setup template `setup-video.html` that can be included in the `head` of your page:

```
<head>
  {% include "setup-video.html" %}
</head>
```

This adds a set of classes that can be used to style your video:

```
.background-video {
  height: 100vh;
  position: fixed;
  top: 0;
  left: 50%;
  transform: translateX(-50%);
  z-index: -1;
}
```

In order to render the video on the page you must use an `img` tag with the `src` attribute of `video.mjpg`:

```
<body>
  <img src="video.mjpg" class="background-video"></img>
</body>
```

It is also possible to add multiple VideoBlueprints to a WebServer:

```
from pitop import Camera
from pitop.labs import WebServer, VideoBlueprint

camera_one = Camera(index=0)
camera_two = Camera(index=1)
```

(continues on next page)

(continued from previous page)

```
server = WebServer(blueprints=[
    VideoBlueprint(name="video-one", get_frame=camera_one.get_frame),
    VideoBlueprint(name="video-two", get_frame=camera_two.get_frame)
])

server.serve_forever()
```

This makes it possible to to add multiple video feeds to the page, where the `src` attribute uses the name of the VideoBlueprint with a `.mjpg` extension:

```
<body>
  <img src="video-one.mjpg"></img>
  <img src="video-two.mjpg"></img>
</body>
```

If you want to use the static files on your page without using templates you can do so by adding them to the page yourself:

```
<head>
  <link rel="stylesheet" href="/video/styles.css"></link>
</head>
```

ControllerBlueprint

ControllerBlueprint combines blueprints that are useful in creating web apps that interact with your pi-top. The blueprints it combines are the *BaseBlueprint*, *WebComponentsBlueprint*, *MessagingBlueprint* and *VideoBlueprint*.

```
from pitop import Camera
from pitop.labs import WebServer, ControllerBlueprint

camera = Camera()

def on_dinner_change(data):
    print(f'dinner is now {data}')

server = WebServer(blueprints=[
    ControllerBlueprint(
      get_frame=camera.get_frame,
      message_handlers={'dinner_changed': on_dinner_change}
    )
])

server.serve_forever()
```

To simplify setup ControllerBlueprint provides a `base-controller.html` template which includes all the setup snippets for it's children blueprints:

```
{% extends "base.html" %}

{% block title %}
  Web Controller
{% endblock %}
```

(continues on next page)

(continued from previous page)

```
{% block head %}
  {{ super() }}
  {% include "setup-video.html" %}
  {% include "setup-messaging.html" %}
  {% include "setup-webcomponents.html" %}
{% endblock %}
```

`base-controller.html` extends `base.html`, this means you can use blocks defined in `base.html` when extending `base-controller.html`:

```
{% extends "base-controller.html" %}

{% block title %}My WebController{% endblock %}

{% block head %}
  <!-- call super() to setup blueprints -->
  {{ super() }}
  <link rel="stylesheet" href="custom-styles.css"></link>
{% endblock %}

{% block main %}
  <h1>Video</h1>
  <img src="video.mjpg"></img>
{% endblock %}
```

RoverControllerBlueprint

RoverControllerBlueprint uses the *ControllerBlueprint* to create a premade web controller specifically built for rover projects.

```
from pitop import Pitop, Camera
from pitop.labs import WebServer, RoverControllerBlueprint

rover = Pitop()
rover.add_component(Camera())
rover.add_component(DriveController())
rover.add_component(PanTiltController())

server = WebServer(blueprints=[
    RoverControllerBlueprint(
      get_frame=rover.camera.get_frame,
      drive=rover.drive,
      pan_tilt=rover.pan_tilt
    )
])

server.serve_forever()
```

RoverControllerBlueprint provides a page template `base-rover.html` which has a background video and two joysticks:

By default the right joystick is used to drive the rover around and the left joystick controls the pan tilt mechanism. The `drive` keyword argument is required, but the `pan_tilt` keyword argument is optional; if it is not passed the left joystick is not rendered.

It is possible to customise the page by extending the `base-rover.html` template:

```
{% extends "base-rover.html" %}

{% block title %}My Rover Controller{% endblock %}

{% block main %}
  <!-- call super() to keep video and joysticks -->
  {{ super() }}

  <button onclick="publish({ type: 'clicked' })"></button>
{% endblock %}
```

It is also possible to customise the message handlers used by the RoverControllerBlueprint, for example to swap the joysticks so the left drives the rover and the right controls pan tilt:

```
from pitop import Camera, DriveController, PanTiltController, Pitop
from pitop.labs import RoverWebController
from pitop.labs.web.blueprints.rover import drive_handler, pan_tilt_handler

rover = Pitop()
rover.add_component(DriveController())
rover.add_component(PanTiltController())
rover.add_component(Camera())

rover_controller = RoverWebController(
    get_frame=rover.camera.get_frame,
    message_handlers={
        "left_joystick": lambda data: drive_handler(rover.drive, data),
```

(continues on next page)

(continued from previous page)

```
        "right_joystick": lambda data: pan_tilt_handler(rover.pan_tilt, data),
    },
)

rover_controller.serve_forever()
```

Note that when `left_joystick` or `right_joystick` are in `message_handlers` the `pan_tilt` and `drive` arguments do not need to be passed respectively.

3.11 More Information

3.11.1 Frequently Asked Questions

How does this SDK work?

What is PMA?

I keep getting an Exception - what is the problem?

Where did this SDK come from?

Note: epoch version

I was using an older version of the Python libraries. How can I update to use this SDK?

Check out the Python SDK Migration GitHub repository for more information about this.

You may also find it helpful to check out the examples to see how to use the new components.

I lost my miniscreen menu - where is it?

Check out *Key Concepts: pi-top [4] Miniscreen* for useful information about how this works.

3.11.2 API Changes

This section aims to outline key changes made between versions, to support upgrading.

3.11.3 Contributing

Check out the Contributing to pi-topOS article in the pi-top knowledge base to learn how to contribute.

3.11.4 References

- pi-top's Knowledge Base
- pi-top's Forum
- gpiozero

- imageio
- numpy
- luma
- Pillow

3.11.5 Requirements

The following Debian packages are required for this library to work:

Package Name	Usage
`alsa-utils`	Used for configuring the system audio; such as setting the correct audio card when connecting a pi-topSPEAKER.
`coreutils`	Used to perform basic OS operations and commands; such as `ls` and `chmod`
`fonts-droid-fallback`	Minimum essential font used by the OLED screen.
`i2c-tools`	Communicate with pi-top I2C devices.
`pi-topd`	Allows communication with pi-top's hub; such as getting battery state. This package installs a `systemd` service that needs to be running for this library to work properly
`raspi-config`	Required to communicate and set parameters to the Raspberry Pi.

3.11.6 License

For an alphabetized list of terms used in this SDK with links, check out the genindex.

Python Module Index

p

Index

E

F

G

H

I

K

L

M

O

P

R

S

T

U

V

W

Z