AIY Projects Documentation

Release 2018-11-16

Google LLC

Nov 14, 2019

Common APIs

1 aiy.board 3

2 aiy.leds 5

3 aiy.pins 11

4 Vision Kit overview 13
4.1 Software 13
4.2 Vision Bonnet 13
4.3 Troubleshooting 15

5 aiy.toneplayer 17

6 aiy.trackplayer 19

7 aiy.vision.annotator 25

8 aiy.vision.inference 27

9 aiy.vision.models 31
9.1 aiy.vision.models.dish_classification 31
9.2 aiy.vision.models.dish_detection 32
9.3 aiy.vision.models.face_detection 32
9.4 aiy.vision.models.image_classification 32
9.5 aiy.vision.models.inaturalist_classification 33
9.6 aiy.vision.models.object_detection 33

10 Voice Kit overview 35
10.1 Software 35
10.2 Voice Bonnet (Voice Kit V2) 35
10.3 Voice HAT (Voice Kit V1) 36
10.4 Troubleshooting 37

11 aiy.assistant 39
11.1 aiy.assistant.grpc 40
11.2 aiy.assistant.library 41
11.3 aiy.assistant.auth_helpers 41

12 aiy.cloudspeech **43**

13 aiy.voice.audio **45**
13.1 Recording 45
13.2 Playback 46
13.3 Audio format 48

14 aiy.voice.tts **49**

15 API indices **51**

Python Module Index **53**

Index **55**

This is the Python API reference for the AIY Projects library, which is provided with the Vision Kit and Voice Kit projects.

For more information about the Vision and Voice kits, including assembly guides and makers guides, go to aiyprojects.withgoogle.com.

CHAPTER 1

aiy.board

APIs to control the button (and button LED) that's attached to the Vision Bonnet and Voice Bonnet/HAT's button connector. For example:

```
from aiy.board import Board, Led

def main():
    print('LED is ON while button is pressed (Ctrl-C for exit).')
    with Board() as board:
        while True:
            board.button.wait_for_press()
            print('ON')
            board.led.state = Led.ON
            board.button.wait_for_release()
            print('OFF')
            board.led.state = Led.OFF


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

class aiy.board.**Board**(*button_pin=23*, *led_pin=25*)

Bases: `object`

An interface for the connected AIY board.

button

Returns a *Button* representing the button connected to the button connector.

close()

led

Returns an *Led* representing the LED in the button.

class aiy.board.**Button**(*channel*, *edge='falling'*, *pull_up_down='up'*, *debounce_time=0.08*)

Bases: `object`

An interface for the button connected to the AIY board's button connector.

close()
Internal method to clean up the object when done.

wait_for_press(*timeout=None*)
Pauses the script until the button is pressed or the timeout is reached.

Parameters **timeout** – Seconds to wait before proceeding. By default, this is `None`, which means wait indefinitely.

wait_for_release(*timeout=None*)
Pauses the script until the button is released or the timeout is reached.

Parameters **timeout** – Seconds to wait before proceeding. By default, this is `None`, which means wait indefinitely.

when_pressed
A function to run when the button is pressed.

when_released
A function to run when the button is released.

class aiy.board.**Led**

Controls the LED in the button. Get an instance from `Board.led`.

This class is primarily intended for compatibility with the Voice HAT (V1 Voice Kit), and it also works on the Voice/Vision Bonnet. However, if you're using *only* the Voice/Vision Bonnet, then you should instead use `aiy.leds`, which provides more controls for the button's unique RGB LED.

brightness(*value*)
Sets the button LED brightness

Parameters **value** – The brightness, between 0.0 and 1.0

state
Sets the button LED state. Can be one of the values below.

OFF

ON

BLINK

BLINK_3

BEACON

BEACON_DARK

DECAY

PULSE_SLOW

PULSE_QUICK

CHAPTER 2

aiy.leds

APIs to control the RGB LED in the button that connects to the Vision/Voice Bonnet, and the privacy LED with the Vision Kit.

These APIs are **not compatible** with the Voice HAT (V1 Voice Kit). To control the Voice HAT's button LED, instead use *aiy.board.Led*.

For example, here's how to blink the button's red light:

```
import time
from aiy.leds import Leds, Color

with Leds() as leds:
    for _ in range(4):
        leds.update(Leds.rgb_on(Color.RED))
        time.sleep(1)
        leds.update(Leds.rgb_off())
        time.sleep(1)
```

For more examples, see leds_example.py.

These APIs are only for the RGB LED in the button and the Vision Kit's privacy LED. To control LEDs you've attached to the bonnet's GPIO pins or the LEDs named `LED_1` and `LED_2` on the Vision/Voice Bonnet, instead use *aiy.pins*.

class aiy.leds.**Color**

Bases: `object`

Defines colors as RGB tuples that can be used as color values with *Leds*.

BLACK = (0, 0, 0)

BLUE = (0, 0, 255)

CYAN = (0, 255, 255)

GREEN = (0, 255, 0)

PURPLE = (255, 0, 255)

RED = (255, 0, 0)

WHITE = (255, 255, 255)

YELLOW = (255, 255, 0)

static blend(*color_a*, *color_b*, *alpha*)
Creates a color that is a blend between two colors.

Parameters

- **color_a** – One of two colors to blend.
- **color_b** – One of two colors to blend.
- **alpha** – The alpha blend to apply between `color_a` and `color_b`, from 0.0 to 1.0, respectively. That is, 0.0 makes `color_a` transparent so only `color_b` is visible; 0.5 blends the two colors evenly; 1.0 makes `color_b` transparent so only `color_a` is visible.

Returns An RGB tuple.

class aiy.leds.**Leds**(*reset=True*)
Bases: `object`

Class to control the KTD LED driver chip in the button used with the Vision and Voice Bonnet.

class Channel(*state*, *brightness*)
Bases: `object`

Defines the configuration for each channel in the KTD LED driver.

You should not instantiate this class directly; instead create a dictionary of `Channel` objects with the other methods below, which you can then pass to *`update()`*.

Parameters

- **state** – Either *`ON`*, *`OFF`*, or *`PATTERN`*.
- **brightness** – A value between 0 and 255.

OFF = 0

ON = 1

PATTERN = 2

static installed()
Internal method to verify the `Leds` class is available.

pattern
Defines a blink pattern for the button's LED. Must be set with a *`Pattern`* object. For example:

```
with Leds() as leds:
    leds.pattern = Pattern.blink(500)
    leds.update(Leds.rgb_pattern(Color.RED))
    time.sleep(5)
```

static privacy(*enabled*, *brightness=255*)
Creates a configuration for the privacy LED (channel 4).

You can instead use *`privacy_on()`* and *`privacy_off()`*.

Parameters

- **enabled** – `True` to turn on the light; `False` to turn it off.

- **brightness** – A value from 0 to 255.

Returns A dictionary with one `Channel` for the privacy LED (channel 4).

static privacy_off()
Creates an "off" configuration for the privacy LED (the front LED on the Vision Kit).

Returns A dictionary with one `Channel` for the privacy LED (channel 4).

static privacy_on(*brightness=255*)
Creates an "on" configuration for the privacy LED (the front LED on the Vision Kit).

Parameters **brightness** – A value from 0 to 255.

Returns A dictionary with one `Channel` for the privacy LED (channel 4).

reset()
Resets the LED driver to a clean state.

static rgb(*state*, *rgb*)
Creates a configuration for the RGB channels: 1 (red), 2 (green), 3 (blue).

Generally, you should instead use convenience constructors such as `rgb_on()` and `rgb_pattern()`.

Parameters

- **state** – Either `Channel.ON`, `Channel.OFF`, or `Channel.PATTERN`.
- **rgb** – Either one of the `Color` constants or your own tuple of RGB values.

Returns A dictionary of 3 `Channel` objects, representing red, green, and blue values.

static rgb_off()
Creates an "off" configuration for the button's RGB LED.

Returns A dictionary of 3 `Channel` objects, representing red, green, and blue values, all turned off.

static rgb_on(*rgb*)
Creates an "on" configuration for the button's RGB LED.

Parameters **rgb** – Either one of the `Color` constants or your own tuple of RGB values.

Returns A dictionary of 3 `Channel` objects, representing red, green, and blue values.

static rgb_pattern(*rgb*)
Creates a "pattern" configuration for the button's RGB LED, using the light pattern set with `pattern` and the color set here. For example:

```
with Leds() as leds:
    leds.pattern = Pattern.blink(500)
    leds.update(Leds.rgb_pattern(Color.RED))
    time.sleep(5)
```

Parameters **rgb** – Either one of the `Color` constants or your own tuple of RGB values.

Returns A dictionary of 3 `Channel` objects, representing red, green, and blue values.

update(*channels*)
Changes the state of an LED. Takes a dictionary of LED channel configurations, provided by various methods such as `rgb_on()`, `rgb_off()`, and `rgb_pattern()`.

For example, turn on the red light:

```
with Leds() as leds:
    leds.update(Leds.rgb_on(Color.RED))
    time.sleep(2)
    leds.update(Leds.rgb_off())
```

Or turn on the privacy LED (Vision Kit only):

```
with Leds() as leds:
    leds.update(Leds.privacy_on())
    time.sleep(2)
    leds.update(Leds.privacy_off())
```

Parameters **channels** – A dictionary of one or more *Channel* objects. Use the `rgb_` and `privacy_` methods to create a dictionary.

class aiy.leds.**Pattern**(*period_ms*, *on_percent=0.5*, *rise_ms=0*, *fall_ms=0*)
Bases: `object`

Defines an LED blinking pattern. Pass an instance of this to *Leds.pattern*.

Parameters

- **period_ms** – The period of time (in milliseconds) for each on/off sequence.
- **on_percent** – Percent of time during the period to turn on the LED (the LED turns on at the beginning of the period).
- **rise_ms** – Duration of time to fade the light on.
- **fall_ms** – Duration of time to fade the light off.

The parameters behave as illustrated below.

```
rise_ms /----------\ fall_ms
       /            \
      /  on_percent  \
     #--------------------------------#
                  period_ms
```

static blink(*period_ms*)
Convenience method to create a blinking pattern.

Parameters **period_ms** – The period of time (in milliseconds) for each on/off sequence.

Returns A *Pattern*.

static breathe(*period_ms*)
Convenience method to create a breathing pattern (a blink that fades in and out).

Parameters **period_ms** – The period of time (in milliseconds) for each on/off sequence.

Returns A *Pattern*.

class aiy.leds.**PrivacyLed**(*leds*, *brightness=32*)
Bases: `object`

Helper class to turn Privacy LED off automatically.

When instantiated, the privacy LED turns on. It turns off whenever the code exits the scope in which this was created. For example:

```
# Turn the privacy LED on for 2 seconds
with PrivacyLed(Leds()):
    time.sleep(2)
```

Parameters

- **leds** – An instance of *Leds*.
- **brightness** – A value between 0 and 255.

class aiy.leds.**RgbLeds**(*leds*, *channels*)

Bases: object

Helper class to turn RGB LEDs off automatically.

When instantiated, the privacy LED turns on. It turns off whenever the code exits the scope in which this was created. For example:

```
# Turn on the green LED for 2 seconds
with RgbLeds(Leds(), Leds.rgb_on(Color.GREEN)):
    time.sleep(2)
```

Parameters

- **leds** – An instance of *Leds*.
- **channels** – A dictionary of one or more `Channel` objects. Use the `Leds.rgb_` and `Leds.privacy_` methods to create a dictionary.

CHAPTER 3

aiy.pins

GPIO pin definitions for the Vision Bonnet and Voice Bonnet, for use with gpiozero APIs.

These APIs are **not compatible** with the Voice HAT (V1 Voice Kit).

For example, here's how to create a `gpiozero.Servo` with `PIN_B`:

```
from gpiozero import Servo
from aiy.pins import PIN_B

# Create a servo with the custom values to give the full dynamic range.
tuned_servo = Servo(PIN_B, min_pulse_width=.0005, max_pulse_width=.0019)
```

Or here's how to light up `LED_1` on the bonnet when you press the button:

```
from gpiozero import Button
from gpiozero import LED
from aiy.pins import BUTTON_GPIO_PIN
from aiy.pins import LED_1

# Set up a gpiozero LED using the first onboard LED on the vision hat.
led = LED(LED_1)
# Set up a gpiozero Button using the button included with the vision hat.
button = Button(BUTTON_GPIO_PIN)

while True:
    if button.is_pressed:
        led.on()
    else:
        led.off()
```

For more examples, see src/examples/gpiozero/.

`aiy.pins.`**`PIN_A`**

`aiy.pins.`**`PIN_B`**

`aiy.pins.`**`PIN_C`**

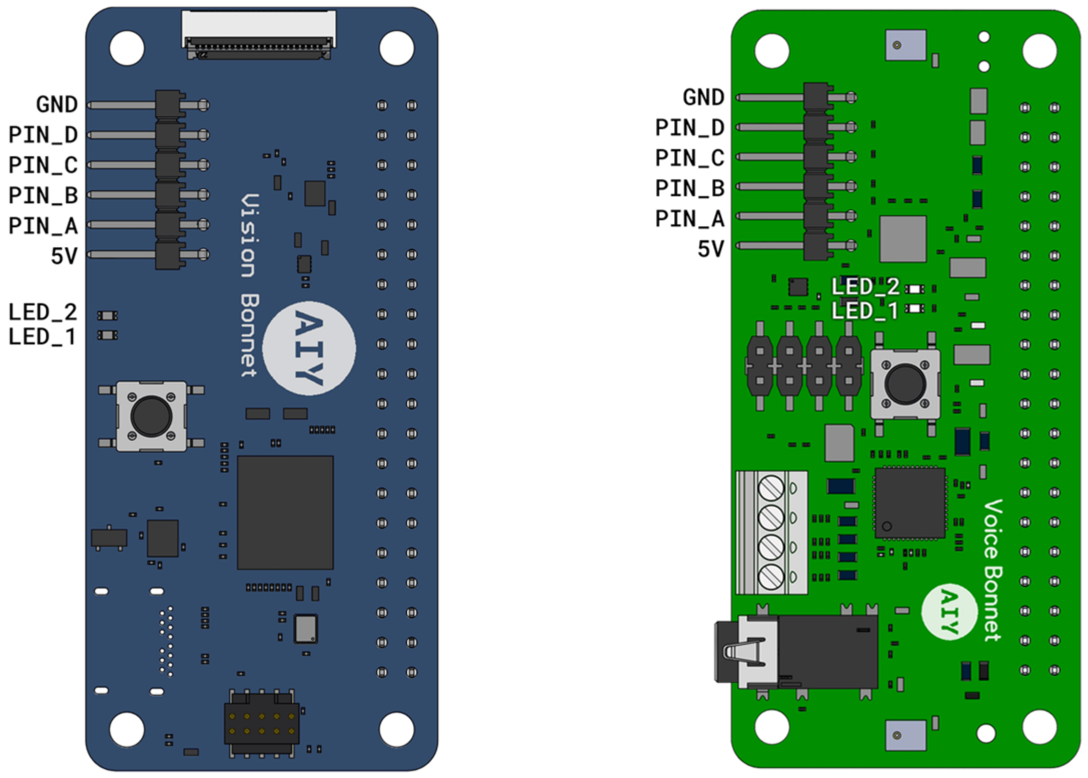

Fig. 1: **Figure 1.** Pin and LED positions on the Vision and Voice Bonnet.

`aiy.pins.`**`PIN_D`**

`aiy.pins.`**`LED_1`**

Use this with `gpiozero.LED` to control LED_1 on the Vision/Voice Bonnet.

`aiy.pins.`**`LED_2`**

Use this with `gpiozero.LED` to control LED_2 on the Vision/Voice Bonnet.

`aiy.pins.`**`BUZZER_GPIO_PIN`**

The pin on the Raspberry Pi where the Vision Kit's piezo buzzer is connected (BCM 22). This should be used with *`aiy.toneplayer.TonePlayer`*.

`aiy.pins.`**`BUTTON_GPIO_PIN`**

The pin on the Raspberry Pi where the Vision/Voice Kit's button is connected (BCM 23). This should be used with `gpiozero.Button`.

CHAPTER 4

Vision Kit overview

The AIY Vision Kit is a do-it-yourself intelligent camera built with a Raspberry Pi and the Vision Bonnet.

After you assemble the kit and run the included demos, you can extend the kit with your own software and hardware.

Also see the Vision Kit assembly guide.

4.1 Software

To execute ML models and perform other actions with the Vision Kit, the system image includes the Python library with the following modules:

- `aiy.toneplayer`: A simple melodic music player for the piezo buzzer.
- `aiy.trackplayer`: A tracker-based music player for the piezo buzzer.
- `aiy.vision.annotator`: An annotation library that draws overlays on the Raspberry Pi's camera preview.
- `aiy.vision.inference`: An inference engine that communicates with the Vision Bonnet from the Raspberry Pi side.
- `aiy.vision.models`: A collection of modules that perform ML inferences with specific types of image classification and object detection models.
- `aiy.board`: APIs to use the button that's attached to the Vision Bonnet's button connector.
- `aiy.leds`: APIs to control certain LEDs, such as the LEDs in the button and the privacy LED.
- `aiy.pins`: Pin definitions for the bonnet's extra GPIO pins, for use with gpiozero.

4.2 Vision Bonnet

4.2.1 Hardware

- SOC: `Myriad 2450`

- MCU: `ATSAMD09D14` [I^2C address: `0x51`]
- LED Driver: `KTD2027A` [I^2C address: `0x30`]
- Crypto (optional): `ATECC608A` [I^2C address: `0x60`]
- IMU: `BMI160`

4.2.2 Drivers

- MCU driver: `modinfo aiy-io-i2c`
- MCU PWM driver: `modinfo pwm-aiy-io`
- MCU GPIO driver: `modinfo gpio-aiy-io`
- MCU ADC driver: `modinfo aiy-adc`
- LED driver: `modinfo leds-ktd202x`
- Software PWM driver for buzzer: `modinfo pwm-soft`
- Myriad driver: `modinfo aiy-vision`

To reset MCU:

```
echo 1 | sudo tee /sys/bus/i2c/devices/1-0051/reset
```

To get MCU status message (including firmware version) and last error code:

```
cat /sys/bus/i2c/devices/1-0051/{status_message,error_code}
```

4.2.3 Pinout (40-pin header)

```
                 3.3V --> 1    2 <-- 5V
              I2C_SDA --> 3    4 <-- 5V
              I2C_SCL --> 5    6 <-- GND
                          7    8
                  GND --> 9   10
                          11  12
                          13  14 <-- GND
  (GPIO_22) BUZZER_GPIO --> 15  16 <-- BUTTON_GPIO (GPIO_23)
                 3.3V --> 17  18
             SPI_MOSI --> 19  20 <-- GND
             SPI_MISO --> 21  22
             SPI_SCLK --> 23  24 <-- SPI_CE_MRD
                  GND --> 25  26
               ID_SDA --> 27  28 <-- ID_SCL
                          29  30 <-- GND
         PI_TO_MRD_IRQ --> 31  32
         MRD_TO_PI_IRQ --> 33  34 <-- GND
                          35  36
            MRD_UNUSED --> 37  38
                  GND --> 39  40
```

Also see the Vision Bonnet on pinout.xyz.

4.3 Troubleshooting

See the Vision Kit help.

CHAPTER 5

aiy.toneplayer

A simple melodic music player for the piezo buzzer.

This API is designed for the Vision Kit, but has no dependency on the Vision Bonnet, so may be used without it. It only requires a piezo buzzer connected to `aiy.pins.BUZZER_GPIO_PIN`.

class `aiy.toneplayer.`**`Note`**(*name*, *octave=4*, *bpm=120*, *period=4*)

Bases: `aiy.toneplayer.Rest`

Simple internal class to represent a musical note.

Used in part with the TonePlayer class, this object represents a musical note, including its name, octave, and how long it is played. End users shouldn't have to care about this too much and instead focus on the music language described in the TonePlayer class.

`BASE_OCTAVE = 4`

`to_frequency`(*tuning=440.0*)

Converts from a name and octave to a frequency in Hz.

Uses the specified tuning.

Parameters **`tuning`** – the frequency of the natural A note, in Hz.

class `aiy.toneplayer.`**`Rest`**(*bpm=120*, *period=4*)

Bases: `object`

Simple internal class to represent a musical rest note.

Used in part with the TonePlayer class, this object represents a period of time in a song where no sound is made. End users shouldn't have to care about this too much and instead focus on the music language described in the TonePlayer class.

`EIGHTH = 8`

`HALF = 2`

`QUARTER = 4`

`SIXTEENTH = 16`

WHOLE = 1

to_length_secs()

Converts from musical notation to a period of time in seconds.

class aiy.toneplayer.**TonePlayer**(*gpio*, *bpm=120*, *debug=False*)

Bases: `object`

Class to play a simplified music notation via a PWMController.

This class makes use of a very simple music notation to play simple musical tones out of a PWM controlled piezo buzzer.

The language consists of notes and rests listed in an array. Rests are moments in the song when no sound is produced, and are written in this way:

r<length>

The <length> may be one of the following five characters, or omitted:

w: whole note h: half note q: quarter note (the default – if you don't specify the length, we assume quarter) e: eighth note s: sixteenth note

So a half note rest would be written as "rh". A quarter note rest could be written as "r" or "rq".

Notes are similar to rests, but take the following form:

<note_name><octave><length>

<note_names> are written using the upper and lower case letters A-G and a-g. Uppercase letters are the natural notes, whereas lowercase letters are shifted up one semitone (sharp). Represented on a piano keyboard, the lowercase letters are the black keys. Thus, 'C' is the natural note C, whereas 'c' is actually C#, the first black key to the right of the C key.

The octave is optional, but is the numbers 1-8. If omitted, the TonePlayer assumes octave 4. Like the rests, the <length> may also be omitted and uses the same notation as the rest <length> parameter. If omitted, TonePlayer assumes a length of one quarter.

With this in mind, a middle C whole note could be written "C3w". Likewise, a C# quarter note in the 4th octave could be written as "c" or "c4q" or "cq".

NOTE_RE = re.compile('(?P<name>[A-Ga-g])(?P<octave>[1-8])?(?P<length>[whqes])?')

PERIOD_MAP = {'e': 8, 'h': 2, 'q': 4, 's': 16, 'w': 1}

REST_RE = re.compile('r(?P<length>[whqes])?')

play(**args*)

Plays a sequence of notes out the piezo buzzer.

CHAPTER 6

aiy.trackplayer

A tracker-based music player for the piezo buzzer.

This API is designed for the Vision Kit, but has no dependency on the Vision Bonnet, so may be used without it. It only requires a piezo buzzer connected to `aiy.pins.BUZZER_GPIO_PIN`.

class `aiy.trackplayer.`**`Arpeggio`**(**args*)

Bases: `aiy.trackplayer.Command`

Plays an arpeggiated chord.

`apply`(*player*, *controller*, *note*, *tick_delta*)

Applies the effect of this command.

`classmethod parse`(**args*)

Parses the arguments to this command into a new command instance.

Returns A tuple of an instance of this class and how many arguments were consumed from the argument list.

class `aiy.trackplayer.`**`Command`**

Bases: `object`

Base class for all commands.

`apply`(*player*, *controller*, *note*, *tick_delta*)

Applies the effect of this command.

`classmethod parse`(**args*)

Parses the arguments to this command into a new command instance.

Returns A tuple of an instance of this class and how many arguments were consumed from the argument list.

class `aiy.trackplayer.`**`Glissando`**(*direction*, *hz_per_tick*)

Bases: `aiy.trackplayer.Command`

Pitchbends a note up or down by the given rate.

apply(*player*, *controller*, *note*, *tick_delta*)
Applies the effect of this command.

classmethod parse(**args*)
Parses the arguments to this command into a new command instance.

Returns A tuple of an instance of this class and how many arguments were consumed from the argument list.

class aiy.trackplayer.**JumpToPosition**(*position*)
Bases: *aiy.trackplayer.Command*

Jumps to the given position in a song.

apply(*player*, *controller*, *note*, *tick_delta*)
Applies the effect of this command.

classmethod parse(**args*)
Parses the arguments to this command into a new command instance.

Returns A tuple of an instance of this class and how many arguments were consumed from the argument list.

class aiy.trackplayer.**NoteOff**
Bases: *aiy.trackplayer.Command*

Stops a given note from playing.

apply(*player*, *controller*, *note*, *tick_delta*)
Applies the effect of this command.

classmethod parse(**args*)
Parses the arguments to this command into a new command instance.

Returns A tuple of an instance of this class and how many arguments were consumed from the argument list.

class aiy.trackplayer.**PulseChange**(*direction*, *usec_per_tick*)
Bases: *aiy.trackplayer.Command*

Changes the pulse width of a note up or down by the given rate.

apply(*player*, *controller*, *note*, *tick_delta*)
Applies the effect of this command.

classmethod parse(**args*)
Parses the arguments to this command into a new command instance.

Returns A tuple of an instance of this class and how many arguments were consumed from the argument list.

class aiy.trackplayer.**Retrigger**(*times*)
Bases: *aiy.trackplayer.Command*

Retriggers a note a consecutive number of times.

apply(*player*, *controller*, *note*, *tick_delta*)
Applies the effect of this command.

classmethod parse(**args*)
Parses the arguments to this command into a new command instance.

Returns A tuple of an instance of this class and how many arguments were consumed from the argument list.

class aiy.trackplayer.**SetPulseWidth**(*pulse_width_usec*)
Bases: `aiy.trackplayer.Command`

Changes the pulse width of a note up or down by the given rate.

apply(*player*, *controller*, *note*, *tick_delta*)
Applies the effect of this command.

classmethod parse(**args*)
Parses the arguments to this command into a new command instance.

Returns A tuple of an instance of this class and how many arguments were consumed from the argument list.

class aiy.trackplayer.**SetSpeed**(*speed*)
Bases: `aiy.trackplayer.Command`

Changes the speed of the given song.

apply(*player*, *controller*, *note*, *tick_delta*)
Applies the effect of this command.

classmethod parse(**args*)
Parses the arguments to this command into a new command instance.

Returns A tuple of an instance of this class and how many arguments were consumed from the argument list.

class aiy.trackplayer.**StopPlaying**
Bases: `aiy.trackplayer.Command`

Stops the TrackPlayer from playing.

apply(*player*, *controller*, *note*, *tick_delta*)
Applies the effect of this command.

classmethod parse(**args*)
Parses the arguments to this command into a new command instance.

Returns A tuple of an instance of this class and how many arguments were consumed from the argument list.

class aiy.trackplayer.**TrackLoader**(*gpio*, *filename*, *debug=False*)
Bases: `object`

Simple track module loader.

This class, given a filename and a gpio will load and parse in the given track file and initialize a TrackPlayer instance to play it.

The format of a track file is a plain text file consisting of a header, followed by a number of pattern definitions. Whitespace is ignored in the header and between the patterns.

The header may contain a set of key value pairs like so:

title Some arbitrary song name speed <speed> order <number> [<number>. . .] end

"title" specifies the title of the song. Optional. This isn't actually used by the player, but is a nice bit of metadata for humans.

"speed" sets the speed in ticks/row. Optional. The argument, <speed> must be an int. If this isn't present, the player defaults to a speed of 3.

"order" sets the order of the patterns. It is a single line of space separated integers, starting at 0. Each integer refers to the pattern in order in the file. This keyword must be present.

The keyword "end", which ends the header.

Patterns take the form:

> pattern [E5] [cmnd [<arg>. . .] . . .] end

Patterns are started with the "pattern" keyword and end with the "end" keyword. Blank lines inside a pattern are significant – they add time to the song. Any notes that were played continue to play unless they were stopped.

Each row of a pattern consists of a note followed by any number of commands and arguments. A note consists of an upper or lowercase letter A-G (lowercase are sharp notes) and an octave number between 1 and 8. Any time a note appears, it will play only on the first tick, augmented by any commands on the same row. Notes are optional per row.

Commands are four letter lowercase words whose effect is applied every tick. A row may contain nothing but commands, if need be. If the current speed is 3, that means each effect will occur 3 times per row. There may be any number of commands followed by arguments on the same row. Commands available as of this writing are as follows:

> glis <direction> <amount-per-tick> puls <direction> <amount-per-tick> spwd <width> arpg [<note>. . .] vibr <depth> <speed> retg <times> noff sspd <speed> jump <position> stop

glis is a glissando effect, which takes in a <direction> (a positive or negative number) as a direction to go in terms of frequency shift. The <amount-per-tick> value is an integer that is how much of a shift in Hz to apply in the given direction every tick.

puls changes the pulse width of the current PWM waveform in the given <direction> by the <amount-per-tick> in microseconds. <direction> is like <direction> to the glis command.

spwd sets the pulse width of the current PWM waveform directly. <width> is the width of the pulse in microseconds.

arpg performs an arpeggio using the currently playing note and any number of notes listed as arguments. Each note is played sequentially, starting with the currently playing note, every tick. Note that to continue the arpeggio, it will be necessary to list multiple arpg commands in sequence.

vibr performs a vibrato using the currently playing note. The vibrato is applied using the given <depth> in Hz, and the given <speed>.

retg retriggers the note every tick the given number of <times>. This allows for very fast momentary effects when combined with glis, puls, and arpg and high speed values.

noff stops any previously playing note.

sspd sets the current playing speed in <speed> ticks per row.

jump jumps to the given row <position> (offset from the start of the pattern) and continues playing.

stop stops the Player from playing.

`COMMANDS = {'arpg': <class 'aiy.trackplayer.Arpeggio'>, 'glis': <class 'aiy.trackpla`

`COMMAND_RE = re.compile('(?P<name>[a-z]{4})')`

`NOTE_RE = re.compile('(?P<name>[A-Ga-g])(?P<octave>[1-8])')`

`load``()`

> Loads the track module from disk.
>
> **Returns** A fully initialized TrackPlayer instance, ready to play.

`class` `aiy.trackplayer.`**`TrackPlayer`**`(`*gpio*, *speed=3*, *debug=False*`)`

> Bases: `object`
>
> Plays a tracker-like song.

add_order(*pattern_number*)
Adds a pattern index to the order.

add_pattern(*pattern*)
Adds a new pattern to the player.

Returns The new pattern index.

play()
Plays the currently configured track.

set_order(*position*, *pattern_number*)
Changes a pattern index in the order.

set_position(*new_position*)
Sets the position inside of the current pattern.

set_speed(*new_speed*)
Sets the playing speed in ticks/row.

stop()
Stops playing any currently playing track.

class aiy.trackplayer.**Vibrato**(*depth_hz*, *speed*)
Bases: *aiy.trackplayer.Command*

Vibrates the frequency by the given amount.

apply(*player*, *controller*, *note*, *tick_delta*)
Applies the effect of this command.

classmethod parse(**args*)
Parses the arguments to this command into a new command instance.

Returns A tuple of an instance of this class and how many arguments were consumed from the argument list.

CHAPTER 7

aiy.vision.annotator

An annotation library that draws overlays on the Raspberry Pi's camera preview.

Annotations include bounding boxes, text overlays, and points. Annotations support partial opacity, however only with respect to the content in the preview. A transparent fill value will cover up previously drawn overlay under it, but not the camera content under it. A color of None can be given, which will then not cover up overlay content drawn under the region.

Note: Overlays do not persist through to the storage layer so images saved from the camera, will not contain overlays.

class `aiy.vision.annotator.`**`Annotator`**(*camera*, *bg_color=None*, *default_color=None*, *dimensions=None*)

Bases: `object`

Utility for managing annotations on the camera preview.

Parameters

- **`camera`** – picamera.PiCamera camera object to overlay on top of.
- **`bg_color`** – PIL.ImageColor (with alpha) for the background of the overlays.
- **`default_color`** – PIL.ImageColor (with alpha) default for the drawn content.

`bounding_box`(*rect*, *outline=None*, *fill=None*)

Draws a bounding box around the specified rectangle.

Parameters

- **`rect`** – (x1, y1, x2, y2) rectangle to be drawn - where (x1,y1) and (x2, y2) are opposite corners of the desired rectangle.
- **`outline`** – PIL.ImageColor with which to draw the outline (defaults to the configured default_color).
- **`fill`** – PIL.ImageColor with which to fill the rectangel (defaults to None
- **`will not cover up drawings under the region.`** (*`which`*) –

clear()
Clears the contents of the overlay - leaving only the plain background.

point(*location*, *radius=1*, *color=None*)
Draws a point of the given size at the given location.

Parameters

- **location** – (x,y) center of the point to be drawn.
- **radius** – the radius of the point to be drawn.
- **color** – The color to draw the point in (defaults to default_color).

stop()
Removes the overlay from the screen.

text(*location*, *text*, *color=None*)
Draws the given text at the given location.

Parameters

- **location** – (x,y) point at which to draw the text (upper left corner).
- **text** – string to be drawn.
- **color** – PIL.ImageColor to draw the string in (defaults to default_color).

update()
Updates the contents of the overlay.

CHAPTER 8

aiy.vision.inference

An inference engine that communicates with the Vision Bonnet from the Raspberry Pi side.

It can be used to load a model, analyze local image or image from camera shot. It automatically unload the model once the associated object is deleted. See image_classification.py and object_recognition.py as examples on how to use this API.

class aiy.vision.inference.**CameraInference**(*descriptor*, *params=None*, *sparse_configs=None*)

Bases: `object`

Helper class to run camera inference.

close()

count

engine

rate

run(*count=None*)

class aiy.vision.inference.**FirmwareVersion**(*major*, *minor*)

Bases: `tuple`

major

Alias for field number 0

minor

Alias for field number 1

exception aiy.vision.inference.**FirmwareVersionException**(**args*, ***kwargs*)

Bases: `Exception`

class aiy.vision.inference.**FromSparseTensorConfig**(*logical_shape*, *tensor_name*, *squeeze_dims*)

Bases: `tuple`

logical_shape

Alias for field number 0

squeeze_dims
Alias for field number 2

tensor_name
Alias for field number 1

class aiy.vision.inference.**ImageInference**(*descriptor*)
Bases: object

Helper class to run image inference.

close()

engine

run(*image*, *params=None*, *sparse_configs=None*)

class aiy.vision.inference.**InferenceEngine**
Bases: object

Class to access InferenceEngine on VisionBonnet board.

Inference result has the following format:

```
message InferenceResult {
  string model_name;  // Name of the model to run inference on.
  int32 width;        // Input image/frame width.
  int32 height;       // Input image/frame height.
  Rectangle window;   // Window inside width x height image/frame.
  int32 duration_ms;  // Inference duration.
  map<string, FloatTensor> tensors;  // Output tensors.

  message Frame {
    int32 index;        // Frame number.
    int64 timestamp_us; // Frame timestamp.
  }

  Frame frame;          // Frame-specific inference data.
}
```

camera_inference()
Returns the latest inference result from VisionBonnet.

close()

get_camera_state()
Returns current camera state.

get_firmware_info()
Returns firmware version as (major, minor) tuple.

get_inference_state()
Returns inference state.

get_system_info()
Returns system information: uptime, memory usage, temperature.

image_inference(*model_name*, *image*, *params=None*, *sparse_configs=None*)
Runs inference on image using model identified by model_name.

Parameters

- **model_name** – string, unique identifier used to refer a model.

- **image** – PIL.Image,
- **params** – dict, additional parameters to run inference

Returns pb2.Response.InferenceResult

load_model(*descriptor*)
Loads model on VisionBonnet.

Parameters **descriptor** – ModelDescriptor, meta info that defines model name, where to get the model and etc.

Returns Model identifier.

reset()

start_camera_inference(*model_name*, *params=None*, *sparse_configs=None*)
Starts inference running on VisionBonnet.

stop_camera_inference()
Stops inference running on VisionBonnet.

unload_model(*model_name*)
Deletes model on VisionBonnet.

Parameters **model_name** – string, unique identifier used to refer a model.

exception aiy.vision.inference.**InferenceException**(**args*, ***kwargs*)
Bases: Exception

class aiy.vision.inference.**ModelDescriptor**(*name*, *input_shape*, *input_normalizer*, *compute_graph*)
Bases: tuple

compute_graph
Alias for field number 3

input_normalizer
Alias for field number 2

input_shape
Alias for field number 1

name
Alias for field number 0

class aiy.vision.inference.**ThresholdingConfig**(*logical_shape*, *threshold*, *top_k*, *to_ignore*)
Bases: tuple

logical_shape
Alias for field number 0

threshold
Alias for field number 1

to_ignore
Alias for field number 3

top_k
Alias for field number 2

CHAPTER 9

aiy.vision.models

A collection of modules that perform ML inferences with specific types of image classification and object detection models.

Each of these modules has a corresponding sample app in src/examples/vision. Also see the instructions to run the models with the Vision Kit.

9.1 aiy.vision.models.dish_classification

API for Dish Classification.

`aiy.vision.models.dish_classification.`**`get_classes`**(*result*, *top_k=None*, *threshold=0.0*)

Converts dish classification model output to list of detected objects.

Parameters

- **`result`** – output tensor from dish classification model.
- **`top_k`** – int; max number of objects to return.
- **`threshold`** – float; min probability of each returned object.

Returns

string, probability: float) pairs ordered by probability from highest to lowest. The number of pairs is not greater than top_k. Each probability is greater than threshold. For example:

[('Ramen', 0.981934) ('Yaka mein, 0.005497)]

Return type A list of (class_name

`aiy.vision.models.dish_classification.`**`model`**()

9.2 aiy.vision.models.dish_detection

API for Dish Detection.

class `aiy.vision.models.dish_detection.`**`Dish`**(*sorted_scores*, *bounding_box*)

Bases: `tuple`

`bounding_box`

Alias for field number 1

`sorted_scores`

Alias for field number 0

`aiy.vision.models.dish_detection.`**`get_dishes`**(*result*, *top_k=3*, *threshold=0.1*)

Returns list of Dish objects decoded from the inference result.

`aiy.vision.models.dish_detection.`**`model`**()

9.3 aiy.vision.models.face_detection

API for Face Detection.

class `aiy.vision.models.face_detection.`**`Face`**(*face_score*, *joy_score*, *bounding_box*)

Bases: `tuple`

`bounding_box`

Alias for field number 2

`face_score`

Alias for field number 0

`joy_score`

Alias for field number 1

`aiy.vision.models.face_detection.`**`get_faces`**(*result*)

Returns list of Face objects decoded from the inference result.

`aiy.vision.models.face_detection.`**`model`**()

9.4 aiy.vision.models.image_classification

API for Image Classification tasks.

`aiy.vision.models.image_classification.`**`get_classes`**(*result*, *top_k=None*, *threshold=0.0*)

Converts image classification model output to list of detected objects.

Parameters

- **`result`** – output tensor from image classification model.
- **`top_k`** – int; max number of objects to return.
- **`threshold`** – float; min probability of each returned object.

Returns

string, probability: float) pairs ordered by probability from highest to lowest. The number of pairs is not greater than top_k. Each probability is greater than threshold. For example:

[('Egyptian cat', 0.767578) ('tiger cat, 0.163574) ('lynx/catamount', 0.039795)]

Return type A list of (class_name

`aiy.vision.models.image_classification.`**`get_classes_sparse`**(*result*)

Converts sparse image classification model output to list of detected objects.

Parameters **`result`** – sparse output tensor from image classification model.

Returns

string, probability: float) pairs ordered by probability from highest to lowest. For example:

[('Egyptian cat', 0.767578) ('tiger cat, 0.163574)

Return type A list of (class_name

`aiy.vision.models.image_classification.`**`model`**(*model_type='image_classification_mobilenet'*)

`aiy.vision.models.image_classification.`**`sparse_configs`**(*top_k=0*, *threshold=0.0*, *model_type='image_classification_mobilenet'*)

9.5 aiy.vision.models.inaturalist_classification

API for detecting plants, insects, and birds from the iNaturalist dataset.

`class` `aiy.vision.models.inaturalist_classification.`**`Model`**

Bases: *`aiy.vision.models.inaturalist_classification.Model`*

`compute_graph`()

`aiy.vision.models.inaturalist_classification.`**`get_classes`**(*result*, *top_k=None*, *threshold=0.0*)

`aiy.vision.models.inaturalist_classification.`**`get_classes_sparse`**(*result*)

`aiy.vision.models.inaturalist_classification.`**`model`**(*model_type*)

`aiy.vision.models.inaturalist_classification.`**`sparse_configs`**(*model_type*, *top_k=None*, *threshold=0.0*)

9.6 aiy.vision.models.object_detection

API for Object Detection tasks.

`class` `aiy.vision.models.object_detection.`**`Object`**(*bounding_box*, *kind*, *score*)

Bases: `object`

Object detection result.

`BACKGROUND = 0`

`CAT = 2`

`DOG = 3`

`PERSON = 1`

`aiy.vision.models.object_detection.`**`get_objects`**(*result*, *threshold=0.3*, *offset=(0, 0)*)

`aiy.vision.models.object_detection.`**`get_objects_sparse`**(*result*, *offset=(0, 0)*) `aiy.vision.models.object_detection.`**`model`**`()`

`aiy.vision.models.object_detection.`**`sparse_configs`**`(`*threshold=0.3*`)`

CHAPTER 10

Voice Kit overview

The AIY Voice Kit is a do-it-yourself intelligent speaker built with a Raspberry Pi and the Voice Bonnet (or Voice HAT if using the V1 Voice Kit).

After you assemble the kit and run the included demos, you can extend the kit with your own software and hardware.

Also see the Voice Kit assembly guide.

10.1 Software

To interact with the Google Assistant, convert speech to text, and perform other actions with the Voice Kit, the system image includes Python library with the following modules:

- `aiy.assistant`: A collection of modules that simplify interaction with the Google Assistant API.
- `aiy.cloudspeech`: APIs to simplify interaction with the Google Cloud Speech-to-Text service.
- `aiy.voice.audio`: APIs to record and play audio files.
- `aiy.voice.tts`: An API that performs text-to-speech.
- `aiy.board`: APIs to use the button that's attached to the Voice Bonnet's button connector.
- `aiy.leds`: APIs to control certain LEDs, such as the LEDs in the button and the privacy LED.
- `aiy.pins`: Pin definitions for the bonnet's extra GPIO pins, for use with gpiozero.

10.2 Voice Bonnet (Voice Kit V2)

10.2.1 Hardware

- Audio Codec: `ALC5645` [I^{2}C address: `0x1A`]
- MCU: `ATSAMD09D14` [I^{2}C address: `0x52`]

- LED Driver: `KTD2027B` [I^2C address: `0x31`]
- Crypto (optional): `ATECC608A` [I^2C address: `0x62`]
- Microphone: `SPH1642HT5H-1` x 2

10.2.2 Drivers

- MCU driver: `modinfo aiy-io-i2c`
- MCU PWM driver: `modinfo pwm-aiy-io`
- MCU GPIO driver: `modinfo gpio-aiy-io`
- MCU ADC driver: `modinfo aiy-adc`
- LED driver: `modinfo leds-ktd202x`
- Software PWM driver for buzzer: `modinfo pwm-soft`
- Sound drivers: `modinfo rl6231 rt5645 snd_aiy_voicebonnet`

10.2.3 Pinout (40-pin header)

```
       3.3V --> 1    2 <-- 5V
                3    4 <-- 5V
                5    6 <-- GND
                7    8
        GND --> 9   10
               11   12 <-- I2S_BCLK
               13   14 <-- GND
               15   16 <-- BUTTON_GPIO (GPIO_23)
       3.3V --> 17  18
               19   20 <-- GND
               21   22 <-- LED_GPIO (GPIO_25)
               23   24
        GND --> 25  26
     ID_SDA --> 27  28 <-- ID_SCL
               29   30 <-- GND
               31   32
               33   34 <-- GND
  I2S_LRCLK --> 35  36 <-- AMP_ENABLE
               37   38 <-- I2S_DIN
        GND --> 39  40 <-- I2S_DOUT
```

Also see the Voice Bonnet on pinout.xyz.

10.3 Voice HAT (Voice Kit V1)

10.3.1 Hardware

- Audio Amplifier: `MAX98357A`
- Microphone: `ICS-43432` x 2

10.3.2 Schematics

- Main Board
- Microphone Board

10.3.3 Drivers

- googlevoicehat-codec.c
- googlevoicehat-soundcard.c
- googlevoicehat-soundcard-overlay.dts

Manual overlay load:

```
sudo dtoverlay googlevoicehat-soundcard
```

Load overlay on each boot:

```
echo "dtoverlay=googlevoicehat-soundcard" | sudo tee -a /boot/config.txt
```

10.3.4 Pinout (40-pin header)

```
       3.3V --> 1    2 <-- 5V
    I2C_SDA --> 3    4 <-- 5V
    I2C_SCL --> 5    6 <-- GND
                7    8
        GND --> 9    10
                11   12 <-- I2S_BCLK
                13   14 <-- GND
                15   16 <-- BUTTON_GPIO (GPIO_23)
       3.3V --> 17   18
                19   20 <-- GND
                21   22
                23   24
        GND --> 25   26
     ID_SDA --> 27   28 <-- ID_SCL
                29   30 <-- GND
                31   32
                33   34 <-- GND
  I2S_LRCLK --> 35   36
                37   38 <-- I2S_DIN
        GND --> 39   40 <-- I2S_DOUT
```

Also see the Voice HAT on pinout.xyz.

10.4 Troubleshooting

See the Voice Kit help.

CHAPTER 11

aiy.assistant

APIs that simplify interaction with the Google Assistant API in one of two ways: using either `aiy.assistant.grpc` or `aiy.assistant.library`, corresponding to the Google Assistant Service and Google Assistant Library, respectively.

Which of these you choose may depend on your intentions. The Google Assistant Service provides a gRPC interface that is generally more complicated. However, the `aiy.assistant.grpc` API offered here does not provide access to those APIs. Instead, it completely wraps the google.assistant.embedded.v1alpha2 APIs. It takes care of all the complicated setup for you, and handles all response events. Thus, if all you want to build a basic version of the Google Assistant, then using `aiy.assistant.grpc` is easiest because it requires the least amount of code. For an example, see src/examples/voice/assistant_grpc_demo.py.

On the other hand, `aiy.assistant.library` is a thin wrapper around google.assistant.library. It overrides the `Assistant.start()` method to handle the device registration, but beyond that, you can and must use the google.assistant.library APIs to respond to all events returned by the Google Assistant. As such, using `aiy.assistant.library` provides you more control, allowing you to build custom device commands based on conversation with the Google Assistant. For an example, see src/examples/voice/assistant_library_with_local_commands_demo.py.

Additionally, only `aiy.assistant.library` includes built-in support for hotword detection (such as "Okay Google"). However, if you're using the Raspberry Pi Zero (provided with the V2 Voice Kit), then you cannot use hotword detection because that feature depends on the ARMv7 architecture and the Pi Zero has only ARMv6. So that feature of the library works only with Raspberry Pi 2/3, and if you're using a Pi Zero, you must instead use the button or another type of trigger to initiate a conversation with the Google Assistant. (Note: The Voice Bonnet can be used on any Raspberry Pi.)

Tip: If all you want to do is create custom voice commands (such as "turn on the light"), then you don't need to interact with the Google Assistant. Instead, you can use `aiy.cloudspeech` to convert your voice commands into text that triggers your actions.

Note: These APIs are designed for the Voice Kit, but have no dependency on the Voice HAT/Bonnet specifically. However, they do require some type of sound card attached to the Raspberry Pi that can be detected by the ALSA subsystem.

11.1 aiy.assistant.grpc

Enables a conversation with the Google Assistant, using the Google Assistant Service, which connects to the Google Assistant using a streaming endpoint over gRPC.

This gRPC service is typically more complicated to set up, compared to the Google Assistant Library, but this API takes care of all the complexity for you. So you simply create an instance of `AssistantServiceClient`, then start the Google Assistant by calling `conversation()`.

This API provides only an interface to initiate a conversation with the Google Assistant. It speaks and prints all responses for you—it does not allow you to handle the response events or create custom commands. For an example, see src/examples/voice/assistant_grpc_demo.py.

If you want to integrate custom device commands with the Google Assistant using the gRPC interface, instead use the Google Assistant Service directly. For an example, see this gRPC sample. Or instead of interacting with the Google Assistant, you can use `aiy.cloudspeech` to convert your voice commands into text that triggers your actions.

class aiy.assistant.grpc.**AssistantServiceClient**(*language_code='en-US'*, *volume_percentage=100*)

Bases: `object`

Provides a simplified interface for the EmbeddedAssistant.

Parameters

- **language_code** – Language expected from the user, in IETF BCP 47 syntax (default is "en-US"). See the list of supported languages.
- **volume_percentage** – Volume level of the audio output. Valid values are 1 to 100 (corresponding to 1% to 100%).

conversation(*deadline=185*)

Starts a conversation with the Google Assistant.

The device begins listening for your query or command and will wait indefinitely. Once it completes a query/command, it returns to listening for another.

Parameters **deadline** – The amount of time (in milliseconds) to wait for each gRPC request to complete before terminating.

volume_percentage

Volume level of the audio output. Valid values are 1 to 100 (corresponding to 1% to 100%).

class aiy.assistant.grpc.**AssistantServiceClientWithLed**(*board*, *language_code='en-US'*, *volume_percentage=100*)

Bases: `aiy.assistant.grpc.AssistantServiceClient`

Same as `AssistantServiceClient` but this also turns the Voice Kit's button LED on and off in response to the conversation.

Parameters

- **board** – An instance of `Board`.
- **language_code** – Language expected from the user, in IETF BCP 47 syntax (default is "en-US"). See the list of supported languages.
- **volume_percentage** – Volume level of the audio output. Valid values are 1 to 100 (corresponding to 1% to 100%).

conversation(*deadline=185*)

Starts a conversation with the Google Assistant.

The device begins listening for your query or command and will wait indefinitely. Once it completes a query/command, it returns to listening for another.

Parameters **`deadline`** – The amount of time (in milliseconds) to wait for each gRPC request to complete before terminating.

`volume_percentage`
Volume level of the audio output. Valid values are 1 to 100 (corresponding to 1% to 100%).

11.2 aiy.assistant.library

Facilitates access to the Google Assistant Library, which provides APIs to initiate conversations with the Google Assistant and create custom device commands commands.

This includes a wrapper for the `Assistant` class only. You must import all other Google Assistant classes directly from the google.assistant.library module to handle each of the response events.

Note: Hotword detection (such as "Okay Google") is not supported with the Raspberry Pi Zero (only with Raspberry Pi 2/3). If you're using a Pi Zero, you must instead use the button or another type of trigger to initiate a conversation with the Google Assistant.

class `aiy.assistant.library.`**`Assistant`**(*credentials*)
Bases: google.assistant.library.Assistant

A wrapper for the Assistant class that handles model and device registration based on the project name in your OAuth credentials (`assistant.json`) file.

All the `Assistant` APIs are available through this class, such as start() to start the Assistant, and start_conversation() to start a conversation, but they are not documented here. Instead refer to the Google Assistant Library for Python documentation.

To get started, you must call *`get_assistant_credentials()`* and pass the result to the `Assistant` constructor. For example:

```
from google.assistant.library.event import EventType
from aiy.assistant import auth_helpers
from aiy.assistant.library import Assistant

credentials = auth_helpers.get_assistant_credentials()
with Assistant(credentials) as assistant:
    for event in assistant.start():
        process_event(event)
```

For more example code, see src/examples/voice/assistant_library_demo.py.

Parameters **`credentials`** – The Google OAuth2 credentials for the device. Get this from *`get_assistant_credentials()`*.

11.3 aiy.assistant.auth_helpers

Authentication helper for the Google Assistant API.

`aiy.assistant.auth_helpers.`**`get_assistant_credentials`**(*credentials_file=None*)
Retreives the OAuth credentials required to access the Google Assistant API.

If you're using *aiy.assistant.library*, you must call this function and pass the result to the *Assistant* constructor.

If you're using *aiy.assistant.grpc*, you do not need this function because the *AssistantServiceClient* calls this during initialization (using the credentials file at `~/assistant.json`).

Parameters **`credentials_file`** – Absolute path to your JSON credentials file. If None, it looks for the file at `~/assistant.json`. To get a credentials file, follow these instructions.

Returns The device OAuth credentials, as a `google.oauth2.credentials.Credentials` object.

CHAPTER 12

aiy.cloudspeech

APIs that simplify interaction with the Google Cloud Speech-to-Text service so you can convert voice commands into actions. To use this service, you must have a Google Cloud account and a corresponding credentials file. For more information, see these setup instructions.

For an example, see src/examples/voice/cloudspeech_demo.py.

Note: These APIs are designed for the Voice Kit, but have no dependency on the Voice HAT/Bonnet specifically. However, they do require some type of sound card attached to the Raspberry Pi that can be detected by the ALSA subsystem.

class `aiy.cloudspeech.`**`CloudSpeechClient`**(*service_accout_file=None*)

Bases: `object`

A simplified version of the Google Cloud `SpeechClient` class.

Parameters **`service_accout_file`** – Absolute path to your JSON account credentials file. If None, it looks for the file at `~/cloud_speech.json`. To get a credentials file, these setup instructions.

`recognize`(*language_code='en-US', hint_phrases=None*)

Performs speech-to-text for a single utterance using the default ALSA soundcard driver. Once it detects the user is done speaking, it stops listening and delivers the top result as text.

By default, this method calls *`start_listening()`* and *`stop_listening()`* as the recording begins and ends, respectively.

Parameters

- **`language_code`** – Language expected from the user, in IETF BCP 47 syntax (default is "en-US"). See the list of Cloud's supported languages.
- **`hint_phrase`** – A list of strings containing words and phrases that may be expected from the user. These hints help the speech recognizer identify them in the dialog and improve the accuracy of your results.

Returns The text transcription of the user's dialog.

recognize_bytes(*data*, *language_code='en-US'*, *hint_phrases=None*)

Performs speech-to-text for a single utterance using the given data source. Once it detects the user is done speaking, it stops listening and delivers the top result as text.

Parameters

- **data** – The audio data source. Must be encoded with a sample rate of 16000Hz.
- **language_code** – Language expected from the user, in IETF BCP 47 syntax (default is "en-US"). See the list of Cloud's supported languages.
- **hint_phrase** – A list of strings containing words and phrases that may be expected from the user. These hints help the speech recognizer identify them in the dialog and improve the accuracy of your results.

Returns The text transcription of the user's dialog.

start_listening()

By default, this simply prints "Start listening" to the log.

This method is provided as a convenience method that you can override in a derived class to do something else that indicates the status to the user, such as change the LED state.

Called by *recognize()* when recording begins.

stop_listening()

By default, this simply prints "Stop listening" to the log.

This method is provided as a convenience method that you can override in a derived class to do something else that indicates the status to the user, such as change the LED state.

Called by *recognize()* when recording ends.

CHAPTER 13

aiy.voice.audio

APIs to record and play audio files.

Note: These APIs are designed for the Voice Kit, but have no dependency on the Voice HAT/Bonnet specifically. However, many of the APIs require some type of sound card attached to the Raspberry Pi that can be detected by the ALSA subsystem.

13.1 Recording

`aiy.voice.audio.`**`arecord`**(*fmt*, *filetype='raw'*, *filename=None*, *device='default'*)

Returns an `arecord` command-line command.

Parameters

- **`fmt`** – The audio format; an instance of *`AudioFormat`*.
- **`filetype`** – The type of file. Must be either 'wav', 'raw', 'voc', or 'au'.
- **`filename`** – The audio file to play.
- **`device`** – The PCM device name. Leave as `default` to use the default ALSA soundcard.

`aiy.voice.audio.`**`record_file`**(*fmt*, *filename*, *filetype*, *wait*, *device='default'*)

Records an audio file (blocking). The length of the recording is determined by a blocking `wait` function that you provide. When your `wait` function finishes, so does this function and the recording.

For an example, see src/examples/voice/voice_recorder.py.

Parameters

- **`fmt`** – The audio format; an instance of *`AudioFormat`*.
- **`filename`** – The file where the recording should be saved.
- **`filetype`** – The type of file. Must be either 'wav', 'raw', 'voc', or 'au'.

- **wait** – A blocking function that determines the length of the recording (and thus the length of time that this function is blocking).
- **device** – The PCM device name. Leave as `default` to use the default ALSA soundcard.

`aiy.voice.audio.`**`record_file_async`**(*fmt*, *filename*, *filetype*, *device='default'*)

Records an audio file, asynchronously. To stop the recording, terminate the returned `Popen` object.

Parameters

- **fmt** – The audio format; an instance of *`AudioFormat`*.
- **filename** – The file where the recording should be saved.
- **filetype** – The type of file. Must be either 'wav', 'raw', 'voc', or 'au'.
- **device** – The PCM device name. Leave as `default` to use the default ALSA soundcard.

Returns The `Popen` object for the subprocess in which audio is recording.

class `aiy.voice.audio.`**`Recorder`**

Bases: `object`

done()

Stops the recording that started via *`record()`*.

join()

record(*fmt*, *chunk_duration_sec*, *device='default'*, *num_chunks=None*, *on_start=None*, *on_stop=None*, *filename=None*)

Records audio with the ALSA soundcard driver, via `arecord`.

Parameters

- **fmt** – The audio format; an instance of *`AudioFormat`*.
- **chunk_duration_sec** – The duration of each audio chunk, in seconds (may be float).
- **device** – The PCM device name. Leave as `default` to use the default ALSA soundcard.
- **num_chunks** – The number of chunks to record. Leave as `None` to instead record indefinitely, until you call *`done()`*.
- **on_start** – A function callback to call when recording starts.
- **on_stop** – A function callback to call when recording stops.
- **filename** – A filename to use if you want to save the recording as a WAV file.

Yields A chunk of audio data. Each chunk size = `chunk_duraction_sec * fmt.bytes_per_second`

13.2 Playback

`aiy.voice.audio.`**`aplay`**(*fmt*, *filetype='raw'*, *filename=None*, *device='default'*)

Returns an `aplay` command-line command.

Parameters

- **fmt** – The audio format; an instance of *`AudioFormat`*.
- **filetype** – The type of file. Must be either 'wav', 'raw', 'voc', or 'au'.
- **filename** – The audio file to play.

- **device** – The PCM device name. Leave as `default` to use the default ALSA soundcard.

`aiy.voice.audio.`**`play_raw`**(*fmt*, *filename_or_data*)

Plays raw audio data (blocking).

Parameters

- **fmt** – The audio format; an instance of *AudioFormat*.
- **filename_or_data** – The file or bytes to play.

`aiy.voice.audio.`**`play_raw_async`**(*fmt*, *filename_or_data*)

Plays raw audio data asynchronously.

Parameters

- **fmt** – The audio format; an instance of *AudioFormat*.
- **filename_or_data** – The file or bytes to play.

Returns The `Popen` object for the subprocess in which audio is playing.

`aiy.voice.audio.`**`play_wav`**(*filename_or_data*)

Plays a WAV file or data (blocking).

Parameters **filename_or_data** – The WAV file or bytes to play.

`aiy.voice.audio.`**`play_wav_async`**(*filename_or_data*)

Plays a WAV file or data asynchronously.

Parameters **filename_or_data** – The WAV file or bytes to play.

Returns The `Popen` object for the subprocess in which audio is playing.

class `aiy.voice.audio.`**`BytesPlayer`**

Bases: `aiy.voice.audio.Player`

Plays audio from a given byte data source.

`join`()

`play`(*fmt*, *device='default'*)

Parameters

- **fmt** – The audio format; an instance of *AudioFormat*.
- **device** – The PCM device name. Leave as `default` to use the default ALSA soundcard.

Returns A closure with an inner function `push()` that accepts the byte data.

class `aiy.voice.audio.`**`FilePlayer`**

Bases: `aiy.voice.audio.Player`

Plays audio from a file.

`join`()

`play_raw`(*fmt*, *filename*, *device='default'*)

Plays a raw audio file.

Parameters

- **fmt** – The audio format; an instance of *AudioFormat*.
- **filename** – The audio file to play.

- **`device`** – The PCM device name. Leave as `default` to use the default ALSA soundcard.

`play_wav` (*filename*, *device='default'*)
Plays a WAV file.

Parameters

- **`filename`** – The WAV file to play.
- **`device`** – The PCM device name. Leave as `default` to use the default ALSA soundcard.

13.3 Audio format

`aiy.voice.audio.`**`wave_set_format`** (*wav_file*, *fmt*)
Sets the format for the given WAV file, using the given *`AudioFormat`*.

Parameters

- **`wav_file`** – A `wave.Wave_write` object.
- **`fmt`** – A *`AudioFormat`* object.

`aiy.voice.audio.`**`wave_get_format`** (*wav_file*)
Returns the *`AudioFormat`* corresponding to the WAV file provided.

Parameters **`wav_file`** – A `wave.Wave_read` object.

`class` `aiy.voice.audio.`**`AudioFormat`**
Bases: *`aiy.voice.audio.AudioFormat`*

`CD = AudioFormat(sample_rate_hz=44100, num_channels=2, bytes_per_sample=2)`

`bytes_per_second`

CHAPTER 14

aiy.voice.tts

An API that performs text-to-speech.

You can also use this to perform text-to-speech from the command line:

```
python ~/AIY-projects-python/src/aiy/voice/tts.py "hello world"
```

`aiy.voice.tts.`**`say`**(*text*, *lang='en-US'*, *volume=60*, *pitch=130*, *speed=100*, *device='default'*)

Speaks the provided text.

Parameters

- **`text`** – The text you want to speak.
- **`lang`** – The language to use. Supported languages are: en-US, en-GB, de-DE, es-ES, fr-FR, it-IT.
- **`volume`** – Volume level for the converted audio. The normal volume level is 100. Valid volume levels are between 0 (no audible output) and 500 (increasing the volume by a factor of 5). Values higher than 100 might result in degraded signal quality due to saturation effects (clipping) and is not recommended. To instead adjust the volume output of your device, enter `alsamixer` at the command line.
- **`pitch`** – The pitch level for the voice. The normal pitch level is 100, the allowed values lie between 50 (one octave lower) and 200 (one octave higher).
- **`speed`** – The speed of the voice. The normal speed level is 100, the allowed values lie between 20 (slowing down by a factor of 5) and 500 (speeding up by a factor of 5).
- **`device`** – The PCM device name. Leave as `default` to use the default ALSA soundcard.

CHAPTER 15

API indices

- Full index
- Module index

Python Module Index

a

aiy.assistant.auth_helpers, 41
aiy.assistant.grpc, 40
aiy.assistant.library, 41
aiy.board, 3
aiy.cloudspeech, 43
aiy.leds, 5
aiy.pins, 11
aiy.toneplayer, 17
aiy.trackplayer, 19
aiy.vision.annotator, 25
aiy.vision.inference, 27
aiy.vision.models.dish_classification, 31
aiy.vision.models.dish_detection, 32
aiy.vision.models.face_detection, 32
aiy.vision.models.image_classification, 32
aiy.vision.models.inaturalist_classification, 33
aiy.vision.models.object_detection, 33
aiy.voice.audio, 45
aiy.voice.tts, 49

Index

A

add_order() (*aiy.trackplayer.TrackPlayer method*), 22
add_pattern() (*aiy.trackplayer.TrackPlayer method*), 23
aiy.assistant.auth_helpers (*module*), 41
aiy.assistant.grpc (*module*), 40
aiy.assistant.library (*module*), 41
aiy.board (*module*), 3
aiy.cloudspeech (*module*), 43
aiy.leds (*module*), 5
aiy.pins (*module*), 11
aiy.toneplayer (*module*), 17
aiy.trackplayer (*module*), 19
aiy.vision.annotator (*module*), 25
aiy.vision.inference (*module*), 27
aiy.vision.models.dish_classification (*module*), 31
aiy.vision.models.dish_detection (*module*), 32
aiy.vision.models.face_detection (*module*), 32
aiy.vision.models.image_classification (*module*), 32
aiy.vision.models.inaturalist_classification (*module*), 33
aiy.vision.models.object_detection (*module*), 33
aiy.voice.audio (*module*), 45
aiy.voice.tts (*module*), 49
Annotator (*class in aiy.vision.annotator*), 25
aplay() (*in module aiy.voice.audio*), 46
apply() (*aiy.trackplayer.Arpeggio method*), 19
apply() (*aiy.trackplayer.Command method*), 19
apply() (*aiy.trackplayer.Glissando method*), 19
apply() (*aiy.trackplayer.JumpToPosition method*), 20
apply() (*aiy.trackplayer.NoteOff method*), 20
apply() (*aiy.trackplayer.PulseChange method*), 20
apply() (*aiy.trackplayer.Retrigger method*), 20
apply() (*aiy.trackplayer.SetPulseWidth method*), 21
apply() (*aiy.trackplayer.SetSpeed method*), 21
apply() (*aiy.trackplayer.StopPlaying method*), 21
apply() (*aiy.trackplayer.Vibrato method*), 23
arecord() (*in module aiy.voice.audio*), 45
Arpeggio (*class in aiy.trackplayer*), 19
Assistant (*class in aiy.assistant.library*), 41
AssistantServiceClient (*class in aiy.assistant.grpc*), 40
AssistantServiceClientWithLed (*class in aiy.assistant.grpc*), 40
AudioFormat (*class in aiy.voice.audio*), 48

B

BACKGROUND (*aiy.vision.models.object_detection.Object attribute*), 33
BASE_OCTAVE (*aiy.toneplayer.Note attribute*), 17
BEACON (*aiy.board.Led attribute*), 4
BEACON_DARK (*aiy.board.Led attribute*), 4
BLACK (*aiy.leds.Color attribute*), 5
blend() (*aiy.leds.Color static method*), 6
BLINK (*aiy.board.Led attribute*), 4
blink() (*aiy.leds.Pattern static method*), 8
BLINK_3 (*aiy.board.Led attribute*), 4
BLUE (*aiy.leds.Color attribute*), 5
Board (*class in aiy.board*), 3
bounding_box (*aiy.vision.models.dish_detection.Dish attribute*), 32
bounding_box (*aiy.vision.models.face_detection.Face attribute*), 32
bounding_box() (*aiy.vision.annotator.Annotator method*), 25
breathe() (*aiy.leds.Pattern static method*), 8
brightness() (*aiy.board.Led method*), 4
button (*aiy.board.Board attribute*), 3
Button (*class in aiy.board*), 3
BUTTON_GPIO_PIN (*aiy.pins.aiy.pins attribute*), 12
BUZZER_GPIO_PIN (*aiy.pins.aiy.pins attribute*), 12
bytes_per_second (*aiy.voice.audio.AudioFormat attribute*), 48

BytesPlayer (*class in aiy.voice.audio*), 47

C

camera_inference() (*aiy.vision.inference.InferenceEngine method*), 28
CameraInference (*class in aiy.vision.inference*), 27
CAT (*aiy.vision.models.object_detection.Object attribute*), 33
CD (*aiy.voice.audio.AudioFormat attribute*), 48
clear() (*aiy.vision.annotator.Annotator method*), 25
close() (*aiy.board.Board method*), 3
close() (*aiy.board.Button method*), 3
close() (*aiy.vision.inference.CameraInference method*), 27
close() (*aiy.vision.inference.ImageInference method*), 28
close() (*aiy.vision.inference.InferenceEngine method*), 28
CloudSpeechClient (*class in aiy.cloudspeech*), 43
Color (*class in aiy.leds*), 5
Command (*class in aiy.trackplayer*), 19
COMMAND_RE (*aiy.trackplayer.TrackLoader attribute*), 22
COMMANDS (*aiy.trackplayer.TrackLoader attribute*), 22
compute_graph (*aiy.vision.inference.ModelDescriptor attribute*), 29
compute_graph() (*aiy.vision.models.inaturalist_classification.Model method*), 33
conversation() (*aiy.assistant.grpc.AssistantServiceClient method*), 40
conversation() (*aiy.assistant.grpc.AssistantServiceClientWithLed method*), 40
count (*aiy.vision.inference.CameraInference attribute*), 27
CYAN (*aiy.leds.Color attribute*), 5

D

DECAY (*aiy.board.Led attribute*), 4
Dish (*class in aiy.vision.models.dish_detection*), 32
DOG (*aiy.vision.models.object_detection.Object attribute*), 33
done() (*aiy.voice.audio.Recorder method*), 46

E

EIGHTH (*aiy.toneplayer.Rest attribute*), 17
engine (*aiy.vision.inference.CameraInference attribute*), 27
engine (*aiy.vision.inference.ImageInference attribute*), 28

F

Face (*class in aiy.vision.models.face_detection*), 32
face_score (*aiy.vision.models.face_detection.Face attribute*), 32
FilePlayer (*class in aiy.voice.audio*), 47
FirmwareVersion (*class in aiy.vision.inference*), 27
FirmwareVersionException, 27
FromSparseTensorConfig (*class in aiy.vision.inference*), 27

G

get_assistant_credentials() (*in module aiy.assistant.auth_helpers*), 41
get_camera_state() (*aiy.vision.inference.InferenceEngine method*), 28
get_classes() (*in module aiy.vision.models.dish_classification*), 31
get_classes() (*in module aiy.vision.models.image_classification*), 32
get_classes() (*in module aiy.vision.models.inaturalist_classification*), 33
get_classes_sparse() (*in module aiy.vision.models.image_classification*), 33
get_classes_sparse() (*in module aiy.vision.models.inaturalist_classification*), 33
get_dishes() (*in module aiy.vision.models.dish_detection*), 32
get_faces() (*in module aiy.vision.models.face_detection*), 32
get_firmware_info() (*aiy.vision.inference.InferenceEngine method*), 28
get_inference_state() (*aiy.vision.inference.InferenceEngine method*), 28
get_objects() (*in module aiy.vision.models.object_detection*), 33
get_objects_sparse() (*in module aiy.vision.models.object_detection*), 33
get_system_info() (*aiy.vision.inference.InferenceEngine method*), 28
Glissando (*class in aiy.trackplayer*), 19
GREEN (*aiy.leds.Color attribute*), 5

H

HALF (*aiy.toneplayer.Rest attribute*), 17

I

image_inference() (*aiy.vision.inference.InferenceEngine method*), 28
ImageInference (*class in aiy.vision.inference*), 28
InferenceEngine (*class in aiy.vision.inference*), 28
InferenceException, 29

input_normalizer (*aiy.vision.inference.ModelDescriptor attribute*), 29
input_shape (*aiy.vision.inference.ModelDescriptor attribute*), 29
installed() (*aiy.leds.Leds static method*), 6

J

join() (*aiy.voice.audio.BytesPlayer method*), 47
join() (*aiy.voice.audio.FilePlayer method*), 47
join() (*aiy.voice.audio.Recorder method*), 46
joy_score (*aiy.vision.models.face_detection.Face attribute*), 32
JumpToPosition (*class in aiy.trackplayer*), 20

L

led (*aiy.board.Board attribute*), 3
Led (*class in aiy.board*), 4
LED_1 (*aiy.pins.aiy.pins attribute*), 12
LED_2 (*aiy.pins.aiy.pins attribute*), 12
Leds (*class in aiy.leds*), 6
Leds.Channel (*class in aiy.leds*), 6
load() (*aiy.trackplayer.TrackLoader method*), 22
load_model() (*aiy.vision.inference.InferenceEngine method*), 29
logical_shape (*aiy.vision.inference.FromSparseTensorConfig attribute*), 27
logical_shape (*aiy.vision.inference.ThresholdingConfig attribute*), 29

M

major (*aiy.vision.inference.FirmwareVersion attribute*), 27
minor (*aiy.vision.inference.FirmwareVersion attribute*), 27
Model (*class in aiy.vision.models.inaturalist_classification*), 33
model() (*in module aiy.vision.models.dish_classification*), 31
model() (*in module aiy.vision.models.dish_detection*), 32
model() (*in module aiy.vision.models.face_detection*), 32
model() (*in module aiy.vision.models.image_classification*), 33
model() (*in module aiy.vision.models.inaturalist_classification*), 33
model() (*in module aiy.vision.models.object_detection*), 33
ModelDescriptor (*class in aiy.vision.inference*), 29

N

name (*aiy.vision.inference.ModelDescriptor attribute*), 29
Note (*class in aiy.toneplayer*), 17
NOTE_RE (*aiy.toneplayer.TonePlayer attribute*), 18
NOTE_RE (*aiy.trackplayer.TrackLoader attribute*), 22
NoteOff (*class in aiy.trackplayer*), 20

O

Object (*class in aiy.vision.models.object_detection*), 33
OFF (*aiy.board.Led attribute*), 4
OFF (*aiy.leds.Leds.Channel attribute*), 6
ON (*aiy.board.Led attribute*), 4
ON (*aiy.leds.Leds.Channel attribute*), 6

P

parse() (*aiy.trackplayer.Arpeggio class method*), 19
parse() (*aiy.trackplayer.Command class method*), 19
parse() (*aiy.trackplayer.Glissando class method*), 20
parse() (*aiy.trackplayer.JumpToPosition class method*), 20
parse() (*aiy.trackplayer.NoteOff class method*), 20
parse() (*aiy.trackplayer.PulseChange class method*), 20
parse() (*aiy.trackplayer.Retrigger class method*), 20
parse() (*aiy.trackplayer.SetPulseWidth class method*), 21
parse() (*aiy.trackplayer.SetSpeed class method*), 21
parse() (*aiy.trackplayer.StopPlaying class method*), 21
parse() (*aiy.trackplayer.Vibrato class method*), 23
pattern (*aiy.leds.Leds attribute*), 6
PATTERN (*aiy.leds.Leds.Channel attribute*), 6
Pattern (*class in aiy.leds*), 8
PERIOD_MAP (*aiy.toneplayer.TonePlayer attribute*), 18
PERSON (*aiy.vision.models.object_detection.Object attribute*), 33
PIN_A (*aiy.pins.aiy.pins attribute*), 11
PIN_B (*aiy.pins.aiy.pins attribute*), 11
PIN_C (*aiy.pins.aiy.pins attribute*), 11
PIN_D (*aiy.pins.aiy.pins attribute*), 11
play() (*aiy.toneplayer.TonePlayer method*), 18
play() (*aiy.trackplayer.TrackPlayer method*), 23
play() (*aiy.voice.audio.BytesPlayer method*), 47
play_raw() (*aiy.voice.audio.FilePlayer method*), 47
play_raw() (*in module aiy.voice.audio*), 47
play_raw_async() (*in module aiy.voice.audio*), 47
play_wav() (*aiy.voice.audio.FilePlayer method*), 48
play_wav() (*in module aiy.voice.audio*), 47
play_wav_async() (*in module aiy.voice.audio*), 47
point() (*aiy.vision.annotator.Annotator method*), 26
privacy() (*aiy.leds.Leds static method*), 6
privacy_off() (*aiy.leds.Leds static method*), 7
privacy_on() (*aiy.leds.Leds static method*), 7
PrivacyLed (*class in aiy.leds*), 8
PULSE_QUICK (*aiy.board.Led attribute*), 4
PULSE_SLOW (*aiy.board.Led attribute*), 4
PulseChange (*class in aiy.trackplayer*), 20

PURPLE (*aiy.leds.Color attribute*), 5

Q

QUARTER (*aiy.toneplayer.Rest attribute*), 17

R

rate (*aiy.vision.inference.CameraInference attribute*), 27
recognize() (*aiy.cloudspeech.CloudSpeechClient method*), 43
recognize_bytes() (*aiy.cloudspeech.CloudSpeechClient method*), 44
record() (*aiy.voice.audio.Recorder method*), 46
record_file() (*in module aiy.voice.audio*), 45
record_file_async() (*in module aiy.voice.audio*), 46
Recorder (*class in aiy.voice.audio*), 46
RED (*aiy.leds.Color attribute*), 5
reset() (*aiy.leds.Leds method*), 7
reset() (*aiy.vision.inference.InferenceEngine method*), 29
Rest (*class in aiy.toneplayer*), 17
REST_RE (*aiy.toneplayer.TonePlayer attribute*), 18
Retrigger (*class in aiy.trackplayer*), 20
rgb() (*aiy.leds.Leds static method*), 7
rgb_off() (*aiy.leds.Leds static method*), 7
rgb_on() (*aiy.leds.Leds static method*), 7
rgb_pattern() (*aiy.leds.Leds static method*), 7
RgbLeds (*class in aiy.leds*), 9
run() (*aiy.vision.inference.CameraInference method*), 27
run() (*aiy.vision.inference.ImageInference method*), 28

S

say() (*in module aiy.voice.tts*), 49
set_order() (*aiy.trackplayer.TrackPlayer method*), 23
set_position() (*aiy.trackplayer.TrackPlayer method*), 23
set_speed() (*aiy.trackplayer.TrackPlayer method*), 23
SetPulseWidth (*class in aiy.trackplayer*), 20
SetSpeed (*class in aiy.trackplayer*), 21
SIXTEENTH (*aiy.toneplayer.Rest attribute*), 17
sorted_scores (*aiy.vision.models.dish_detection.Dish attribute*), 32
sparse_configs() (*in module aiy.vision.models.image_classification*), 33
sparse_configs() (*in module aiy.vision.models.inaturalist_classification*), 33
sparse_configs() (*in module aiy.vision.models.object_detection*), 34
squeeze_dims (*aiy.vision.inference.FromSparseTensorConfig attribute*), 27
start_camera_inference() (*aiy.vision.inference.InferenceEngine method*), 29
start_listening() (*aiy.cloudspeech.CloudSpeechClient method*), 44
state (*aiy.board.Led attribute*), 4
stop() (*aiy.trackplayer.TrackPlayer method*), 23
stop() (*aiy.vision.annotator.Annotator method*), 26
stop_camera_inference() (*aiy.vision.inference.InferenceEngine method*), 29
stop_listening() (*aiy.cloudspeech.CloudSpeechClient method*), 44
StopPlaying (*class in aiy.trackplayer*), 21

T

tensor_name (*aiy.vision.inference.FromSparseTensorConfig attribute*), 28
text() (*aiy.vision.annotator.Annotator method*), 26
threshold (*aiy.vision.inference.ThresholdingConfig attribute*), 29
ThresholdingConfig (*class in aiy.vision.inference*), 29
to_frequency() (*aiy.toneplayer.Note method*), 17
to_ignore (*aiy.vision.inference.ThresholdingConfig attribute*), 29
to_length_secs() (*aiy.toneplayer.Rest method*), 18
TonePlayer (*class in aiy.toneplayer*), 18
top_k (*aiy.vision.inference.ThresholdingConfig attribute*), 29
TrackLoader (*class in aiy.trackplayer*), 21
TrackPlayer (*class in aiy.trackplayer*), 22

U

unload_model() (*aiy.vision.inference.InferenceEngine method*), 29
update() (*aiy.leds.Leds method*), 7
update() (*aiy.vision.annotator.Annotator method*), 26

V

Vibrato (*class in aiy.trackplayer*), 23
volume_percentage (*aiy.assistant.grpc.AssistantServiceClient attribute*), 40
volume_percentage (*aiy.assistant.grpc.AssistantServiceClientWithLed attribute*), 41

W

wait_for_press() (*aiy.board.Button method*), 4

wait_for_release() (*aiy.board.Button method*), 4
wave_get_format() (*in module aiy.voice.audio*), 48
wave_set_format() (*in module aiy.voice.audio*), 48
when_pressed (*aiy.board.Button attribute*), 4
when_released (*aiy.board.Button attribute*), 4
WHITE (*aiy.leds.Color attribute*), 6
WHOLE (*aiy.toneplayer.Rest attribute*), 17

Y

YELLOW (*aiy.leds.Color attribute*), 6