3. Basic Recipes

The following recipes should be reasonably accessible to Python programmers of all skill levels. Please feel free to suggest enhancements or additional recipes.

3.1. Capturing to a file

Capturing an image to a file is as simple as specifying the name of the file as the output of whatever `capture()` method you require:

```
from time import sleep
from picamera import PiCamera

camera = PiCamera()
camera.resolution = (1024, 768)
camera.start_preview()
# Camera warm-up time
sleep(2)
camera.capture('foo.jpg')
```

Note that files opened by picamera (as in the case above) will be flushed and closed so that when the capture method returns, the data should be accessible to other processes.

3.2. Capturing to a stream

Capturing an image to a file-like object (a `socket()`, a `io.BytesIO` stream, an existing open file object, etc.) is as simple as specifying that object as the output of whatever `capture()` method you're using:

```
from io import BytesIO
from time import sleep
from picamera import PiCamera

# Create an in-memory stream
my_stream = BytesIO()
camera = PiCamera()
camera.start_preview()
# Camera warm-up time
sleep(2)
camera.capture(my_stream, 'jpeg')
```

Note that the format is explicitly specified in the case above. The `BytesIO` object has no filename, so the camera can't automatically figure out what format to use.

One thing to bear in mind is that (unlike specifying a filename), the stream is *not* automatically closed after capture; picamera assumes that since it didn't open the stream it can't presume to close it either. However, if the object has a `flush` method, this will be called prior to capture returning. This should ensure that once capture returns the data is accessible to other processes although the object still needs to be closed:

```
from time import sleep
from picamera import PiCamera

# Explicitly open a new file called my_image.jpg
my_file = open('my_image.jpg', 'wb')
camera = PiCamera()
camera.start_preview()
sleep(2)
camera.capture(my_file)
# At this point my_file.flush() has been called, but the file has
# not yet been closed
my_file.close()
```

Note that in the case above, we didn't have to specify the format as the camera interrogated the `my_file` object for its filename (specifically, it looks for a `name` attribute on the provided object). As well as using stream classes built into Python (like `BytesIO`) you can also construct your own custom outputs.

3.3. Capturing to a PIL Image

This is a variation on Capturing to a stream. First we'll capture an image to a `BytesIO` stream (Python's in-memory stream class), then we'll rewind the position of the stream to the start, and read the stream into a PIL Image object:

```
from io import BytesIO
from time import sleep
from picamera import PiCamera
from PIL import Image

# Create the in-memory stream
stream = BytesIO()
camera = PiCamera()
camera.start_preview()
sleep(2)
camera.capture(stream, format='jpeg')
# "Rewind" the stream to the beginning so we can read its content
stream.seek(0)
image = Image.open(stream)
```

3.4. Capturing resized images

Sometimes, particularly in scripts which will perform some sort of analysis or processing on images, you may wish to capture smaller images than the current resolution of the camera. Although such resizing can be performed using libraries like PIL or OpenCV, it is considerably more efficient to have the Pi's GPU perform the resizing when capturing the image. This can be done with the *resize* parameter of the `capture()` methods:

```
from time import sleep
from picamera import PiCamera

camera = PiCamera()
camera.resolution = (1024, 768)
camera.start_preview()
# Camera warm-up time
sleep(2)
camera.capture('foo.jpg', resize=(320, 240))
```

The *resize* parameter can also be specified when recording video with the `start_recording()` method.

3.5. Capturing consistent images

You may wish to capture a sequence of images all of which look the same in terms of brightness, color, and contrast (this can be useful in timelapse photography, for example). Various attributes need to be used in order to ensure consistency across multiple shots. Specifically, you need to ensure that the camera's exposure time, white balance, and gains are all fixed:

- To fix exposure time, set the `shutter_speed` attribute to a reasonable value.
- Optionally, set `iso` to a fixed value.
- To fix exposure gains, let `analog_gain` and `digital_gain` settle on reasonable values, then set `exposure_mode` to `'off'`.
- To fix white balance, set the `awb_mode` to `'off'`, then set `awb_gains` to a (red, blue) tuple of gains.

It can be difficult to know what appropriate values might be for these attributes. For `iso`, a simple rule of thumb is that 100 and 200 are reasonable values for daytime, while 400 and 800 are better for low light. To determine a reasonable value for `shutter_speed` you can query the `exposure_speed` attribute. For exposure gains, it's usually enough to wait until `analog_gain` is greater than 1 (the default, which will produce entirely black frames) before `exposure_mode` is set to `'off'`. Finally, to determine reasonable values for `awb_gains` simply query the property while `awb_mode` is set to something other than `'off'`. Again, this will tell you the camera's white balance gains as determined by the auto-white-balance algorithm.

The following script provides a brief example of configuring these settings:

```
from time import sleep
from picamera import PiCamera

camera = PiCamera(resolution=(1280, 720), framerate=30)
# Set ISO to the desired value
camera.iso = 100
# Wait for the automatic gain control to settle
sleep(2)
# Now fix the values
camera.shutter_speed = camera.exposure_speed
camera.exposure_mode = 'off'
g = camera.awb_gains
camera.awb_mode = 'off'
camera.awb_gains = g
# Finally, take several photos with the fixed settings
camera.capture_sequence(['image%02d.jpg' % i for i in range(10)])
```

3.6. Capturing timelapse sequences

The simplest way to capture long time-lapse sequences is with the `capture_continuous()` method. With this method, the camera captures images continually until you tell it to stop. Images are automatically given unique names and you can easily control the delay between captures. The following example shows how to capture images with a 5 minute delay between each shot:

```
from time import sleep
from picamera import PiCamera

camera = PiCamera()
camera.start_preview()
sleep(2)
for filename in camera.capture_continuous('img{counter:03d}.jpg'):
    print('Captured %s' % filename)
    sleep(300) # wait 5 minutes
```

However, you may wish to capture images at a particular time, say at the start of every hour. This simply requires a refinement of the delay in the loop (the `datetime` module is slightly easier to use for calculating dates and times; this example also demonstrates the `timestamp` template in the captured filenames):

```
from time import sleep
from picamera import PiCamera
from datetime import datetime, timedelta

def wait():
    # Calculate the delay to the start of the next hour
    next_hour = (datetime.now() + timedelta(hour=1)).replace(
        minute=0, second=0, microsecond=0)
    delay = (next_hour - datetime.now()).seconds
    sleep(delay)

camera = PiCamera()
camera.start_preview()
wait()
for filename in camera.capture_continuous('img{timestamp:%Y-%m-%d-%H-%M}.jpg'):
    print('Captured %s' % filename)
    wait()
```

3.7. Capturing in low light

Using similar tricks to those in Capturing consistent images, the Pi's camera can capture images in low light conditions. The primary objective is to set a high gain, and a long exposure time to allow the camera to gather as much light as possible. However, the `shutter_speed` attribute is constrained by the camera's `framerate` so the first thing we need to do is set a very slow framerate. The following script captures an image with a 6 second exposure time (the maximum the Pi's V1 camera module is capable of; the V2 camera module can manage 10 second exposures):

```
from picamera import PiCamera
from time import sleep
from fractions import Fraction

# Set a framerate of 1/6fps, then set shutter
# speed to 6s and ISO to 800
camera = PiCamera(resolution=(1280, 720), framerate=Fraction(1, 6))
camera.shutter_speed = 6000000
camera.iso = 800
# Give the camera a good long time to set gains and
# measure AWB (you may wish to use fixed AWB instead)
sleep(30)
camera.exposure_mode = 'off'
# Finally, capture an image with a 6s exposure. Due
# to mode switching on the still port, this will take
# longer than 6 seconds
camera.capture('dark.jpg')
```

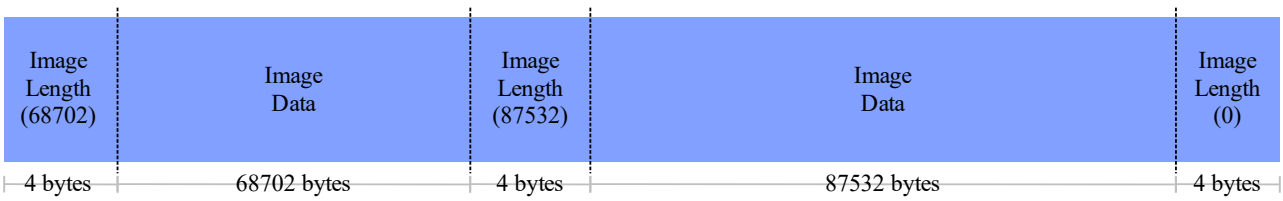
In anything other than dark conditions, the image produced by this script will most likely be completely white or at least heavily over-exposed.

Note

The Pi's camera module uses a rolling shutter. This means that moving subjects may appear distorted if they move relative to the camera. This effect will be exaggerated by using longer exposure times.

3.8. Capturing to a network stream

This is a variation of Capturing timelapse sequences. Here we have two scripts: a server (presumably on a fast machine) which listens for a connection from the Raspberry Pi, and a client which runs on the Raspberry Pi and sends a continual stream of images to the server. We'll use a very simple protocol for communication: first the length of the image will be sent as a 32-bit integer (in Little Endian format), then this will be followed by the bytes of image data. If the length is 0, this indicates that the connection should be closed as no more images will be forthcoming. This protocol is illustrated below:



Firstly the server script (which relies on PIL for reading JPEGs, but you could replace this with any other suitable graphics library, e.g. OpenCV or GraphicsMagick):

```
import io
import socket
import struct
from PIL import Image

# Start a socket listening for connections on 0.0.0.0:8000 (0.0.0.0 means
# all interfaces)
server_socket = socket.socket()
server_socket.bind(('0.0.0.0', 8000))
server_socket.listen(0)

# Accept a single connection and make a file-like object out of it
connection = server_socket.accept()[0].makefile('rb')
try:
    while True:
        # Read the length of the image as a 32-bit unsigned int. If the
        # length is zero, quit the loop
        image_len = struct.unpack('<L', connection.read(struct.calcsize('<L')))[0]
        if not image_len:
            break
        # Construct a stream to hold the image data and read the image
        # data from the connection
        image_stream = io.BytesIO()
        image_stream.write(connection.read(image_len))
        # Rewind the stream, open it as an image with PIL and do some
        # processing on it
        image_stream.seek(0)
        image = Image.open(image_stream)
        print('Image is %dx%d' % image.size)
        image.verify()
        print('Image is verified')
finally:
    connection.close()
    server_socket.close()
```

Now for the client side of things, on the Raspberry Pi:

```
import io
import socket
import struct
import time
import picamera

# Connect a client socket to my_server:8000 (change my_server to the
# hostname of your server)
client_socket = socket.socket()
client_socket.connect(('my_server', 8000))

# Make a file-like object out of the connection
connection = client_socket.makefile('wb')
try:
    camera = picamera.PiCamera()
    camera.resolution = (640, 480)
    # Start a preview and let the camera warm up for 2 seconds
    camera.start_preview()
    time.sleep(2)

    # Note the start time and construct a stream to hold image data
    # temporarily (we could write it directly to connection but in this
    # case we want to find out the size of each capture first to keep
    # our protocol simple)
    start = time.time()
    stream = io.BytesIO()
    for foo in camera.capture_continuous(stream, 'jpeg'):
        # Write the length of the capture to the stream and flush to
        # ensure it actually gets sent
        connection.write(struct.pack('<L', stream.tell()))
        connection.flush()
        # Rewind the stream and send the image data over the wire
        stream.seek(0)
        connection.write(stream.read())
        # If we've been capturing for more than 30 seconds, quit
        if time.time() - start > 30:
            break
        # Reset the stream for the next capture
        stream.seek(0)
        stream.truncate()
    # Write a length of zero to the stream to signal we're done
    connection.write(struct.pack('<L', 0))
finally:
    connection.close()
    client_socket.close()
```

The server script should be run first to ensure there's a listening socket ready to accept a connection from the client script.

3.9. Recording video to a file

Recording a video to a file is simple:

```
import picamera

camera = picamera.PiCamera()
camera.resolution = (640, 480)
camera.start_recording('my_video.h264')
camera.wait_recording(60)
camera.stop_recording()
```

Note that we use `wait_recording()` in the example above instead of `time.sleep()` which we've been using in the image capture recipes above. The `wait_recording()` method is similar in that it will pause for the number of seconds specified, but unlike `time.sleep()` it will continually check for recording errors (e.g. an out of disk space condition) while it is waiting. If we had used `time.sleep()` instead, such errors would only be raised by the `stop_recording()` call (which could be long after the error actually occurred).

3.10. Recording video to a stream

This is very similar to Recording video to a file:

```
from io import BytesIO
from picamera import PiCamera

stream = BytesIO()
camera = PiCamera()
camera.resolution = (640, 480)
camera.start_recording(stream, format='h264', quality=23)
camera.wait_recording(15)
camera.stop_recording()
```

Here, we've set the *quality* parameter to indicate to the encoder the level of image quality that we'd like it to try and maintain. The camera's H.264 encoder is primarily constrained by two parameters:

- *bitrate* limits the encoder's output to a certain number of bits per second. The default is 17000000 (17Mbps), and the maximum value is 25000000 (25Mbps). Higher values give the encoder more "freedom" to encode at higher qualities. You will likely find that the default doesn't constrain the encoder at all except at higher recording resolutions.
- *quality* tells the encoder what level of image quality to maintain. Values can be between 1 (highest quality) and 40 (lowest quality), with typical values providing a reasonable trade-off between bandwidth and quality being between 20 and 25.

As well as using stream classes built into Python (like `BytesIO`) you can also construct your own custom outputs. This is particularly useful for video recording, as discussed in the linked recipe.

3.11. Recording over multiple files

If you wish split your recording over multiple files, you can use the `split_recording()` method to accomplish this:

```
import picamera

camera = picamera.PiCamera(resolution=(640, 480))
camera.start_recording('1.h264')
camera.wait_recording(5)
for i in range(2, 11):
    camera.split_recording('%d.h264' % i)
    camera.wait_recording(5)
camera.stop_recording()
```

This should produce 10 video files named `1.h264`, `2.h264`, etc. each of which is approximately 5 seconds long (approximately because the `split_recording()` method will only split files at a key-frame).

The `record_sequence()` method can also be used to achieve this with slightly cleaner code:

```
import picamera

camera = picamera.PiCamera(resolution=(640, 480))
for filename in camera.record_sequence(
        '%d.h264' % i for i in range(1, 11)):
    camera.wait_recording(5)
```

Changed in version 1.3: The `record_sequence()` method was introduced in version 1.3

3.12. Recording to a circular stream

This is similar to Recording video to a stream but uses a special kind of in-memory stream provided by the picamera library. The `PiCameraCircularIO` class implements a ring buffer based stream, specifically for video recording. This enables you to keep an in-memory stream containing the last *n* seconds of video recorded (where *n* is determined by the bitrate of the video recording and the size of the ring buffer underlying the stream).

A typical use-case for this sort of storage is security applications where one wishes to detect motion and only record to disk the video where motion was detected. This example keeps 20 seconds of video in memory until the `write_now` function returns `True` (in this implementation this is random but one could, for example, replace this with some sort of motion detection

algorithm). Once `write_now` returns `True`, the script waits 10 more seconds (so that the buffer contains 10 seconds of video from before the event, and 10 seconds after) and writes the resulting video to disk before going back to waiting:

```
import io
import random
import picamera

def motion_detected():
    # Randomly return True (like a fake motion detection routine)
    return random.randint(0, 10) == 0

camera = picamera.PiCamera()
stream = picamera.PiCameraCircularIO(camera, seconds=20)
camera.start_recording(stream, format='h264')
try:
    while True:
        camera.wait_recording(1)
        if motion_detected():
            # Keep recording for 10 seconds and only then write the
            # stream to disk
            camera.wait_recording(10)
            stream.copy_to('motion.h264')
finally:
    camera.stop_recording()
```

In the above script we use the special `copy_to()` method to copy the stream to a disk file. This automatically handles details like finding the start of the first key-frame in the circular buffer, and also provides facilities like writing a specific number of bytes or seconds.

Note

Note that *at least* 20 seconds of video are in the stream. This is an estimate only; if the H.264 encoder requires less than the specified bitrate (17Mbps by default) for recording the video, then more than 20 seconds of video will be available in the stream.

New in version 1.0.

Changed in version 1.11: Added use of the `copy_to()`

3.13. Recording to a network stream

This is similar to Recording video to a stream but instead of an in-memory stream like `BytesIO`, we will use a file-like object created from a `socket()`. Unlike the example in Capturing to a network stream we don't need to complicate our network protocol by writing things like the

length of images. This time we're sending a continual stream of video frames (which necessarily incorporates such information, albeit in a much more efficient form), so we can simply dump the recording straight to the network socket.

Firstly, the server side script which will simply read the video stream and pipe it to a media player for display:

```
import socket
import subprocess

# Start a socket listening for connections on 0.0.0.0:8000 (0.0.0.0 means
# all interfaces)
server_socket = socket.socket()
server_socket.bind(('0.0.0.0', 8000))
server_socket.listen(0)

# Accept a single connection and make a file-like object out of it
connection = server_socket.accept()[0].makefile('rb')
try:
    # Run a viewer with an appropriate command line. Uncomment the mplayer
    # version if you would prefer to use mplayer instead of VLC
    cmdline = ['vlc', '--demux', 'h264', '-']
    #cmdline = ['mplayer', '-fps', '25', '-cache', '1024', '-']
    player = subprocess.Popen(cmdline, stdin=subprocess.PIPE)
    while True:
        # Repeatedly read 1k of data from the connection and write it to
        # the media player's stdin
        data = connection.read(1024)
        if not data:
            break
        player.stdin.write(data)
finally:
    connection.close()
    server_socket.close()
    player.terminate()
```

Note

If you run this script on Windows you will probably need to provide a complete path to the VLC or mplayer executable. If you run this script on Mac OS X, and are using Python installed from MacPorts, please ensure you have also installed VLC or mplayer from MacPorts.

You will probably notice several seconds of latency with this setup. This is normal and is because media players buffer several seconds to guard against unreliable network streams. Some media players (notably mplayer in this case) permit the user to skip to the end of the buffer (press the right cursor key in mplayer), reducing the latency by increasing the risk that delayed / dropped network packets will interrupt the playback.

Now for the client side script which simply starts a recording over a file-like object created from the network socket:

```
import socket
import time
import picamera

# Connect a client socket to my_server:8000 (change my_server to the
# hostname of your server)
client_socket = socket.socket()
client_socket.connect(('my_server', 8000))

# Make a file-like object out of the connection
connection = client_socket.makefile('wb')
try:
    camera = picamera.PiCamera()
    camera.resolution = (640, 480)
    camera.framerate = 24
    # Start a preview and let the camera warm up for 2 seconds
    camera.start_preview()
    time.sleep(2)
    # Start recording, sending the output to the connection for 60
    # seconds, then stop
    camera.start_recording(connection, format='h264')
    camera.wait_recording(60)
    camera.stop_recording()
finally:
    connection.close()
    client_socket.close()
```

It should also be noted that the effect of the above is much more easily achieved (at least on Linux) with a combination of `netcat` and the `raspivid` executable. For example:

```
server-side: nc -l 8000 | vlc --demux h264 -
client-side: raspivid -w 640 -h 480 -t 60000 -o - | nc my_server 8000
```

However, this recipe does serve as a starting point for video streaming applications. It's also possible to reverse the direction of this recipe relatively easily. In this scenario, the Pi acts as the server, waiting for a connection from the client. When it accepts a connection, it starts streaming video over it for 60 seconds. Another variation (just for the purposes of demonstration) is that we initialize the camera straight away instead of waiting for a connection to allow the streaming to start faster on connection:

```
import socket
import time
import picamera

camera = picamera.PiCamera()
camera.resolution = (640, 480)
camera.framerate = 24

server_socket = socket.socket()
server_socket.bind(('0.0.0.0', 8000))
server_socket.listen(0)

# Accept a single connection and make a file-like object out of it
connection = server_socket.accept()[0].makefile('wb')
try:
    camera.start_recording(connection, format='h264')
    camera.wait_recording(60)
    camera.stop_recording()
finally:
    connection.close()
    server_socket.close()
```

One advantage of this setup is that no script is needed on the client side - we can simply use VLC with a network URL:

```
vlc tcp/h264://my_pi_address:8000/
```

Note

VLC (or mplayer) will *not* work for playback on a Pi. Neither is (currently) capable of using the GPU for decoding, and thus they attempt to perform video decoding on the Pi's CPU (which is not powerful enough for the task). You will need to run these applications on a faster machine (though "faster" is a relative term here: even an Atom powered netbook should be quick enough for the task at non-HD resolutions).

3.14. Overlaying images on the preview

The camera preview system can operate multiple layered renderers simultaneously. While the picamera library only permits a single renderer to be connected to the camera's preview port, it does permit additional renderers to be created which display a static image. These overlaid renderers can be used to create simple user interfaces.

Note

> Overlay images will *not* appear in image captures or video recordings. If you need to embed additional information in the output of the camera, please refer to Overlaying text on the output.

One difficulty of working with overlay renderers is that they expect unencoded RGB input which is padded up to the camera's block size. The camera's block size is 32x16 so any image data provided to a renderer must have a width which is a multiple of 32, and a height which is a multiple of 16. The specific RGB format expected is interleaved unsigned bytes. If all this sounds complicated, don't worry; it's quite simple to produce in practice.

The following example demonstrates loading an arbitrary size image with PIL, padding it to the required size, and producing the unencoded RGB data for the call to `add_overlay()`:

```
import picamera
from PIL import Image
from time import sleep

camera = picamera.PiCamera()
camera.resolution = (1280, 720)
camera.framerate = 24
camera.start_preview()

# Load the arbitrarily sized image
img = Image.open('overlay.png')
# Create an image padded to the required size with
# mode 'RGB'
pad = Image.new('RGB', (
    ((img.size[0] + 31) // 32) * 32,
    ((img.size[1] + 15) // 16) * 16,
    ))
# Paste the original image into the padded one
pad.paste(img, (0, 0))

# Add the overlay with the padded image as the source,
# but the original image's dimensions
o = camera.add_overlay(pad.tostring(), size=img.size)
# By default, the overlay is in layer 0, beneath the
# preview (which defaults to layer 2). Here we make
# the new overlay semi-transparent, then move it above
# the preview
o.alpha = 128
o.layer = 3

# Wait indefinitely until the user terminates the script
while True:
    sleep(1)
```

Alternatively, instead of using an image file as the source, you can produce an overlay directly from a numpy array. In the following example, we construct a numpy array with the same resolution as the screen, then draw a white cross through the center and overlay it on the preview as a simple cross-hair:

```
import time
import picamera
import numpy as np

# Create an array representing a 1280x720 image of
# a cross through the center of the display. The shape of
# the array must be of the form (height, width, color)
a = np.zeros((720, 1280, 3), dtype=np.uint8)
a[360, :, :] = 0xff
a[:, 640, :] = 0xff

camera = picamera.PiCamera()
camera.resolution = (1280, 720)
camera.framerate = 24
camera.start_preview()
# Add the overlay directly into layer 3 with transparency;
# we can omit the size parameter of add_overlay as the
# size is the same as the camera's resolution
o = camera.add_overlay(np.getbuffer(a), layer=3, alpha=64)
try:
    # Wait indefinitely until the user terminates the script
    while True:
        time.sleep(1)
finally:
    camera.remove_overlay(o)
```

Given that overlaid renderers can be hidden (by moving them below the preview's `layer` which defaults to 2), made semi-transparent (with the `alpha` property), and resized so that they don't `fill the screen`, they can be used to construct simple user interfaces.

New in version 1.8.

3.15. Overlaying text on the output

The camera includes a rudimentary annotation facility which permits up to 255 characters of ASCII text to be overlaid on all output (including the preview, image captures and video recordings). To achieve this, simply assign a string to the `annotate_text` attribute:

```
import picamera
import time

camera = picamera.PiCamera()
camera.resolution = (640, 480)
camera.framerate = 24
camera.start_preview()
camera.annotate_text = 'Hello world!'
time.sleep(2)
# Take a picture including the annotation
camera.capture('foo.jpg')
```

With a little ingenuity, it's possible to display longer strings:

```
import picamera
import time
import itertools

s = "This message would be far too long to display normally..."

camera = picamera.PiCamera()
camera.resolution = (640, 480)
camera.framerate = 24
camera.start_preview()
camera.annotate_text = ' ' * 31
for c in itertools.cycle(s):
    camera.annotate_text = camera.annotate_text[1:31] + c
    time.sleep(0.1)
```

And of course, it can be used to display (and embed) a timestamp in recordings (this recipe also demonstrates drawing a background behind the timestamp for contrast with the `annotate_background` attribute):

```
import picamera
import datetime as dt

camera = picamera.PiCamera(resolution=(1280, 720), framerate=24)
camera.start_preview()
camera.annotate_background = picamera.Color('black')
camera.annotate_text = dt.datetime.now().strftime('%Y-%m-%d %H:%M:%S')
camera.start_recording('timestamped.h264')
start = dt.datetime.now()
while (dt.datetime.now() - start).seconds < 30:
    camera.annotate_text = dt.datetime.now().strftime('%Y-%m-%d %H:%M:%S')
    camera.wait_recording(0.2)
camera.stop_recording()
```

New in version 1.7.

3.16. Controlling the LED

In certain circumstances, you may find the camera module's red LED a hindrance. For example, in the case of automated close-up wild-life photography, the LED may scare off animals. It can also cause unwanted reflected red glare with close-up subjects.

One trivial way to deal with this is simply to place some opaque covering on the LED (e.g. blue-tack or electricians tape). Another method is to use the `disable_camera_led` option in the boot configuration.

However, provided you have the RPi.GPIO package installed, and provided your Python process is running with sufficient privileges (typically this means running as root with `sudo python`), you can also control the LED via the `led` attribute:

```
import picamera

camera = picamera.PiCamera()
# Turn the camera's LED off
camera.led = False
# Take a picture while the LED remains off
camera.capture('foo.jpg')
```

Warning

Be aware when you first use the LED property it will set the GPIO library to Broadcom (BCM) mode with `GPIO.setmode(GPIO.BCM)` and disable warnings with `GPIO.setwarnings(False)`. The LED cannot be controlled when the library is in BOARD mode.