# pyrana Documentation

***Release 0.6***

**Francesco Romani**

January 05, 2015

Contents

Pyrana is a pure-python package which provides easy, pythonic and powerful handling of multimedia files.

- easy: pyrana does not submerge you with tons of options and details, but is filled with sane defaults. pyrana aims to be multimedia processing what requests is for http.
- pythonic: pyrana wants to play nice and work well with the other well-established relevant python packages: Pillow, pygame, PySDL2, PyAudio, numpy compatibility is coming soon.
- powerful: pyrana provides an independent API, but is built on the great foundations provided by the powerful FFMpeg libraries.

pyrana is a modern, pure python package which is developed for python 3 and compatible with python 2.7, which takes great advantage of CFFI, so the compatibility with pypy is just one step away.

pyrana offers a minimum 100% unit-test and documentation coverage, and put great emphasis on small, yet complete and workable examples. Last but not least, pyrana is released under the very liberal ZLIB license.

More documentation about pyrana on this series of blog posts

CHAPTER 1

# Contents

## 1.1 Tutorial

This is a multi-part tutorial about how to write a simple yet complete and fully functional media player using pyrana. The structure of the tutorial intentionally resemples as closely as possible the FFmpeg tutorial.



### 1.1.1 Overview

Movie files have a few basic components. First, the file itself is called a *container*, and the type of container determines where the information in the file goes. Examples of containers are AVI, Quicktime, Matroska (MKV) or ogg. Next, you have a bunch of *streams*; for example, you usually have an audio stream and a video stream. (A "stream" is just a fancy word for "a succession of data elements made available over time".) The data elements in a stream are called *frames*. Each stream is encoded by a different kind of **codec**. The codec defines how the actual data is COded and DECoded - hence the name CODEC. Examples of codecs are WEBM, H.264, MP3 or Vorbis. *Packets* are then read from the stream. Packets are pieces of data that can contain bits of data that are decoded into raw frames that we can finally manipulate for our application. For our purposes, each packet contains complete frames, or multiple frames in the case of audio.

At its very basic level, dealing with video and audio streams is very easy:

```
with open_stream("video.ogg") as video:
    frame = video.read_packet()
if not frame.complete:
        continue
    do_something(frame)
```

We will see soon enough that in real python code, thanks to pyrana, the real code is not very different from this pseudo code above. However, some programs might have a very complex `do_something` step. So in this tutorial, we're going to open a file, read from the video stream inside it, and our `do_something` is going to be writing the frame to a PPM file.

### 1.1.2 Opening the File

First, let's see how we open a file in the first place. With pyrana, you have to first initialize the library.

```
import pyrana

# somewhere, once per run

pyrana.setup()
```

This registers all available file formats and codecs with the library so they will be used automatically when a file with the corresponding format/codec is opened. Note that you only need to call pyrana.setup() once, but it is safe to do it multiple times, if you cannot avoid it.

Now we can actually open the media file:

```
with open(sys.argv[1], "rb") as src:
    dmx = pyrana.Demuxer(src)
```

Here `dmx` is one of the most common shortcut names for `demuxer`. We get our filename from the first argument. The Demuxer instance needs a valid, already open, binary data provider to be used as underlying source of data. Now you can access the stream informations using the `streams` attribute. `demuxer.streams` is just a collections of data structures, so let's find the zero-th (aka the first) video stream in the collection.

```
from pyrana.formats import find_stream, MediaType
# ...
sid = find_stream(demuxer.streams,
                  0,
                  MediaType.AVMEDIA_TYPE_VIDEO)
# sid: Stream ID
vstream = dmx.streams[sid]
```

Now we can have all the available metadata about the stream (e.g. width and height for a video stream, channel count and bytes per sample for an audio stream). However, we still need a decoder for that video stream. Simple enough:

```
vdec = dmx.open_decoder(sid)
```

Simple as that! now `codec` is ready to roll and decode the frames that will be sent to it.

### 1.1.3 Reading the Data

What we're going to do is read through the entire video stream by reading in the packet, decoding it into our frame, and once our frame is complete, we will convert and save it.

Since we're planning to output PPM files, which are stored in 24-bit RGB, we're going to have to convert our frame from its native format to RGB. pyrana will do these conversions for us. For most projects (including ours) we're going to want to convert our initial frame to a specific format. It's enough to ask an Image from a (video) Frame using the image() method and specifying the desired pixel format. The default value for image() is the same as the video stream.

```
with open(sys.argv[1], "rb") as src:
   dmx = pyrana.formats.Demuxer(src)
   sid = pyrana.formats.find_stream(dmx.streams,
                                    0,
                                    MediaType.AVMEDIA_TYPE_VIDEO)
   num = 0
   vdec = dmx.open_decoder(sid)
   frame = vdec.decode(dmx.stream(sid))
   image = frame.image(pyrana.video.PixelFormat.AV_PIX_FMT_RGB24)
```

### 1.1.4 A note on packets

Technically a packet can contain partial frames or other bits of data, but pyrana's Demuxers (thanks to the ffmpeg libraries) ensures that the packets we get contain either complete or multiple frames.

The process, again, is simple: Demuxer.read_frame() reads in a packet and stores it in a Packet object. Decoder.decode() converts the packet to a frame for us. However, we might not have all the information we need for a frame after decoding a packet, so Decoder.decode() raises NeedFeedError if it is not able to decode the next frame. Finally, we use Image.convert() to convert from the native format (Image.pixel_format) to RGB.

Now all we need to do is make the save_frame function to write the RGB information to a file in PPM format. We're going to be kind of sketchy on the PPM format itself; trust us, it works.

```
def ppm_write(frame, seqno):
    """
    saves a raw frame in a PPM image. See man ppm for details.
    the 'seqno' parameter is just to avoid to overwrite them without
    getting too fancy with the filename generation.
    """
    image = frame.image(pyrana.video.PixelFormat.AV_PIX_FMT_RGB24)
    with open("frame%d.ppm" % (seqno), "wb") as dst:
        header = "P6\n%i %i\n255\n" % (image.width, image.height)
        dst.write(header.encode("utf-8"))
        dst.write(bytes(image))
```

In this case we require the full frame (not just the image) to be passed to be sure to get an image with the conformant Pixel Format. We do a bit of standard file opening, etc., and then write the RGB data. We write the file one line at a time. A PPM file is simply a file that has RGB information laid out in a long string. If you know HTML colors, it would be like laying out the color of each pixel end to end like #ff0000#ff0000.... would be a red screen. (It's stored in binary and without the separator, but you get the idea.) The header indicated how wide and tall the image is, and the max size of the RGB values.

Most image programs should be able to open PPM files. Test it on some movie files.

The full working code used in this post is available here.

### 1.1.5 Pygame and Video

To draw to the screen, we're going to use pygame. pygame is an excellent and well known module which advertises itself as

```
Pygame is a set of Python modules designed for writing games. Pygame adds functionality
on top of the excellent SDL library. This allows you to create fully featured games and
multimedia programs in the python language. Pygame is highly portable and runs on
nearly every platform and operating system.
```

You can get the package at the official website or on PyPI.

Pygame has various methods for drawing images to the screen, and it has one in particularly well suited for displaying movies on the screen - what it calls a YUV overlay. YUV (technically not YUV but YCbCr: generally speaking, YUV is an analog format and YCbCr is a digital format. However, they are often -and incorrectly- used as synonims) is a way of storing raw image data like RGB. Roughly speaking, Y is the brightness (or "luma") component, and U and V are the color components. pygame's YUV overlay takes in a triplet of bytes (strings in py2.x) containing the YUV data and displays it. It accepts an handful of different kinds of YUV formats, but YV12 is most often the fastest. There is another YUV format called YUV420P that is the same as YV12, except the U and V arrays are switched. The 420 means it is subsampled, at a ratio of 4:2:0, basically meaning there is 1 color sample for every 4 luma samples, so the color information is quartered. This is a good way of saving bandwidth, as the human eye does not percieve this change. The "P" in the name means that the format is "planar" – simply meaning that the Y, U, and V components are

in separate arrays. pyrana can convert images to YUV420P, with the added bonus that many video streams are in that format already, or are easily converted to that format.

So our current plan is to replace the `ppm_write` function from part 1, and instead output our frame to the screen. But first we have to start by seeing how to use the pygame package. First we have to import and to initialize it, once again one time only:

```
import pygame
...
pygame.init()
```

### 1.1.6 Creating and using an Overlay

Now we need a place on the screen to put stuff. The basic area for displaying images with SDL is called an *overlay*:

```
pygame.display.set_mode((width, height))
self._ovl = pygame.Overlay(pygame.YV12_OVERLAY, (width, height))
self._ovl.set_location(0, 0, wwidth, height)
```

As we said before, we are using YV12 to display the image:

```
self._ovl.display((Y, U, V))
```

The overlay object takes care of locking, so we don't have to. The `Y`, `U` and `V` objects are bytes() (strings in py2.x) filled with the actual data to display. Of course since we are dealing with YUV420P here, we only have 3 channels, and therefore only 3 sets of data. Other formats might have a fourth pointer for an alpha channel or something.

The code which draws using the pygame overlays can be packed in an handy class:

```
class PygameViewer(object):
def __init__(self):
    self._ovl = None
    self._frames = 0

@property
def frames(self):
    return self._frames

def setup(self, w, h):
    pygame.display.set_mode((w, h))
    self._ovl = pygame.Overlay(pygame.YV12_OVERLAY, (w, h))
    self._ovl.set_location(0, 0, w, h)

def show(self, Y, U, V):
    self._ovl.display((Y, U, V))
    self._frames += 1
```

### 1.1.7 Drawing the Image

What is stil left is to fetch the plane data and pass it to pygame's overlay in order to actually display it. No worries, this is very simple as well:

```
while True:
    frame = vdec.decode(dmx.stream(sid))
    img = frame.image()
    view.show(img.plane(0), img.plane(1), img.plane(2))
```

Where `view` is of course an instance -already set up and ready- of a `PygameViewer` defined above. This is actually the whole decoding loop! The Image objects provides an handy `plane()` method with just returns the `bytes()` of the selected plane.

What happens when you run this program? The video is going crazy! In fact, we're just displaying all the video frames as fast as we can extract them from the movie file. We don't have any code right now for figuring out *when* we need to display video. Eventually (in part 5), we'll get around to syncing the video. But first we're missing something even more important: sound!

The full working code (well, a slightly enhanced version of) used in this post is available here.

## 1.2 errors

The pyrana exception hierarchy. Outside the pyrana package it is expected to catch those exception, not to raise them. However, doing so should'nt harm anyone.

**exception** `pyrana.errors.`**`EOSError`**
End Of Stream. Kinda more akin to StopIteration than EOFError.

**exception** `pyrana.errors.`**`LibraryVersionError`**
Missing the right library version for the expected dependency.

**exception** `pyrana.errors.`**`NeedFeedError`**
More data is needed to obtain a Frame or a Packet. Feed more data in the raising object and try again.

**exception** `pyrana.errors.`**`NotFoundError`**
cannot satisfy the user request: asked for an inexistent attribute or for unsupported parameter combination.

**exception** `pyrana.errors.`**`ProcessingError`**
Runtime processing error.

**exception** `pyrana.errors.`**`PyranaError`**
Root of the pyrana error tree. You should'nt use it directly, not even in an except clause.

**exception** `pyrana.errors.`**`SetupError`**
Error while setting up a pyrana object. Check again the parameters.

**exception** `pyrana.errors.`**`UnsupportedError`**
Requested an unsupported feature. Did you properly initialized everything?

**exception** `pyrana.errors.`**`WrongParameterError`**
Unknown or invalid parameter supplied.

## 1.3 packets

This module provides the transport layer Packet support code. For internal usage only: do not use nor import directly.

**class** `pyrana.packet.`**`Packet`**(*stream_id=None*, *data=None*, *pts=-9223372036854775808*, *dts=-9223372036854775808*, *is_key=False*)
a Packet object represents an immutable, encoded packet of a multimedia stream.

**`blob`**()
returns the bytes() dump of the object

**`data`**
the raw data (bytes) this packet carries.

**dts**
the Decoding TimeStamp of this packet.

**classmethod from_cdata**(*cpkt*)
builds a pyrana Packet from (around) a (cffi-wrapped) libav* packet object. The libav object must be already initialized and ready to go. WARNING: raw access. Use with care.

**is_key**
boolean flag. Is this packet a key frame? (provided by libav*)

**pts**
the Presentation TimeStamp of this packet.

**raw_pkt**(**args*, ***kwds*)
raw access to the underlying FFmpeg packet. used by decoders in some corner but important cases. For internal usage only. TODO: ensure R/O and (thus) simplify

**size**
Size of the packet data (bytes)

**stream_id**
the identifier of the logical stream which this packet belongs to.

**class** pyrana.packet.**PacketFlags**
wrapper for the (wannabe)enum of AVPktFlag in libavcodec/avcodec.h

pyrana.packet.**bind_packet**(**args*, ***kwds*)
allocates an AVPacket and cleans it up on exception.

pyrana.packet.**raw_packet**(**args*, ***kwds*)
context manager for a raw ffmpeg packet of the given size.

## 1.4 formats

This module provides the transport layer interface: encoded packets, Muxer, Demuxers and their support code.

**class** pyrana.formats.**AVFmtFlags**
wrapper for the (wannabe)enum in libavformat/avformat.h

**class** pyrana.formats.**Demuxer**(*src*, *name=None*, *delay_open=False*, *streaming=False*)
Demuxer object. Use a file-like for real I/O. The file-like must be already open, and must support read() returning bytes (not strings). If the file format is_seekable but the file-like doesn't support seek, expect weird things.

**close**()
close the underlying demuxer.

**next**()
python 2.x iterator hook.

**open**(*name=None*)
open the underlying demuxer.

**open_decoder**(*stream_id*)
create and returns a full-blown decoder Instance capable to decode the selected stream. Like doing things manually, just easily.

**read_frame**(*stream_id=-1*)
reads and returns a new complete encoded frame (enclosed in a Packet) from the demuxer. if the optional 'stream_id' argument is !ANY, returns a frame belonging to the specified streams.

raises EndOfStreamError if - a stream id is specified, and such streams doesn't exists. - the streams ends.

**seek_frame**(*frameno*, *stream_id=-1*)
seek to the given frame number in the stream.

**seek_ts**(*tstamp*, *stream_id=-1*)
seek to the given timestamp (msecs) in the stream.

**stream**(*sid=-1*)
generator that returns all packets that belong to a specified stream id.

**streams**
streams: read-only attribute list of StreamInfo objects describing the streams found by the demuxer (as in old pyrana, no changes)

**class** pyrana.formats.**FormatFlags**
wrapper for the (wannabe)enum of AVFormatFlags in libavformat/avformat.h

**class** pyrana.formats.**Muxer**(*sink*, *name=None*, *streaming=True*)
Muxer object. Use a file-like for real I/O. The file-like must be already open, and must support write() returning bytes (not strings). If the file format is_seekable but the file-like doesn't support seek, expect weird things.

**add_stream**(*encoder*)
register a new stream into the Muxer for the given Encoder. XXX add more docs

**open_encoder**(*output_codec*, *params*)
create and returns a full-blown enccoder Instance capable, given the encoder parameters, already bound and registered as stream in the Muxer.

**write_frame**(*packet*)
writes a data frame, enclosed into an encoded Packet, in the stream.

**write_header**()
Writes the header into the output stream.

**write_trailer**()
Writes the trailer (if any) into the output stream. Requires the header to be written (and, likely, some data) Must be the last operation before to release the Muxer.

**class** pyrana.formats.**SeekFlags**
wrapper for the (wannabe)enum of AVSeekFlags in libavformat/avformat.h

pyrana.formats.**find_stream**(*streams*, *nth*, *media*)
find the nth stream of the specified media a streams info (as in Demuxer().streams). Return the corresponding stream_id. Raise NotFoundError otherwise.

## 1.5 codec

Common code shared by audio and video codecs. This module is not part of the pyrana public API.

**class** pyrana.codec.**BaseDecoder**(*input_codec*, *params=None*, *delay_open=False*)
Decoder base class. Common both to audio and video decoders.

**decode**(*packets*)
Decode data from a logical stream of packets, and returns when the first next frame is available. The input stream can be - a materialized sequence of packets (list, tuple...) - a generator (e.g. Demuxer.stream()).

**decode_packet**(*packet*)
Generator method. Decode a single packet (as in returned by a Demuxer) and extracts all the frames encoded into it. An encoded packet can legally contain more than one frame, altough this is not so common. This method deals with the [one packet -> many frames] scenario. The internal underlying decoder does its own buffer, so you can freely dispose the packet(s) fed into this method after it exited. raises

ProcessingError if decoding fails; raises NeedFeedError if decoding partially succeeds, but more data is needed to reconstruct a full frame.

**flush**()
emits all frames that can be recostructed by the data buffered into the Decoder, and empties such buffers. Call it last, do not intermix with decode*() calls. caution: more than one frame can be buffered. Raises NeedFeedError if all the internal buffers are empty.

**classmethod from_cdata**(*ctx*)
builds a pyrana Decoder from (around) a (cffi-wrapped) libav* decoder object. The libav object must be already initialized and ready to go. WARNING: raw access. Use with care.

**open**(*ffh=None*)
opens the codec into the codec context.

**class** pyrana.codec.**BaseEncoder**(*output_codec*, *params*, *delay_open=False*)
Encoder base class. Common both to audio and video encoders.

**encode**(*frame*)
Encode a logical frame in one or possibly more)packets, and return an iterable which will yield all the packets produced.

**flush**()
emits all packets which may have been buffered by the Encoder and empties such buffers. Call it last, do not intermix with encode*() calls. caution: more than one encoded frame (thus many packets) can be buffered. Raises NeedFeedError if all the internal buffers are empty.

**classmethod from_cdata**(*ctx*, *params*, *codec=None*)
builds a pyrana Encoder from (around) a (cffi-wrapped) libav* decoder object. The libav object must be already initialized and ready to go. WARNING: raw access. Use with care.

**class** pyrana.codec.**BaseFrame**
Abstract Frame class. Provides bookkeeping and access to attributes common to frames of all media types. Do not use directly.

**cdata**
Direct access to the internal C AVFrame object.

**classmethod from_cdata**(*ppframe*)
builds a pyrana generic Base Frame from (around) a (cffi-wrapped) libav* AVFrame object. The libav object must be already initialized and ready to go. WARNING: raw access. Use with care.

**is_key**
Is this a key frame?

**pts**
The Presentation TimeStamp of this Frame.

**class** pyrana.codec.**CodecFlag**
wrapper for the (wannabe) enum in avcodec.h CODEC_FLAG_*

**class** pyrana.codec.**CodecFlag2**
wrapper for the (wannabe) enum in avcodec.h CODEC_FLAG2_*

**class** pyrana.codec.**CodecMixin**(*params=None*)
Mixin. Abstracts the common codec attributes: parameters reference, read-only access, extradata management.

**extra_data**
bytearray-like, read-write

**media_type**
the codec media type.

**open**(*ffh=None*)
opens the codec into the codec context.

**params**
the codec parameters.

**ready**
is the codec readu to go?

**setup**()
Dispach the given parameters to the internal (FFmpeg) data structures.

**class** pyrana.codec.**Payload**
Generic media-agnostic frame payload.

**blob**()
returns the bytes() dump of the object.

pyrana.codec.**bind_frame**(**args*, ***kwds*)
allocates an AVFrame and cleans it up on exception.

pyrana.codec.**find_encoder**(*output_codec*, *ffh=None*)
Finds a suitable encoder for the given output codec. Raises SetupError if the codec isn't supported.

pyrana.codec.**make_codec**(*vcodec*, *acodec*, *stream_id*, *ctx*, **args*)
builds the right decoder for a given stream of an AVCodecContext.

pyrana.codec.**make_fetcher**(*seq*)
Builds a callable which extracts, deletes from the originating sequence-like (either materialized or generating) and returns an item.

pyrana.codec.**make_payload**(*cls*, *ffh*, *ppframe*, *parent*)
Setups the common fields of every multimedia payload object.

pyrana.codec.**wire_decoder**(*dec*, *av_decode*, *new_frame*, *mtype*)
Injects the specific decoding hooks in a generic decoder.

pyrana.codec.**wire_encoder**(*enc*, *av_encode*, *mtype*)
Injects the specific encoding hooks in a generic encoder.

## 1.6 audio

this module provides the audio codec interface. Encoders, Decoders and their support code.

**class** pyrana.audio.**AVRounding**
Rounding methods. Maybe should be moved into a more generic module.

**class** pyrana.audio.**Decoder**(*input_codec*, *params=None*)
Decodes audio Packets into audio Frames.

**classmethod from_cdata**(*ctx*)
builds a pyrana Audio Decoder from (around) a (cffi-wrapped) libav* (audio)decoder object. The libav object must be already initialized and ready to go. WARNING: raw access. Use with care.

**static wire**(*dec*)
wire up the Decoder. See codec.wire_decoder

**class** pyrana.audio.**Encoder**(*output_codec*, *params*)
Encode audio Frames into Packets.

**classmethod from_cdata**(*ctx*, *params*, *codec=None*)
builds a pyrana audio Encoder from (around) a (cffi-wrapped) liabv* (audio) context. WARNING: raw access. Use with care.

**static wire**(*enc*)
wire up the Encoder. See codec.wire_encoder

**class** pyrana.audio.**Frame**(*rate*, *layout*, *samplefmt*)
An Audio frame.

**samples**(*smpfmt=None*)
Returns a new Image object which provides access to the Picture (thus the pixel as bytes()) data.

**class** pyrana.audio.**Samples**
Represents the Sample data inside a Frame.

**blob**()
returns the bytes() dump of the object

**bps**
Bytes per sample.

**channel**(*idx*)
Read-only byte access to a single channel of the Samples.

**channels**
The number of audio channels, only used for audio.

**convert**(*smpfmt*)
convert the Samples data in a new SampleFormat. returns a brand new, independent Image.

**classmethod from_cdata**(*ppframe*, *swr=None*, *parent=None*)
builds a pyrana Image from a (cffi-wrapped) libav* Frame object. The Picture data itself will still be hold in the Frame object. The libav object must be already initialized and ready to go. WARNING: raw access. Use with care.

**is_shared**
Is the underlying C-Frame shared with the parent py-Frame?

**num_samples**
The number of audio samples (per channel) described by this frame.

**sample_format**
Frame sample format. Expected to be always equal to the stream sample format.

**sample_rate**
Sample rate of the audio data.

pyrana.audio.**fill_s16**(*frame*)
fill a audio frame with a single tone sound

## 1.7 video

this module provides the video codec interface. Encoders, Decoders and their support code.

**class** pyrana.video.**Decoder**(*input_codec*, *params=None*)
Decodes video Packets into video Frames.

**classmethod** **from_cdata**(*ctx*)
builds a pyrana Video Decoder from (around) a (cffi-wrapped) libav* (video)decoder object. The libav object must be already initialized and ready to go. WARNING: raw access. Use with care.

**static** **wire**(*dec*)
wire up the Decoder. See codec.wire_decoder

**class** pyrana.video.**Encoder**(*output_codec*, *params*)
Encode video Frames into Packets.

**classmethod** **from_cdata**(*ctx*, *params*, *codec=None*)
builds a pyrana video Encoder from (around) a (cffi-wrapped) liabv* (audio) context. WARNING: raw access. Use with care.

**static** **wire**(*enc*)
wire up the Encoder. See codec.wire_encoder

**class** pyrana.video.**Frame**(*width*, *height*, *pixfmt*)
A Video frame.

**asr**
The sample aspect ratio of the frame.

**coded_pict_number**
Picture number in bitstream order.

**display_pict_number**
Picture number in display order.

**image**(*pixfmt=None*)
Returns a new Image object which provides access to the Picture (thus the pixel as bytes()) data.

**is_interlaced**
Is the content of the picture interlaced?

**pict_type**
Picture type of the frame, see AVPictureType.

**top_field_first**
If is_interlaced(), is top field displayed first?

**class** pyrana.video.**Image**
Represents the Picture data inside a Frame.

**blob**()
returns the bytes() dump of the object.

**convert**(*pixfmt*)
convert the Image data in a new PixelFormat. returns a brand new, independent Image.

**classmethod** **from_cdata**(*ppframe*, *sws=None*, *parent=None*)
builds a pyrana Image from a (cffi-wrapped) libav* Frame object. The Picture data itself will still be hold in the Frame object. The libav object must be already initialized and ready to go. WARNING: raw access. Use with care.

**height**
Frame height. Expected to be always equal to the stream height.

**is_shared**
Is the underlying C-Frame shared with the parent py-Frame?

**pixel_format**
Frame pixel format. Expected to be always equal to the stream pixel format.

**plane**(*idx*)
Read-only byte access to a single plane of the Image.

**planes**
Return the number of planes in the Picture data. e.g. RGB: 1; YUV420: 3

**width**
Frame width. Expected to be always equal to the stream width.

**class** pyrana.video.**SWSMode**
SWS operational flags. This wasn't a proper enum, rather a collection of #defines, and that's the reason why it is defined here.

pyrana.video.**fill_yuv420p**(*frame*, *i*)
fill a video frame with a test pattern.

CHAPTER 2

# Indices and tables

- *genindex*
- *modindex*
- *search*

## p