PyOgg

Feb 15, 2021

Contents

1 Installation **3**
- 1.1 Windows . . . 3
- 1.2 macOS . . . 3
- 1.3 Linux . . . 4

2 Testing the Installation **5**

3 Getting Started **7**
- 3.1 First Project . . . 7
- 3.2 Next Steps . . . 8

4 Examples **9**
- 4.1 Play an OggOpus file . . . 9
- 4.2 Stream an OggOpus file . . . 11
- 4.3 Encode and Decode Opus-Packets . . . 13
- 4.4 Buffered-Encode and Decode Opus-Packets . . . 15
- 4.5 Write an OggOpus File . . . 17

5 PyOgg API **19**
- 5.1 Opus-Related API . . . 19
- 5.2 Vorbis-Related API . . . 23
- 5.3 FLAC-Related API . . . 24
- 5.4 Ctypes Low-Level API . . . 24

6 PyOgg Development **25**
- 6.1 Editable Install . . . 25
- 6.2 Automated Tests . . . 25
- 6.3 Static Type Checking . . . 26
- 6.4 Examples . . . 27
- 6.5 Documentation . . . 27
- 6.6 Building Wheels . . . 27

Index **29**

PyOgg provides Python bindings for Xiph.org's Opus, Vorbis and FLAC audio file formats as well as their Ogg container format.

PyOgg:

- Reads and streams Opus, Vorbis, and FLAC audio formats in their standard file format (that is, from within Ogg containers).
- Writes Opus files (that it, Opus-formatted packets into Ogg containers)
- Reads and writes Opus-formatted packets (transported, for example, via UDP)

Further, should you wish to have still lower-level access, PyOgg provides ctypes interfaces that give direct access to the C functions and datatypes found in the libraries.

Under Windows, PyOgg comes bundled with the required dynamic libraries (DLLs) in the Windows Wheel distributions.

Under macOS, the required libraries can be easily installed using Homebrew.

PyOgg is not capable of playing audio, however, you can use Python audio libraries such as simpleaudio, sounddevice, or PyOpenAL to play audio. PyOpenAL even offers 3D playback.

CHAPTER 1

Installation

We assume you have both Python and pip installed.

Some parts of PyOgg use NumPy. Although NumPy isn't required in order to use PyOgg, it certainly can make things easier. To install Numpy:

```
pip install numpy
```

1.1 Windows

PyOgg's Wheel distribution for Windows comes with the required DLLs. Before installing PyOgg, ensure that Python's Wheel package is installed:

```
pip install wheel
```

Then you can install PyOgg:

```
pip install pyogg
```

1.2 macOS

A Wheel distribution for PyOgg under macOS is a work-in-progress. However, it is very easy to install the required libraries using Homebrew.

First, ensure Homebrew itself is installed. To install Homebrew, follow the instruction found on Homebrew's home page (there is only one step).

With Homebrew installed, you may install all the libraries used by PyOgg using this command:

```
brew install libogg opus opusfile libopusenc libvorbis flac
```

Depending on how you wish to use PyOgg, many of these libraries may be optional.

Next, install PyOgg:

```
pip install pyogg
```

Finally, you may also find useful the command-line tools `opusenc`, `opusdec` and `opusinfo`. These tools provide encoding, decoding, and general information about OggOpus-encoded files. They may be installed with the command:

```
brew install opus-tools
```

1.3 Linux

Use the appropriate package installer for your Linux platform. You may find the following list helpful; it is the list of libraries used by PyOgg, with links to their GitLab sources, which can be compiled directly should pre-compiled libraries not be available:

- Ogg
- Vorbis (which includes the vorbis, vorbisfile, and vorbisenc libraries)
- Opus
- Opusfile
- libopusenc
- flac

A note on the Opus library installed under Amazon Linux 2: Amazon provides an outdated version of Opus. We recommend that you compile your own version (1.3.1 or later) from the source.

CHAPTER 2

Testing the Installation

A very quick check, to ensure that your newly installed PyOgg package is working, is to start Python and type `import pyogg`. You should not see any errors:

```
$ python
Python 3.8.5 (default, Jul 21 2020, 10:41:41)
[Clang 10.0.0 (clang-1000.11.45.5)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import pyogg
>>>
```

You can further test that PyOgg was able to find required libraries. For example, let's test that PyOgg was able to locate and load the Ogg library correctly:

```
>>> pyogg.PYOGG_OGG_AVAIL
True
```

If PyOgg can also locate the Opus and OpusFile libraries, you are ready to move on to *Getting Started*:

```
>>> pyogg.PYOGG_OPUS_AVAIL
True
>>> pyogg.PYOGG_OPUS_FILE_AVAIL
True
```

If you find that PyOgg is having difficulty finding the shared libraries, please run the file `00-test-library-availability.py` (found in the examples directory of the project's repository):

```
python 00-test-library-availability.py
```

The output shows which libraries were found and, depending on your platform, where those libraries were found. This information can be useful when trying to debug shared-library issues. Below is an example of possible output:

```
$ python 00-test-library-availability.py
Testing the availability of libraries used by PyOgg.

```

(continues on next page)

(continued from previous page)

```
If there are libraries that are not available, PyOgg's abilities will
be limited.

All libraries used by PyOgg were available.

The libraries that were loaded were found in the following file names:
 - Ogg: /usr/local/lib/libogg.dylib
 - Vorbis: /usr/local/lib/libvorbis.dylib
 - VorbisFile: /usr/local/lib/libvorbisfile.dylib
 - VorbisEnc: /usr/local/lib/libvorbisenc.dylib
 - Opus: /usr/local/lib/libopus.dylib
 - OpusFile: /usr/local/lib/libopusfile.dylib
 - OpusEnc: /usr/local/lib/libopusenc.dylib
 - Flac: /usr/local/lib/libFLAC.dylib

In Linux, from Python version 3.6, the value of the environment variable
LD_LIBRARY_PATH is used when searching for libraries, if a library cannot
be found by any other means.

For more information on the process used to locate shared libraries, see
https://docs.python.org/3/library/ctypes.html#finding-shared-libraries
```

CHAPTER 3

Getting Started

To get started with PyOgg, make sure you've followed the installation instructions (see *Installation*). The next section gives a very easy introduction for a first project with PyOgg.

3.1 First Project

As a first project, let's create a program that outputs the duration of an Opus-encoded audio file.

We'll make it easy for ourselves and use NumPy. Make sure you've installed it. To install NumPy run the command:

```
pip install numpy
```

Next, create a directory (folder) to save our work.

Download the example OggOpus file and save it as `left-right-demo-5s.opus` in the directory you just created. This file is *exactly* five seconds long.

Create a Python program in your favourite editor, let's call the file `getting_started.py`.

As the first lines of your file, Import the PyOgg and NumPy libraries with the lines:

```
import numpy
import pyogg
```

We'll now use the PyOgg class `OpusFile` to read the example OggOpus file into memory:

```
filename = "left-right-demo-5s.opus"
opus_file = pyogg.OpusFile(filename)
```

The contents of the file, stored in PCM can now be obtained using the method `as_array()`.

The method returns a NumPy array containing all the audio in the file. NumPy provides a `shape` attribute that gives us the size of the array. In our case, this will give us `(240000,2)`. What do these numbers mean?

The second number in the tuple is easy to understand: the `2` tells us there are two channels, thus this file is in stereo.

But what about the first number, the 240,000? That's the number of samples per channel. The quality of an audio recording is partially governed by the number of samples recorded per second. Opus-encoded recordings are typically saved at 48,000 samples per second. We can get this number from `opus_file.frequency`.

Now we have enough information to calculate the duration of the audio:

```
pcm = opus_file.as_array()
duration_seconds = pcm.shape[0] / opus_file.frequency
print("Audio duration (seconds):", duration_seconds)
```

Note: If you're not using Python 3, you will have to adapt the `print` command by removing the parentheses in that line.

Run your Python program from within the directory we created at the start and you should see something similar to:

```
$ python getting_started.py
Audio duration (seconds): 5.0
```

Below is the complete example:

```
import numpy
import pyogg

filename = "left-right-demo-5s.opus"
opus_file = pyogg.OpusFile(filename)

pcm = opus_file.as_array()
duration_seconds = pcm.shape[0] / opus_file.frequency
print("Audio duration (seconds):", duration_seconds)
```

3.2 Next Steps

To continue learning more about PyOgg, you might like to try running a few of the example files provided with PyOgg (see *Examples*). You might like to consider the first example and actually play our demonstration OggOpus file.

You may also like to jump in and explore the *PyOgg API*.

CHAPTER 4

Examples

The following examples can be found in the examples directory of the PyOgg GitHub repository.

You can run these examples either by downloading the appropriate file(s) or cloning the repository. Note that some examples assume that the demonstration wave and Opus files are available.

4.1 Play an OggOpus file

```
# This is an example of the use of PyOgg.
#
# Author: Matthew Walker 2020-06-01
#
# An Ogg Opus file (a file containing an Opus stream wrapped inside an
# Ogg container) is loaded using the Opusfile library.  This provides
# the entire file in one PCM encoded buffer.  That buffer is converted
# to a NumPy array and then played using simpleaudio.
#
# On successful execution of this program, you should hear the audio
# being played and the console will display comething like:
#
#    $ python 01-play-opus-simpleaudio.py
#    Reading Ogg Opus file...
#
#    Read Ogg Opus file
#    Channels:
#       2
#    Frequency (samples per second):
#       48000
#    Buffer Length (bytes):
#       960000
#    Shape of numpy array (number of samples per channel, number of channels):
#       (240000, 2)
#
```

(continues on next page)

(continued from previous page)

```
#    Playing...
#    Finished.


try:
    import pyogg
    import simpleaudio as sa # type: ignore
    import numpy # type: ignore
except ImportError:
    import os
    should_install_requirements = input(\
        "This example requires additional libraries to work.\n" +
        "  py-simple-audio (simpleaudio),\n" +
        "  NumPy (numpy)\n" +
        "  And PyOgg or course.\n" +
        "Would you like to install them right now?\n"+
        "(Y/N): ")
    if should_install_requirements.lower() == "y":
        import subprocess, sys

        install_command = [
            sys.executable,
            "-m",
            "pip",
            "install",
            "-r",
            os.path.realpath("01-play-opus-simpleaudio.requirements.txt")
        ]

        popen = subprocess.Popen(install_command,
                                 stdout=subprocess.PIPE, universal_newlines=True)


        assert popen.stdout is not None
        for stdout_line in iter(popen.stdout.readline, ""):
            print(stdout_line, end="")

        popen.stdout.close()

        popen.wait()

        print("Done.\n")

        import pyogg
        import simpleaudio as sa # type: ignore
        import numpy # type: ignore

    else:
        os._exit(0)

import ctypes
from datetime import datetime


# Specify the filename to read
filename = "left-right-demo-5s.opus"
```

(continues on next page)

(continued from previous page)

```
# Read the file using OpusFile
print("Reading Ogg Opus file...")
opus_file = pyogg.OpusFile(filename)

# Display summary information about the audio
print("\nRead Ogg Opus file")
print("Channels:\n  ", opus_file.channels)
print("Frequency (samples per second):\n  ",opus_file.frequency)
print("Buffer Length (bytes):\n  ", len(opus_file.buffer))

# Get the data as a NumPy array
buf = opus_file.as_array()

# The shape of the array can be read as
# "(number of samples per channel, number of channels)".
print(
    "Shape of numpy array (number of samples per channel, "+
    "number of channels):\n  ",
    buf.shape
)

# Play the audio
print("\nPlaying...")
play_obj = sa.play_buffer(buf,
                          opus_file.channels,
                          opus_file.bytes_per_sample,
                          opus_file.frequency)

# Wait until sound has finished playing
play_obj.wait_done()

print("Finished.")
```

4.2 Stream an OggOpus file

```
"""Reads an Ogg-Opus file using OpusFile and OpusFileStream.

Reads an Ogg-Opus file using OpusFileStream and compares it to the
results of reading it with OpusFile.  Gives timing information for the
two approaches.

A typical output:

Read 240000 samples from OpusFile (in 53.8 milliseconds).
Read 240000 samples from the OpusFileStream
(in 252 reads averaging 0.23 milliseconds each).
OpusFileStream data was identical to OpusFile data.

"""

import time

import numpy # type: ignore
import pyogg
```

(continues on next page)

(continued from previous page)

```
# Specify a file to process
opus_file_filename = "left-right-demo-5s.opus"
opus_file_stream_filename = "left-right-demo-5s.opus"

# Open the file using OpusFile, which reads the entire file
# immediately and places it into an internal buffer.
start_time = time.time()
opus_file = pyogg.OpusFile(opus_file_filename)
end_time = time.time()
duration = (end_time-start_time)*1000
array = opus_file.as_array()
array_index = 0
print("Read {:d} samples from OpusFile (in {:.1f} milliseconds).".format(
    len(array),
    duration
))

# Open the file using OpusFileStream, which does not read the entire
# file immediately.
stream = pyogg.OpusFileStream(opus_file_stream_filename)

# Loop through the OpusFileStream until we've read all the data
samples_read = 0
identical = True
times = []
while True:
    # Read the next part of the stream
    start_time = time.time()
    buf = stream.get_buffer_as_array()
    end_time = time.time()
    duration = (end_time-start_time)*1000
    times.append(duration)

    # Check if we've reached the end of the stream
    if buf is None:
        break

    # Increment the number of samples read
    samples_read += len(buf)

    # Check we've not read too much data from the stream
    if array_index+len(buf) > len(array):
        print("OpusFileStream data was identical to OpusFile data,\n"+
              "however there was more data in the OpusFileStream than\n"+
              "in the OpusFile.")
        identical = False
        break

    # Compare the stream with the OpusFile data.  (They should be
    # identical.)
    comparison = array[array_index:array_index+len(buf)] == buf
    if not numpy.all(comparison):
        print("OpusFileStream data was NOT identical to OpusFile data.")
        identical = False
        break
```

(continues on next page)

(continued from previous page)

```
    # Move the OpusFile index along
    array_index += len(buf)

avg_time = numpy.mean(times)
print(
    ("Read {:d} samples from the OpusFileStream\n"+
     "(in {:d} reads averaging {:.2f} milliseconds each).").format(
         samples_read,
         len(times),
         avg_time
     )
)

if identical == False:
    # We've finished our work here
    pass
elif array_index == len(array):
    # We completed the comparison successfully.
    print("OpusFileStream data was identical to OpusFile data.")
else:
    # There was remaining data
    print("OpusFileStream data was identical to OpusFile data,\n"+
          "however there was more data in the OpusFile than\n"+
          "in the OpusFileStream.")
```

4.3 Encode and Decode Opus-Packets

```
import wave
from pyogg import OpusEncoder
from pyogg import OpusDecoder

if __name__ == "__main__":
    # Setup encoding
    # ==============

    # Read a wav file to obtain PCM data
    filename = "left-right-demo-5s.wav"
    wave_read = wave.open(filename, "rb")
    print("Reading wav from file '{:s}'".format(filename))

    # Extract the wav's specification
    channels = wave_read.getnchannels()
    print("Number of channels:", channels)
    samples_per_second = wave_read.getframerate()
    print("Sampling frequency:", samples_per_second)
    bytes_per_sample = wave_read.getsampwidth()

    # Create an Opus encoder
    opus_encoder = OpusEncoder()
    opus_encoder.set_application("audio")
    opus_encoder.set_sampling_frequency(samples_per_second)
    opus_encoder.set_channels(channels)

    # Calculate the desired frame size (in samples per channel)
```

(continues on next page)

(continued from previous page)

```
    desired_frame_duration = 20/1000 # milliseconds
    desired_frame_size = int(desired_frame_duration * samples_per_second)

    # Setup decoding
    # ==============

    # Create an Opus decoder
    opus_decoder = OpusDecoder()
    opus_decoder.set_channels(channels)
    opus_decoder.set_sampling_frequency(samples_per_second)

    # Open an output wav for the decoded PCM
    output_filename = "output-"+filename
    wave_write = wave.open(output_filename, "wb")
    print("Writing wav into file '{:s}'".format(output_filename))

    # Save the wav's specification
    wave_write.setnchannels(channels)
    wave_write.setframerate(samples_per_second)
    wave_write.setsampwidth(bytes_per_sample)

    # Execute encode-decode
    # =====================

    # Loop through the wav file's PCM data and encode it as Opus
    bytes_encoded = 0
    while True:
        # Get data from the wav file
        pcm = wave_read.readframes(desired_frame_size)

        # Check if we've finished reading the wav file
        if len(pcm) == 0:
            break

        # Calculate the effective frame size from the number of bytes
        # read
        effective_frame_size = (
            len(pcm) # bytes
            // bytes_per_sample
            // channels
        )

        # Check if we've received enough data
        if effective_frame_size < desired_frame_size:
            # We haven't read a full frame from the wav file, so this
            # is most likely a final partial frame before the end of
            # the file.  We'll pad the end of this frame with silence.
            pcm += (
                b"\x00"
                * ((desired_frame_size - effective_frame_size)
                   * bytes_per_sample
                   * channels)
            )

        # Encode the PCM data
        encoded_packet = opus_encoder.encode(pcm)
        bytes_encoded += len(encoded_packet)
```

(continues on next page)

(continued from previous page)

```
        # At this stage we now have a buffer containing an
        # Opus-encoded packet.  This could be sent over UDP, for
        # example, and then decoded with OpusDecoder.  However it
        # cannot really be saved to a file without wrapping it in the
        # likes of an Ogg stream; for this see OggOpusWriter.

        # For this example, we will now immediately decode this
        # encoded packet using OpusDecoder.
        decoded_pcm = opus_decoder.decode(encoded_packet)

        # Save the decoded PCM as a new wav file
        wave_write.writeframes(decoded_pcm)

    wave_read.close()
    wave_write.close()
    print("Total bytes of encoded packets:", bytes_encoded)
    print("Finished.")
```

4.4 Buffered-Encode and Decode Opus-Packets

```
import wave
from pyogg import OpusBufferedEncoder
from pyogg import OpusDecoder

if __name__ == "__main__":
    # Setup encoding
    # ==============

    # Read a wav file to obtain PCM data
    filename = "left-right-demo-5s.wav"
    wave_read = wave.open(filename, "rb")
    print("Reading wav from file '{:s}'".format(filename))

    # Extract the wav's specification
    channels = wave_read.getnchannels()
    print("Number of channels:", channels)
    samples_per_second = wave_read.getframerate()
    print("Sampling frequency:", samples_per_second)
    bytes_per_sample = wave_read.getsampwidth()

    # Create an Opus encoder
    opus_encoder = OpusBufferedEncoder()
    opus_encoder.set_application("audio")
    opus_encoder.set_sampling_frequency(samples_per_second)
    opus_encoder.set_channels(channels)
    desired_frame_duration = 20 # ms
    opus_encoder.set_frame_size(desired_frame_duration)

    # Setup decoding
    # ==============

    # Create an Opus decoder
    opus_decoder = OpusDecoder()
```

(continues on next page)

(continued from previous page)

```
    opus_decoder.set_channels(channels)
    opus_decoder.set_sampling_frequency(samples_per_second)

    # Open an output wav for the decoded PCM
    output_filename = "output-"+filename
    wave_write = wave.open(output_filename, "wb")
    print("Writing wav into file '{:s}'".format(output_filename))

    # Save the wav's specification
    wave_write.setnchannels(channels)
    wave_write.setframerate(samples_per_second)
    wave_write.setsampwidth(bytes_per_sample)

    # Execute encode-decode
    # =====================

    # Loop through the wav file's PCM data and encode it as Opus
    bytes_encoded = 0
    finished = False
    while not finished:
        # Get data from the wav file
        frames_per_read = 1000
        pcm = wave_read.readframes(frames_per_read)

        # Check if we've finished reading the wav file
        if len(pcm) == 0:
            # Encode what's left of the PCM and flush it by filling
            # any partial frames with silence.
            finished = True

        # Encode the PCM data
        encoded_packets = opus_encoder.buffered_encode(
            memoryview(bytearray(pcm)), # FIXME
            flush=finished
        )

        # At this stage we now have a list of Opus-encoded packets.
        # These could be sent over UDP, for example, and then decoded
        # with OpusDecoder.  However they cannot really be saved to a
        # file without wrapping them in the likes of an Ogg stream;
        # for this see OggOpusWriter.

        # For this example, we will now immediately decode the encoded
        # packets using OpusDecoder.
        for encoded_packet, _, _ in encoded_packets:
            bytes_encoded += len(encoded_packet)

            decoded_pcm = opus_decoder.decode(encoded_packet)

            # Save the decoded PCM as a new wav file
            wave_write.writeframes(decoded_pcm)

    wave_read.close()
    wave_write.close()
    print("Total bytes of encoded packets:", bytes_encoded)
    print("Finished.")
```

4.5 Write an OggOpus File

```
import wave
import pyogg

if __name__ == "__main__":
    # Read a wav file to obtain PCM data
    filename = "left-right-demo-5s.wav"
    wave_read = wave.open(filename, "rb")
    print("Reading wav from file '{:s}'".format(filename))

    # Extract the wav's specification
    channels = wave_read.getnchannels()
    print("Number of channels:", channels)
    samples_per_second = wave_read.getframerate()
    print("Sampling frequency:", samples_per_second)
    bytes_per_sample = wave_read.getsampwidth()
    original_length = wave_read.getnframes()
    print("Length:", original_length)

    # Create a OpusBufferedEncoder
    opus_buffered_encoder = pyogg.OpusBufferedEncoder()
    opus_buffered_encoder.set_application("audio")
    opus_buffered_encoder.set_sampling_frequency(samples_per_second)
    opus_buffered_encoder.set_channels(channels)
    opus_buffered_encoder.set_frame_size(20) # milliseconds

    # Create an OggOpusWriter
    output_filename = filename+".opus"
    print("Writing OggOpus file to '{:s}'".format(output_filename))
    ogg_opus_writer = pyogg.OggOpusWriter(
        output_filename,
        opus_buffered_encoder
    )

    # Calculate the desired frame size (in samples per channel)
    desired_frame_duration = 20/1000 # milliseconds
    desired_frame_size = int(desired_frame_duration * samples_per_second)

    # Loop through the wav file's PCM data and write it as OggOpus
    chunk_size = 1000 # bytes
    while True:
        # Get data from the wav file
        pcm = wave_read.readframes(chunk_size)

        # Check if we've finished reading the wav file
        if len(pcm) == 0:
            break

        # Encode the PCM data
        ogg_opus_writer.write(
            memoryview(bytearray(pcm)) # FIXME
        )

    # We've finished writing the file
    ogg_opus_writer.close()
```

(continues on next page)

(continued from previous page)

```
    # Check that the output file is that same length as the original
    print("Reading output file:", output_filename)
    opus_file = pyogg.OpusFile(output_filename)
    print("File read")
    output_length = opus_file.as_array().shape[0]
    print("Output length:", output_length)

    if original_length != output_length:
        print("ERROR: The original length is different to the output length")

    print("Finished.")
```

CHAPTER 5

PyOgg API

PyOgg is able to read, stream and write Opus-formatted audio. It is also able to read and stream Vorbis and FLAC-encoded audio.

PyOgg also offers a low-level ctypes interface to the C functions and datatypes.

5.1 Opus-Related API

The following classes depend on the availability of libraries.

If the required libraries are available then the classes can be imported directly from the `pyogg` module, for example using the statement `from pyogg import OpusFile`. Or you may import the package using `import pyogg` and reference the classes explicitly, such as `pyogg.OpusFile`.

If the required libraries are not available then instantiating a class will raise a `PyOggError` exception.

5.1.1 OpusFile

To read an entire OggOpus-encoded audio file into memory, use the `OpusFile` class. For a basic example of its use see *Getting Started*. For a more elaborate example see *Play an OggOpus file*.

This class requires the shared libraries Ogg, Opus, and Opusfile.

class pyogg.opus_file.**OpusFile**(*path: str*)

bytes_per_sample = None
Bytes per sample

channels = None
Number of channels in audio file.

frequency = None
Number of samples per second (per channel). Always 48,000.

5.1.2 OpusFileStream

Often, reading an entire file into memory is not desirable. Rather, we prefer to stream the file by reading small chunks at a time, processing the small section of the PCM and then moving on to the next section. This can be achieved with the `OpusFileStream` class. For an example of its use see *Stream an OggOpus file*.

This class requires the shared libraries Ogg, Opus, and Opusfile.

class pyogg.opus_file_stream.**OpusFileStream**(*path*)

bytes_per_sample = None
Bytes per sample

channels = None
Number of channels in audio file

frequency = None
Number of samples per second (per channel)

get_buffer()
Obtains the next frame of PCM samples.

Returns an array of signed 16-bit integers. If the file is in stereo, the left and right channels are interleaved.

Returns None when all data has been read.

The array that is returned should be either processed or copied before the next call to get_buffer() or get_buffer_as_array() as the array's memory is reused for each call.

get_buffer_as_array()
Provides the buffer as a NumPy array.

Note that the underlying data type is 16-bit signed integers.

Does not copy the underlying data, so the returned array should either be processed or copied before the next call to get_buffer() or get_buffer_as_array().

pcm_size = None
Total PCM Length

5.1.3 OggOpusWriter

To write OggOpus encoded files, use the `OggOpusWriter` class.

This class requires the shared libraries Ogg and Opus.

5.1.4 OpusEncoder

class pyogg.opus_encoder.**OpusEncoder**
Encodes PCM data into Opus frames.

encode(*pcm: Union[bytes, bytearray, memoryview]*) → memoryview
Encodes PCM data into an Opus frame.

pcm must be formatted as bytes-like, with each sample taking two bytes (signed 16-bit integers; interleaved left, then right channels if in stereo).

If *pcm* is not writeable, a copy of the array will be made.

get_algorithmic_delay()
Gets the total samples of delay added by the entire codec.

This can be queried by the encoder and then the provided number of samples can be skipped on from the start of the decoder's output to provide time aligned input and output. From the perspective of a decoding application the real data begins this many samples late.

The decoder contribution to this delay is identical for all decoders, but the encoder portion of the delay may vary from implementation to implementation, version to version, or even depend on the encoder's initial configuration. Applications needing delay compensation should call this method rather than hard-coding a value.

set_application(*application: str*) → None
Set the encoding mode.

This must be one of 'voip', 'audio', or 'restricted_lowdelay'.

'voip': Gives best quality at a given bitrate for voice signals. It enhances the input signal by high-pass filtering and emphasizing formants and harmonics. Optionally it includes in-band forward error correction to protect against packet loss. Use this mode for typical VoIP applications. Because of the enhancement, even at high bitrates the output may sound different from the input.

'audio': Gives best quality at a given bitrate for most non-voice signals like music. Use this mode for music and mixed (music/voice) content, broadcast, and applications requiring less than 15 ms of coding delay.

'restricted_lowdelay': configures low-delay mode that disables the speech-optimized mode in exchange for slightly reduced delay. This mode can only be set on an newly initialized encoder because it changes the codec delay.

set_channels(*n: int*) → None
Set the number of channels.

n must be either 1 or 2.

set_max_bytes_per_frame(*max_bytes: int*) → None
Set the maximum number of bytes in an encoded frame.

Size of the output payload. This may be used to impose an upper limit on the instant bitrate, but should not be used as the only bitrate control.

TODO: Use OPUS_SET_BITRATE to control the bitrate.

set_sampling_frequency(*samples_per_second: int*) → None
Set the number of samples (per channel) per second.

This must be one of 8000, 12000, 16000, 24000, or 48000.

Regardless of the sampling rate and number of channels selected, the Opus encoder can switch to a lower audio bandwidth or number of channels if the bitrate selected is too low. This also means that it is safe to always use 48 kHz stereo input and let the encoder optimize the encoding.

5.1.5 OpusBufferedEncoder

class pyogg.opus_buffered_encoder.**OpusBufferedEncoder**

buffered_encode(*pcm_bytes: memoryview, flush: bool = False, callback: Callable[[memoryview, int, bool], None] = None*) → List[Tuple[memoryview, int, bool]]
Gets encoded packets and their number of samples.

This method returns a list, where each item in the list is a tuple. The first item in the tuple is an Opus-encoded frame stored as a bytes-object. The second item in the tuple is the number of samples encoded (excluding silence).

If *callback* is supplied then this method will instead return an empty list but call the callback for every Opus-encoded frame that would have been returned as a list. This option has the desireable property of eliminating the copying of the encoded packets, which is required in order to form a list. The callback should take two arguments, the encoded frame (a Python bytes object) and the number of samples encoded per channel (an int). The user must either process or copy the data as the data may be overwritten once the callback terminates.

encode(*pcm: Union[bytes, bytearray, memoryview]*) → memoryview
Encodes PCM data into an Opus frame.

pcm must be formatted as bytes-like, with each sample taking two bytes (signed 16-bit integers; interleaved left, then right channels if in stereo).

If *pcm* is not writeable, a copy of the array will be made.

get_algorithmic_delay()
Gets the total samples of delay added by the entire codec.

This can be queried by the encoder and then the provided number of samples can be skipped on from the start of the decoder's output to provide time aligned input and output. From the perspective of a decoding application the real data begins this many samples late.

The decoder contribution to this delay is identical for all decoders, but the encoder portion of the delay may vary from implementation to implementation, version to version, or even depend on the encoder's initial configuration. Applications needing delay compensation should call this method rather than hard-coding a value.

set_application(*application: str*) → None
Set the encoding mode.

This must be one of 'voip', 'audio', or 'restricted_lowdelay'.

'voip': Gives best quality at a given bitrate for voice signals. It enhances the input signal by high-pass filtering and emphasizing formants and harmonics. Optionally it includes in-band forward error correction to protect against packet loss. Use this mode for typical VoIP applications. Because of the enhancement, even at high bitrates the output may sound different from the input.

'audio': Gives best quality at a given bitrate for most non-voice signals like music. Use this mode for music and mixed (music/voice) content, broadcast, and applications requiring less than 15 ms of coding delay.

'restricted_lowdelay': configures low-delay mode that disables the speech-optimized mode in exchange for slightly reduced delay. This mode can only be set on an newly initialized encoder because it changes the codec delay.

set_channels(*n: int*) → None
Set the number of channels.

n must be either 1 or 2.

set_frame_size(*frame_size: float*) → None
Set the desired frame duration (in milliseconds).

Valid options are 2.5, 5, 10, 20, 40, or 60ms.

set_max_bytes_per_frame (*max_bytes: int*) → None
Set the maximum number of bytes in an encoded frame.

Size of the output payload. This may be used to impose an upper limit on the instant bitrate, but should not be used as the only bitrate control.

TODO: Use OPUS_SET_BITRATE to control the bitrate.

set_sampling_frequency (*samples_per_second: int*) → None
Set the number of samples (per channel) per second.

This must be one of 8000, 12000, 16000, 24000, or 48000.

Regardless of the sampling rate and number of channels selected, the Opus encoder can switch to a lower audio bandwidth or number of channels if the bitrate selected is too low. This also means that it is safe to always use 48 kHz stereo input and let the encoder optimize the encoding.

5.1.6 OpusDecoder

class pyogg.opus_decoder.**OpusDecoder**

decode (*encoded_bytes: memoryview*)
Decodes an Opus-encoded packet into PCM.

decode_missing_packet (*frame_duration*)
Obtain PCM data despite missing a frame.

frame_duration is in milliseconds.

set_channels (*n*)
Set the number of channels.

n must be either 1 or 2.

The decoder is capable of filling in either mono or interleaved stereo pcm buffers.

set_sampling_frequency (*samples_per_second*)
Set the number of samples (per channel) per second.

samples_per_second must be one of 8000, 12000, 16000, 24000, or 48000.

Internally Opus stores data at 48000 Hz, so that should be the default value for Fs. However, the decoder can efficiently decode to buffers at 8, 12, 16, and 24 kHz so if for some reason the caller cannot use data at the full sample rate, or knows the compressed data doesn't use the full frequency range, it can request decoding at a reduced rate.

5.2 Vorbis-Related API

5.2.1 VorbisFile

To read an entire OggVorbis-encoded audio file into memory, use the `VorbisFile` class.

class pyogg.**VorbisFile**(***kw*)

5.2.2 VorbisFileStream

class pyogg.**VorbisFileStream**(***kw*)

5.3 FLAC-Related API

5.3.1 FlacFile

To read an entire OggFlac-encoded audio file into memory, use the `FlacFile` class.

class pyogg.**FlacFile**(**kw*)

5.3.2 FlacFileStream

class pyogg.**FlacFileStream**(**kw*)

5.4 Ctypes Low-Level API

All the functions, structures and datatypes are the same as in the C implementation, except for some that couldn't be translated. If you want to use them natively you will have to use ctypes' data types. Please refer to the official documentation and the C headers.

You can import the various functions from pyogg.ogg, pyogg.vorbis, pyogg.opus and pyogg.flac or use the predefined classes and functions from pyogg.

CHAPTER 6

PyOgg Development

If you are interested in contributing to the development of PyOgg:

- The master git repository can be found on the project's GitHub site.
- If you would like to report a bug or have a feature request, consider posting to an issue on the project's GitHub site.
- Finally, please make a pull request if you develop code you'd like to have added into PyOgg.

6.1 Editable Install

If you're developing PyOgg, you may find it more convenient if PyOgg is installed in editable mode. From the same directory as `setup.py` can be found, run:

```
pip install -e .
```

6.2 Automated Tests

PyOgg includes a collection of automated tests. They require `pytest`, which can be installed with:

```
pip install pytest
```

Note: If you have an old version of pytest installed, make sure to install an up-to-date version (say, in a virtual environment). If you do so, please restart Terminal as you may otherwise encounter spurious `ModuleNotFoundError` errors. For more information, see Dirk Avery's blog post on pytest or check your version of pytest by running:

```
pytest --version
```

Once you have pytest installed, you may run the tests with the commands:

```
cd tests
pytest
```

This should output something similar to:

```
$ pytest
========================= test session starts ==========================
platform darwin -- Python 3.8.5, pytest-6.0.1, py-1.9.0, pluggy-0.13.1
rootdir: /Users/matthew/Desktop/VirtualChoir/PyOgg/PyOgg
collected 39 items

test_ogg_opus_writer.py ......                                   [ 15%]
test_opus_buffered_encoder.py ....                               [ 25%]
test_opus_decoder.py .........                                   [ 48%]
test_opus_encoder.py ............                                [ 79%]
test_opus_file.py ....                                           [ 89%]
test_opus_file_stream.py ....                                    [100%]

========================== 39 passed in 1.20s ==========================
```

As at August 2020, we had about 75% code coverage. This can be examined by installing the Python coverage package:

```
pip install coverage
```

And then running the tests with the command:

```
coverage run --source=../pyogg -m pytest
coverage html
```

You can then open the file `htmlcov/index.html`, which gives a detailed line-by-line analysis of the tests' coverage.

6.3 Static Type Checking

As at November 2020, a considerable portion of PyOgg has had type hinting added. This allows for static type checkers such as mypy to be used, which can help detect bugs without even running your code.

Mypy is run as part of the Travis continuous integration script; checking is performed on both the PyOgg package itself and also its automated tests.

To run the mypy checks yourself, you will need to have mypy installed:

```
pip install mypy
```

The tests for the PyOgg package can then by run from the root of the git repository using:

```
mypy -p pyogg
```

Checking of the automated tests can be done (also from the root of the repository) with:

```
mypy tests/*.py
```

6.4 Examples

The examples directory of the git repository provides example audio files and example code showing how to use PyOgg.

When run, some of the code examples use simpleaudio to play the audio files.

The majority of the examples are run as part of the Travis continuous integration script. Specifically, those examples that play audio files are skipped. Including the example files as part of the continuous integration ensures that the examples are kept up-to-date with any changes made to PyOgg's API. The Travis script calls `examples/run-all-non-playing-examples.sh`, which excludes any Python scripts with "play" in their name.

6.5 Documentation

The documentation is built automatically by Read the Docs everytime there is an update of the master branch of the git repository. Thus, the latest version, and indeed the previous versions, of the documentation are always available at Read the Docs.

Further, the documentation is also built as part of the Travis continuous integration script.

To build the documentation yourself requires Sphinx. To install it:

```
pip install sphinx
```

If you are building the documentation under Windows, you may need to install Make for Windows.

To build the documentation run:

```
cd docs
make html
```

You will then find the latest documentation at `docs/_build/html/index.html`.

6.6 Building Wheels

Wheels can be built for macOS, 32-bit Windows and 64-bit Windows. For these systems, pre-compiled shared libraries can be found in the project repository under `pyogg/libs/`.

To build a Wheel you will need to have installed setuptools and wheel:

```
pip install --upgrade setuptools
pip install --upgrade wheel
```

By default, the build script will create a Wheel for your current platform:

```
python setup.py build bdist_wheel
```

If you wish to create a Wheel for a different platform, set the environment variable `PYTHON_PYOGG_PLATFORM` to either `Darwin` for a macOS wheel, or `Windows` for Microsoft Windows platforms. For Windows, you will also need to set the environment variable `PYTHON_PYOGG_ARCHITECTURE` to either `32bit` or `64bit` as required. Finally, run the same build command list above.

Ensure that the version for your wheel is correct. The version definition can be found in `pyogg/__init__.py`.

- genindex

Index

B

buffered_encode() (*pyogg.opus_buffered_encoder.OpusBufferedEncoder method*), 21
bytes_per_sample (*pyogg.opus_file.OpusFile attribute*), 19
bytes_per_sample (*pyogg.opus_file_stream.OpusFileStream attribute*), 20

C

channels (*pyogg.opus_file.OpusFile attribute*), 19
channels (*pyogg.opus_file_stream.OpusFileStream attribute*), 20

D

decode() (*pyogg.opus_decoder.OpusDecoder method*), 23
decode_missing_packet() (*pyogg.opus_decoder.OpusDecoder method*), 23

E

encode() (*pyogg.opus_buffered_encoder.OpusBufferedEncoder method*), 22
encode() (*pyogg.opus_encoder.OpusEncoder method*), 20

F

FlacFile (*class in pyogg*), 24
FlacFileStream (*class in pyogg*), 24
frequency (*pyogg.opus_file.OpusFile attribute*), 19
frequency (*pyogg.opus_file_stream.OpusFileStream attribute*), 20

G

get_algorithmic_delay() (*pyogg.opus_buffered_encoder.OpusBufferedEncoder method*), 22
get_algorithmic_delay() (*pyogg.opus_encoder.OpusEncoder method*), 21
get_buffer() (*pyogg.opus_file_stream.OpusFileStream method*), 20
get_buffer_as_array() (*pyogg.opus_file_stream.OpusFileStream method*), 20

O

OpusBufferedEncoder (*class in pyogg.opus_buffered_encoder*), 21
OpusDecoder (*class in pyogg.opus_decoder*), 23
OpusEncoder (*class in pyogg.opus_encoder*), 20
OpusFile (*class in pyogg.opus_file*), 19
OpusFileStream (*class in pyogg.opus_file_stream*), 20

P

pcm_size (*pyogg.opus_file_stream.OpusFileStream attribute*), 20

S

set_application() (*pyogg.opus_buffered_encoder.OpusBufferedEncoder method*), 22
set_application() (*pyogg.opus_encoder.OpusEncoder method*), 21
set_channels() (*pyogg.opus_buffered_encoder.OpusBufferedEncoder method*), 22
set_channels() (*pyogg.opus_decoder.OpusDecoder method*), 23
set_channels() (*pyogg.opus_encoder.OpusEncoder method*), 21
set_frame_size() (*pyogg.opus_buffered_encoder.OpusBufferedEncoder method*), 22
set_max_bytes_per_frame() (*pyogg.opus_buffered_encoder.OpusBufferedEncoder method*), 22

set_max_bytes_per_frame() (*pyogg.opus_encoder.OpusEncoder method*), 21
set_sampling_frequency() (*pyogg.opus_buffered_encoder.OpusBufferedEncoder method*), 23
set_sampling_frequency() (*pyogg.opus_decoder.OpusDecoder method*), 23
set_sampling_frequency() (*pyogg.opus_encoder.OpusEncoder method*), 21

V

VorbisFile (*class in pyogg*), 23
VorbisFileStream (*class in pyogg*), 23