ffmpy Documentation

Release 0.3.1

Andriy Yurchuk

Jul 17, 2023

Contents

1 Installation **3**

2 Quickstart **5**

3 Documentation **7**

3.1 ffmpy 7

3.2 Examples 8

Python Module Index **13**

Index **15**

ffmpy is a Python wrapper for FFmpeg. It compiles FFmpeg command line from provided arguments and their respective options and executes it using Python's subprocess.

ffmpy resembles the command line approach FFmpeg uses. It can read from an arbitrary number of input "files" (regular files, pipes, network streams, grabbing devices, etc.) and write into arbitrary number of output "files". See FFmpeg documentation for further details about how FFmpeg command line options and arguments work.

ffmpy supports FFmpeg pipe protocol. This means that it is possible to pass input data to `stdin` and get output data from `stdout`.

At this moment ffmpy has wrappers for `ffmpeg` and `ffprobe` commands, but it should be possible to run other FFmpeg tools with it (e.g. `ffserver`).

CHAPTER 1

Installation

```
pip install ffmpy
```

CHAPTER 2

Quickstart

```
>>> import ffmpy
>>> ff = ffmpy.FFmpeg(
...     inputs={'input.mp4': None},
...     outputs={'output.avi': None}
... )
>>> ff.run()
```

This takes `input.mp4` file in the current directory as the input, changes the video container from MP4 to AVI without changing any other video parameters and creates a new output file `output.avi` in the current directory.

CHAPTER 3

Documentation

3.1 ffmpy

class `ffmpy.`**`FFmpeg`**(*executable='ffmpeg'*, *global_options=None*, *inputs=None*, *outputs=None*)

Wrapper for various FFmpeg related applications (ffmpeg, ffprobe).

Initialize FFmpeg command line wrapper.

Compiles FFmpeg command line from passed arguments (executable path, options, inputs and outputs). `inputs` and `outputs` are dictionares containing inputs/outputs as keys and their respective options as values. One dictionary value (set of options) must be either a single space separated string, or a list or strings without spaces (i.e. each part of the option is a separate item of the list, the result of calling `split()` on the options string). If the value is a list, it cannot be mixed, i.e. cannot contain items with spaces. An exception are complex FFmpeg command lines that contain quotes: the quoted part must be one string, even if it contains spaces (see *Examples* for more info). For more info about FFmpeg command line format see here.

Parameters

- **`executable`** (*`str`*) – path to ffmpeg executable; by default the `ffmpeg` command will be searched for in the `PATH`, but can be overridden with an absolute path to `ffmpeg` executable
- **`global_options`** (*`iterable`*) – global options passed to `ffmpeg` executable (e.g. `-y`, `-v` etc.); can be specified either as a list/tuple/set of strings, or one space-separated string; by default no global options are passed
- **`inputs`** (*`dict`*) – a dictionary specifying one or more input arguments as keys with their corresponding options (either as a list of strings or a single space separated string) as values
- **`outputs`** (*`dict`*) – a dictionary specifying one or more output arguments as keys with their corresponding options (either as a list of strings or a single space separated string) as values

`run`(*input_data=None*, *stdout=None*, *stderr=None*, *env=None*, ***kwargs*)

Execute FFmpeg command line.

`input_data` can contain input for FFmpeg in case `pipe` protocol is used for input. `stdout` and `stderr` specify where to redirect the `stdout` and `stderr` of the process. By default no redirection is done, which means all output goes to running shell (this mode should normally only be used for debugging purposes). If FFmpeg `pipe` protocol is used for output, `stdout` must be redirected to a pipe by passing `subprocess.PIPE` as `stdout` argument. You can pass custom environment to ffmpeg process with `env`.

Returns a 2-tuple containing `stdout` and `stderr` of the process. If there was no redirection or if the output was redirected to e.g. `os.devnull`, the value returned will be a tuple of two `None` values, otherwise it will contain the actual `stdout` and `stderr` data returned by ffmpeg process.

More info about `pipe` protocol here.

Parameters

- **`input_data`** (*`str`*) – input data for FFmpeg to deal with (audio, video etc.) as bytes (e.g. the result of reading a file in binary mode)
- **`stdout`** – redirect FFmpeg `stdout` there (default is `None` which means no redirection)
- **`stderr`** – redirect FFmpeg `stderr` there (default is `None` which means no redirection)
- **`env`** – custom environment for ffmpeg process
- **`kwargs`** – any other keyword arguments to be forwarded to subprocess.Popen

Returns a 2-tuple containing `stdout` and `stderr` of the process

Return type tuple

Raise *`FFRuntimeError`* in case FFmpeg command exits with a non-zero code; *`FFExecutableNotFoundError`* in case the executable path passed was not valid

class `ffmpy.`**`FFprobe`**(*executable='ffprobe'*, *global_options=''*, *inputs=None*)

Wrapper for ffprobe.

Create an instance of FFprobe.

Compiles FFprobe command line from passed arguments (executable path, options, inputs). FFprobe executable by default is taken from `PATH` but can be overridden with an absolute path. For more info about FFprobe command line format see here.

Parameters

- **`executable`** (*`str`*) – absolute path to ffprobe executable
- **`global_options`** (*`iterable`*) – global options passed to ffmpeg executable; can be specified either as a list/tuple of strings or a space-separated string
- **`inputs`** (*`dict`*) – a dictionary specifying one or more inputs as keys with their corresponding options as values

exception `ffmpy.`**`FFExecutableNotFoundError`**

Raise when FFmpeg/FFprobe executable was not found.

exception `ffmpy.`**`FFRuntimeError`**(*cmd*, *exit_code*, *stdout*, *stderr*)

Raise when FFmpeg/FFprobe command line execution returns a non-zero exit code.

The resulting exception object will contain the attributes relates to command line execution: `cmd`, `exit_code`, `stdout`, `stderr`.

3.2 Examples

- *Format conversion*
- *Transcoding*
- *Demultiplexing*
- *Multiplexing*
- *Using `pipe` protocol*
- *Complex command lines*

3.2.1 Format conversion

The simplest example of usage is converting media from one format to another (in this case from MPEG transport stream to MP4) preserving all other attributes:

```
>>> from ffmpy import FFmpeg
... ff = FFmpeg(
...     inputs={'input.ts': None},
...     outputs={'output.mp4': None}
... )
>>> ff.cmd
'ffmpeg -i input.ts output.mp4'
>>> ff.run()
```

3.2.2 Transcoding

If at the same time we wanted to re-encode video and audio using different codecs we'd have to specify additional output options:

```
>>> ff = FFmpeg(
...     inputs={'input.ts': None},
...     outputs={'output.mp4': '-c:a mp2 -c:v mpeg2video'}
... )
>>> ff.cmd
'ffmpeg -i input.ts -c:a mp2 -c:v mpeg2video output.mp4'
>>> ff.run()
```

3.2.3 Demultiplexing

A more complex usage example would be demultiplexing an MPEG transport stream into separate elementary (audio and video) streams and save them in MP4 containers preserving the codecs (note how a list is used for options here):

```
>>> ff = FFmpeg(
...     inputs={'input.ts': None},
...     outputs={
...         'video.mp4': ['-map', '0:0', '-c:a', 'copy', '-f', 'mp4'],
...         'audio.mp4': ['-map', '0:1', '-c:a', 'copy', '-f', 'mp4']
...     }
... )
>>> ff.cmd
```

(continues on next page)

(continued from previous page)

```
'ffmpeg -i input.ts -map 0:1 -c:a copy -f mp4 audio.mp4 -map 0:0 -c:a copy -f mp4␣
↪video.mp4'
>>> ff.run()
```

Warning: Note that it is not possible to mix the expression formats for options, i.e. it is not possible to have a list that contains strings with spaces (an exception to this is *Complex command lines*). For example, this command line will not work with `FFmpeg`:

```
>>> from subprocess import PIPE
>>> ff = FFmpeg(
...     inputs={'input.ts': None},
...     outputs={
...         'video.mp4': ['-map 0:0', '-c:a copy', '-f mp4'],
...         'audio.mp4': ['-map 0:1', '-c:a copy', '-f mp4']
...     }
... )
>>> ff.cmd
'ffmpeg -hide_banner -i input.ts "-map 0:1" "-c:a copy" "-f mp4" audio.mp4 "-map 0:0
↪" "-c:a copy" "-f mp4" video.mp4'
>>>
>>> ff.run(stderr=PIPE)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Users/ay/projects/personal/ffmpy/ffmpy.py", line 104, in run
    raise FFRuntimeError(self.cmd, ff_command.returncode, out[0], out[1])
ffmpy.FFRuntimeError: `ffmpeg -hide_banner -i input.ts "-map 0:1" "-c:a copy" "-f␣
↪mp4" audio.mp4 "-map 0:0" "-c:a copy" "-f mp4" video.mp4` exited with status 1

STDOUT:


STDERR:
Unrecognized option 'map 0:1'.
Error splitting the argument list: Option not found

>>>
```

Notice how the actual `FFmpeg` command line contains unnecessary quotes.

3.2.4 Multiplexing

To multiplex video and audio back into an MPEG transport stream with re-encoding:

```
>>> ff = FFmpeg(
...     inputs={'video.mp4': None, 'audio.mp3': None},
...     outputs={'output.ts': '-c:v h264 -c:a ac3'}
... )
>>> ff.cmd
'ffmpeg -i audio.mp4 -i video.mp4 -c:v h264 -c:a ac3 output.ts'
>>> ff.run()
```

There are cases where the order of inputs and outputs must be preserved (e.g. when using FFmpeg -map option). In these cases the use of regular Python dictionary will not work because it does not preserve order. Instead, use OrderedDict. For example we want to multiplex one video and two audio streams into an MPEG transport streams re-encoding both audio streams using different codecs. Here we use an OrderedDict to preserve the order of inputs so they match the order of streams in output options:

```
>>> from collections import OrderedDict
>>> inputs = OrderedDict([('video.mp4', None), ('audio_1.mp3', None), ('audio_2.mp3',␣
↪None)])
>>> outputs = {'output.ts', '-map 0 -c:v h264 -map 1 -c:a:0 ac3 -map 2 -c:a:1 mp2'}
>>> ff = FFmpeg(inputs=inputs, outputs=outputs)
>>> ff.cmd
'ffmpeg -i video.mp4 -i audio_1.mp3 -i audio_2.mp3 -map 0 -c:v h264 -map 1 -c:a:0 ac3␣
↪-map 2 -c:a:1 mp2 output.ts'
>>> ff.run()
```

3.2.5 Using `pipe` protocol

`ffmpy` can read input from `STDIN` and write output to `STDOUT`. This can be achieved by using FFmpeg pipe protocol. The following example reads data from a file containing raw video frames in RGB format and passes it to `ffmpy` on `STDIN`; `ffmpy` in its turn will encode raw frame data with H.264 and pack it in an MP4 container passing the output to `STDOUT` (note that you must redirect `STDOUT` of the process to a pipe by using `subprocess.PIPE` as `stdout` value, otherwise the output will get lost):

```
>>> import subprocess
>>> ff = FFmpeg(
...     inputs={'pipe:0': '-f rawvideo -pix_fmt rgb24 -s:v 640x480'},
...     outputs={'pipe:1': '-c:v h264 -f mp4'}
... )
>>> ff.cmd
'ffmpeg -f rawvideo -pix_fmt rgb24 -s:v 640x480 -i pipe:0 -c:v h264 -f mp4 pipe:1'
>>> stdout, stderr = ff.run(input_data=open('rawvideo', 'rb').read(),␣
↪stdout=subprocess.PIPE)
```

3.2.6 Complex command lines

`FFmpeg` command line can get pretty complex, for example, when using filtering. Therefore it is important to understand some of the rules for building command lines building with `ffmpy`. If an option contains quotes, it must be specified as a separate item in the options list **without** the quotes. However, if a single string is used for options, the quotes of the quoted option must be preserved in the string:

```
>>> ff = FFmpeg(
...     inputs={'input.ts': None},
...     outputs={'output.ts': ['-vf', 'adif=0:-1:0, scale=iw/2:-1']}
... )
>>> ff.cmd
'ffmpeg -i input.ts -vf "adif=0:-1:0, scale=iw/2:-1" output.ts'
>>>
>>> ff = FFmpeg(
...     inputs={'input.ts': None},
...     outputs={'output.ts': '-vf "adif=0:-1:0, scale=iw/2:-1"'}
... )
>>> ff.cmd
'ffmpeg -i input.ts -vf "adif=0:-1:0, scale=iw/2:-1" output.ts'
```

An even more complex example is a command line that burns the timecode into video:

```
ffmpeg -i input.ts -vf "drawtext=fontfile=/Library/Fonts/Verdana.ttf: timecode=
↪'09\:57\:00\:00': r=25: x=(w-tw)/2: y=h-(2*lh): fontcolor=white: box=1:␣
↪boxcolor=0x00000000@1" -an output.ts
```

In `ffmpy` it can be expressed in the following way:

```
>>> ff = FFmpeg(
...     inputs={'input.ts': None},
...     outputs={'output.ts': ['-vf', "drawtext=fontfile=/Library/Fonts/Verdana.ttf:␣
↪timecode='09\:57\:00\:00': r=25: x=(w-tw)/2: y=h-(2*lh): fontcolor=white: box=1:␣
↪boxcolor=0x00000000@1", '-an']}
... )
>>> ff.cmd
'ffmpeg -i input.ts -vf "drawtext=fontfile=/Library/Fonts/Verdana.ttf: timecode=\
↪'09\:57\:00\:00\': r=25: x=(w-tw)/2: y=h-(2*lh): fontcolor=white: box=1:␣
↪boxcolor=0x00000000@1" -an output.ts'
```

The same command line can be compiled by passing output option as a single string, while keeping the quotes:

```
>>> ff = FFmpeg(
...     inputs={'input.ts': None},
...     outputs={'output.ts': ["-vf \"drawtext=fontfile=/Library/Fonts/Verdana.ttf:␣
↪timecode='09\:57\:00\:00': r=25: x=(w-tw)/2: y=h-(2*lh): fontcolor=white: box=1:␣
↪boxcolor=0x00000000@1\" -an"}
... )
>>> ff.cmd
'ffmpeg -i input.ts -vf "drawtext=fontfile=/Library/Fonts/Verdana.ttf: timecode=\
↪'09\:57\:00\:00\': r=25: x=(w-tw)/2: y=h-(2*lh): fontcolor=white: box=1:␣
↪boxcolor=0x00000000@1" -an output.ts'
```

Python Module Index

f

ffmpy, 7

Index

F

`FFExecutableNotFoundError`, 8
`FFmpeg` (*class in ffmpy*), 7
`ffmpy` (*module*), 7
`FFprobe` (*class in ffmpy*), 8
`FFRuntimeError`, 8

R

`run()` (*ffmpy.FFmpeg method*), 7