

---

# **ffmpeg3 Documentation**

*Release 0.2.3*

**Eric Ahn**

February 03, 2017



<b>1</b>	<b>Installation</b>	<b>3</b>
<b>2</b>	<b>Quickstart</b>	<b>5</b>
<b>3</b>	<b>Documentation</b>	<b>7</b>
3.1	ffmpy3 . . . . .	7
3.2	Examples . . . . .	9
	<b>Python Module Index</b>	<b>15</b>



ffmpy3 is a Python wrapper for [FFmpeg](#), originally forked from the [ffmpy](#) project. It compiles FFmpeg command line from provided arguments and their respective options and executes it using Python's `subprocess`.

ffmpy3 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.

ffmpy3 supports FFmpeg's [pipe](#) protocol. This means that it is possible to pass input data to `stdin` and get output data from `stdout`.

At this moment ffmpy3 has wrappers for `ffmpeg` and `ffprobe` commands, but it should be possible to run other FFmpeg tools with it (e.g. `ffserver`).



---

## Installation

---

```
pip install ffmpeg3
```





---

### Quickstart

---

```
>>> import ffmpeg3
>>> ff = ffmpeg3.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.



### 3.1 ffmpeg3

**exception** `ffmpeg3.FFExecutableNotFoundError`

Raised when FFmpeg/FFprobe executable was not found.

**exception** `ffmpeg3.FFRuntimeError` (*cmd*, *exit\_code*, *stdout=b''*, *stderr=b''*)

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

**cmd**

*str*

The command used to launch the executable, with all command line options.

**exit\_code**

*int*

The resulting exit code from the executable.

**stdout**

*bytes*

The contents of stdout (only if executed synchronously).

**stderr**

*bytes*

The contents of stderr (only if executed synchronously).

**class** `ffmpeg3.FFmpeg` (*executable='ffmpeg'*, *global\_options=None*, *inputs=None*, *outputs=None*)

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

Compiles FFmpeg command line from passed arguments (executable path, options, inputs and outputs).

*inputs* and *outputs* are dictionaries 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 of 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).

#### 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*)

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.

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.

#### Parameters

- **input\_data** (*bytes*) – input data for FFmpeg to deal with (audio, video etc.) as bytes (e.g. the result of reading a file in binary mode)
- **stdout** – Where to redirect FFmpeg `stdout` to. Default is `None`, meaning no redirection.
- **stderr** – Where to redirect FFmpeg `stderr` to. Default is `None`, meaning no redirection.

#### Raises

- `FFExecutableNotFoundError` – The executable path passed was not valid.
- `FFRuntimeError` – The process exited with an error.

**Returns** A 2-tuple containing `stdout` and `stderr` from the process.

#### Return type `tuple`

**run\_async** (*input\_data=None, stdout=None, stderr=None*)

Asynchronously execute FFmpeg command line.

`input_data` can contain input for FFmpeg in case `pipe`

`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.

Note that the parent process is responsible for reading any output from `stdout/stderr`. This should be done even if the output will not be used since the process may otherwise deadlock. This can

be done by awaiting on `asyncio.subprocess.Process.communicate()` on the returned `asyncio.subprocess.Process` or by manually reading from the streams as necessary.

Returns a reference to the child process created for use by the parent program.

#### Parameters

- **input\_data** (*bytes*) – input data for FFmpeg to deal with (audio, video etc.) as bytes (e.g. the result of reading a file in binary mode)
- **stdout** – Where to redirect FFmpeg `stdout` to. Default is *None*, meaning no redirection.
- **stderr** – Where to redirect FFmpeg `stderr` to. Default is *None*, meaning no redirection.

**Raises** `FFExecutableNotFoundError` – The executable path passed was not valid.

**Returns** The child process created.

**Return type** `asyncio.subprocess.Process`

#### `wait()`

Asynchronously wait for the process to complete execution.

**Raises** `FFRuntimeError` – The process exited with an error.

**Returns** 0 if the process finished successfully, or *None* if it has not been started

**Return type** `int` or *None*

**class** `ffmpy3.FFprobe` (*executable='ffprobe', global\_options='', inputs=None*)  
 Wrapper for `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.

#### Parameters

- **executable** (*str*) – absolute path to `ffprobe` executable
- **global\_options** (*iterable*) – global options passed to `ffprobe` 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

## 3.2 Examples

- *Format conversion*
- *Transcoding*
- *Demultiplexing*
- *Multiplexing*
- *Using pipe protocol*
- *Asynchronous execution*
- *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 ffmpy3 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
'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" "
>>>
>>> ff.run(stderr=PIPE)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Users/ay/projects/personal/ffmpy3/ffmpy3.py", line 104, in run
    raise FFRuntimeError(self.cmd, ff_command.returncode, out[0], out[1])
ffmpy3.FFRuntimeError: `ffmpeg -hide_banner -i input.ts "-map 0:1" "-c:a copy" "-f mp4" audio.mp4 "
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 r
>>> ff.run()
```

### 3.2.5 Using pipe protocol

ffmpy3 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 ffmpy3 on STDIN; ffmpy3 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 Asynchronous execution

In certain cases, one may not wish to run FFmpeg and block on waiting for results or introduce multithreading into one's application. In this case, asynchronous execution using `asyncio` is possible.

```
>>> ff = ffmpy3.FFmpeg(
...     inputs={'input.mp4': None},
...     outputs={'output.avi': None}
... )
>>> ff.run_async()
>>> await ff.wait()
```

Processing FFmpeg output without multithreading or blocking is also possible. The following code snippet replaces CR with LF from FFmpeg's progress output and echoes it to STDERR while FFmpeg processes the input video.

```
>>> import asyncio
>>> import sys
>>> ff = ffmpy3.FFmpeg(
...     inputs={'input.mp4': None},
...     outputs={'output.avi': None},
... )
>>> await ff.run_async(stderr=asyncio.subprocess.PIPE)
>>> line_buf = bytearray()
>>> while True:
>>>     in_buf = (await my_stderr.read(128)).replace(b'\r', b'\n')
>>>     if not in_buf:
>>>         break
>>>     line_buf.extend(in_buf)
>>>     while b'\n' in line_buf:
>>>         line, _, line_buf = line_buf.partition(b'\n')
>>>         print(str(line), file=sys.stderr)
>>> await ff.wait()
```



### 3.2.7 Complex command lines

FFmpeg command line options can get pretty complex, like when using [filtering](#). Therefore, it is important to understand some of the rules for building command lines building with `ffmpy3`. 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=2
```

In `ffmpy3` 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=2"}
... )
>>> ff.cmd
'ffmpeg -i input.ts -vf "drawtext=fontfile=/Library/Fonts/Verdana.ttf: timecode=\'09\57\00\00\': r=2'
```

The same command line can be compiled by passing output options 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=2"}]}
... )
>>> ff.cmd
'ffmpeg -i input.ts -vf "drawtext=fontfile=/Library/Fonts/Verdana.ttf: timecode=\'09\57\00\00\': r=2'
```



**f**

ffmpeg3, 7



## C

cmd (ffmpy3.FFRuntimeError attribute), 7

## E

exit\_code (ffmpy3.FFRuntimeError attribute), 7

## F

FFExecutableNotFoundError, 7

FFmpeg (class in ffmpy3), 7

ffmpy3 (module), 7

FFprobe (class in ffmpy3), 9

FFRuntimeError, 7

## R

run() (ffmpy3.FFmpeg method), 8

run\_async() (ffmpy3.FFmpeg method), 8

## S

stderr (ffmpy3.FFRuntimeError attribute), 7

stdout (ffmpy3.FFRuntimeError attribute), 7

## W

wait() (ffmpy3.FFmpeg method), 9