MoviePy v2.0 have introduced breaking changes, see Updating from v1.X to v2.X for more info.

Modifying clips and apply effects

Of course, once you will have loaded a `Clip` the next step of action will be to modify it to be able to integrate it in your final video.

To modify a clip, there is three main courses of actions :

- The built-in methods of `VideoClip` or `AudioClip` modifying the properties of the object.
- The already-implemented effects of MoviePy you can apply on clips, usually affecting the clip by applying filters on each frame of the clip at rendering time.
- The transformation filters that you can apply using `transform()` and `time_transform()`.

How modifications are applied to a clip ?

Clip copy during modification

The first thing you must know is that when modifying a clip, MoviePy **will never modify that clip directly**. Instead it will return **a modified copy of the original** and let the original untouched. This is known as out-place instead of in-place behavior.

To illustrate:

```
# Import everything needed to edit video clips
from moviepy import VideoFileClip

# Load example.mp4
clip = VideoFileClip("example.mp4")
```

Skip to main content

```
# which you will loose immediatly as you don't store it
# If you was to render clip now, the audio would still be at full volume
clip.with_volume_scaled(0.1)

# This create a copy of clip in clip_whisper with a volume of only 10% the origina
# but does not modify the original clip
# If you was to render clip right now, the audio would still be at full volume
# If you was to render clip_whisper, the audio would be a 10% of the original volu
clip_whisper = clip.with_volume_scaled(0.1)

# This replace the original clip with a copy of it where volume is only 10% of
# the original. If you was to render clip now, the audio would be at 10%
# The original clip is now lost
clip = clip.with_volume_scaled(0.1)
```

This is an important point to understand, because it is one of the most recurrent source of bug for newcomers.

Memory consumption of effect and modifications

When applying an effect or modification, it does not immediately apply the effect to all the frames of the clip, but only to the first frame: all the other frames will only be modified when required (that is, when you will write the whole clip to a file or when you will preview it).

It means that creating a new clip is neither time nor memory hungry, all the computation happen during the final rendering.

Time representations in MoviePy

Many methods that we will see accept duration or timepoint as arguments. For instance **`clip.subclipped(t_start, t_end)`** which cuts the clip between two timepoints.

MoviePy usually accept duration and timepoint as either:

- a number of seconds as a `float`.
- a `tuple` with `(minutes, seconds)` or `(hours, minutes, seconds)`.
- a `string` such as `'00:03:50.54'`.

Also, you can usually provide negative times, indicating a time from the end of the clip. For example, `clip.subclipped(-20, -10)` cuts the clip between 20s before the end and 10s before

Skip to main content

Modify a clip using the `with_*` methods

The first way to modify a clip is by modifying internal properties of your object, thus modifying his behavior.

These methods usually start with the prefix `with_` or `without_`, indicating that they will return a copy of the clip with the properties modified.

So, you may write something like:

```
from moviepy import VideoFileClip

myclip = VideoFileClip("example.mp4")
myclip = myclip.with_end(5)  # stop the clip after 5 sec
myclip = myclip.without_audio()  # remove the audio of the clip
```

In addition to the `with_*` methods, a handful of very common methods are also accessible under shorter names:

- `resized()`
- `crop()`
- `rotate()`

For a list of all those methods, see `Clip` and `VideoClip`.

Modify a clip using effects

The second way to modify a clip is by using effects that will modify the frames of the clip (which internally are no more than numpy arrays) by applying some sort of functions on them.

MoviePy come with many effects implemented in `moviepy.video.fx` for visual effects and `moviepy.audio.fx` for audio effects. For practicality, these two modules are loaded in MoviePy as `vfx` and `afx`, letting you import them as `from moviepy import vfx, afx`.

To use these effects, you simply need to instantiate them as object and apply them on your `Clip` using method `with_effects()`, with a list of `Effect` objects you want to apply.

For convenience the effects are also dynamically added as method of `VideoClip` and

Skip to main content

So, you may write something like:

```python
from moviepy import VideoFileClip
from moviepy import vfx, afx

myclip = VideoFileClip("example.mp4")
# resize clip to be 460px in width, keeping aspect ratio
myclip = myclip.with_effects([vfx.Resize(width=460)])

# fx method return a copy of the clip, so we can easily chain them
# double the speed and half the audio volume
myclip = myclip.with_effects([vfx.MultiplySpeed(2), afx.MultiplyVolume(0.5)])

# because effects are added to Clip at runtime, you can also call
# them directly from your clip as methods
myclip = myclip.with_effects([vfx.MultiplyColor(0.5)])  # darken the clip
```

Note

MoviePy effects are automatically applied to both the sound and the mask of the clip if it is relevant, so that you don't have to worry about modifying these.

For a list of those effects, see `moviepy.video.fx` and `moviepy.audio.fx`.

In addition to the effects already provided by MoviePy, you can obviously Creating your own effects and use them the same way.

Modify a clip appearance and timing using filters

In addition to modifying a clip's properties and using effects, you can also modify the appearance or timing of a clip by using your own custom *filters* with `time_transform()`, `image_transform()`, and more generally with `transform()`.

All these methods work by taking as first parameter a callback function that will receive either a clip frame, a timepoint, or both, and return a modified version of these.

Modify only the timing of a Clip

Skip to main content

You can change the timeline of the clip with `time_transform(your_filter)`. Where `your_filter` is a callback function taking clip time as a parameter and returning a new time:

```
from moviepy import VideoFileClip
import math

my_clip = VideoFileClip("example.mp4")

# Let's accelerate the video by a factor of 3
modified_clip1 = my_clip.time_transform(lambda t: t * 3)
# Let's play the video back and forth with a "sine" time-warping effect
modified_clip2 = my_clip.time_transform(lambda t: 1 + math.sin(t))
```

Now the clip `modified_clip1` plays three times faster than `my_clip`, while `modified_clip2` will be oscillating between 00:00:00 to 00:00:02 of `my_clip`. Note that in the last case you have created a clip of infinite duration (which is not a problem for the moment).

Note

By default `time_transform()` will only modify the clip main frame, without modifying clip audio or mask for `VideoClip`.

If you wish to also modify audio and/or mask you can provide the parameter `apply_to` with either `'audio'`, `'mask'`, or `['audio', 'mask']`.

Modifying only the appearance of a Clip

For `VideoClip`, you can change the appearance of the clip with `image_transform(your_filter)`. Where `your_filter` is a callback function, taking clip frame (a numpy array) as a parameter and returning the transformed frame:

```
"""Let's invert the green and blue channels of a video."""

from moviepy import VideoFileClip
import numpy

my_clip = VideoFileClip("example.mp4")


def invert_green_blue(image: numpy.ndarray) -> numpy.ndarray:
    return image[:, :, [0, 2, 1]]
```

Skip to main content

```
modified_clip1 = my_clip.image_transform(invert_green_blue)
```

Now the clip `modified_clip1` will have his green and blue canals inverted.

Note

You can define if transformation should be applied to audio and mask same as for `time_transform()`.

Note

Sometimes need to treat clip frames and mask frames in a different way. To distinguish between the two, you can always look at their shape, clips are `H*W*3`, and masks `H*W`.

Modifying both the appearance and the timing of a Clip

Finally, you may want to process the clip by taking into account both the time and the frame picture, for example to apply visual effects variating with time. This is possible with the method `transform(your_filter)`. Where `your_filter` is a callback function taking two parameters, and returning a new frame picture. Where first argument is a `get_frame` method (i.e. a function `get_frame(time)` which given a time returns the clip's frame at that time), and the second argument is the time.

```
"""Let's create a scolling video effect from scratch."""

from moviepy import VideoFileClip

my_clip = VideoFileClip("example.mp4")


def scroll(get_frame, t):
    """
    This function returns a 'region' of the current frame.
    The position of this region depends on the time.
    """
    frame = get_frame(t)
    frame_region = frame[int(t) : int(t) + 360, :]
    return frame_region
```

Skip to main content

```
modified_clip1 = my_clip.transform(scroll)
```

This will scroll down the clip, with a constant height of 360 pixels.

Note

You can define if transformation should be applied to audio and mask same as for `time_transform()`.

Note

When programming a new effect, whenever it is possible, prefer using `time_transform` and `image_transform` instead of `transform` when implementing new effects. The reason is that, though they both internally rely on `transform` when these effects are applied to `ImageClip` objects, MoviePy will recognize they only need to be applied once instead of on each frame, resulting in faster renderings.

To keep things simple, we have only addressed the case of `VideoClip`, but know that the same principle applies to `AudioClip`, except that instead of a picture frame, you will have an audio frame, which is also a numpy array.

Previous
Loading resources as clips

Next
Creating your own effects