vedit Documentation

Release 0.0.1rc1

Matthew Hayward

October 26, 2016

Contents

Contents:

CHAPTER 1

vedit Overview

Project homepage at: https://github.com/digitalmacgyver/vedit

vedit is a Python library that simplifies editing and combining video files using `ffmpeg`.

Examples of the sorts of things vedit makes easy include:

- Extracting a clip from a longer video
- Combining videos or clips together by concatenating them end to end
- Composing videos together, for example side by side or overlayed on top of a background image
- Changing the resolution of a video and cropping or padding a video to change its aspect ratio
- Overlaying images onto a video
- Adding an audio track (like a song) to a video

There are numerous stumbling blocks to these tasks - vedit makes the above things easy by automatically handling videos with:

- Different resolutions
- Different sample aspect ratios
- Different pixel formats
- Different frame rates
- Different audio streams and channels

CHAPTER 2

Installation

Assuming you have `pip` installed:

```
pip install vedit
```

However, there is nothing in the package that is special. The only dependencies are the `future` module and a Python 2.7 or later (including Python 3) interpreter. You can just download from the project GitHub repository and put the `vedit` directory in your Python path of an interpreter that also has `future` installed.

CHAPTER 3

Before You Begin

vedit depends on the `ffmpeg` and `ffprobe` programs from the FFmpeg project, and on the libx264 video codec and the libfdk_aac audio codec, for example by configuring `ffmpeg` for compilation with:

> ./configure –enable-gpl –enable-libx264 –enable-nonfree –enable-libfdk-aac –enable-libfreetype –enable-libfontconfig

(`--enable-libfreetype --enable-libfontconfig` only needed if the `audio_desc` option is used).

CHAPTER 4

Table of Contents

CHAPTER 5

Examples

To get an idea of the sorts of things you can do with a few lines of code, consider these examples, which can be generated from the `examples.py` script in the root directory of the `vedit` Python module.

All the examples below assume that FFmpeg is installed as described in *Before You Begin*.

All the examples below begin with the following boilerplate, and assume the `./example_output` directory exists:

```
#!/usr/bin/env python

import vedit
import logging
logging.basicConfig()
log = logging.getLogger()
log.setLevel( logging.DEBUG )
```

Back to *Table of Contents*

5.1 Example 1: Clip 2 seconds out of the middle of a video

Link to input for example: https://youtu.be/9ul6rWAewd4

Link to example output: https://youtu.be/FEr6WMUx_4A

```
# Clipping 2 seconds out of source video from 1.5 seconds to 3.5 seconds.
source = vedit.Video( "./examples/testpattern.mp4" )
output_file = "./example_output/example01.mp4"
clip = vedit.Clip( video=source, start=1.5, end=3.5 )
window = vedit.Window( width=source.get_width(),
                       height=source.get_height(),
                       output_file=output_file )
window.clips = [ clip ]
window.render()
log.info( "Output file at %s" % ( output_file ) )
```

Back to *Table of Contents*

5.2 Example 2: Resize a video with PAD, CROP, or PAN

Link to source input: https://youtu.be/Qmbgrr6WJEY

Links to example outputs:

- Padded clip: https://youtu.be/2bTdwEzraxA
- Panned clip: https://youtu.be/lCpbnudnFyc
- Cropped clip: https://youtu.be/96v-KVq9B-g

```
# Turning a 1280x720 16:9 input video into a 640x480 4:3 video.
source = vedit.Video( "./examples/d005.mp4" )
clip = vedit.Clip( video=source )

#Since the input and output aspect ratios don't match, pad the input onto a blue background.
pad_output = "./example_output/example02-pad.mp4"
pad_display = vedit.Display( display_style=vedit.PAD, pad_bgcolor="Blue" )
window = vedit.Window( width=640, height=480,
                       display=pad_display,
                       output_file=pad_output )
window.clips = [ clip ]
window.render()
log.info( "Pad output file at: %s" % ( pad_output ) )

# Render a cropped version as well. Note the watermark is getting cropped out on the right.
crop_output = "./example_output/example02-crop.mp4"
crop_display = vedit.Display( display_style=vedit.CROP )
window = vedit.Window( width=640, height=480,
                       display=crop_display,
                       output_file=crop_output )
window.clips = [ clip ]
window.render()
log.info( "Crop output file at: %s" % ( crop_output ) )

# Render a version where we pan over the input image as it plays as well. Note the watermark moves fi
pan_output = "./example_output/example02-pan.mp4"
pan_display = vedit.Display( display_style=vedit.PAN )
window = vedit.Window( width=640, height=480,
                       display=pan_display,
                       output_file=pan_output )
window.clips = [ clip ]
window.render()
log.info( "Pan output file at: %s" % ( pan_output ) )
```

Back to *Table of Contents*

5.3 Example 3: Put two videos next to each other

Example output: https://youtu.be/fsYw2jLyuQ4

```
# Lets set up some source videos, and some clips for use below.
video_1 = vedit.Video( "./examples/i030.mp4" )

# Put two clips from video 1 side by side, with audio from the
```

```
# left clip only, ending after 8 seconds (we could also use clips
# from different videos).
clip_1_0_5 = vedit.Clip( video=video_1, start=0, end=5 )
clip_1_10_20 = vedit.Clip( video=video_1, start=10, end=20,
                           display=vedit.Display( include_audio=False ) )

# Set up two windows, one for each clip, and one to hold the other two, and set the duration.
#
# Since clip 1 is 5 seconds long and we are making an 8 second
# video, there will be time when clip 1 is not playing - set the
# background color to green during this time.
output_file = "./example_output/example03.mp4"
base_window = vedit.Window( width=1280*2, height=720, duration=8, bgcolor='Green',
                            output_file=output_file )
# Set the x, y coordinates of this window inside its parent, as
# measure from the top right.
#
# Here we are putting the videos flush side by side, but they
# could be on top of each other, overlapping, centered in a much
# larger base_window, etc., etc..
clip_1_window = vedit.Window( width=1280, height=720, x=0, y=0, clips=[ clip_1_0_5 ] )
clip_2_window = vedit.Window( width=1280, height=720, x=1280, y=0, clips=[ clip_1_10_20 ] )
base_window.windows = [ clip_1_window, clip_2_window ]
base_window.render()
log.info( "Side by side output is at: %s" % ( output_file ) )
```

Back to *Table of Contents*

5.4 Example 4: Replace the audio track of a video

Example outputs:

- Not attributed: https://youtu.be/4Z2Uigssc88
- Attributed song: https://youtu.be/ojgAs5A5bSg

```
source = vedit.Video( "./examples/i010.mp4" )
output_file = "./example_output/example04.mp4"
# Get a clip, but override any Window settings for its audio.
clip = vedit.Clip( video=source, display=vedit.Display( include_audio=False ) )
# Give this window it's own audio track, and set the duration to
# 10 seconds (otherwise it will go on as long as the audio track).
#
# Note - if the window audio track is longer than the video
# content, it fades out starting 5 seconds from the end.
window = vedit.Window( audio_file="./examples/a2.mp4", duration=10,
                       output_file=output_file )
window.clips = [ clip ]
window.render()
log.info( "Replaced audio in output: %s" % ( output_file ) )

# Let's make a version where we attribute the audio with some text.
song_attribution = '''This video features the song:
Chuckie Vs Hardwell Vs Sandro Silva Vs Cedric & Quintino
EPIC CLARITY JUMP- (NC MASHUP) LIVE
```

```
By: NICOLE CHEN
Available under under a Creative Commons License:
http://creativecommons.org/licenses/by/3.0/ license'''

output_file = "./example_output/example04-attributed.mp4"
window = vedit.Window( audio_file="./examples/a2.mp4",
                       audio_desc=song_attribution,
                       duration=10,
                       output_file=output_file )
window.clips = [ clip ]
window.render()
log.info( "Replaced audio in output: %s" % ( output_file ) )
```

Back to *Table of Contents*

5.5 Example 5: Overlay videos on top of other videos

Example outputs:

- All audio tracks (bleagh): https://youtu.be/lqLLlXPYg3c
- Just one audio track: https://youtu.be/hL0t3RXHKAM

```
# Let's overlay two smaller windows on top of a base video.
base_video = vedit.Video( "./examples/i030.mp4" )
base_clip = vedit.Clip( video=base_video )
output_file = "./example_output/example05.mp4"
# Use the default width, height, and display parameters:
# 1280x1024, which happens to be the size of this input.
base_window = vedit.Window( clips = [ base_clip ],
                            output_file=output_file )

# We'll create two smaller windows, each 1/3 the size of the
# base_window, and position them towards the top left, and bottom
# right of the base window.
overlay_window1 = vedit.Window( width=base_window.width/3, height=base_window.height/3,
                                x=base_window.width/12, y=base_window.height/12 )
overlay_window2 = vedit.Window( width=base_window.width/3, height=base_window.height/3,
                                x=7*base_window.width/12, y=7*base_window.height/12 )

# Now let's put some clips in each of the overlay windows.
window_1_clips = [
    vedit.Clip( video=vedit.Video( "./examples/d007.mp4" ) ),
    vedit.Clip( video=vedit.Video( "./examples/d006.mp4" ) ),
]
window_2_clips = [
    vedit.Clip( video=vedit.Video( "./examples/p006.mp4" ) ),
    vedit.Clip( video=vedit.Video( "./examples/p007.mp4" ) ),
    vedit.Clip( video=vedit.Video( "./examples/p008.mp4" ) ),
]

# Now let's embed the clips in the windows, and the overlay
# windows in our base_window and render.
overlay_window1.clips = window_1_clips
overlay_window2.clips = window_2_clips
```

```
base_window.windows = [ overlay_window1, overlay_window2 ]
base_window.render()
log.info( "Made multi-video composition at: %s" % ( output_file ) )

# Well - the last video looks OK, but it sounds terrible - the
# audio from all the videos are being mixed together.
#
# Let's try again but exclude audio from everything but the base
# video.
output_file = "./example_output/example05-single-audio.mp4"
no_audio_display_config = vedit.Display( include_audio=False )
no_audio_overlay_window1 = vedit.Window( width=base_window.width/3, height=base_window.height/3,
                                x=base_window.width/12, y=base_window.height/12,
                                display=no_audio_display_config )
no_audio_overlay_window2 = vedit.Window( width=base_window.width/3, height=base_window.height/3,
                                x=7*base_window.width/12, y=7*base_window.height/12,
                                display=no_audio_display_config )

# Now let's embed the clips in the windows, and the overlay
# windows in our base_window and render.
no_audio_overlay_window1.clips = window_1_clips
no_audio_overlay_window2.clips = window_2_clips
base_window.output_file = output_file
base_window.windows = [ no_audio_overlay_window1, no_audio_overlay_window2 ]
base_window.render()
log.info( "Made multi-video composition with single audio track at: %s" % ( output_file ) )
```

Back to *Table of Contents*

5.6 Example 6: Cascade overlayed videos and images on top of a base video or image

Example output: https://youtu.be/K2SuPqWrG3M

```
import glob
import random

# The OVERLAY display_style when applied to a clip in the window
# makes it shrink a random amount and be played while it scrolls
# across the base window.
#
# Let's use that to combine several things together and make a
# huge mess!
output_file = "./example_output/example06.mp4"
base_video = vedit.Video( "./examples/i030.mp4" )

# Let's use a different audio track for this.
base_clip = vedit.Clip( video=base_video, display=vedit.Display( include_audio=False ) )
base_window = vedit.Window( clips = [ base_clip ],
                            output_file=output_file,
                            duration=30,
                            audio_file="./examples/a2.mp4" )

# Turn our cat images into clips of random length between 3 and 6
```

```
# seconds and have them cascade across the screen from left to
# right.
cat_display = vedit.Display( display_style=vedit.OVERLAY,
                             overlay_direction=vedit.RIGHT,
                             include_audio=False,
                             overlay_concurrency=4,
                             overlay_min_gap=0.8 )
cat_clips = []
for cat_pic in glob.glob( "./examples/cat*jpg" ):
    cat_video_file = vedit.gen_background_video( bgimage_file=cat_pic,
                                                 duration=random.randint( 3, 6 ) )
    cat_video = vedit.Video( cat_video_file )
    cat_clips.append( vedit.Clip( video=cat_video, display=cat_display ) )

# Turn our dog images into clips of random length between 2 and 5
# seconds and have them cascade across the screen from top to
# bottom.
dog_display = vedit.Display( display_style=vedit.OVERLAY,
                             overlay_direction=vedit.DOWN,
                             include_audio=False,
                             overlay_concurrency=4,
                             overlay_min_gap=0.8 )
dog_clips = []
for dog_pic in glob.glob( "./examples/dog*jpg" ):
    dog_video_file = vedit.gen_background_video( bgimage_file=dog_pic,
                                                 duration=random.randint( 3, 6 ) )

    dog_video = vedit.Video( dog_video_file )
    dog_clips.append( vedit.Clip( video=dog_video, display=dog_display ) )

# Throw in the clips from the p series of videos of their full
# duration cascading from bottom to top.
pvideo_display = vedit.Display( display_style=vedit.OVERLAY,
                                overlay_direction=vedit.UP,
                                include_audio=False,
                                overlay_concurrency=4,
                                overlay_min_gap=0.8 )
pvideo_clips = []
for p_file in glob.glob( "./examples/p0*mp4" ):
    pvideo_video = vedit.Video( p_file )
    pvideo_clips.append( vedit.Clip( video=pvideo_video, display=pvideo_display ) )

# Shuffle all the clips together and add them onto the existing
# clips for the base_window.
overlay_clips = cat_clips + dog_clips + pvideo_clips
random.shuffle( overlay_clips )
base_window.clips += overlay_clips
base_window.render()
log.info( "Goofy mashup of cats, dogs, and drone videos over Icelandic countryside at: %s" % ( output
```

Note: Since the composition of this video involves several random elements, the output you get will not be the same as the example output below.

Back to *Table of Contents*

5.7 Example 7: Add an overlay image, such as a watermark

Example output: https://youtu.be/1PrADMtqdRU

```
import glob

# Let's make our background an image with a song.
output_file = "./example_output/example07.mp4"
dog_background = vedit.Window( bgimage_file="./examples/dog03.jpg",
                               width=960, #The dimensions of this image
                               height=640,
                               duration=45,
                               audio_file="./examples/a3.mp4",
                               output_file=output_file )

# Let's put two windows onto this image, one 16:9, and one 9:16.
horizontal_window = vedit.Window( width = 214,
                                  height = 120,
                                  x = (960/2-214)/2, # Center it horizontally on the left half.
                                  y = 80,
                                  display=vedit.Display( include_audio=False, display_style=vedit.CROP
vertical_window = vedit.Window( width=120,
                                height=214,
                                x = 740,
                                y = (640-214)/2, # Center it vertically.
                                display=vedit.Display( include_audio=False, display_style=vedit.PAN )

# Let's let the system distribute a bunch of our 3 second clips
# among the horizontal and vertical windows automatically.
video_clips = []
for video_file in glob.glob( "./examples/*00[5-9].mp4" ):
    video_clips.append( vedit.Clip( end=3, video=vedit.Video( video_file ) ) )

# With these options this will randomize the input clips among
# the two windows, and keep recycling them until the result is 45
# seconds long.
vedit.distribute_clips( clips=video_clips,
                        windows=[ horizontal_window, vertical_window ],
                        min_duration=45,
                        randomize_clips=True )

# Add the overlay windows to the background.
dog_background.windows = [ horizontal_window, vertical_window ]

# Let's set up a watermark image to show over the front and end of
# out video. The transparent01.png watermark image is 160x160
# pixels.
#
# Let's put it in the top left for the first 10 seconds.
front_watermark = vedit.Watermark( filename="./examples/transparent01.png",
                                   x=0,
                                   y=0,
                                   fade_out_start=7,
                                   fade_out_duration=3 )
# Let's put it in the bottom right for the last 15 seconds.
back_watermark = vedit.Watermark( filename="./examples/transparent01.png",
                                  x=dog_background.width-160,
                                  y=dog_background.height-160,
```

```
                                    fade_in_start=-15, # Negative values are times from the end of the
                                    fade_in_duration=5 )

# Add watermarks to the background.
dog_background.watermarks = [ front_watermark, back_watermark ]

dog_background.render()
log.info( "Random clips over static image with watermarks at: %s" % ( output_file ) )
```

Back to *Table of Contents*

CHAPTER 6

Module Concepts

There are four main classes in the `vedit` module:

`Video` `Video` represents a given video or image file on the filesystem.

`Clip` `Clip` represents a portion of a video with a given start and end time. When associated with a `Window` and a `Display` a `Clip` can be rendered into an output video.

`Display` `Display` configures the properties that a given `Clip` has when it is rendered into a given `Window`.

`Window` `Window` objects are the building blocks that are used to compose `Clip` objects together. The `width` and `height` properties of a `Window` determine the size of a `Clip` when it is rendered in that `Window`. In basic usage one or more `Clip` objects are associated with a `Window` which is then rendered. In more advanced usage a `Window` can include any number of child `Window` and `Clip` objects to create complex outputs where several different clips play at the same time.

Back to *Table of Contents*

6.1 Display Configuration

The `Display` object contains configuration that dictates how a given `Clip` appears when the `Window` it is in is rendered.

Each `Clip` can its own `Display`, and so can each `Window`. When considering what `Display` settings to use for a given `Clip` the following selections are made:

1. If the `Clip` has a `Display` object, it is used.
2. Otherwise, if the `Window` has a `Display` object, it is used.
3. Otherwise, the `Default` display elements described below are used.

Constructor arguments:

Argument	Required	Default	Description
display_style	No	vedit.PAD	One of vedit.CROP, PAD, PAN, or OVERLAY
overlay_concurrency	No	3	If display_style is OVERLAY, how many Clips may cascade at the same time
overlay_direction	No	vedit.DOWN	One of UP, DOWN, LEFT, or RIGHT. If display_style is OVERLAY, what direction the Clips cascade
overlay_min_gap	No	4	If display_style is OVERLAY, the shortest period of time between clips cascade
pad_bgcolor	No	'Black'	If display_style is PAD, what color should be on the background of the Clip in [0x|#]RRGGBB format
pan_direction	No	vedit.ALTERNATE	One of vedit.UP, DOWN, LEFT, or RIGHT. If display_style is PAN, what direction the Window should pan over the Clip
include_audio	No	True	Should audio from this Clip be included in the output

Public methods: None

6.2 The `OVERLAY` `display_style`

This `display_style` makes the `Clip` be rendered as a small (randomly sized between 1/2 and 1/3 of the width of its `Window`) tile that cascades across the `Window` while it plays.

The idea here is to make a collage of images or clips. For a silly example see https://youtu.be/K2SuPqWrG3M - the output for *Example 6: Cascade overlayed videos and images on top of a base video or image*.

When a several `Clips` are rendered in a given `Window` with the `OVERLAY` `display_style` the behavior of the cascading is further controlled by:

- `overlay_concurrency` - The number of clips that can be in the `Window` at once.
- `overlay_direction` - One of `vedit.UP`, `DOWN`, `LEFT`, or `RIGHT`. The `Clip` will move across the `Window` in this direction as it plays.
- `overlay_min_gap` - The shortest time in seconds between when two `Clip` objects will move across the `Window`.

6.3 CROP, PAD, and PAN

`display_style`: When the a `Clip` is rendered in a `Window`, if the `Clip` and the `Window` do not have the same aspect ratio, something must be done to make the `Clip` fit in the `Window`.

If the `display_style` is:

CROP: The `Clip` will be scaled to the smallest size such that both its height and width are at least as large as the `Window` it is in. The `Clip` is then centered in the `Window` and any portions of the `Clip` that fall outside the `Window` are cropped away and discarded.

As in *Example 2: Resize a video with PAD, CROP, or PAN* when: https://youtu.be/Qmbgrr6WJEY is cropped the result is: https://youtu.be/96v-KVq9B-g

PAD: The `Clip` will be scaled to the largest size such that both its height and width are no larger than the `Window` it is in. Then any space in the `Window` not covered by the clip is colored the `pad_bgcolor` color (defaults to black).

As in *Example 2: Resize a video with PAD, CROP, or PAN* when: https://youtu.be/Qmbgrr6WJEY is padded onto a blue background the result is: https://youtu.be/2bTdwEzraxA

PAN: The `Clip` will be scaled to the smallest size such that both its height and width are at least as large as the `Window` it is in. The `Clip` then is scrolled through the `Window` in the direction specified by `pan_direction`. `pan_direction` is one of `UP`/`RIGHT`, `DOWN`/`LEFT`, or `ALTERNATE`.

As in *Example 2: Resize a video with PAD, CROP, or PAN* when: https://youtu.be/Qmbgrr6WJEY is panned with `pan_direction` of `vedit.RIGHT` the result is: https://youtu.be/lCpbnudnFyc

Display Examples:

```
# A display that will crop the input and remove the audio:
crop_silent = Display( display_style=vedit.CROP, include_audio=False )

# A display that will pad the input with a green background and include the audio from it:
pad = Display( display_style=vedit.PAD, pad_bgcolor='Green', include_audio=True )
# Or - more concisely relying on the defaults values for display_style and include_audio:
pad = Display( pad_bgcolor='Green' )

# A display that will have up to 5 clips cascading over the Window
at a time, starting no more than once a second, and moving from top
to bottom:
cascade_5 = Display( display_style=vedit.OVERLAY, overlay_concurrency=5, overlay_min_gap=1 )

# A display that will pan over the input from bottom to top or right to left (depending on whether th
pan_up = Display( display_style=vedit.PAN, pan_direction=vedit.UP )
```

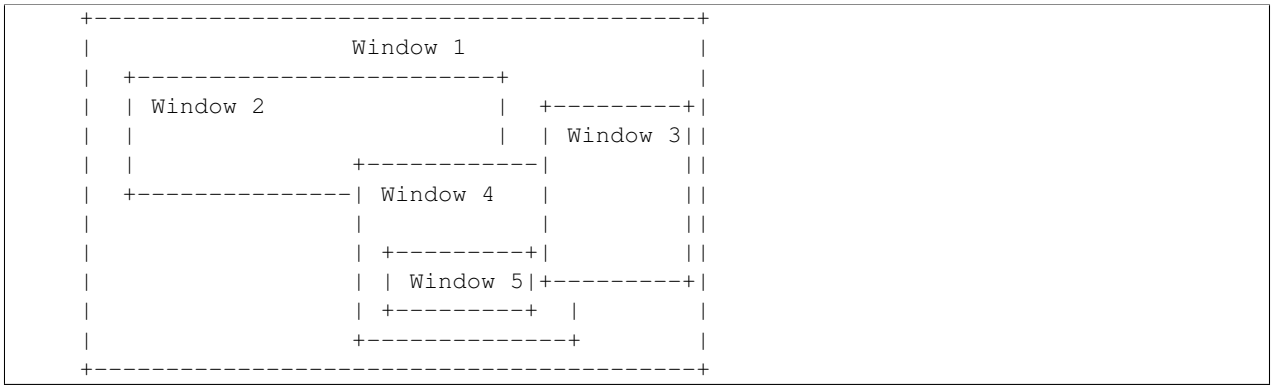
Back to *Table of Contents*

6.4 Windows

The `Window` object is used to compose `Clip` objects together into a rendered video output.

A `Window` has a background of a solid color or static image, and optionally may have:

- A list of `Clip``s that it will show in order (perhaps cascading through the ``Window` as they play if the `Display.display_style` for that `Clip` is `OVERLAY`).
- A list of other `Window` objects that are rendered on top of it, for example several windows can be composed like:



In the example above there are five windows:

- Window 1 has child `Window` objects: Window 2, Window 3, and Window 4
- Window 4 has child `Window`: Window 5

Each of these five `Window` objects would have it's own content of Clips, background images, and/or `Watermark` objects.

Example 5: Overlay videos on top of other videos has an example of two Windows overlayed onto another at: https://youtu.be/hL0t3RXHKAM

The duration of a `Window`'s rendered video output will be:

- The `duration` attribute, if set during construction
- Otherwise, if an `audio_file` is specified during construction, the length of that audio stream
- Otherwise, the longest computed time it will take the clips in this or any of its child windows to play

Constructor arguments: (presented in rough order of importance)

Argument	Required	Default	Description
windows	No	None	A list of child Windows. May be set after construction by assigning to the `.windows` attribute
clips	No	None	A list of Clips to render in this Window. May be set after construction by assigning to the `.clips` attribute
bgcolor	No	'Black'	The background color for this Window that will be shown in regions of this Window that do not otherwise have content (from a Clip, a child Window, or Watermark). May be set with a string in [0x|#]RRGGBB format.
bgimage_file	No	None	If provided, a background image for this Window that will be shown in regions or times where there is not otherwise content. No scaling is done to this image, so it must be sized at the desired width and height.
duration	No	None	If specified, the duration of this Window when rendered. Otherwise will default first to the duration of the optional audio_file for this Window, and then to the maximum duration of the Clips in this Window or any of it's child Windows.
width	No	1280	Width in pixels of this Window
height	No	720	Height in pixels of this Window
output_file	No	./output.mp4	Where to place the output video for when this Window is rendered. Not needed for Windows that are children of other Windows.
display	No	None	An optional Display object that specifies the Display configuration for Clips in this Window. **NOTE**: If a Clip has its own Display object, it will override the Display configuration of the Window it is placed in. The default values are: `display_style=PAD`, `pad_bgcolor='Black'`, `include_audio=True`.
audio_file	No	None	If specified the path to an audio file whose first audio stream will be added to the output of this Window.
x	No	0	If this Window is a child of another Window, the x coordinate of the top left corner of this Window, as measured from the top left of the parent Window
y	No	0	If this Window is the child of another Window, the y coordinate of the top left corner of this Window, as measured from the top left of the parent Window
watermarks	No	None	A list of Watermark objects that can be used to place static images over everything else in this Window at certain times.
audio_desc	No	None	If a string is specified it's text will be placed at the bottom left of the window 5 seconds prior to the end of the video.
z_index	No	None	If not specified Windows will be placed on top of one another in the order they are created, older Windows having lower z_indexes. If specified should be a numeric value, and Windows will be placed underneath other Windows of higher z_index.
pix_fmt	No	None	If specified the pixel format of the output video. Defaults to: yuv420p
sample_aspect_ratio	No	None	The SAR of a video is the aspect ratio of individual pixels. If specified must be in W:H format. The SAR tine `Window` should have when rendered. Defaults to the SAR of the source Video that has provided Clips to this Window. If more than one SAR is present in the inputs a WARNING is issued and 1:1 is used.
overlay_batch_concurrency	No	16	ffmpeg seems to have problems when many overlays are used, resulting in crashes or errors in the resultant video. This parameter configures the maximum number of overlays that will be composed at one time during rendering. If you are having mysterious ffmpeg errors during rendering, try lowering this.

Public methods:

- `.render()` - Compose this `Window`'s: `bgcolor`, `bgimage_file`, `audio_file`, `clips`, child `windows`, `watermarks`, and `audio_desc` into a video of `width` with and `height` height and place the output at `output_file`.
- `compute_duration( clips, include_overlay_timing=False )` - Return a float of how

long the Clips in the `clips` list input would take to render in this `Window`. If the optional `include_overlay_timing` argument is true then instead a tuple will be returned, the first element of which is the duration that would result from the `clips`, and the second is a list of the start and end times of any `clips` whose `Display.display_type` is `OVERLAY`.

Window Examples:

```
# Let's assume we have some existing media objects / files to work with:
clip1 = vedit.Clip( ... )
clip2 = vedit.Clip( ... )
clip3 = vedit.Clip( ... )
watermark = vedit.Watermark( ... )
background_image = "./media/background_01.jpg"
song = "./media/song.mp3"

# A 640x480 window that uses the default Display properties (overridden on a Clip by Clip basis if t
tv = vedit.Window( clips=[ clip1, clip2 ], width=640, height=480 )

# Let's embed the tv window in a 1080x720 window near the top left
# (50 pixels from the left, 60 from the top), with a background_image.
#
# We'll make the hd window 30 seconds long.
#
# We'll add our song to the hd window.
#
# Note: 1080x720 is the default resolution for a Window, so we don't have to set it.

hd = vedit.Window( windows=[ tv ], bgimage_file=background_image, x=50, y=60,
                   duration=30, audio_file=song )

# Let's add a clips to the hd window.
hd.clips.append( clip3 )

# Let's render the result.
#
# Since we didn't set output_file on the hd Window, the output will
# be placed in ./output.mp4
hd.render()
```

Back to *Table of Contents*

6.5 Videos and Clips

The `Video` and `Clip` objects are tightly related.

A `Video` represents a source input file. The primary use of the `Video` object is as an input to the `Clip` object's `video` constructor argument.

Video Constructor arguments:

Argument	Required	Default	Description
filename	Yes	None	The path to a source input file.

Video Public methods:

- `get_width()` - Return the width of this video in pixels

- `get_height()` - Return the height of this video in pixels

Clip Constructor arguments:

Argument	Required	Default	Description
video	Yes	None	A Video object to extract this Clip from
start	No	0	The time in seconds from the start of the Video this Clip should begin at
end	No	End of Video	The time in seconds from the start of the Video this Clip should end at. **NOTE:** The end time is the absolute end time in the source Video, not relative to the start time of this Clip.
display	No	None	If specified, a Display object that determines how this Clip should be rendered

Clip Public methods:

- `get_duration()` - Return the width of this video in pixels
- `get_height()` - Return the height of this video in pixels

Video and Clip Examples:

```
video1 = vedit.Video( "./media/video01.avi" )
video2 = vedit.Video( "./media/video02.wmv" )

# All of video 1
clip1_all = vedit.Clip( video1 )

# Bits of video2, with Display settings that override whatever the
# Display settings of the Windows these are eventually included in.
vid2_display = Display( display_style=vedit.OVERLAY, include_audio=False )
# From second 3-8.5
clip2_a = vedit.Clip( video2, start=3, end=8.5, display=vid2_display )
# From second 12-40
clip2_b = vedit.Clip( video2, start=12, end=40, display=vid2_display )
# From second 99 to the end
clip2_c = vedit.Clip( video2, start=99, display=vid2_display )
```

Back to *Table of Contents*

6.6 Watermarks

The `Watermark` object gives an easy way to place an image or rectangle of a solid color on top of a resulting Window over a certain time in the video.

`Watermark` objects are applied to a Window by sending a list of them to the `watermarks` constructor argument for the `Window`, or can be applied after construction by setting the `.watermarks` attribute of a `Window`.

NOTE: The image file of a watermark is used as is with no scaling, you must ensure the size of the watermark file is appropriate to the size of the `Window` it is placed in.

Constructor arguments:

Argument	Required	Default	Description
filename	Yes	None	Path to an image file to use for the Watermark. Mutually exclusive with bgcolor.
x	No	"0"	Passed to the ffmpeg overlay filter's x argument to position this watermark. Can be a simple numeric value which will be interpreted as a pixel offset from the left, or something more complex like: `"main_w-overlay_w-10"` to position near the right of the screen.
y	No	"0"	Passed to the ffmpeg overlay filter's y argument to position this watermark. Can be a simple numeric value which will be interpreted as a pixel offset from the top, or something more complex like: `"trunc((main_h-overlay_h)/2)"` to position vertically center.
fade_in_start	No	None	If specified the watermark will begin to appear at fade_in_start seconds. Negative values are interpreted as offsets from the end of the video.
fade_in_duration	No	None	If specified, the watermark will fade in over this many seconds to full opacity.
fade_out_start	No	None	If specified, the watermark will begin to vanish at fade_out_start seconds. Negative values are interpreted as offsets from the end of the video.
fade_out_duration	No	None	If specified, the watermark will fade out over this many seconds to full transparency.
bgcolor	No	None	Mutually exclusive with filename. If specified, the width and height arguments are required, and the Watermark will take the form of a rectangle of that size and color.

Watermark Public methods: None

Watermark Examples:

```
# Let's assume we have an existing Window we want to apply watermarks to.
window = vedit.Window( ... )

# And a watermark image.
watermark_file = "./media/watermark_corner.png"

# Let's add the watermark image in the bottom right of the video.
watermark_image = vedit.Watermark( filename=watermark_file, x="main_w-overlay_w-10", y="main_h-overla

# Let's fade in the window from white over 3 seconds.
white_intro = vedit.Watermark( bgcolor='White', width=window.width, height=window.height, fade_out_st

# Let's fade the window out to black over 5 seconds from the end of the video.
black_outro = vedit.Watermark( bgcolor='Black', width=window.width, height=window.height, fade_in_sta

window.watermarks = [ watermark_image, white_intro, black_outro ]

window.render()
```

Back to *Table of Contents*

6.7 Audio

There are a few ways to manipulate the audio of the output:

1. Each Clip can be configured to mix it's audio into the output by virtue of configuring it with a `Display` configuration with `include_audio=True` (the default).
2. Alternatively, if the Clip has no such configuration, the Window it is in can have a `Display` configuration with `include_audio=True`.

3. Finally, each `Window` can have it's own audio track via the `audio_file` constructor argument.

All `Clip` and `Window` who have audio present will see their audio mixed together in the output.

Finally, for `Window` objects with an `audio_file` argument, if the audio file is longer than the `duration` of the window, the volume of that `audio_file` stream will fade out over the last 5 seconds of the duration of the `Window`.

Back to *Table of Contents*

CHAPTER 7

Logging Output

vedit produces lots of output through Python's logging framework. Messages are at these levels:

debug Everything, including command output from `ffmpeg`

info Step by step notifications of commands run, but curtailing the output

warn Only notices where vedit is making some determination about what to do with ambiguous inputs

To enable logging output from a script using `vedit` do something like:

```
import logging
logging.basicConfig()
log = logging.getLogger()
log.setLevel( logging.DEBUG )
```

Back to *Table of Contents*

CHAPTER 8

Getting Help

File an issue on GitHub for this project https://github.com/digitalmacgyver/vedit/issues

Back to *Table of Contents*

CHAPTER 9

Contributing

Feel free to fork and issue a pull request at: https://github.com/digitalmacgyver/vedit

Back to *Table of Contents*

CHAPTER 10

Odds and Ends

- The first video stream encountered in a file is the one used, the rest are ignored.
- The first audio stream encountered in a file is the one used, the rest are ignored.
- The output Sample Aspect Ratio (SAR) for a Window can be set. All inputs and outputs are assumed to have the same SAR. If not set the SAR of the Video input will be used, or 1:1 will be used if there is no Video input.
  - Some video files report strange Sample Aspect Ratio (SAR) via `ffprobe`. The nonsense SAR value of 0:1 is assumed to be 1:1. SAR ratios between 0.9 and 1.1 are assumed to be 1:1.
- The pixel format of the output can be set, the default is yuv420p.
- The output video frame rate will be set to 30000/1001
- The output will be encoded with the H.264 codec.
- The quality of the output video relative to the inputs is set by the ffmpeg -crf option with an argument of 16, which should be visually lossless.
- If all input clips have the same number of audio channels, those channels are in the output. In any other scenario the resultant video will have a single channel (mono) audio stream.

Back to *Table of Contents*

CHAPTER 11

Indices and tables

- genindex
- modindex
- search