

Markov Groove Documentation

Contents

Module markov_groove	2
Sub-modules	2
Module markov_groove.audio_file	2
Classes	2
Class AudioFile	2
Class variables	3
Instance variables	3
Static methods	3
Methods	3
Module markov_groove.onset_detector	4
Classes	4
Class OnsetAlgorithm	4
Ancestors (in MRO)	4
Class variables	4
Class OnsetDetector	5
Class variables	5
Methods	5
Class Window	5
Ancestors (in MRO)	6
Class variables	6
Module markov_groove.sampler	6
Classes	6
Class KeyFunction	6
Ancestors (in MRO)	7
Class variables	7
Class Sampler	7
Class variables	7
Static methods	7
Module markov_groove.sequencer	8
Sub-modules	8
Module markov_groove.sequencer.audio_sequencer	8
Classes	8
Class AudioSequencer	8
Ancestors (in MRO)	8
Class variables	8
Static methods	9
Methods	9
Module markov_groove.sequencer.midi_sequencer	9
Classes	9
Class MidiSequencer	9

Ancestors (in MRO)	10
Class variables	10
Static methods	10
Methods	10
Module <code>markov_groove.sequencer.sequencer</code>	11
Classes	11
Class <code>Sequencer</code>	11
Ancestors (in MRO)	11
Descendants	11
Static methods	11
Methods	11
Module <code>markov_groove.util</code>	12
Functions	12
Function <code>create_knowledge_base</code>	12
Function <code>find_closest</code>	12
Function <code>find_closest_samples</code>	13
Function <code>read_audio_files</code>	13
Function <code>read_midi_files</code>	13

Module `markov_groove`

This framework allows for preprocessing audio files to make them usable in a machine learning context. The `markov_groove` framework was created and written by Jan-Niclas de Vries.

Sub-modules

- [markov_groove.audio_file](#)
- [markov_groove.onset_detector](#)
- [markov_groove.sampler](#)
- [markov_groove.sequencer](#)
- [markov_groove.util](#)

Module `markov_groove.audio_file`

This module consist solely of the `AudioFile` class.

Classes

Class `AudioFile`

```
class AudioFile(
    audio: nptyping.types._ndarray.NDArray,
    bpm: int = 0,
    sample_rate: int = 44100
)
```

This class loads up audio files, regardless of their filetype and sampling rate. Downmixes stereo audio files to mono files resamples them to the given rate. When initiating with an array, make sure the sampling rate is correct.

Args —= `audio` : `NDArray[Float32]` : The audio represented in binary form as `np.array` of `float32`.

`bpm` : **int** Optional, provides additional information for future analysis. Defaults to 0.

sample_rate : int The desired sampling rate of the audio file. Needs to match the sampling rate when reading from binary form. Defaults to 44.1 khz

Attributes ---= **audio : NDArray[Float32]** : The audio in binary form as np.array with dtype float32.

file_path : Window The function to apply to every frame.

sample_rate : int The sampling rate.

bpm : int The bpm. This might not be set on init and can be checked with check_bpm().

Class variables

Variable audio Type: nptyping.types._ndarray.NDArray

Variable file_path Type: pathlib.Path

Variable sample_rate Type: int

Instance variables

Variable bpm Type: int

Returns the bpm as integer value.

Static methods

Method from_file

```
def from_file(
    file_path: pathlib.Path,
    bpm: int = 0,
    sample_rate: int = 44100
)
```

Create a new AudioFile object from a file.

Methods

Method check_bpm

```
def check_bpm(
    self
) -> float
```

This method runs an analyzer to determine the BPM. If the audio is shorter than the setted time margin, it is append multiple times with itself to make up for the missing data and increase accuracy.

Method display

```
def display(
    self,
    autoplay: bool = False
)
```

Display a given audio through IPython. Useful when using in notebooks.

Method mix

```
def mix(
    self,
    snd,
    right: bool = True
) -> NoneType
```

Mix the given audio to the right(default) channel.

Method normalize

```
def normalize(
    self
) -> NoneType
```

Normalize the audio, by scaling the raw audio between one and minus one.

Method save

```
def save(
    self,
    file_path: pathlib.Path = PosixPath('/home/jdevries/Workspace/ba/code/.temp/audio.wav')
) -> NoneType
```

Export the AudioFile at the newly given path.

Module markov_groove.onset_detector

The onset_detector module encapsulates the onset detection of the essentia module. The enums provide easy an replicable way to use the certain parameters, that are available for the onset detection.

Classes**Class** OnsetAlgorithm

```
class OnsetAlgorithm(
    value,
    names=None,
    *,
    module=None,
    qualname=None,
    type=None,
    start=1
)
```

This enum provides the names of the different algorithms available.

Ancestors (in MRO)

- [enum.Enum](#)

Class variables

Variable COMPLEX

Variable COMPLEX_PHASE

Variable FLUX

Variable HFC

Variable MELFLUX

Variable RMS

Class OnsetDetector

```
class OnsetDetector(  
    file: markov_groove.audio_file.AudioFile,  
    algo: markov_groove.onset_detector.OnsetAlgorithm,  
    frame_size: int = 1024,  
    hop_size: int = 512,  
    windowfnc: markov_groove.onset_detector.Window = Window.HANN,  
    normalize: bool = True  
)
```

This class provides the onset detection.

Args ---= file : AudioFile : The audio file as AudioFile object.

algo : **OnsetAlgorithm** The algorithm to estimate the onsets.

frameSize : **int** Not recommended to change. Defaults to 1024.

hopSize : **int** Not recommended to change. Default to 512.

windowfnc : **Window** The function to apply to every frame.

normalize : **bool** Normalize each window. Defaults to True.

Attributes ---= algo : str : String representation of the selected algorithm.

onsets : **NDArray[Float32]** The indices of every onsets in seconds.

Class variables

Variable algo Type: str

Variable onsets Type: nptyping.types._ndarray.NDArray

Methods

Method beep

```
def beep(  
    self  
) -> markov_groove.audio_file.AudioFile
```

Create a new AudioFile where the onsets are represented as beep.

Class Window

```
class Window(  
    value,  
    names=None,  
    *,  
    module=None,  
    qualname=None,  
    type=None,
```

```

        start=1
    )

```

This enum provides the names of the different windowing functions available to be used with a the fft.

Ancestors (in MRO)

- [enum.Enum](#)

Class variables

Variable BLACKMANHARRIS62

Variable BLACKMANHARRIS70

Variable BLACKMANHARRIS74

Variable BLACKMANHARRIS92

Variable HAMMING

Variable HANN

Variable HANNSGCQ

Variable SQUARE

Variable TRIANGULAR

Module `markov_groove.sampler`

The sampler module consist mainly of the Sampler class. The Sampler class is used for creating samples from an audio file with their onsets. The length of each sample varies, and is limited by the next onset index. The KeyFunction enum is used to define the keyfunctions, that are used to describe the samples in a TODO: mathematical way.

Classes

Class `KeyFunction`

```

class KeyFunction(
    value,
    names=None,
    *,
    module=None,
    qualname=None,
    type=None,
    start=1
)

```

This enum provides the names of the different keyfunctions available.

Ancestors (in MRO)

- [enum.Enum](#)

Class variables

Variable CENTROID

Variable MAX

Variable MELBANDS

Variable MELBANDS_LOG

Variable MFCC

Variable RMS

Class Sampler

```
class Sampler(  
    onsets: <function array at 0x7f956e527790>,  
    samples: Dict[float, array],  
    sample_rate: int  
)
```

This class holds samples with their matching onset frames. They can be determined automatically by using the `from_audio` constructor.

Args —= `onsets` : List[float] : The onsets of each sample. The length of this has to match the amount of the given samples.

`samples` : **Dict[Any, NDArray[Float32]]** The samples or audio snippets that have been detected, when determining the onsets. The length has to match the amount of the given onsets.

`sample_rate` : **int** The sampling rate of the samples.

Attributes —= `onsets` : List[float] : The onsets of each sample.

`samples` : **Dict[Any, NDArray[Float32]]** The samples.

`sample_rate` : **int** The sampling rate of the samples.

Class variables

Variable `onsets` Type: List[float]

Variable `sample_rate` Type: int

Variable `samples` Type: Dict[Any, nptypes._ndarray.NDArray]

Static methods

Method from_audio

```
def from_audio(
    audio: markov_groove.audio_file.AudioFile,
    windowfnc: markov_groove.onset_detector.Window = Window.HANN,
    onsets: <function array at 0x7f956e527790> = None,
    onset_algorithm: markov_groove.onset_detector.OnsetAlgorithm = OnsetAlgorithm.COMPLEX,
    keyfnc_type: markov_groove.sampler.KeyFunction = KeyFunction.CENTROID
)
```

Creates a sampler from a given AudioFile. Detects the onsets via OnsetDetector, when no onsets are given.

Args: audio (AudioFile): The audio represented as AudioFile object. windowfnc (Window): The windowing function both used in the onset detection and to estimate the key features. onsets (List[float]): Optional, provides additional information for future analysis. Defaults to 0. samples (int): The desired sampling rate of the audio file. Needs to match the sampling rate when reading from binary form. Defaults to 44.1 khz

Module markov_groove.sequencer

Sub-modules

- [markov_groove.sequencer.audio_sequencer](#)
- [markov_groove.sequencer.midi_sequencer](#)
- [markov_groove.sequencer.sequencer](#)

Module markov_groove.sequencer.audio_sequencer

Classes

Class AudioSequencer

```
class AudioSequencer(
    pattern: nptyping.types._ndarray.NDArray,
    bpm: int,
    beats: int,
    steps: int
)
```

TODO: This is part of bla

Ancestors (in MRO)

- [markov_groove.sequencer.sequencer.Sequencer](#)
- [abc.ABC](#)

Class variables

Variable beats Type: Final[int]

Variable bpm Type: Final[int]

Variable pattern Type: Final[nptypes._ndarray.NDArray]

Variable steps Type: Final[int]

Static methods

Method decode

```
def decode(
    string_pattern: List[str],
    bpm: int,
    beats: int,
    steps: int
)
```

Decodes the pattern of a string and create a sequencer from it.

Method from_sampler

```
def from_sampler(
    sampler: markov_groove.sampler.Sampler,
    bpm: int,
    beats: int = 8,
    steps: int = 16
)
```

Create a sequencer with an audio sampler by creating the pattern from it.

Methods

Method create_beat

```
def create_beat(
    self,
    samples: Dict[float, nptypes._ndarray.NDArray] = None,
    sample_rate: int = 44100
) -> markov_groove.audio_file.AudioFile
```

Create a beat from the given samples, which have to match the length of occurrences in the pattern.

Method encode

```
def encode(
    self
) -> List[str]
```

Encodes the pattern in a string.

Module markov_groove.sequencer.midi_sequencer

Classes

Class MidiSequencer

```
class MidiSequencer(
    pattern: nptyping.types._ndarray.NDArray,
    bpm: int,
    beats: int,
    steps: int
)
```

TODO: This is part of bla

Ancestors (in MRO)

- [markov_groove.sequencer.sequencer.Sequencer](#)
- [abc.ABC](#)

Class variables

Variable beats Type: Final[int]

Variable bpm Type: Final[int]

Variable pattern Type: Final[nptypes._ndarray.NDArray]

Variable steps Type: Final[int]

Static methods

Method decode

```
def decode(
    string_pattern: List[str],
    bpm: int,
    beats: int,
    steps: int
)
```

Decode the pattern of a string list and create a sequencer from it.

Method decode2

```
def decode2(
    string_pattern: List[str],
    bpm: int,
    beats: int,
    steps: int
)
```

Decode the pattern of a string list and create a sequencer from it.

Method from_file

```
def from_file(
    mid: pretty_midi.pretty_midi.PrettyMIDI,
    bpm: int,
    beats: int = 8,
    steps: int = 16
)
```

TODO: This is part of bla

Methods

Method `create_beat`

```
def create_beat(
    self,
    samples: Dict[float, nptypes._ndarray.NDArray] = None,
    sample_rate: int = 44100
) -> markov_groove.audio_file.AudioFile
```

Create a beat from the pattern in the sequencer.

Method `encode`

```
def encode(
    self
) -> List[str]
```

Encode the pattern in a list of strings.

Method `encode2`

```
def encode2(
    self
) -> List[str]
```

Encode the pattern in a list of strings.

Module `markov_groove.sequencer.sequencer`

Classes

Class `Sequencer`

```
class Sequencer
```

A sequencer can be initialized

Ancestors (in MRO)

- [abc.ABC](#)

Descendants

- [markov_groove.sequencer.audio_sequencer.AudioSequencer](#)
- [markov_groove.sequencer.midi_sequencer.MidiSequencer](#)

Static methods

Method `decode`

```
def decode(
    string_pattern: List[str],
    bpm: int,
    beats: int,
    steps: int
)
```

Decode the pattern of a string and create a sequencer from it.

Methods

Method create_beat

```
def create_beat(
    self,
    samples: Dict[float, nptypes._ndarray.NDArray] = None,
    sample_rate: int = 44100
) -> markov_groove.audio_file.AudioFile
```

Create a beat from the pattern in the sequencer. This method requires different parameters for every implementation of Sequencer.

Method encode

```
def encode(
    self
) -> List[str]
```

Encode the pattern in a string.

Method visualize

```
def visualize(
    self,
    ax_subplot,
    color: Union[nptypes._ndarray.NDArray, str],
    marker: str
)
```

Visualize the pattern.

Module markov_groove.util

The util module holds various helper functions, that are useful, when for instance preprocessing larger datasets or reading multiple audio files.

Functions**Function** create_knowledge_base

```
def create_knowledge_base(
    audios: List[markov_groove.audio_file.AudioFile],
    onset_algo: markov_groove.onset_detector.OnsetAlgorithm,
    beats: int,
    steps: int,
    verbose: bool = False,
    keyfnc_type: markov_groove.sampler.KeyFunction = KeyFunction.CENTROID
) -> Tuple[List[markov_groove.sequencer.sequencer.Sequencer], Dict[float, nptypes._ndarray.NDArray]
```

Create the knowledge base from multiple files. Prints the File name and its bpm, as well as when a doubled key was found. Returns a list of sequences and the samples as dict, which can later be looked up.

Function find_closest

```
def find_closest(
    array: nptyping.types._ndarray.NDArray,
    value
)
```

Find the closest value in an array. The value and the values stored in the array can only be of numeric nature. Furthermore the dtype of array and the value have to be the same.

Function `find_closest_samples`

```
def find_closest_samples(
    sequencer: markov_groove.sequencer.audio_sequencer.AudioSequencer,
    samples: Dict[float, nptypes._ndarray.NDArray]
)
```

TODO: Find the closest sample in a given Dictionary of samples by using the samples in the sequencer.

Function `read_audio_files`

```
def read_audio_files(
    path: Union[pathlib.Path, str],
    regex: str
) -> List[markov_groove.audio_file.AudioFile]
```

Reads audio files in given folder and returns a list of AudioFile. For all following directories use `**/..`.

Function `read_midi_files`

```
def read_midi_files(
    path: Union[pathlib.Path, str],
    regex: str
) -> List[pretty_midi.pretty_midi.PrettyMIDI]
```

Reads mid files in given folder and returns a list of PrettyMIDI. For all following directories use `**/..`.

Generated by *pdoc* 0.9.1 (<https://pdoc3.github.io>).