fretwork Documentation

Release 0.5.0

FoFiX team

Jan 04, 2021

Contents

This is a shared library between FoFiX and FoF:R.

CHAPTER 1

Misc

1.1 How to release

Here is the release process for `fretwork`:

- bump the version on a PR (`fretwork/version.py` and `source/conf.py`):

```
pip install bumpversion
bumpversion --allow-dirty part
```

- make a tag
- make wheels (see below)
- upload sources and wheels on GitHub in a new release.

About wheels:

> Since `fretwork` uses `cython`, one needs to compile extensions in several plateforms. Then, one need to make a wheel for each platform.

1.1.1 Linux

1. Install dependencies:

- `SDL 1.2`
- `SDL_mixer`
- `libsoundtouch`
- `libvorbisfile`
- `pkg-config`
- `portmidi`

2. Install Python dependencies:

```
pip install -r requirements.txt
```

3. Compile extensions:

```
python2 setup.py build_ext --inplace --force
```

4. make a wheel:

```
python2 setup.py sdist
python2 setup.py bdist_wheel --inplace
```

5. repeat 3 and 4 with `python3`.

1.1.2 Windows (32 bits)

Python 2.7

- Download & install Python 2.7 32 bits (x86)
- Download & install Microsoft Visual C++ Compiler for Python 2.7 (9.0)
- Open a Visual C++ Compiler console (32 bits)

Python 3.6

- Download & install Python 3.6 32 bits (x86)
- Download Microsoft Visual C++ 2015
- Open a Visual C++ 2015 x86 x64 cross build tools command prompt

All

- Go to the fretwork directory
- Copy the Windows dependency pack into `win32`
- Install some python dependencies:

```
pip.exe install cython setuptools wheel
```

- Compile extensions:

```
python.exe setup.py build_ext --inplace --force
```

- Make a wheel:

```
python.exe setup.py sdist
python.exe setup.py bdist_wheel
```

1.2 Release notes

1.2.1 0.5.0 (2021-01-04)

- Build: specify numpy versions according to Python versions
- Build: update classifiers (Python versions 3.7 & 3.8)
- Deps: add a requirements.txt file
- Deps: do not pin requirements in *setup.py*
- Deps: update Cython version
- Deps: update NumPy version
- Deps: update PyOpenGL version
- Deps: update Pygame version
- Doc: fix the Windows dependency pack link in the README file
- Setup: update contact information
- Windows: replace *dprintf* with *fdprintf* (MixStream)

Details: https://github.com/fofix/fretwork/milestone/3?closed=1

1.2.2 0.4.0 (2019-02-10)

- Pin versions
- Make the log level customizable
- Fix *SDL_mixer* init
- Syntax: fix relative imports in the *midi* module
- Python 3: fix an exception syntax in *midi.EventDispatcher*
- Python 3: Use python3 imports for *StringIO* and *StringType*
- Doc: remove the useless *midi* module content
- Doc: add the *mixstream.VorbisFileMixStream* content
- Tests: add some tests for *midi.MidiInFile*
- Tests: add some tests for *midi.DataTypeConverters*
- Tests: add some tests for the *Task* module
- Deps: remove *Pillow* from requirements

Details: https://github.com/fofix/fretwork/milestone/2?closed=1

1.2.3 0.3.0 (2017-11-09)

- use the *logging* module
- port the code to python 3
- add docs

- add tests
- release on PyPi

Details: https://github.com/fofix/fretwork/milestone/1?closed=1

1.2.4 0.2.0 (2015-11-22)

- Add the *midi* module
- add the *time* module
- remove compiled code
- add DLLs path for Windows

1.2.5 0.1.1 (2015-09-21)

Initial release

CHAPTER 2

Code

2.1 fretwork

2.1.1 fretwork package

Subpackages

fretwork.midi package

Submodules

fretwork.midi.DataTypeConverters module

This module contains functions for reading and writing the special data types that a midi file contains.

`fretwork.midi.DataTypeConverters.`**`fromBytes`**(*value*)
Turns a list of bytes into a string

`fretwork.midi.DataTypeConverters.`**`getNibbles`**(*byte*)
Returns hi and lo bits in a byte as a tuple

```
>>> getNibbles(142)
(8, 14)

Asserts byte value in byte range
>>> getNibbles(256)
Traceback (most recent call last):
...
ValueError: Byte value out of range 0-255: 256
```

`fretwork.midi.DataTypeConverters.`**`readBew`**(*value*)
Reads string as big endian word, (asserts len(value) in [1,2,4])

```
>>> readBew('aáâã')
0
>>> readBew('aá')
0
```

fretwork.midi.DataTypeConverters.**readVar**(*value*)
Converts varlength format to integer. Just pass it 0 or more chars that might be a varlen and it will only use the relevant chars. use varLen(readVar(value)) to see how many bytes the integer value takes. asserts len(value) >= 0

```
>>> readVar('@')
1081408
>>> readVar('áâãa')
295821045191137
```

fretwork.midi.DataTypeConverters.**setNibbles**(*hiNibble*, *loNibble*)
Returns byte with value set according to hi and lo bits Asserts hiNibble and loNibble in range(16)

```
>>> setNibbles(8, 14)
142

>>> setNibbles(8, 16)
Traceback (most recent call last):
...
ValueError: Nible value out of range 0-15: (8, 16)
```

fretwork.midi.DataTypeConverters.**toBytes**(*value*)
Turns a string into a list of byte values

fretwork.midi.DataTypeConverters.**to_n_bits**(*value*, *length=1*, *nbits=7*)
returns the integer value as a sequence of nbits bytes

fretwork.midi.DataTypeConverters.**varLen**(*value*)
Returns the the number of bytes an integer will be when converted to varlength

fretwork.midi.DataTypeConverters.**writeBew**(*value*, *length*)
Write int as big endian formatted string, (asserts length in [1,2,4]) Difficult to print the result in doctest, so I do a simple roundabout test.

```
>>> readBew(writeBew(25057, 2))
25057
>>> readBew(writeBew(1642193635L, 4))
1642193635
```

fretwork.midi.DataTypeConverters.**writeVar**(*value*)
Converts an integer to varlength format

fretwork.midi.EventDispatcher module

class fretwork.midi.EventDispatcher.**EventDispatcher**(*outstream*)

channel_messages(*hi_nible*, *channel*, *data*)
Dispatches channel messages

continuous_controllers(*channel*, *controller*, *value*)
Dispatches channel messages

eof()
End of file!

header(*format*, *nTracks*, *division*)
Triggers the header event

meta_event(*meta_type*, *data*)
Dispatches meta events

reset_time()
Updates relative/absolute time.

start_of_track(*current_track*)
Triggers the start of track event

sysex_event(*data*)
Dispatcher for sysex events

system_commons(*common_type*, *common_data*)
Dispatches system common messages

update_time(*new_time=0*, *relative=1*)
Updates relative/absolute time.

fretwork.midi.MidiFileParser module

class fretwork.midi.MidiFileParser.**MidiFileParser**(*raw_in*, *outstream*)
Bases: object

The MidiFileParser is the lowest level parser that see the data as midi data. It generates events that gets triggered on the outstream.

parseMThdChunk()
Parses the header chunk

parseMTrkChunk()
Parses a track chunk. This is the most important part of the parser.

parseMTrkChunks()
Parses all track chunks.

fretwork.midi.MidiInFile module

class fretwork.midi.MidiInFile.**MidiInFile**(*outStream*, *infile*)
Bases: object

Parses a midi file, and triggers the midi events on the outStream object.

Get example data from a minimal midi file, generated with cubase.

```
Do parsing, and generate events with MidiToText,
so we can see what a minimal midi file contains
>>> from MidiToText import MidiToText
>>> midi_in = MidiInFile(MidiToText(), test_file)
>>> midi_in.read()
format: 0, nTracks: 1, division: 480
---------------------------------

Start - track #0
```

(continues on next page)

(continued from previous page)

```
sequence_name: Type 0
tempo: 500000
time_signature: 4 2 24 8
note_on  - ch:00,  note:48,  vel:64 time:0
note_off - ch:00,  note:48,  vel:40 time:480
End of track

End of file
```

read()
Start parsing the file

setData(*data=''*)
Sets the data from a plain string

fretwork.midi.MidiInStream module

class fretwork.midi.MidiInStream.**MidiInStream**(*midiOutStream*, *device*)
Bases: object

Takes midi events from the midi input and calls the apropriate method in the eventhandler object

close()
Stop the MidiInstream

read(*time=0*)
Start the MidiInstream.

Parameters **time** – sets timer to specific start value.

resetTimer(*time=0*)
Resets the timer, probably a good idea if there is some kind of looping going on

fretwork.midi.MidiOutFile module

class fretwork.midi.MidiOutFile.**MidiOutFile**(*raw_out=''*)
Bases: *fretwork.midi.MidiOutStream.MidiOutStream*

MidiOutFile is an eventhandler that subclasses MidiOutStream.

aftertouch(*channel=0*, *note=64*, *velocity=64*)

Parameters

- **channel** – 0-15
- **note** – 0-127
- **velocity** – 0-127

channel_pressure(*channel*, *pressure*)

Parameters

- **channel** – 0-15
- **pressure** – 0-127

continuous_controller(*channel*, *controller*, *value*)

Parameters

- **channel** – 0-15
- **controller** – 0-127
- **value** – 0-127

copyright (*text*)
Copyright notice

Parameters **text** – string

cuepoint (*text*)

Parameters **text** – string

end_of_track ()
Writes the track to the buffer.

eof ()
End of file. No more events to be processed.

event_slice (*slc*)
Writes the slice of an event to the current track. Correctly inserting a varlen timestamp too.

header (*format=0*, *nTracks=1*, *division=96*)

Parameters

- **format** – type of midi file in [0,1,2]
- **nTracks** – number of tracks. 1 track for type 0 file
- **division** – timing division ie. 96 ppq.

instrument_name (*text*)

Parameters **text** – string

key_signature (*sf*, *mi*)

Parameters

- **sf** – a byte specifying the number of flats (-ve) or sharps (+ve) that identifies the key signature (-7 = 7 flats, -1 = 1 flat, 0 = key of C, 1 = 1 sharp, etc).
- **mi** – a byte specifying a major (0) or minor (1) key.

lyric (*text*)

Parameters **text** – string

marker (*text*)

Parameters **text** – string

meta_event (*meta_type*, *data*)
Handles any undefined meta events

meta_slice (*meta_type*, *data_slice*)
Writes a meta event

midi_ch_prefix (*channel*)

Parameters **channel** – midi channel for subsequent data (deprecated in the spec)

midi_port (*value*)

Parameters **value** – Midi port (deprecated in the spec)

midi_time_code(*msg_type*, *values*)

Parameters

- **msg_type** – 0-7
- **values** – 0-15

note_off(*channel=0*, *note=64*, *velocity=64*)

Parameters

- **channel** – 0-15
- **note** – 0-127
- **velocity** – 0-127

note_on(*channel=0*, *note=64*, *velocity=64*)

Parameters

- **channel** – 0-15
- **note** – 0-127
- **velocity** – 0-127

patch_change(*channel*, *patch*)

Parameters

- **channel** – 0-15
- **patch** – 0-127

pitch_bend(*channel*, *value*)

Parameters

- **channel** – 0-15
- **value** – 0-16383

sequence_name(*text*)
Sequence/track name

Parameters **text** – string

sequence_number(*value*)

Parameters **value** – 0-65535

sequencer_specific(*data*)

Parameters **data** – The data as byte values

smtp_offset(*hour*, *minute*, *second*, *frame*, *framePart*)

Parameters

- **hour** – a byte specifying the hour (0-23). Should be encoded with the SMPTE format, just as it is in MIDI Time Code.
- **minute** – a byte specifying the minute (0-59)
- **second** – a byte specifying the second (0-59)
- **frame** – a byte specifying the number of frames per second (one of : 24, 25, 29, 30).

- **framePart** – a byte specifying the number of fractional frames, in 100ths of a frame (even in SMPTE-based tracks using a different frame subdivision, defined in the MThd chunk).

song_position_pointer(*value*)

Parameters **value** – 0-16383

song_select(*songNumber*)

Parameters **songNumber** – 0-127

start_of_track(*n_track=0*)

Parameters **n_track** – number of track

system_exclusive(*data*)

Parameters **data** – list of values in range(128)

tempo(*value*)
tempo in us/quarternote (to calculate value from bpm: int(60,000,000.00 / BPM))

Parameters **value** – 0-2097151

text(*text*)
Text event

Parameters **text** – string

time_signature(*nn*, *dd*, *cc*, *bb*)

Parameters

- **nn** – Numerator of the signature as notated on sheet music
- **dd** – Denominator of the signature as notated on sheet music The denominator is a negative power of 2: 2 = quarter note, 3 = eighth, etc.
- **cc** – The number of MIDI clocks in a metronome click
- **bb** – The number of notated 32nd notes in a MIDI quarter note (24 MIDI clocks)

tuning_request()

write()

fretwork.midi.MidiOutStream module

class fretwork.midi.MidiOutStream.**MidiOutStream**
Bases: object

MidiOutstream is Basically an eventhandler. It is the most central class in the Midi library. You use it both for writing events to an output stream, and as an event handler for an input stream.

This makes it extremely easy to take input from one stream and send it to another. Ie. if you want to read a Midi file, do some processing, and send it to a midiport.

All time values are in absolute values from the opening of a stream. To calculate time values, please use the MidiTime and MidiDeltaTime classes.

abs_time()
Returns the absolute time

active_sensing()

aftertouch (*channel=0*, *note=64*, *velocity=64*)

Parameters

- **channel** – 0-15
- **note** – 0-127
- **velocity** – 0-127

channel_message (*message_type*, *channel*, *data*)
The default event handler for channel messages

channel_pressure (*channel*, *pressure*)

Parameters

- **channel** – 0-15
- **pressure** – 0-127

continuous_controller (*channel*, *controller*, *value*)

Parameters

- **channel** – 0-15
- **controller** – 0-127
- **value** – 0-127

copyright (*text*)
Copyright notice

Parameters **text** – string

cuepoint (*text*)

Parameters **text** – string

end_of_track ()

Parameters **n_track** – number of track

eof ()
End of file. No more events to be processed.

get_current_track ()
Returns the current track number

get_run_stat ()
Set the new running status

header (*format=0*, *nTracks=1*, *division=96*)

Parameters

- **format** – type of midi file in [1,2]
- **nTracks** – number of tracks
- **division** – timing division

instrument_name (*text*)

Parameters **text** – string

key_signature (*sf*, *mi*)

Parameters

- **sf** – is a byte specifying the number of flats (-ve) or sharps (+ve) that identifies the key signature (-7 = 7 flats, -1 = 1 flat, 0 = key of C, 1 = 1 sharp, etc).
- **mi** – is a byte specifying a major (0) or minor (1) key.

lyric(*text*)

Parameters **text** – string

marker(*text*)

Parameters **text** – string

meta_event(*meta_type*, *data*)
Handles any undefined meta events

midi_ch_prefix(*channel*)

Parameters **channel** – midi channel for subsequent data (deprecated in the spec)

midi_port(*value*)

Parameters **value** – Midi port (deprecated in the spec)

midi_time_code(*msg_type*, *values*)

Parameters

- **msg_type** – 0-7
- **values** – 0-15

note_off(*channel=0*, *note=64*, *velocity=64*)

Parameters

- **channel** – 0-15
- **note** – 0-127
- **velocity** – 0-127

note_on(*channel=0*, *note=64*, *velocity=64*)

Parameters

- **channel** – 0-15
- **note** – 0-127
- **velocity** – 0-127

patch_change(*channel*, *patch*)

Parameters

- **channel** – 0-15
- **patch** – 0-127

pitch_bend(*channel*, *value*)

Parameters

- **channel** – 0-15
- **value** – 0-16383

rel_time()
Returns the relative time

reset_run_stat()
Invalidates the running status

reset_time()
reset time to 0

sequence_name(*text*)
Sequence/track name

Parameters **text** – string

sequence_number(*value*)

Parameters **value** – 0-16383

sequencer_specific(*data*)

Parameters **data** – The data as byte values

set_current_track(*new_track*)
Sets the current track number

set_run_stat(*new_status*)
Set the new running status

smtp_offset(*hour*, *minute*, *second*, *frame*, *framePart*)

Parameters

- **hour** – a byte specifying the hour (0-23). Should be encoded with the SMPTE format, just as it is in MIDI Time Code.
- **minute** – a byte specifying the minute (0-59)
- **second** – a byte specifying the second (0-59)
- **frame** – A byte specifying the number of frames per second (one of : 24, 25, 29, 30).
- **framePart** – A byte specifying the number of fractional frames, in 100ths of a frame (even in SMPTE-based tracks using a different frame subdivision, defined in the MThd chunk).

song_continue()

song_position_pointer(*value*)

Parameters **value** – 0-16383

song_select(*songNumber*)

Parameters **songNumber** – 0-127

song_start()

song_stop()

start_of_track(*n_track=0*)

Parameters **n_track** – number of track

system_exclusive(*data*)

Parameters **data** – list of values in range(128)

system_reset()

tempo(*value*)
Tempo in us/quarternote (to calculate value from bpm: int(60,000,000.00 / BPM))

Parameters **value** – 0-2097151

text (*text*)
Text event

Parameters **text** – string

time_signature (*nn*, *dd*, *cc*, *bb*)

Parameters

- **nn** – Numerator of the signature as notated on sheet music
- **dd** – Denominator of the signature as notated on sheet music The denominator is a negative power of 2: 2 = quarter note, 3 = eighth, etc.
- **cc** – The number of MIDI clocks in a metronome click
- **bb** – The number of notated 32nd notes in a MIDI quarter note (24 MIDI clocks)

timing_clock ()

tuning_request ()

update_time (*new_time=0*, *relative=1*)
Updates the time, if relative is true, new_time is relative, else it's absolute.

fretwork.midi.MidiToText module

class fretwork.midi.MidiToText.**MidiToText**
Bases: *fretwork.midi.MidiOutStream.MidiOutStream*

This class renders a midi file as text. It is mostly used for debugging

aftertouch (*channel=0*, *note=64*, *velocity=64*)

Parameters

- **channel** – 0-15
- **note** – 0-127
- **velocity** – 0-127

channel_message (*message_type*, *channel*, *data*)
The default event handler for channel messages

channel_pressure (*channel*, *pressure*)

Parameters

- **channel** – 0-15
- **pressure** – 0-127

continuous_controller (*channel*, *controller*, *value*)

Parameters

- **channel** – 0-15
- **controller** – 0-127
- **value** – 0-127

copyright (*text*)
Copyright notice

Parameters **text** – string

cuepoint(*text*)

Parameters **text** – string

end_of_track()

Parameters **n_track** – number of track

eof()
End of file. No more events to be processed.

header(*format=0*, *nTracks=1*, *division=96*)

Parameters

- **format** – type of midi file in [1,2]
- **nTracks** – number of tracks
- **division** – timing division

instrument_name(*text*)

Parameters **text** – string

key_signature(*sf*, *mi*)

Parameters

- **sf** – is a byte specifying the number of flats (-ve) or sharps (+ve) that identifies the key signature (-7 = 7 flats, -1 = 1 flat, 0 = key of C, 1 = 1 sharp, etc).
- **mi** – is a byte specifying a major (0) or minor (1) key.

lyric(*text*)

Parameters **text** – string

marker(*text*)

Parameters **text** – string

meta_event(*meta_type*, *data*)
Handles any undefined meta events

midi_ch_prefix(*channel*)

Parameters **channel** – midi channel for subsequent data (deprecated in the

spec)

midi_port(*value*)

Parameters **value** – Midi port (deprecated in the spec)

midi_time_code(*msg_type*, *values*)

Parameters

- **msg_type** – 0-7
- **values** – 0-15

note_off(*channel=0*, *note=64*, *velocity=64*)

Parameters

- **channel** – 0-15

- **note** – 0-127
- **velocity** – 0-127

note_on(*channel=0*, *note=64*, *velocity=64*)

Parameters

- **channel** – 0-15
- **note** – 0-127
- **velocity** – 0-127

patch_change(*channel*, *patch*)

Parameters

- **channel** – 0-15
- **patch** – 0-127

pitch_bend(*channel*, *value*)

Parameters

- **channel** – 0-15
- **value** – 0-16383

sequence_name(*text*)
Sequence/track name

Parameters **text** – string

sequence_number(*value*)

Parameters **value** – 0-16383

sequencer_specific(*data*)

Parameters **data** – The data as byte values

smtp_offset(*hour*, *minute*, *second*, *frame*, *framePart*)

Parameters

- **hour** – a byte specifying the hour (0-23). Should be encoded with the SMPTE format, just as it is in MIDI Time Code.
- **minute** – a byte specifying the minute (0-59)
- **second** – a byte specifying the second (0-59)
- **frame** – A byte specifying the number of frames per second (one of : 24, 25, 29, 30).
- **framePart** – A byte specifying the number of fractional frames, in 100ths of a frame (even in SMPTE-based tracks using a different frame subdivision, defined in the MThd chunk).

song_position_pointer(*value*)

Parameters **value** – 0-16383

song_select(*songNumber*)

Parameters **songNumber** – 0-127

start_of_track(*n_track=0*)

Parameters **n_track** – number of track

sysex_event(*data*)

system_exclusive(*data*)

Parameters **data** – list of values in range(128)

tempo(*value*)
Tempo in us/quarternote (to calculate value from bpm: int(60,000,000.00 / BPM))

Parameters **value** – 0-2097151

text(*text*)
Text event

Parameters **text** – string

time_signature(*nn*, *dd*, *cc*, *bb*)

Parameters

- **nn** – Numerator of the signature as notated on sheet music
- **dd** – Denominator of the signature as notated on sheet music The denominator is a negative power of 2: 2 = quarter note, 3 = eighth, etc.
- **cc** – The number of MIDI clocks in a metronome click
- **bb** – The number of notated 32nd notes in a MIDI quarter note (24 MIDI clocks)

tuning_request()

fretwork.midi.RawInstreamFile module

class fretwork.midi.RawInstreamFile.**RawInstreamFile**(*infile=''*)
Bases: object

It parses and reads data from an input file. It takes care of big endianess, and keeps track of the cursor position. The midi parser only reads from this object. Never directly from the file.

getCursor()
Returns the value of the cursor

moveCursor(*relative_position=0*)
Moves the cursor to a new relative position

nextSlice(*length*, *move_cursor=1*)
Reads the next text slice from the raw data, with length

readBew(*n_bytes=1*, *move_cursor=1*)
Reads n bytes of date from the current cursor position. Moves cursor if move_cursor is true

readVarLen()
Reads a variable length value from the current cursor position. Moves cursor if move_cursor is true

setCursor(*position=0*)
Sets the absolute position if the cursor

setData(*data=''*)
Sets the data from a string.

fretwork.midi.RawOutstreamFile module

class fretwork.midi.RawOutstreamFile.**RawOutstreamFile**(*outfile=''*)
Bases: object

Writes a midi file to disk.

getvalue()

write()
Writes to disc

writeBew(*value*, *length=1*)
Writes a value to the file as big endian word

writeSlice(*str_slice*)
Writes the next text slice to the raw data

writeVarLen(*value*)
Writes a variable length word to the file

fretwork.midi.constants module

A collection of constants from the midi spec.

fretwork.midi.constants.**is_status**(*byte*)

fretwork.mixstream package

Module contents

Submodules

fretwork.audio module

fretwork.log module

Logging module

Usage

Configure the logger in your launcher:

```
from fretwork import log

# configure the logger
log.configure('file.log', logging.DEBUG)
```

Import and use it:

```
import logging

logger = logging.getLogger(__name__)
```

fretwork.log.**configure**(*log_filename*, *log_level=20*)
Configure logging.

Parameters

- **log_filename** – name of the file where logs are saved.
- **log_level** – level of logs (default: logging.INFO).

fretwork.task module ++++++++++++++++++++-

class fretwork.task.**Task**
Bases: object

run(*ticks*)

started()

stopped()

class fretwork.task.**TaskEngine**(*engine*)
Bases: object

addTask(*task*, *synced=True*)
Add a task

checkTask(*task*)
Check if a task exists

exit()
Remove all tasks.

pauseTask(*task*)
Pause a task

removeTask(*task*)
Remove a task

resumeTask(*task*)
Resume a paused task

run()
Run one cycle of the task scheduler engine.

runTask(*task*, *tick=0*)

fretwork.timer module

class fretwork.timer.**FpsTimer**
Bases: *fretwork.timer.Timer*

delay(*fps*)
Reimplementation of pygame.time.Clock.tick() delay functionality. Needed for fps limiting.

get_fps()
Calculates and return the average fps then resets the counter.

tick()
Calculates time delta since last call. Also accumulates the delta and increments frame counter.

class fretwork.timer.**Timer**
Bases: object

delta_time()
Return time delta since startTime

tick()
Returns the delta between the current and previous ticks

time()
Get current time in milliseconds

fretwork.unicode module

Miscellaneous functions for helping us handle Unicode correctly in the face of what we've done in the past.

fretwork.unicode.**unicodify**(*s*)
Turns *s* into a Unicode string

Parameters **s** – input string

Returns Unicode version of s

fretwork.unicode.**utf8**(*s*)
Turns s into a valid UTF-8 bytestring. @param s: input @return: UTF-8-encoded version of s

fretwork.version module

Python Module Index

f

Index

F

G

H

I

K

L

M

N

P

R

S

T

U

V

W