MIDIUtil Documentation

Release 1.1.1

Mark Conway Wirt

Mar 04, 2018

Contents

1 Creating the MIDIFile Object 1

2 Common Events and Function 5

3 Tuning and Microtonalities 9

4 Extending the Library 13

5 Class Reference 17

6 Introduction 25

7 Installation 27

8 Quick Start 29

9 Thank You 31

10 Indices and tables 33

CHAPTER 1

Creating the MIDIFile Object

The first step in using the library is creating a `MIDIFile` object. There are only a few parameters that need be specified, but they affect the functioning of the library, so it's good to understand what they do.

The signature of of the `MIDIFile __init__()` function is as follows:

```
def __init__(self,
          numTracks=1,
          removeDuplicates=True,
          deinterleave=True,
          adjust_origin=False,
          file_format=1,
          ticks_per_quarternote=TICKSPERQUARTERNOTE,
          eventtime_is_ticks=False):
```

where the parameters do the following:

1.1 numTracks

`numTracks` specifies the number of tracks the MIDI file should have. It should be set to at least 1 (after all, a MIDI file without tracks isn't very useful), but it may be set higher for a multi-track file.

This parameter defaults to `1`.

1.2 removeDuplicates

If set to `True` (the default), duplicate notes will be removed from the file. This is done on a track-by-track basis.

Notes are considered duplicates if they occur at the same time, and have equivalent pitch, and MIDI channel. If set to `False` no attempt is made to remove notes which appear to be duplicates.

`removeDuplicates()` also attempts to remove other kinds of duplicates. For example, if there are two tempo events at the same time and same tempo, they are considered duplicates.

Of course, it's best not to insert duplicate events in the first place, but this could be unavoidable in some instances – for example, if the software is used in the creation of Generative Music using an algorithm that can create duplication of events.

1.3 deinterleave

If `deinterleave` is set to `True` (the default), an attempt will be made to remove interleaved notes.

To understand what an *interleaved* note is, it is useful to have some understanding of the MIDI standard.

To make this library more human-centric, one of the fundamental concepts used is that of the **note**. But the MIDI standard doesn't have notes; instead, it has **note on** and **note off** events. These are correlated by channel and pitch.

So if, for example, you create two notes of duration 1 and separated by 1/2 of a beat, ie:

```
time = 0
duration = 1
MyMIDI.addNote(track,channel,pitch,time,duration,volume)
time = 0.5
MyMIDI.addNote(track,channel,pitch,time,duration,volume)
```

you end up with a note on event at 0, another note on event a 0.5, and two note off events, one at 1.0 and one at 1.5. So when the first note off event is processed it raises the question: which note on event does it correspond to? The channel and pitch are the same, so there is some ambiguity in the way that a hardware or software instrument will respond.

if `deinterleave` is `True` the library tries to disambiguate the situation by shifting the first note's off event to be immediately before the second note's on event. Thus in the example above the first note on would be at 0, the first note off would be at 0.5, the second note on would also be at 0.5 (but would be processed after the note off at that time), and the last note off would be at 1.5.

If this parameter is set to `False` no events will be shifted.

1.4 adjust_origin

If `adjust_origin` is `True` the library will find the earliest event in all the tracks and shift all events so that that time is t=0. If it is `False` no time-shifting will occur. The defaul value is `False`.

1.5 file_format

This specifies the format of the file to be written. Both format 1 (the default) and format 2 files are supported.

In the format 1 file there is a separate "tempo" track to which tempo and time signature events are written. The calls to create these events – `addTemo()` and `addTimeSignature()` accept a track parameter, but in a format 1 file these are ignored. In format 2 files they are interpreted literally (and zero-origined, so that a two track file has indices `0` and `1`).

Track indexing is always zero-based, but with the format 1 file the tempo track is not indexed. Thus if you create a one track file:

```
MyMIDI = MIDIFile(1, file_format=1)
```

you would only have `0` as a valid index; the tempo track is managed independently for you. Thus:

```
track = 0
big_track = 1000
MyMIDI.addTempo(big_track, 0, 120)
MyMIDI.addNote(track, 0, 69, 0, 1, 100)
```

works, even though "track 0" is really the second track in the file, and there is no track 1000.

1.6 ticks_per_quarternote

The MIDI ticks per quarter note. Defaults to 960. This defines the finest level of time resolution available in the file. 120, 240, 384, 480, and 960 are common values.

1.7 eventtime_is_ticks

If set to `True`, all times passed into the event creation functions should be specified in ticks. Otherwise they should be specified in quarter-notes (the default).

CHAPTER 2

Common Events and Function

This page lists some of the more common things that a user is likely to do with the MIDI file. It is not exhaustive; see the class reference for a more complete list of public functions.

2.1 Adding Notes

As the MIDI standard is all about music, creating notes will probably be the lion's share of what you're doing. This is done with the `addNote()` function.

`MIDIFile.`**`addNote`**(*track*, *channel*, *pitch*, *time*, *duration*, *volume*, *annotation=None*)

Add notes to the MIDIFile object

Parameters

- **`track`** – The track to which the note is added.
- **`channel`** – the MIDI channel to assign to the note. [Integer, 0-15]
- **`pitch`** – the MIDI pitch number [Integer, 0-127].
- **`time`** – the time at which the note sounds. The value can be either quarter notes [Float], or ticks [Integer]. Ticks may be specified by passing eventtime_is_ticks=True to the MIDIFile constructor. The default is quarter notes.
- **`duration`** – the duration of the note. Like the time argument, the value can be either quarter notes [Float], or ticks [Integer].
- **`volume`** – the volume (velocity) of the note. [Integer, 0-127].
- **`annotation`** – Arbitrary data to attach to the note.

The `annotation` parameter attaches arbitrary data to the note. This is not used in the code, but can be useful anyway. As an example, I have created a project that uses MIDIFile to write csound orchestra files directly from the class `EventList`.

As an example, the following code-fragment adds two notes to an (already created) MIDIFile object:

```
track    = 0   # Track numbers are zero-origined
channel  = 0   # MIDI channel number
pitch    = 60  # MIDI note number
time     = 0   # In beats
duration = 1   # In beats
volume   = 100 # 0-127, 127 being full volume

MyMIDI.addNote(track,channel,pitch,time,duration,volume)
time  = 1
pitch = 61
MyMIDI.addNote(track,channel,pitch,time,duration,volume)
```

2.2 Add a Tempo

Every track can have tempos specified (the unit of which is beats per minute).

`MIDIFile.`**`addTempo`**(*track*, *time*, *tempo*)

Add notes to the MIDIFile object

Parameters

- **`track`** – The track to which the tempo event is added. Note that in a format 1 file this parameter is ignored and the tempo is written to the tempo track
- **`time`** – The time (in beats) at which tempo event is placed
- **`tempo`** – The tempo, in Beats per Minute. [Integer]

Example:

```
track = 0
time  = 0   # beats, beginning of track
tempo = 120 # BPM
MyMIDI.addTempo(track, time, tempo)
```

2.3 Assign a Name to a Track

`MIDIFile.`**`addTrackName`**(*track*, *time*, *trackName*)

Name a track.

Parameters

- **`track`** – The track to which the name is assigned.
- **`time`** – The time (in beats) at which the track name event is placed. In general this should probably be time 0 (the beginning of the track).
- **`trackName`** – The name to assign to the track [String]

In general, the time should probably be t=0

Example:

```
track      = 0
time       = 0
track_name = "Bassline 1"
MyMIDI.addTrackName(track, time, track_name)
```

2.4 Adding a Program Change Event

The program change event tells the the instrument what voice a certain track should sound. As an example, if the instrument you're using supports General MIDI, you can use the GM numbers to specify the instrument.

Important Note: Within this library program numbers are zero-origined (as they are on a byte-level within the MIDI standard), but most of the documentation you will see is musician-centric, so they are usually given as one-origined. So, for example, if you want to sound a Cello, you would use a program number of 42, not the 43 which is given in the link above.

`MIDIFile.`**`addProgramChange`**(*tracknum*, *channel*, *time*, *program*)

Add a MIDI program change event.

Parameters

- **`tracknum`** – The zero-based track number to which program change event is added.
- **`channel`** – the MIDI channel to assign to the event. [Integer, 0-15]
- **`time`** – The time (in beats) at which the program change event is placed [Float].
- **`program`** – the program number. [Integer, 0-127].

Example:

```
track   = 0
channel = 0
time    = 8 # Eight beats into the composition
program = 42 # A Cello

MyMIDI.addProgramChange(track, channel, time, program)
```

2.5 Writing the File to Disk

Ultimately, you'll need to write your data to disk to use it.

`MIDIFile.`**`writeFile`**(*fileHandle*)

Write the MIDI File.

Parameters **`fileHandle`** – A file handle that has been opened for binary writing.

Example:

```
with open("mymidifile.midi", 'wb') as output_file:
    MyMIDI.writeFile(output_file)
```

2.6 Additional Public Function

The above list is not exhaustive. For example, the library includes methods to create arbitrary channel control events, SysEx and Universal SysEx events, Registered Parameter calls and Non-Registered Parameter calls, etc. Please see the *Class Reference* for a more complete list of public functions.

CHAPTER 3

Tuning and Microtonalities

One of my interests is microtonalities/non-standard tunings, so support for such explorations has been included in the library.

There are several ways that tuning data can be specified in the MIDI standard, two of the most common being note pitch-bend and bulk tuning dumps. In this library I have implemented the real-time change note tuning of the MIDI tuning standard. I chose that as a first implementation because most of the soft-synthesizers I use support this standard.

Note, however, that implementation of the MIDI tuning standard is somewhat spotty, so you may want to verify that your hardware and/or software supports it before you spend too much time.

Recently the pitch bend message had been implemented. The advantage of the pitch bend is that almost all software and hardware understand it. The disadvantage is that different software and hardware can interpret the values differently.

The main function to support a tuning change is `changeNoteTuning`.

`MIDIFile.`**`changeNoteTuning`**(*track*, *tunings*, *sysExChannel=127*, *realTime=True*, *tuningProgam=0*)

Add a real-time MIDI tuning standard update to a track.

Parameters

- **`track`** – The track to which the tuning is applied.
- **`tunings`** – A list to tuples representing the tuning. See below for an explanation.
- **`sysExChannel`** – The SysEx channel of the event. This is mapped to "manufacturer ID" in the event which is written. Unless there is a specific reason for changing it, it should be left at its default value.
- **`realTime`** – Speicifes if the Universal SysEx event should be flagged as real-time or non-real-time. As with the `sysExChannel` argument, this should in general be left at it's default value.
- **`tuningProgram`** – The tuning program number.

This function specifically implements the "real time single note tuning change" (although the name is misleading, as multiple notes can be included in each event). It should be noted that not all hardware or software implements the MIDI tuning standard, and that which does often does not implement it in its entirety.

The `tunings` argument is a list of tuples, in (*note number*, *frequency*) format. As an example, if one wanted to change the frequency on MIDI note 69 to 500 (it is normally 440 Hz), one could do it thus:

```
from midiutil.MidiFile import MIDIFile
MyMIDI = MIDIFile(1)
tuning = [(69, 500)]
MyMIDI.changeNoteTuning(0, tuning, tuningProgam=0)
```

3.1 Tuning Program

With some instruments, such as timidity, this is all you need to do: timidity will apply the tuning change to the notes. Other instruments, such as fluidsynth, require that the tuning program be explicitly assigned. This is done with the `changeTuningProgram` function:

`MIDIFile.`**`changeTuningProgram`**(*track*, *channel*, *time*, *program*, *time_order=False*)

Change the tuning program for a selected track

Parameters

- **`track`** – The track to which the data should be written
- **`channel`** – The channel for the event
- **`time`** – The time of the event
- **`program`** – The tuning program number (0-127)
- **`time_order`** – Preserve the ordering of the component events by ordering in time. See `makeRPNCall()` for a discussion of when this may be necessary

Note that this is a convenience function, as the same functionality is available from directly sequencing controller events.

The specified tuning should already have been written to the stream with `changeNoteTuning`.

3.2 Tuning Bank

The tuning bank can also be specified (fluidsynth assumes that any tuning you transmit via `changeNoteTuning` is assigned to bank zero):

`MIDIFile.`**`changeTuningBank`**(*track*, *channel*, *time*, *bank*, *time_order=False*)

Change the tuning bank for a selected track

Parameters

- **`track`** – The track to which the data should be written
- **`channel`** – The channel for the event
- **`time`** – The time of the event
- **`bank`** – The tuning bank (0-127)
- **`time_order`** – Preserve the ordering of the component events by ordering in time. See `makeRPNCall()` for a discussion of when this may be necessary

Note that this is a convenience function, as the same functionality is available from directly sequencing controller events.

The specified tuning should already have been written to the stream with `changeNoteTuning`.

3.3 An Example

So, as a complete example, the following code fragment would get rid of that pesky 440 Hz A and tell the instrument to use the tuning that you just transmitted:

```
track   = 0
channel = 0
tuning  = [(69, 500)]
program = 0
bank    = 0
time    = 0
MyMIDI.changeNoteTuning(track, tuning, tuningProgam=program)
MyMIDI.changeTuningBank(track, channel, time, bank) # may or may not be needed
MyMIDI.changeTuningProgram(track, channel, time, program) # ditto
```

3.4 Using Pitch Bend

Pitch bend is a *channel level event*, meaning that if you pass a pitch wheel event, all notes on that channel will be affected.

`MIDIFile.`**`addPitchWheelEvent`**(*track*, *channel*, *time*, *pitchWheelValue*)

Add a channel pitch wheel event

Parameters

- **`track`** – The track to which the event is added.
- **`channel`** – the MIDI channel to assign to the event. [Integer, 0-15]
- **`time`** – The time (in beats) at which the event is placed [Float].
- **`pitchWheelValue`** – 0 for no pitch change. [Integer, -8192-8192]

3.5 To Do

- Implement the tuning change with bank select event type.

CHAPTER 4

Extending the Library

The choice of MIDI event types included in the library is somewhat idiosyncratic; I included the events I needed for another software project I was writting. You may find that you need additional events in your work. For this reason I am including some instructions on extending the library. The process isn't too hard (provided you have a working knowledge of Python and the MIDI standard), so the task shouldn't present a competent coder too much difficulty. Alternately (if, for example, you *don't* have a working knowledge of MIDI and don't desire to gain it), you can submit new feature requests to me, and I will include them into the development branch of the code, subject to the constraints of time.

To illustrate the process I show below how the MIDI tempo event is incorporated into the code. This is a relatively simple event, so while it may not illustrate some of the subtleties of MIDI programing, it provides a good, illustrative case.

The MID standard defines the Tempo event as the following byte-stream:

```
FF 51 03 tttttt
```

where `FF 51` is the code and sub-code of the event, `03` is the data length of the event, and `tttttt` are the three bytes of data, encoded as microseconds per quarter note.

4.1 Create a New Event Type

The majority of work involved with creating a new event type is the creation of a new subclass of the `GenericEvent` object of the MIDIFile module. This subclass:

- Initializes any specific instance data that is needed for the MIDI event, and
- Defines how the data are serialized to the byte stream
- Defines the order in which an event is written to the byte stream (see below)

In the case of the tempo, the actual data conversion is easy: people tend to specify tempos in beats per minute, so the input will need to be divided into 60000000.

The serialization strategy is defined in the subclass' `serialize` member function, which is presented below.

`sec_sort_order` and `insertion_order` are used to order the events in the MIDI stream. Events are first ordered in time. Events at the same time are then ordered by `sec_sort_order`, with lower numbers appearing in the stream first. Lastly events are sorted on the `self.insertion_order` member. This strategy makes it possible to, say, create a Registered Parameter Number call from a collection of Control Change events. Since all the CC events will have the same time and class (and therefore default secondary sort order), you can control the order of the events by the order in which you add them to the MIDIFile.

Al of this will perhaps be more clear if we examine the code:

```
class Tempo(GenericEvent):
    '''
    A class that encapsulates a tempo meta-event
    '''
    evtname = 'Tempo'
    sec_sort_order = 3

    def __init__(self, tick, tempo, insertion_order=0):
        self.tempo = int(60000000 / tempo)
        super(Tempo, self).__init__(tick, insertion_order)

    def __eq__(self, other):
        return (self.evtname == other.evtname and
                self.tick == other.tick and
                self.tempo == other.tempo)

    __hash__ = GenericEvent.__hash__

    def serialize(self, previous_event_tick):
        """Return a bytestring representation of the event, in the format required for
        writing into a standard midi file.
        """

        midibytes = b""
        code = 0xFF
        subcode = 0x51
        fourbite = struct.pack('>L', self.tempo)  # big-endian uint32
        threebite = fourbite[1:4]  # Just discard the MSB
        varTime = writeVarLength(self.tick - previous_event_tick)
        for timeByte in varTime:
            midibytes += struct.pack('>B', timeByte)
        midibytes += struct.pack('>B', code)
        midibytes += struct.pack('>B', subcode)
        midibytes += struct.pack('>B', 0x03)  # length in bytes of 24-bit tempo
        midibytes += threebite
        return midibytes
```

The event name (`evtname`) and secondary sort order are defined in class data; any class that you create will do the same. `tick` is the time in MIDI ticks of the event and insertion order will be set in the code. All events should accept these parameters. `tempo` is the specific instance data needed for this event type.

The `__init__()` member converts the tempo to number needed by the standard and then calls the super-class' initialization function with tick and insertion order. All event subclasses should do this.

Next, the `__eq__()` function is defined that specifies when two events are the same. In this case they are the same if the tick, event name, and tempo are the same. This code is used to identify and remove duplicate events. `__hash__()` should explicitly be brought down from the parent class, in in Python 3 it is not implicitly inherited.

Lastly, the `serialize` member function should be created. This will return a byte stream representing the MIDI data. A few things to note about this:

- All MIDI events begin with a time, which is written in an idiosyncratic variable-length format. Use the `writeVarLength` utility function to calculate this.
- Note that in the case of the tempo event, the standard only uses three bytes, whereas in python a long will be packed into four bytes. Hence we just discard the MSB.
- In the temo the actual data is packed: - The time - The code (0xFF) - The subcode (0x51) - The length of that (defined in the event as 3 bytes) - The data proper

4.2 Create an Accessor Function

Next, an accessor function should be added to MIDITrack to create an event of this type. Continuing the example of the tempo event:

```
def addTempo(self, tick, tempo, insertion_order=0):
    '''
    Add a tempo change (or set) event.
    '''
    self.eventList.append(Tempo(tick, tempo,
                          insertion_order=insertion_order))
```

(Most/many MIDI events require a channel specification, but the tempo event does not.)

This is more-or-less boilerplate code, and just needs to appropriately create the object you defined above.

Note that this function can in some cases create multiple events. For example, when one adds a note, both a `NoneOn` and a `NoteOff` event will be created.

Lastly, the public accessor function is via the MIDIFile object, and must include the track number to which the event is written. So in `MIDIFile`:

```
def addTempo(self, track, time, tempo):
    """

    Add notes to the MIDIFile object

    :param track: The track to which the tempo event  is added. Note that
        in a format 1 file this parameter is ignored and the tempo is
        written to the tempo track
    :param time: The time (in beats) at which tempo event is placed
    :param tempo: The tempo, in Beats per Minute. [Integer]
    """
    if self.header.numeric_format == 1:
        track = 0
    self.tracks[track].addTempo(self.time_to_ticks(time), tempo,
                                insertion_order=self.event_counter)
    self.event_counter += 1
```

Note that a track has been added (which is zero-origined and needs to be constrained by the number of tracks that the `MIDIFile` was created with), and `insertion_order` is taken from the class `event_counter` data member. This should be followed in each function you add. Also note that the tempo event is handled differently in format 1 files and format 2 files. This function ensures that the tempo event is written to the first track (track 0) for a format 1 file, otherwise it writes it to the track specified. In most of the public functions a check it done on format, and the track is incremented by one for format 1 files so that the event is not written to the tempo track (but preserving the zero-origined convention for all tracks in both formats.)

The only other complexity is that the public functions accept by default a time in quarter-notes, not MIDI ticks. So the public accessor function should pass the time through the `time_to_ticks()` member. If the MIDIFile was

instantiated with `eventtime_is_ticks = True`, this is just an identity fucntion and the public accessor will expect time in ticks. Otherwise it will convert from quarter-notes to ticks (suing the `TICKSPERQUARTERNOTE` instance data)

This is the function you will use in your code to create an event of the desired type.

4.3 Write Some Tests

Yea, it's a hassle, but you know it's the right thing to do!

CHAPTER 5

Class Reference

class midiutil.MidiFile.**MIDIFile**(*numTracks=1*, *removeDuplicates=True*, *deinterleave=True*, *adjust_origin=False*, *file_format=1*, *ticks_per_quarternote=960*, *eventtime_is_ticks=False*)

A class that encapsulates a full, well-formed MIDI file object.

This is a container object that contains a header (`MIDIHeader`), one or more tracks (class:*MIDITrack*), and the data associated with a proper and well-formed MIDI file.

__init__(*numTracks=1*, *removeDuplicates=True*, *deinterleave=True*, *adjust_origin=False*, *file_format=1*, *ticks_per_quarternote=960*, *eventtime_is_ticks=False*)

Initialize the MIDIFile class

Parameters

- **numTracks** – The number of tracks the file contains. Integer, one or greater
- **removeDuplicates** – If set to `True` remove duplicate events before writing to disk
- **deinterleave** – If set to `True` deinterleave the notes in the stream
- **adjust_origin** – If set to `True` shift all the events in the tracks so that the first event takes place at time t=0. Default is `False`
- **file_format** – The format of the multi-track file. This should either be `1` (the default, and the most widely supported format) or `2`.
- **ticks_per_quarternote** – The number of ticks per quarter note is what the Standard MIDI File Format Specification calls "division". Ticks are the integer unit of time in the SMF, and in most if not all MIDI sequencers. Common values are 120, 240, 384, 480, 960. Note that all these numbers are evenly divisible by 2,3,4,6,8,12,16, and 24, except 120 does not have 16 as a divisor.
- **eventtime_is_ticks** – If set True means event time and duration argument values are integer ticks instead of fractional quarter notes.

Note that the default for `adjust_origin` will change in a future release, so one should probably explicitly set it.

In a format 1 file, it would be a rare cirumstance where adjusting the origin of each track to the track's first note makes any sense.

Example:

```
# Create a two-track MIDIFile

from midiutil.MidiFile import MIDIFile
midi_file = MIDIFile(1, adjust_origin=False)
```

In previous versions of this code the file written was format 2 (which can be thought of as a collection of independent tracks) but was identified as format 1. In this version one can specify either format 1 or 2.

In format 1 files there is a separate tempo track which contains tempo and time signature data, but contains no note data. If one creates a single track format 1 file the actual file has two tracks – one for tempo data and one for note data. In the track indexing the tempo track can be ignored. In other words track 0 is the note track (the second track in the file). However, tempo and time signature data will be written to the first, tempo track. This is done to try and preserve as much interoperability with previous versions as possible.

In a format 2 file all tracks are indexed and the track parameter is interpreted literally.

addControllerEvent (*track*, *channel*, *time*, *controller_number*, *parameter*)

Add a channel control event

Parameters

- **track** – The track to which the event is added.
- **channel** – the MIDI channel to assign to the event. [Integer, 0-15]
- **time** – The time (in beats) at which the event is placed [Float].
- **controller_number** – The controller ID of the event.
- **parameter** – The event's parameter, the meaning of which varies by event type.

addCopyright (*track*, *time*, *notice*)

Add a copyright notice to the MIDIFile object

Parameters

- **track** – The track to which the notice is added.
- **time** – The time (in beats) at which notice event is placed. In general this sould be time t=0
- **notice** – The copyright notice [String]

addKeySignature (*track*, *time*, *accidentals*, *accidental_type*, *mode*, *insertion_order=0*)

Add a Key Signature to a track

Parameters

- **track** – The track to which this should be added
- **time** – The time at which the signature should be placed
- **accidentals** – The number of accidentals in the key signature
- **accidental_type** – The type of accidental
- **mode** – The mode of the scale

The easiest way to use this function is to make sure that the symbolic constants for accidental_type and mode are imported. By doing this:

```
from midiutil.MidiFile import *
```

one gets the following constants defined:

- `SHARPS`
- `FLATS`
- `MAJOR`
- `MINOR`

So, for example, if one wanted to create a key signature for a minor scale with three sharps:

```
MyMIDI.addKeySignature(0, 0, 3, SHARPS, MINOR)
```

addNote (*track*, *channel*, *pitch*, *time*, *duration*, *volume*, *annotation=None*)

Add notes to the MIDIFile object

Parameters

- **track** – The track to which the note is added.
- **channel** – the MIDI channel to assign to the note. [Integer, 0-15]
- **pitch** – the MIDI pitch number [Integer, 0-127].
- **time** – the time at which the note sounds. The value can be either quarter notes [Float], or ticks [Integer]. Ticks may be specified by passing eventtime_is_ticks=True to the MIDIFile constructor. The default is quarter notes.
- **duration** – the duration of the note. Like the time argument, the value can be either quarter notes [Float], or ticks [Integer].
- **volume** – the volume (velocity) of the note. [Integer, 0-127].
- **annotation** – Arbitrary data to attach to the note.

The `annotation` parameter attaches arbitrary data to the note. This is not used in the code, but can be useful anyway. As an example, I have created a project that uses MIDIFile to write csound orchestra files directly from the class `EventList`.

addPitchWheelEvent (*track*, *channel*, *time*, *pitchWheelValue*)

Add a channel pitch wheel event

Parameters

- **track** – The track to which the event is added.
- **channel** – the MIDI channel to assign to the event. [Integer, 0-15]
- **time** – The time (in beats) at which the event is placed [Float].
- **pitchWheelValue** – 0 for no pitch change. [Integer, -8192-8192]

addProgramChange (*tracknum*, *channel*, *time*, *program*)

Add a MIDI program change event.

Parameters

- **tracknum** – The zero-based track number to which program change event is added.
- **channel** – the MIDI channel to assign to the event. [Integer, 0-15]
- **time** – The time (in beats) at which the program change event is placed [Float].
- **program** – the program number. [Integer, 0-127].

addSysEx (*track*, *time*, *manID*, *payload*)
Add a System Exclusive event.

Parameters

- **track** – The track to which the event should be written
- **time** – The time of the event.
- **manID** – The manufacturer ID for the event
- **payload** – The payload for the event. This should be a binary-packed value, and will vary for each type and function.

Note: This is a low-level MIDI function, so care must be used in constructing the payload. It is recommended that higher-level helper functions be written to wrap this function and construct the payload if a developer finds him or herself using the function heavily.

addTempo (*track*, *time*, *tempo*)
Add notes to the MIDIFile object

Parameters

- **track** – The track to which the tempo event is added. Note that in a format 1 file this parameter is ignored and the tempo is written to the tempo track
- **time** – The time (in beats) at which tempo event is placed
- **tempo** – The tempo, in Beats per Minute. [Integer]

addText (*track*, *time*, *text*)
Add a text event

Parameters

- **track** – The track to which the notice is added.
- **time** – The time (in beats) at which text event is placed.
- **text** – The text to adde [ASCII String]

addTimeSignature (*track*, *time*, *numerator*, *denominator*, *clocks_per_tick*, *notes_per_quarter=8*)
Add a time signature event.

Parameters

- **track** – The track to which the signature is assigned. Note that in a format 1 file this parameter is ignored and the event is written to the tempo track
- **time** – The time (in beats) at which the event is placed. In general this should probably be time 0 (the beginning of the track).
- **numerator** – The numerator of the time signature. [Int]
- **denominator** – The denominator of the time signature, expressed as a power of two (see below). [Int]
- **clocks_per_tick** – The number of MIDI clock ticks per metronome click (see below).
- **notes_per_quarter** – The number of annotated 32nd notes in a MIDI quarter note. This is almost always 8 (the default), but some sequencers allow this value to be changed. Unless you know that your sequencing software supports it, this should be left at its default value.

The data format for this event is a little obscure.

The `denominator` should be specified as a power of 2, with a half note being one, a quarter note being two, and eight note being three, etc. Thus, for example, a 4/4 time signature would have a `numerator` of 4 and a `denominator` of 2. A 7/8 time signature would be a `numerator` of 7 and a `denominator` of 3.

The `clocks_per_tick` argument specifies the number of clock ticks per metronome click. By definition there are 24 ticks in a quarter note, so a metronome click per quarter note would be 24. A click every third eighth note would be 3 * 12 = 36.

The `notes_per_quarter` value is also a little confusing. It specifies the number of 32nd notes in a MIDI quarter note. Usually there are 8 32nd notes in a quarter note (8/32 = 1/4), so the default value is 8. However, one can change this value if needed. Setting it to 16, for example, would cause the music to play at double speed, as there would be 16/32 (or what could be considered *two* quarter notes for every one MIDI quarter note.

Note that both the `clocks_per_tick` and the `notes_per_quarter` are specified in terms of quarter notes, even is the score is not a quarter-note based score (i.e., even if the denominator is not `4`). So if you're working with a time signature of, say, 6/8, one still needs to specify the clocks per quarter note.

addTrackName(*track*, *time*, *trackName*)

Name a track.

Parameters

- **track** – The track to which the name is assigned.
- **time** – The time (in beats) at which the track name event is placed. In general this should probably be time 0 (the beginning of the track).
- **trackName** – The name to assign to the track [String]

addUniversalSysEx(*track*, *time*, *code*, *subcode*, *payload*, *sysExChannel=127*, *realTime=False*)

Add a Univeral System Exclusive event.

Parameters

- **track** – The track to which the event should be written
- **time** – The time of the event, in beats.
- **code** – The event code. [Integer]
- **subcode** – The event sub-code [Integer]
- **payload** – The payload for the event. This should be a binary-packed value, and will vary for each type and function.
- **sysExChannel** – The SysEx channel.
- **realTime** – Sets the real-time flag. Defaults to non-real-time.
- **manID** – The manufacturer ID for the event

Note: This is a low-level MIDI function, so care must be used in constructing the payload. It is recommended that higher-level helper functions be written to wrap this function and construct the payload if a developer finds him or herself using the function heavily. As an example of such a helper function, see the `changeNoteTuning()` function, which uses the event to create a real-time note tuning update.

changeNoteTuning(*track*, *tunings*, *sysExChannel=127*, *realTime=True*, *tuningProgam=0*)

Add a real-time MIDI tuning standard update to a track.

Parameters

- **track** – The track to which the tuning is applied.
- **tunings** – A list to tuples representing the tuning. See below for an explanation.
- **sysExChannel** – The SysEx channel of the event. This is mapped to "manufacturer ID" in the event which is written. Unless there is a specific reason for changing it, it should be left at its default value.
- **realTime** – Speicifes if the Universal SysEx event should be flagged as real-time or non-real-time. As with the `sysExChannel` argument, this should in general be left at it's default value.
- **tuningProgram** – The tuning program number.

This function specifically implements the "real time single note tuning change" (although the name is misleading, as multiple notes can be included in each event). It should be noted that not all hardware or software implements the MIDI tuning standard, and that which does often does not implement it in its entirety.

The `tunings` argument is a list of tuples, in (*note number*, *frequency*) format. As an example, if one wanted to change the frequency on MIDI note 69 to 500 (it is normally 440 Hz), one could do it thus:

```
from midiutil.MidiFile import MIDIFile
MyMIDI = MIDIFile(1)
tuning = [(69, 500)]
MyMIDI.changeNoteTuning(0, tuning, tuningProgam=0)
```

changeTuningBank(*track*, *channel*, *time*, *bank*, *time_order=False*)

Change the tuning bank for a selected track

Parameters

- **track** – The track to which the data should be written
- **channel** – The channel for the event
- **time** – The time of the event
- **bank** – The tuning bank (0-127)
- **time_order** – Preserve the ordering of the component events by ordering in time. See `makeRPNCall()` for a discussion of when this may be necessary

Note that this is a convenience function, as the same functionality is available from directly sequencing controller events.

The specified tuning should already have been written to the stream with `changeNoteTuning`.

changeTuningProgram(*track*, *channel*, *time*, *program*, *time_order=False*)

Change the tuning program for a selected track

Parameters

- **track** – The track to which the data should be written
- **channel** – The channel for the event
- **time** – The time of the event
- **program** – The tuning program number (0-127)
- **time_order** – Preserve the ordering of the component events by ordering in time. See `makeRPNCall()` for a discussion of when this may be necessary

Note that this is a convenience function, as the same functionality is available from directly sequencing controller events.

The specified tuning should already have been written to the stream with `changeNoteTuning`.

makeNRPNCall(*track*, *channel*, *time*, *controller_msb*, *controller_lsb*, *data_msb*, *data_lsb*, *time_order=False*)

Perform a Non-Registered Parameter Number Call

Parameters

- **track** – The track to which this applies
- **channel** – The channel to which this applies
- **time** – The time of the event
- **controller_msb** – The Most significant byte of thecontroller. In common usage this will usually be 0
- **controller_lsb** – The least significant byte for the controller message. For example, for a fine-tunning change this would be 01.
- **data_msb** – The most significant byte of the controller's parameter.
- **data_lsb** – The least significant byte of the controller's parameter. If none is needed this should be set to `None`
- **time_order** – Order the control events in time (see below)

The `time_order` parameter is something of a work-around for sequencers that do not preserve the order of events from the MIDI files they import. Within this code care is taken to preserve the order of events as specified, but some sequencers seem to transmit events occurring at the same time in an arbitrary order. By setting this parameter to `True` something of a work-around is performed: each successive event (of which there are three or four for this event type) is placed in the time stream a small delta from the preceding one. Thus, for example, the controllers are set before the data bytes in this call.

makeRPNCall(*track*, *channel*, *time*, *controller_msb*, *controller_lsb*, *data_msb*, *data_lsb*, *time_order=False*)

Perform a Registered Parameter Number Call

Parameters

- **track** – The track to which this applies
- **channel** – The channel to which this applies
- **time** – The time of the event
- **controller_msb** – The Most significant byte of the controller. In common usage this will usually be 0
- **controller_lsb** – The Least significant Byte for the controller message. For example, for a fine-tuning change this would be 01.
- **data_msb** – The Most Significant Byte of the controller's parameter.
- **data_lsb** – The Least Significant Byte of the controller's parameter. If not needed this should be set to `None`
- **time_order** – Order the control events in time (see below)

As an example, if one were to change a channel's tuning program:

```
makeRPNCall(track, channel, time, 0, 3, 0, program)
```

(Note, however, that there is a convenience function, `changeTuningProgram`, that does this for you.)

Controller number 6 (Data Entry), in conjunction with Controller numbers 96 (Data Increment), 97 (Data Decrement), 98 (Registered Parameter Number LSB), 99 (Registered Parameter Number MSB), 100 (Non-Registered Parameter Number LSB), and 101 (Non-Registered Parameter Number MSB), extend the number of controllers available via MIDI. Parameter data is transferred by first selecting the parameter number to be edited using controllers 98 and 99 or 100 and 101, and then adjusting the data value for that parameter using controller number 6, 96, or 97.

RPN and NRPN are typically used to send parameter data to a synthesizer in order to edit sound patches or other data. Registered parameters are those which have been assigned some particular function by the MIDI Manufacturers Association (MMA) and the Japan MIDI Standards Committee (JMSC). For example, there are Registered Parameter numbers assigned to control pitch bend sensitivity and master tuning for a synthesizer. Non-Registered parameters have not been assigned specific functions, and may be used for different functions by different manufacturers.

The `time_order` parameter is something of a work-around for sequencers that do not preserve the order of events from the MIDI files they import. Within this code care is taken to preserve the order of events as specified, but some sequencers seem to transmit events occurring at the same time in an arbitrary order. By setting this parameter to `True` something of a work-around is performed: each successive event (of which there are three or four for this event type) is placed in the time stream a small delta from the preceding one. Thus, for example, the controllers are set before the data bytes in this call.

writeFile(*fileHandle*)

Write the MIDI File.

Parameters **fileHandle** – A file handle that has been opened for binary writing.

CHAPTER 6

Introduction

MIDIUtil is a pure Python library that allows one to write multi-track Musical Instrument Digital Interface (MIDI) files from within Python programs (both format 1 and format 2 files are now supported). It is object-oriented and allows one to create and write these files with a minimum of fuss.

MIDIUtil isn't a full implementation of the MIDI specification. The actual specification is a large, sprawling document which has organically grown over the course of decades. I have selectively implemented some of the more useful and common aspects of the specification. The choices have been somewhat idiosyncratic; I largely implemented what I needed. When I decided that it could be of use to other people I fleshed it out a bit, but there are still things missing. Regardless, the code is fairly easy to understand and well structured. Additions can be made to the library by anyone with a good working knowledge of the MIDI file format and a good, working knowledge of Python. Documentation for extending the library is provided.

This software was originally developed with Python 2.5.2 and made use of some features that were introduced in 2.5. More recently Python 2 and 3 support has been unified, so the code should work in both environments. However, support for versions of Python previous to 2.7 has been dropped. Any mission-critical music generation systems should probably be updated to a version of Python supported and maintained by the Python foundation, lest society devolve into lawlessness.

This software is distributed under an Open Source license and you are free to use it as you see fit, provided that attribution is maintained. See License.txt in the source distribution for details.

CHAPTER 7

Installation

he latest, stable version of MIDIUtil is hosted at the Python Package Index and can be installed via the normal channels:

```
pip install MIDIUtil
```

Source code is available on Github , and be cloned with one of the following URLS:

```
git clone git@github.com:MarkCWirt/MIDIUtil.git
# or
git clone https://github.com/MarkCWirt/MIDIUtil.git
```

depending on if you want to use SSH or HTTPS. (The source code for stable releases can also be downloaded from the Releases page.)

To use the library one can either install it on one's system:

```
python setup.py install
```

or point your `$PYTHONPATH` environment variable to the directory containing `midiutil` (i.e., `src`).

MIDIUtil is pure Python and should work on any platform to which Python has been ported.

If you're using this software in your own projects you may want to consider distributing the library bundled with yours; the library is small and self-contained, and such bundling makes things more convenient for your users. The best way of doing this is probably to copy the midiutil directory directly to your package directory and then refer to it with a fully qualified name. This will prevent it from conflicting with any version of the software that may be installed on the target system.

CHAPTER 8

Quick Start

Using the software is easy:

- The package must be imported into your namespace
- A MIDIFile object is created
- Events (notes, tempo-changes, etc.) are added to the object
- The MIDI file is written to disk.

Detailed documentation is provided; what follows is a simple example to get you going quickly. In this example we'll create a one track MIDI File, assign a tempo to the track, and write a C-Major scale. Then we write it to disk.

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

from midiutil import MIDIFile

degrees  = [60, 62, 64, 65, 67, 69, 71, 72] # MIDI note number
track    = 0
channel  = 0
time     = 0   # In beats
duration = 1   # In beats
tempo    = 60  # In BPM
volume   = 100 # 0-127, as per the MIDI standard

MyMIDI = MIDIFile(1) # One track, defaults to format 1 (tempo track
                     # automatically created)
MyMIDI.addTempo(track,time, tempo)

for pitch in degrees:
    MyMIDI.addNote(track, channel, pitch, time, duration, volume)
    time = time + 1

with open("major-scale.mid", "wb") as output_file:
    MyMIDI.writeFile(output_file)
```

There are several additional event types that can be added and there are various options available for creating the MIDIFile object, but the above is sufficient to begin using the library and creating note sequences.

The above code is found in machine-readable form in the examples directory. A detailed class reference and documentation describing how to extend the library is provided in the documentation directory.

Have fun!

CHAPTER 9

Thank You

I'd like to mention the following people who have given feedback, bug fixes, and suggestions on the library:

- Bram de Jong
- Mike Reeves-McMillan
- Egg Syntax
- Nils Gey
- Francis G.
- cclauss (Code formating cleanup and PEP-8 stuff, which I'm not good at following).
- Philippe-Adrien Nousse (Adphi) for the pitch bend implementation.
- meteorsw (https://github.com/meteorsw) for major restructuring and clean-up of code.

CHAPTER 10

Indices and tables

- genindex
- search

Index

Symbols

__init__() (midiutil.MidiFile.MIDIFile method), 17

A

addControllerEvent() (midiutil.MidiFile.MIDIFile method), 18
addCopyright() (midiutil.MidiFile.MIDIFile method), 18
addKeySignature() (midiutil.MidiFile.MIDIFile method), 18
addNote() (midiutil.MidiFile.MIDIFile method), 19
addPitchWheelEvent() (midiutil.MidiFile.MIDIFile method), 11, 19
addProgramChange() (midiutil.MidiFile.MIDIFile method), 19
addSysEx() (midiutil.MidiFile.MIDIFile method), 20
addTempo() (midiutil.MidiFile.MIDIFile method), 20
addText() (midiutil.MidiFile.MIDIFile method), 20
addTimeSignature() (midiutil.MidiFile.MIDIFile method), 20
addTrackName() (midiutil.MidiFile.MIDIFile method), 21
addUniversalSysEx() (midiutil.MidiFile.MIDIFile method), 21

C

changeNoteTuning() (midiutil.MidiFile.MIDIFile method), 9, 21
changeTuningBank() (midiutil.MidiFile.MIDIFile method), 10, 22
changeTuningProgram() (midiutil.MidiFile.MIDIFile method), 10, 22

M

makeNRPNCall() (midiutil.MidiFile.MIDIFile method), 23
makeRPNCall() (midiutil.MidiFile.MIDIFile method), 23
MIDIFile (class in midiutil.MidiFile), 17

W

writeFile() (midiutil.MidiFile.MIDIFile method), 24