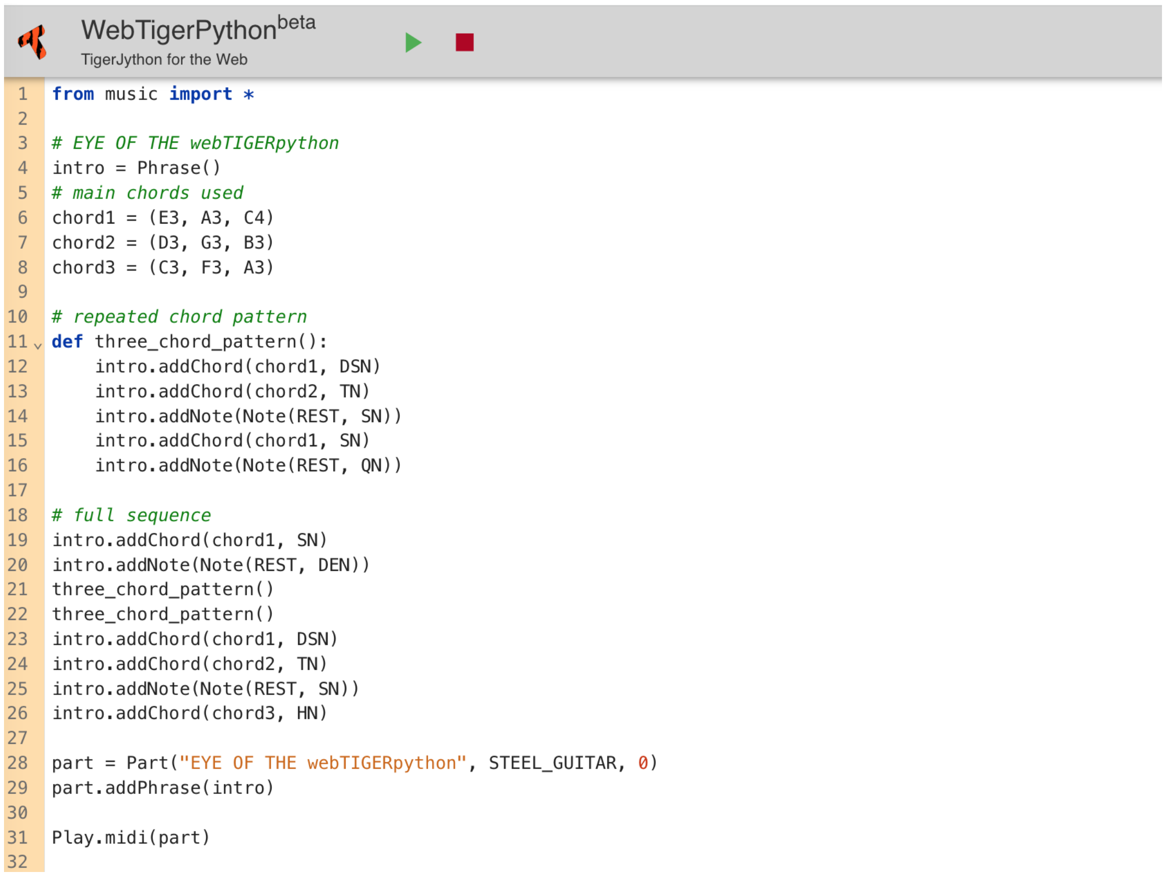
ETH zürich

Department of Computer Science
Algorithms and Didactics



```
from music import *

# EYE OF THE webTIGERpython
intro = Phrase()
# main chords used
chord1 = (E3, A3, C4)
chord2 = (D3, G3, B3)
chord3 = (C3, F3, A3)

# repeated chord pattern
def three_chord_pattern():
    intro.addChord(chord1, DSN)
    intro.addChord(chord2, TN)
    intro.addNote(Note(REST, SN))
    intro.addChord(chord1, SN)
    intro.addNote(Note(REST, QN))

# full sequence
intro.addChord(chord1, SN)
intro.addNote(Note(REST, DEN))
three_chord_pattern()
three_chord_pattern()
intro.addChord(chord1, DSN)
intro.addChord(chord2, TN)
intro.addNote(Note(REST, SN))
intro.addChord(chord3, HN)

part = Part("EYE OF THE webTIGERpython", STEEL_GUITAR, 0)
part.addPhrase(intro)

Play.midi(part)
```

JythonMusic for WebTigerPython

Natasha Savic

Bachelor's Thesis

25 May 2025

Supervisors: Prof. Dr. Dennis Komm
Alexandra Maximova
Clemens Bachmann

Abstract

WebTigerPython is an online Python IDE supporting many functionalities including Turtle Graphics, GPanel, and Robotics [1]. This thesis augmented its feature set through the addition of a music library, allowing users to generate and edit music programmatically as well as export their creations as MIDI files. The library was based on JythonMusic's music library [2], which is already in active circulation and contains a structured approach to music making. Additionally, the created library added support for live recording via external MIDI controllers.

In order to reach this goal, JythonMusic's functionalities were implemented using Web Audio API [3] and Tone.js [4] for sound playback and Mido library [5] to parse data into the MIDI file format for file export. Web MIDI API [6] assisted with the MIDI device input and live performance recording features.

As a result, WebTigerPython now contains its very own music library mimicking JythonMusic. Users can create, modify and playback musical material, upload and listen to `.wav` and `.aif` audio files and change their sound trajectory, download generated material as MIDI files, upload and edit MIDI files, make use of metronomes, and record live performances on MIDI instruments, all of which can simply be done from within WebTigerPython's code editor. These additional features broaden the spectrum of WebTigerPython's use cases by adding new possibilities for creative and musical outlet while programming in and outside the classroom.

Acknowledgment

I would like to thank Prof. Dr. Dennis Komm for enabling me to pursue this thesis at the Algorithms and Didactics group.

I also would like to extend my gratitude to my project supervisors Alexandra Maximova and Clemens Bachmann, who were available for help with both technical as well as organisational queries at every step of the process and provided me with invaluable feedback and continuous support.

A special thank you goes to Prof. Dr. Tobias Kohn for the helpful suggestions and guidance throughout.

Lastly, I would like to thank my family and friends for their encouragement and support.

Contents

Chapter 1

Introduction

The goal of this thesis was to create a working library replicating JythonMusic's music library for WebTigerPython. Ideally, the created library would mimic the existing library's behaviour as closely as possible. Moreover, it should also provide support to connect external MIDI devices and access them through the WebTigerPython code editor to play and record melodies for added music functionality.

1.1 Motivation

WebTigerPython is a browser-based Python IDE [7] created by ABZ, ETH Zurich's Center of Computer Science Education [1]. Its built-in support for Turtle Graphics, GPanel, and Robotics renders it a useful platform for academic environments. As music is a commonly used tool in classroom settings to spark interest and increase motivation [8], expanding WebTigerPython's features to provide a creative, musical outlet could broaden the IDE's application and potentially the scope of the target user demographic.

JythonMusic is a library that allows users to generate and manipulate music using Jython [2]. It is already in use in select established educational and creative groups [2], which makes it an ideal candidate as WebTigerPython's musical extension. To enhance JythonMusic's music library's functionalities, additional support for external MIDI instruments was also to be added.

Prior to this thesis, there was no support for any type of music features, i.e. neither for any of JythonMusic's functionalities nor for MIDI device input.

1.2 Main Contributions

The main contributions of this thesis are comprised of the music library created, providing the functionalities of JythonMusic's music library for WebTigerPython. It was implemented in the front-end with Web Audio API [3] and Tone.js [4] for the JavaScript auditory functionalities, and Mido library [5] for the parsing of MIDI file contents to Python objects. Additional support for MIDI device input was also added by means of the Web MIDI API [6]. For the code editor input, the existing WebTigerPython configuration with Pyodide was used.

Chapter 2

Background

This chapter serves as an overview of the main technologies and libraries used for the implementation. It includes WebTigerPython's infrastructure [1] and its compatibility with JythonMusic's music library [2], numerous tools used for browser-based sound processing such as Web Audio API [3], Web MIDI API [6] and Tone.js [4], alongside the MIDI protocol [9] and a corresponding Mido library [5]. Lastly, the instrument soundfont FluidR3-GM [10] is mentioned.

2.1 WebTigerPython

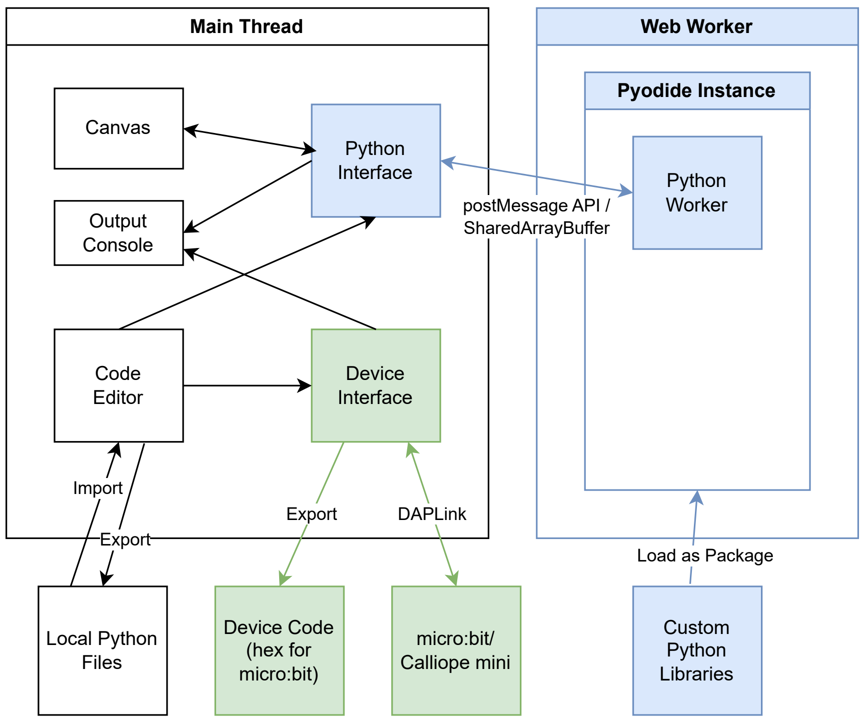


Figure 2.1. WebTigerPython's prior infrastructure [7]

WebTigerPython allows users to interact with it as with any standard IDE by entering their Python code into the code editor, clicking the run button, and potentially getting feedback via the output console depending on their program [11]. Both the editor and the console are routed through the Python interface to reach the browser's Python worker. The code is run directly in the browser by running Pyodide [12] in a Web Worker. Using the CPython interpreter, Pyodide converts the Python code into WebAssembly, which is then executed directly in the browser [12].

There was support for local Python file uploads and select device connectivity (Calliope mini, micro:bit) [1].

2.2 JythonMusic

JythonMusic is a library for Jython containing an abundance of classes and methods to write, create, and edit music using the Jython Environment for Music (JEM) [2]. It contains all quintessential functionalities for programmatic melodic composing and songwriting by programming in Jython, allowing users to create entire musical scores starting from individual notes, to melodies and chords, to multiple melodies and multi-layered tracks with different instruments. It also allows for exporting, importing, and, in special cases, manipulating MIDI files. An exhaustive list of all available functions can be found in the official JythonMusic documentation [2].

2.3 MIDI

The **musical instrument digital interface**, or **MIDI**, is a protocol originally created for digital music synthesizers [9, p. 1 - 9]. It is the industry standard for most forms of digital music and for connecting electronic instruments to a computer. A MIDI message is a binary file structured as illustrated below [13].

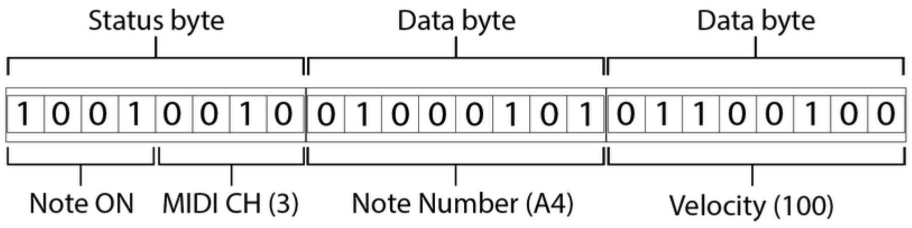


Figure 2.2. Structure of a MIDI message [13]

The data bytes encode the characteristics of each note recorded, such as its pitch or velocity (i.e., volume). By following the instructions of the corresponding message's data encoding, the device is able to reproduce the audio recorded on the MIDI controller, e.g. on a MIDI keyboard. Thus, MIDI files do not contain any actual musical or audio files, but rather the instructions for the recipient to follow in order to reproduce the audio independently [9].

2.4 Used Frameworks

Web Audio API, Web MIDI API

Web Audio API works by principle of an audio routing graph consisting of audio nodes [3]. Initially, users need to define an audio context and their sources within the context. These sources can be either sound file samples or mathematically computed oscillator synthesizer nodes. The input can then be altered by routing through effect nodes, such as GainNodes for volume control, StereoPannerNodes for spatial sound design, high-/low-/bandpassfiltering effect nodes, delay effect nodes, and more. To complete the audio routing graph, the final effect of the chain must be linked to the destination of the audio context.

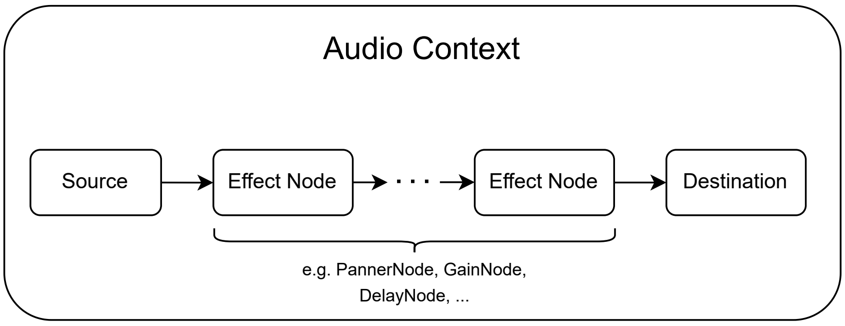


Figure 2.3. Web Audio API's audio routing graph approach

Web MIDI API provides MIDI functionality [6]. Upon requesting access, it uses event listeners to detect changes in in-/output device connectivity and activity, and handles the messages accordingly. It is designed with minimal delay, rendering it optimal for real-time use and hence for playing or recording music.

Tone.js

Tone.js is a framework built on Web Audio API [4]. In addition to simplifying the process, it offers additional support for tempo and time, scales, and other traits often needed in musical projects. For example, it has built-in synthesizers with oscillators, functions to easily trigger a note being played, a bpm attribute, and support for scheduling, which allows for manipulation and temporal shifting of sound files.

Tone.js has a Midiclass responsible for conversion calculations between different musical representations [14]. It can convert MIDI note numbers (0-127) to frequencies or the equivalent pitch names, and can transpose them. It requires files to be in JSON format to work with, so it cannot directly interact with MIDI files, and since Tone.js is more high-level, it also does not provide low-level support for the actual MIDI in- or output on a hardware level.

Mido Library

MIDI messages are binary files. Mido, MIDI Objects for Python, is a library used to parse the MIDI `NOTE_ON`, `NOTE_OFF` and other triggers into and from their correct MIDI message formats and treat them as Python objects.

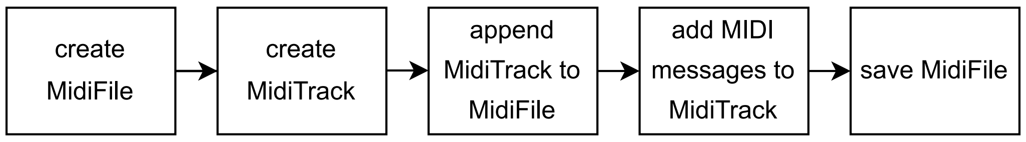


Figure 2.4. Parsing Python objects into MIDI with Mido library

Initially, a new MidiFile representing an empty `.mid` file is created and a MidiTrack is appended. MidiTracks are lists of MIDI events constituting individual melodies. The Mido messages get appended to the MidiTrack, where the messages are defined as:

```
    msg = Message(msg_type, note=pitch, velocity=velocity,
       time=difference)
```

Calling `save()` is the last step in saving the MIDI file.

FluidR3_GM Soundfont Library

FluidR3_GM is an open-source collection of `.mp3` sound files. It provides a wide range of soundfont samples that can be used in MIDI-based music playback, and is compatible with playback use from directly within the browser. JythonMusic's provided instrument samples align with those of the soundfont, making it an ideal candidate for this project.

Chapter 3

Design

In this section, the implemented project's design is explored by analyzing JythonMusic's structure, its constants, object hierarchy and modification features. Various supported file types including their import and export, as well as handling performance and playback, are mentioned. The limitations and deviations from the original library, especially in regard to external MIDI device input, are highlighted at the end.

3.1 JythonMusic's Structure

As the objective was to replicate JythonMusic as accurately as possible, it is imperative to understand its structure, as it predominantly dictated the structure of the created library.

JythonMusic contains various libraries, but the only one relevant to this project was the music library, which supports some **MIDI constants** [15].

MIDI Constants

Each musical note can have a **pitch**, denoting the actual musical note or frequency equivalent to notes octaves C-1 to G9, a **duration**, and a **dynamic**, describing its volume. Further, users can define and modify notes' **panning** for non-mono audio and the **instrument** sound sample used for the music playback. **Drums and percussion** sound samples reside in a separate category, as their sound samples are not mappable to conventional notes. Lastly, users can make use of some predefined scale constants stored in **scale and mode**.

Transcription

JythonMusic's atomic elements are individual **notes** [16]. Of the aforementioned attributes, notes inherently have a pitch and duration, and their further attributes such as volume, pan, and length, can optionally be defined as well. **Notes** can be combined to create **phrases** [17] by assembling either note sequences, rests, or chords consisting of multiple notes. Stacking several **phrases**, either simultaneously, sequentially, or shifted by a pre-defined offset, creates objects of type **part** [18]. A **part** necessitates all of its **phrases** to be played by the same instrument. It is possible to layer different instrument **parts**, resulting in **scores** [19]. At each of these levels, there are accessor and mutator functions, allowing users to readily modify their objects [16–19].

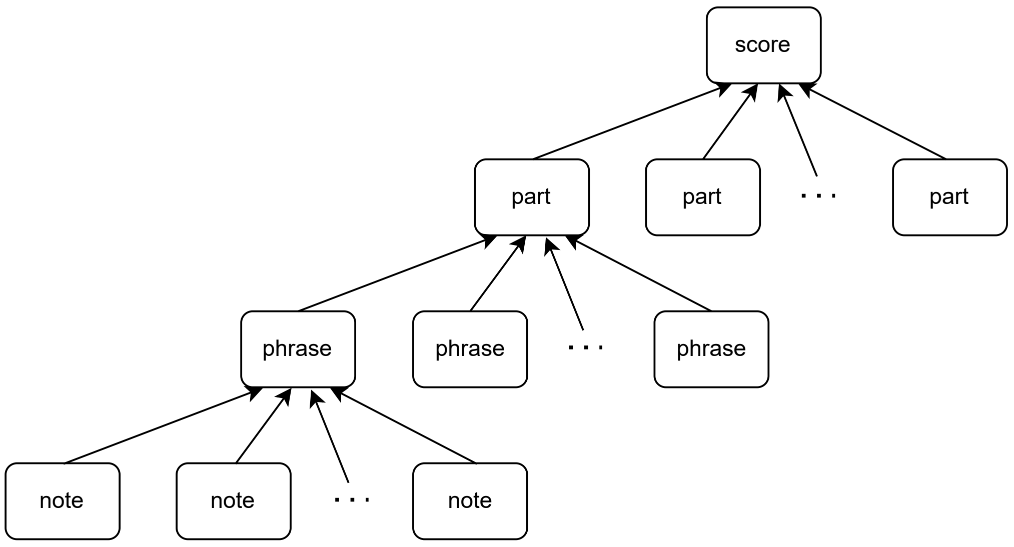


Figure 3.1. JythonMusic's hierarchical `note-part-phrase-score` structure

Composition

Mod

As part of its music composition functionalities, JythonMusic provides a number of **Mod functions** to alter the created melodies in retrospect [20]. Notable examples are appending phrases to each other, repeating phrases, palindrome functionalities, cycles, inversion, transposing, and many more, which all affect the original input's durations and pitches. Additionally, sound design modifications are also possible with accents, crescendo, fade-in/-out and compression functions. Lastly, this class incorporates randomization with shuffle and other randomized methods.

View

There are a few functions to visualize the music [21]. However, as all visual and GUI components were out of scope and thus omitted, they are not relevant to this project.

Read/Write

The **Read** and **Write** classes [22] assist users with conversions to and from MIDI files.

```
Read.midi(score_1, "uploaded_midi_file.mid")
```

accepts user-uploaded MIDI files, converts them into **score** objects, and saves them into a previously instantiated `score_1` **score** object. For the inverse operation,

```
Write.midi(score_1, "exported_midi_file.mid")
```

can be used to export any type of music material (here a **score** object `score_1`, but it can be any instance of **phrases**, **parts** or **scores**) into a MIDI file `"exported_-midi_file.mid"`.

Performance

Users can perform all of the above-mentioned musical manipulation functionalities not only on created musical scores, but also on uploaded audio samples of the `.wav` or `.aif` file format. The **performance** class allows for playback, stopping, pausing, and resuming audio samples, as well as modifications regarding the volume trajectory of each sound sample over time [23].

It is possible to treat created musical objects as audio samples by instantiating a `MidiSequence` with music material, and then similarly playing, stopping, pausing, and resuming as with audio samples [24].

Metronomes are also part of this class, either outputting audible clicks or alternatively displaying their metronome ticks on the console. Its tempo and time signature are adjustable [25].

Play

The **Play** class contains varying playback functionalities depending on whether users want to play material using a digital synthesizer (`Play.midi()`), play an uploaded audio sample (`Play.audio()`), or play a musical piece they've programmed (`Play.code()`) [26].

Mapping Values, Pitch and Frequency, Microtonality

There are a number of methods to convert between pitch names given as constants and their corresponding frequencies [27], as well as to map to certain note ranges [28]. Microtonality, which describes intermediate steps between full notes [29], is an additional supported feature that however was not implemented as part of the scope of this project.

3.2 Design Deviation from JythonMusic

One of the goals of this project was to enable MIDI device input. Outside of JythonMusic's music library, there is a MIDI library that permits users to connect a MIDI device via the `MidiIn` class to trigger generic functions as with any type of external controller [30]. However, there is no built-in functionality to directly play or record music from an external MIDI device such as a MIDI keyboard. For this reason, a slightly different approach to the `MidiIn` class was taken in order to mitigate this shortcoming. Thus, instead of the commands `onNoteOn()` and `onNoteOff()`, methods `startMIDIrecording()` and `endMIDIrecording()` were implemented.

Chapter 4

Implementation

This chapter outlines how the fundamental features were realized and provides insight into the mechanisms of the library.

4.1 Hierarchical Structure

The levels **note** → **phrase** → **part** → **score** were implemented as simple Python classes with the necessary attributes and functions residing in the Pyodide Web Worker. Any attribute modifications in a higher level are propagated down to the lowest note level to ensure synchronicity at all times. Certain modifying functions pertaining to sound design, such as left-right audio panning or volume control, cannot be realized in the Web Worker and hence additionally contain a callback to the main thread via `postMessage` as

```
defaultrunner.callback('music', data=['setdynamic', dynamic])
```

There, the functionality is implemented using Tone.js' `Tone.Panner` and `Tone.Volume`, which are wrappers for Web Audio API's `StereoPannerNode` and `GainNode` respectively.

4.2 MIDI Functionality

MIDI File Export

In order to enable users to export their musical creations as universally recognized MIDI files (corresponding to JythonMusic's `Write.midi()` functionality [22]), the Mido "Midi Objects for Python" library [5] was used. First, the entire score is iterated over to retrieve its corresponding parts, phrases and notes, creating an event list of all notes in the score. Chords are flattened to lists of pitches in order to account for each note in the chord separately. For each note detected, two events are generated and added to the event list: a `NOTE_ON` element with its start time, and a second `NOTE_OFF` element with start time + δ for δ = note's duration.

This event list maintains timestamps of all note-activity given in ticks, which is the time unit used in MIDI files [31]. Additionally, it keeps track of other attributes relevant to the MIDI encoding, namely the pitch and velocity. Only once all notes have been accounted for in the event list can the list be sorted by timestamps to ensure correct processing of simultaneous chord notes as well as notes of different parts or phrases being played concurrently. Should any `NOTE_ON` and `NOTE_OFF` timestamps coincide, priority is given to the `NOTE_ON` event. This ensures smooth transitions between notes and prevents notes with zero duration from sustaining infinitely.

The sorted event list is manually converted into a list of Mido message objects, whose attributes are derived from the corresponding event dictionary and translated from absolute time to delta time. This conversion is crucial, as MIDI files do not store absolute timestamps, but rather encode rest durations between events. Each message gets appended to a new `MidiTrack`, which is eventually added to a `MidiFile`, the in-memory representation of a MIDI file yet to be serialized. Once it is serialized into a bytes object, it is transferred to the main thread via `postMessage`, where it is saved, resulting in a download in the user's browser of the created MIDI file.

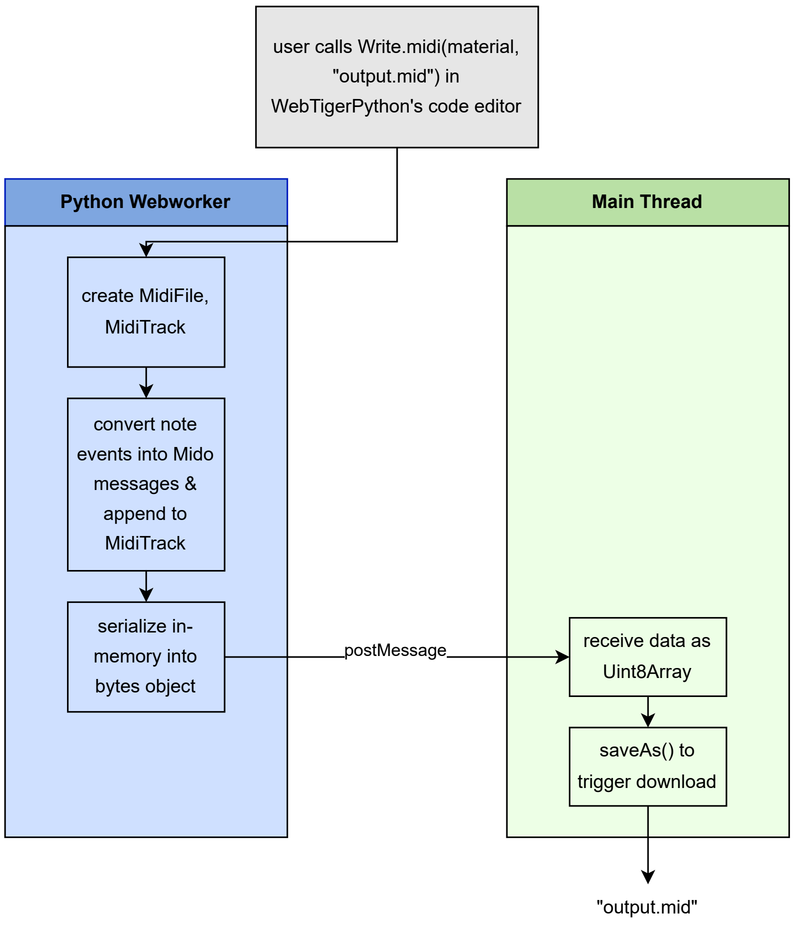

user calls Write.midi(material, "output.mid") in WebTigerPython's code editor
Python Webworker
Main Thread
create MidiFile, MidiTrack
convert note events into Mido messages & append to MidiTrack
serialize in-memory into bytes object
postMessage
receive data as Uint8Array
saveAs() to trigger download
"output.mid"

Figure 4.1. Exporting music material as MIDI file

MIDI File Import

JythonMusic also provides a `Read.midi()` function [22], which permits users to import MIDI files and dissect them into the **note/phrase/part/score**-hierarchy. As WebTigerPython already supports file-upload, additional acceptance of `.mid` files was added to the preexisting `uploadFile()` method. These files are read as ArrayBuffers and parsed in the frontend with `@tonejs/midi` into a `midi` JavaScript object [32]. Iterating over all of the object's tracks and notes, the data is manually mapped to JSON format and stored in the in-browser virtual file system via `putFileFS()`. When a MIDI file is to be processed into **score, part, phrase** or **note** objects, the saved JSON data is passed to the Python side, where it is deconstructed back into JythonMusic's hierarchical data model.

MIDI Device Recording and File Export

As JythonMusic's music library does not directly support external MIDI device connection and the focus of this thesis was on providing core music functionality, a bonus MIDI recording feature that is not directly provided by JythonMusic was added.

Web MIDI API's MIDI device connectivity is initialised using `requestMIDI Access()` of the API's Navigator interface [33] in the main thread, granting access to MIDI in-/output devices connected to the browser. Once access is established, real-time MIDI `NOTE_ON` and `NOTE_OFF` messages can be processed by means of the `onmidimessage` handler, where we distinguish between **0b10010000** (=144) for `NOTE_ON` and **0b10000000** (=128) for `NOTE_OFF`.

For `NOTE_ON` triggers, the note is added to a list of active notes, including the respective pitch and velocity data. During recording, this is also when a note playback is triggered via Tone.js' synthesizers to mimic actual playing. Additionally, the timestamp of when the note was triggered is stored. For `NOTE_OFF` triggers, the note's initial start-time timestamp is subtracted from the current timestamp to derive the total note duration, which is written to the list of active notes. All affected notes are pushed to the final recording list and deleted from the active note list. If this is during recording, the playback triggerattack started earlier is now released.

Once the recording is halted, a new track is added to the `Midi` object via `midi.addTrack()` and populated with all the recording list data by iterating over all notes. The approach here differs slightly from the regular MIDI file export, as we remain in the main TypeScript thread, hence we make use of `@tonejs/midi`'s **Midi** class for this. Lastly, calling `midi.toArray()` converts our data into a binary MIDI file which is then saved in the same fashion as our regular MIDI file export.

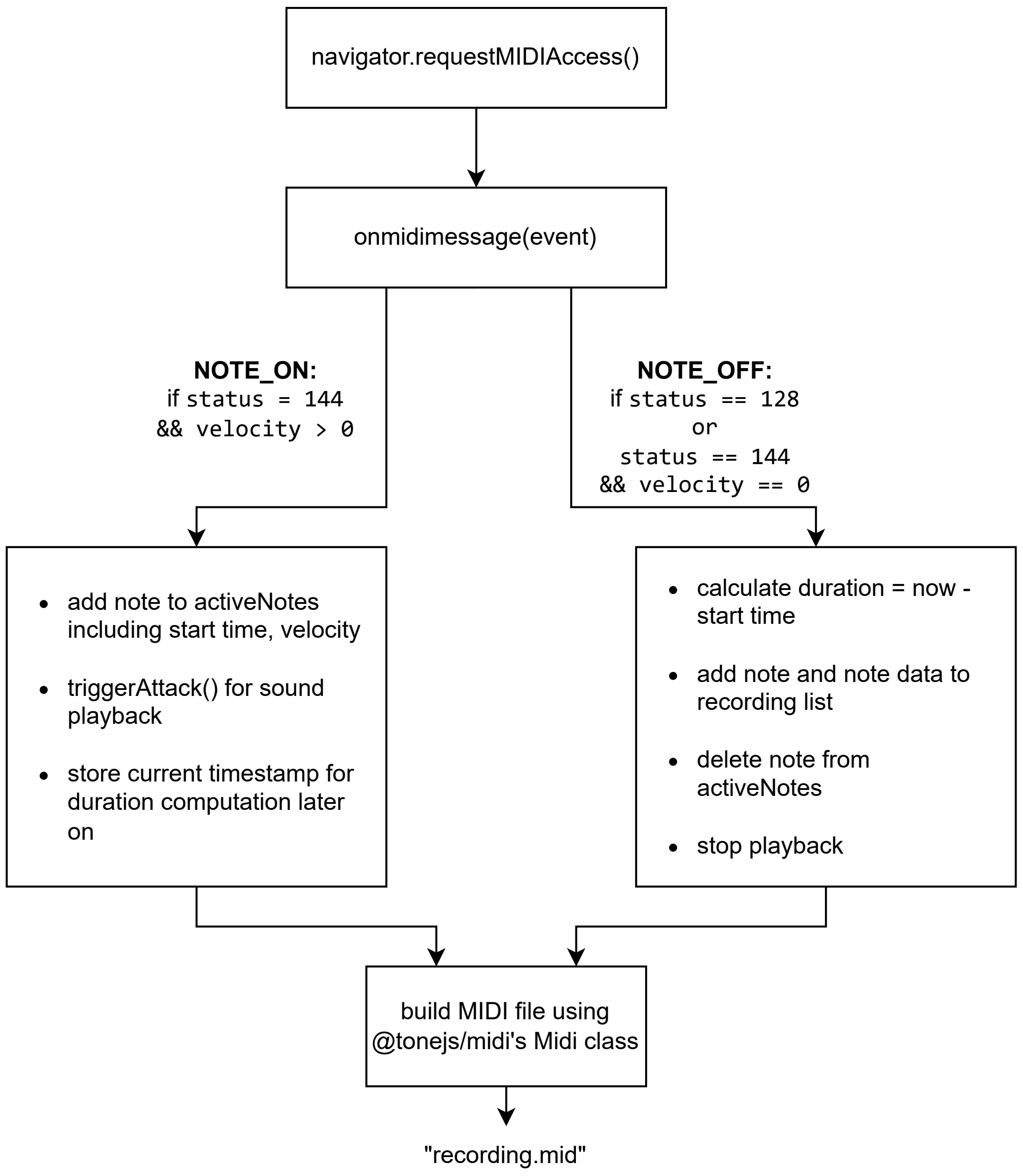

navigator.requestMIDIAccess()
onmidimessage(event)
NOTE_ON:
if status = 144
&& velocity > 0
NOTE_OFF:
if status == 128
or
status == 144
&& velocity == 0
add note to activeNotes including start time, velocity
triggerAttack() for sound playback
store current timestamp for duration computation later on
calculate duration = now - start time
add note and note data to recording list
delete note from activeNotes
stop playback
build MIDI file using @tonejs/midi's Midi class
"recording.mid"

Figure 4.2. Real-time recording with external MIDI controller

As of now, there is only support for single device connectivity at once.

4.3 Metronome

The JEM provides metronome functionalities according to JythonMusic's Metronome objects [25]. The WebTigerPython implementation is done entirely using Tone.js' Transport scheduling system [34]. Upon starting a metronome, a global audio transport is started which initiates a recurring tick event using `Tone.Transport.scheduleRepeat()` [34]. The default interval is 4n, corresponding to a default 4/4 time signature. If sound is toggled, the first tick of every bar is synthesized to a higher pitch tick, followed by the remaining lower pitch ticks for the downbeats of the bar. Storing and updating `globalAbsoluteBeat` (i.e. the total beat counter) and `beatInMeasure` (i.e. the `globalAbsoluteBeat` mod the time signature's denominator) constants at every beat enables this distinguishment. The concrete attributes such as time signature, bpm, or metronome output type (audio vs. console tick logs) can be modified using the implemented Metronome class' functions.

There was one feature set of JythonMusic's metronome class that could not be implemented with the given structure of WebTigerPython, namely the function scheduler. This permits users to define functions to be triggered recurringly on a specified beat of the metronome, which demands that multiple threads run concurrently. Given the system's setup with Pyodide driving the code in the browser's main thread in a blocking manner, it becomes clear that there is no possible circumvention that would warrant this functionality. Unfortunately, this means a deviation from JythonMusic, with the current implementation of this method simply returning an error message. The similarly functioning Play.code() function's implementation was omitted for the same reason.

However, it remains possible to set up an audible metronome while playing and recording on an external MIDI instrument with neither the two procedures nor their playback interfering with one another.

4.4 Mod Functions

Most of the implemented **mod** functions work by initially identifying the type of input, recursively traversing these nested structures until the atomic note objects are reached, applying the transformations to the individual note objects or their ordering, and finally updating the structure's end-times where applicable in case of lengthening or shortening. It's important to note that deep copies of the note objects are created where needed (such as in cycle or palindrome) to avoid corrupting the original objects.

The Mod class is not limited to structural modifications, but also allows for sound modifications, which were all implemented using Tone.js' effect nodes (`StereoPannerNode` and `GainNode` for panning and volume control) and functions (`linearRampToValueAtTime(), rampTo()` for changing volume for crescendos or fade-ins).

4.5 Audio Files

The `AudioSample` class provides mechanisms to upload, playback, loop, pause, resume and stop audio files. As opposed to MIDI Files, audio file samples' output is not intended to be generated or synthesized by following the files' instructions, but is instead directly played using Tone.js' `Tone.Player`.

The `fileUpload()` method was extended to now also support `.wav`, `.aif` and `.aiff` file uploads. The file's contents are wrapped in a Blob and stored directly in the virtual file-system, with the filename being the unique key to fetch the corresponding file. It is only decoded into a `Tone.Buffer` once and cached, so subsequent calls can reuse the existing instance and avoid unnecessarily decoding the same files multiple times. Every time a new sample is played, all previous volume and panning settings are reset. Starting and stopping the playback as well as looping is implemented using Tone.js' `Player.start()`, `Player.stop()` methods and `loopStart`, `loopEnd` parameters. When pausing or resuming files, the passed time is stored as an offset and used as an input parameter to the `Player.start()` function.

4.6 Sound Design

Envelopes are used to describe the sound's intensity trajectory over time [35].

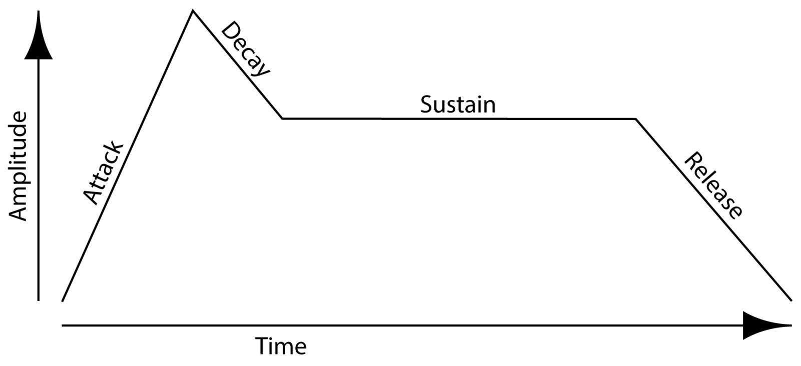


Figure 4.3. The 4 ADSR stages of an Envelope: Attack, Decay, Sustain, Release [36]

The initial duration until the sample's maximum peak is reached is referred to as the **Attack**. This is followed by a **Decay** period until the **Sustain** amplitude is reached. **Sustain** denotes the phase of consistent volume level before fade-out. Lastly, the **Release** is the remaining attenuation duration until complete halt, i.e. until an amplitude of zero is reached. [35, 37]

JythonMusic allows users to instantiate envelopes on uploaded AudioSample files and adjust its ADSR-stages [36]. To mimic this functionality, once again Tone.js methods such as `linearRampToValue()` and `setValueAtTime()` were used on the AudioSample's volume nodes after a conversion of the volume input into decibels.

4.7 Playback Using Instrument Samples

By default, playback of the created musical structures is performed by Tone.js' synthesizer. With JythonMusic's collection of instrument samples, it is also possible to select one of 128 instrument [38] samples to reproduce the music instead. To implement this, **FluidR3_GM** [10] was used. It is a popular, open-source MIDI soundfont containing the sound samples of the instruments needed for JythonMusic's implementation in `.mp3` format.

When the user selects an instrument sample with `setInstrument()`, the MIDI number corresponding to the sample is first obtained from an `INSTRUMENT_MAP` created on the Python side. This is sent along with the entire musical note data from the Pyodide worker to the main thread via `postMessage`. Here, the MIDI number is mapped to the correct sample url, which the browser uses to fetch the sample from the library in real time and cache for subsequent use. Using this sample, each note's playback is scheduled with the needed parameters using Tone.js' Sampler.

Chapter 5

Testing, Results, Issues

A combination of automated and manual testing provided further insight into both the referenced as well as the created library. The resulting observations call attention to some of the difficulties and limitations faced during the implementation stages.

5.1 Playwright Testing

For prior WebTigerPython functionality testing, Playwright was utilized, hence it was also used for this project. Playwright simulates real user interaction with the browser, performing commands according to the given instructions and comparing the desired output with the actual behaviour of the browser [39]. Playwright can compare and analyze certain artifacts such as traces and their corresponding console output, screenshots, or file contents. However, comparing sound output and audio playback is not possible. This largely limited the extent of testing that was possible with Playwright. The core functionalities that could be tested by these means were the following:

- All **structural functionalities**, from the creation and generation of different scores to the manipulation thereof using modifying functions. The volume and panning attributes' values could be checked, but the actual perceived sound and panning changes had to be manually tested.
- All **Metronome functionalities** save the sound output. As metronome ticks can also be output to the console, this was the metric used for comparison.
- **MIDI file generation**. The test checked whether the contents of the binary file were of the MIDI type and whether they match the expected content of the exported MIDI file.

The remaining functionalities all had to be rigorously manually tested to the best extent possible. This entailed invoking every function of all implemented classes and features using numerous edge cases and various intertwinings of method calls or program execution paths. The perceived audio was assessed in terms of volume, panning, musical correctness and timed output length. This output was also compared to the corresponding behaviour of the JEM where possible. Additionally, the exported MIDI files' sound and structure were inspected for correctness using the Digital Audio Workstation GarageBand [40] and Tone.js' MIDI to JSON online parsing tool [41].

5.2 Difficulties

JythonMusic-Specific Difficulties

Naming inconsistencies

Choosing the approach of replicating JythonMusic was advantageous, as having it as a reference provided guidance for structure as well as potential features to implement, especially as a starting point. However, there were some issues that arose due to this choice of approach. In particular, JythonMusic's vast functionalities are structured in a rather complex manner, making it slightly difficult to overview. This is amplified by certain naming inconsistencies of the same attributes or parameters across functionalities. For instance, depending on the function, the same **volume**-attribute takes on the term **volume** [18], **velocity** [24], or **dynamic** [16].

Even sometimes within the same class, such as in the **part** class, where the accessor and corresponding mutator methods are called `.getVolume()` and `.setDynamic(dynamic)` [18]. This may seem intuitive, but causes confusion when certain other similar parameters such as **duration** and **length** refer to two entirely different attributes.

Implementation Inconsistencies

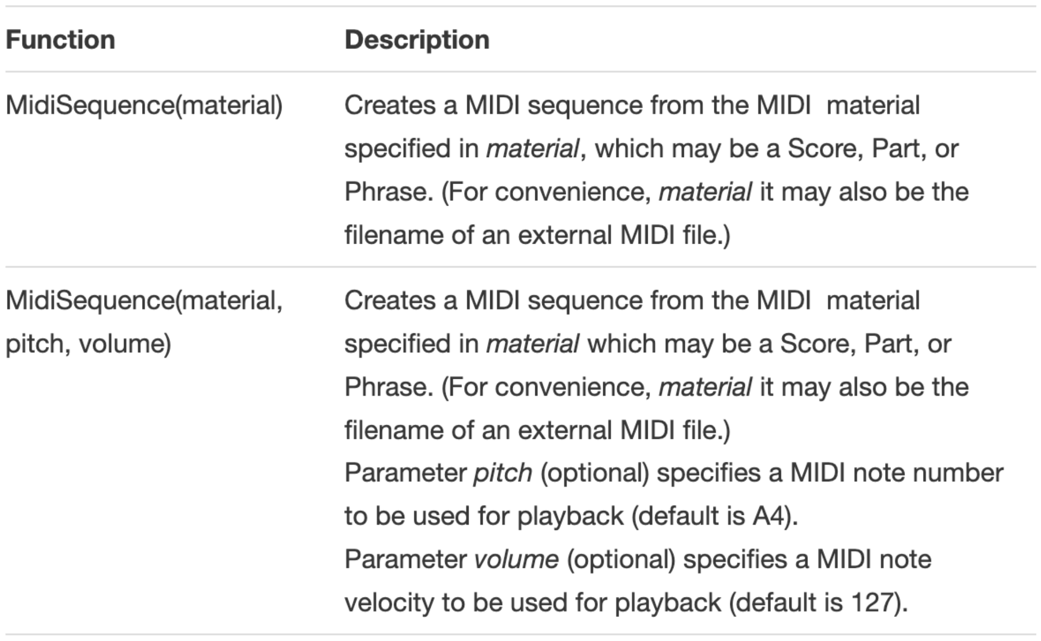
The inconsistencies linger not only in the actual naming, but can also be found between certain definitions in the documentation and their actual implementations in the JEM. As an example, the `Mod.quantize()` function's implementation in the JEM is non-existent for input values above 2, and for the remaining values, the implementation of `Mod.quantize()` does not correspond to the standard definition of quantization [42, p. 1170], which is also the definition provided in JythonMusic's documentation [20]. For this project, this perceived cutoff value was ignored and the quantize function was implemented for all integer values.

Varying Input Ranges

Comparing the input ranges of certain functions also highlighted some disparities between components. For example, setting panners for **notes, phrases, parts** or **scores** accepted values in the range [**0.0, 1.0**], whereas the panner of AudioSamples required the panner's parameters to lie within the range [**0, 127**].

Another source for confusion was the MidiSequence constructor defined in the JythonMusic documentation [24].

It is possible to create a `MidiSequence` MIDI object from some musical input. However, the optional arguments allow for further specifications, including setting the pitch or volume. Which, by definition of how MIDI audio works, poses a contradiction, as those attribute are already determined; both for **score** objects, as those are non-optional arguments of its atomic **note** objects, as well as for MIDI-files, as that is part of the MIDI data payload. Accommodating a volume attribute modification at this stage can be reasoned for; the same cannot be said for the pitch argument. The



Function	Description
MidiSequence(material)	Creates a MIDI sequence from the MIDI material specified in *material*, which may be a Score, Part, or Phrase. (For convenience, *material* it may also be the filename of an external MIDI file.)
MidiSequence(material, pitch, volume)	Creates a MIDI sequence from the MIDI material specified in *material* which may be a Score, Part, or Phrase. (For convenience, *material* it may also be the filename of an external MIDI file.) Parameter *pitch* (optional) specifies a MIDI note number to be used for playback (default is A4). Parameter *volume* (optional) specifies a MIDI note velocity to be used for playback (default is 127).

Figure 5.1. JythonMusic's MidiSequence [24]

description clarifies that the accepted pitch input must be a "MIDI note number", i.e. a single note. Does this imply overwriting the entire score or MIDI file with one note, or perhaps setting all notes of the material to that specified note? Alternatively, would it aim to transpose the entire sequence in `material` by that note, or transpose it to start at the given note? In which case, what would happen in case of transposing an already high note by another high note, crossing the threshold of the highest possible note or MIDI channel? Testing out this particularity in the JEM does not offer further insight, as adding as well as omitting values for those arguments produce the same results. It is unclear whether this is a definition or implementation error, thus for the time-being, the implementation of the JEM was followed, meaning adding arguments for pitch and volume when instantiating a `MidiSequence` object has no effect.

Overwhelming Structure

There was clearly an effort made to make JythonMusic as user-friendly as possible and provide as many possible angles and options for creation and modification. This results in certain overhead in terms of duplicate functionalities, such as being able to modify individual notes' attributes at varying levels [16,18], or being able to trigger playback in many different ways, such as instantiating material as a `MidiSequence` prior to calling `MidiSequence.play()` [24] or directly calling `Play.midi(material)` [43]. In every case, JythonMusic's Music library's current structure and functionalities were mimicked, including duplicate functionalities.

Overrides

Another example of this is the **setInstrument()** function. Users can instantiate **Parts** with a chosen instrument and one of 16 output channels as input parameters, where channels simply group playbacks sharing the same settings. It is also possible to map a specific instrument to a channel using `Play`'s `setInstrument(channel)` method. This raises the question: When both are set, which one overrides? The initial implementation of this project had aimed to treat all overrides equally and honor the order of the invocations, making the last call of `setInstrument()` the decisive party. JythonMusic, on the contrary, warns users to set instruments at the Part as opposed to at the Phrase level whenever possible to avoid undesired results. Regarding `Play` class' global `setInstrument(channel)`, there is no explicitly mentioned order of precedence. Manually testing examples in the JEM proved that `Play`'s `setInstrument` always nullifies any previous instrument assignment, so ultimately that behaviour was mimicked in this project.

Implementation Difficulties

Panner

One of the first hurdles faced was in regard to the implementation of the panner. For some sound samples or instruments, especially certain synths, panning just simply is not enabled or supported, so testing that functionality will not render any perceivable results. At first glance, this makes the implementation appear faulty, when all that was required to rectify it was choosing a different instrument.

Processing Chords

There were some initial difficulties when it came to processing the MIDI file export of simultaneous notes in chords or notes of different parts played at the same time. This was solved by flattening chords to lists of notes with the same timestamps, as well as first adding all notes part by part and sorting them by their timestamps at the end.

Approach to MIDI Export and Import

The biggest design issue was posed by the question of approach for seamless alternation back and forth between the **score** objects and MIDI files. One option would have been to immediately write every note initialized to a MIDI file, and to directly overwrite that file with any changes made. The alternative would be to remain in the hierarchical score structure for as long as possible and only writing to MIDI whenever necessary, which is only when a file export is triggered. Ultimately, the latter option was chosen. One reason for this approach is that it is significantly easier for nearly all functionality implementations to be able to access the objects and their attributes directly. For many programs, MIDI file export also will not even be used, meaning this way we can avoid the unnecessary overhead of writing each note to a MIDI file and converting back to a note object multiple times.

Browser Support

Unfortunately, Web MIDI API and Web Audio API do not provide full support for Safari and Firefox [3,6]. This is a big issue, but WebTigerPython currently also does not fully support those browsers, so the use of these APIs does not further restrict WebTigerPython's scope.

Metronome's Scheduler

JythonMusic's Metronome contains a method `add(function, parameters, beatToStart, repeatFlag)`, which allows users to create and schedule functions to be invoked once or repeatedly on specific beats of the measure. With WebTigerPython's current infrastructure, the Python code gets executed by Pyodide in a Web Worker in the browser's main thread and is blocking. There is no support for concurrency or multi-threads. Adding repeated metronome callbacks on specific beats would require a scheduler to call the added functions in a non-blocking and asynchronous fashion, which is not possible with the given setup. Therefore, this functionality had to be omitted. Instead, an error is shown for users trying to call this function.

Testing

As mentioned above, ensuring optimal test coverage was infeasible with the provided setup. The compromise was to create test cases for the attributes verifiable with Playwright, and test the remainder of the scope manually as much as possible.

Console Output

For features such as the metronome, scheduled console output to the WebTigerPython output console is necessary. However, Pyodide sends all console output to the browser's developer console instead. This issue was bypassed by using WebTigerPython's already existent `consoleStore`, which is used to manage console messages. Calling

```
consoleStore().appendOutput("...");
```

suffices to exhibit the desired output to the correct WebTigerPython console visible to users.

Keyword "repeat"

There was a collision between the WebTigerPython keyword `repeat` and JythonMusic's mod class' equivalently named function, which prevented that method from carrying the same name. An attempt was made to alias it as follows:

```
Mod.repeat = Mod.modRepeat
```

but it could not be resolved. Hence, the function instead had to be defined as `modRepeat(material, times)`.

Re-running Without Refreshing the Browser

Certain features initially worked, but re-running the same code without refreshing the browser between executions resulted in faulty behaviour. For instance, playing uploaded AudioSample files proved to be erroneous when trying to decode the same file repeatedly in different runs of the same code segment. The fetching behaviour was adapted to accommodate only decoding and caching the file once and fetching using the same key, which resolved this bug.

Microtonality

Finally, JythonMusic's microtonality feature was entirely excluded from this project's implementation. As mainly sound samples and midi notes, which are clearly defined, are being used, this functionality was omitted for the time being and could possibly be included in a future expansion of the library.

Chapter 6

Conclusion, Future Work

The final product lays the foundation for an initial musical extension of WebTiger-Python with some opportunities for expansion in the future.

Conclusion

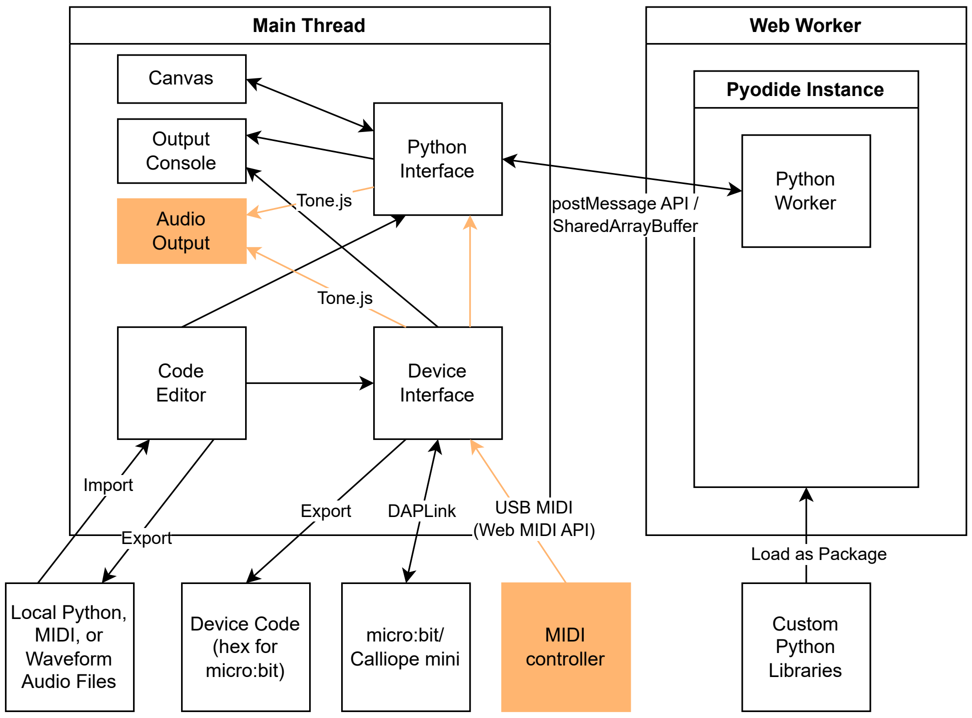


Figure 6.1. Updated WebTigerPython overview [7] with the new components highlighted

A functioning library implementing JythonMusic's core music library's features was created. This includes the structural musical components from **notes** all the way to **scores**, transformations and manipulations thereof, audible playback of that material as well as of uploaded audio files, MIDI file upload and processing, and exporting generated material as MIDI files. Additionally, support for MIDI device input was added. This enables the recording, playback and saving of musical pieces played on MIDI instruments in real-time as MIDI files.

This thesis was an addition the already versatile toolkit of WebTigerPython. JythonMusic's structure was imitated, as it is an already established library and actively in use in certain academic and artistic environments. The incorporation of these features expands the possibilities of WebTigerPython's use cases and increases its potential to motivate users to refine their Python skills while satisfying their desire for a creative, musical outlet.

Future Work

GUI Elements

The most intriguing expansion of this project would be the integration of GUI elements to visualize the musical material. Some suggestions can be found in JythonMusic's GUI library, such as displaying the generated music in staff notation, or as a piano-roll display. Alternatively, providing some visualizations in real-time would also be conceivable, such as with a piano visualizer.

Microtonality

Support for microtonality was completely forgone in this thesis. It could be beneficial to add this functionality to further close the gap between the created library and JythonMusic's library.

Drums and Percussive Sound Samples

128 instrument sound samples were integrated into this project. JythonMusic additionally allocates a separate channel and mapping for percussive instrument sound samples, which have yet to be incorporated into this thesis' library.

Metronome Functions

The metronome class' function scheduler feature could not be implemented with the current state of WebTigerPython with Pyodide. This is not ideal and should be rectified, should the infrastructure ever change.

Expand Browser Compatibility

At the very latest once all of WebTigerPython's features seamlessly support more browsers is when the created library will be the limiting factor due to Web Audio API's and Web MIDI API's browser support. If alternative APIs offering the same functionalities with better browser support come into existence, it could be worth exploring a rework of the current implementation making use of those libraries instead.

Bibliography

[1] Webtigerpython. https://python-online.ch/index.php?inhalt_links=home/navigation.inc.php&inhalt_mitte=home/webtp.inc.php/. Accessed 2024-11-28.

[2] B. Manaris and A. Brown. JythonMusic Transcription. https://jythonmusic.me/transcription/. Accessed 2025-02-18.

[3] W3C Audio Working Group. Web audio api. https://developer.mozilla.org/en-US/docs/Web/API/Web_Audio_API, 2024. Accessed 2025-05-04.

[4] Yotam Mann. Tone.js. https://tonejs.github.io/, 2014. Accessed 2025-05-04.

[5] Ole Martin Bjørndalen and Raphaël Doursenaud. Mido library. https://mido.readthedocs.io/en/stable/. Accessed 2025-05-03.

[6] W3C Web MIDI Community Group. Web midi api. https://developer.mozilla.org/en-US/docs/Web/API/Web_MIDI_API. Accessed 2025-05-04.

[7] Clemens Bachmann, Alexandra Maximova, Tobias Kohn, and Dennis Komm. WebTigerPython: A Low-Floor High-Ceiling Python IDE for the Browser, oct 2024. Working paper.

[8] Janice L. Killian and John B. Wayman. The prevalence of the use of music as a teaching tool among selected american classroom educators: A preliminary examination. Texas Music Education Research, 2015. ERIC Number: EJ1152561.

[9] Joseph Rothstein. MIDI: A Comprehensive Introduction. A–R Editions, Inc., 1995.

[10] Frank Wen. FluidR3_GM. https://member.keymusician.com/Member/FluidR3_GM/index.html. Version 1.51.1, Accessed 2025-05-19.

[11] Webtigerpython ide. https://webtigerpython.ethz.ch/. Accessed 2024-11-28.

[12] The Pyodide Development Team. Pyodide. https://pyodide.org/en/stable/. Accessed 2025-05-11.

[13] Bernardo Breve et al. Perceiving space through sound: mapping human movements into midi. In ResearchGate, 2020. Accessed: 2025-05-14.

[14] Yotam Mann. Tone.js MidiClass. https://tonejs.github.io/docs/14.9.17/classes/MidiClass.html, 2014. Accessed 2025-05-04.

[15] B. Manaris and A. Brown. JythonMusic Constants, howpublished= https://jythonmusic.me/api/midi-constants/, note = Accessed 2025-05-03.

[16] B. Manaris and A. Brown. JythonMusic Note. https://jythonmusic.me/note/. Accessed 2025-05-03.

[17] B. Manaris and A. Brown. JythonMusic Phrase. https://jythonmusic.me/api/music-library-functions/phrase/. Accessed 2025-05-03.

[18] B. Manaris and A. Brown. JythonMusic Part. https://jythonmusic.me/api/music-library-functions/part/. Accessed 2025-05-03.

[19] B. Manaris and A. Brown. JythonMusic Score. https://jythonmusic.me/api/music-library-functions/score/. Accessed 2025-05-03.

[20] B. Manaris and A. Brown. JythonMusic Mod Functions. https://jythonmusic.me/api/music-library-functions/mod-functions/. Accessed 2025-05-04.

[21] B. Manaris and A. Brown. JythonMusic View. https://jythonmusic.me/api/music-library-functions/view-functions/. Accessed 2025-05-03.

[22] B. Manaris and A. Brown. JythonMusic Read and Write. https://jythonmusic.me/api/music-library-functions/read-write/. Accessed 2025-05-04.

[23] B. Manaris and A. Brown. Jythonmusic performance. https://jythonmusic.me/performance/. Accessed 2025-05-04.

[24] B. Manaris and A. Brown. JythonMusic MidiSequence. https://jythonmusic.me/api/midisequence/. Accessed 2025-05-07.

[25] B. Manaris and A. Brown. JythonMusic Metronome. https://jythonmusic.me/metronome/. Accessed 2025-05-03.

[26] B. Manaris and A. Brown. Jythonmusic play. https://jythonmusic.me/play/. Accessed 2025-05-04.

[27] B. Manaris and A. Brown. JythonMusic Pitch and Frequency. https://jythonmusic.me/pitch-to-frequency-conversions/. Accessed 2025-05-04.

[28] B. Manaris and A. Brown. Jythonmusic mapping values. https://jythonmusic.me/mapping-values/. Accessed 2025-05-04.

[29] B. Manaris and A. Brown. JythonMusic Microtonality. https://jythonmusic.me/microtonality/. Accessed 2025-05-04.

[30] B. Manaris and A. Brown. JythonMusic MIDI Library. https://jythonmusic.me/midi-library/. Accessed 2025-05-04.

[31] Ole Martin Bjørndalen and Raphaël Doursenaud. Mido ticks. https://mido.readthedocs.io/en/stable/glossary.html#term-ticks. Accessed 2025-05-03.

[32] Yotam Mann. Tone.js Midi. https://github.com/Tonejs/Midi. Accessed 2025-05-03.

[33] W3C Web MIDI Community Group. Webmidiapi navigator. https://developer.mozilla.org/en-US/docs/Web/API/Navigator/requestMIDIAccess, 2021. Accessed 2025-05-04.

[34] Yotam Mann. Tone.js Transport. https://tonejs.github.io/docs/r13/Transport, 2014. Accessed 2025-05-04.

[35] TeachMeAudio. Sound envelopes. https://www.teachmeaudio.com/recording/sound-reproduction/envelopes, 2020. Accessed 2025-05-06.

[36] B. Manaris and A. Brown. JythonMusic Envelope. https://jythonmusic.me/envelope/. Accessed 2025-05-06.

[37] Phillip L. De Leon. Computer music in undergraduate digital signal processing. In Proceedings of the American Society for Engineering Education Gulf-Southwest Annual Conference, Las Cruces, NM, 2000. American Society for Engineering Education.

[38] B. Manaris and A. Brown. JythonMusic Instruments. https://jythonmusic.me/api/midi-constants/instrument/. Accessed 2025-05-04.

[39] Microsoft. Playwright. https://playwright.dev/. Version 1.51.1, Accessed 2025-05-03.

[40] Apple Inc. Garageband. https://www.apple.com/mac/garageband/. Accessed 2025-05-23.

[41] Yotam Mann. Tone.js' MIDI to JSON tool. https://tonejs.github.io/Midi/, 2014. Accessed 2025-05-14.

[42] Glen M. Ballou, editor. Handbook for Sound Engineers. Focal Press, Burlington, Mass, 4 edition, 2008.

[43] B. Manaris and A. Brown. JythonMusic Play.midi(). https://jythonmusic.me/play-midi/. Accessed 2025-05-07.

List of Figures

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Eigenständigkeitserklärung

Die unterzeichnete Eigenständigkeitserklärung ist Bestandteil jeder während des Studiums verfassten schriftlichen Arbeit. Eine der folgenden zwei Optionen ist **in Absprache mit der verantwortlichen Betreuungsperson** verbindlich auszuwählen:

☒ Ich erkläre hiermit, dass ich die vorliegende Arbeit eigenverantwortlich verfasst habe, namentlich, dass mir niemand beim Verfassen der Arbeit geholfen hat. Davon ausgenommen sind sprachliche und inhaltliche Korrekturvorschläge der Betreuungsperson. Es wurden keine Technologien der generativen künstlichen Intelligenz[1] verwendet.

☐ Ich erkläre hiermit, dass ich die vorliegende Arbeit eigenverantwortlich verfasst habe. Dabei habe ich nur die erlaubten Hilfsmittel verwendet, darunter sprachliche und inhaltliche Korrekturvorschläge der Betreuungsperson sowie Technologien der generativen künstlichen Intelligenz. Deren Einsatz und Kennzeichnung ist mit der Betreuungsperson abgesprochen.

Titel der Arbeit:

JythonMusic for WebTigerPython

Verfasst von:
Bei Gruppenarbeiten sind die Namen aller Verfasserinnen und Verfasser erforderlich.

Name(n):	**Vorname(n):**
Savic	Natasha

Ich bestätige mit meiner Unterschrift:
- Ich habe mich an die Regeln des «Zitierleitfadens» gehalten.
- Ich habe alle Methoden, Daten und Arbeitsabläufe wahrheitsgetreu und vollständig dokumentiert.
- Ich habe alle Personen erwähnt, welche die Arbeit wesentlich unterstützt haben.

Ich nehme zur Kenntnis, dass die Arbeit mit elektronischen Hilfsmitteln auf Eigenständigkeit überprüft werden kann.

Ort, Datum	**Unterschrift(en)**
Zürich, 23.05.2025	Natasha Savic

Bei Gruppenarbeiten sind die Namen aller Verfasserinnen und Verfasser erforderlich. Durch die Unterschriften bürgen sie grundsätzlich gemeinsam für den gesamten Inhalt dieser schriftlichen Arbeit.

[1] Für weitere Informationen konsultieren Sie bitte die Webseiten der ETH Zürich, bspw. https://ethz.ch/de/die-eth-zuerich/lehre/ai-in-education.html und https://library.ethz.ch/forschen-und-publizieren/Wissenschaftliches-Schreiben-an-der-ETH-Zuerich.html (Änderungen vorbehalten).