impromptu

interactive music notation

Team members: Ivy Malao, Zoë Naidoo, Annie Chen (Mengyuan Chen), Rachel Pinsker, Zakir Gowani, Sofia Wyetzner

MILESTONE 3.B

Library dependencies to be downloaded and installed:

Front end:

1. LilyPond (Macs: `http://lilypond.org/macos-x.html`, Windows: `http://lilypond.org/windows.html`)
2. Abjad 2.17 (http://abjad.mbrsi.org/) *run* "`pip install abjad`"
3. Flask (http://flask.pocoo.org/) *run* "`pip install Flask`"

Backend:

4. Python-midi (https://github.com/vishnubob/python-midi*) Download and extract the ZIP located at the GitHub link. Change into the directory and run* "`python setup.py install`"

(1) and (2) how to compile and run code

When inside the `src` folder, type '`Python app.py`' into the terminal in order to run the program.

Go to the URL http://127.0.0.1:1995/ where the web application should be running on your localhost.

You can also run the backend functionality independently by typing '`Python Tune.py -f [filename]`' into the terminal, where [filename] is a MIDI file path, e.g. '`c-major-scale-treble.mid`'. The program will print out the MIDI file pattern and all of the Notes in the MIDI file to the terminal.

Note that the "obsolete" folders contain older versions of code that we are keeping as reference.

(3) how to run the unit test cases

Backend tests: When inside the src folder, run: `python ../tests/backendtests.py`

Frontend tests: When inside the src folder, run: `python ../tests/flaskFrontendTests.py`

(4) please suggest some acceptance tests for the TA to try (i.e., what inputs to use, and what outputs are expected)

Main acceptance tests are uploading valid MIDI files. To do this:

1. Download MIDI files provided in tests/MIDITestFiles. Other files can be downloaded from: https://www.basicmusictheory.com/major-scales if you would like to test more.
2. Click "UPLOAD A MIDI FILE".
3. Choose a MIDI File.
4. Click "Convert". The page will reload and display a PDF.

The correct PDFs for each of the MIDI files are also in the MIDITestFilesFolder with a name matching the name of the corresponding MIDI file to be compared to for accuracy.

Please note for our first iteration, we provide **support only for one Format 1 MIDI files** (i.e. channel data is not all on one track) **and will append additional tracks to the end of the tune (**i.e., only one note is played at a time so there are no chords). Additionally, **MIDI files must contain both NoteOn and NoteOff events in the track**. You can print the MIDI file pattern to see whether these conditions are satisfied following instructions in item (2). Further information can be found in item (5) description. All of the provided MIDI Files should render.

Additional acceptance tests:

1. Enter a title and click the "-->". The title should display on the PDF and after reloading the webpage, should still show at the top of the PDF.
2. Enter contributors and click the "-->". The contributors should display on the PDF and after reloading the webpage, should still show at the top of the PDF.
3. Click "Upload A MIDI File". Without choosing a file, click "Convert". The last PDF will still be displaying.
4. Enter bad strings for title and contributors. These include a string longer than 128 characters, a string that is just spaces, or a string that has a newline character. The PDF should not update.

(5) and (6) text description of what is implemented (can refer to the use cases and user stories) and who did what and who was paired with whom

We split into frontend and backend groups. The front end group was Sofia, Rachel, and Zakir; the backend group was Ivy, Zoë, and Annie.

Frontend (Rachel, Sofia, Zakir)
Implemented all code in app.py, as well as all html, javascript, and css files.

- *Create initial website template*: set up a webpage to allow for the uploading of a file, entering of strings for name and contributors, and the display of a PDF.
- *User can upload MIDI files to website*: make sure that MIDI files, and only MIDI files, can be uploaded and stored and the server.
- *Display static sheet music as pdf by rendering the Tune object*: after providing the backend with the MIDI file, do conversions between our application's internal representation of a Tune to something compatible as input for abjad to then create a lilypond file and PDF.
- *User can download Sheet Music (as a pdf)*: once the MIDI file has been converted and displayed, the user should be able to download the pdf.
- *User can edit title and names (Composer, Producer, Arranger as contributors)*: user can enter strings for title and names (in this string, user can specify Composer, Producer, Arranger. For example, one could enter "Composer: x, Producer: y, Arranger: z" into the contributors text box. One could also just enter "x, y, z" to add general contributors.

The main goal for the frontend was to provide a working webpage that gives the user an interface to use our application. We worked with the backend team to provide a MIDI file from the user to be converted, and then used their returned information on that file to render the Tune visually as a PDF.

How we split up the work:

- At first, we all three "pair" programmed together to create a very initial website template. At this point we worked together to set up Flask for our site after deciding that using AJAX requests was not a good way to build this application. Additionally, we worked together on how to create lilypond files and use abjad to generate PDFs.
- **Rachel:** Worked on the generation of the displayed PDF, specifically writing makeLilypondFile(), updatePDFWithNewLY(lilypondFile), tuneToNotes(tune), and parts of tune() related to the conversion into a PDF. Worked on the parts of tune() related to storing of title and contributors, so that user input into these fields is saved in the backend and rendered again when the page is refreshed. Also wrote new python tests: test_delete_old_PDF(), test_make_lilypond_file(), test_tune_to_notes(), and test_title_and_name_input().
- **Sofia:** Primarily worked on the UI. Created the HTML page and styled it with CSS, including the button and form functionality. Implemented the Flask side of changing the title and contributors (the if statements under "POST" in app.py) along with the editing that portion of the lilypond file. Wrote the python test: test_change_title_and_contributors().
- **Zakir:** Drafted frontend buttons and wrote Flask handler for MIDI file upload and extension validation (in the functions tune(), allowed_file(), uploaded_file()). Wrote the corresponding test_midi_upload() function.

Backend (Zoë, Annie, Ivy)
Implemented all of the classes (note that constructors take in a dictionary to allow a variable number of input arguments):

- Enum classes: *Duration* (sixteenth, eighth, quarter, half, or whole note), *Clef* (treble or bass), and *Accidental* (natural, sharp, or flat)
- *Pitch*: includes the letter (a-g), octave (0-8), and Accidental. The MIDInotetoPitch method takes a MIDI note and generates a Pitch instance (e.g. MIDI note 60 is pitch C4).
- *Note*: has a frequency, Duration, duration in seconds, onset in seconds (relative to start of MIDI track), and pitch. The frequency attribute is not used in the first iteration because pitches can be computed directly from MIDI notes but will be useful for future iterations involving audio file inputs (e.g. mp3).
- *Key:* includes a isMajor boolean and a Pitch
- *Tune*: includes a Time Signature, Key, clef, Title of piece, list of contributors, MIDI file, and list of Notes. The constructor uses the MIDI file and constructs the list of Notes using helper other methods in the Tune class.

Generate a Tune instance given MIDI file: The main goal for the backend was to parse a MIDI file and construct a Tune object with that information, which we then passed on to the frontend for them to render as sheet music in the web application. MIDI files are represented as a hierarchical set of objects: a Pattern containing a list of Tracks. For our first iteration, we worked with only one Track MIDI files (i.e., only one note is played at a time so there are no chords). A Track is a list of MIDI events, which has information about a note's onset, duration, and MIDI pitch and which can then be used to compute the duration of rests. For our first iteration, we worked only with MIDI files that have both NoteOn and NoteOff events to compute duration.

How we split up the work:

- We did not do formal pair-programming but rather, we each implemented the classes and methods corresponding to the unit tests we wrote for Milestone 2. Afterwards, we met and together reviewed, revised, and ran all of the code and tests.
- **Ivy:** Wrote the toString methods used for printing class instances in debugging, MIDInotetoPitch method in the Pitch class (where the mapping is stored in dictionary as a Pitch class attribute), and the Tune constructor. The Tune constructor parses the midi file for the Time Signature and to compute the list of Notes. Ivy also helped Annie with the calculateRests method in the Tune class.
- **Zoe:** Wrote the enum classes, Pitch class, Note class, Key class, and did the bulk of the work in revising the unit tests (see item (7) on changes below).
- **Annie:** Wrote the tickstoTime, secondstoDuration, computeOnset, and calculateRests methods in the Tune Class. computeOnset takes a MIDI file, calculates onset of the sounding notes, and returns a list of Notes representing each sounding note in the MIDI. It uses the Python-midi library to parse the MIDI file. The returned list is then taken by calculateRests, which calculates rest Notes based on the spaces between the notes. It returns a new list of note Notes and rest Notes, ordered by onset. This list becomes the notes attribute in a Tune object. tickstoTime and secondstoDuration assist with converting the midi library values to our Note attributes.

(7) changes: have you made any design changes or unit test changes from earlier milestones?

Frontend:

We decided to use Flask because our original thought to use javascript and AJAX requests turned out to be unnecessary for what we were trying to implement. This meant we needed to rewrite our unit tests from JavaScript to python to test our Flask implementations. The functionality of the unit tests and the test cases are very similar, though they have all been rewritten. The biggest changes are as follows:

1. The tests for displaying the static sheet music have been broken down into smaller steps based on the python functions written (converting tune object to abjad notes and making lilypond files).
2. We no longer need to test for downloading the PDF functionality because browsers provide that for us when we display the PDF.
3. We added a test for deleting an old PDF when a new file is uploaded so that the currentTune folder only has one PDF at a time.
4. There are no longer button-click tests because these events have been converted to form submissions.

Backend:

Classes:

1. In the Note class, we *added an attribute s_duration* to represent the duration of a Note in seconds. We found that it was helpful to have this value for calculation purposes. We eliminated the computeNoteOrder function (which combines a list of pitched Notes and a list of Rests into one list) since we implemented its functionality in calculateRests (it was easier to do so).

2. We also decided to *use the Python-midi library for parsing MIDI files rather than the aubio library*, because the Python-midi library is easier to work with and has better documentation. Furthermore, the aubio library functionality is not best suited for working with MIDI files and is designed primarily for audio files (i.e. working with frequencies). This removes the need to test methods related to the Note frequency attribute in this iteration.

Unit tests:

3. Following item 1, we removed our computeNoteOrder tests and revised our calculateRests tests.

4. Following item 2, we had to revise our unit tests to work with the Python-midi library and to use MIDI files as inputs rather than constructed Note objects with frequencies. This made the testComputeFrequency test unnecessary as we did not implement a ComputeFrequency method for the Note class. The testComputeNotes test was implemented with Notes based on frequency, so is likewise, unecessary for this iteration

(8) other: whatever you want to let the TA know

How to submit

Please create a directory in code repository for this milestone, and submit through your code repository.

Send an email to us to let us know when it is ready for us to checkout

Deadline

11:59pm, May 11th