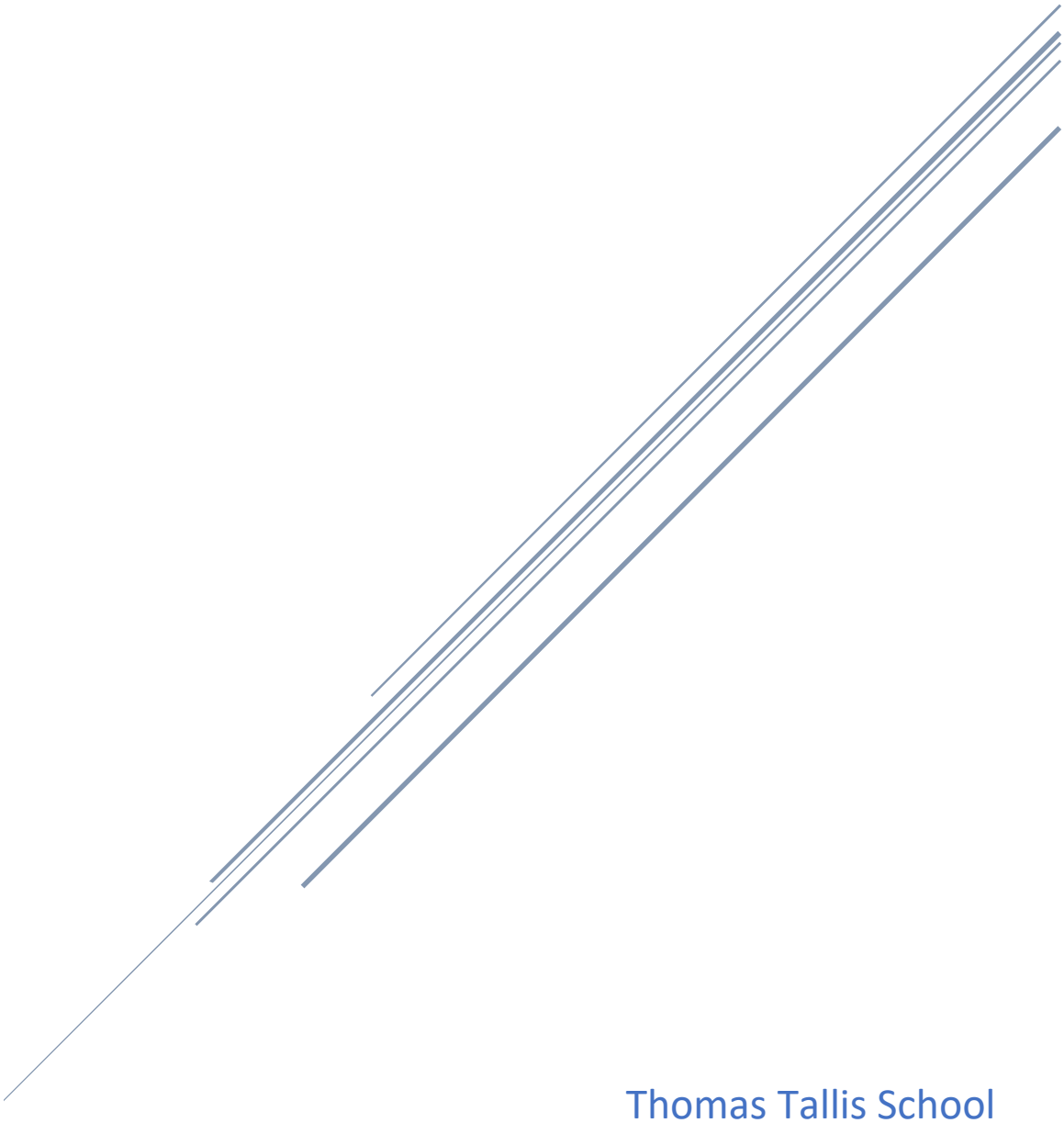
A LEVEL PROGRAMMING PROJECT

Music Composing Website - ScoreLily



Thomas Tallis School
Beauty Oluokun

Table of Contents

Developing the Coded Solution

First Cycle

Setting Up the Database

After creating the 'AlevelProject' Django project, I created the accounts app and the libraries app. These apps act as the 'User Accounts' and 'Score Libraries' modules respectively.

```
PS C:\Users\BOluo\djangoprojects\ALevelProject> py manage.py startapp accounts
PS C:\Users\BOluo\djangoprojects\ALevelProject> py manage.py startapp libraries
```

In the models.py file inside the accounts app directory I defined the User class. Then in the models.py file in the libraries app directory I defined the Score class.

```python
from django.db import models

# Create your models here.

#User model that creates User table in database
#Has username (primary key), email  and password attributes
class User(models.Model):
    email = models.EmailField()
    username = models.CharField(max_length = 50, primary_key = True)
    password = models.CharField(max_length = 100)
    def __str__(self):
        return str(self.username)
```

Both the Score and User classes inherit from Django's Model class which allows then to be used to created database tables.

However, I forgot the public attribute.

```python
from django.db import models
from accounts.models import User
from django.utils import timezone

# Create your models here.

#Score model that creates the public library table in the database. Contains all scores.
#Has scoreID (primary key), title, author, date_created, last_modified, score_file,
#composition_file, music_file, pdf_file and user (foreign key) attributes
#Scores specific to users can be filtered with the user attribute
class Score(models.Model):
    scoreID = models.AutoField(primary_key = True)
    title = models.CharField(max_length = 200)
    author = models.CharField(max_length = 200)
    date_created = models.DateTimeField(default = timezone.now)
    last_modified = models.DateTimeField(auto_now = True)
    score_file = models.FileField(upload_to = "", max_length = 1000000)
    composition_file = models.FileField(upload_to = "", max_length =1000000, default = "None")
    music_file = models.FileField(upload_to = "", max_length = 10000000, default = "None")
    pdf_file = models.FileField(upload_to = "", max_length = 100000000, default = "None")
    user = models.ForeignKey(User, on_delete = models.CASCADE)
    def __str__(self):
        return str(self.title) + ' - ' + str(self.author)
```

By running the makemigrations and migrate commands in the terminal, I created the database.

```
PS C:\Users\BOluo\djangoprojects\ALevelProject> py manage.py makemigrations
Migrations for 'accounts':
  accounts\migrations\0001_initial.py
    - Create model User
Migrations for 'libraries':
  libraries\migrations\0001_initial.py
    - Create model Score
PS C:\Users\BOluo\djangoprojects\ALevelProject>
```

Experimenting with Mingus Library

```
from mingus.containers import Note, NoteContainer, Bar, Instrument, Track, Composition, Piano, Guitar, MidiInstrument
from mingus.extra import lilypond
import random

noteSelection = [Note('A'), Note('B'), Note('C'), Note('D'), Note('E'), Note('F'),
        Note('G'), Note('Ab'), Note('Bb'), Note('C#'), Note('Db'),
        Note('Eb'), Note('F#'), Note('G#')
        ]
#Creating composition
c = Composition()

#Creating tracks for piano, guitar and a MidiInstrument
track1 = Track(Piano()) #Remember brackets after Instrument object!!
track2 = Track(Guitar())
track3 = Track(MidiInstrument('Violin'))

#Adding tracks to composition
c + track1
c + track2
c + track3
print(c)
c.title = "test score"
c.author = "Beauty"

#Randomly adding notes to tracks
for t in c.tracks:
  t + Bar('C', (3, 4))
  for i in range(8):
    t + random.choice(noteSelection)
print(c)

# Converting to lilypond format
lilystr = lilypond.from_Composition(c)
print(lilystr)

#Creating pdf from lilypond
print(lilypond.to_pdf(lilystr, 'testPDF'))
```

To test how I could use the 'to_PDF' function to create a pdf file from a Composition object I created a small program that randomly generates a Composition object with tracks and notes and then converts the Composition information into the lilypond format as a string; this string is then used to create a pdf of the score.

When I ran this program, I didn't get any errors, but a pdf file wasn't created despite the fact the lilypond.to_pdf function returned a 'True' value so show it had executed without any issues. So, I opened the lilypond.py file in the mingus/extra directory in IDLE and amended the save_string_and_execute_LilyPond function (the function called within the to_pdf function) so that it would actually create a pdf file when run. The red comment is a line in the original code. I changed that line to:

command = ".\%s.ly" %(filename)

I also added the line:

os.remove(filename + '.log')

```
def save_string_and_execute_LilyPond(ly_string, filename, command):
    """A helper function for to_png and to_pdf. Should not be used directly."""
    ly_string = '\\version "2.10.33"\n' + ly_string
    if filename[-4:] in [".pdf", ".png"]:
        filename = filename[:-4]
    try:
        f = open(filename + ".ly", "w")
        f.write(ly_string)
        f.close()
    except:
        return False
    #command = 'lilypond %s -o "%s" "%s.ly"' % (command, filename, filename)
    command = ".\%s.ly" %(filename)
    print("Executing: %s" % command)
    p = subprocess.Popen(command, shell=True).wait()
    os.remove(filename + ".ly")
    os.remove(filename + ".log")
    return True
```

instruments.py

```
59
60  class AcousticBass(MidiInstrument):
61      name = "Acoustic Bass"
62      range = (Note('E', 1), Note('Eb', 5))
63      def __init__(self):
64          MidiInstrument.__init__(self)
65
66  class ElectricBass(MidiInstrument):
67      name = " Electric Bass (finger)"
68      range = (Note('E', 1), Note('Eb', 5))
69      def __init__(self):
70          MidiInstrument.__init__(self)
71
72
73  class Violin(MidiInstrument):
74      def __init__(self):
75          MidiInstrument.__init__(self)
76          self.name = "Violin"
77          self.range = (Note('G', 3), Note('A', 7))
78          self.instrument_nr = 41
79
80  class Viola(MidiInstrument):
81      name = "Viola"
82      range = (Note('C', 3), Note('E', 6))
83      def __init__(self):
84          MidiInstrument.__init__(self)
```

```
import random
from mingus.containers import Note, Instrument, Track, Bar, Composition, MidiInstrument
from instruments import Violin, Clarinet
from mingus.midi import midi_file_out

m = MidiInstrument()
m.name = 'Acoustic guitar'
m.instrument_nr = 24
m_track = Track(m)

violin_track = Track(Violin())
clarinet_track = Track(Clarinet())

violin_track.add_bar(Bar('A', (3, 4)))
clarinet_track.add_bar(Bar('A', (3, 4)))

noteSelection = [Note('A'), Note('B'), Note('C'), Note('D'), Note('E'), Note('F'), Note('G')]
for i in range(8):
  violin_track + random.choice(noteSelection)
  clarinet_track + random.choice(noteSelection)
  m_track + random.choice(noteSelection)

print(m_track)
#print(violin_track)
#print(clarinet_track)

c = Composition()
c + m_track
#c + violin_track
#c + clarinet_track

print(midi_file_out.write_Composition('midiTest.midi', c))
```

I defined several different classes that inherit from Mingus' MidiInstrument class to represent the different instruments options. Then to test the midi_file_out.write_Composititon function I created a small program that randomly generates a Composition object and uses the Composition information to create a midi file. I originally tried to use two of the instrument classes I defined myself to create the Composition object but, even though a midi file was created, the voice of the instruments in the audio didn't match – i.e. the violin and clarinet sounded like a grand piano. However, when I used a MidiInstrument object directly and changed the instrument_nr attribute of the object, the voice in the audio matched the instrument it was supposed to represent. I then decided not to use the classes I originally designed and to use MidiInstrument objects directly instead.

Coding Score Editor Subroutines

```
from mingus.containers import Track, Note, Composition, Bar, MidiInstrument
import mingus.core.value as value
import mingus.core.keys as keys
from mingus.extra import lilypond
from mingus.midi import midi_file_out
from django.shortcuts import get_object_or_404
from libraries.models import Score
import os
import subprocess
```

I created the editor app, which acts as the score editor module, and created the BuildScore.py file to create the score editor subroutines. These are the import statements at the top of file.

```
#Creates and returns a new score record linked to curent user
def new_score(title, current_user):
    #removing spaces from title so it can be used in file names
    title_strip = title.replace(' ', '_')
    #Creating empty files needed
    p = subprocess.Popen('echo > editor/static/libraries/%s'%(title_strip+'.ly'), shell=True).wait()
    p = subprocess.Popen('echo > editor/static/libraries/%s'%(title_strip+'_comp.txt'), shell=True).wait()
    p = subprocess.Popen('echo > editor/static/libraries/%s'%(title_strip+'_music.midi'), shell=True).wait()
    p = subprocess.Popen('echo > editor/static/libraries/%s'%(title_strip+'.pdf'), shell=True).wait()
    #Creating new Score record
    score_new = Score(title = title, author = current_user.username, score_file = title_strip+".ly",
    composition_file = title_strip+'_comp.txt', music_file = title_strip+'_music.midi',
    pdf_file = title_strip+'.pdf', user = current_user)
    score_new.save()
    return score_new
```

In the new_score function, I first make the variable 'title_strip' which contains a string the same as the argument 'title' but has all the spaces in the replaced with underscores because the string used in the file names shouldn't have spaces in it. Then I created the 4 files needed for the score record using the subprocess.Popen function to open a temporary terminal and run the echo command. Then I create a new Score object and call the save method on it which permanently adds a new Score record to the database. Finally, the Score object, 'score_new' is returned.

```
#Creates and returns a new composition object for new score made
def new_composition(current_score):
    #Creating Composition object
    comp = Composition()
    #Setting the title and author
    comp.set_author(current_score.author)
    comp.set_title(current_score.title)
    return comp
```

In the new_composition function, I create a new Composition object and then set author and title attributes to corresponding attributes of the 'current_score' argument.

```
def save_comp(composition, score_id):
    # Opens text file that is going to hold the
    #information to save composition object details
    # Each track is separated by a '?'. Instrument name and
    #bars are separated by a ';'. Bar details are separated by a '~'.
    score = get_object_or_404(Score, pk = score_id)
    file = score.composition_file
    file.open('w')
    comp_string = ''
    #Iterating through each track
    for t in composition.tracks:
        #Saving the track name (instrument name)
        comp_string += t.instrument.name + ';'
        for b in t.bars:
            #Saving the key, meter and bar list of
            #each Bar in the track
            comp_string += b.key + '~'
            comp_string += str(b.meter[0]) +'~'
            comp_string += str(b.meter[1]) + '~'
            comp_string += str(b.bar) + ';'
        comp_string += '?'
    file.write(comp_string)
    file.close()
    score.save()
```

In the save_comp procedure, the argument 'score_id' is to retrieve the Score object representing the selected record from the database. Django's get_object_or_404 function is used to do this as it retrieves the record or flags up a 404 error if record is not found. The object representing the text file used to store the composition data is then put into the variable 'file'. This file is then opened in write only mode and an empty string is put in the variable 'comp_string'. A for loop is used to iterate through each Track object within the Composition object. For each track, the name of the instrument is added to 'comp_string' then a for loop is used to iterate through each Bar object in the Track object. For each bar, string versions of the key, two integers in the metre tuple and list of notes within the bar are added to 'comp_string'. Different punctuation is used to separate the different elements. Then 'comp_string' is written to the text file and the score record is saved.

```
#Creates new composition object containing all of the information
#to replicate previously saved score
def remake_composition(score):
    #Reading composition object details kept in text file into a string
    file = score.composition_file
    file.open('r')
    comp_string = file.read()
    file.close()
    #Creating composition object
    comp = Composition()
    comp.set_author(score.author)
    comp.set_title(score.title)
    #Splitting up tracks
    tracks_list = comp_string.split('?')
    for t in tracks_list:
        t_list = t.split(';')
        #Choosing correct instrument for track
        instrument_name = t_list.pop(0)
        instrument = select_instrument(instrument_name)
        if instrument != None:
            track = Track(instrument)
            #Adding bars to the track
            for b in t_list:
                b_list = b.split('~')
                new_bar = Bar(b_list[0], (int(b_list[1]), int(b_list[2])))
                new_bar.bar = list(b_list[3])
                track + new_bar
            comp + track
    return comp
```

In the remake_composition function, the object representing the text file used to store the composition data put in the variable 'file'. The file is then opened in read only mode and the text in the file is copied as a string to the variable 'comp_string'. A Composition object is then put in the variable 'comp' and its author and title attributes are set. Then a list of strings is created and put in the variable 'tracks_list' by breaking up 'comp_string' by the delimiter '?'. The list created contains a string per track in the saved composition object. A for loop is then used to iterate through each track string and for each the string is split by the delimiter ';' into a list and put in the variable 't_list'. The first element is the instrument name, and this is popped from the list and passed into the function select_instrument to obtain the correct MidiInstrument object (code explained later). A new Track object is made using the MidiInstrument object and put in the variable 'track'. A for loop is then used to iterate through the remaining elements in 't_list' which are the strings representing each bar in the track. Each bar string is split by the delimter '~' into a list and put in the variable 'b_list'. The first element in 'b_list' is the key of the bar, the second is the first number or the metre tuple, the third is the second number of the meter tuple and the last element is the list that was in the bar attribute of the Bar object whose data was saved. A Bar object is created using the key and metre information and put in the variable 'new_bar' and then the bar attribute of 'new_bar' is set to the list version of the last element in 'b_list'. This new Bar object is then added to the variable 'track'. When a Track is done being created, it is then added to the variable 'comp'. When all needed tracks have been added, the variable 'comp' is returned.

```
#Updates score when change is made
def update_score(current_score, composition):
    #Converting composition object into lilypond format
    lilypond_string = '\\version "2.10.33"\n' + lilypond.from_Composition(composition)
    #Updating lilypond file in score record
    lily_file = current_score.score_file
    lily_file.open('w')
    lily_file.write(lilypond_string)
    lily_file.close()
    #Creating pdf format of file
    command =  "./editor/static/libraries/%s" %(lily_file.name)
    p = subprocess.Popen(command, shell=True).wait()
    os.remove(lily_file.name[:-3] + ".log")
    #Updating MIDI file in score record
    midi_file = current_score.music_file
    m_path = "./editor/static/libraries/%s" %(midi_file.name)
    midi_file_out.write_Composition(m_path, composition)
    current_score.save()
```

In the update_score procedure, the variable 'lilypond_string' is created which contains the string '\\version "2.20.33"\n' concatenated with the string returned from the lilypond.from_Composition function being called on the 'composition' argument. This converts the Composition object data into the lilypond format and stores it as a string so it can easily be written to a lilypond file.

Instead of using the lilypond.to_pdf function (my modified version) to create the pdf file of the score, as my algorithm design stated, I decided to write out the code required to convert the lilypond string into a pdf. I chose to do this because the file needed is represented by an instance of Django's FileField class which the lilypond.to_pdf function is not coded to work with. The object representing the score's lilypond file is put in the variable lily_file, the file is opened in write only mode and the string in 'lilypond_string' is written to the file. Like in Mingus' lilypond.save_string_and_execute_LilyPond function, the variable 'command' is created which holds the path of the lilypond file as a string and the function subprocess.Popen is used to run the command in a temporary terminal. The .log file created by this command is then removed. After this, the object representing the midi file is put in the variable 'midi_file' and the path to this file as a string is stored in the variable 'm_path'. The function midi_file_out.write_Composition is then used to convert the Composition data into the MIDI format and then write it to the score's midi file. The score record is then saved.

Beginning

There is more code in the middle but it's similar so there's no reason to include all of it

Ending

```
#Selects the correct instrument when given the
#instrument name as a string
def select_instrument(instrument_name):
    if instrument_name == 'Acoustic Grand Piano':
        #instance of a MidiInstrument object is created
        #with the corresponding instrument name
        instrument = MidiInstrument(instrument_name)
        #instrument_nr attribute is changed to set the
        #sound of the instrument in the midi audio file
        instrument.instrument_nr = 1
        instrument.range = (Note('F', 0), Note('B', 8))
    elif instrument_name == 'Harpsichord':
        instrument = MidiInstrument(instrument_name)
        instrument.instrument_nr = 7
        instrument.range = (Note('F', 1), Note('F', 6))
    elif instrument_name == 'Glockenspiel':
        instrument = MidiInstrument(instrument_name)
        instrument.instrument_nr = 10
        instrument.range = (Note('G', 5), Note('C', 8))
    elif instrument_name == 'Marimba':
        instrument = MidiInstrument(instrument_name)
        instrument.instrument_nr = 13
        instrument.range = (Note('C', 2), Note('C', 7))
    elif instrument_name == 'Accordion':
        instrument = MidiInstrument(instrument_name)
        instrument.instrument_nr = 22
        instrument.range = (Note('F', 3), Note('A', 6))
```

```
    elif instrument_name == 'Bassoon':
        instrument = MidiInstrument(instrument_name)
        instrument.instrument_nr = 71
        instrument.range = (Note('Bb', 1), Note('Eb', 5))
    elif instrument_name == 'Clarinet':
        instrument = MidiInstrument(instrument_name)
        instrument.instrument_nr = 72
        instrument.range = (Note('E', 3), Note('C', 7))
    elif instrument_name == 'Piccolo':
        instrument = MidiInstrument(instrument_name)
        instrument.instrument_nr = 73
        instrument.range = (Note('D', 4), Note('C', 7))
    elif instrument_name == 'Flute':
        instrument = MidiInstrument(instrument_name)
        instrument.instrument_nr = 74
        instrument.range = (Note('C', 4), Note('D', 7))
    else:
        instrument = None
    return instrument
```

I decided to create the function select_instrument as I made it easier to convert the strings of then instrument names used in my main subroutines to MidiInstrument objects with the correct attributes. If the argument 'instrument_name' equates to one of the instrument strings (e.g. 'Acoustic Grand Piano' or 'Piccolo'), a MidiInstrument object with the name attribute set to instrument_name is stored in the variable 'instrument'. Then the instrument_nr attribute is set to the corresponding instrument's general MIDI instrument number and the range attribute is set to correct pitch range. If 'instrument_name' is not equal to any of the strings, 'instrument' is set to None. Then the variable 'instrument' is returned.

```
def select_track(instrument_track, composition):
    #Selecting the correct track
    for track in composition:
        if instrument_track == track.name:
            t = track
            return t
            break
```

In the select_track function, a for loop is used to iterate through each Track object in the argument 'composition'. Then the name of the track is compared to the argument instrument_track and if they are equal the track is returned, and the loop is broken.

In the set_bar_key procedure, a for loop is used to iterate through each Track object in the argument 'composition'. Then another for loop is used to access each Bar object in the track from the index one less than the argument 'bar_num' to the end of the track. Each bar accessed will have its key attribute set to the argument 'key_sig'.

```
#Sets the key of multiple bars in a track (from given
#bar_num to end of track)
#Returns the notes in that key as a list
def set_bar_key(composition, key_sig = "C", bar_num = 1):
    for t in composition.tracks:
        for i in range(bar_num - 1, len(t)):
            t[i].key = key_sig
```

```
#Sets the metre of multiple bars in a track (from given bar
#bar_num to the end of the track)
#If no bar given, default bar 1. If no metre given, default 4/4.
def set_bar_metre(composition, metre = (4, 4), bar_num = 1):
    for t in composition.tracks:
        for i in range(bar_num - 1, len(t)):
            t[i].set_meter(metre)
```

In the set_bar_metre procedure, like the set_bar_key function, each Bar object, from the index one less than the argument 'bar_num' to the end, in each Track object in the argument 'composition' is accessed and has its meter attribute set to the argument 'metre'.

In the add_instrument procedure, the select_instrument function is called, passing the argument instrument_name as a parameter, and the result is stored in the variable 'instrument'. A new Track object with its instrument attribute equal to the variable 'instrument' is then stored in the variable 'new_track'. The track's name attribute is then set to the instrument name. To ensure that multiple tracks with the same instrument don't have the same exact name, the variable 'n' is used as a counter for the number of tracks that have the same instrument as 'new_track'. 'n' is initially set to 1 and then each track object in the argument 'composition' is iterated through. If the name of the instrument on the track is equal to 'new_track's' instrument, then n is incremented. If, after the for loop, 'n' is greater than 1 there must be duplicate instruments. So, the name attribute of 'new_track' is appended with its duplicate number, 'n', in brackets. Then an empty Bar object is added to 'new_track' to make sure it is not empty and 'new_track' is added to 'composition'.

```
#Adds instrument track to composition object holding score
def add_instrument(composition, instrument_name):
    instrument = select_instrument(instrument_name)
    new_track = Track(instrument)
    new_track.name = instrument.name
    #Dealing with duplicate instrument names
    n = 1
    for track in composition:
        if track.instrument.name == new_track.name:
            n += 1
    if n > 1:
        new_track.name += ' (' + str(n) + ')'
    new_track + Bar()
    composition + new_track
```

Not all code for the add_note procedure has been screenshotted because they are too similar, i.e. many elif statements

```
def add_note(composition, duration, instrument_track,
    note_name = None, octave = None, bar = None, position = None):
    t = select_track(instrument_track, composition)
    #Choosing between a not and a rest
    if note_name == None and octave == None:
        new_note = None
    else:
        new_note = Note(note_name, octave)
    # Selects the length of note
    if duration == "semibreve":
        length = value.semibreve
    elif duration == "minim":
        length = value.minim
    elif duration == "crotchet":
        length = value.crotchet
```

1

```
    #Triplet rhythms
    elif duration == "triplet crotchet":
        length = value.triplet(value.crotchet)
    elif duration == "triplet quaver":
        length = value.triplet(value.quaver)
    elif duration == "triplet semiquaver":
        length = value.triplet(value.semiquaver)
    #Dotted rhythms
    elif duration == "dotted semibreve":
        length = value.dots(value.semibreve)
    elif duration == "dotted minim":
        length = value.dots(value.minim)
```

2

```
    #Selecting correct bar from chosen track
    #Adding note to the very end of track
    if bar == None and position == None:
        chosen_bar = t[-1]
        #Checking if note will fit in bar
        space_left = chosen_bar.value_left()
        #If not, splitting up the note
        if space_left < length:
            chosen_bar.place_notes(new_note, space_left)
            t.add_notes(new_note, length - space_left)
        else:
            chosen_bar.place_notes(new_note, length)
```

3

```
    #Adding note to specifc bar and position
    elif bar != None and position != None:
        chosen_bar = t[bar - 1]
        #Converting the position to the format used in the bar object
        beat = (position - 1) / (chosen_bar.length * 4)
        space_left = 1 / (chosen_bar.length - beat)
        temp1 = chosen_bar.current_beat
        chosen_bar.current_beat = beat
        if space_left < length:
            chosen_bar.place_notes(new_note, space_left)
            try:
                next_bar = t[bar]
            except IndexError:
                t + Bar()
                next_bar = t[bar]
            temp2 = next_bar.current_beat
            next_bar.current_beat = 0.0
            next_bar.place_notes(new_note, length - space_left)
            next_bar.current_beat = temp2
        else:
            chosen_bar.place_notes(new_note, length)
        chosen_bar.current_beat = temp1
```

4

In the add_note procedure, the select_track function is called, and the result is stored in the variable 't'. If the arguments 'note_name' and 'octave' are both equal to None the variable 'new_note' will be set to None – this is used when a rest is to be added. If not, new_note is set to a Note object. Then a large if-elif-else statement is used to determine length of the note. There are 3 types of durations: normal (e.g. crotchet), triplet (e.g. triplet crotchet) and dotted (e.g. dotted crotchet). Then if the arguments 'bar' and 'position' are both equal to None, the variable 'chosen_bar' is set to the last Bar object. 'space_left' holds the length of free note space left in the bar. If the space left is less than the desired length of the new note, a note that is the duration equal to 'space_left' is placed in 'chosen_bar'. Then a note that is the duration of 'length' - 'space_length' is placed in a new bar using the add_notes Track method. If not, a not that is the duration equal to 'length' is place in 'chosen_bar'. This is used when the note is to be placed at the very end of the track. If there is a specific bar and position chosen – if 'bar' and 'position' are not equal None – 'chosen_bar' is set to the Bar object at index 'bar' minus one. The variable 'beat' stores the position the note should be placed at in the format used in Bar objects. 'space_left' holds the note space left from the position in 'beat' to the end of the bar. Again, if 'space_left' is less than 'length', the note duration is split between to bars and if not, the note is placed at the specified beat.

```
#Removes a specific note from the score and replaces it with a rest
def delete_note(composition, instrument_track, bar, position):
    t = select_track(instrument_track, composition)
    chosen_bar = t[bar - 1]
    beat = (position - 1) / (chosen_bar.length * 4)
    temp = chosen_bar.current_beat
    chosen_bar.current_beat = beat
    #Finding the note at the specified position
    for n in chosen_bar:
        if n[0] == beat:
            duration = n[1]
            chosen_bar.place_rest(duration)
```

In the delete_note procedure, the select_track function is called, and the result is stored in the variable 't'. 'chosen_bar' is set to the Bar object at the index 'bar' minus one and 'beat' holds the position in a better format. Then a for loop is used to iterate through the Note objects in 'chosen_bar'. If the position of the Note object in the bar – the first element of the Note – is equal to 'beat', 'duration' is set to the length of the note – the second element of the Note – and a rest is placed at that position – replacing the note that was there before.

In the delete_track procedure, the select_track function is called, and the result is stored in the variable 'track'. Then the remove list method is used delete the Track object stored in 'track' from the tracks attribute of the argument 'composition'.

```
#Removes a track from the score
def delete_track(composition, instrument_track):
    track = select_track(instrument_track, composition)
    composition.tracks.remove(track)
```

Setting Up the Edit Webpage

```
urlpatterns = [
    # e.g. /monica34/editor/3
    path('<str:username>/editor/<int:score_id>/', views.open_editor, name = 'open_editor'),
]
```

To create the url path that when typed in at the end of the web address directs the user to correct page in Django you define a url pattern using the path function. The first parameter of the function is the string representing the route of the url; for the edit webpage, after the base domain the path is: the username of the user opening the score, then the word editor and then the scoreID of the score being opened. The username and score_id parts of the route pass the data put there as parameters 'username' and 'score_id' to the open_editor view.

```
def open_editor(request, username, score_id = None):
    # Get User record
    user = get_object_or_404(User, pk = username)
    # Retrieve existing score record
    if score_id != None:
        score = get_object_or_404(Score, pk = score_id)
        composition = remake_composition(score)
    # Or create new score record
    else:
        score = new_score(title, user)
        composition = new_composition(score)
        update_score(score, composition)
        save_comp(score, composition)
```

1

A Django view is a type of webpage that generally serves a specific function and has a specific template. Each view is represented by a function. I used the open_editor view as my open_score module I described in the design section. Using the argument 'username', passed from the url, the object representing the user's record is stored in the variable 'user'. If a new score is being created, no value for 'score_id' will be passed so it will have the default value None. If 'score_id' isn't equal to None, the object representing the score's record is stored in the variable 'score'. The remake_composition function is called and

the Composition object returned is stored in the variable ‘composition’. Else, the function new_score is called and the Score object returned is stored in ‘score’ then the function new_composition is called, and the Composition object returned is stored in ‘composition’. The update_score and save_comp functions are then called to make none of the score files (e.g. lilypond, PDF etc.) are empty.

```
    # Option lists needed
    instrument_list = [
    "Acoustic Grand Piano", "Harpsichord", "Glockenspiel", "Marimba", "Church Organ",
    "Accordion", "Harmonica", "Electric Guitar", "Acoustic Bass", 'Electric Bass',
    'Violin', 'Viola', 'Cello', 'Contrabass', 'Orchestral Harp', 'Timpani', 'Trumpet',
    'Trombone', 'Tuba', 'Soprano Sax', 'Oboe', 'Bassoon', 'Clarinet', 'Piccolo', 'Flute'
    ]
    metre_list = [
    '4/4', '3/4', '2/4', '5/4', '2/2', '3/2', '3/8', '6/8', '9/8', '12/8'
    ]
    key_list = [
    'C major', 'D major', 'E major', 'F major', 'G major', 'A major', 'B major',
    'F# major', 'C# major', 'Cb major', 'Gb major', 'Db major', 'Ab major', 'Eb major',
    'Bb major', 'A minor', 'B minor', 'C minor', 'D minor', 'E minor', 'F minor', 'G minor',
    'F# minor', 'C# minor', 'G# minor', 'D# minor', 'A# minor', 'Ab minor', 'Eb minor', 'Bb minor'
    ]
```

2

```
    note_list = [
    'C', 'C#', 'D', 'Db', 'D#', 'E', 'Eb', 'F', 'F#', 'G', 'Gb', 'G#', 'A', 'Ab', 'A#', 'B', 'Bb'
    ]
    duration_list = [
    'semibreve', 'minim', 'crotchet', 'quaver', 'semiquaver', 'demisemiquaver', 'triplet crotchet',
    'triplet quaver', 'triplet semiquaver', 'dotted semibreve', 'dotted minim', 'dotted crotchet',
    'dotted quaver', 'dotted semiquaver', 'dotted demisemiquaver'
    ]
    return render(request, 'editor/edit.html', {'score': score, 'composition': composition,
    'user': user, 'instrument_list' : instrument_list, 'metre_list': metre_list, 'key_list': key_list,
    'note_list': note_list, 'duration_list': duration_list})
```

3

‘instrument_list’, ‘metre_list’, ‘key_list’, ‘note_list’ and ‘duration_list’ are all variables containing the lists of input options needed in the forms on the edit webpage. render is a Django shortcut function that combines a given template (html file) with a given context dictionary and returns a HttpResponse object with that rendered text. HttpResponse objects are responsible loading webpages. The template used for this view is the edit.html file found in the editor/templates/editor directory and the context dictionary used stores the variables ‘score’, ‘composition’, ‘user’, ‘instrument_list’, ‘metre_list’, ‘key_list’, ‘note_list’ and ‘duration_list’. The contest dictionary allows these variables to accessed in edit.html template.

```
from django.shortcuts import render, get_object_or_404
from accounts.models import User
from libraries.models import Score
from .Python_Modules.BuildScore import new_score, new_composition, remake_composition
```

Import statements at the beginning of the editor/views.py file.

```
{% load static %}
{% get_static_prefix as STATIC_PREFIX %}
{% load editorFunctions %}
<!DOCTYPE html>
<html lang="en" dir="ltr">
  <head>
    <meta charset="utf-8">
    <link rel="stylesheet" href="{% static 'editor/main_style_sheet.css' %}">
    <title>{{ score.title }} - editing...</title>
    <nav class="Navigation bar">
      <a href="#">HOME</a> |
      <a href="#">PUBLIC LIBRARY</a> |
      <a href="#">MY LIBRARY</a>
    </nav>
  </head>
```

Django templates are html files which can incorporate Django's template language; this language allows the elements inside the context dictionary to be accessed as variables and python-like functions to be used within the html code. Template variables are have double curly brackets around them, e.g. {{ var }}, and template functions have curly brackets and percentage marks around them, e.g. {% function %}.

The editor/edit.html file is what defines the general layout of the 'editor' webpage. It is linked to the stylesheet editor/main_style_sheet.css. Using the <title> tag, I set the name of the tab to be the name of the score opened followed by "- editing…". Then using the <nav> tag, I created the basic skeleton of the navigation bar with links to the home webpage, public library webpage and user library webpage.

I created the division with the class 'Top-bar-on-editor' to group all the elements that are used to create the top bar on the editor webpage. In this top bar, the name of the score is shown , there is a button which has the same 'save', another button with the name 'close' which acts a link and an audio plugin that is used to embed the score's midi file into the webpage – attribute 'controls' is used to specify that the play/pause controls should be displayed also. The template variable 'STATIC_PREFIX' stores the static files path which is the location all files that need to be accessed by the webpage should be stored.

```
<body>
  <div class="Top-bar-on-editor">
    <em><strong>{{ score.title }}</strong></em>  |
    <button type="button" name="Save" value="Save" onclick="">Save</button>
    <a href="#"><input type="button" name="Close" value="Close"></a>
    <audio controls>
      <source src="{{ STATIC_PREFIX }}libraries/{{ score.music_file }}" type="audio/midi">
      Your browser does not support the audio element.
    </audio>
  </div>
```

```
@register.simple_tag(takes_context  = True)
#Adds instrument track to composition object holding score
def add_instrument(context, instrument_name):
    composition = context['composition']
    instrument = select_instrument(instrument_name)
    new_track = Track(instrument)
    if instrument != None:
        new_track.name = instrument.name
        #Dealing with duplicate instrument names
        n = 1
        for track in composition:
            if track.instrument.name == new_track.name:
                n += 1
            if n > 1:
                new_track.name += ' (' + str(n) + ')'
        composition + new_track
        context['composition'] = composition
```

To use the subroutines, I defined in BuildScore.py with the edit.html file I converted then into template tags (Django template functions) and saved these in a new file called editorFunctions.py in the editor/templatetages directory. In each "tag", the first parameter passed must be 'context' which is the context dictionary. I then amended the subroutines so the elements in the context library where being modified directly. This is an example of one the template tags.

```
<div class="dropdown">
  <button name="Add Instrument" class="dropdownbtn" onclick="dropfunction()">Add Instrument</button>
  <div id="instrument-dropdown" class="dropdown-content">
  <div class="vertical-menu" id="instrument-menu">
    {% for i in instrument_list %}
      <button type="button" name="{{ i }}" onclick="{% add_instrument i %}">{{ i }}</button>
    {% endfor %}
  </div>
  </div>
</div>
```

The first menu I created was the one for adding instruments to the score. I created a division with the class ‘dropdown’ and within it there is a button named ‘Add Instrument’ with the class ‘dropdownbtn’. When the button is clicked, the JavaScript function ‘dropfunction’ is called. Then there is another division with the id ‘instrument-dropdown’ and the class ‘dropdown-content’. This division holds the elements that are to be hidden until the ‘Add Instrument’ button is clicked, then they are shown. In this division there is another division with the class ‘vertical-menu’ and the id ‘instrument-menu’. In the ‘instrument-menu’ division the for template tag is used to iterate through each string in the element in the context with the key ‘instrument_list’ – this is the list of instrument names from the open_editor view. For each instrument name, there is a button that when clicked should call the add_instrument tag and pass that instrument name (held in the variable ’i’) into the tag.

```
/* Styling for menus on score editor page */
.vertical-menu {
  width: 200px;
  height: 150px;
  overflow-y: auto;
}
.vertical-menu button {
  background-color: #eee;
  color: black;
  display: block;
  padding: 12px;
}
.vertical-menu button:hover {
  background-color: #ccc;
}

/* Dropdown button */
.dropdownbtn {
  background-color: #FF7F50;
  color: white;
  padding: 16px;
  font-size: 16px;
  border: none;
  cursor: pointer;
}
```

```
.dropdownbtn:hover, .dropdownbtn:focus {
  background-color: #E9967A;
}
.dropdown {
  position: relative;
  display: inline-block;
}
.dropdown-content {
  display: none;
  position: absolute;
  background-color: #f1f1f1;
  min-width: 160px;
  box-shadow: 0px 8px 16px rgba(0, 0, 0, 0.2);
  z-index: 1
}
.show {
  display: block;
}
```

In the file main_style_sheet.css, I wrote some basic styling for the vertical dropdown menus. Other than setting the colour of the buttons and backgrounds, this styling restricts the size of the dropdown menu and allows an overflow so the user can scroll through the menu content. It also changes the colour of a button when the cursor is over it and hides the elements in the dropdown-content division.

```
<!-- Scripts -->
<script type="text/javascript">
  function dropfunction(iden) {
    document.getElementById(iden).classList.toggle("show");
  }
</script>
```

Using the <script> tag, I defined a JavaScript function in the file edit.html. the parameter 'iden' passed into the function should be the id of the dropdown-content to be shown when the button is clicked. The element with an id equal to the argument 'iden' is accessed and its class name is toggled between 'show' and 'dropdown-content'. This causes the menu to switch between being shown and hidden as in the css file the display attribute of 'show' is block (so it is shown) and the display attribute of 'dropdown-content' is none (so it is hidden).

Fixing Edit Webpage using New Views

When casually testing out the buttons in the 'Add Instrument' menu I realised that the add_instrument procedure wasn't being carried out when I clicked the buttons so I decided to use a different method to implement the functions in editorFunctions.py. I created a new file, editorFunctionsNONTAG.py which held the functions in the form they were before I converted them into template tags.

```
urlpatterns = [
    # e.g. /monica34/editor/3
    path('<str:username>/editor/<int:score_id>/', views.open_editor, name = 'open_editor'),
    # e.g. /monica34/editor/3/i
    path('<str:username>/editor/<int:score_id>/<str:action>', views.edit, name = 'edit'),
]
```

I decided to revert to my original idea of using forms. To process the data input into forms with Django, views must be used. In the url.py file in the editor directory, I added another url pattern which has a route very similar to the first url pattern but has an extra part on the end which passes the parameter 'action' to the view edit.

```
<div class="Editor input menus">
  <div class="dropdown">
    <button type="button" name="Add Instrument" class="dropdownbtn" onclick="dropfunction('instrument-dropdown')">Add Instrument</button>
    <div id="instrument-dropdown" class="dropdown-content">
      <form id="instrument-menu" class="editor-menu" action="{% url 'edit' user.username score.scoreID 'i' %}" method="post">
        {% csrf_token %}
        <select name="instrument" size="4" required>
          {% for i in instrument_list %}
            <option value="{{ i }}">{{ i }}</option>
          {% endfor %}
        </select>
        <input type="submit" name="Choose" value="Choose">
      </form>
    </div>
  </div>
```

To change the 'Add Instrument' menu, I replaced the elements in the instrument-dropdown division with a form element. This form has the id 'instrument-menu' and the class 'editor-menu' and the post method is chosen instead of the get method as submitting this form will alter data on the server side. The action attribute is set to {% url 'edit' user.username score.scoreID 'I' %}; the template tag url will select the url pattern with the name "edit", and then pass the 3 following terms as parameters to the corresponding view function to carry out the form processing. Within the form element the template tag csrf_token is used to protect the form data against Cross Site Request Forgeries. The <select> tag is used to create limited list of instrument names the user can chose from – this acts as lookup validation. The form includes a submit button. The code for the other dropdown menus is very similar, it's only the form elements and last parameter passed to the url template tag that vary.

```
<div class="dropdown">
  <button type="button" name="button" class="dropdownbtn" onclick="dropfunction('metre-dropdown')">Choose Metre</button>
  <div id="metre-dropdown" class="dropdown-content">
      <form id="metre-menu" class="editor-menu" action="{% url 'edit' user.username score.scoreID 'm' %}" method="post">
          {% csrf_token %}
          <label for="bar">Bar number</label>
          <select name="bar" size="3" required>
            {% for b in composition.0.bars %}
            <option value="{{ forloop.counter }}">{{ forloop.counter }}</option>
            {% endfor %}
          </select>
          <label for="metre">Metre</label>
          <select name="metre" size="3" required>
            {% for m in metre_list %}
              <option value="{{ m }}">{{ m }}</option>
            {% endfor %}
          </select>
          <input type="submit" name="Choose" value="Choose">
      </form>
  </div>
</div>
```

```
<div class="dropdown">
  <button type="button" name="button" class="dropdownbtn" onclick="dropfunction('key-dropdown')">Choose Key Signiture</button>
  <div id="key-dropdown" class="dropdown-content">
    <form id="key-menu" class="editor-menu" action="{% url 'edit' user.username score.scoreID 'k' %}" method="post">
      {% csrf_token %}
      <label for="bar">Bar number</label>
      <select name="bar" size="3" required>
        {% for b in composition.0.bars %}
        <option value="{{ forloop.counter }}">{{ forloop.counter }}</option>
        {% endfor %}
      </select>
      <label for="key">Key</label>
      <select name="key" size="3" required>
        {% for k in key_list %}
          <option value="{{ k }}">{{ k }}</option>
        {% endfor %}
      </select>
      <input type="submit" name="Choose" value="Choose">
    </form>
  </div>
</div>
```

```
<div class="dropdown">
  <button type="button" name="button" class="dropdownbtn" onclick="dropfunction('note-dropdown')">Add Note</button>
  <div id="note-dropdown" class="dropdown-content">
    <form id="note-menu" class="editor-menu" action="{% url 'edit' user.username score.scoreID 'n' %}" method="post">
      {% csrf_token %}
      <label for="track">Instrument</label>
      <select name="track" size="3" required>
        {% for t in composition %}
          <option value="{{ t.instrument.name }}">{{ t.instrument.name }}</option>
        {% endfor %}
      </select>
```

1

```
<label for="note">Note</label>
<select name="note" size="3" required>
  {% for n in note_list %}
    <option value="{{ n }}">{{ n }}</option>
  {% endfor %}
</select>
<label for="octave">Octave</label>
<select name="octave" size="3" required>
  <option value="0">0</option>
  <option value="1">1</option>
  <option value="2">2</option>
  <option value="3">3</option>
  <option value="4">4 (middle octave)</option>
  <option value="5">5</option>
  <option value="6">6</option>
  <option value="7">7</option>
</select>
```

2

```
      <label for="duration">Note Duration</label>
      <select name="duration" size="3" required>
        {% for d in duration_list %}
          <option value="{{ d }}">{{ d }}</option>
        {% endfor %}
      </select>
      <label for="bar">Bar number</label>
      <select name="bar" size="3" required>
        {% for b in composition.0.bars %}
        <option value="{{ forloop.counter }}">{{ forloop.counter }}</option>
        {% endfor %}
      </select>
      <label for="position">Bar beat</label>
      <input type="number" name="position" min="1" max="7" required>
      <input type="submit" name="Choose" value="Choose">
    </form>
  </div>
</div>
```

3

```
<div class="dropdown">
  <button type="button" name="button" class="dropdownbtn" onclick="dropfunction('rest-dropdown')">Add Rest</button>
  <div id="rest-dropdown" class="dropdown-content">
    <form id="rest-menu" class="editor-menu" action="{% url 'edit' user.username score.scoreID 'r' %}" method="post">
      {% csrf_token %}
      <label for="track">Instrument</label>
      <select name="track" size="3" required>
        {% for t in composition %}
          <option value="{{ t.instrument.name }}">{{ t.instrument.name }}</option>
        {% endfor %}
      </select>
      <label for="duration">Note Duration</label>
```

1

```
      <select name="duration" size="3" required>
        {% for d in duration_list %}
          <option value="{{ d }}">{{ d }}</option>
        {% endfor %}
      </select>
      <label for="bar">Bar number</label>
      <select name="bar" size="3" required>
        {% for b in composition.0.bars %}
        <option value="{{ forloop.counter }}">{{ forloop.counter }}</option>
        {% endfor %}
      </select>
      <label for="position">Bar beat</label>
      <input type="number" name="position" min="1" max="7" required>
      <input type="submit" name="Choose" value="Choose">
    </form>
  </div>
</div>
```

2

```
<div class="dropdown">
  <button type="button" name="button" class="dropdownbtn" onclick="dropfunction('delete-note-dropdown')">Delete Note</button>
  <div id="delete-note-dropdown" class="dropdown-content">
    <form id="delete-note-menu" class="editor-menu" action="{% url 'edit' user.username score.scoreID 'dn' %}" method="post">
      {% csrf_token %}
      <label for="track">Instrument</label>
      <select name="track" size="3" required>
        {% for t in composition %}
          <option value="{{ t.instrument.name }}">{{ t.instrument.name }}</option>
        {% endfor %}
      </select>
      <label for="bar">Bar number</label>
      <select name="bar" size="3" required>
        {% for b in composition.0.bars %}
        <option value="{{ forloop.counter }}">{{ forloop.counter }}</option>
        {% endfor %}
      </select>
      <label for="position">Bar beat</label>
      <input type="number" name="position" value="1" min="1" max="7" required>
      <input type="submit" name="Choose" value="Choose">
    </form>
  </div>
</div>
```

```
<div class="dropdown">
  <button type="button" name="button" class="dropdownbtn" onclick="dropfunction('delete-instrument-dropdown')">Delete Instrument</button>
  <div id="delete-instrument-dropdown" class="dropdown-content">
    <form id="delete-instrument-menu" class="editor-menu" action="{% url 'edit' user.username score.scoreID 'di' %}" method="post">
      {% csrf_token %}
      <label for="track">Instrument</label>
      <select name="track" size="3" required>
        {% for t in composition %}
          <option value="{{ t.instrument.name }}">{{ t.instrument.name }}</option>
        {% endfor %}
      </select>
      <input type="submit" name="Choose" value="Choose">
    </form>
  </div>
</div>
```

```
def edit(request, username, score_id, action):
    user = get_object_or_404(User, pk = username)
    score = get_object_or_404(Score, pk = score_id)
    composition = remake_composition(score)
    if action == 'i':
        selected_instrument = request.POST['instrument']
        add_instrument(composition, selected_instrument)
    elif action == 'm':
        selected_metre = request.POST['metre']
        selected_metre = selected_metre.split('/')
        metre = (int(selected_metre[0]), int(selected_metre[1]))
        selected_bar = int(request.POST['bar'])
        set_bar_metre(composition, metre, selected_bar)
    elif action == 'k':
        selected_key = request.POST['key']
        selected_key = selected_key.split(' ')
        if selected_key[1] == 'major':
            key = selected_key[0]
        elif selected_key[1] == 'minor':
            key = selected_key[0].lower()
        selected_bar = int(request.POST['bar'])
        set_bar_key(composition, key, selected_bar)
```

1

I defined the edit view is views.py in the editor directory. Like the open_editor view, the 'username' argument is used to retrieve the object representing the user's record and the 'score_id' argument is used to retrieve the object representing the score record. If action equals "i", data was sent from the instrument menu form. The variable 'selected_instrument' stores the data from the field in the form with the name attribute "instrument". The add_instrument procedure is then called, passing in the parameters composition and selected_instrument. If action equals "m", data was sent from the metre menu form. 'selected_metre' stores the data from the metre field. Post data is always returned as a string so 'selected_metre' is split into a list by the delimiter '/' and put inside the same variable. 'metre' is then set to a tuple containing the integer version of the 2 strings in the list. This puts the metre chosen in

```
    elif action == 'n':
        selected_track = request.POST['track']
        selected_note = request.POST['note']
        selected_octave = int(request.POST['octave'])
        selected_duration = request.POST['duration']
        selected_bar = int(request.POST['bar'])
        selected_position = int(request.POST['position'])
        add_note(composition, selected_duration, selected_track,
        selected_note, selected_octave, selected_bar, selected_position)
    elif action == 'r':
        selected_track = request.POST['track']
        selected_duration = request.POST['duration']
        selected_bar = int(request.POST['bar'])
        selected_position = int(request.POST['position'])
        add_note(composition, selected_duration, selected_track,
        bar = selected_bar, position = selected_position)
```

2

```
elif action == 'dn':
    selected_track = request.POST['track']
    selected_bar = int(request.POST['bar'])
    selected_position = int(request.POST['position'])
    delete_note(composition, selected_track, selected_bar,
    selected_position)
elif action == 'di':
    selected_track = request.POST['track']
    delete_track(composition, selected_track)
update_score(score, composition)
save_comp(score, composition)
```

3

the correct format. ‘selected_bar’ holds the integer version of the data from the bar field. Then the set_bar_metre function is called, passing in ‘composition’, ‘metre’ and ‘selected_bar’. If action equals “k”, data was sent from the key menu form. ‘selected_key’ stores the data from the key field. The string in ‘selected_key’ is split by the delimiter ‘ ’ to make a list where the first element is the letters of the key (e.g. “Bb”) and the other is either “major” or “minor”. If the second element of the list is “major”, ‘key’ stores the first element of the list and if the second element is “minor”, ‘key’ stores the lower case version of the first element of the list. This is done because the mingus library uses lower case strings for minor keys and upper-case strings for major keys. ‘selected_bar’ stores the integer version of the data from the bar field. Then the set_bar_key procedure is called, passing in ‘composition’, ‘key’ and ‘selected_bar’. If action equals “n”, data was sent from the note menu form. ‘selected_track’ holds the data from the track field, ‘selected_note’ holds the data from the note field, ‘selected_octave’ holds the integer version of the data field, ‘selected_duration’ hold the data from the duration field, ‘selected_bar’ holds the integer version of the data from the bar field and ‘selected_position’ holds the integer version of the data from the position field. Then the add_note procedure is called, passing in ‘composition’, ‘selected_duration’, ‘selected_track’, ‘selected_note’, ‘selected_octave’, ‘selected_bar’ and ‘selected_position’. If action equals “r”, data was sent from the rest menu form. ‘selected_track’ stores the data from the track field, ‘selected_duration’ stores the data from the duration field, ‘selected_bar’ stores the integer version of the data from the bar field and ‘selected_position’ stores the integer version of the data from the position field. Then the add_note procedure is called, passing in ‘composition’, ‘selected_duration’, ‘selected_track’, ‘selected_bar’ and ‘selected_position’ (missing arguments for ‘note’ and ‘octave’). If action equals “dn”, data was sent from the delete note menu form. ‘selected_track’ holds the data from the track field, ‘selected_bar’ holds the integer version of the data from the bar field and ‘selected_position’ holds the integer version of the data. Then the delete_track procedure is called, passing in ‘composition’, ‘selected_track’, ‘selected_bar’ and ‘selected_position’. If action equals “di”, data was sent from the delete instrument menu form. ‘selected_track’ holds the data from the track field. Then the delete_track procedure is called, passing in ‘composition’ and ‘selected_track’. After the correct procedure has be executed the procedures update_score and save_comp are called to save the change made to score.

```
    return render(request, 'editor/edit.html', {'score': score, 'composition': composition,
    'user': user, 'instrument_list' : instrument_list, 'metre_list': metre_list,
    'key_list': key_list, 'note_list': note_list, 'duration_list': duration_list})
```

The list variables (‘instrument_list’ etc.) from the open_editor view are also declared in the edit view. Then the render function is used to create a HttpResponse using the same edit.html template so, to the user is essentially returning to the same editor webpage.

Amending Subroutines

When examining the text file of a Score record, I created to quickly test my code, I realised the notes within bars weren't being saved properly. This was because the line "new_bar.bar = list(b_list[3])" in remake_composition broke up the string in a weird way creating the list wrongly.

```
def save_comp(score, composition):
    # Opens text file that is going to hold the information to save compos
    # Each track is separated by a '?'. Instrument name and bars are separ
    file = score.composition_file
    file.open('w')
    comp_string = ''
    #Iterating through each track
    for t in composition.tracks:
        #Saving the track name (instrument name)
        comp_string += t.instrument.name + ';'
        for b in t.bars:
            #Saving the key, meter and bar list of each Bar in the track
            comp_string += b.key.key + '~'
            comp_string += str(b.meter[0]) +'~'
            comp_string += str(b.meter[1]) + '~'
            b_list = [str(n) for n in b.bar]
            b_string = '+'.join(b_list)
            comp_string += b_string + ';'
        comp_string += '?'
    file.write(comp_string)
    file.close()
    score.save()
```

remake_composition

```
        if instrument != None:
            track = Track(instrument)
            #Adding bars to the track
            for b in t_list:
                if b != '' and b != None:
                    b_list = b.split('~')
                    new_bar = Bar(b_list[0], (int(b_list[1]), int(b_list[2])))
                    bar_string = b_list[3]
                    notes = bar_string.split('+')
                    for n in notes:
                        n = n.strip('[')
                        n = n.strip(']')
                        n = n.split(', ')
                        try:
                            d = float(n[1])
                            note = n[2]
                            note = note.strip('[')
                            note = note.strip("'")
                            note = note.split('-')
                            note = Note(note[0], int(note[1]))
                            new_bar.place_notes(note, d)
                        except IndexError:
                            continue
                    track + new_bar
            comp + track
        else:
            continue
    return comp
```

Firstly, I changed the save_comp procedure to change the way the bar objects are stored in the text file. After storing the key and metre of a bar, a list of the string representations of each note in the bar is created using a list comprehension and then stored in the variable 'b_list'. 'b_string' then stores a string made by concatenating all the strings in 'b_list' with "+" between them. 'b_string' is then concatenated with 'comp_string and ";".

I then changed the remake_composition procedure so that after the 'new_bar' variable is created the 4th element in 'b_list' is put in the variable 'bar_string' – this is the string made from the concatenated note strings. A list is then made of the note strings by splitting 'bar_string' by the delimiter '+' and is stored in 'notes'. A for loop is then used to iterate through each note string. An example of a note string is "[0.25, 4, [B-4]]". The outer square brackets are striped from the string and it is then split by the delimiter ', ' to create a list. A try-except statement is used deal with a case where the bar was empty (contained no note objects). If this the case the first value passed into 'n' would be an empty string so when n is split into a list there would be no element at index 1, causing an IndexError. The try-execpt statement will catch this and skip that iteration of the for loop. As the bar is empty there wouldn't be another element in 'notes' so the loop ends. Then the function continues the same as before.

Second Cycle

Creating Home Webpage

urls.py file in editor directory

```
urlpatterns = [
    # e.g. /monica34/editor/3
    path('<str:username>/editor/<int:score_id>/', views.open_editor, name = 'open_editor'),
    # e.g. /monica34/editor/3/i
    path('<str:username>/editor/<int:score_id>/<str:action>', views.edit, name = 'edit'),
    path('', views.home, 'home'),
]
```

urls.py file in AlevelProject directory

```
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('editor.urls')),
    path('accounts/', include('accounts.urls')),
    path('libraries/', include('libraries.urls'))
]
```

I created the home webpage in the editor app because I wanted the URL route to be empty, so home webpage is what users first meet when they type the domain name. In the main url patterns defined in the AlevelProject directory make the webpages in the accounts app have routes that must start with 'accounts/' and webpages in the libraries app have routes that must start with 'libraries/'.

The home view is very basic. It just returns the HttpResponse object that will load the home webpage.

```
def home(request):
    return render(request, 'editor/home.html')
```

```
<!DOCTYPE html>
<html lang="en" dir="ltr">
  <head>
    <meta charset="utf-8">
    <title>ScoreLily - Home</title>
    <nav class="navigation-bar">
      <a href="{% url 'home' %}">HOME</a> |
      <a href="{% url 'login' %}">PUBLIC LIBRARY</a> |
      <a href="{% url 'login' %}">MY LIBRARY</a>
    </nav>
  </head>
  <body>
    <div id="home-title">
      ScoreLily
    </div>
    <div class="text">
      ScoreLily is platform the create and share music scores.
    </div>
    <div id="account-links">
      <a href="{% url 'create_account' %}">
        <button type="button" name="create-account">Create Account</button>
      </a>
      <a href="{% url 'login' %}">Or login</a>
    </div>
  </body>
</html>
```

On the home webpage, in the navigation bar, 'HOME' links back to the home page as expected but 'PUBLIC LIBRARY' and 'MY LIBRARY' link to the login page to prevent a person who hasn't logged in from accessing those pages.

After some text, there is a button that's acts as a link to the create account page. There is also a text link to the login page.

Coding Subroutines for User Accounts Module

```
def check_details(username, password):
    found = False
    records = User.objects.all()
    for account in records:
        if account.username == username and account.password == password:
            found = True
    return found
```

In the check_details function, the flag variable 'found' is set to False and a list of all the User objects representing the records in the data is stored in the variable 'records'. A for loop is used to iterate through each User object and check if the username attribute is equal to the argument 'username' and if the password attribute is equal to the argument 'password'. If this is true, found is set to True and will not be changed in subsequent iterations. The variable 'found' is then returned.

```
def validate_details(email, password):
    valid_password = False
    valid_email = False
    special_characters = [
    '@', '!', '£', '%', '^', '&',
    '*', '(', ')', '-', '_', '=',
    '+', '?', '[', ']', '{', '}'
    ]
    lower = 0
    upper = 0
    digit = 0
    special = 0
    for char in password:
        if char in special_characters:
            special += 1
        elif char.isalpha():
            if char.islower():
                lower += 1
            elif char.isupper():
                upper += 1
        elif char.isdigit():
            digit += 1
```

1

```
    if special > 0 and lower > 0 and upper > 0 and digit > 0 and len(password) >= 8:
        valid_password = True
    if '@' in email and '.' in email and email[-1] != '@' and email[-1] != '.':
        valid_email = True
    if valid_password and valid_email:
        return True
    else:
        return False
```

2

In the validate details function, the flag variables 'valid_password' and 'valid_email' are set to False. The list 'special_characters' stores the non-alphanumeric characters that are allowed in the password. 'lower', 'upper', 'digit' and 'special' are all counter variables. A for loop is used to iterate through each character in the string 'password'. If that character is an element of the 'special_characters' list, then the counter 'special' is incremented. If the character is in the alphabet, it is then checked if it is lowercase, if it is 'lower' is incremented and if the character is uppercase then 'upper' is incremented. If the character is a digit (0 – 9) then 'digit' is incremented. After the for loop is complete, if the values of 'special', 'lower', 'upper' and digit are all over 0 it shows the password contains at least one of each. If this is true along with the length of the password being at least 8, 'valid_password' is set to True. If the string in the argument 'email' contains a '@' and a '.'and doesn't end with a '.' Or '@', 'valid_email' is set to True. If both 'valid_password' and 'valid_email' are true, the value True is returned, if not False is returned.

```
def make_record(username, email, password):
    account = User(username = username, email = email, password = password)
    account.save()
    return account
```

In the make_record function, a new User object is created using the arguments 'username', 'password' and 'email' and stored in the variable 'account'. The variable is then used to save this information to a record in the database. And then 'account' is returned.

Creating Login Webpage

```
urlpatterns = [
    path('login/', views.login, name = 'login'),
    path('create/', views.create_account, name = 'create_account')
]
```

In urls.py in the accounts directory, I created 2 url patterns: one for the login webpage and the other for the create account webpage. Both routes are just basic strings that don't pass any parameters to their respective view functions.

```
def login(request):
    try:
        username = request.POST['username']
        password = request.POST['password']
        if check_details(username, password):
            return redirect('user_library', username = username)
        else:
            message = "The details you have entered are incorrect. Please try again."
            return render(request, 'accounts/login.html', {'message': message})
    except KeyError:
        return render(request, 'accounts/login.html', {'message' : 'None'})
```

In the login view function, a try-except statement is used to separate the cases the view is being used to handle a form or just to load the webpage. If its being used to just load the webpage the line username = request.POST['username'] would cause a KeyError so the rest of that sequence is skipped and the function render is used to load the webpage with the context variable 'message' storing the string 'None'. If the line doesn't fail, the check_details function is run and if that returns True, the redirect function is used to take run the user_library view to open that webpage. If check_details returns False, the variable 'message is set to a string and render is used to load the webpage with the context variable 'message' storing the normal variable 'message'.

```
<!DOCTYPE html>
<html lang="en" dir="ltr">
  <head>
    <meta charset="utf-8">
    <title>ScoreLily - Login</title>
    <nav class="navigation-bar">
      <a href="{% url 'home' %}">HOME</a> |
      <a href="{% url 'login' %}">PUBLIC LIBRARY</a> |
      <a href="{% url 'login' %}">MY LIBRARY</a>
    </nav>
  </head>
  <body>
      <div class="page-title">
        Welcome Back
      </div>
```

On the login page, the navigation bar is the same as on the home page.

```
        <form id="login-form" class="accounts-form" action="{% url 'login' %}" method="post">
          {% csrf_token %}
          <label for="username">Username:</label>
          <input type="text" name="username">
          <label for="password">Password:</label>
          <input type="password" name="password">
          <input type="submit" name="login" value="Login">
        </form>
        <div id="failed-login-message" class="error-message">
          {% if message != "None" %}
            {{ message }}
          {% endif %}
        </div>
        <div id="create-account-link">
          Don't have an account?<br>
          <a href="{% url 'create_account' %}">Create One</a>
        </div>
    </body>
```

The form element in login.html has the action attribute set to {% url 'login' %} so the from processing is handled by the login view. The input element for the username field has the text type whereas the input element for password field has the password type so the text is hidden as the user is entering their password. Underneath the form, if the value of the context variable 'message' is not the string 'None' then the contents of 'message' will be shown – this happens when the login attempt is unsuccessful. Under this is a text link to the create account page.

Creating Create Account Webpage

```
def create_account(request):
    try:
        username = request.POST['username']
        email = request.POST['email']
        password1 = request.POST['password1']
        password2 = request.POST['password2']
        if password1 == password2:
            if validate_details(email, password1):
                user = make_record(username, email, password1)
                return redirect('user_library', username = user.username)
            else:
                message = """Invaild email or password entered.
                Password must contain at least one lowercase letter, uppercase letter, digit and special character.
                Special characters that are allowed are: @, !, £, %, ^, &, *, (, ), -, _, =, +, ?, [, ], { and }."""
                return render(request, 'accounts/create_account.html', {'message': message})
        else:
            message = "The passwords entered do not match."
            return render(request, 'accounts/create_account.html', {'message': message})
    except KeyError:
        return render(request, 'account/create_account.html', {'message': 'None'})
```

The create_account view is very similar to the login view in that it uses a try-except statement to control the journey through the program. If the sequence in the try clause fails due to a KeyError, the render function is just used to load the webpage. If the first line doesn't fail the 2 password variables are compared for equality, if they

are the validate_details function is called. If that returns True, the make_record function is called, and the value is returns is stored in the variable user. The redirect function is then used to call the user_library view and load the user library page. If the validate_details function returns False, the variable 'message' is set to a string and the render function is used to load the create account page while passing in 'message'. If the passwords weren't equal, 'message' is set to a different string and create account page is loaded using render.

```
<html lang="en" dir="ltr">
  <head>
    <meta charset="utf-8">
    <title>ScoreLily - Create Account</title>
    <nav class="navigation-bar">
      <a href="{% url 'home' %}">HOME</a> |
      <a href="{% url 'login' %}">PUBLIC LIBRARY</a> |
      <a href="{% url 'login' %}">MY LIBRARY</a>
    </nav>
  </head>
  <body>
    <div class="page-title">
      Please Create An Account
    </div>
```

On the create account page, the navigation bar is the same as on the home and login pages.

```
    <form id="create-account-form" class="accounts-form" action="{% url 'create_account' %}" method="post">
      {% csrf_token %}
      <label for="username">Username:</label>
      <input type="text" name="username">
      <label for="email">Email:</label>
      <input type="email" name="email">
      <label for="password1">Password:</label>
      <input type="password" name="password1">
      <label for="password2">Re-enter Password:</label>
      <input type="password" name="password2">
      <input type="submit" name="create" value="Create">
    </form>
    <div id="invalid-details-message" class="error-message">
      {% if message != "None" %}
        {{ message }}
      {% endif %}
    </div>
    <div id="login-link">
      If you already have an account: <a href="{% url 'login' %}">Login</a>
    </div>
  </body>
```

The form element in create_account.html has the action attribute set to {% url 'create_account' %} so the form processing is handled by the create_account view. There are two password fields (password1 and password2) to allow for double entry vertification of the password the user wants to enter. This is very important as since the input elements have the password type, the user won't see that they have typed so can easily make a mistake. The type attribute of the input element for the email field is email so some validation can be done on the client side when the form is submitted. Like in login.html, an if template tag is used to decide whether an error message should be shown. And a text link to the login page is underneath.

Coding Subroutines for Libraries Module

```
def title_sort(lst):
    if len(lst) > 1:
        mid = len(lst) // 2
        left_half = lst[:mid]
        right_half = lst[mid:]
        title_sort(left_half)
        title_sort(right_half)
        i, j, k = 0, 0, 0
        while i < len(left_half) and j < len(right_half):
            if left_half[i].title < right_half[j].title:
                lst[k] = left_half[i]
                i += 1
            else:
                lst[k] = right_half[j]
                j += 1
            k += 1
        while i < len(left_half):
            lst[k] = left_half[i]
            i += 1
            k += 1
        while j < len(right_half):
            lst[k] = right_half[j]
            j += 1
            k += 1
```

```
def edited_date_sort(lst):
    if len(lst) > 1:
        mid = len(lst) // 2
        left_half = lst[:mid]
        right_half = lst[mid:]
        edited_date_sort(left_half)
        edited_date_sort(right_half)
        i, j, k = 0, 0, 0
        while i < len(left_half) and j < len(right_half):
            if left_half[i].last_modified > right_half[j].last_modified:
                lst[k] = left_half[i]
                i += 1
            else:
                lst[k] = right_half[j]
                j += 1
            k += 1
        while i < len(left_half):
            lst[k] = left_half[i]
            i += 1
            k += 1
        while j < len(right_half):
            lst[k] = right_half[j]
            j += 1
            k += 1
```

```
def created_date_sort(lst):
    if len(lst) > 1:
        mid = len(lst) // 2
        left_half = lst[:mid]
        right_half = lst[mid:]
        created_date_sort(left_half)
        created_date_sort(right_half)
        i, j, k = 0, 0, 0
        while i < len(left_half) and j < len(right_half):
            if left_half[i].date_created > right_half[j].date_created:
                lst[k] = left_half[i]
                i += 1
            else:
                lst[k] = right_half[j]
                j += 1
            k += 1
        while i < len(left_half):
            lst[k] = left_half[i]
            i += 1
            k += 1
        while j < len(right_half):
            lst[k] = right_half[j]
            j += 1
            k += 1
```

```
def author_sort(lst):
    if len(lst) > 1:
        mid = len(lst) // 2
        left_half = lst[:mid]
        right_half = lst[mid:]
        author_sort(left_half)
        author_sort(right_half)
        i, j, k = 0, 0, 0
        while i < len(left_half) and j < len(right_half):
            if left_half[i].author < right_half[j].author:
                lst[k] = left_half[i]
                i += 1
            else:
                lst[k] = right_half[j]
                j += 1
            k += 1
        while i < len(left_half):
            lst[k] = left_half[i]
            i += 1
            k += 1
        while j < len(right_half):
            lst[k] = right_half[j]
            j += 1
            k += 1
```

‘title_sort’, ‘edited_date_sort’, ‘created_date_sort’ and ‘author_sort’ are all procedures that implement the merge sort algorithm they just used different things to determine the order. ‘title_sort’ and ‘author_sort’ put the list into ascending order whereas ‘created_date_sort’ and ‘edited_date_sort’ put the list into descending order. I’ll use title_sort to describe how all four procedures work. Firstly, the length of the variable ‘lst’ is checked to see if it is more than one (as a list containing one element is the base case of this recursive function), if this is true, the result of the integer division of the length of ‘lst’ by 2 is stored in ‘mid’. ‘mid’ is then used to slice ‘lst’ in half with the first half stored in ‘left_half’ and the second half in ‘right_half’. Then the procedure itself is called on ‘left_half’ then ‘right_half’. This recusion causes smaller and smaller section of the original ‘lst’ to split in half until there is only one element left, then the smaller lists start being merged. To do this, the counter variables ‘i’, ‘j’ and ‘k’ are all set to zero and a while loop is used to iterate through both ‘left_half’ and ‘right_half’ at different rates. If the selected ‘left_half’ element comes before the selected ‘right_half’ element alphatbetically, then the ‘left_half’ element is placed at the kth index of ‘lst’. If not, the ‘right_half’ element is placed at the kth index of ‘lst’. This continues until one of ‘left_half’ or ‘right_half’ finishes. Then the next two while loops are used to place the remaining elements of the unfinished list at the end of ‘lst’.

```
def sort(scores, criterion):
    if criterion == 'title':
        title_sort(scores)
    elif criterion == 'author':
        author_sort(scores)
    elif criterion == 'created date':
        created_date_sort(scores)
    elif criterion == 'last modified':
        edited_date_sort(scores)
```

The procedure sort simply uses an if-elif statement to select which sorting procedure should be used to sort the scores by comparing the value of the variable ‘criterion.

In the search function, the variable ‘filtered_scores’ is set to an empty list. A for loop is then used to iterate through every Score object that represents a record in the database. If the Score object selected’s title attribute contains the string ‘search_text’, it’s added to ‘filtered_scores’. If not, the author attribute is then checked if it contains ‘search_text’ and if so that Score object is added to ‘filtered_scores’. After the loop has ended, ‘filtered_scores’ is returned.

```
def search(search_text, scores):
    filtered_scores = []
    for s in scores:
        if search_text in s.title:
            filtered_scores.append(s)
        elif search_text in s.author:
            filtered_scores.append(s)
    return filtered_scores
```

```
def retrieve_scores(library, username = None):
    scores = []
    if library == 'public':
        for s in Score.objects.all():
            if s.public == True:
                scores.append(s)
    elif library == 'user':
        for s in Score.objects.all():
            if s.user.username == username:
                scores.append(s)
    return scores
```

In the retrieve_scores function, the variable ‘scores’ is set to an empty list. Then if value of the argument ‘library’ is “public”, a for loop is used to iterate through all the Score objects and if their public attribute is equal to True, they are added to the list ‘scores’. If the value of ‘library’ is “user”, a for loop is used to iterate through all the Score objects and if their user attribute’s username attribute is equal to the argument ‘username’, they are added to the ‘scores’. At the end, ‘scores’ is returned.

```
def make_public(score_id):
    score = Score.objects.get(pk = score_id)
    score.public = True
    score.save()
```

The make_public procedure, gets the Score object representing the database record with the primary key equal to the argument score_id and stores it in ‘score’. The public attribute of ‘score’ is then set to True and the score record is saved.

Creating User Library Webpage

Creating Public Library Webpage

Third Cycle

Evaluation