Song Recognition Using Audio Fingerprinting

DEPARTMENT OF ELECTRICAL AND ELECTRONIC ENGINEERING, BUET

Course No	**:**	EEE 312
Course Title	**:**	Digital Signal Processing 1
Submission Date:		02-09-2022

Submitted to

Dr. Mohammad Ariful Haque	Shafin-Bin-Hamid
Professor	Lecturer
Department of EEE	Department of EEE
BUET	BUET

Submitted by

Md. Ashiqul Haider Chowdhury	1806086	Department of EEE Section: B1 Group: 07 Level-3 term-1
Fazle Rabbi	1806087	
Md. Ayenul Azim Jahin	1806089	
Indrojit Sarkar	1806090	
Al Amin Patwary	1706127	

Objective:

We have developed a flexible audio search engine. The algorithm is noise and distortion resistant, computationally efficient, and capable of quickly identifying a short segment of music captured through a microphone in the presence of foreground voices and other dominant noise, and through voice codec compression, out of a database of many tracks. The algorithm uses a hashed time-frequency constellation analysis of the audio. Furthermore, for applications such as radio monitoring, search times on the order of a few milliseconds per query are attained, even on a massive music database.

Introduction:

Audio fingerprinting is the process of representing an audio signal in a compact way by extracting relevant features of the audio content. It works on the principle of human fingerprinting. It records the fingerprint of ingested audio content and later can be used to match with the recorded audio or playlists on mobile, TV or any other device. Audio fingerprinting allows monitoring of the audio independent of its format and without the need for metadata. A robust acoustic fingerprinting identifies the audio track even after compression and any degradation in sound quality. Some of the major applications of acoustic fingerprinting include content-based audio retrieval, broadcast monitoring, etc.

There can be multiple reasons to have audio fingerprinting technology in place. Along with audio track identification, one might want to know the singer/ writer of the song, where in the audio track we are listening to. This allows us to synchronize the multiple audio pieces and provides more interactive experiences. It is an efficient mechanism to establish the perceptual equality of two audio objects. The major advantages in developing a system with audio fingerprints are reduced memory /storage requirements, valuable comparison, perceptual irrelevancies getting removed and useful search. In this project we tried to establish a music retrieval application.

Familiarization with audio fingerprinting system:

Fingerprint systems are over one hundred years old. In 1893 Sir Francis Galton was the first to “prove” that no two fingerprints of human beings were alike. This system relies on the pattern of dermal ridges on the fingertips and still forms the basis of all “human” fingerprinting techniques of today. Conceptually a fingerprint can be seen as a “human” summary or signature that is unique for every human being. It is important to note that a human fingerprint differs from a textual summary in that it does not allow the reconstruction of other aspects of the original. For example, a human fingerprint does not convey any information about the color of the person’s hair or eyes.

Recent years have seen a growing scientific and industrial interest in computing fingerprints of multimedia object. The prime objective of multimedia fingerprinting is an efficient mechanism to establish the perceptual equality of two multimedia objects: not by comparing the (typically large) objects themselves, but by comparing the associated fingerprints (small by design). In most systems using fingerprinting technology, the fingerprints of a large number of multimedia objects, along with their associated meta-data (e.g. name of artist, title and album) are stored in a database. The fingerprints serve as an index to the meta-data. The meta-data of unidentified multimedia content are then retrieved by computing a fingerprint and using this as a query in the fingerprint/meta-data database.

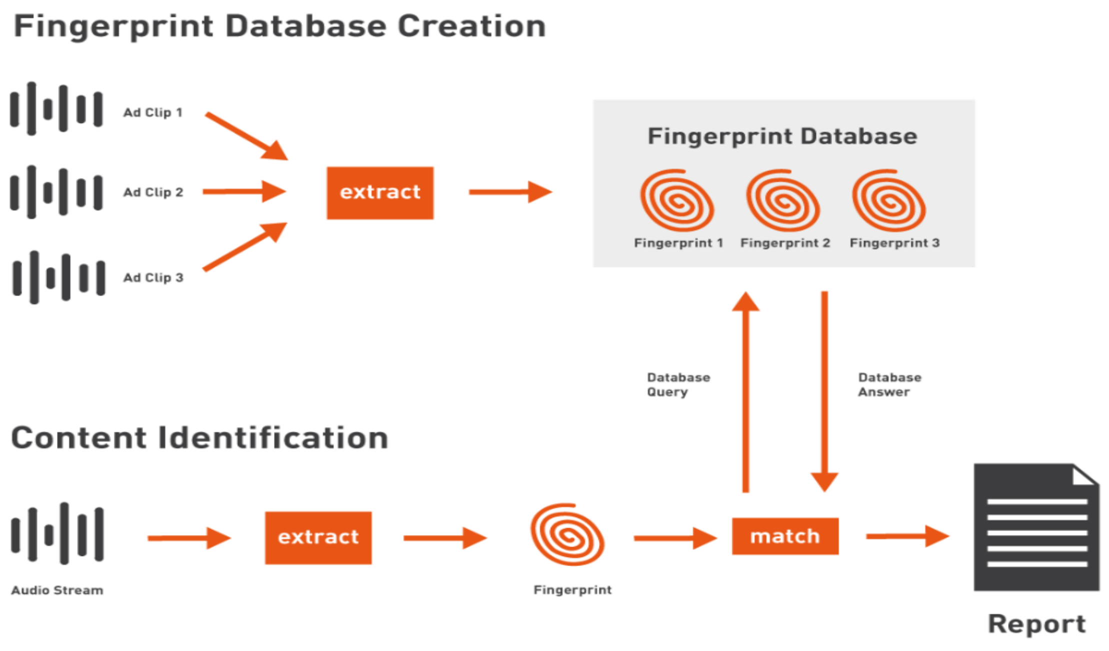


Figure: Audio fingerprinting system

The advantage of using fingerprints instead of the multimedia content itself is three-fold:

- Reduced memory/storage requirements as fingerprints are relatively small.
- Efficient comparison as perceptual irrelevancies have already been removed from fingerprints;
- Efficient searching as the dataset to be searched is smaller.

As can be concluded from above, a fingerprint system generally consists of two components: a method to extract fingerprints and a method to efficiently search for matching fingerprints in a fingerprint database. It is sufficient to store the set of hash values $\{hi = H(Yi)\}$, and to compare $H(X)$ with this set of hash values. At first one might think that cryptographic hash functions are a good candidate for fingerprint functions. However, instead of strict mathematical equality, we are interested in perceptual similarity. For example, an original CD quality version of 'Rolling Stones – Angie' and an MP3 version at 128Kb/s sound the same to the human auditory system, but their waveforms can be quite different. Although the two versions are perceptually similar, they are mathematically quite different.
Therefore, cryptographic hash functions cannot decide upon perceptual equality of these two versions. Even worse, cryptographic hash functions are typically bit-sensitive: a single bit of difference in the original object results in a completely different hash value. We propose to construct a fingerprint function in such a way that perceptual similar audio objects result in similar fingerprints. Furthermore, in order to be able to discriminate between different audio objects, there must be a very high probability that dissimilar audio objects result in dissimilar fingerprints.

Audio Fingerprint System Parameters:

Having a proper definition of an audio fingerprint we now focus on the different parameters of an audio fingerprint system. The main parameters are:

- **Robustness**: In order to achieve high robustness, the fingerprint should be based on perceptual features that are invariant (at least to a certain degree) with respect to signal degradations. Preferably, severely degraded audio still leads to very similar fingerprints. The false negative rate is generally used to express the robustness. A false negative occurs when the fingerprints of perceptually similar audio clips are too different to lead to a positive match.

- **Reliability**: How often is a song incorrectly identified? E.g. "Rolling Stones – Angie" being identified as "Beatles – Yesterday". The rate at which this occurs is usually referred to as the false positive rate.

- **Fingerprint size:** To enable fast searching, fingerprints are usually stored in RAM memory. Therefore, the fingerprint size, usually expressed in bits per second or bits per song, determines to a large degree the memory resources that are needed for a fingerprint database server.

- **Granularity**: How many seconds of audio is needed to identify an audio clip? Granularity is a parameter that can depend on the application. In some applications the whole song can be used for identification, in others one prefers to identify a song with only a short excerpt of audio.

- **Search speed and scalability:** How long does it take to find a fingerprint in a fingerprint database? What if the database contains thousands and thousands of songs? For the commercial deployment of audio fingerprint systems, search speed and scalability are a key parameter. Search speed should be in the order of milliseconds for a database containing over huge number of songs using only limited computing resources.

These five basic parameters have a large impact on each other. For instance, if one wants a lower granularity, one needs to extract a larger fingerprint to obtain the same reliability. This is due to the fact that the false positive rate is inversely related to the fingerprint size. Another example: search speed generally increases when one designs a more robust fingerprint. This is due to the fact that a fingerprint search is a proximity search. I.e. a similar (or the most similar) fingerprint has to be found. If the features are more robust the proximity is smaller. Therefore the
search speed can increase.

Some applications of audio fingerprinting:

- **Broadcast Monitoring**

Broadcast monitoring is probably the most well-known application for audio fingerprinting. It refers to the automatic playlist generation of radio, television or web broadcasts for, among others, purposes of royalty collection, program verification, advertisement verification and people metering. A large-scale broadcast monitoring system based on fingerprinting consists of several monitoring sites and a central site where the fingerprint server is located. At the monitoring sites fingerprints are extracted from all the (local) broadcast channels. The central site collects the fingerprints from the monitoring sites. Subsequently, the fingerprint server, containing a huge fingerprint database, produces the playlists of all the broadcast channels.

- **Connected Audio**

Connected audio is a general term for consumer applications where music is somehow connected to additional and supporting information. The example given in the abstract, using a mobile phone to identify a song is one of these examples. This business is actually pursued by a number of companies. The audio signal in this application is severely degraded due to processing applied by radio stations, FM/AM transmission, the acoustical path between the loudspeaker and the microphone of the mobile phone, speech coding and finally the transmission over the mobile network. Therefore, from a technical point of view, this is a very challenging application. Other examples of connected audio are (car) radios with an identification button or fingerprint applications “listening” to the audio streams leaving or entering a soundcard on a PC. By pushing an “info” button in the fingerprint application, the user could be directed to a page on the Internet containing information about the artist. Or by pushing a “buy” button the user would be able to buy the album on the Internet. In other words, audio fingerprinting can provide a universal linking system for audio content.

- **Filtering Technology for File Sharing**

Filtering refers to active intervention in content distribution. Audio fingerprinting can be applied in monitoring the content of a file, if it is copyrighted or illegal to share. The prime example for filtering technology for file sharing was Napster. Starting in June 1999, users who downloaded the Napster client could share and download a large collection of music for free. Later, due to a court case by the music industry, Napster users were forbidden to download copyrighted songs. Therefore, in March 2001 Napster installed an audio filter based on file names, to block downloads of copyrighted songs. The filter was not very effective, because users started to intentionally misspell filenames. In May 2001 Napster introduced an audio fingerprinting system by Relatable, which aimed at filtering out copyrighted material even if it was misspelled. Owing to Napster's closure only two months later, the effectiveness of that specific fingerprint system is, to the best of the author's knowledge, not publicly known.

- **Automatic Music Library Organization**

Nowadays many PC users have a music library containing several hundred, sometimes even thousands, of songs. The music is generally stored in compressed format (usually MP3) on their hard-drives. When these songs are obtained from different sources, such as ripping from a CD or downloading from file sharing networks, these libraries are often not well organized. Meta-data is often inconsistent, incomplete and sometimes even incorrect. Assuming that the fingerprint database contains correct meta-data, audio fingerprinting can make the meta-data of the songs in the library consistent, allowing easy organization base on, for example, album or artist. For example, ID3Man, a tool powered by Auditude fingerprinting technology is already available for tagging unlabeled or mislabeled MP3 files. A similar tool from Moodlogic is available as a Winamp plug-in.

About our project:

The main task of the project can be divided in to two parts: a method to extract fingerprints and a method to efficiently search for matching fingerprints from a fingerprint database.

Firstly, we have created a database consisting of fingerprints of songs. For collecting fingerprint, we have used spectrogram analysis. A visual representation of a spectrogram of a song is shown below. Here, the reddish parts indicate higher frequency components whereas the greenish parts indicate lower frequency components. Local maxima of the spectrogram are marked by black points.

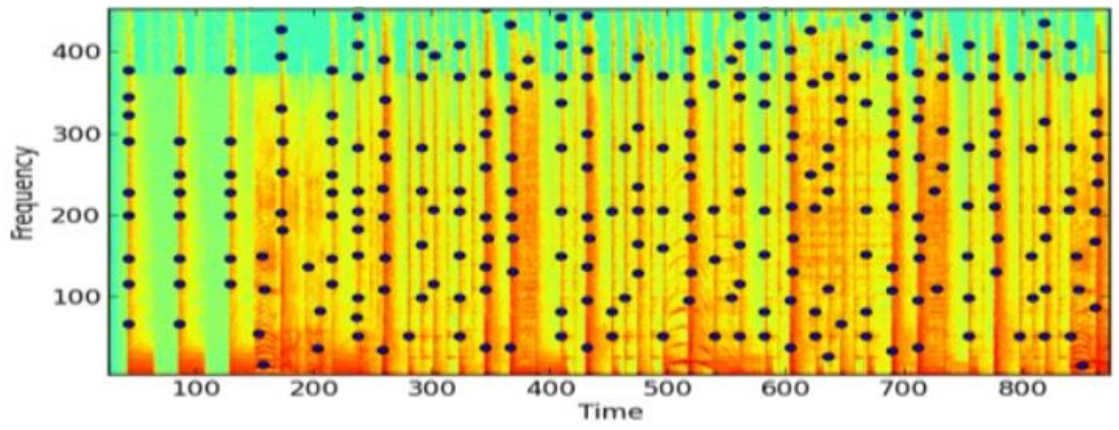


Figure: Spectrogram of a song with peak points marked

Our main goal is to match the peak points of the sample data with the peaks of the original song. Which can be observed in the following plot.

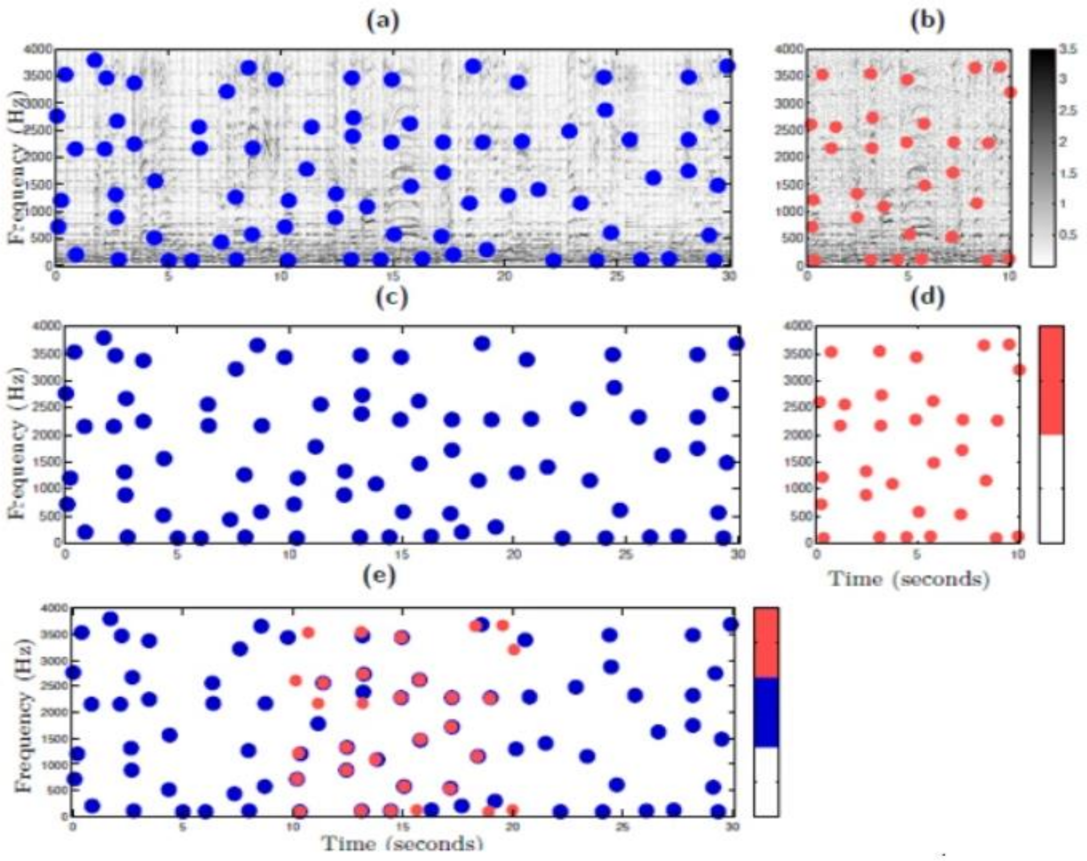
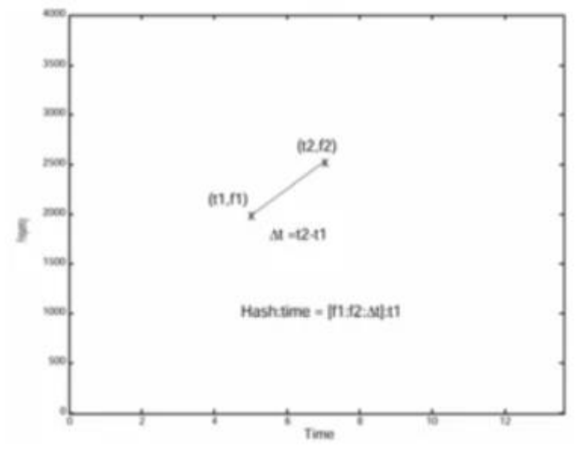


Figure: Matching sample's peak points with song's peak points

In the above figure, part (a) shows the spectrogram of a song with peak points marked. In part (b), the spectrogram of a small clip of the song with peak points marked. In part(c) and (d) the peaks of the song and sample are filtered out. The peak frequencies of the sample are only slightly altered with respect to the original songs. And it overlaps with the song's peaks correctly. So, the peak points can used as a unique identification quality. We used the peak points as features extraction.



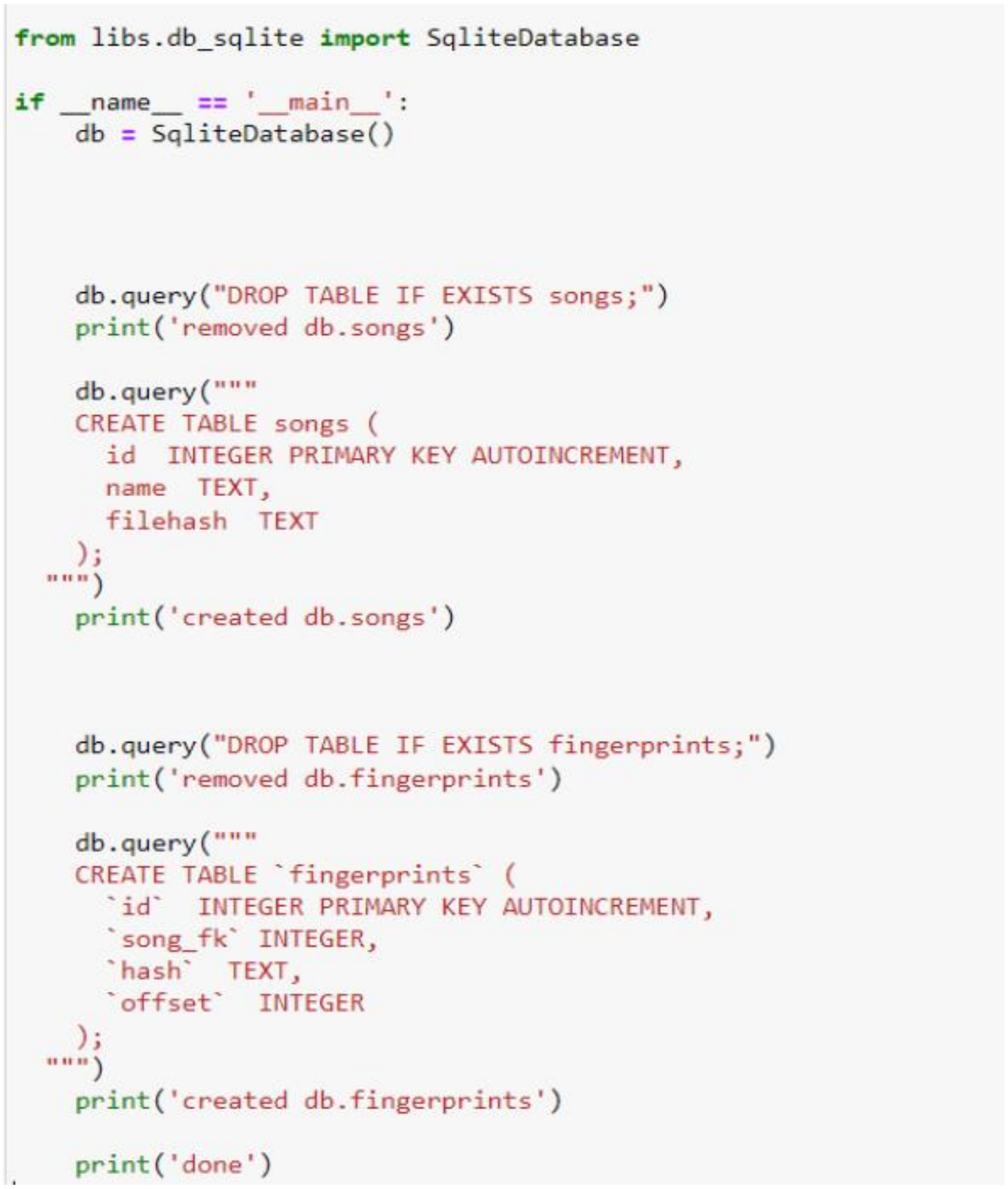
Here we took peak point pairs to create the hash table which is the unique parameter can be used to distinguish audio clips. At first, we mark a peak point, say point 1. Then a pair is formed with the next peak point, say point 2. Frequencies of two peak points and the time difference between are stored. The information is stacked into a string. Then Sha1 function of hashlib is used to convert the string in a hexadecimal sequence. First 20 indexes of the hexadecimal sequence are taken into consideration. After that, hash sequence, time t1 (time of peak 1) and song id are stored in the hash table. Same is done for point 1 and point 3. For a single peak point next 15 neighbor peak point are considered to form a pair and hash sequence is generated. A hash table of hexadecimal sequence are formed for all the songs in database. Later using same procedure, sequences of hexadecimal are generated of a sample clip.

Here we iterated through the Database and checked if our extracted fingerprint of a particular song matches with any of the songs of our DataBase. Initially we stored fingerprints in Hashes. So while matching we actually checked how many matches we found in for each songs in the Database and returned the song with the maximum number of matches along with the number of hashes. Here we iterated through the Database and checked if our extracted fingerprint of a particular song matches with any of the songs of our DataBase. Initially we stored fingerprints in Hashes. So while matching we actually checked how many matches we found in for each songs in the Database and returned the song with the maximum number of matches along with the number of hashes.

Code and description:

Reset-database:

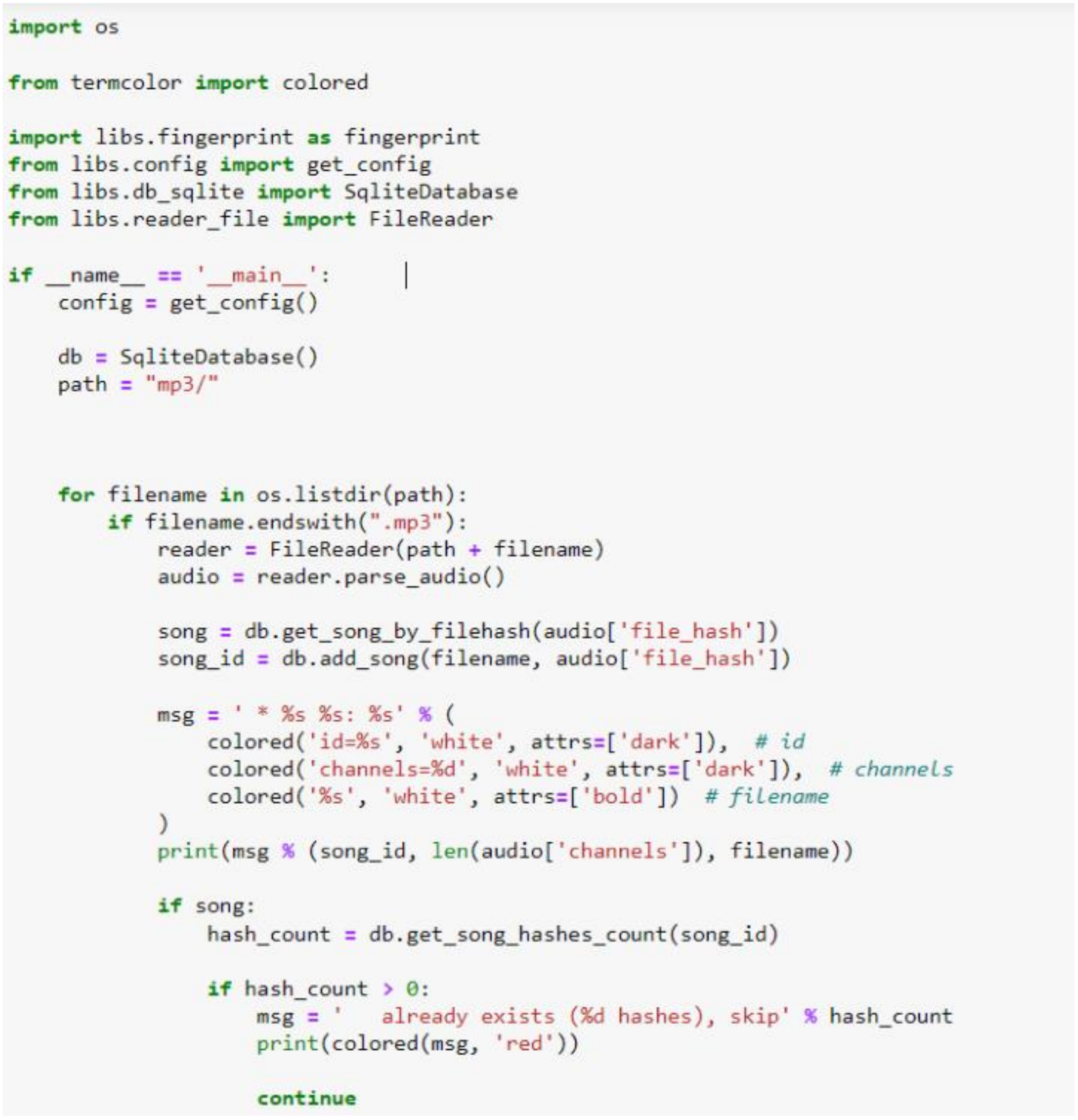
By running ‘reset_database.py’, we are deleting our existing database and prepare for making new database.



```
from libs.db_sqlite import SqliteDatabase

if __name__ == '__main__':
    db = SqliteDatabase()



    db.query("DROP TABLE IF EXISTS songs;")
    print('removed db.songs')

    db.query("""
    CREATE TABLE songs (
      id  INTEGER PRIMARY KEY AUTOINCREMENT,
      name  TEXT,
      filehash  TEXT
    );
  """)
    print('created db.songs')



    db.query("DROP TABLE IF EXISTS fingerprints;")
    print('removed db.fingerprints')

    db.query("""
    CREATE TABLE `fingerprints` (
      `id`  INTEGER PRIMARY KEY AUTOINCREMENT,
      `song_fk` INTEGER,
      `hash`  TEXT,
      `offset`  INTEGER
    );
  """)
    print('created db.fingerprints')

    print('done')
```

Collecting fingerprint from songs:

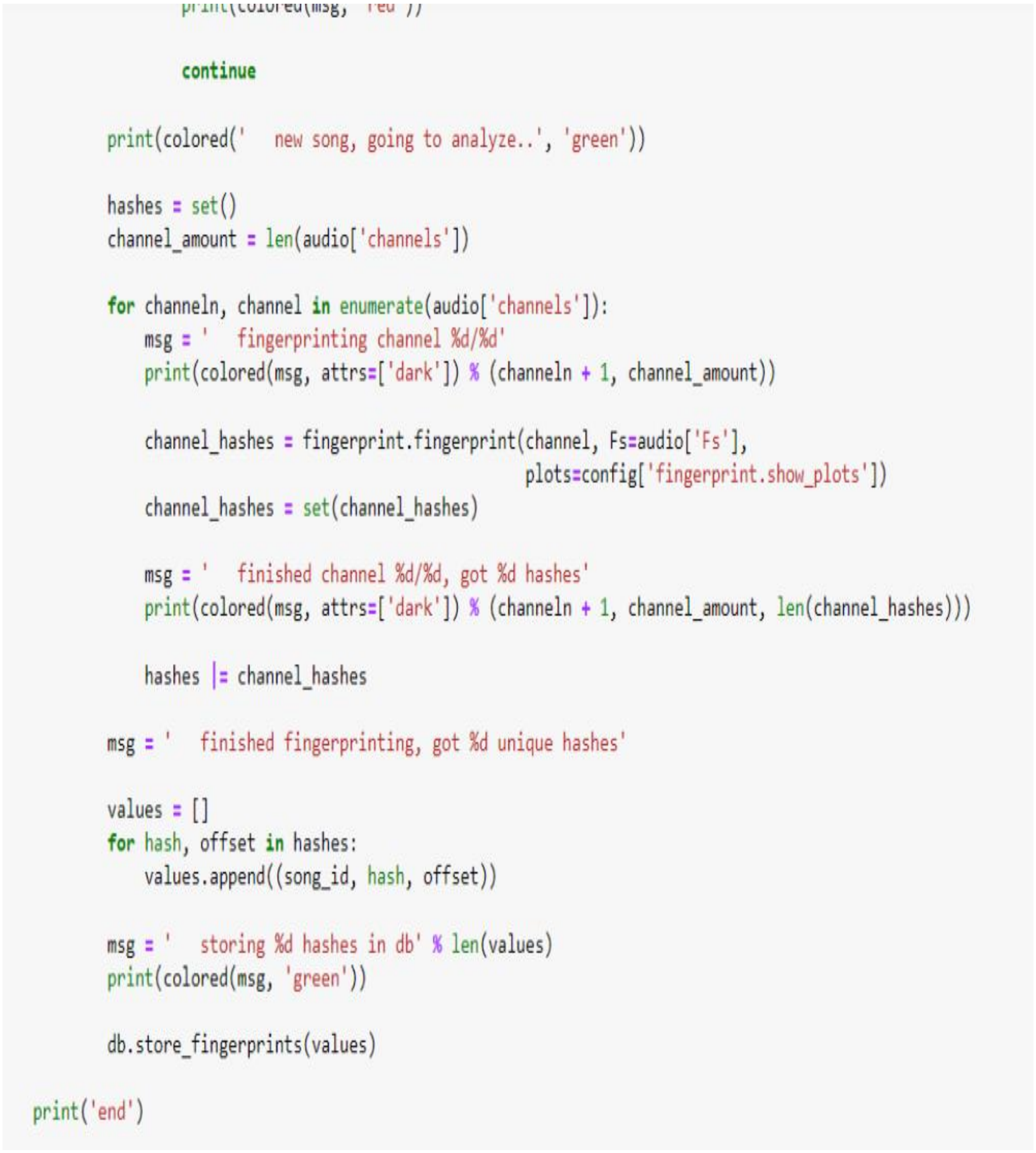
A script is written for collecting fingerprint from songs and storing it to database. First portion of script is shown below:



```python
import os

from termcolor import colored

import libs.fingerprint as fingerprint
from libs.config import get_config
from libs.db_sqlite import SqliteDatabase
from libs.reader_file import FileReader

if __name__ == '__main__':
    config = get_config()

    db = SqliteDatabase()
    path = "mp3/"



    for filename in os.listdir(path):
        if filename.endswith(".mp3"):
            reader = FileReader(path + filename)
            audio = reader.parse_audio()

            song = db.get_song_by_filehash(audio['file_hash'])
            song_id = db.add_song(filename, audio['file_hash'])

            msg = ' * %s %s: %s' % (
                colored('id=%s', 'white', attrs=['dark']),  # id
                colored('channels=%d', 'white', attrs=['dark']),  # channels
                colored('%s', 'white', attrs=['bold'])  # filename
            )
            print(msg % (song_id, len(audio['channels']), filename))

            if song:
                hash_count = db.get_song_hashes_count(song_id)

                if hash_count > 0:
                    msg = '   already exists (%d hashes), skip' % hash_count
                    print(colored(msg, 'red'))

                    continue
```

We have set the file path to ‘mp3/’ folder where we have collected songs for creating our database. At first, we have checked if the song exists in the database in the first place. If the song exists in the database, there is no need to add the song if it already exists, else duplicate songs entry will be present.

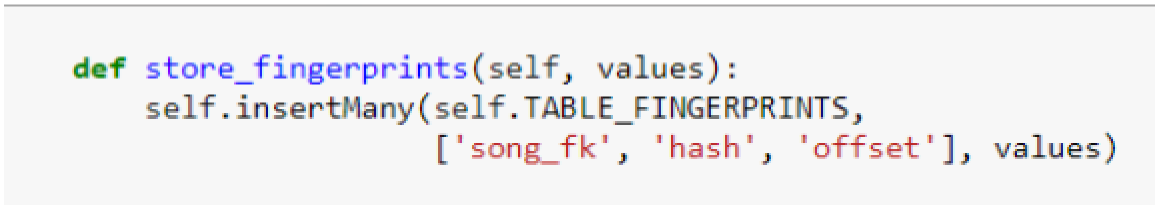
If the song is not found in the database, the following commands are executed.



```
            print(colored(msg, 'red'))

            continue

        print(colored('   new song, going to analyze..', 'green'))

        hashes = set()
        channel_amount = len(audio['channels'])

        for channeln, channel in enumerate(audio['channels']):
            msg = '   fingerprinting channel %d/%d'
            print(colored(msg, attrs=['dark']) % (channeln + 1, channel_amount))

            channel_hashes = fingerprint.fingerprint(channel, Fs=audio['Fs'],
                                                     plots=config['fingerprint.show_plots'])
            channel_hashes = set(channel_hashes)

            msg = '   finished channel %d/%d, got %d hashes'
            print(colored(msg, attrs=['dark']) % (channeln + 1, channel_amount, len(channel_hashes)))

            hashes |= channel_hashes

        msg = '   finished fingerprinting, got %d unique hashes'

        values = []
        for hash, offset in hashes:
            values.append((song_id, hash, offset))

        msg = '   storing %d hashes in db' % len(values)
        print(colored(msg, 'green'))

        db.store_fingerprints(values)

print('end')
```

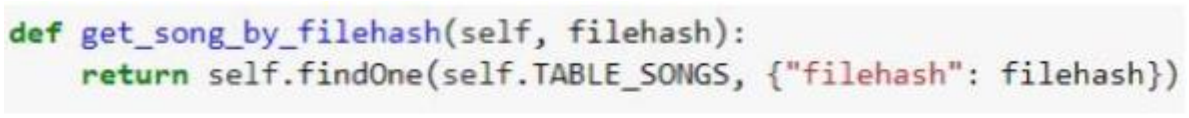
Here we have extracted features from each song and converted the features to a unique number which is stored in ‘hashes’. We will have a better insight of how feature extraction is done and how we are making so call unique sequence named ’hashes’. The hashes are stored in db.store_fingerprints.

A function ‘store_fingerprints’ defined in db class is inserting the hash values in the fingerprint tables named ‘TABLE_FINGERPRINTS’.



```
def store_fingerprints(self, values):
    self.insertMany(self.TABLE_FINGERPRINTS,
                    ['song_fk', 'hash', 'offset'], values)
```

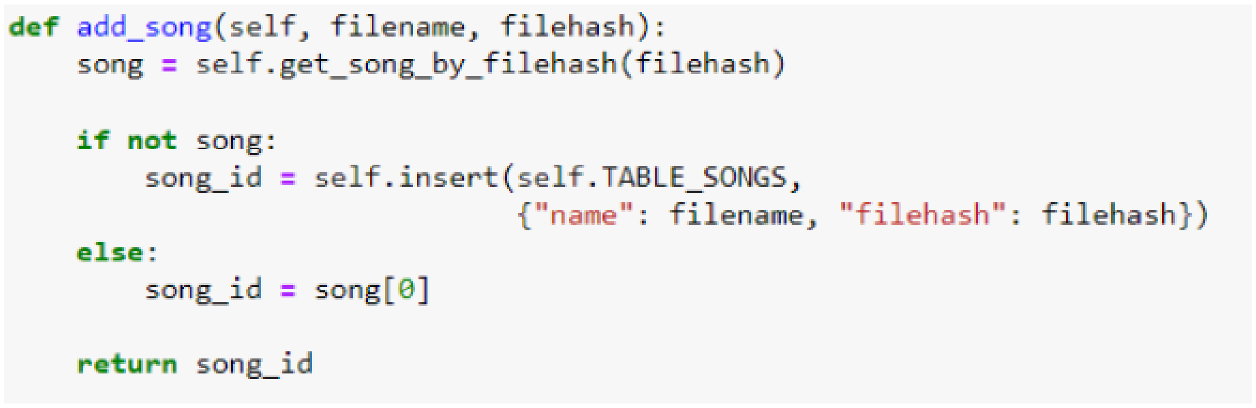
Other function defined from in db file:



```
def get_song_by_filehash(self, filehash):
    return self.findOne(self.TABLE_SONGS, {"filehash": filehash})
```

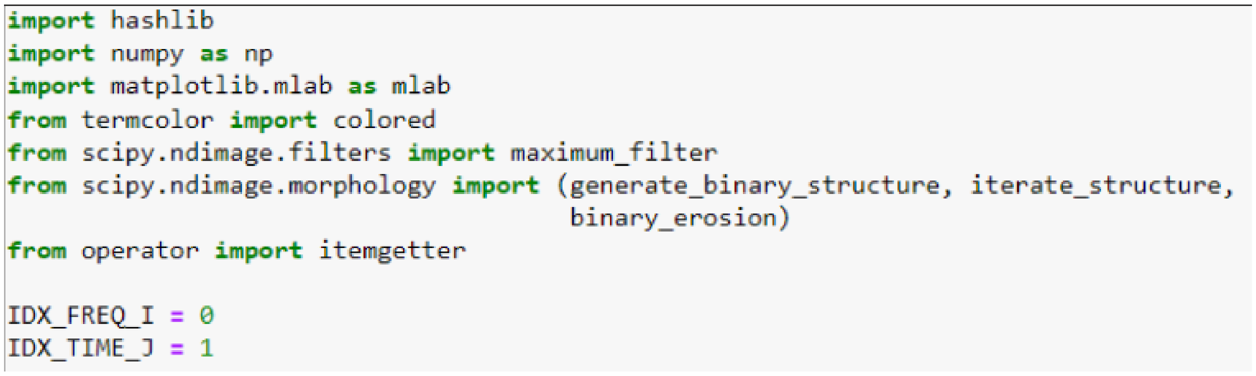
It used to check if song exists in the TABLE_SONGS, else duplicate entries will be present in the table.

If song does not exist in TABLE_SONGS, a new entry is made to the which is done by the function below.

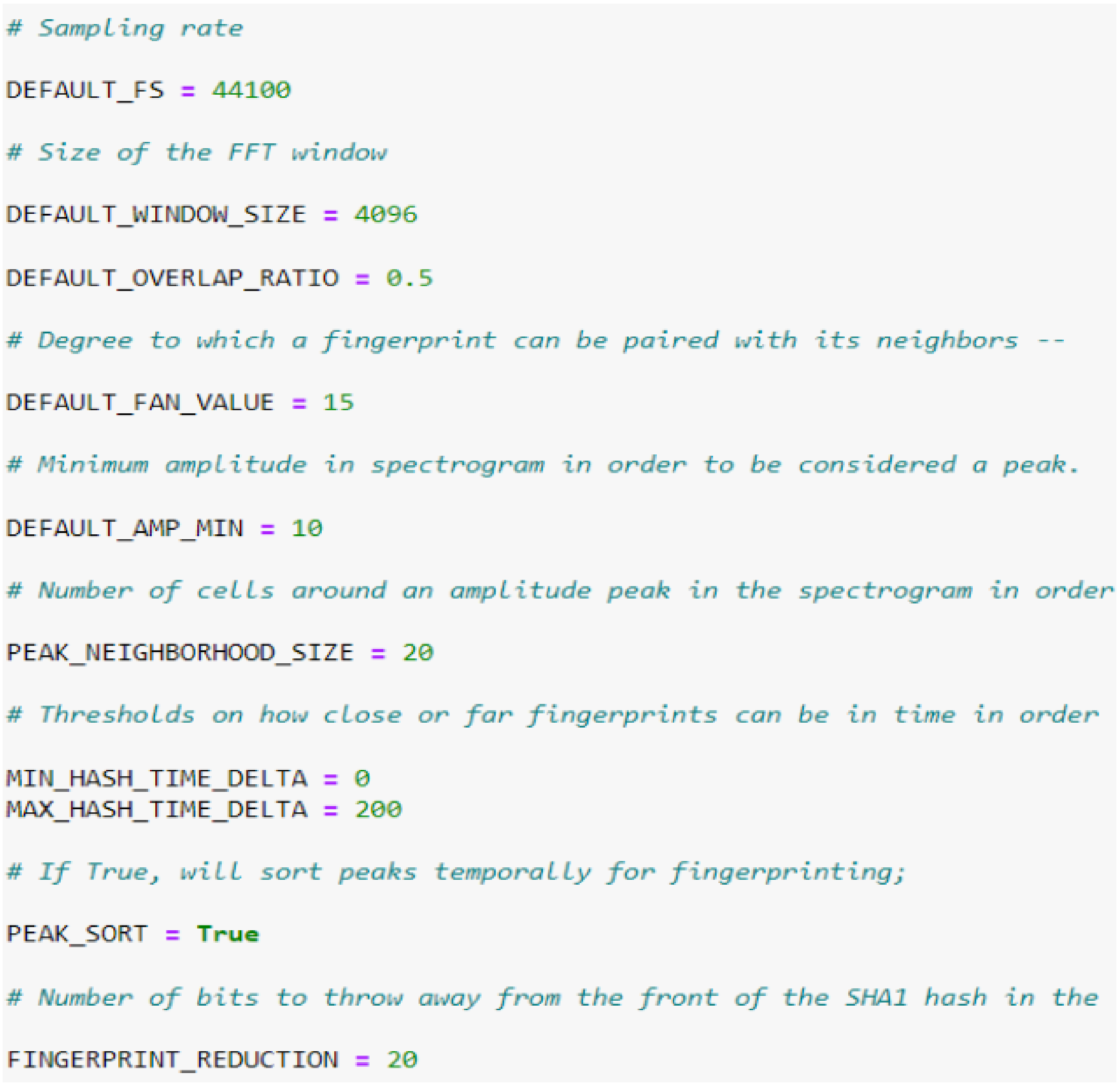


```
def add_song(self, filename, filehash):
    song = self.get_song_by_filehash(filehash)

    if not song:
        song_id = self.insert(self.TABLE_SONGS,
                              {"name": filename, "filehash": filehash})
    else:
        song_id = song[0]

    return song_id
```

The feature extraction and ‘hashes’ are done using ‘fingerprint.py’ function. Following libraries are imported in the function.

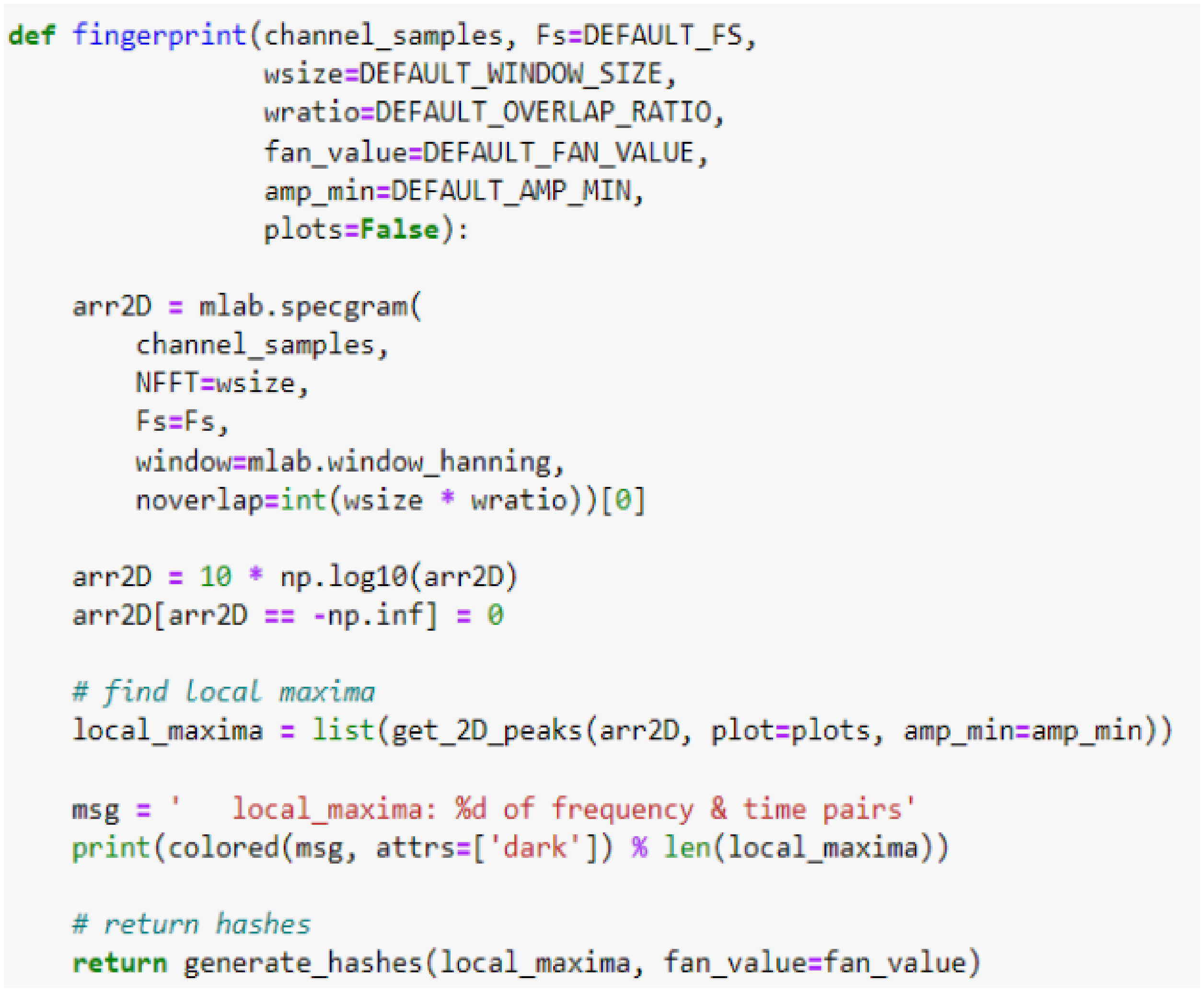


```
import hashlib
import numpy as np
import matplotlib.mlab as mlab
from termcolor import colored
from scipy.ndimage.filters import maximum_filter
from scipy.ndimage.morphology import (generate_binary_structure, iterate_structure,
                                      binary_erosion)
from operator import itemgetter

IDX_FREQ_I = 0
IDX_TIME_J = 1
```



```
# Sampling rate

DEFAULT_FS = 44100

# Size of the FFT window

DEFAULT_WINDOW_SIZE = 4096

DEFAULT_OVERLAP_RATIO = 0.5

# Degree to which a fingerprint can be paired with its neighbors --

DEFAULT_FAN_VALUE = 15

# Minimum amplitude in spectrogram in order to be considered a peak.

DEFAULT_AMP_MIN = 10

# Number of cells around an amplitude peak in the spectrogram in order

PEAK_NEIGHBORHOOD_SIZE = 20

# Thresholds on how close or far fingerprints can be in time in order

MIN_HASH_TIME_DELTA = 0
MAX_HASH_TIME_DELTA = 200

# If True, will sort peaks temporally for fingerprinting;

PEAK_SORT = True

# Number of bits to throw away from the front of the SHA1 hash in the

FINGERPRINT_REDUCTION = 20
```

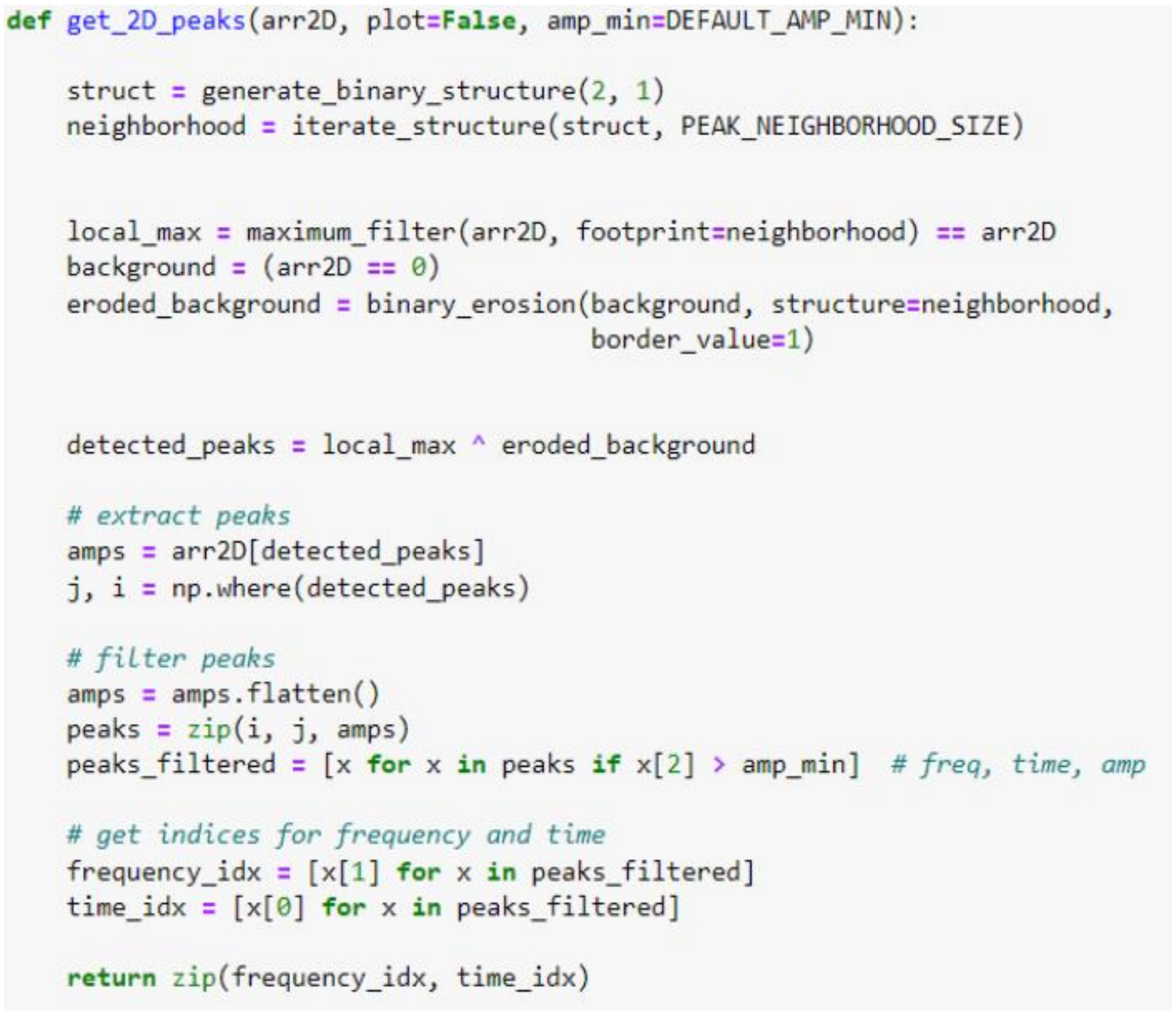
Here we have defined some parameters for the function. The description of the parameters is mentioned in the comment section.



```
def fingerprint(channel_samples, Fs=DEFAULT_FS,
                wsize=DEFAULT_WINDOW_SIZE,
                wratio=DEFAULT_OVERLAP_RATIO,
                fan_value=DEFAULT_FAN_VALUE,
                amp_min=DEFAULT_AMP_MIN,
                plots=False):

    arr2D = mlab.specgram(
        channel_samples,
        NFFT=wsize,
        Fs=Fs,
        window=mlab.window_hanning,
        noverlap=int(wsize * wratio))[0]

    arr2D = 10 * np.log10(arr2D)
    arr2D[arr2D == -np.inf] = 0

    # find local maxima
    local_maxima = list(get_2D_peaks(arr2D, plot=plots, amp_min=amp_min))

    msg = '   local_maxima: %d of frequency & time pairs'
    print(colored(msg, attrs=['dark']) % len(local_maxima))

    # return hashes
    return generate_hashes(local_maxima, fan_value=fan_value)
```

In ‘fingerprint’ function, we have passed arguments namely samples of a channel, window size or required frame size which will be used in short Fourier transform, sampling rate, window type to be used (which is Hanning window), overlapping ration between frames to be used in spectrogram. In ‘fingerprint’ function another function named ‘get_2D_peaks’ is called. The description of the function is mentioned below.

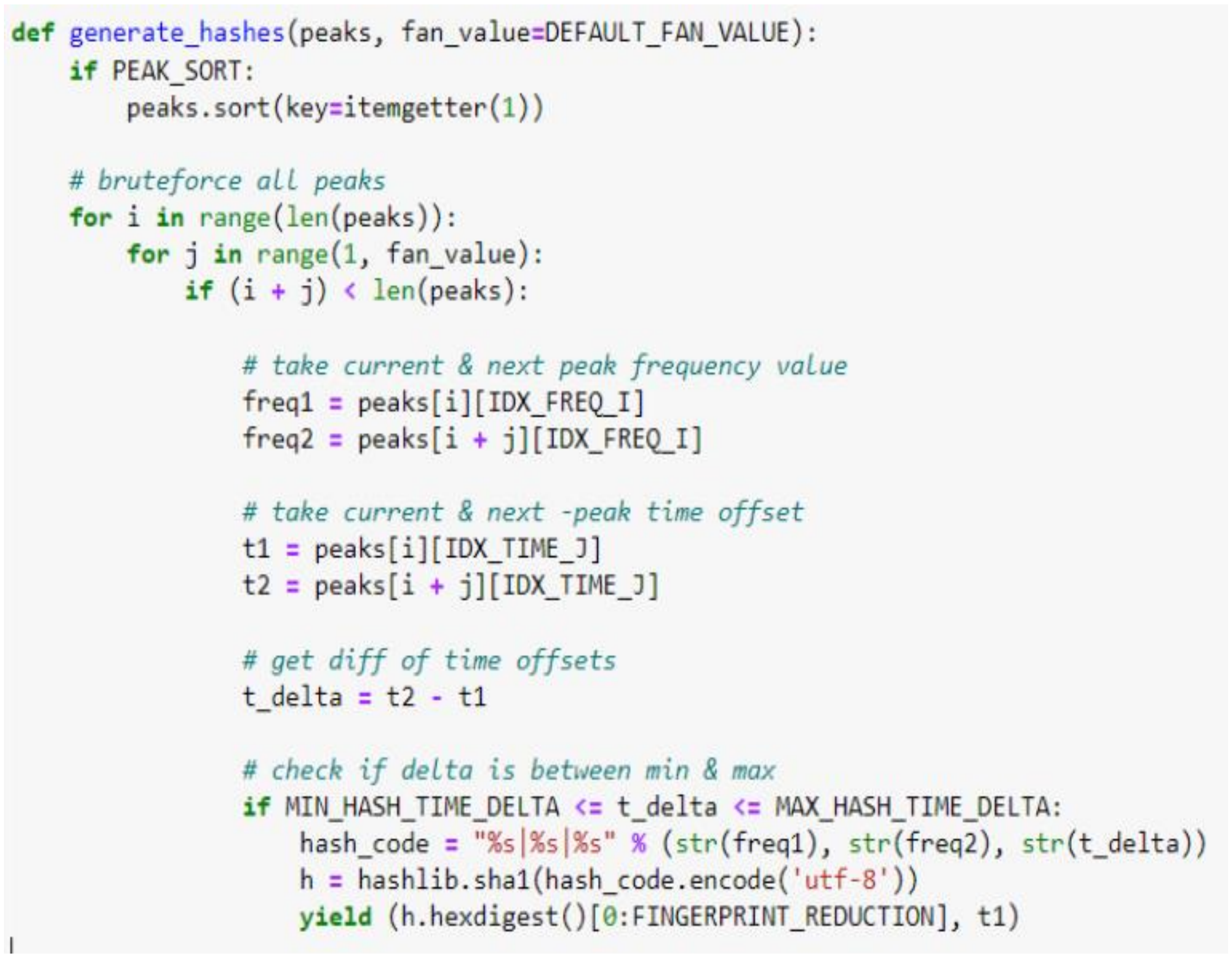
‘get_2D_peaks’ function takes the spectrogram as input, which contains the peaks, their time and frequency index and the amplitudes of the peaks. It returns frequency and time index of the maximum peaks.



```python
def get_2D_peaks(arr2D, plot=False, amp_min=DEFAULT_AMP_MIN):

    struct = generate_binary_structure(2, 1)
    neighborhood = iterate_structure(struct, PEAK_NEIGHBORHOOD_SIZE)


    local_max = maximum_filter(arr2D, footprint=neighborhood) == arr2D
    background = (arr2D == 0)
    eroded_background = binary_erosion(background, structure=neighborhood,
                                       border_value=1)


    detected_peaks = local_max ^ eroded_background

    # extract peaks
    amps = arr2D[detected_peaks]
    j, i = np.where(detected_peaks)

    # filter peaks
    amps = amps.flatten()
    peaks = zip(i, j, amps)
    peaks_filtered = [x for x in peaks if x[2] > amp_min]  # freq, time, amp

    # get indices for frequency and time
    frequency_idx = [x[1] for x in peaks_filtered]
    time_idx = [x[0] for x in peaks_filtered]

    return zip(frequency_idx, time_idx)
```

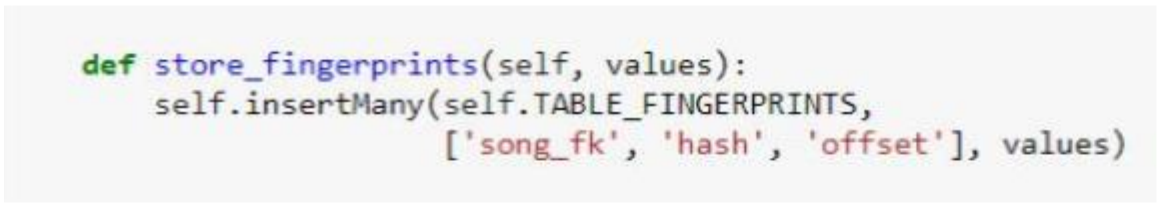
We are detecting local maximum in the spectrogram, they have a magnitude greater than 10 in db. The frequency and time index of the peaks are zipped and returned to the main function.

In the main function the ‘generate_hashes’ function is called. The peak point’s time and frequency index are passed as arguments in the functions. The function output is a hash sequence in hexadecimal. The function is defined below.



```python
def generate_hashes(peaks, fan_value=DEFAULT_FAN_VALUE):
    if PEAK_SORT:
        peaks.sort(key=itemgetter(1))

    # bruteforce all peaks
    for i in range(len(peaks)):
        for j in range(1, fan_value):
            if (i + j) < len(peaks):

                # take current & next peak frequency value
                freq1 = peaks[i][IDX_FREQ_I]
                freq2 = peaks[i + j][IDX_FREQ_I]

                # take current & next -peak time offset
                t1 = peaks[i][IDX_TIME_J]
                t2 = peaks[i + j][IDX_TIME_J]

                # get diff of time offsets
                t_delta = t2 - t1

                # check if delta is between min & max
                if MIN_HASH_TIME_DELTA <= t_delta <= MAX_HASH_TIME_DELTA:
                    hash_code = "%s|%s|%s" % (str(freq1), str(freq2), str(t_delta))
                    h = hashlib.sha1(hash_code.encode('utf-8'))
                    yield (h.hexdigest()[0:FINGERPRINT_REDUCTION], t1)
```

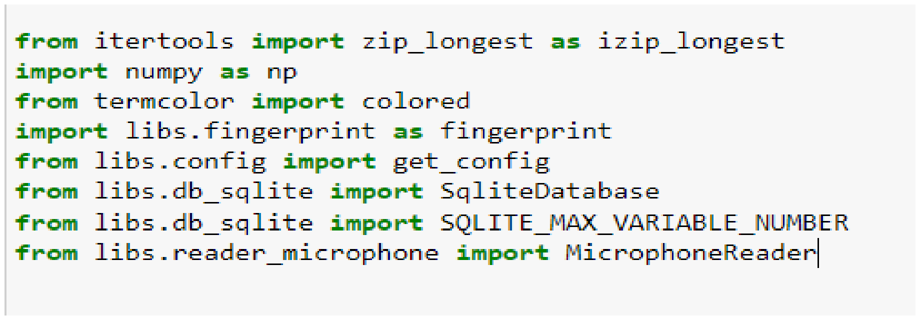
First, we have sorted the peak points according to time index. Then we have taken a pair of peaks. For ‘i’th peak, we have taken up to the next 15 th peaks to pair with. ‘fan_value’ represents the number of neighboring peak, which is paired with ‘i’th peak. In case of hashing we note the frequency of ‘i’th peak and ‘i+j’ th peak and the time difference between them. Each of this information is stacked in a string which is then converted into binary and inputted in sha1().hexdigest() function which creates a unique hexadecimal sequence for each set of values. It is then return with ‘i’th time index. The hashes and time index of ‘i’th peak are stored using ‘db.store_fingerprints’ along with song id in TABLE_FINGERPRINTS.



```python
    def store_fingerprints(self, values):
        self.insertMany(self.TABLE_FINGERPRINTS,
                        ['song_fk', 'hash', 'offset'], values)
```

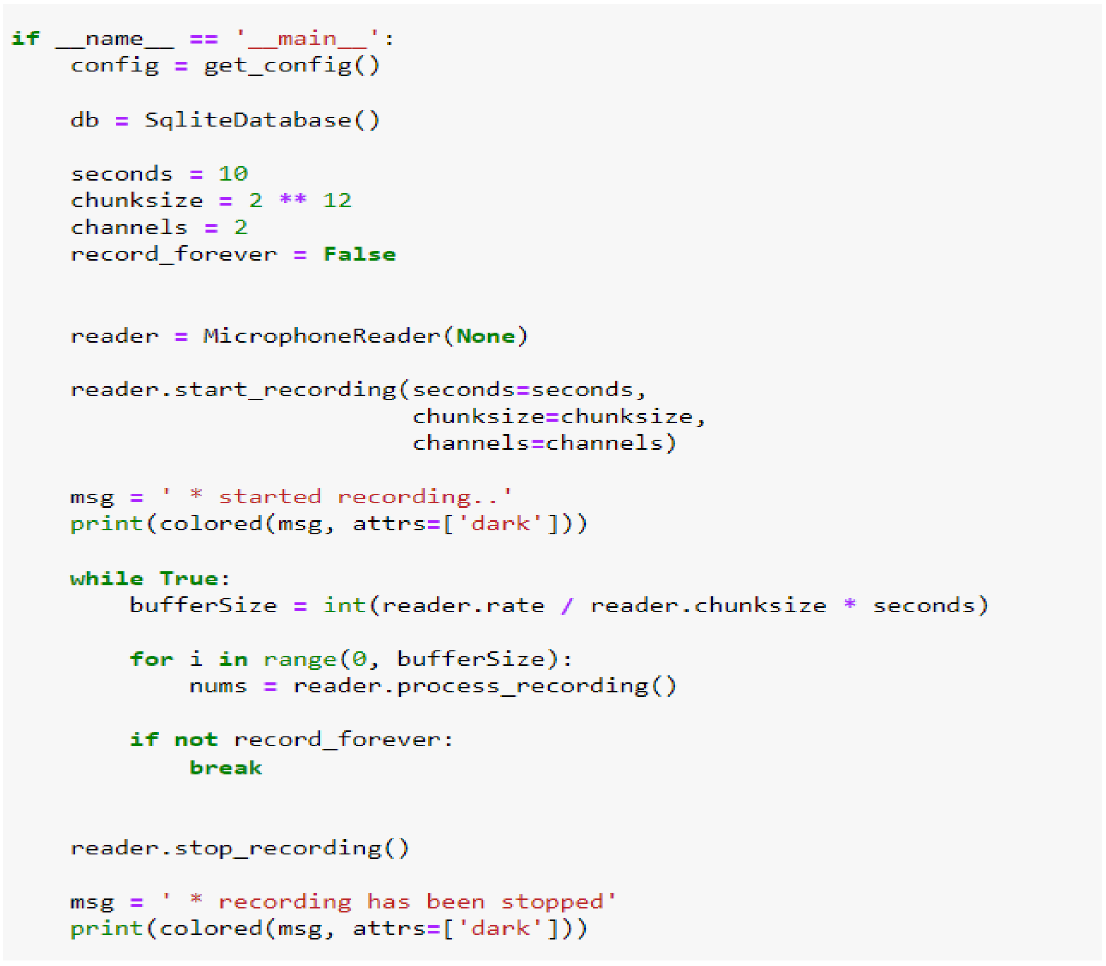
Recognize from microphone:

A script is written for Recognize the song from microphone and matching it to stored database. First portion (imported module) of script is shown below:



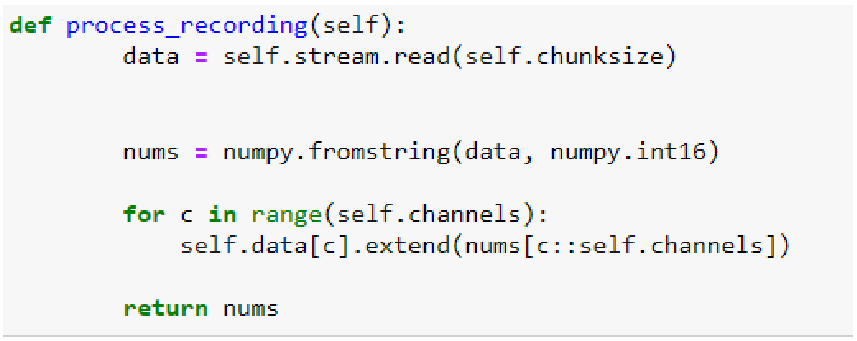
```python
from itertools import zip_longest as izip_longest
import numpy as np
from termcolor import colored
import libs.fingerprint as fingerprint
from libs.config import get_config
from libs.db_sqlite import SqliteDatabase
from libs.db_sqlite import SQLITE_MAX_VARIABLE_NUMBER
from libs.reader_microphone import MicrophoneReader
```

To start the recording from microphone following syntex has been evaluated. The recording time is 10 sec. recoded file has been divided into chunks and passed to reader.process_recording for further processing.



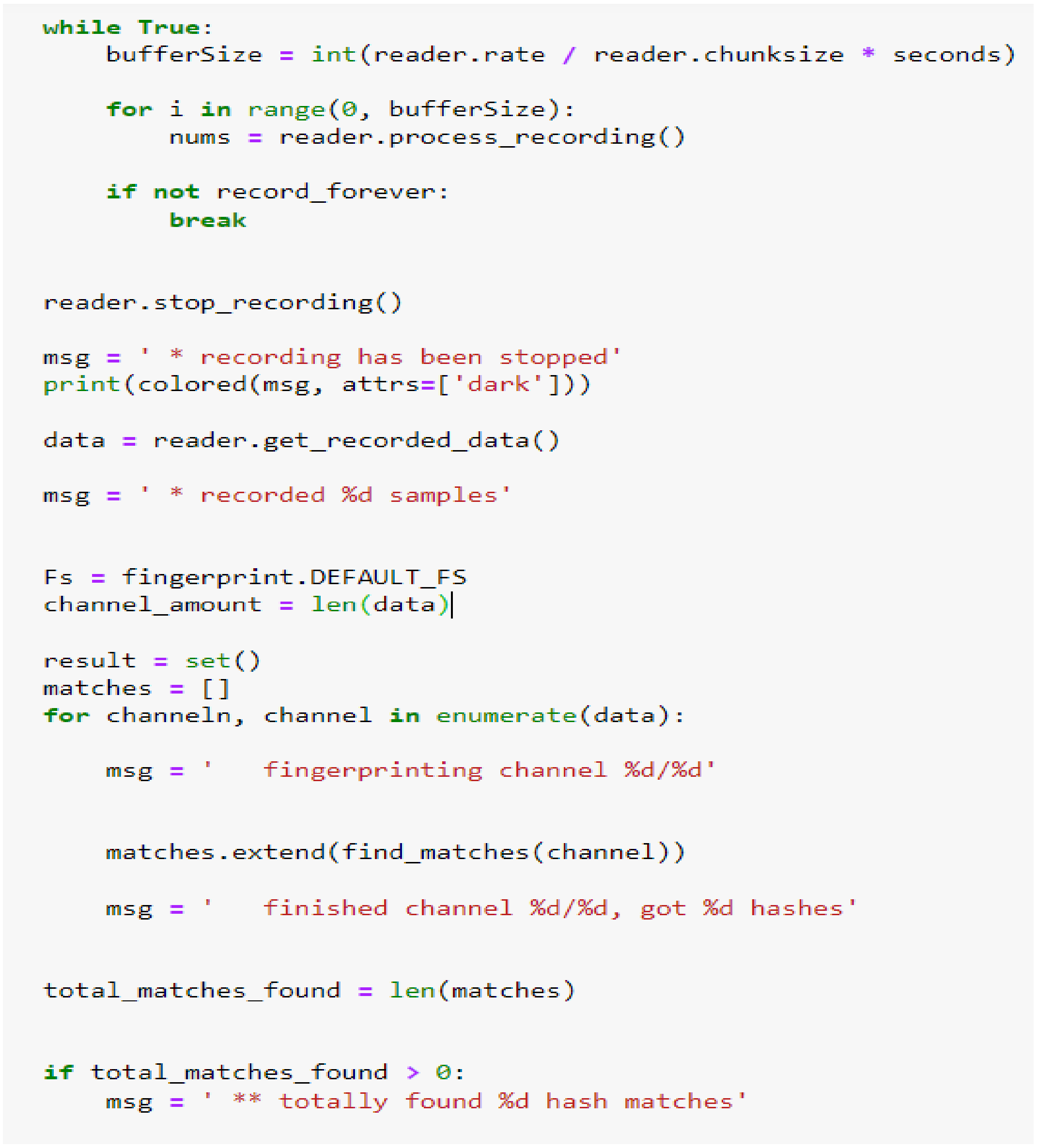
```python
if __name__ == '__main__':
    config = get_config()

    db = SqliteDatabase()

    seconds = 10
    chunksize = 2 ** 12
    channels = 2
    record_forever = False


    reader = MicrophoneReader(None)

    reader.start_recording(seconds=seconds,
                           chunksize=chunksize,
                           channels=channels)

    msg = ' * started recording..'
    print(colored(msg, attrs=['dark']))

    while True:
        bufferSize = int(reader.rate / reader.chunksize * seconds)

        for i in range(0, bufferSize):
            nums = reader.process_recording()

        if not record_forever:
            break


    reader.stop_recording()

    msg = ' * recording has been stopped'
    print(colored(msg, attrs=['dark']))
```

Reader.process_recording:



```
def process_recording(self):
        data = self.stream.read(self.chunksize)


        nums = numpy.fromstring(data, numpy.int16)

        for c in range(self.channels):
            self.data[c].extend(nums[c::self.channels])

        return nums
```

Generating Fingerpront of recorded audio file of microphone and then match it with database.



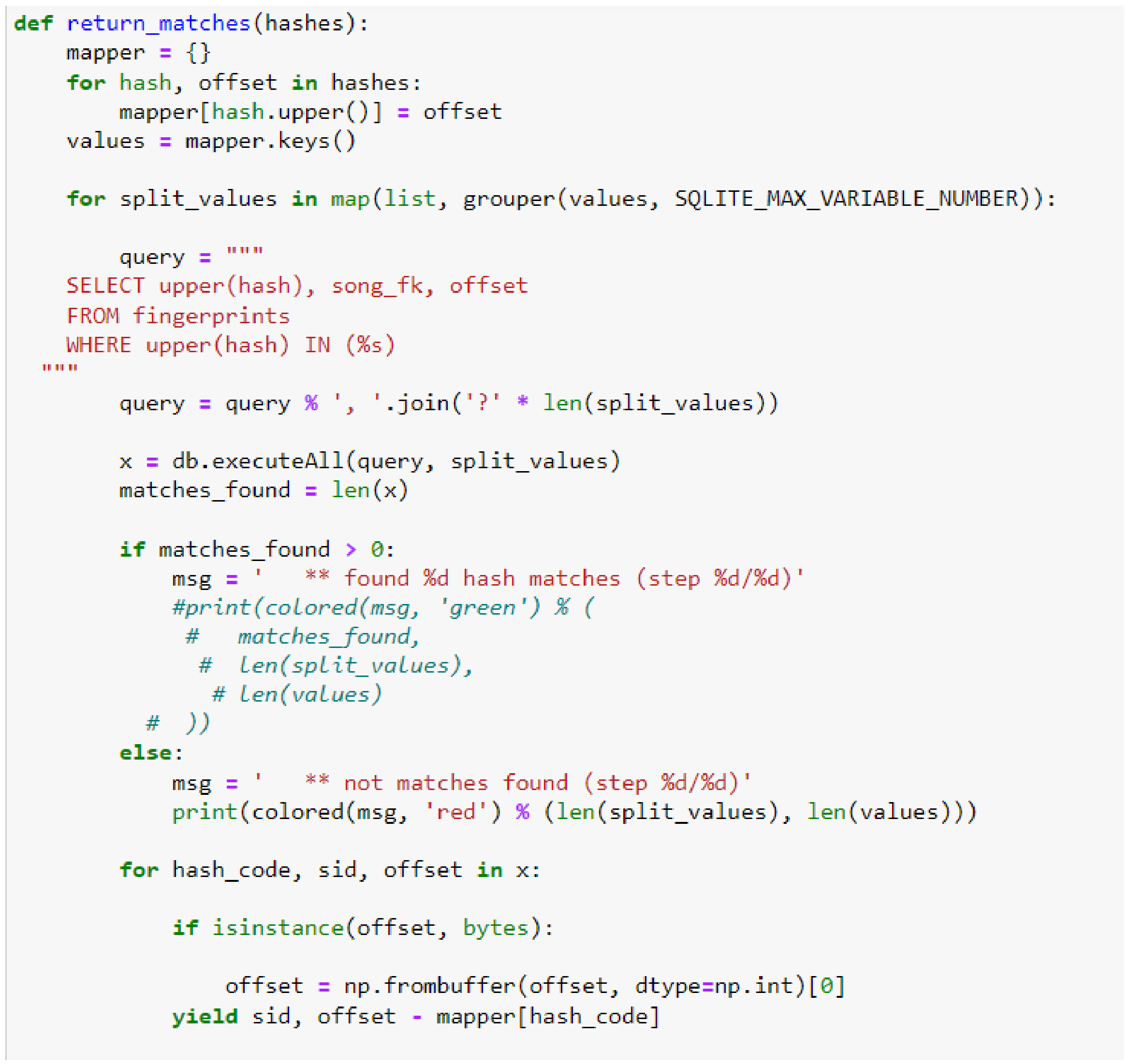
```
while True:
    bufferSize = int(reader.rate / reader.chunksize * seconds)

    for i in range(0, bufferSize):
        nums = reader.process_recording()

    if not record_forever:
        break


reader.stop_recording()

msg = ' * recording has been stopped'
print(colored(msg, attrs=['dark']))

data = reader.get_recorded_data()

msg = ' * recorded %d samples'


Fs = fingerprint.DEFAULT_FS
channel_amount = len(data)

result = set()
matches = []
for channeln, channel in enumerate(data):

    msg = '   fingerprinting channel %d/%d'


    matches.extend(find_matches(channel))

    msg = '   finished channel %d/%d, got %d hashes'


total_matches_found = len(matches)


if total_matches_found > 0:
    msg = ' ** totally found %d hash matches'
```

```
if total_matches_found > 0:
    msg = ' ** totally found %d hash matches'


    song = align_matches(matches)

    msg = ' => song: %s (id=%d)\n'

    print(colored(msg, 'green') % (song['SONG_NAME'], song['SONG_ID']))
else:
    msg = ' ** not matches found at all'
    print(colored(msg, 'red'))
```

In find matches return matches function is called.

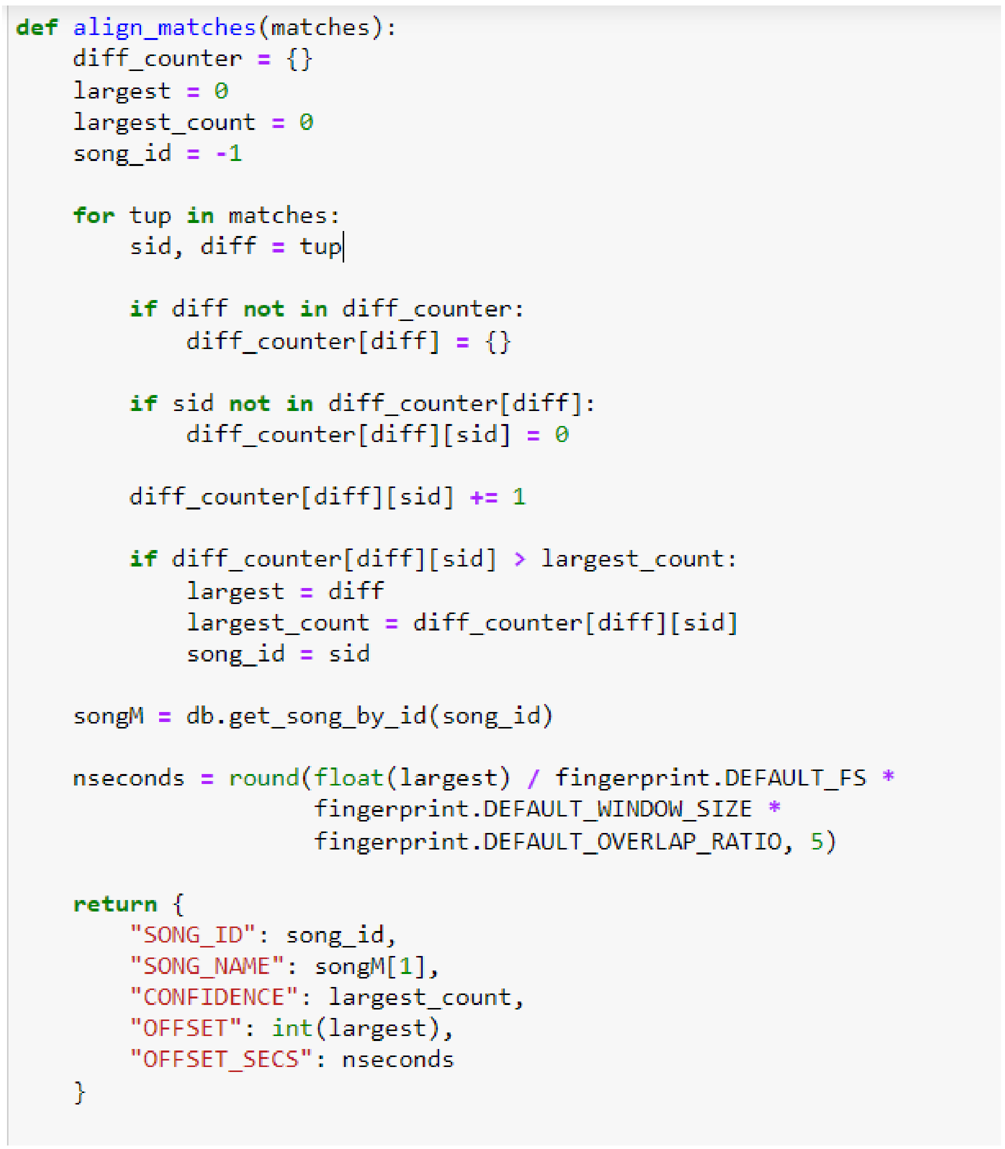
```
def grouper(iterable, n, fillvalue=None):
    args = [iter(iterable)] * n
    return (filter(None, values)
            for values in izip_longest(fillvalue=fillvalue, *args))


def find_matches(samples, Fs=fingerprint.DEFAULT_FS):
    hashes = fingerprint.fingerprint(samples, Fs=Fs)
    return return_matches(hashes)
```

In return_matches database searching operation is done and the matches are returned.



```python
def return_matches(hashes):
    mapper = {}
    for hash, offset in hashes:
        mapper[hash.upper()] = offset
    values = mapper.keys()

    for split_values in map(list, grouper(values, SQLITE_MAX_VARIABLE_NUMBER)):

        query = """
    SELECT upper(hash), song_fk, offset
    FROM fingerprints
    WHERE upper(hash) IN (%s)
  """
        query = query % ', '.join('?' * len(split_values))

        x = db.executeAll(query, split_values)
        matches_found = len(x)

        if matches_found > 0:
            msg = '   ** found %d hash matches (step %d/%d)'
            #print(colored(msg, 'green') % (
             #   matches_found,
              #  len(split_values),
               # len(values)
          #  ))
        else:
            msg = '   ** not matches found (step %d/%d)'
            print(colored(msg, 'red') % (len(split_values), len(values)))

        for hash_code, sid, offset in x:

            if isinstance(offset, bytes):

                offset = np.frombuffer(offset, dtype=np.int)[0]
            yield sid, offset - mapper[hash_code]
```

In align_matches the song name & id of maximum matched song from database has been returned.



```python
def align_matches(matches):
    diff_counter = {}
    largest = 0
    largest_count = 0
    song_id = -1

    for tup in matches:
        sid, diff = tup

        if diff not in diff_counter:
            diff_counter[diff] = {}

        if sid not in diff_counter[diff]:
            diff_counter[diff][sid] = 0

        diff_counter[diff][sid] += 1

        if diff_counter[diff][sid] > largest_count:
            largest = diff
            largest_count = diff_counter[diff][sid]
            song_id = sid

    songM = db.get_song_by_id(song_id)

    nseconds = round(float(largest) / fingerprint.DEFAULT_FS *
                     fingerprint.DEFAULT_WINDOW_SIZE *
                     fingerprint.DEFAULT_OVERLAP_RATIO, 5)

    return {
        "SONG_ID": song_id,
        "SONG_NAME": songM[1],
        "CONFIDENCE": largest_count,
        "OFFSET": int(largest),
        "OFFSET_SECS": nseconds
    }
```

Finally we get desired song name from a 10 sec audio recording of that particular song.

Performance & Challenges:

Challenges we tried to overcome and performance:

1. Background noise in some environments and significant noise power:

The use of this tools is mostly done in various concert and noisy environments. The noise power reduces the signal to noise ration. In this case the sensing of peak intensities can be problem. In some noisy environment our project can recognize some audio. But if there is huge noise then it fails to recognize the music. In our test run we add some human noise, environment noise it can recognize 65% of test music.

2. Distortion arising from sampling equipment:

Technology of the microphones vary for different handheld devices. The sampling frequency can be very fast sometimes and sometimes very slow. We have run this program in two devices. We get various output in two different devices. A song which is recognized by a device, may be unrecognized by another. This is due to the difference in microphone equipment quality.

3. Types of playback:

Some songs are being played at a faster rate (remix) and some at a slower rate (instrumental). The time difference between two recorded intensity points may vary with the original song in the database. If we play a music at .75,1.25 or 1.5 times of its original playback speed, we get some satisfying result for 1.25 times speed. But for 1.5 times speed we don’t get any successful result. Again for heavy metal song program fails to recognize because there are various frequency component and in 0.1s time interval there may be high dense of peak pints ,which can be improved in future.

4. Database management:

We have 150 songs in our database. If we increase our database, it will take much more time to complete the execution. Moreover, there will need more memory allocation and more time to search database index This has to be done without increasing the wait time for the result.

Result analysis:

In our database there were 150 songs (English, Bangla, Hindi). We have tested 3 times for a particular song, each length 10 sec from first 1/3, middle 1/3 and last 1/3. During this test background noise was also present. If the song passes 3 times, then we mark it as a detectable song.

Among 150 songs, 97 songs can be detected correctly.

Accuracy = (97/150)* 100% = 64.67%

We made an excel sheet of our tests, link provided here:

https://drive.google.com/drive/folders/1Z35H5dUgLlYDlsZmyrhyGcu9GDng5Oq_?usp=sharing

.

Limitation:

1. Different versions of song (with different singer/ with different instrument) cannot be detected.
2. Song with higher dominating frequency, can't be detect. also slow songs were very hard to detect. Almost 80% undetectable songs were included in this list.
3. The runtime of each searching was about 1 min which is not desirable for large database.

Future improvements:

1. To detect different versions of song, we have to enrich our database with popular versions of each song.

2. Slow songs and high dominating frequency performance is low. To improve this limitation, we have to classify the song first. According to which different parameter should be modified. Like, we can change the area size in which a value is marked as local maximum peak. Sampling rate, frame intervals can be modified according to song classification. We can classify a song with respect to song genre, bits per minutes etc. For song genre classification we can use CNN.

3. Better search algorithm can be used to improve runtime. Also, song classification will also narrow down our search region and give better performance.

Conclusion:

Audio fingerprinting is an important field in signal processing sector. Using feature extraction technique, we have tried to recognize random songs that's present in our database. We also test these songs in wild environment, varying speed (from 0.75-1.25x) and making distorted (Pause and play). By expanding our database and developing our algorithm this can be more robust.

References:

1. A review of algorithms for audio fingerprinting

https://ieeexplore.ieee.org/abstract/document/1203274

2. Audio Identification via Fingerprinting Achieving Robustness to Severe Signal Modifications

https://epub.jku.at/obvulihs/download/pdf/1951627?originalFilename=true