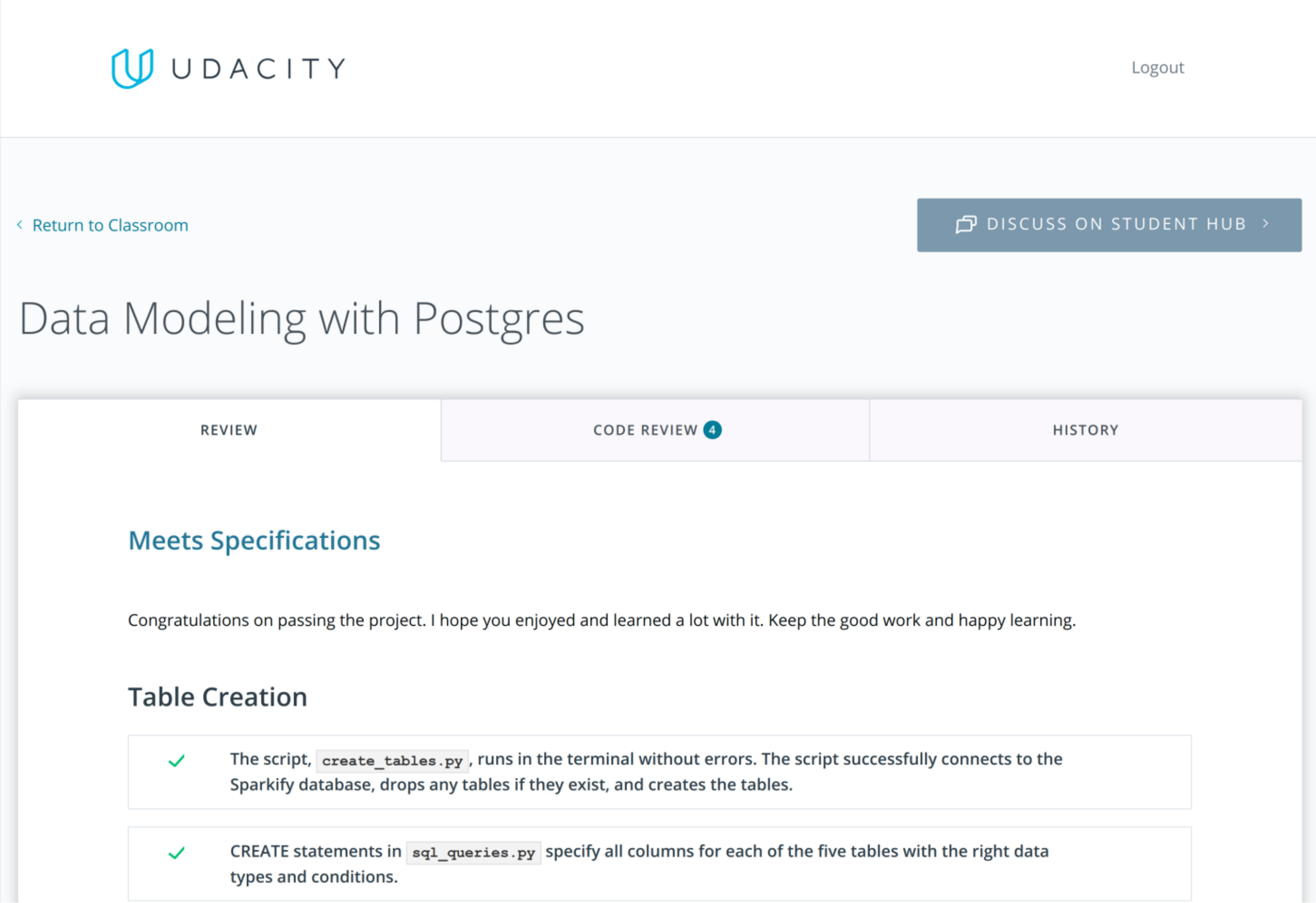


UDACITY

Logout

‹ Return to Classroom

DISCUSS ON STUDENT HUB ›

Data Modeling with Postgres

REVIEW | CODE REVIEW 4 | HISTORY

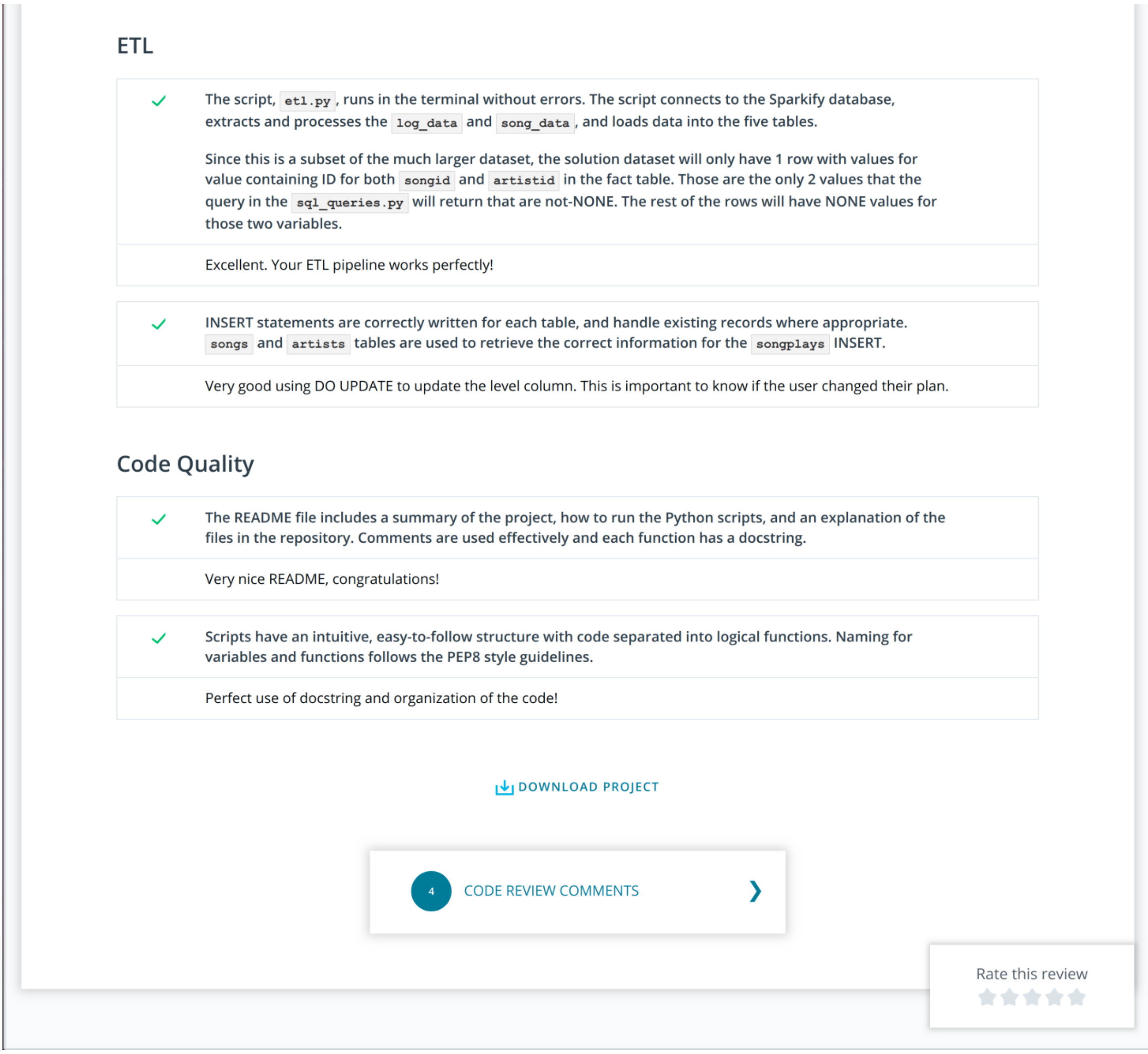
Meets Specifications

Congratulations on passing the project. I hope you enjoyed and learned a lot with it. Keep the good work and happy learning.

Table Creation

✓ The script, `create_tables.py`, runs in the terminal without errors. The script successfully connects to the Sparkify database, drops any tables if they exist, and creates the tables.

✓ CREATE statements in `sql_queries.py` specify all columns for each of the five tables with the right data types and conditions.



ETL

✓ The script, `etl.py`, runs in the terminal without errors. The script connects to the Sparkify database, extracts and processes the `log_data` and `song_data`, and loads data into the five tables.

Since this is a subset of the much larger dataset, the solution dataset will only have 1 row with values for value containing ID for both `songid` and `artistid` in the fact table. Those are the only 2 values that the query in the `sql_queries.py` will return that are not-NONE. The rest of the rows will have NONE values for those two variables.

Excellent. Your ETL pipeline works perfectly!

✓ INSERT statements are correctly written for each table, and handle existing records where appropriate. `songs` and `artists` tables are used to retrieve the correct information for the `songplays` INSERT.

Very good using DO UPDATE to update the level column. This is important to know if the user changed their plan.

Code Quality

✓ The README file includes a summary of the project, how to run the Python scripts, and an explanation of the files in the repository. Comments are used effectively and each function has a docstring.

Very nice README, congratulations!

✓ Scripts have an intuitive, easy-to-follow structure with code separated into logical functions. Naming for variables and functions follows the PEP8 style guidelines.

Perfect use of docstring and organization of the code!

Data Modeling with Postgres

REVIEW | CODE REVIEW 4 | HISTORY

▾ sql_queries.py 3

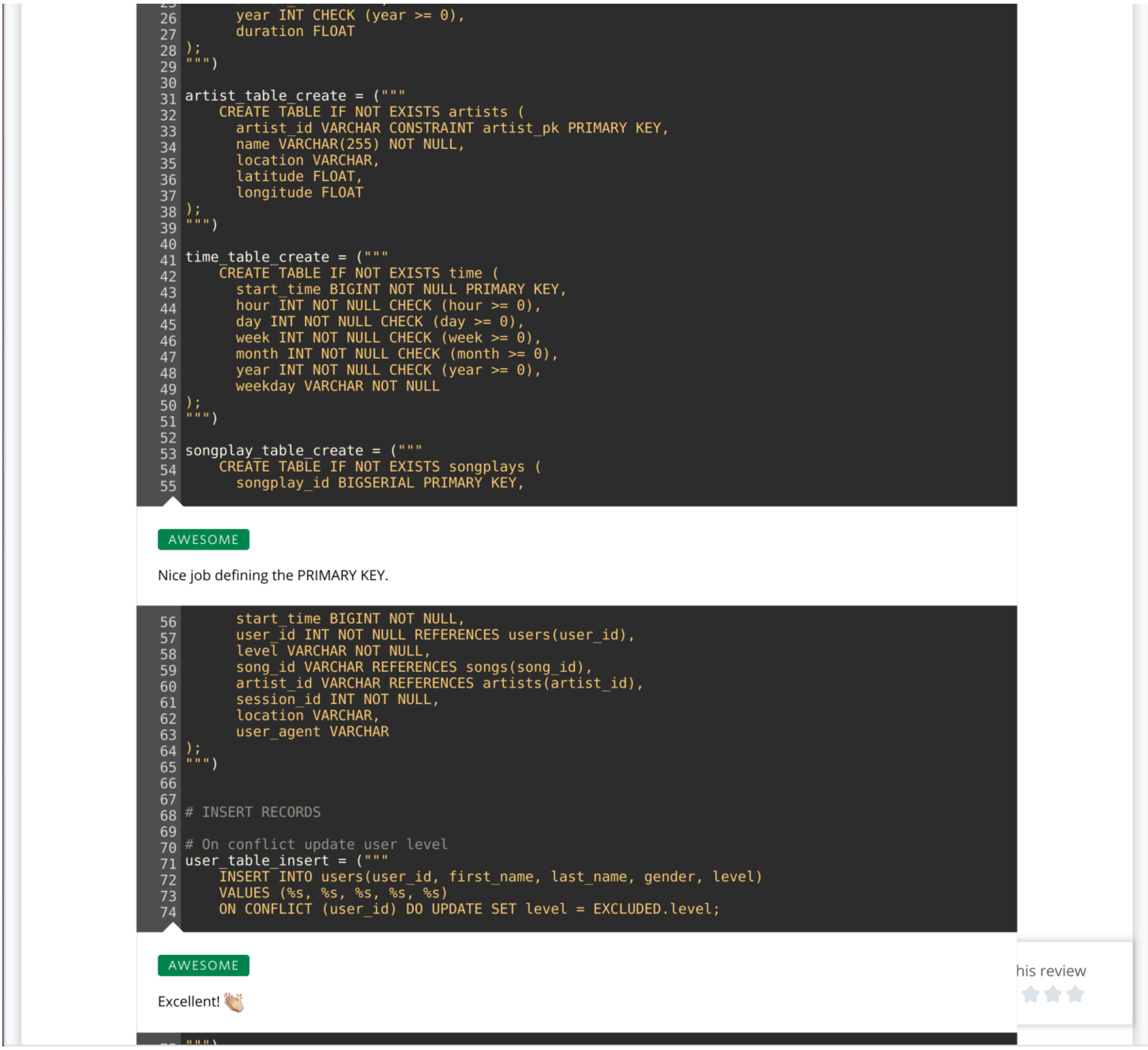
```
# DROP TABLES
```

AWESOME

Perfect use of IF EXISTS.

```

songplay_table_drop = "DROP TABLE IF EXISTS songplays;"
user_table_drop = "DROP TABLE IF EXISTS users;"
song_table_drop = "DROP TABLE IF EXISTS songs;"
artist_table_drop = "DROP TABLE IF EXISTS artists;"
time_table_drop = "DROP TABLE IF EXISTS time;"

# CREATE TABLES

user_table_create = ("""
    CREATE TABLE IF NOT EXISTS users (
      user_id INT PRIMARY KEY,
      first_name VARCHAR,
      last_name VARCHAR,
      gender VARCHAR,
      level VARCHAR
);
""")

song_table_create = ("""
    CREATE TABLE IF NOT EXISTS songs (
      song_id VARCHAR NOT NULL PRIMARY KEY,
      title VARCHAR(255),
      artist_id VARCHAR,
      year INT CHECK (year >= 0),
```

Rate this review ☆☆☆☆☆



```
      year INT CHECK (year >= 0),
      duration FLOAT
);
""")

artist_table_create = ("""
    CREATE TABLE IF NOT EXISTS artists (
      artist_id VARCHAR CONSTRAINT artist_pk PRIMARY KEY,
      name VARCHAR(255) NOT NULL,
      location VARCHAR,
      latitude FLOAT,
      longitude FLOAT
);
""")

time_table_create = ("""
    CREATE TABLE IF NOT EXISTS time (
      start_time BIGINT NOT NULL PRIMARY KEY,
      hour INT NOT NULL CHECK (hour >= 0),
      day INT NOT NULL CHECK (day >= 0),
      week INT NOT NULL CHECK (week >= 0),
      month INT NOT NULL CHECK (month >= 0),
      year INT NOT NULL CHECK (year >= 0),
      weekday VARCHAR NOT NULL
);
""")

songplay_table_create = ("""
    CREATE TABLE IF NOT EXISTS songplays (
      songplay_id BIGSERIAL PRIMARY KEY,
```

AWESOME

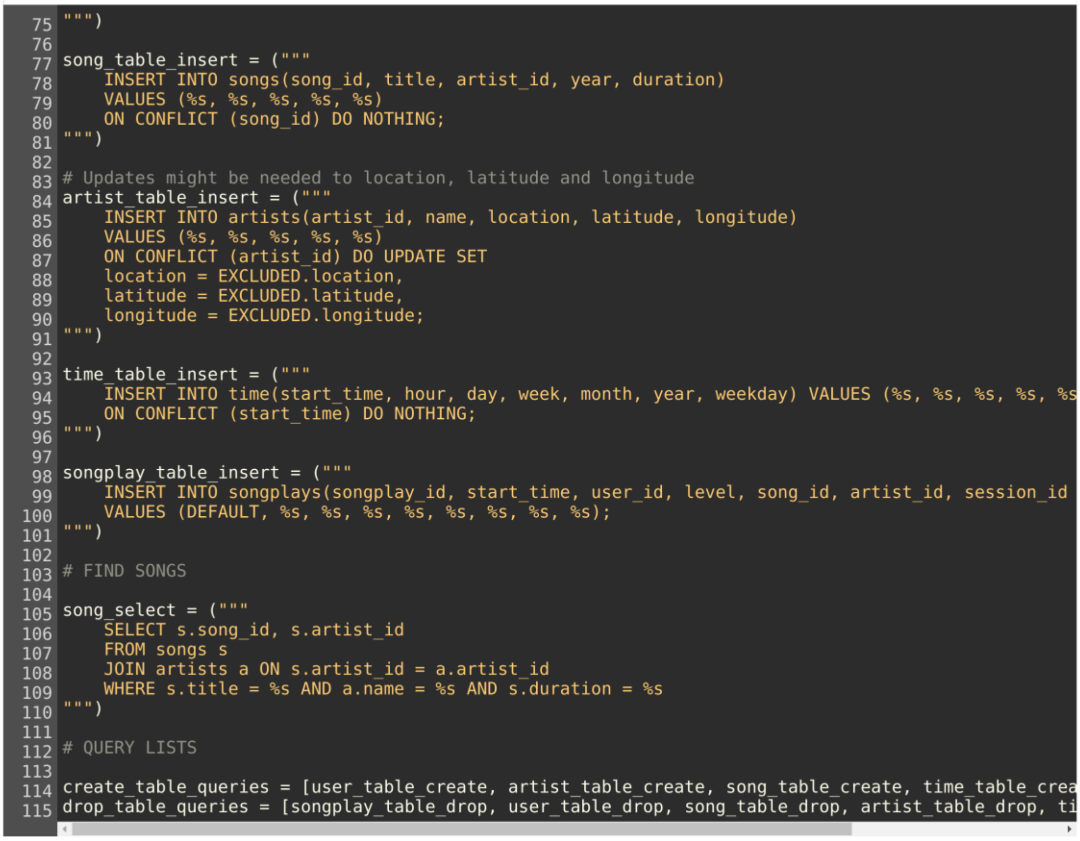
Nice job defining the PRIMARY KEY.

```
      start_time BIGINT NOT NULL,
      user_id INT NOT NULL REFERENCES users(user_id),
      level VARCHAR NOT NULL,
      song_id VARCHAR REFERENCES songs(song_id),
      artist_id VARCHAR REFERENCES artists(artist_id),
      session_id INT NOT NULL,
      location VARCHAR,
      user_agent VARCHAR
);
""")


# INSERT RECORDS

# On conflict update user level
user_table_insert = ("""
    INSERT INTO users(user_id, first_name, last_name, gender, level)
    VALUES (%s, %s, %s, %s, %s)
    ON CONFLICT (user_id) DO UPDATE SET level = EXCLUDED.level;
```

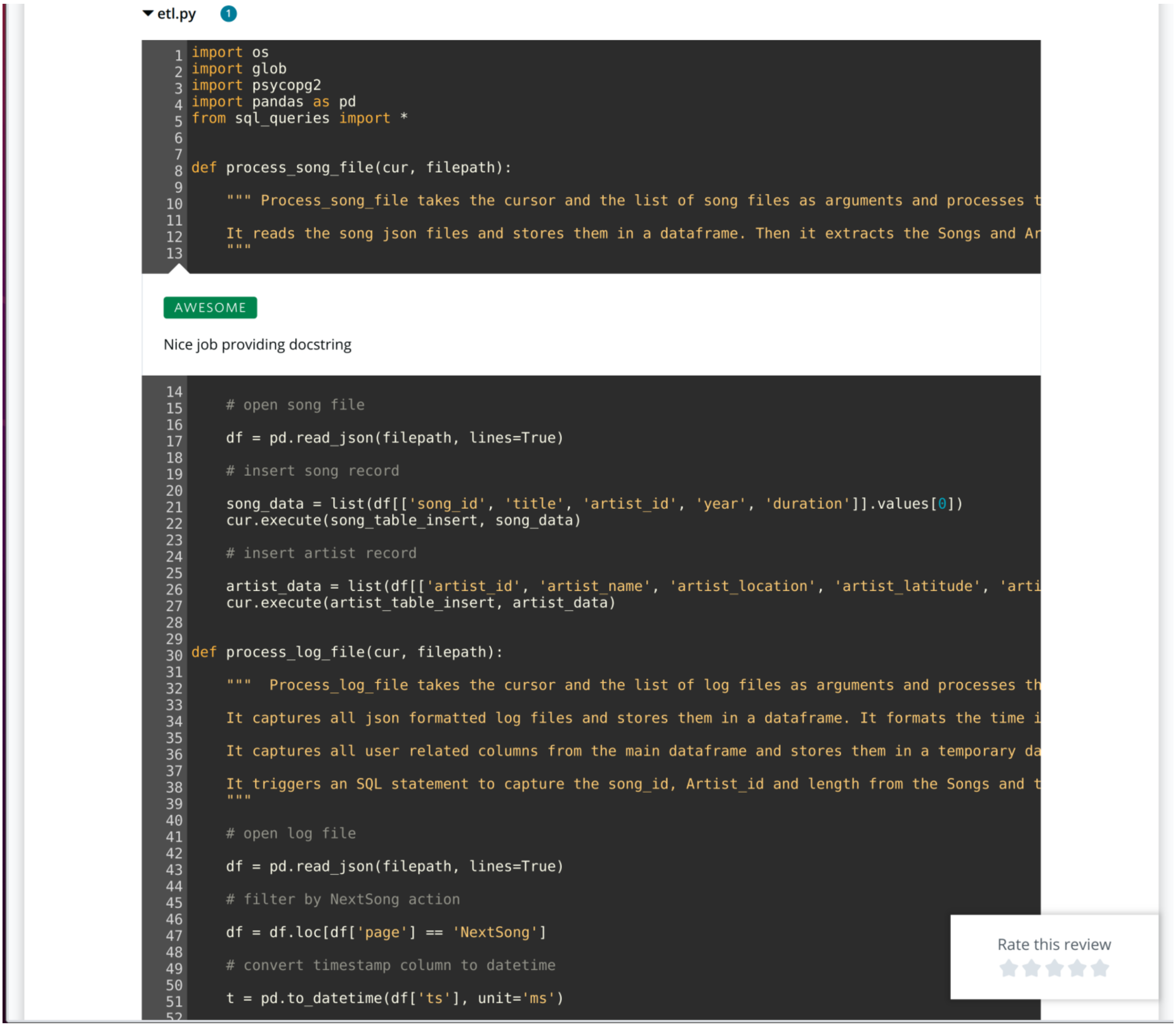
AWESOME

Excellent! 👏

```
""")
```



```
""")

song_table_insert = ("""
    INSERT INTO songs(song_id, title, artist_id, year, duration)
    VALUES (%s, %s, %s, %s, %s)
    ON CONFLICT (song_id) DO NOTHING;
""")

# Updates might be needed to location, latitude and longitude
artist_table_insert = ("""
    INSERT INTO artists(artist_id, name, location, latitude, longitude)
    VALUES (%s, %s, %s, %s, %s)
    ON CONFLICT (artist_id) DO UPDATE SET
    location = EXCLUDED.location,
    latitude = EXCLUDED.latitude,
    longitude = EXCLUDED.longitude;
""")

time_table_insert = ("""
    INSERT INTO time(start_time, hour, day, week, month, year, weekday) VALUES (%s, %s, %s, %s, %s
    ON CONFLICT (start_time) DO NOTHING;
""")

songplay_table_insert = ("""
    INSERT INTO songplays(songplay_id, start_time, user_id, level, song_id, artist_id, session_id
    VALUES (DEFAULT, %s, %s, %s, %s, %s, %s, %s, %s);
""")

# FIND SONGS

song_select = ("""
    SELECT s.song_id, s.artist_id
    FROM songs s
    JOIN artists a ON s.artist_id = a.artist_id
    WHERE s.title = %s AND a.name = %s AND s.duration = %s
""")

# QUERY LISTS

create_table_queries = [user_table_create, artist_table_create, song_table_create, time_table_crea
drop_table_queries = [songplay_table_drop, user_table_drop, song_table_drop, artist_table_drop, ti
```



etl.py 1

```
import os
import glob
import psycopg2
import pandas as pd
from sql_queries import *


def process_song_file(cur, filepath):

    """ Process_song_file takes the cursor and the list of song files as arguments and processes t

    It reads the song json files and stores them in a dataframe. Then it extracts the Songs and Ar
    """
```

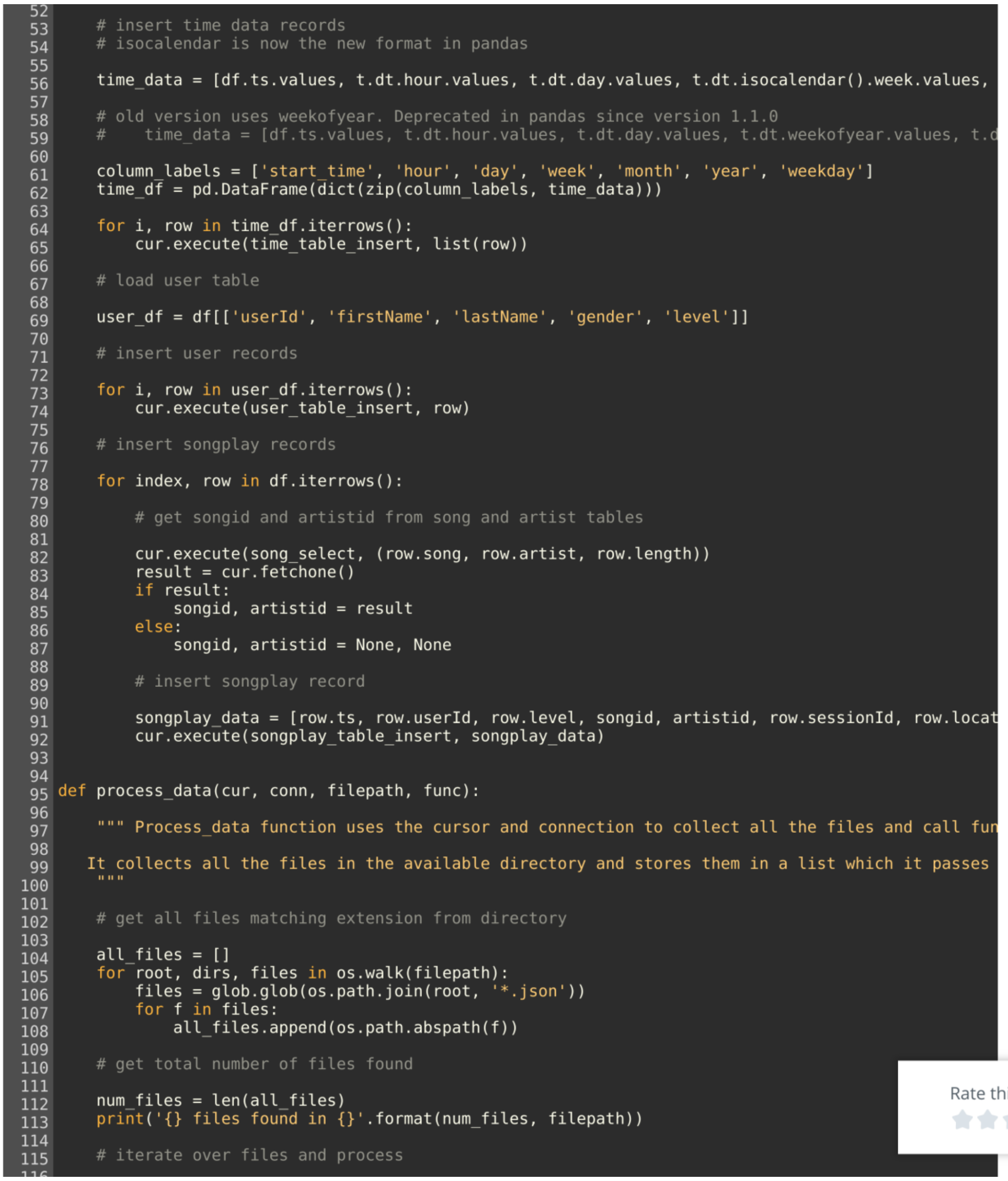
AWESOME

Nice job providing docstring

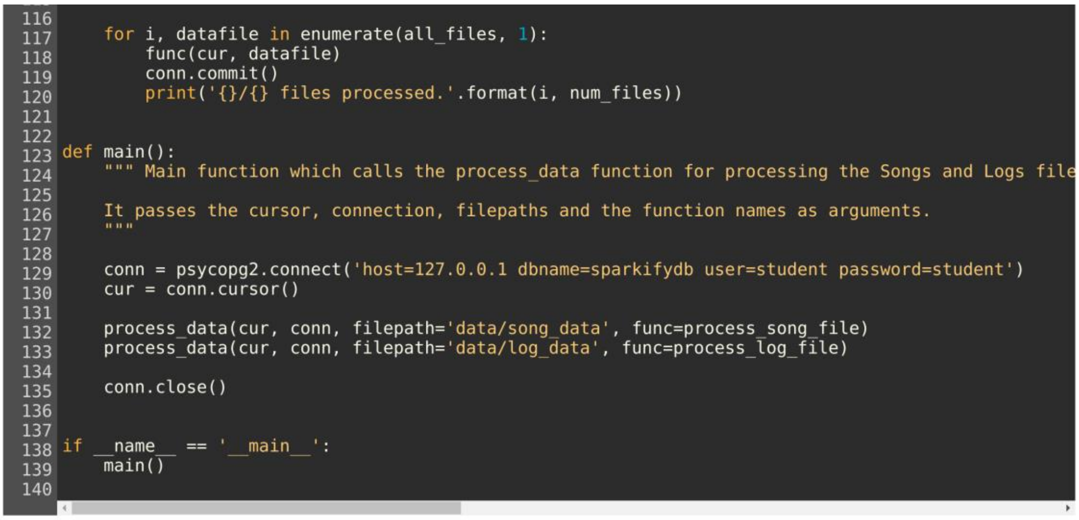
```

    # open song file

    df = pd.read_json(filepath, lines=True)

    # insert song record

    song_data = list(df[['song_id', 'title', 'artist_id', 'year', 'duration']].values[0])
    cur.execute(song_table_insert, song_data)

    # insert artist record

    artist_data = list(df[['artist_id', 'artist_name', 'artist_location', 'artist_latitude', 'arti
    cur.execute(artist_table_insert, artist_data)


def process_log_file(cur, filepath):

    """  Process_log_file takes the cursor and the list of log files as arguments and processes th

    It captures all json formatted log files and stores them in a dataframe. It formats the time i

    It captures all user related columns from the main dataframe and stores them in a temporary da

    It triggers an SQL statement to capture the song_id, Artist_id and length from the Songs and t
    """

    # open log file

    df = pd.read_json(filepath, lines=True)

    # filter by NextSong action

    df = df.loc[df['page'] == 'NextSong']

    # convert timestamp column to datetime

    t = pd.to_datetime(df['ts'], unit='ms')
```

Rate this review

☆☆☆☆☆



```
    # insert time data records
    # isocalendar is now the new format in pandas

    time_data = [df.ts.values, t.dt.hour.values, t.dt.day.values, t.dt.isocalendar().week.values,

    # old version uses weekofyear. Deprecated in pandas since version 1.1.0
    #    time_data = [df.ts.values, t.dt.hour.values, t.dt.day.values, t.dt.weekofyear.values, t.d

    column_labels = ['start_time', 'hour', 'day', 'week', 'month', 'year', 'weekday']
    time_df = pd.DataFrame(dict(zip(column_labels, time_data)))

    for i, row in time_df.iterrows():
        cur.execute(time_table_insert, list(row))

    # load user table

    user_df = df[['userId', 'firstName', 'lastName', 'gender', 'level']]

    # insert user records

    for i, row in user_df.iterrows():
        cur.execute(user_table_insert, row)

    # insert songplay records

    for index, row in df.iterrows():

        # get songid and artistid from song and artist tables

        cur.execute(song_select, (row.song, row.artist, row.length))
        result = cur.fetchone()
        if result:
            songid, artistid = result
        else:
            songid, artistid = None, None

        # insert songplay record

        songplay_data = [row.ts, row.userId, row.level, songid, artistid, row.sessionId, row.locat
        cur.execute(songplay_table_insert, songplay_data)


def process_data(cur, conn, filepath, func):

    """ Process_data function uses the cursor and connection to collect all the files and call fun

   It collects all the files in the available directory and stores them in a list which it passes
    """

    # get all files matching extension from directory

    all_files = []
    for root, dirs, files in os.walk(filepath):
        files = glob.glob(os.path.join(root, '*.json'))
        for f in files:
            all_files.append(os.path.abspath(f))

    # get total number of files found

    num_files = len(all_files)
    print('{} files found in {}'.format(num_files, filepath))

    # iterate over files and process
```



```
    for i, datafile in enumerate(all_files, 1):
        func(cur, datafile)
        conn.commit()
        print('{}/{} files processed.'.format(i, num_files))


def main():
    """ Main function which calls the process_data function for processing the Songs and Logs file

    It passes the cursor, connection, filepaths and the function names as arguments.
    """

    conn = psycopg2.connect('host=127.0.0.1 dbname=sparkifydb user=student password=student')
    cur = conn.cursor()

    process_data(cur, conn, filepath='data/song_data', func=process_song_file)
    process_data(cur, conn, filepath='data/log_data', func=process_log_file)

    conn.close()


if __name__ == '__main__':
    main()
```