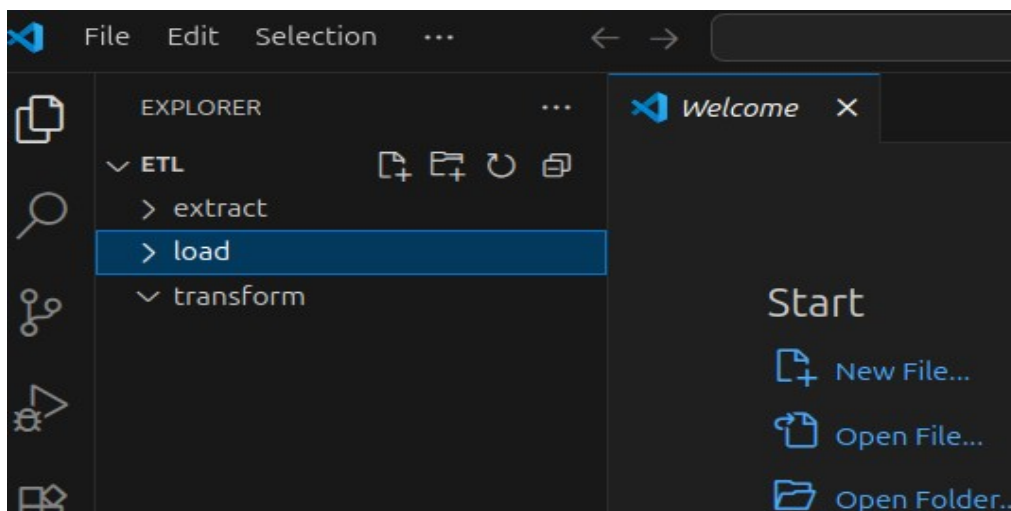


We will work on creating an ETL pipeline this week.

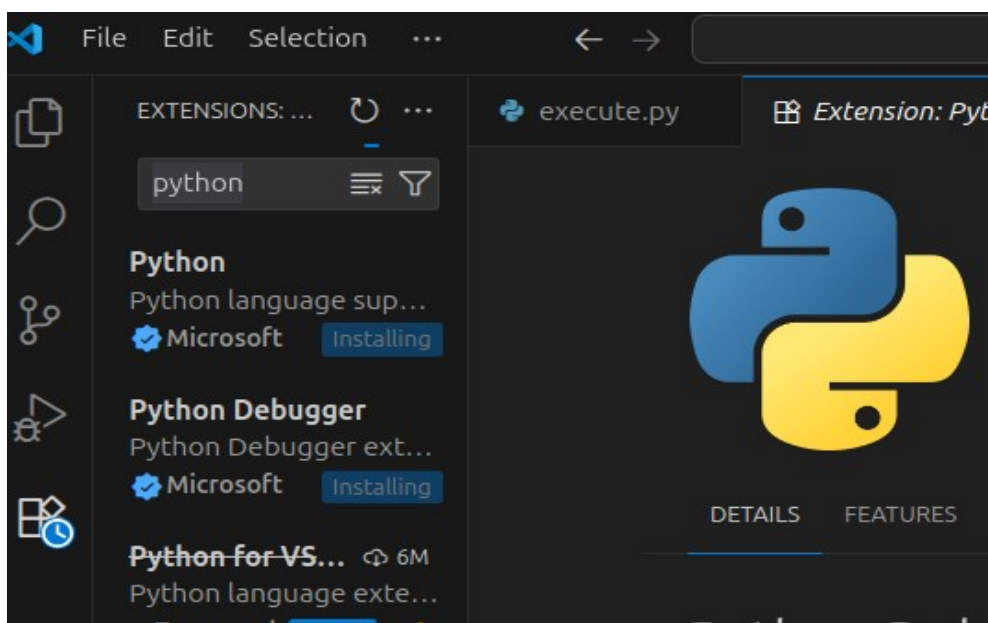
1. Create a new directory named ETL, cd into that directory, and open vscode in that directory. You can do this using GUI as well.

```
ardent@ardent:~/Workspace$ mkdir etl
ardent@ardent:~/Workspace$ cd etl
ardent@ardent:~/Workspace/etl$ code .
ardent@ardent:~/Workspace/etl$
```

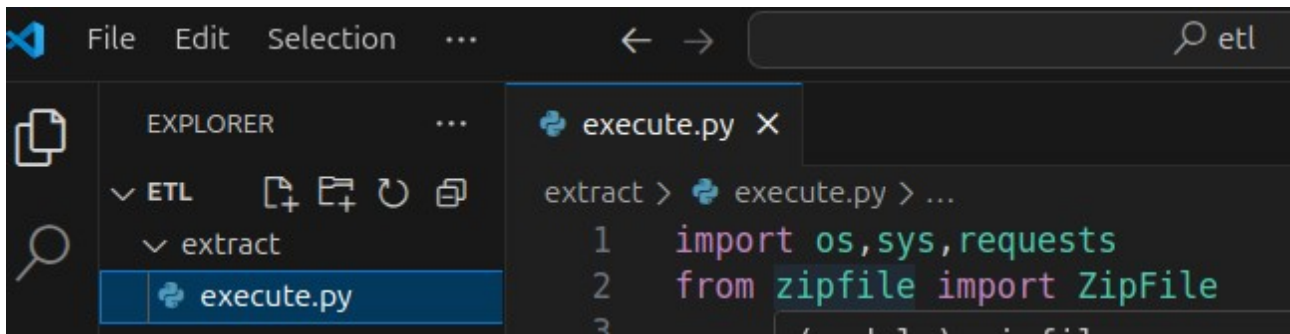
2. In vscode create 3 new folders extract, transform, and load. We will be writing code for extraction, transformation, and loading in their respective folder.



3. First let us install extensions that relate to python to make our job easier. Go to extension tab on the left → Search for python → Install two extensions



4. In the extract folder create new python file execute.py and import required libraries at the top of file



We will be using os module to interact directly with the operating system. It will be used for tasks like creating new directories

We will be using sys module to get the argument passed during execution of the program

We will be using requests library to hit the kaggle url and download the file

We will be using zipfile module to extract the content from downloaded zip file

5. Let us define a function that will take the url of the file to download, and the full path of directory to download it to

```
from zipfile import ZipFile

def download_zip_file(url, output_dir):
    response = requests.get(url, stream=True)
    os.makedirs(output_dir, exist_ok=True)
    if response.status_code == 200:
        filename = os.path.join(output_dir, "downloaded.zip")
        with open(filename, "wb") as f:
            for chunk in response.iter_content(chunk_size=8192):
                if chunk:
                    f.write(chunk)
        print(f"Downloaded zip file : {filename}")
        return filename
    else:
        raise Exception(f"Failed to download file: Status code {response.status_code}")
```

Initial we use requests library to get the response from the url. Stream is set to true so that we could read the response in chunk.

We use os module to create the output directory if it does not exists

If the url returned an OK(200) response code we write the content to a file, if not an exception is raised

This will return the filename of the download zip filename

6. Create new function to extract files from the downloaded zip filename

```
17
18 def extract_zip_file(zip_filename, output_dir):
19
20     with ZipFile(zip_filename, "r") as zip_file:
21         zip_file.extractall(output_dir)
22
23     print(f"Extracted files written to: {output_dir}")
24     print("Removing the zip file")
25     os.remove(zip_filename)
26
```

This function will take two arguments, First is the name of the zip file returned by previous function and second is the output directory to write the extracted files to.

Here ZipFile module is used to read and extract all files from the downloaded zip file to the output directory. Finally, the initially downloaded zip file is deleted. This was not necessary but is done for cleaning up purpose.

7. Define a function to fix the json file that is not supported by pyspark.

Pyspark either supports new line delimited or ndjson files or json file having array of dictionary. However, our file only contains one dictionary object so we need to convert it to format pyspark supports

```
27 def fix_json_dict(output_dir):
28     import json
29     file_path = os.path.join(output_dir, "dict_artists.json")
30     with open(file_path, "r") as f:
31         data = json.load(f)
32
33     with open(os.path.join(output_dir, "fixed_da.json"), "w", encoding="utf-8") as f_out:
34         for key, value in data.items():
35             record = {"id": key, "related_ids": value}
36             json.dump(record, f_out, ensure_ascii=False)
37             f_out.write("\n")
38     print(f"File {file_path} has been fixed and written to {output_dir} as fixed_da.json")
39     print("Removing the original file")
40     os.remove(file_path)
41
```

Initially, we import the json module. Notice how it has been imported inside the function and its access will be limited to that function. We could have done same of ZipFile module. There is no performance gain to importing this way but might help avoid some bugs by maintaining proper scoping of accessibility.

Initially, we read load the json file into a variable as a dictionary.

Next, we open a new file to write, which will have the content in a ndjson format.

Then we iterate over the loaded json content and assign key value pair to each record and write out a new line character after each record..

Finally, the initial json file is removed. In this case it might even be a bad idea to remove the original file as it could provide some context but ultimately it is a design choice at the hand of the engineer.

8. Implement the initial/startup logic which is ran when the python file (extract module) is executed

```
41
42 if __name__ == "__main__":
43     if len(sys.argv) < 2:
44         print("Extraction path is required")
45         print("Exame Usage:")
46         print("python3 execute.py /home/ardent-sharma/Data/Extraction")
47     else:
48         try:
49             print("Starting Extraction Engine...")
50             EXTRACT_PATH = sys.argv[1]
51             KAGGLE_URL = "https://storage.googleapis.com/kaggle-data-sets/1993933/3294812/bund
52             zip_filename = download_zip_file(KAGGLE_URL, EXTRACT_PATH)
53             extract_zip_file(zip_filename, EXTRACT_PATH)
54             fix_json_dict(EXTRACT_PATH)
55             print("Extraction Sucessfully Completed!!!")
56         except Exception as e:
57             print(f"Error: {e}")
```

This will be entry point of the program. The `__name__` is a dunder (double underscore) variable assigned during runtime to generally be name of the file. However, if the file at hand is the file being executed then the `__name__` variable is set to be string `"__main__"`. This might be confusing at first but basically the python file being executed will have the `__name__` variable set to `__main__` and during run time we check for that to decide whether to run the program or not.

Initially, we check if at least one argument has been provided which is expected to be the path to the directory to write the extracted files to. If the argument is not provided error message is printed and program execution is stopped.

If the argument is passed. We store it as `EXTRACT_PATH`. Also a static `KAGGLE_URL` is kept. This could also be passed as argument but since in our case it is static rather than dynamic we are hard-coding it. You can get this url by going to the browser and downloading the dataset and during download right clicking and selecting copy link. We could use Kaggle API to handle these cases but it is beyond the scope of this course so it is avoided.

Finally, all 3 functions we defined before are called one after the other. This is kept in try except block to properly handle error and stop the execution of program.

9. After everything is setup, let us try to execute the program.

If the program is executed without passing any arguments an error message is shown as below.

```

ardent@ardent:~/Workspace/etl$ python3 extract/execute.py
Extraction path is required
Exame Usage:
python3 execute.py /home/ardent-sharma/Data/Extraction
ardent@ardent:~/Workspace/etl$

```

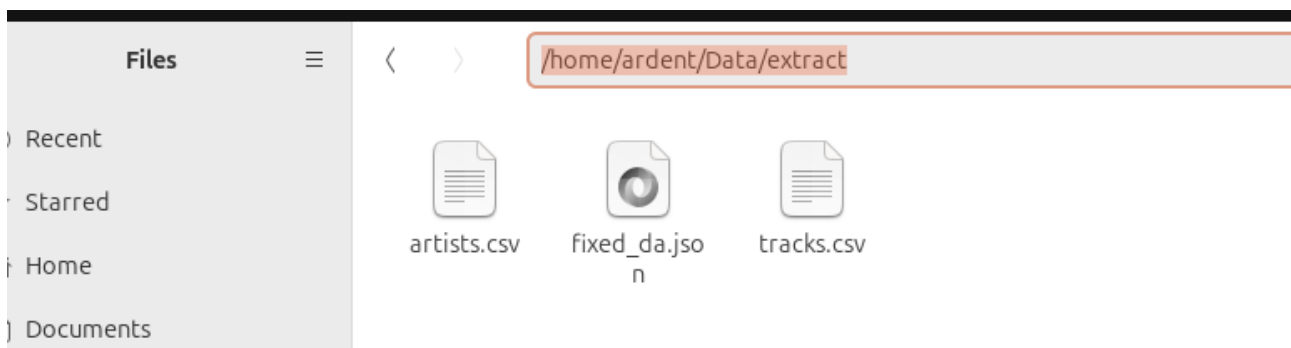
Now, run the program by passing the full path to a directory you want to save the extracted files to

```

Exame Usage:
python3 execute.py /home/ardent-sharma/Data/Extraction
ardent@ardent:~/Workspace/etl$ python3 extract/execute.py /home/ardent/Data/extract
Starting Extraction Engine...
Downloaded zip file : /home/ardent/Data/extract/downloaded.zip
Extracted files written to: /home/ardent/Data/extract
Removing the zip file
File /home/ardent/Data/extract/dict_artists.json has been fixed and written to /home/ardent/Data/extract/fixed_da.json
Removing the original file
Extraction Sucessfully Completed!!!
ardent@ardent:~/Workspace/etl$

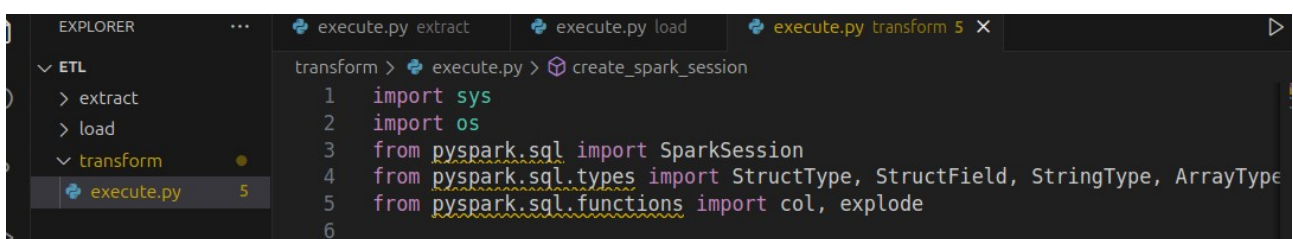
```

Go to the output directory and verify that files have been created. There must be 3 files: artists.csv, fixed_da.json, tracks.csv

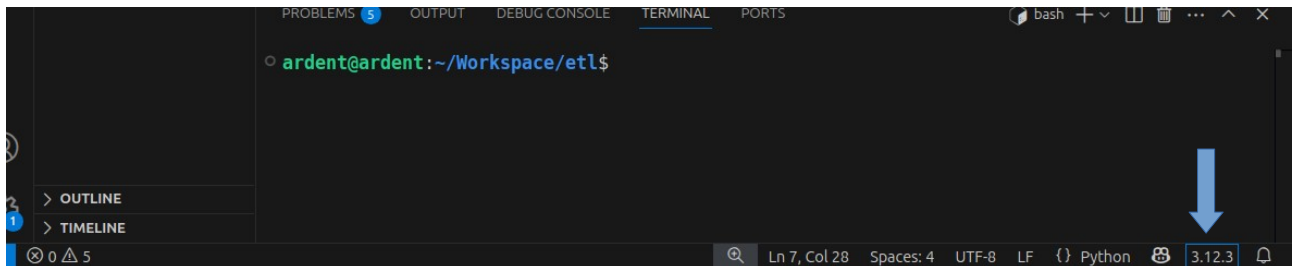


With that the process of extraction has been completed. Now we will move on to transformation:

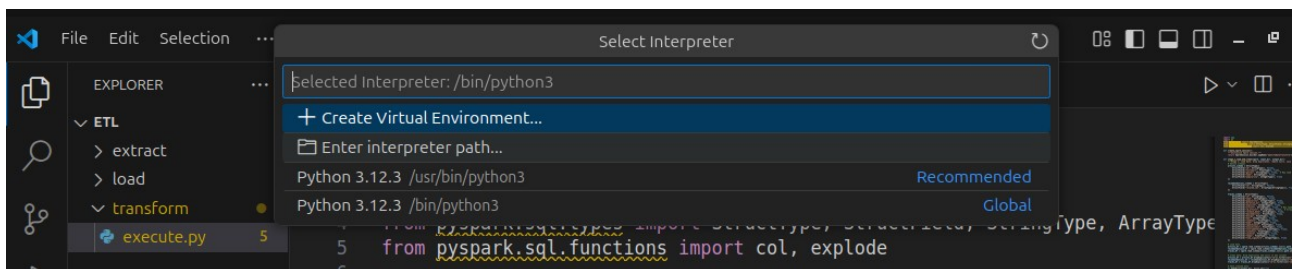
1. Create new file execute.py inside the transform directory. In the execute.py file let us add the required imports.



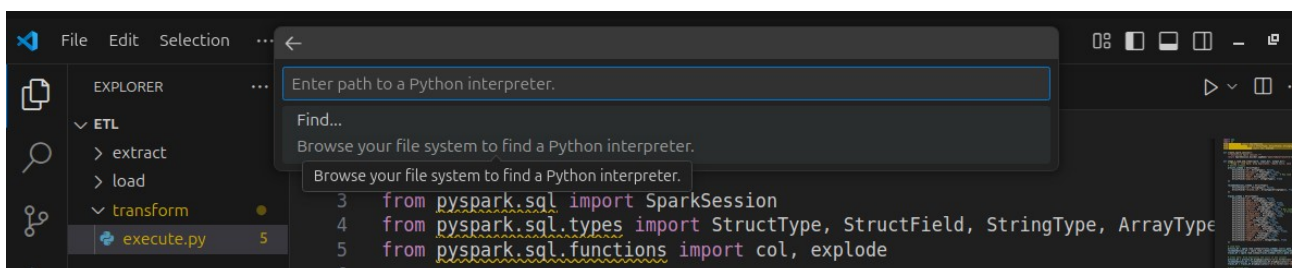
Initially, we can see that pyspark related imports have yellow squiggly lines. This means that the currently selected interpreter cannot find the said packages. Let us change the interpreter to be the virtual environment we have created.



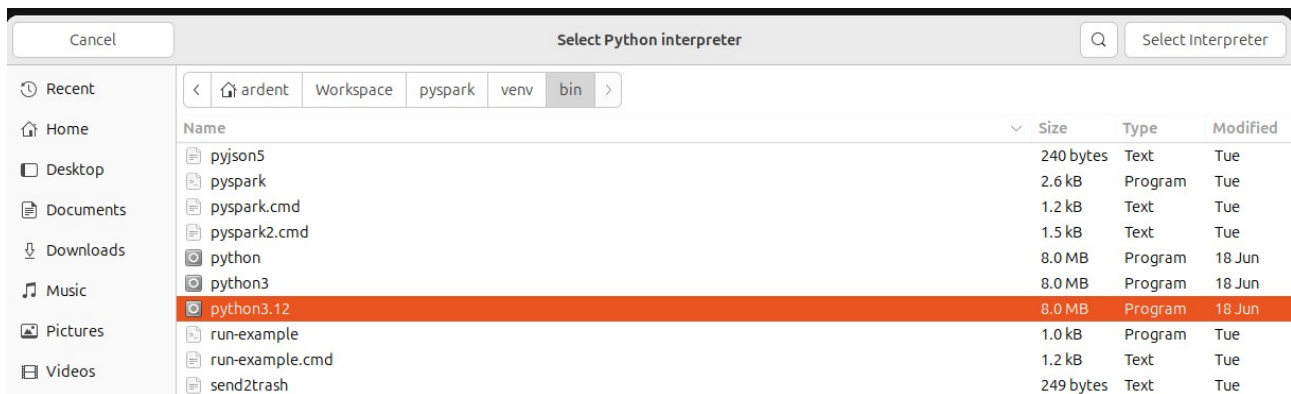
You can find the python interpreter at the bottom right which if you click an dialog will show up where you can press the Enter interpreter path option. Optionally, you can also press CTRL+SHIFT+P and on the search box enter “select interpreter”.



Select Enter interpreter path and then select Find, system browser will open up then



Browse to the binary of virtual environment and select the python3.12 inside it. This will be our new interpreter. If you have been following the manual. Your path also must be `~/Workspace/pyspark/venv/bin/python3.12` Here `~` is your home directory



Now that you have selected the proper interpreter. If you then close the file and re-open it, the interpreter can recognize the pyspark packages as well.

```
import sys
import os
from pyspark.sql import SparkSession
from pyspark.sql import types as T
from pyspark.sql import functions as F
```

2. Create a function that return a spark session.

```
def create_spark_session():
    """Initialize Spark session."""
    return SparkSession.builder.appName("SpotifyDataTransform").getOrCreate()
```

SparkSession can take multiple different input for config, cores to use and so on. For now, we are setting everything to default (by not passing anything) and either getting or creating a session and returning it directly. getOrCreate method will create session if the session with the given name does not exist and return the preexisting session if it exists.

3. Define function that loads the extracted data perform some cleaning and write it as a parquet file.

```
transform > execute.py > load_and_clean > spark
14
15
16 def load_and_clean(spark, input_dir, output_dir):
17     """Stage 1: Load data, drop duplicates, remove nulls, save cleaned data."""
18     # Define schemas
19     artists_schema = T.StructType([
20         T.StructField("id", T.StringType(), False),
21         T.StructField("followers", T.FloatType(), True),
22         T.StructField("genres", T.StringType(), True),
23         T.StructField("name", T.StringType(), True),
24         T.StructField("popularity", T.IntegerType(), True)
25     ])
26
27     recommendations_schema = T.StructType([
```

The function will take spark-session, and the path to input and output directory as its parameter

Initially, we define schema for all the files. If we do not define then spark will try to infer by self, which might be correct or incorrect depending on different reasons. Also, the schema provided below might not be the most optimal so use your own judgment in your implementation.

```
23         T.StructField("name", T.StringType(), True),
24         T.StructField("popularity", T.IntegerType(), True)
25     ])
26
27     recommendations_schema = T.StructType([
28         T.StructField("id", T.StringType(), False),
29         T.StructField("related_ids", T.ArrayType(T.StringType()), True)
30     ])
31
32     tracks_schema = T.StructType([
33         T.StructField("id", T.StringType(), False),
34         T.StructField("name", T.StringType(), True),
35         T.StructField("popularity", T.IntegerType(), True),
36         T.StructField("duration_ms", T.IntegerType(), True),
```



```
tracks_schema = T.StructType([
    T.StructField("id", T.StringType(), False),
    T.StructField("name", T.StringType(), True),
    T.StructField("popularity", T.IntegerType(), True),
    T.StructField("duration_ms", T.IntegerType(), True),
    T.StructField("explicit", T.IntegerType(), True),
    T.StructField("artists", T.StringType(), True),
    T.StructField("id_artists", T.StringType(), True),
    T.StructField("release_date", T.StringType(), True),
    T.StructField("danceability", T.FloatType(), True),
    T.StructField("energy", T.FloatType(), True),
    T.StructField("key", T.IntegerType(), True),
    T.StructField("loudness", T.FloatType(), True),
    T.StructField("mode", T.IntegerType(), True),
    T.StructField("speechiness", T.FloatType(), True),
    T.StructField("acousticness", T.FloatType(), True),
    T.StructField("instrumentalness", T.FloatType(), True),
    T.StructField("liveness", T.FloatType(), True),
    T.StructField("valence", T.FloatType(), True),
    T.StructField("tempo", T.FloatType(), True),
    T.StructField("time_signature", T.IntegerType(), True)
])
```

Then we load all the files extracted from our extraction phase in pyspark as their individual data frames

```
# Load data
artists_df = spark.read.schema(artists_schema).csv(os.path.join(input_dir, "artists.csv"), header=True)
recommendations_df = spark.read.schema(recommendations_schema).json(os.path.join(input_dir, "fixed_da.json"))
tracks_df = spark.read.schema(tracks_schema).csv(os.path.join(input_dir, "tracks.csv"), header=True)
```

Then we drop any duplicates on the id key which is the primary key. Then we filter for data that do not have id. Multiple other cleanings can be performed. Perform them on your own.

```
59
60 artists_df = artists_df.dropDuplicates(["id"]).filter(F.col("id").isNotNull())
61 recommendations_df = recommendations_df.dropDuplicates(["id"]).filter(F.col("id").isNotNull())
62 tracks_df = tracks_df.dropDuplicates(["id"]).filter(F.col("id").isNotNull())
63
```

Finally, write the processed dataframes to a file in parquet format. Parquet is used because it is most optimized and implements efficient compression techniques.

```
63
64 artists_df.write.mode("overwrite").parquet(os.path.join(output_dir, "stage1", "artists"))
65 recommendations_df.write.mode("overwrite").parquet(os.path.join(output_dir, "stage1", "recommendations"))
66 tracks_df.write.mode("overwrite").parquet(os.path.join(output_dir, "stage1", "tracks"))
67
68 print("Stage 1: Cleaned data saved")
69 return artists_df, recommendations_df, tracks_df
```

4. Create function that takes the returned dataframe from previous function as input and creates a master table out of it. Master table generally contain all the data but is used for recording purposes rather than querying.

```
def create_master_table(output_dir, artists_df, recommendations_df, tracks_df):
    """Stage 2: Create master table by joining artists, tracks, and recommendations."""

    tracks_df = tracks_df.withColumn("id_artists_array", F.from_json(F.col("id_artists"), F.ArrayType(F.StringType())))

    tracks_exploded = tracks_df.select("id", "name", "popularity", "id_artists_array").withColumn("artist_id", F.explode("id_artists_array"))

    # Join tracks with artists
```

Convert id_artists column to array and then explode it to flatten the array items into multiple single row values

```
8 # Join tracks with artists
9 master_df = tracks_exploded.join(artists_df, tracks_exploded.artist_id == artists_df.id, "left") \
10     .select(
11         tracks_exploded.id.alias("track_id"),
12         tracks_exploded.name.alias("track_name"),
13         tracks_exploded.popularity.alias("track_popularity"),
14         artists_df.id.alias("artist_id"),
15         artists_df.name.alias("artist_name"),
16         artists_df.followers,
17         artists_df.genres,
18         artists_df.popularity.alias("artist_popularity")
19     )
```

Join the exploded tracks data frame to artists data frame. Left join is used so that we do not loose track information even though the artist information related to it might not exist.

```
# Join with recommendations
master_df = master_df.join(recommendations_df, master_df.artist_id == recommendations_df.id, "left") \
    .select(
        master_df.track_id,
        master_df.track_name,
        master_df.track_popularity,
        master_df.artist_id,
        master_df.artist_name,
        master_df.followers,
        master_df.genres,
        master_df.artist_popularity,
        recommendations_df.related_ids
    )
```

Again, recommendation data frame is joined using left join for similar reason as before.

```
94 # Save master table
95 master_df.write.mode("overwrite").parquet(os.path.join(output_dir, "stage2", "master_table"))
96 print("Stage 2: Master table saved")
97
98
99
```

Finally, the dataframe is written as master table to a new directory in parquet format.

5. Create third and final function that will be used to create new datasets with useful information. This function will take output directory and the 3 data frames which we got as an output of first functions

```
def create_query_tables(output_dir, artists_df, recommendations_df, tracks_df):  
    """Stage 3: Create query-optimized tables."""  
  
    recommendations_exploded = recommendations_df.withColumn("related_id", F.explode("related_ids")) \  
        .select("id", "related_id")  
    recommendations_exploded.write.mode("overwrite").parquet(os.path.join(output_dir, "stage3", "recommendations_exploded"))
```

The first thing we do is explode the recommendation data frame and write it to a new directory.

```
16  
17 tracks_exploded = tracks_df.withColumn("id_artists_array", F.from_json(F.col("id_artists"), T.ArrayType(T.StringT  
18     .withColumn("artist_id", F.explode("id_artists_array")) \  
19     .select("id", "artist_id")  
20 tracks_exploded.write.mode("overwrite").parquet(os.path.join(output_dir, "stage3", "artist_track"))  
21
```

Then we gather the track id as well as the artist_id for that track and write it to a directory.

```
121  
122 tracks_metadata = tracks_df.select(  
123     "id", "name", "popularity", "duration_ms", "danceability", "energy", "tempo"  
124 )  
125 tracks_metadata.write.mode("overwrite").parquet(os.path.join(output_dir, "stage3", "track_metadata"))  
126  
127 artists_metadata = artists_df.select("id", "name", "followers", "popularity")  
128 artists_metadata.write.mode("overwrite").parquet(os.path.join(output_dir, "stage3", "artist_metadata"))  
129  
130 print("Stage 3: Query-optimized tables saved")  
131
```

Finally, we create metadata dataframe for both track and artist that contain only information regarding them and write it to a directory as well. All the files are saved in parquet format.

```
131  
132 if __name__ == "__main__":  
133     if len(sys.argv) != 3:  
134         print("Usage: python script.py <input_dir> <output_dir>")  
135         sys.exit(1)  
136  
137     input_dir = sys.argv[1]  
138     output_dir = sys.argv[2]  
139  
140     spark = create_spark_session()  
141  
142     artists_df, recommendations_df, tracks_df = load_and_clean(spark, input_dir, output_dir)  
143     create_master_table(output_dir, artists_df, recommendations_df, tracks_df)  
144     create_query_tables(output_dir, artists_df, recommendations_df, tracks_df)  
145  
146     print("Transformation pipeline completed")  
147
```

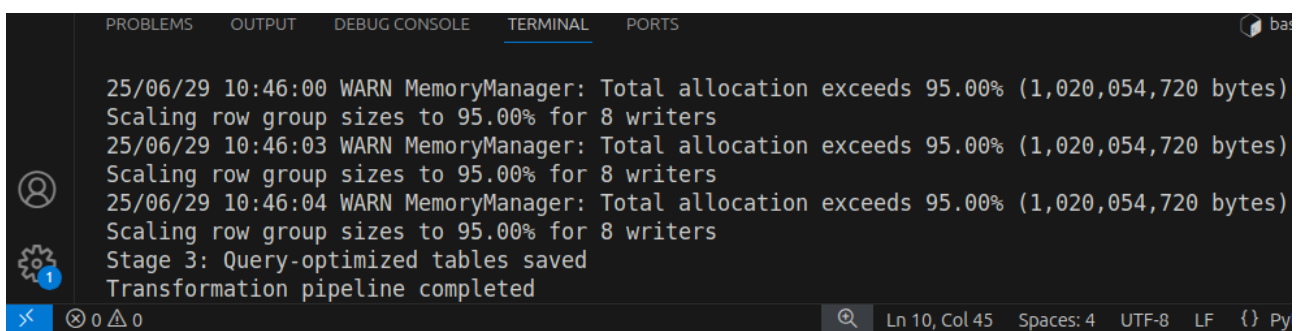
Finally, we write the logic as before which will only be ran when we run the python file directly. We check if the two required parameters have been passed i.e., input_directory and output_directory. Then we call create spark session and call the remaining 3 functions one by one

6. Run the newly created transformation code.

```
Transformation pipeline completed
(venv) ardent@ardent:~/Workspace/etl$ python3 transform/execute.py /home/ardent/Data/extract /home/ardent/Data/transform
```

You have your transformation part ready and all the files must be written to a directory you have specified.

You might get some warning message like below. Almost all of these are config related issue and can be solve by modifying the config.



```
25/06/29 10:46:00 WARN MemoryManager: Total allocation exceeds 95.00% (1,020,054,720 bytes)
Scaling row group sizes to 95.00% for 8 writers
25/06/29 10:46:03 WARN MemoryManager: Total allocation exceeds 95.00% (1,020,054,720 bytes)
Scaling row group sizes to 95.00% for 8 writers
25/06/29 10:46:04 WARN MemoryManager: Total allocation exceeds 95.00% (1,020,054,720 bytes)
Scaling row group sizes to 95.00% for 8 writers
Stage 3: Query-optimized tables saved
Transformation pipeline completed
```

```
7 def create_spark_session():
8     """Initialize Spark session."""
9     return (SparkSession.builder
10            .appName("SpotifyDataTransform")
11            .config("spark.driver.memory", "2g")
12            .config("spark.executor.memory", "4g")
13            .getOrCreate()
14            )
```

There are multiple config inputs for jars, cores, memory, shuffle, partitioning, broadcast and so on and all can be used depending on requirement.

7. Read the files written in stage 1 and stage 2. into the pyspark and shell and view the columns, sample data, perform some query. Verify everything is working as expected. Do this task by self.