Generation
SINGAPORE

SCTP JDE04
ZetaZenith

Interim Project

Netflix Titles (Movies/TV Shows)

Submitted by:

Team Members
SITI RAHAYU BINTE JOHARI
SHUI HUI YAN, TAMMIE SARAH
OOI JUN SHENG

Submission Date: **29/08/2024**

Table Of Content

1. Background

The purpose of this document specifies the overall requirements for the interim project. It explains the outline of the development process within the development environment and defines the scope of the project including problem statement, assumptions, objectives, use cases, the source of data, tools used, methodology for the ETL process, analysis of data, challenges or limitations we have encountered, how we circumvent it and our conclusion.

2. Problem Statement

Netflix, as a leading global streaming service, continuously expands its content library to cater to a diverse and growing audience. To stay ahead of the competition, it is essential for Netflix to understand the distribution of genres within its catalog and how these genres have evolved over time. By analyzing genre trends, Netflix can make informed decisions on content acquisition, production, and recommendation strategies.

3. Assumptions

1. The Netflix dataset consists of popular titles (Movies/TV shows) from 2018 to 2021.
2. There is existing demand for the titles if it is already streaming on Netflix.
3. Missing dates (NaT) found for date_added assumed to be “2018-12-31”.
4. Missing ratings for show_id: 5542, 5795, 5814 assumed to be the same “TV-MA”.

4. Objectives

The primary objective of this report is to analyze the distribution and trends of genres in Netflix's catalog over time. The analysis aims to answer key questions such as:

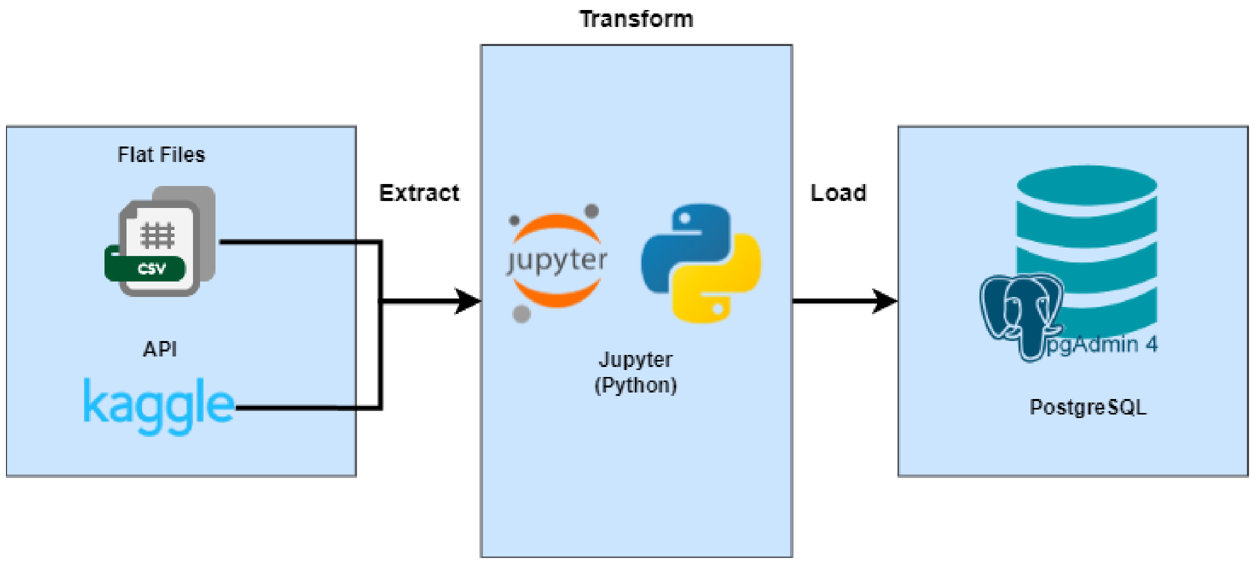
- Which genres have become more or less popular over the years?
- What are the most common genres in Netflix's catalog?
- How do genre preferences vary across different countries?
- Which countries are producing the most diverse set of genres?
- How can these insights inform Netflix’s content strategy, including content acquisition, production, and regional marketing efforts?

5. Use Cases

The Jupyter Notebook will act as the centralized software to develop the extract-transform-load (ETL) process of this entire project.

Python modules will be imported to facilitate in extracting data from external sources, transform the data and finally loading them into PostgreSQL (PgAdmin).

The data architecture model for extract, transform and load (ETL) process:



Data Extract:

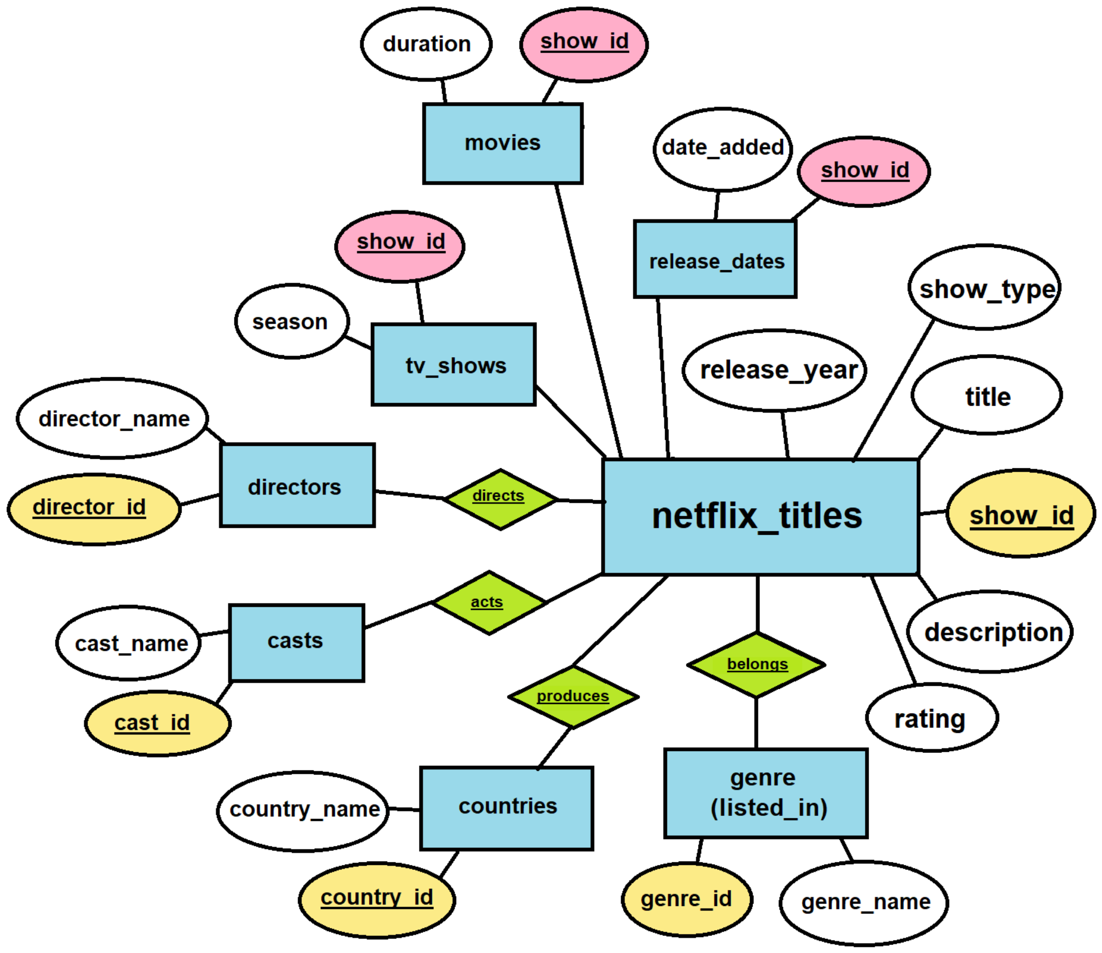
- Data will be extracted from flat files and Kaggle API.
    - Kaggle API is used to extract data sets labelled Netflix shows on Kaggle's repository. The dataset will be in flat file format (.csv).

Data Transform:

- The dataset will then be transformed within the Jupyter Notebook environment, through:
    - Converting it into a pandas DataFrame.
    - Converting it to appropriate data types (dtypes).
    - Replacing "NA", "NaT", NULL and missing values.
    - Checking to ensure all fields are filled correctly.
    - Perform normalization up till 3rd Normal Form.

    - With proper junction tables created for many-to-many (N:M) relationships.
  - Data persistence by exporting .csv as backups once 3rd Normal Form is done and before loading into the database.
- The finalized 3rd Normal Form tables will be loaded into PostgreSQL (PgAdmin4)
  - New database "netflix_db" will be created if it does not exist.
  - All tables (including junction tables) will be created along with the appropriate data type fields, primary and foreign key constraints.
  - Append into "netflix_db" upon loading to 3rd Normalised DataFrames.
  - For data validation, the database will be queried after each creation and loading of table to ensure that the records in DataFrames matches the newly populated tables within "netflix_db".

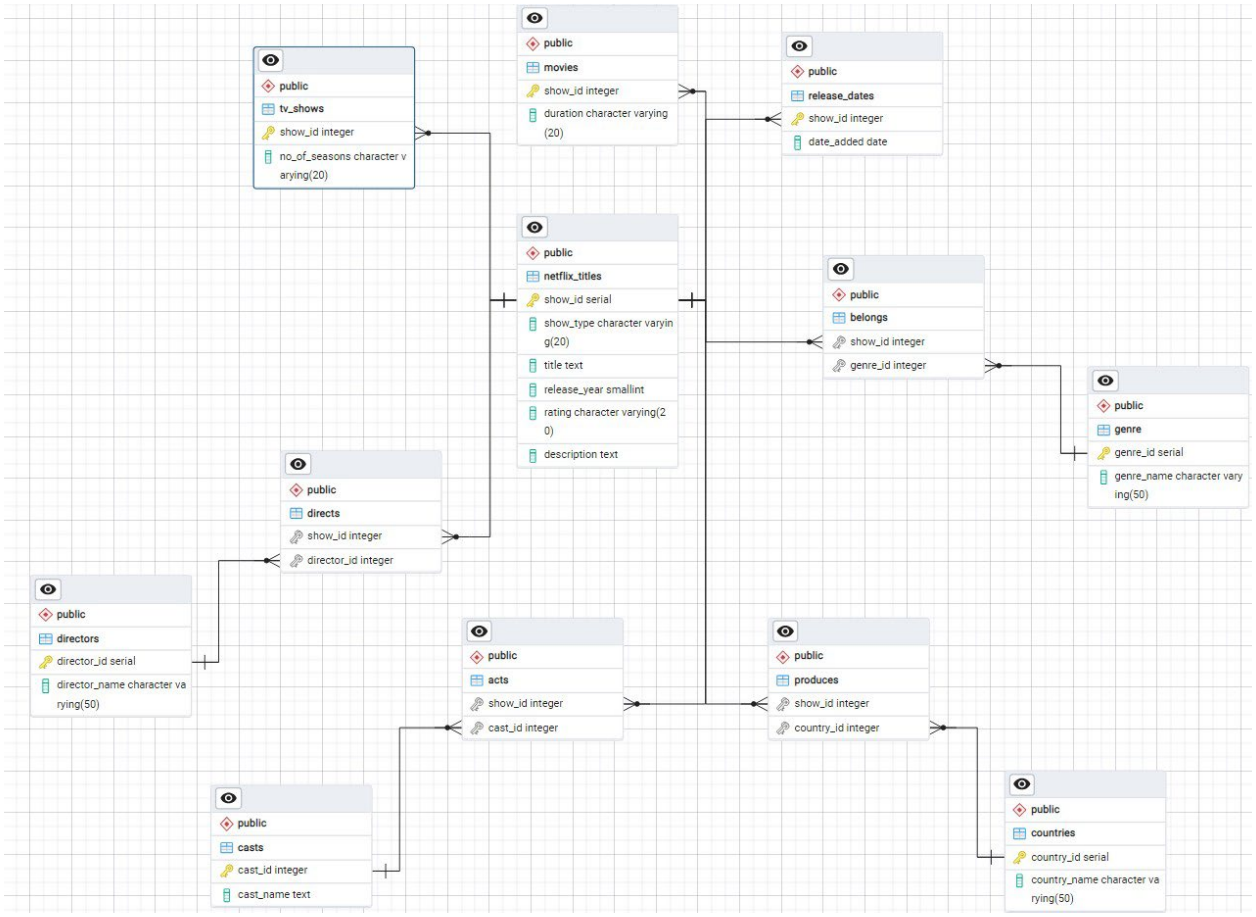
The finalised 3rd Normalised conceptual diagram looks like this:



Data Load:

- The 3rd Normalised DataFrames will be loaded into PostgreSQL within the Jupyter Notebook:
  - Using `SQLAlchemy` and `psycopg2` to perform the following functions:
    - Create database `netflix_db` in PostgreSQL.
    - Create 12 tables within `netflix_db` in PostgreSQL.
    - Set foreign key and primary constraints.
    - Load the 3rd Normalised DataFrames into the 12 tables via Notebook.

And upon loading into PostgreSQL database with referential integrity and constraints, the entity relationship diagram will be autogenerated:



6. Source of Data

The data we have decided on will be "Netflix Movies and TV Shows" published by Shivam Bansal on Kaggle. It was last updated 3 years ago but has popular titles from movies and tv shows, streamed on Netflix and up till the year of 2021.

The dataset also consists of the following columns:

Column Name	Description
show_id	Unique ID for every movie/tv show
type	A type of identifier – whether it is \`movie\` or \`tv show\`
title	Title of the movie/tv show
director	Director of the movie/tv show (can have one director or more)
cast	Actors involved in the movie/tv show (can have one actor or more)
country	Countries which the movie/tv show was produced (can have one or more)
date_added	Date it was added on Netflix
release_year	Actual release year of movie/tv show
rating	TV rating of the movie/tv show*
duration	Total duration in minutes or number of seasons*
listed_in	Genre of the movie/tv show (can have one or more)
description	Summary of the movie/tv show

**Note:*

- For rating it refers to the TV ratings not user or viewer ratings.
  - For example, \`PG-13\`, \`TV-MA\`, \`TV-14\` and so on.
- For duration, TV shows are generally given the value as the number of seasons.
  - For example, 3 season.
  - Whereas Movies are in the total runtime in minutes (e.g. 60 min).

7. Tools Used

The main software used is Jupyter Notebook and coded in Python for the entire ETL process. PostgreSQL will be used at the very final stage of the pipeline to automatically generate the final entity-relationship diagram (ERD).

Extract

- import kaggle
  - To run API commands from Kaggle.
- import os
  - To run file check if the imported dataset from Kaggle exists on user's path.

Transform

- Import pandas as pd
  - To perform standard DataFrame operations such as: cleaning, renaming, checking for null values, changing of data types, etc.
  - Importing and exporting of .csv files.
- pd.options.mode.chained_assignment = None
  - This is to set "warnings" to not show up, as the nature of our work will cause many warnings to be flagged due to "chained assignments".
  - This syntax essentially makes the notebook looks cleaner for the reader/user.

Load

- import psycopg2
  - psycopg2 is a PostgreSQL database adapter developed for Python.
  - It allows us to create, drop and modify tables in a pseudo-SQL environment but still retaining in our original working software.
  - We use it to create the database "netflix_db".
  - We also check for any existing instance of "netflix_db", so if there is already a table it will proceed to drop it before creating a new one.
- import SQLAlchemy
  - SQLAlchemy allow us to create connections to the local PostgreSQL client via a connection URI and can also be used to create tables with proper primary and foreign key constraints set.

- It is also used to verify by reflecting all created tables to ensure that the tables within "netflix_db" are created correctly.
- It is also used to query the constraints from the information.schema table to verify that all the primary key and foreign key constraints are set correctly.
- It is also used to load the DataFrames into the created tables via "to_sql".
- Finally, as a last step of data validation, SQLAlchemy is used to query the database to ensure the number of records matches the DataFrame that was loaded into it.

Data Visualisation

- pandas
    - Data Manipulation: The `pandas` library was used extensively for data manipulation, including merging DataFrames, handling missing values, and performing group-by operations to prepare the data for visualization.
- matplotlib and seaborn
    - Data Visualization: `Matplotlib` and `Seaborn` were the primary libraries used for creating the visualizations. These libraries provided a wide range of customization options for the plots, ensuring clarity and visual appeal.
    - Customization: Titles, labels, color palettes, and figure sizes were customized to enhance the readability and interpretability of the visualizations.

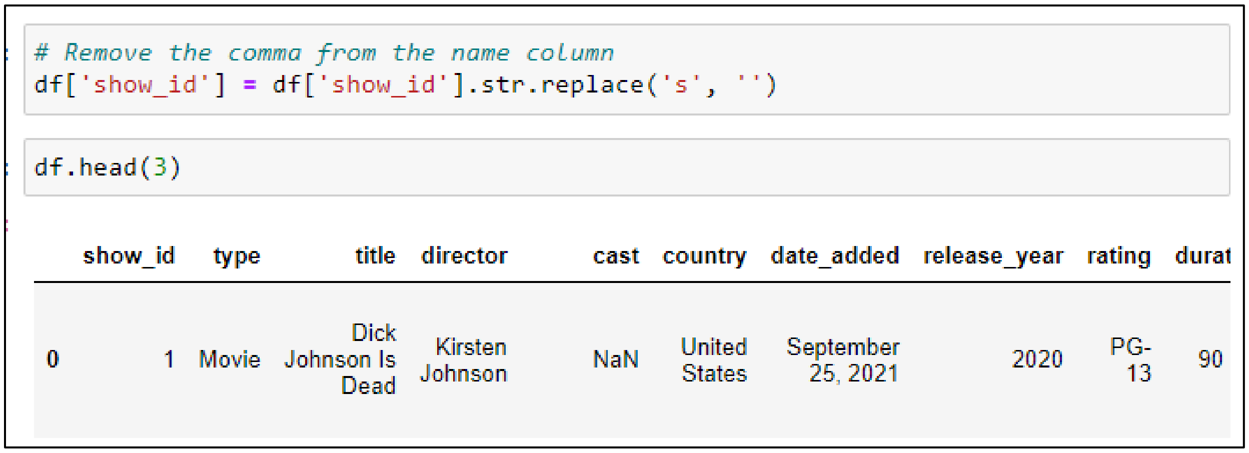
Others

- PostgreSQL (PgAdmin4)
    - To verify that tables are all correct.
    - To auto generate the entity-relationship-diagram (ERD).
- Draw.io (https://app.diagrams.net/)
    - To draft out the Data architecture model.
- Documentations
    - From various Python modules to check how each function work: psycopg2, SQLAlchemy, Kaggle API, PostgreSQL and pandas

8. Methodology

Data Cleaning

The dataset originally had the ‘show_id’ as a string, which is wrong if we plan to normalize the table. To fix this, we remove the character ‘s’ in front of the ‘show_id’ and then convert it into integer using .astype(int).

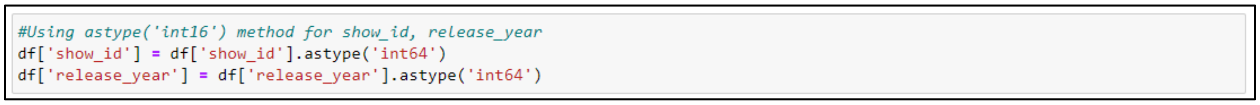


```
# Remove the comma from the name column
df['show_id'] = df['show_id'].str.replace('s', '')
```

```
df.head(3)
```

	show_id	type	title	director	cast	country	date_added	release_year	rating	durat
0	1	Movie	Dick Johnson Is Dead	Kirsten Johnson	NaN	United States	September 25, 2021	2020	PG-13	90

We then further change the format to int64 and apply the same methodology to ’release_year’:



```
#Using astype('int16') method for show_id, release_year
df['show_id'] = df['show_id'].astype('int64')
df['release_year'] = df['release_year'].astype('int64')
```

The rationale for this is because when we normalize the table there will be too many records, so having int64 essentially allows for more records instead of constraining to limited integer values.

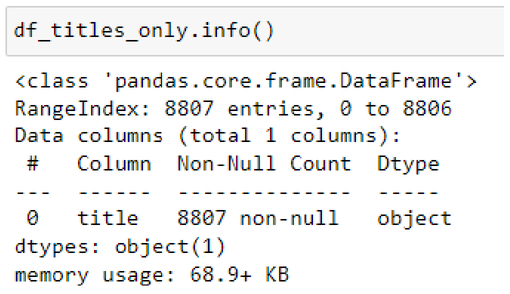
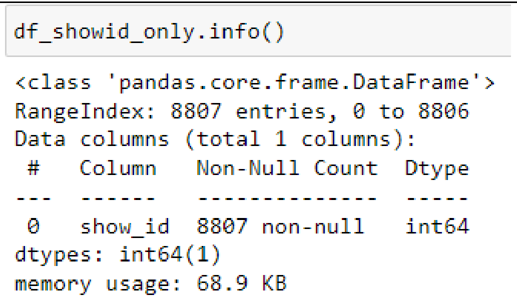
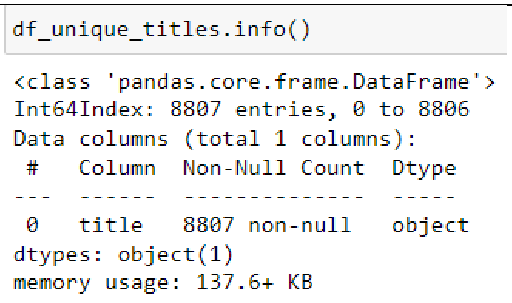
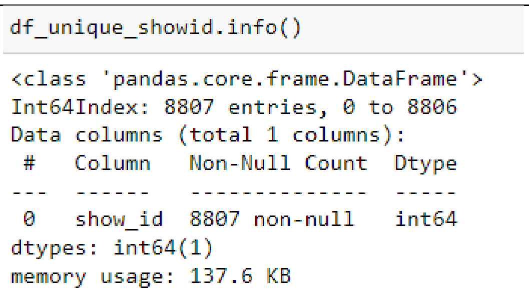
INT16	**INT32**	**<u>INT64</u>**
Takes 2 bytes in memory	Takes 4 bytes in memory	Takes 8 bytes in memory
-32,768 to 32,767	-2,147,483,648 to 2,147,483,647	-9223372036854775808 to +9223372036854775807.

Although the dataset we are dealing is small in comparison to commercial data, we are also managing a fairly small size project/environment, so having int64 provides a piece of mind for later.

We also isolated the columns 'show_id' and 'title' into a new DataFrame for each of them. To count the number of unique entries and ensure no duplicated entries.

This was done in the following steps:

1. Isolate DataFrame into titles and show_id respectively using df[['col_name']]
2. Use df.info() to count entries (8807).
3. Use df.drop_duplicates() to drop duplicated entries.
4. Run df.info() again to count entries (8807).

Before .drop_duplicates()	**Before .drop_duplicates()**
	
After .drop_duplicates()	**After .drop_duplicates()**
	

```
df_titles_only.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 8807 entries, 0 to 8806
Data columns (total 1 columns):
 #   Column  Non-Null Count  Dtype 
---  ------  --------------  ----- 
 0   title   8807 non-null   object
dtypes: object(1)
memory usage: 68.9+ KB
```

```
df_showid_only.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 8807 entries, 0 to 8806
Data columns (total 1 columns):
 #   Column   Non-Null Count  Dtype
---  ------   --------------  -----
 0   show_id  8807 non-null   int64
dtypes: int64(1)
memory usage: 68.9 KB
```

```
df_unique_titles.info()
```

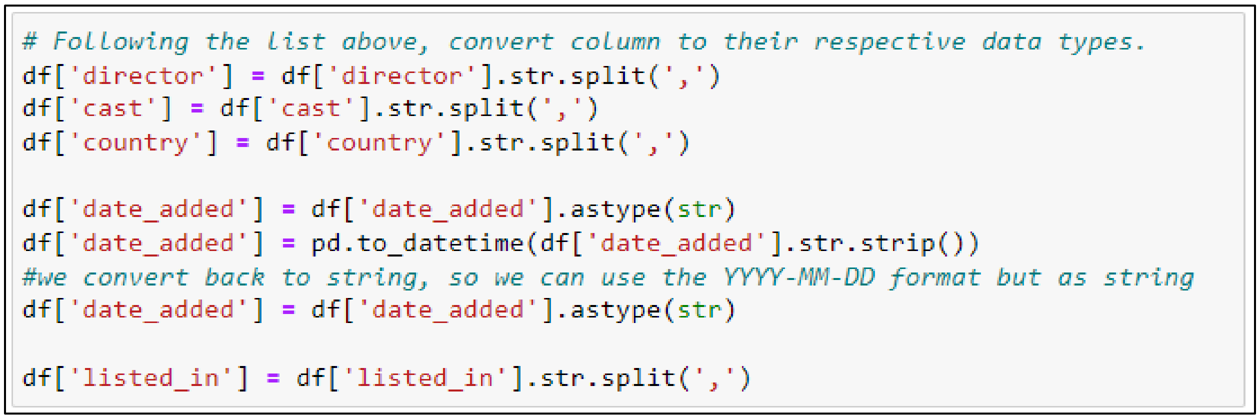
```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 8807 entries, 0 to 8806
Data columns (total 1 columns):
 #   Column  Non-Null Count  Dtype 
---  ------  --------------  ----- 
 0   title   8807 non-null   object
dtypes: object(1)
memory usage: 137.6+ KB
```

```
df_unique_showid.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 8807 entries, 0 to 8806
Data columns (total 1 columns):
 #   Column   Non-Null Count  Dtype
---  ------   --------------  -----
 0   show_id  8807 non-null   int64
dtypes: int64(1)
memory usage: 137.6 KB
```

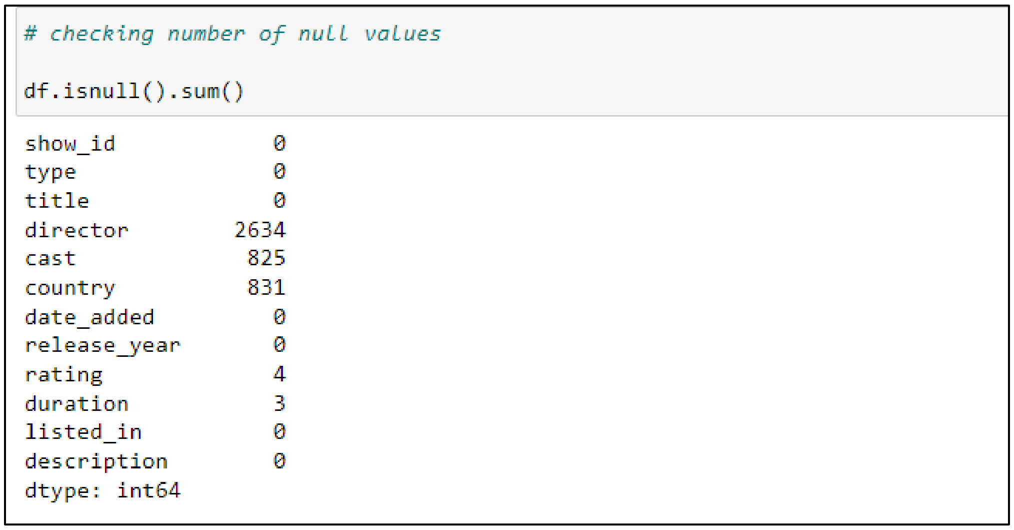
The result we have gotten for both titles and show_id indicates that they retain 8807 entries.

The data types for each other columns were also changed to string format via .str.split to easily filter out missing or NULL values and also for using the .explode() method when we atomize the columns with multiple values such as directors, casts, countries and listed_in.



```
# Following the list above, convert column to their respective data types.
df['director'] = df['director'].str.split(',')
df['cast'] = df['cast'].str.split(',')
df['country'] = df['country'].str.split(',')

df['date_added'] = df['date_added'].astype(str)
df['date_added'] = pd.to_datetime(df['date_added'].str.strip())
#we convert back to string, so we can use the YYYY-MM-DD format but as string
df['date_added'] = df['date_added'].astype(str)

df['listed_in'] = df['listed_in'].str.split(',')
```

For checking of NULL values, we first use df.isnull.sum():



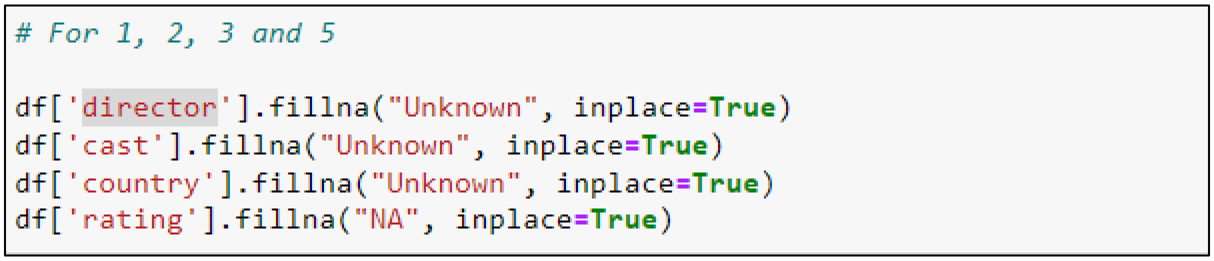
```
# checking number of null values

df.isnull().sum()
```

```
show_id            0
type               0
title              0
director        2634
cast             825
country          831
date_added         0
release_year       0
rating             4
duration           3
listed_in          0
description        0
dtype: int64
```

Based on some basic reasoning, we have decided on the following to clean up NULL values:

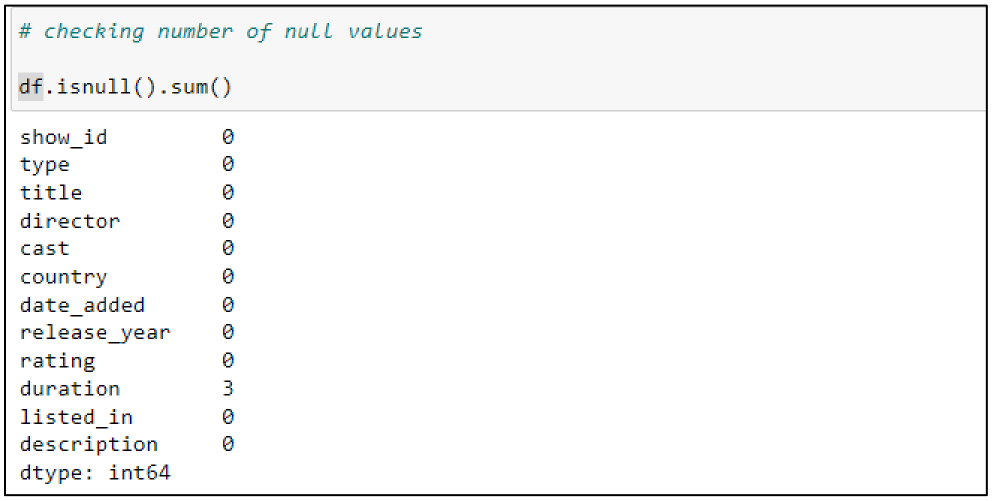
1. director – should be listed as "Unknown"
2. cast – same as director
3. country – same
4. rating – put as "NA"

The rest such as duration will be dealt with later (as there are minutes and number of seasons).



```
# For 1, 2, 3 and 5

df['director'].fillna("Unknown", inplace=True)
df['cast'].fillna("Unknown", inplace=True)
df['country'].fillna("Unknown", inplace=True)
df['rating'].fillna("NA", inplace=True)
```

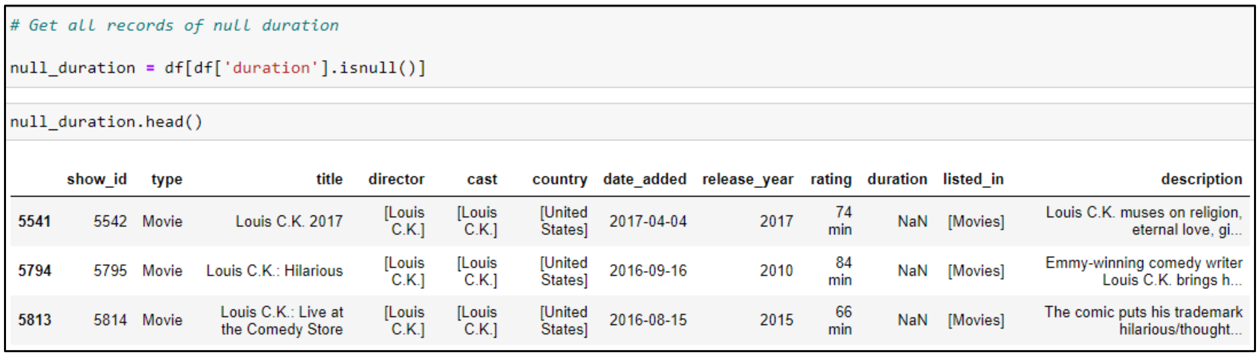
And running a second check using df.isnull.sum():



```
# checking number of null values

df.isnull().sum()
```

```
show_id         0
type            0
title           0
director        0
cast            0
country         0
date_added      0
release_year    0
rating          0
duration        3
listed_in       0
description     0
dtype: int64
```

And then for duration, we will run it separately:



```
# Get all records of null duration

null_duration = df[df['duration'].isnull()]
```

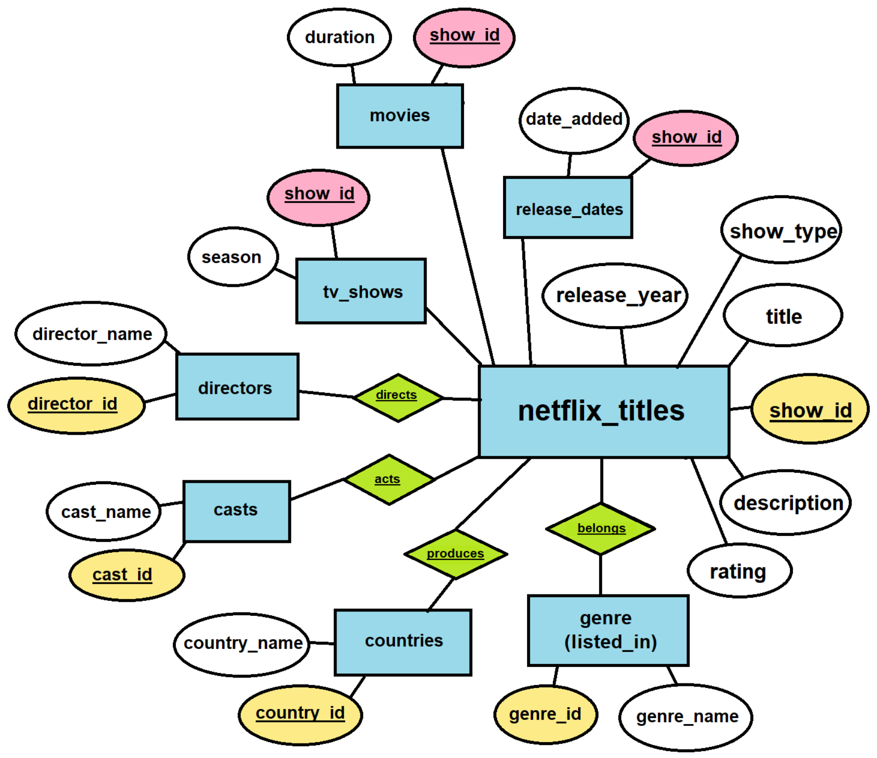
```
null_duration.head()
```

	show_id	type	title	director	cast	country	date_added	release_year	rating	duration	listed_in	description
5541	5542	Movie	Louis C.K. 2017	[Louis C.K.]	[Louis C.K.]	[United States]	2017-04-04	2017	74 min	NaN	[Movies]	Louis C.K. muses on religion, eternal love, gi...
5794	5795	Movie	Louis C.K.: Hilarious	[Louis C.K.]	[Louis C.K.]	[United States]	2016-09-16	2010	84 min	NaN	[Movies]	Emmy-winning comedy writer Louis C.K. brings h...
5813	5814	Movie	Louis C.K.: Live at the Comedy Store	[Louis C.K.]	[Louis C.K.]	[United States]	2016-08-15	2015	66 min	NaN	[Movies]	The comic puts his trademark hilarious/thought...

We then spotted that the rating were actually supposed to be duration values instead, so it was swapped over and the rating for show_id '5542', '5795' and '5814' is set to TV-MA based on Google Search results for the show_id(s) as mentioned.

Normalisation (3rd NF)

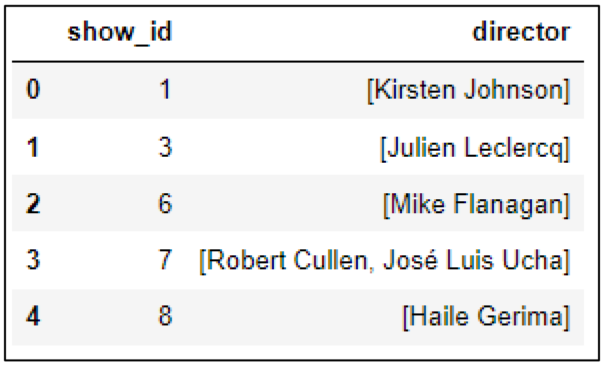
After cleaning the table and coming up with the conceptual diagram:



We will have to split the cleaned DataFrame into 12 tables as shown on the diagram, which are:

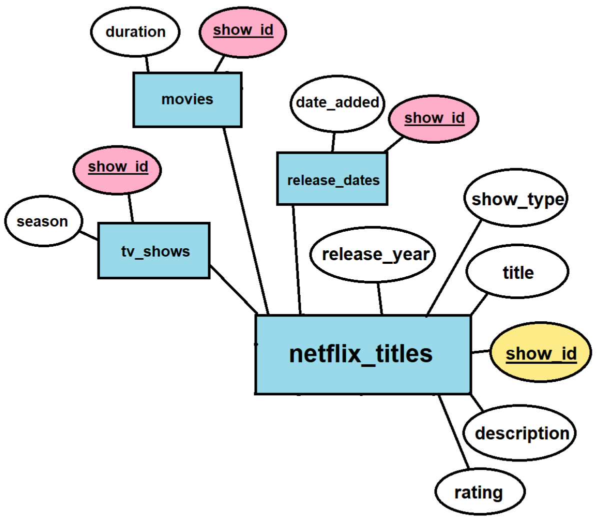
1. netflix_titles
2. movies
3. tv_shows
4. release_dates
5. directors
6. directs
7. casts
8. acts
9. countries
10. produces
11. genre
12. belongs

The reason why there are 12 tables is because we have taken into consideration 1NF and 2NF to reach 3NF.



	show_id	director
0	1	[Kirsten Johnson]
1	3	[Julien Leclercq]
2	6	[Mike Flanagan]
3	7	[Robert Cullen, José Luis Ucha]
4	8	[Haile Gerima]

Firstly, to satisfy 1NF, the records must have atomic values, so we will have to split genre, directors and casts into their respective tables, as these columns hold multiple values in a single cell.

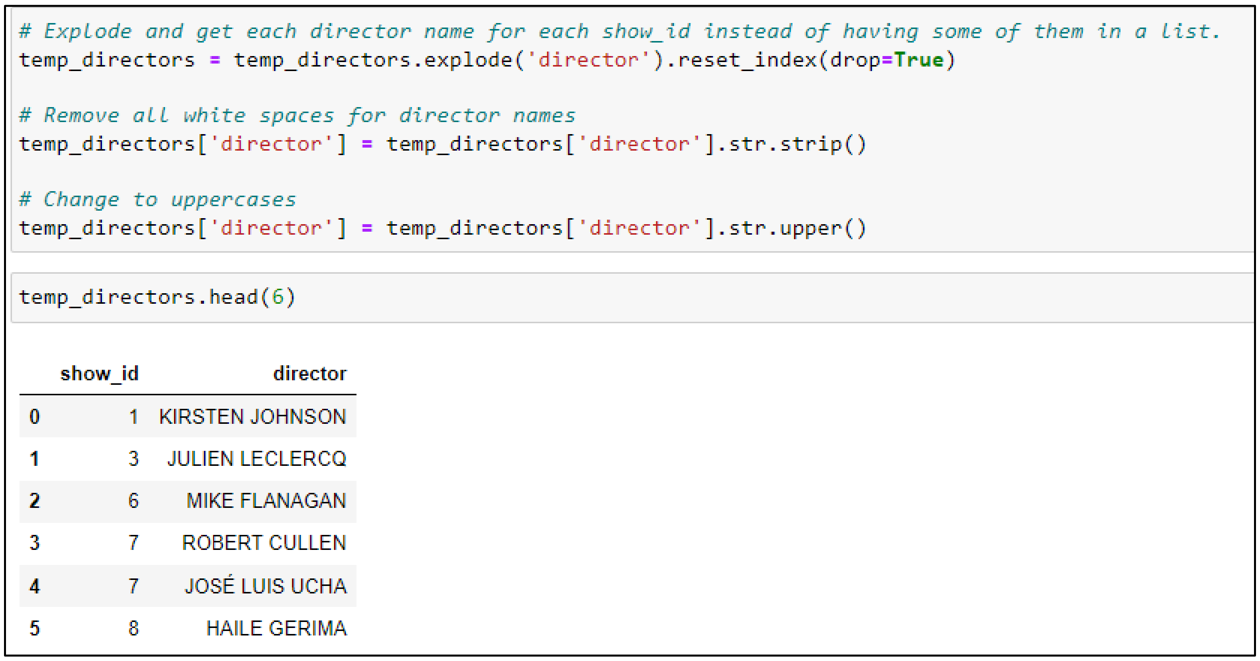


For 2NF, there must be no partial dependencies, meaning that all non-key attributes must depend on the primary key, which is why movies and tv_shows have their own tables, as the duration or number of seasons columns depends on the type of the show (whether it is movie or tv).

And finally, to reach 3NF, all the non-key columns must be directly dependent on the primary key, which explains why the netflix_titles have attributes: release_year, show_type, title, rating and description as they all depend on the primary key: show_id and nothing else.

Similarly, the 3NF concept is applied to the other DataFrames by including a junction table too.

To translate this logic to Python, we perform manual splitting of the tables one by one, such as using the method .explode() on tables with multiple values:

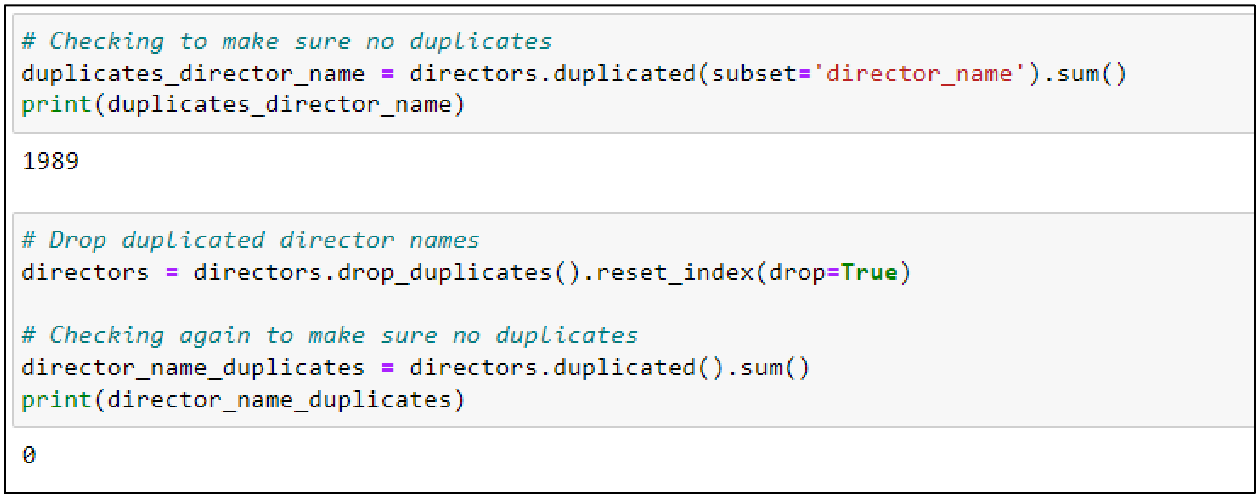


```python
# Explode and get each director name for each show_id instead of having some of them in a list.
temp_directors = temp_directors.explode('director').reset_index(drop=True)

# Remove all white spaces for director names
temp_directors['director'] = temp_directors['director'].str.strip()

# Change to uppercases
temp_directors['director'] = temp_directors['director'].str.upper()
```

```python
temp_directors.head(6)
```

	show_id	director
0	1	KIRSTEN JOHNSON
1	3	JULIEN LECLERCQ
2	6	MIKE FLANAGAN
3	7	ROBERT CULLEN
4	7	JOSÉ LUIS UCHA
5	8	HAILE GERIMA

We also made sure they are no duplicated director names, by remove white spaces or padding via .str.strip() and change all cases to upper via .str.upper() and dropping the duplicates* like so:



```python
# Checking to make sure no duplicates
duplicates_director_name = directors.duplicated(subset='director_name').sum()
print(duplicates_director_name)
```

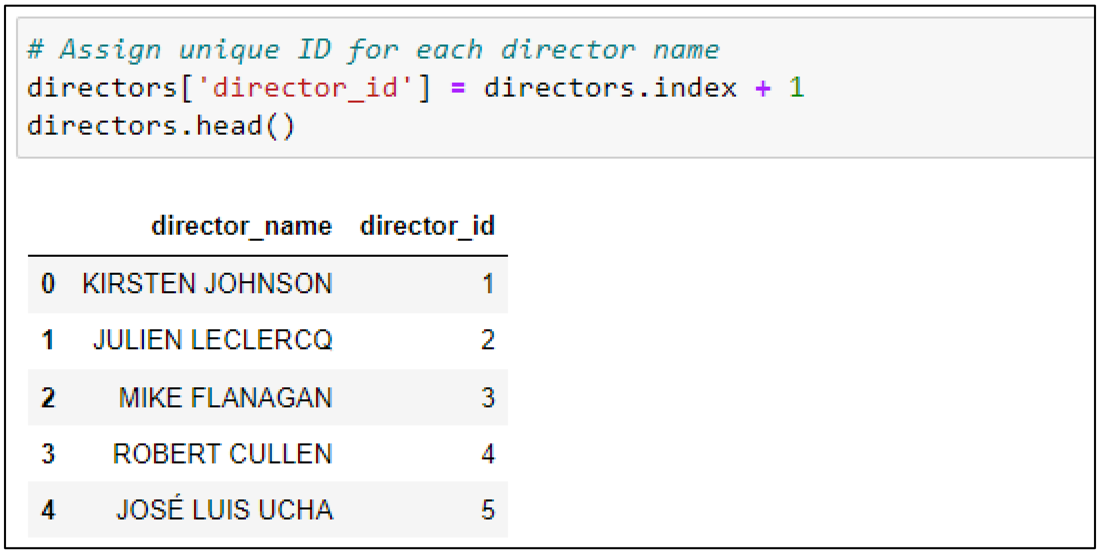
```
1989
```

```python
# Drop duplicated director names
directors = directors.drop_duplicates().reset_index(drop=True)

# Checking again to make sure no duplicates
director_name_duplicates = directors.duplicated().sum()
print(director_name_duplicates)
```

```
0
```

**Note*: The rationale for .str.upper() is because some actors will not be flagged/detected by pandas as duplicates, if the alphabet cases are different. For instance, JuJu, JuJU, JUJU, juju are all the same but not flagged as duplicates!

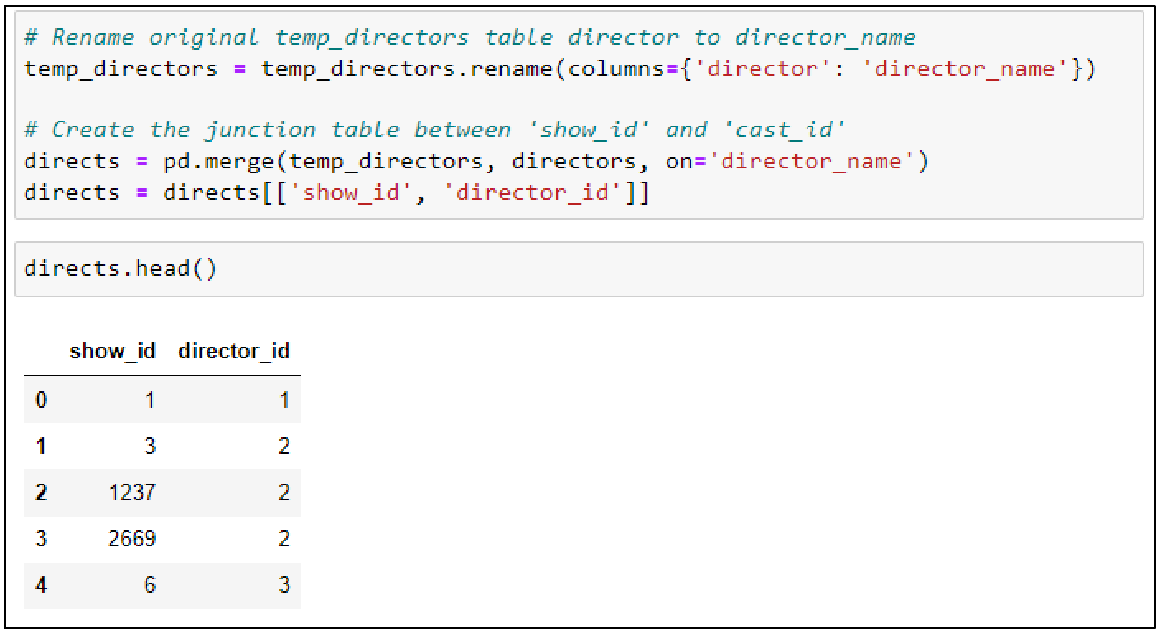
After that we assign a new unique id to each director called `director_id` using the index + 1:



```
# Assign unique ID for each director name
directors['director_id'] = directors.index + 1
directors.head()
```

	director_name	director_id
0	KIRSTEN JOHNSON	1
1	JULIEN LECLERCQ	2
2	MIKE FLANAGAN	3
3	ROBERT CULLEN	4
4	JOSÉ LUIS UCHA	5

And then, we perform a merge with the old DataFrame to get the junction table, for example in this case, while we create the directors table with ‘director_id’ and ‘director_name’. We can perform a merge with the old table to get the junction table, to get the results of only: ‘show_id’ and ‘director_id’ which are both to be used as foreign keys later.

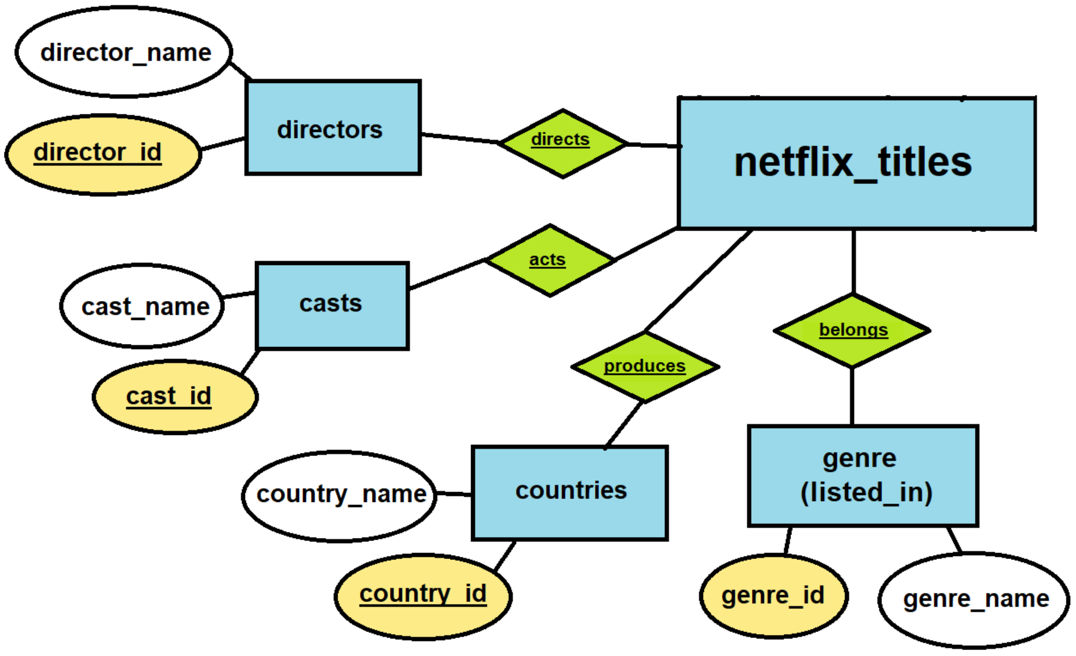


```
# Rename original temp_directors table director to director_name
temp_directors = temp_directors.rename(columns={'director': 'director_name'})

# Create the junction table between 'show_id' and 'cast_id'
directs = pd.merge(temp_directors, directors, on='director_name')
directs = directs[['show_id', 'director_id']]
```

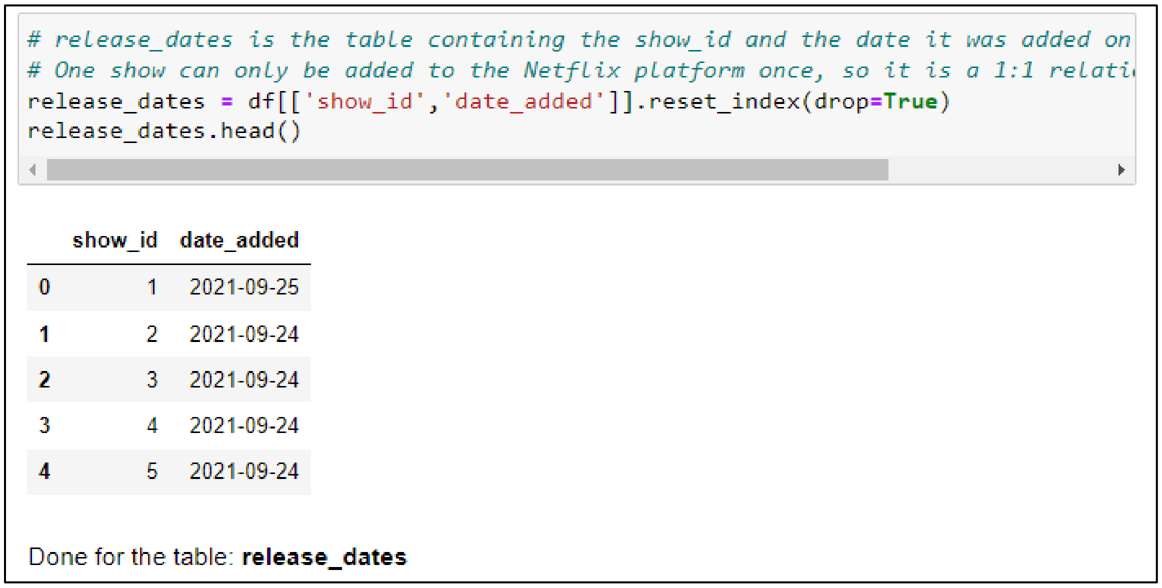
```
directs.head()
```

	show_id	director_id
0	1	1
1	3	2
2	1237	2
3	2669	2
4	6	3

This process, is repeated for the tables with N:M relationships, such as directors, casts, countries and genres:



As for the 3 other tables: movies, tv_shows and release_dates with 1:1 relationship, it can be done by splitting up the original DataFrame respectively, assigning a unique id to each and renaming the columns. For example, for the table: release_dates:

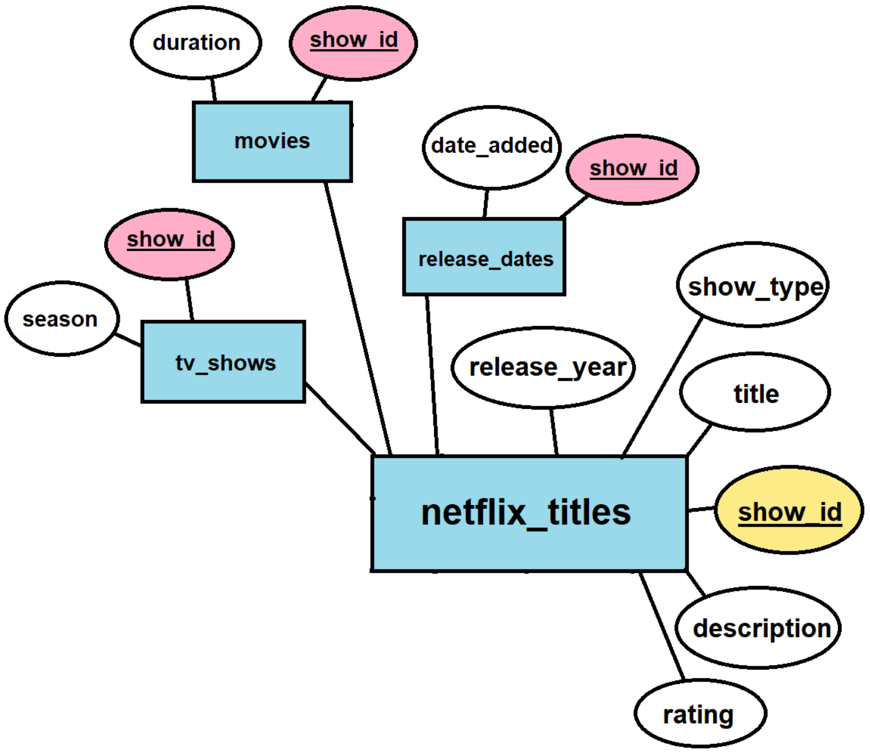


```
# release_dates is the table containing the show_id and the date it was added on
# One show can only be added to the Netflix platform once, so it is a 1:1 relati
release_dates = df[['show_id','date_added']].reset_index(drop=True)
release_dates.head()
```

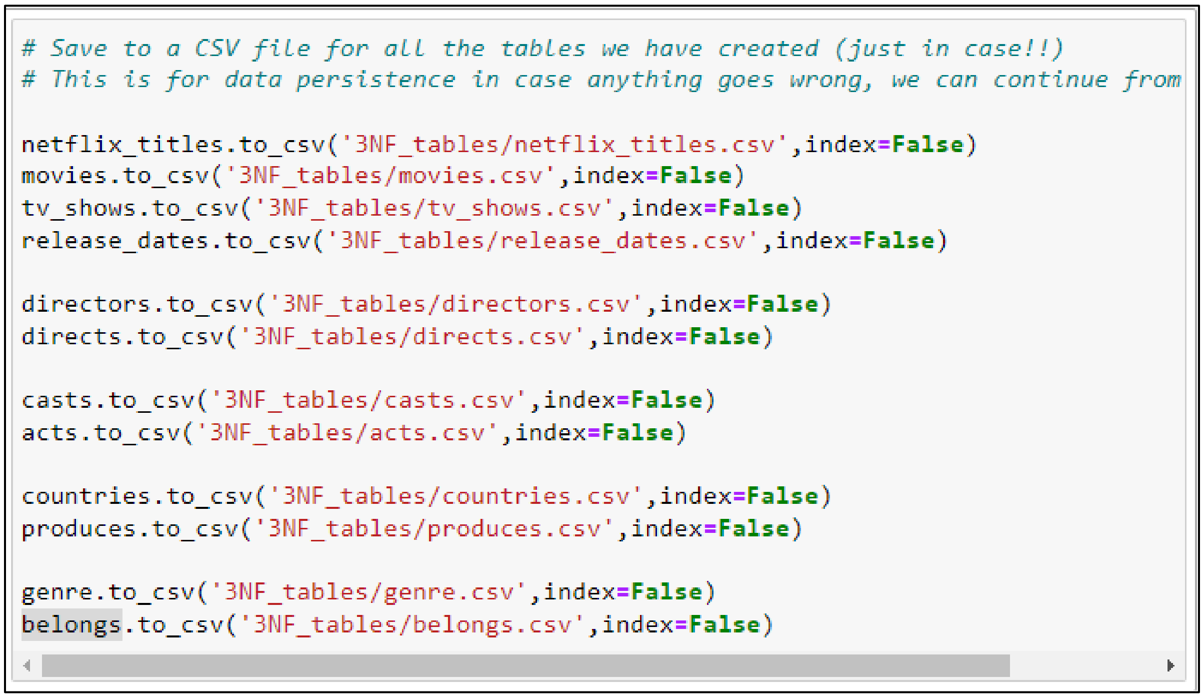
	show_id	date_added
0	1	2021-09-25
1	2	2021-09-24
2	3	2021-09-24
3	4	2021-09-24
4	5	2021-09-24

Done for the table: **release_dates**

Applying the above coding logic to the movies and tv_shows DataFrames, will fulfilled the tables with 1:1 relationship on the conceptual diagram (*see below*):

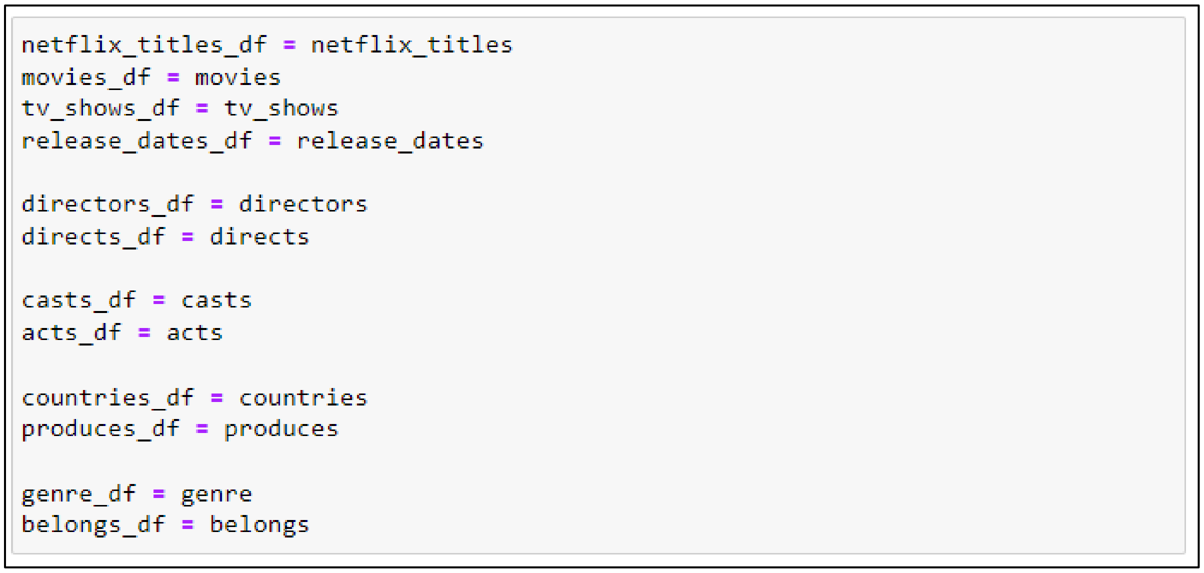


Once all 12 DataFrames has been created, we can export the DataFrames via pandas df.to_csv to ensure data persistence and as a form of backup in case anything goes wrong within the pipeline.



```
# Save to a CSV file for all the tables we have created (just in case!!)
# This is for data persistence in case anything goes wrong, we can continue from

netflix_titles.to_csv('3NF_tables/netflix_titles.csv',index=False)
movies.to_csv('3NF_tables/movies.csv',index=False)
tv_shows.to_csv('3NF_tables/tv_shows.csv',index=False)
release_dates.to_csv('3NF_tables/release_dates.csv',index=False)

directors.to_csv('3NF_tables/directors.csv',index=False)
directs.to_csv('3NF_tables/directs.csv',index=False)

casts.to_csv('3NF_tables/casts.csv',index=False)
acts.to_csv('3NF_tables/acts.csv',index=False)

countries.to_csv('3NF_tables/countries.csv',index=False)
produces.to_csv('3NF_tables/produces.csv',index=False)

genre.to_csv('3NF_tables/genre.csv',index=False)
belongs.to_csv('3NF_tables/belongs.csv',index=False)
```

And finally, renaming all 12 DataFrames with a suffix _df:



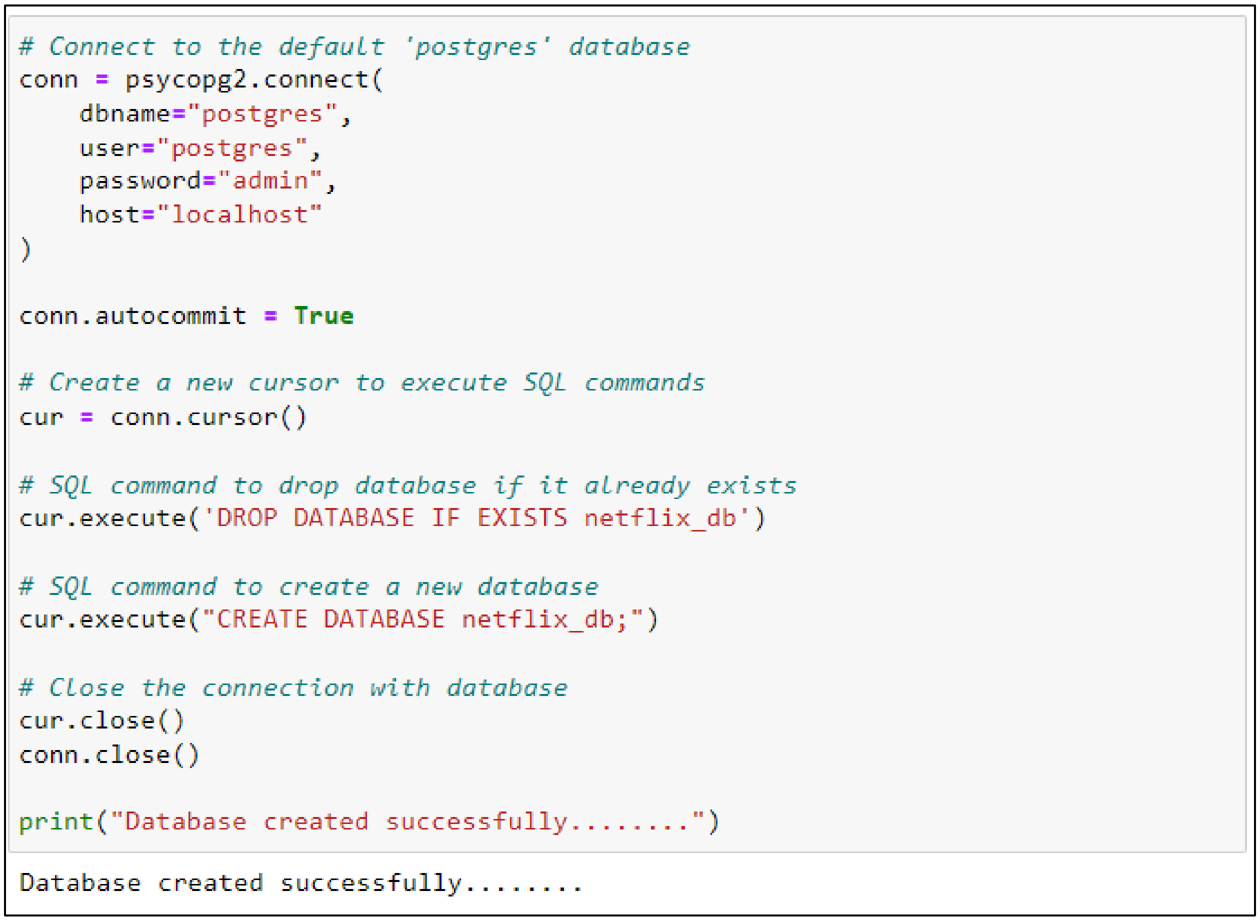
```
netflix_titles_df = netflix_titles
movies_df = movies
tv_shows_df = tv_shows
release_dates_df = release_dates

directors_df = directors
directs_df = directs

casts_df = casts
acts_df = acts

countries_df = countries
produces_df = produces

genre_df = genre
belongs_df = belongs
```

The reason why is because the same names will be used later for creating the tables for PostgreSQL, this is to prevent confusion when running the pipeline and maintaining code readability for any user.

Additionally, if the DataFrame and table names were the same when trying to load using SQLAlchemy, it will throw an error as it cannot be the same name, so adding the suffix at this point also make sense.

Loading to PostgreSQL Database

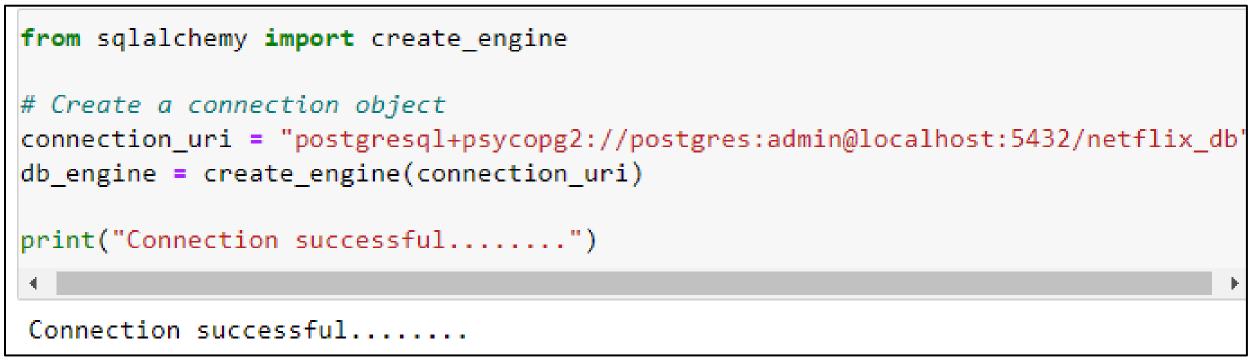
To load our 12 DataFrames into the PostgreSQL database, we can use a combination of psycopg2 and SQLAlchemy. First, a connection object can be created by creating a connection URI and engine via SQLAlchemy, which we can use to create cursors to create our database in PostgreSQL called ‘netflix_db’.



```
# Connect to the default 'postgres' database
conn = psycopg2.connect(
    dbname="postgres",
    user="postgres",
    password="admin",
    host="localhost"
)

conn.autocommit = True

# Create a new cursor to execute SQL commands
cur = conn.cursor()

# SQL command to drop database if it already exists
cur.execute('DROP DATABASE IF EXISTS netflix_db')

# SQL command to create a new database
cur.execute("CREATE DATABASE netflix_db;")

# Close the connection with database
cur.close()
conn.close()

print("Database created successfully........")
```

```
Database created successfully........
```

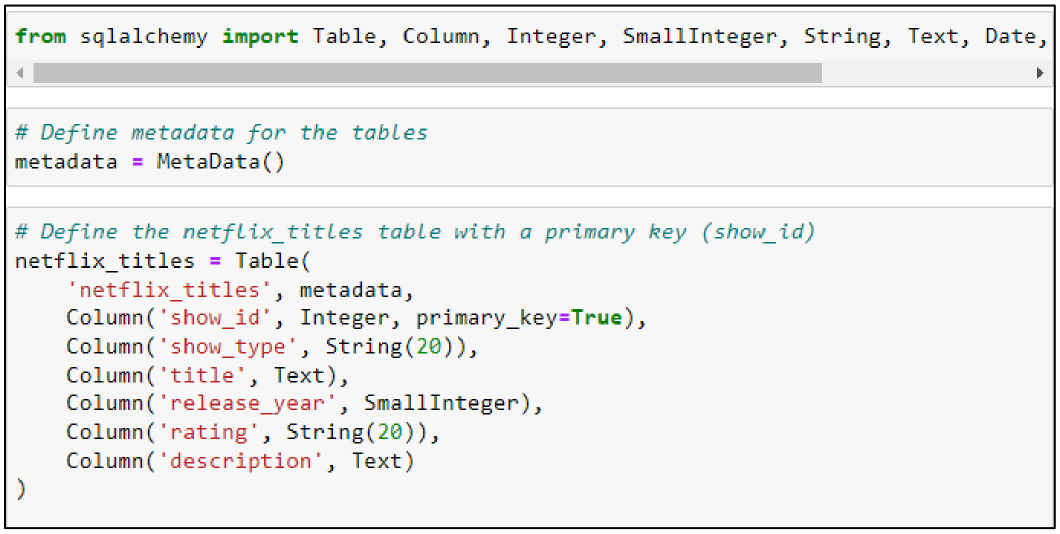
After which, we can amend the connection URI to connect to our ’netflix_db’ and creating an engine with the help of SQLAlchemy again:



```
from sqlalchemy import create_engine

# Create a connection object
connection_uri = "postgresql+psycopg2://postgres:admin@localhost:5432/netflix_db"
db_engine = create_engine(connection_uri)

print("Connection successful........")
```

```
Connection successful........
```

This allows us to create our tables which can be loaded on to 'netflix_db' later. To create the tables, first, we define the metadata which will store all the table objects and the schema we defined for the tables.

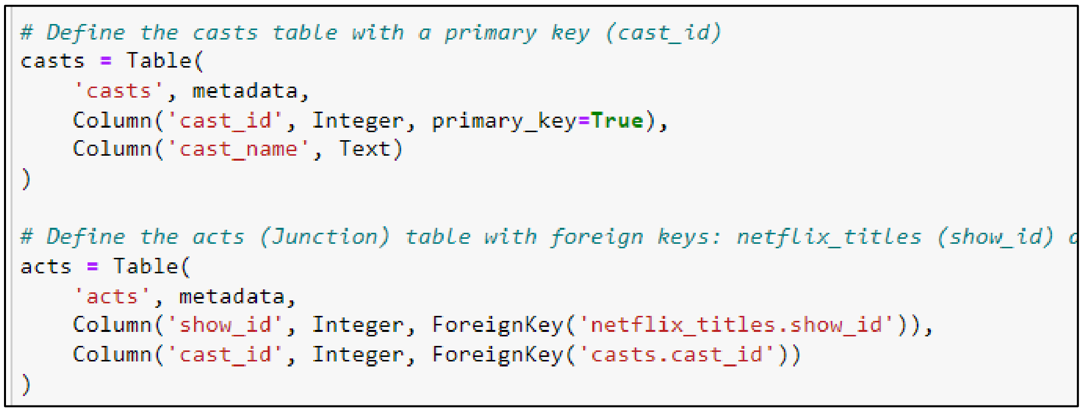


```python
from sqlalchemy import Table, Column, Integer, SmallInteger, String, Text, Date,
```

```python
# Define metadata for the tables
metadata = MetaData()
```

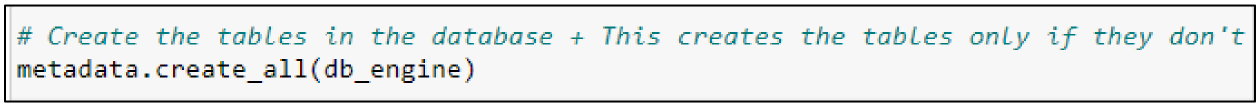
```python
# Define the netflix_titles table with a primary key (show_id)
netflix_titles = Table(
    'netflix_titles', metadata,
    Column('show_id', Integer, primary_key=True),
    Column('show_type', String(20)),
    Column('title', Text),
    Column('release_year', SmallInteger),
    Column('rating', String(20)),
    Column('description', Text)
)
```

For many-to-many relationship tables and their respective junction tables, this is how we establish the referential integrity and the primary key and foreign key constraints:



```python
# Define the casts table with a primary key (cast_id)
casts = Table(
    'casts', metadata,
    Column('cast_id', Integer, primary_key=True),
    Column('cast_name', Text)
)

# Define the acts (Junction) table with foreign keys: netflix_titles (show_id) a
acts = Table(
    'acts', metadata,
    Column('show_id', Integer, ForeignKey('netflix_titles.show_id')),
    Column('cast_id', Integer, ForeignKey('casts.cast_id'))
)
```

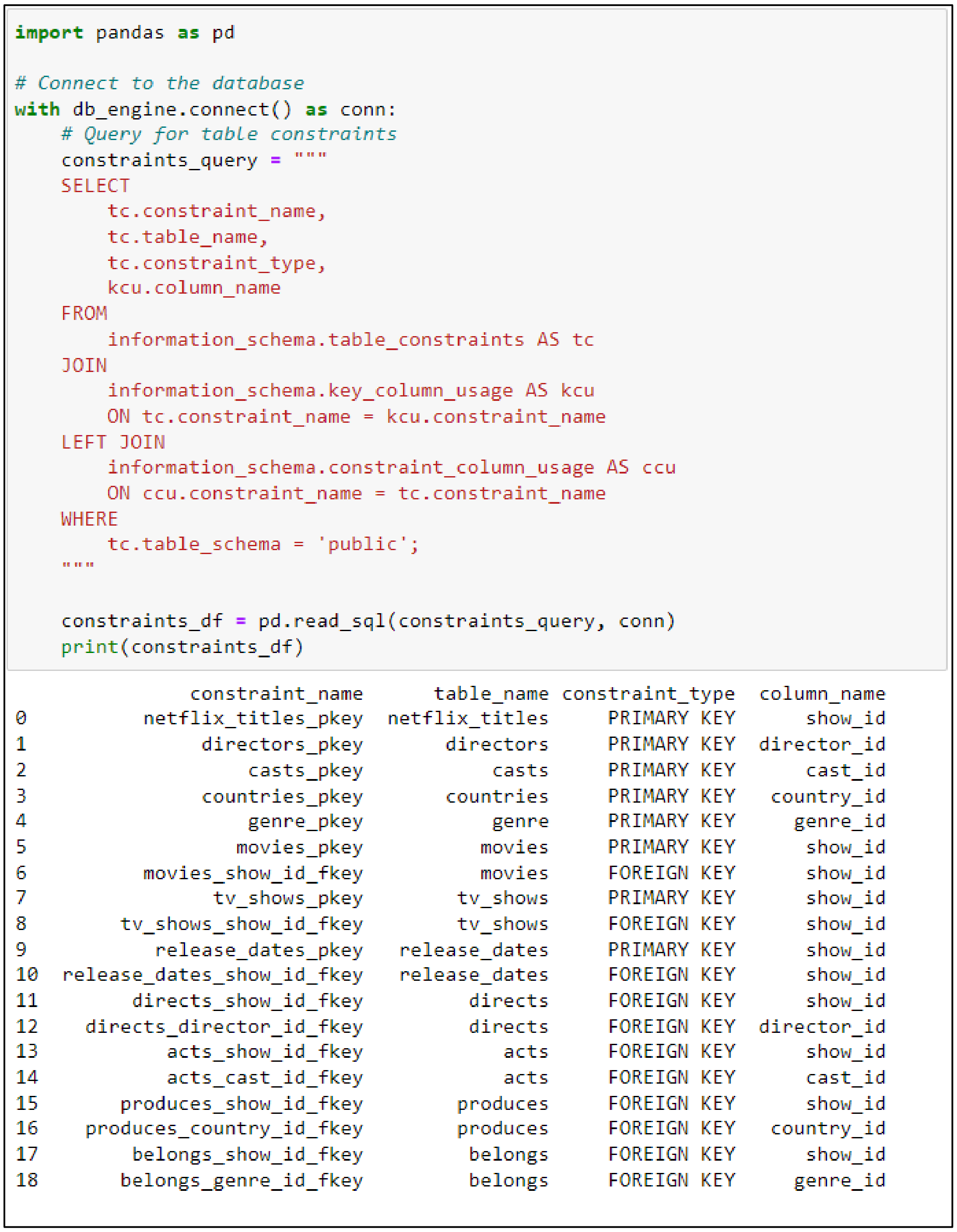
And then, by calling the metadata object and its associated function create_all, PostgreSQL will create the tables within the 'netflix_db' if it has not been created:



```python
# Create the tables in the database + This creates the tables only if they don't
metadata.create_all(db_engine)
```

For more information about the syntax, refer to SQLAlchemy 2.0 Documentation for constraints and indexes, see references [3].

Before proceeding to load the newly created tables with our normalized DataFrames, we also verified the constraints of our tables within the 'netflix_db' database:

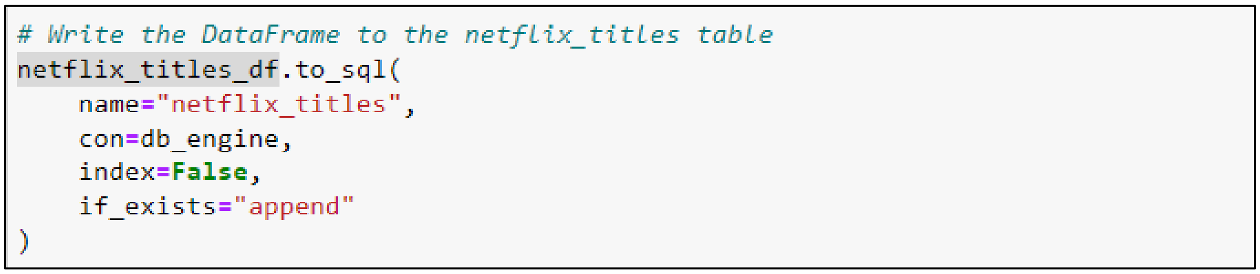


```
import pandas as pd

# Connect to the database
with db_engine.connect() as conn:
    # Query for table constraints
    constraints_query = """
    SELECT
        tc.constraint_name,
        tc.table_name,
        tc.constraint_type,
        kcu.column_name
    FROM
        information_schema.table_constraints AS tc
    JOIN
        information_schema.key_column_usage AS kcu
        ON tc.constraint_name = kcu.constraint_name
    LEFT JOIN
        information_schema.constraint_column_usage AS ccu
        ON ccu.constraint_name = tc.constraint_name
    WHERE
        tc.table_schema = 'public';
    """

    constraints_df = pd.read_sql(constraints_query, conn)
    print(constraints_df)
```

```
               constraint_name      table_name constraint_type  column_name
0            netflix_titles_pkey  netflix_titles     PRIMARY KEY      show_id
1                 directors_pkey       directors     PRIMARY KEY  director_id
2                     casts_pkey           casts     PRIMARY KEY      cast_id
3                 countries_pkey       countries     PRIMARY KEY   country_id
4                     genre_pkey           genre     PRIMARY KEY     genre_id
5                    movies_pkey          movies     PRIMARY KEY      show_id
6            movies_show_id_fkey          movies     FOREIGN KEY      show_id
7                  tv_shows_pkey        tv_shows     PRIMARY KEY      show_id
8          tv_shows_show_id_fkey        tv_shows     FOREIGN KEY      show_id
9             release_dates_pkey   release_dates     PRIMARY KEY      show_id
10    release_dates_show_id_fkey   release_dates     FOREIGN KEY      show_id
11          directs_show_id_fkey         directs     FOREIGN KEY      show_id
12      directs_director_id_fkey         directs     FOREIGN KEY  director_id
13             acts_show_id_fkey            acts     FOREIGN KEY      show_id
14             acts_cast_id_fkey            acts     FOREIGN KEY      cast_id
15         produces_show_id_fkey        produces     FOREIGN KEY      show_id
16      produces_country_id_fkey        produces     FOREIGN KEY   country_id
17          belongs_show_id_fkey         belongs     FOREIGN KEY      show_id
18         belongs_genre_id_fkey         belongs     FOREIGN KEY     genre_id
```

Syntax is taken from PostgreSQL documentation [4], refer to references for more information.

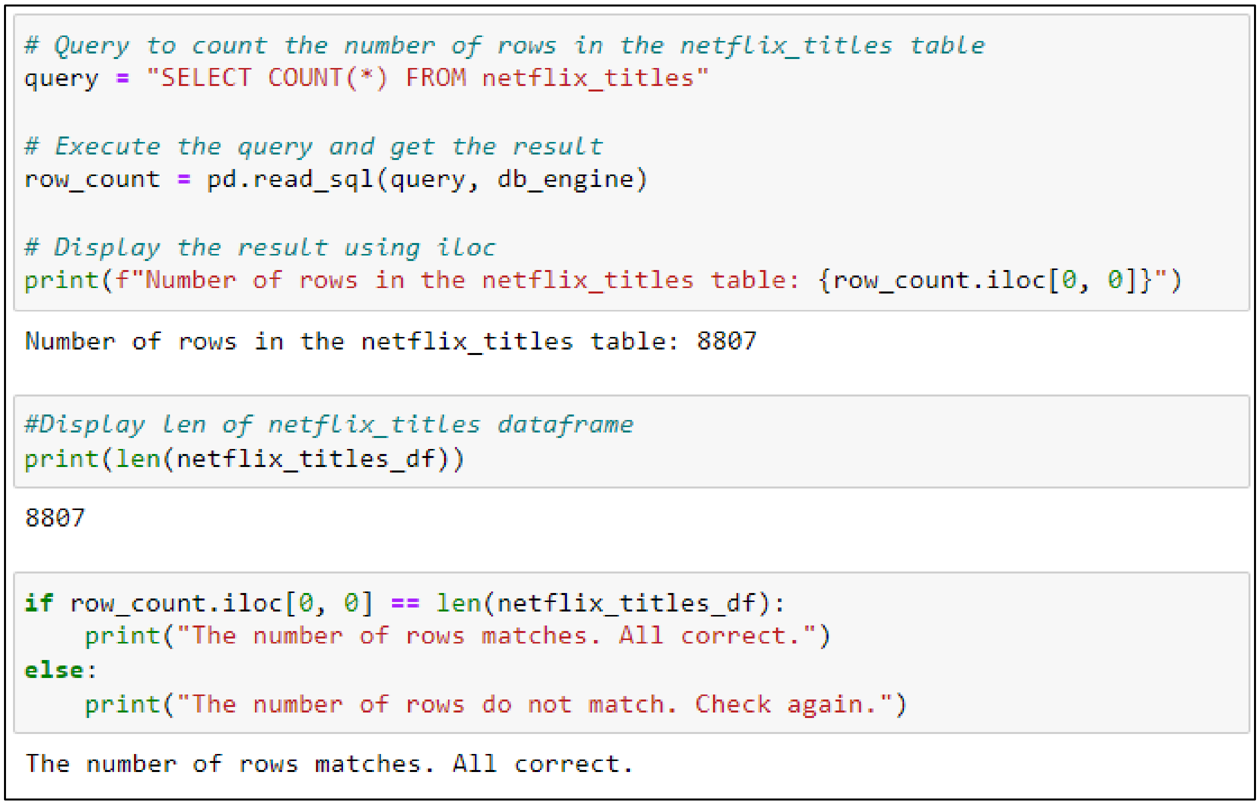
Finally, we can use SQLAlchemy's df.to_sql function to load the 12 normalised DataFrames into the tables created within the 'netflix_db' database.



```python
# Write the DataFrame to the netflix_titles table
netflix_titles_df.to_sql(
    name="netflix_titles",
    con=db_engine,
    index=False,
    if_exists="append"
)
```

After loading, we perform data validation by ensuring that the record in the table matches the rows in their respective DataFrame by:

1. Querying all records from the table using SELECT count (*)
2. Length of DataFrame using: len(df_name)



```python
# Query to count the number of rows in the netflix_titles table
query = "SELECT COUNT(*) FROM netflix_titles"

# Execute the query and get the result
row_count = pd.read_sql(query, db_engine)

# Display the result using iloc
print(f"Number of rows in the netflix_titles table: {row_count.iloc[0, 0]}")
```

```
Number of rows in the netflix_titles table: 8807
```

```python
#Display len of netflix_titles dataframe
print(len(netflix_titles_df))
```

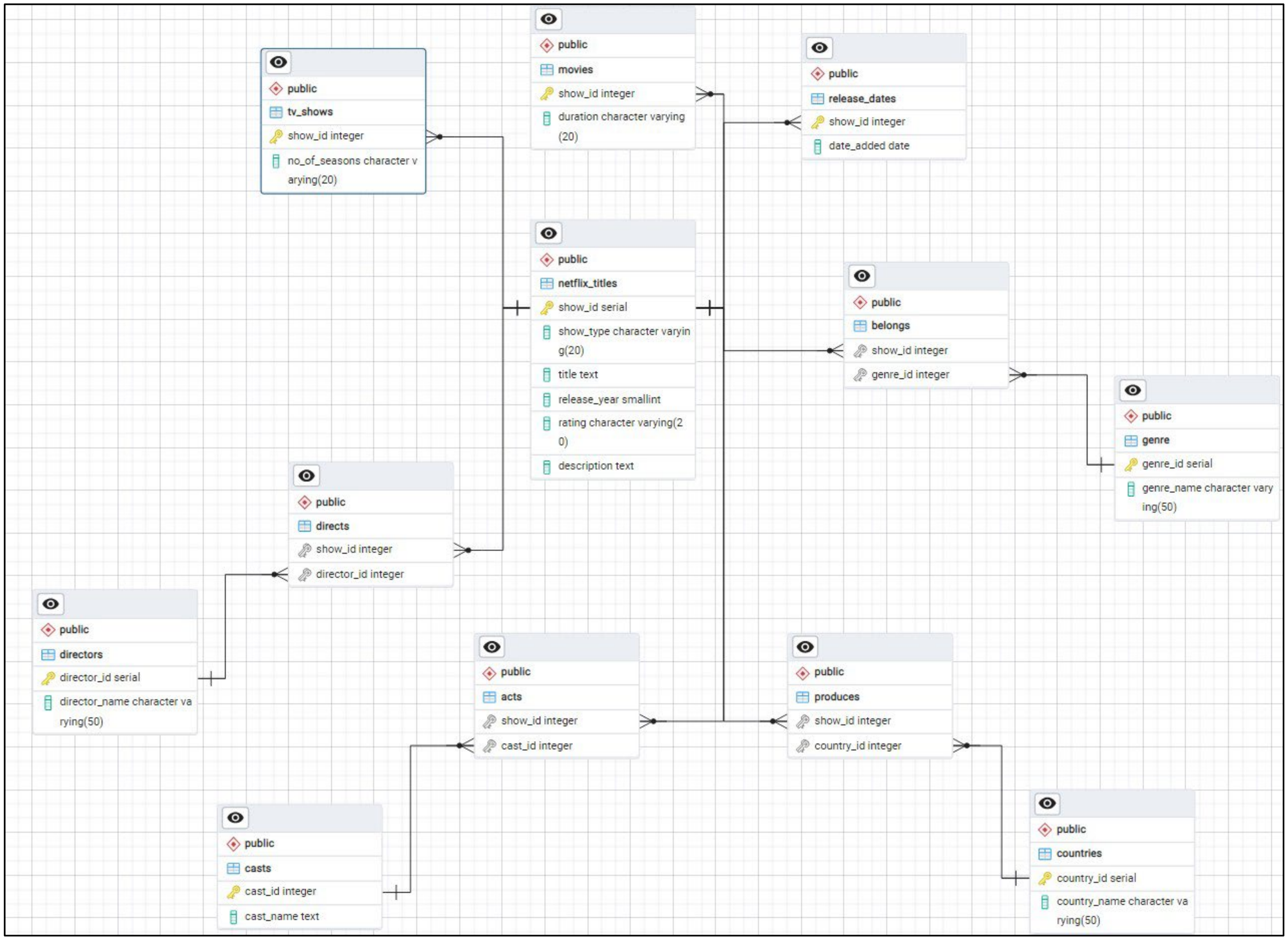
```
8807
```

```python
if row_count.iloc[0, 0] == len(netflix_titles_df):
    print("The number of rows matches. All correct.")
else:
    print("The number of rows do not match. Check again.")
```

```
The number of rows matches. All correct.
```

This step will be repeated for the remaining 11 DataFrames and tables within the `netflix_db` until all has been successfully loaded.

As a final form of check, PostgreSQL will autogenerate the entity-relationship-diagram (ERD) as shown:



Data Visualization

Merging DataFrames: DataFrames containing information on titles, genres, and countries were merged to create a comprehensive dataset (`netflix_with_genres_and_countries`) that could be used for detailed analysis.

The following visualizations were created to explore and present the findings:

1. **Bar Charts**:

- Top 10 Most Common Genres: A bar chart was used to display the most frequently occurring genres in Netflix's catalog. This visualization highlighted the genres with the highest number of titles.

-Top Countries Producing Documentaries: Another bar chart was created to show the top countries that produce the most documentaries, providing insight into regional strengths in documentary filmmaking.

2. **Clustered Bar Charts**:

- Top Genres by Country (Top 10 Countries): Clustered bar charts were used to compare the popularity of different genres across the top 10 content-producing countries. This allowed for a clear comparison of genre preferences across regions.

3. **Line Plots**:

- Trends in Genre Popularity Over the Last Few Years: Line plots were utilized to track the changes in genre popularity over time. This visualization helped identify emerging trends and declining genres, offering insights into shifts in audience preferences.

9. Analysis

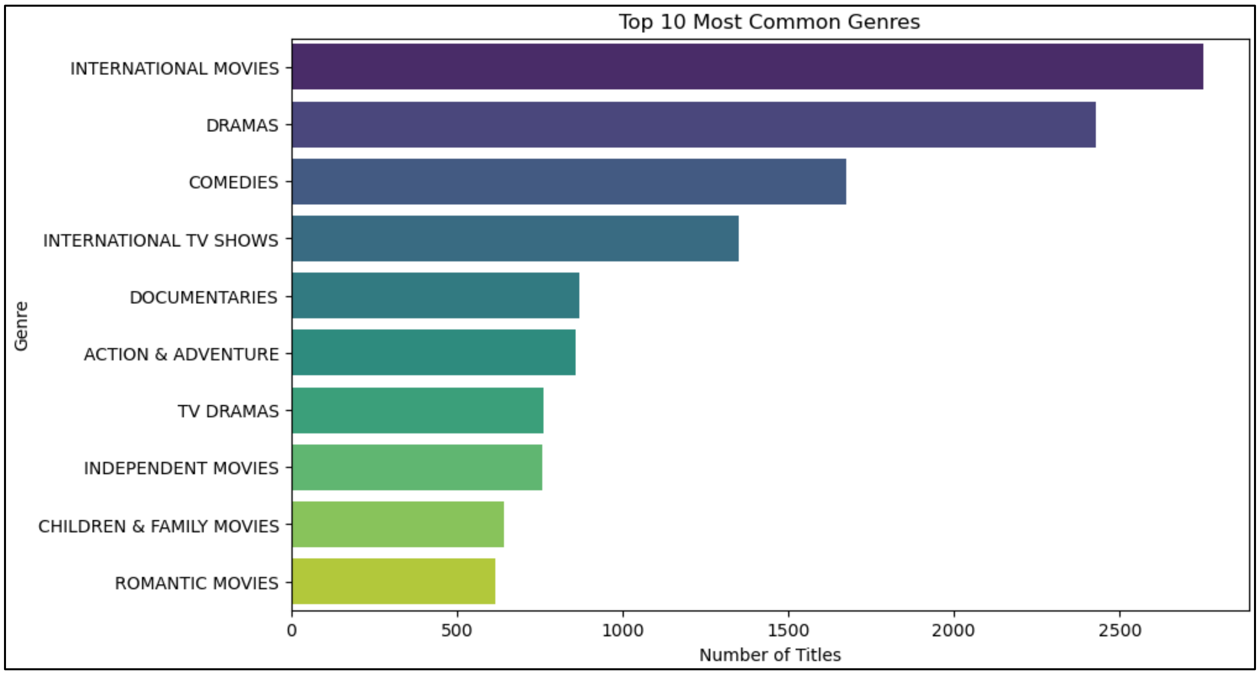
The key objectives of this analysis are:

- To identify the most common genres across the entire Netflix catalogue.
- To analyse genre popularity trends over the last few years.
- To explore the regional differences in genre preferences and production.
- To understand which countries, contribute the most diverse set of genres.

The data visualisation focuses on the following:

- Top 10 Most Common Genres
- Top Genres by Country (Top 10 Countries)
- Top Countries Producing Documentaries
- Trends in Genre Popularity Over The Last Few Years
- Top 10 Countries Watching International Movies
- Top 10 Countries By Total Content
- Top 10 Countries With The Most Diverse Set of Genres

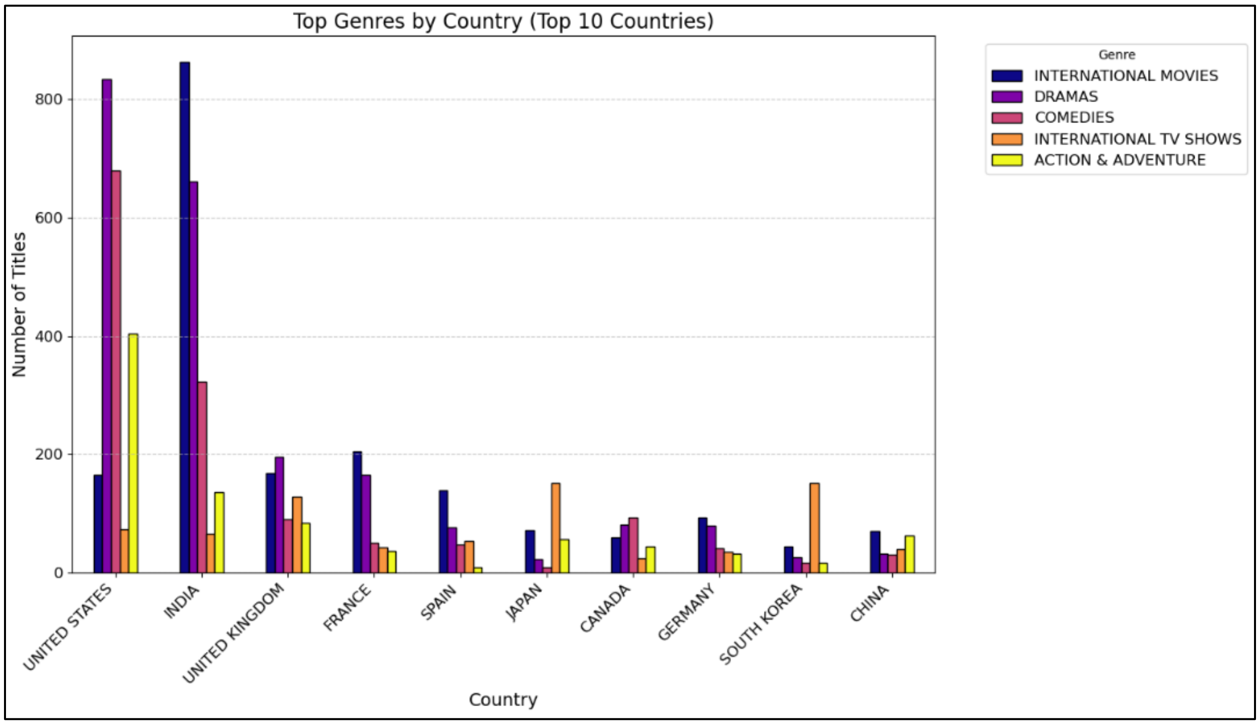
Top 10 Most Common Genres



This analysis identifies the most frequently occurring genres in Netflix's catalogue. The bar chart highlights the top 10 genres that dominate Netflix's offerings.

- **Dominance of Certain Genres**: Genres like "Drama," "Comedy," and "Documentaries" are among the most common, reflecting broad audience appeal and Netflix's strategic focus on these genres.
- **Strategic Implications**: Netflix can leverage the popularity of these genres to continue investing in similar content, while also exploring opportunities to diversify offerings in less common genres.

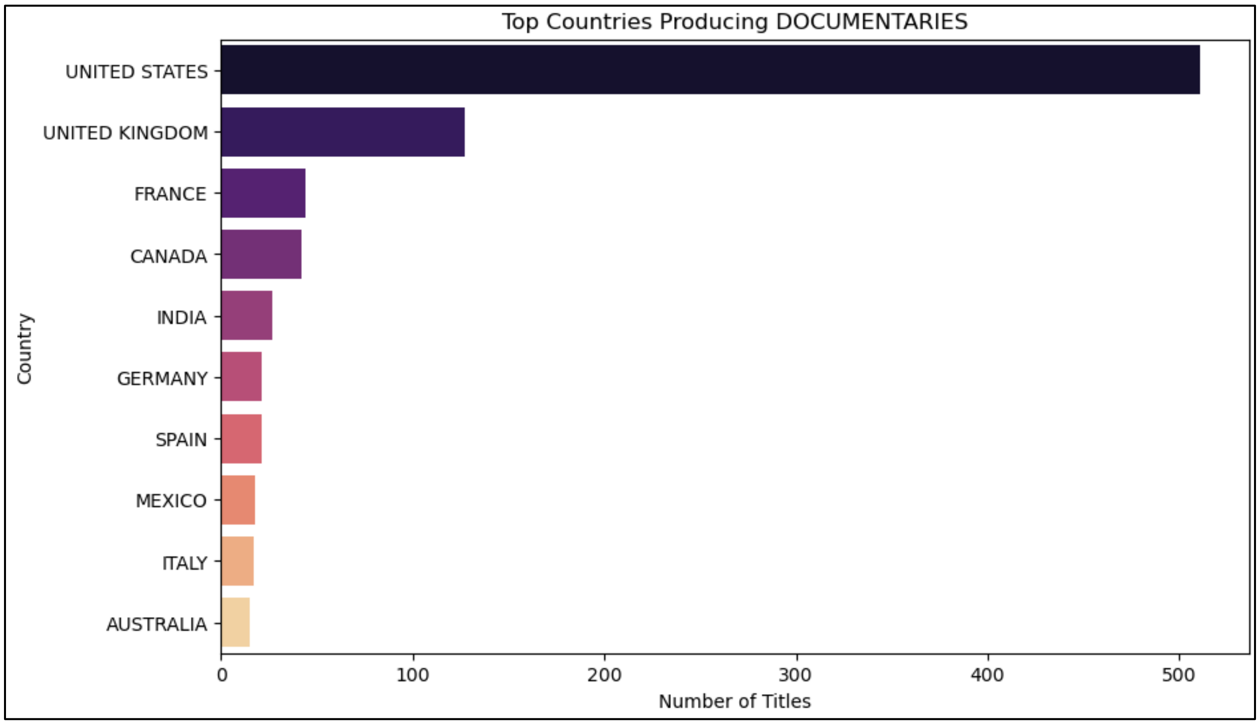
Top Genres by Country (Top 10 Countries)



This analysis examines the top genres across the top 10 countries that contribute the most content to Netflix's catalogue.

- **Regional Preferences**: The analysis reveals that certain genre, such as "Drama" and "Comedy," are universally popular, while others, like "International Movies," show stronger regional appeal.
- **Strategic Implications**: Netflix should consider regional preferences when curating content libraries and developing original content tailored to specific markets.

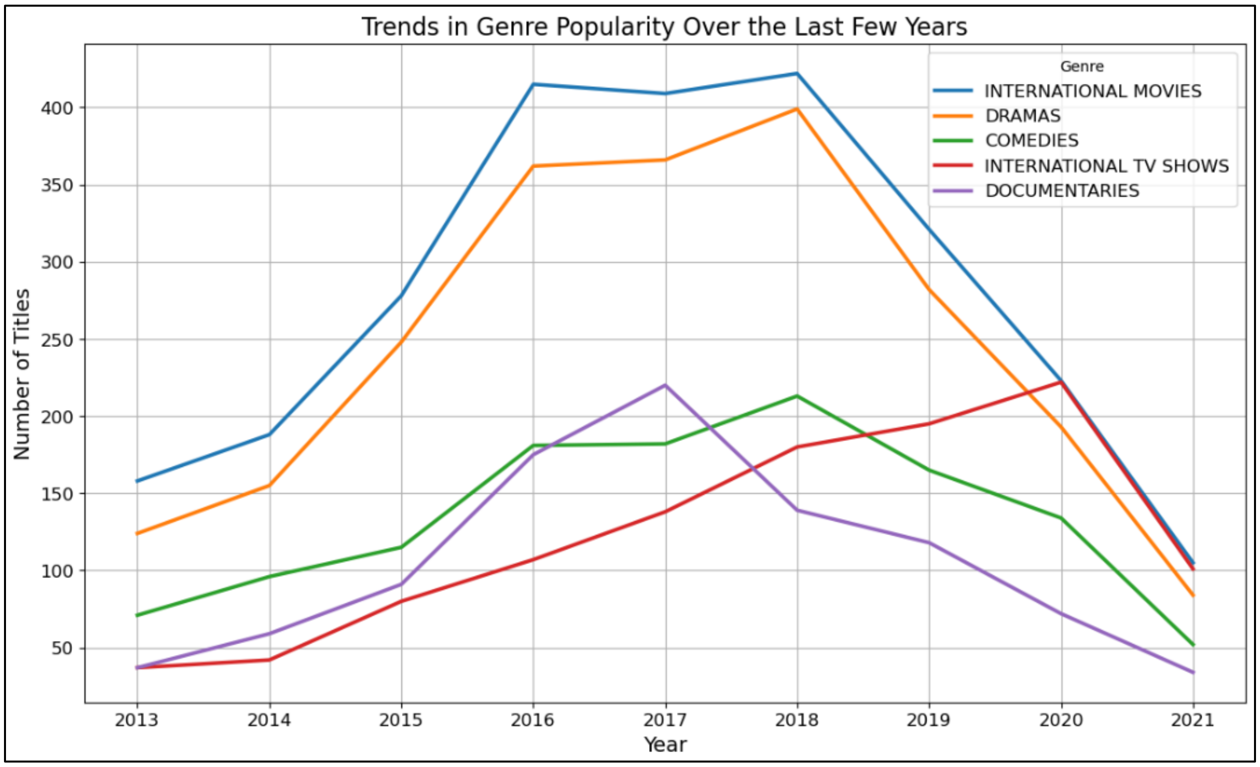
Top Countries Producing Documentaries



This section highlights the top countries that are leading in documentary production.

- **Documentary Production Leaders**: Countries like the United States and the United Kingdom are leading in producing documentaries, likely due to their established film industries and global reach.
- **Strategic Implications**: Netflix can focus on acquiring high-quality documentaries from these countries to enhance its library, particularly as documentaries continue to gain popularity among audiences.

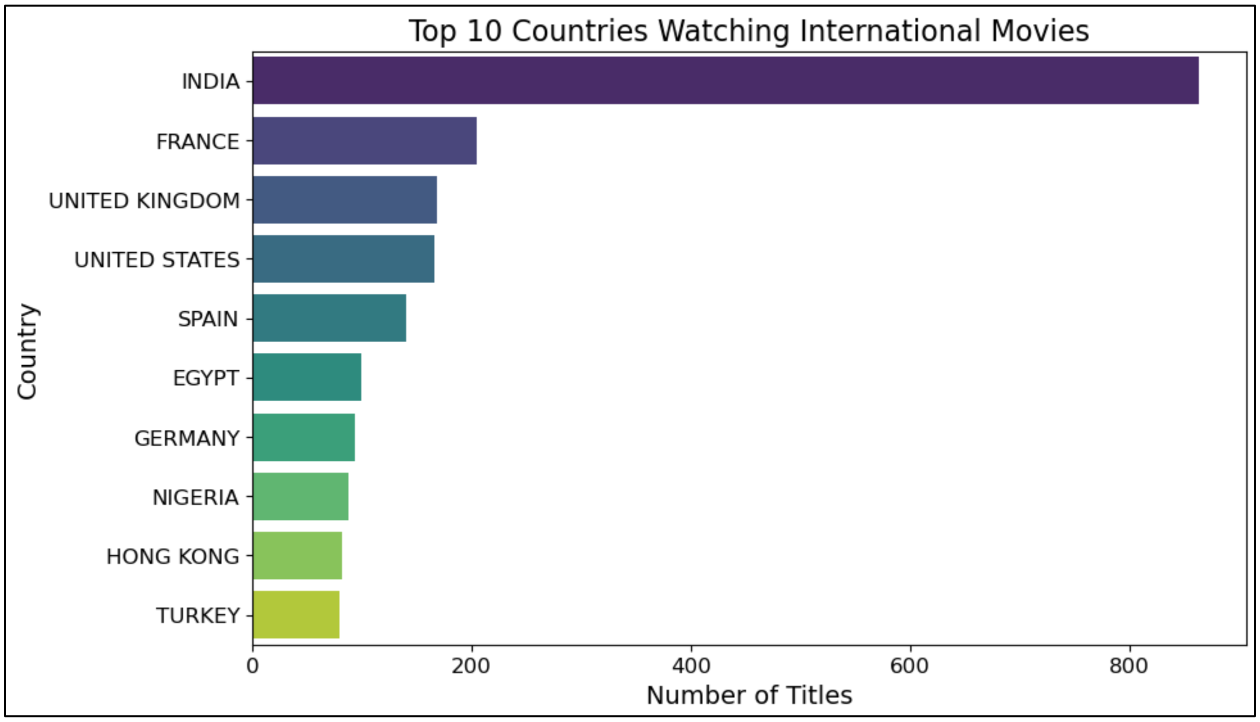
Trends in Genre Popularity Over The Last Few Years



This analysis tracks the popularity of various genres over the last few years, identifying which genres have grown or declined.

- **Emerging Genres**: The analysis shows a rise in the popularity of genres like "Documentaries" and "International Movies," indicating a growing and shifting audience interest in diverse and factual content.
- **Strategic Implications**: Netflix should consider increasing investment in emerging genres while re-evaluating the demand for declining genres. Strategic adjustments will help Netflix to better align with changing viewers' interests and maintain its competitiveness in the market.

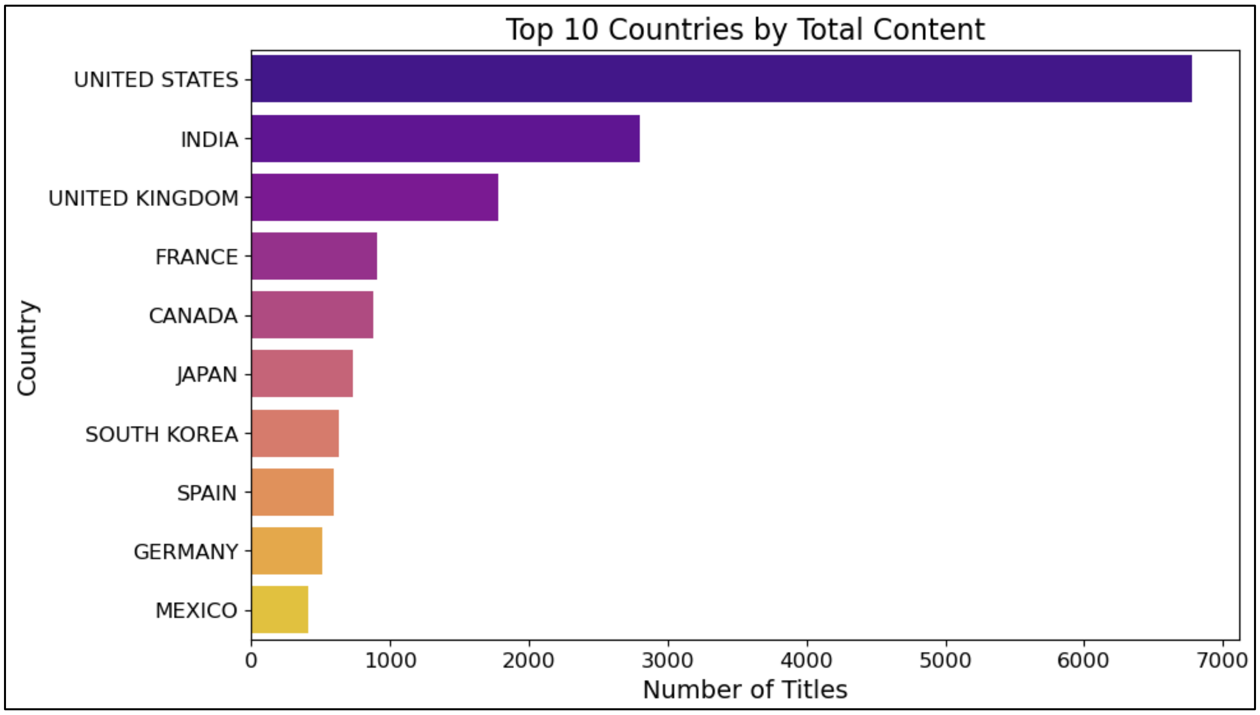
Top 10 Countries Watching International Movies



This analysis focuses on the top 10 countries with the highest engagement in watching international movies.

- **Global Reach**: Countries like India and France are among the top consumers of international movies, reflecting a growing interest in diverse global content.
- **Strategic Implications**: Netflix can further expand its international content offerings to cater to these markets, enhancing localization efforts to appeal to a global audience.

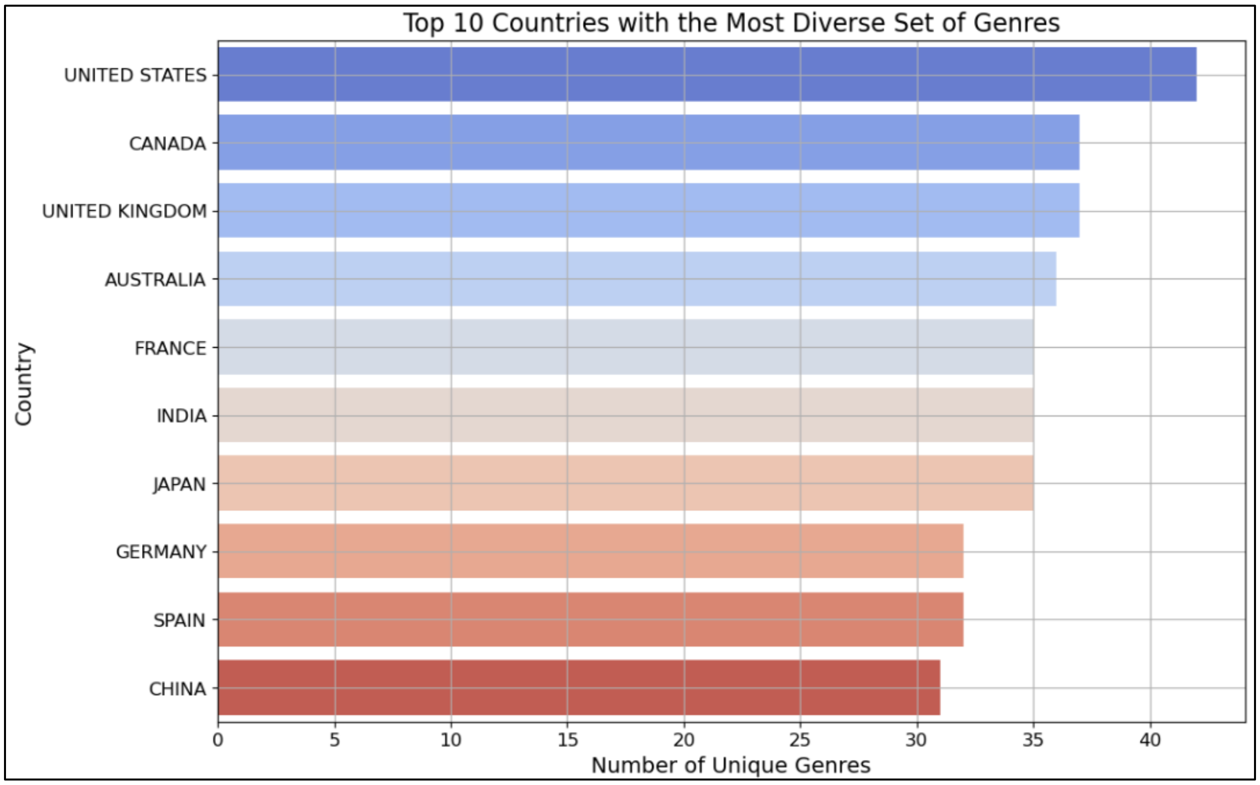
Top 10 Countries By Total Content



This section identifies the top 10 countries that contribute the most content to Netflix's catalogue.

- **Content Powerhouses**: The United States and India lead in total content contributions, showcasing their dominance in content production.
- **Strategic Implications**: Netflix should continue to strengthen partnerships in these regions to maintain a robust and diverse content library.

Top 10 Countries With The Most Diverse Set of Genres



This analysis explores which countries produce the most diverse range of genres, highlighting the richness of their content offerings.

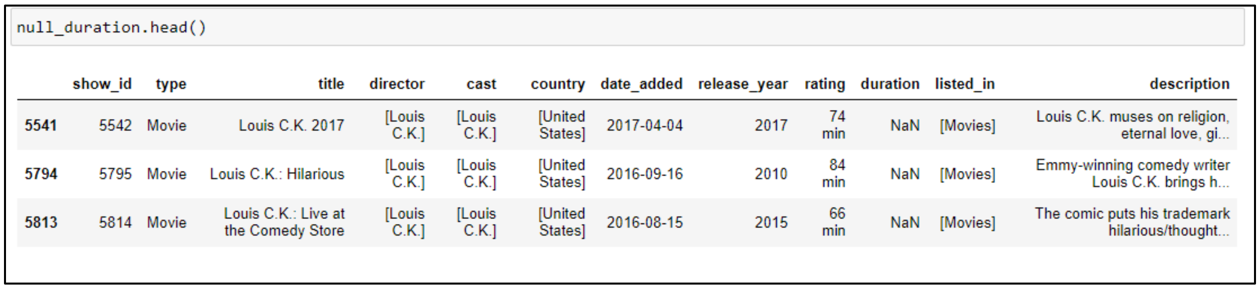
- **Diversity Leaders**: Countries like the United States and United Kingdom exhibit the most genre diversity, likely due to their multicultural societies and well-developed entertainment industries.
- **Strategic Implications**: Netflix can emphasize content diversity from these countries in its global marketing strategies, promoting a wide array of genres to cater to diverse audience tastes.

The insights from this analysis suggest that Netflix should continue to invest in diverse content offerings, particularly in emerging genres and international movies. Regional preferences should guide content curation and marketing efforts, ensuring that Netflix remains a leader in the global streaming market. Further research could involve analysing viewership data to better understand audience engagement with different genres, exploring the impact of global events on genre popularity, and investigating the role of Netflix Originals in shaping genre trends.

10. Challenges/Limitations

Wrongly Filled Columns

The Netflix dataset had some of their column values swapped. Such as in this example:



```
null_duration.head()
```

	show_id	type	title	director	cast	country	date_added	release_year	rating	duration	listed_in	description
5541	5542	Movie	Louis C.K. 2017	[Louis C.K.]	[Louis C.K.]	[United States]	2017-04-04	2017	74 min	NaN	[Movies]	Louis C.K. muses on religion, eternal love, gi...
5794	5795	Movie	Louis C.K.: Hilarious	[Louis C.K.]	[Louis C.K.]	[United States]	2016-09-16	2010	84 min	NaN	[Movies]	Emmy-winning comedy writer Louis C.K. brings h...
5813	5814	Movie	Louis C.K.: Live at the Comedy Store	[Louis C.K.]	[Louis C.K.]	[United States]	2016-08-15	2015	66 min	NaN	[Movies]	The comic puts his trademark hilarious/thought...

The rating columns for were filled with the duration and the duration column with NaN values. This error was only spotted when we tried to perform the 3rd normal form operations for the tables.

NaT Values

The NaT values (which are NA values for data type date columns) were only found at the loading phase of the pipeline. It was during the part of table creation for “release_dates” that an error occurred with the constraints (date value cannot be NA). 12 columns were found to be occupied with “NaT” values instead of their respective “YYYY-MM-DD" format.

Dataset Exclude User Ratings

The Netflix dataset extracted from Kaggle excludes user ratings, which means that by only using the dataset we have certain limitations to work around when it comes to finding out the most popular genre, titles (movies or tv shows), etc. The rating column that was provided by the original dataset refers to the TV and film ratings for age-appropriateness, such as PG, M18, etc.

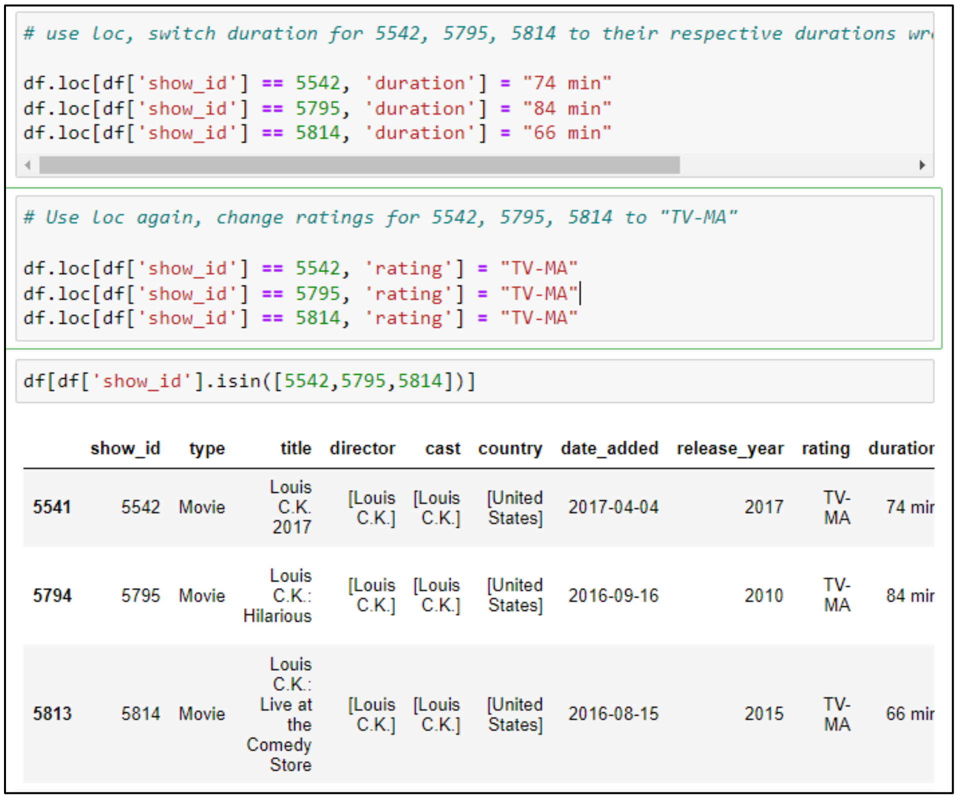
Null Values Popping Up During Data Visualisation Process

We also found out that some DataFrames such as countries were showing up with null values. Upon manual inspection of the flat file which we exported as part of the data persistence procedure, that was not the case. So instead of null values, it was showing up as empty fields. Which is why when we ran the code .isna() it was not picked up by pandas. This resulted in some of the charts being plotted weirdly or out of context.

11. Solutions

Resolving Wrongly Filled Columns

In this scenario, it was easily resolved as we can go back to the data frame and check for null values within the duration column. Which resulted in show_id: 5542, 5795 and 5814, and coincidentally were directed by Louis C.K.



```
# use loc, switch duration for 5542, 5795, 5814 to their respective durations wr

df.loc[df['show_id'] == 5542, 'duration'] = "74 min"
df.loc[df['show_id'] == 5795, 'duration'] = "84 min"
df.loc[df['show_id'] == 5814, 'duration'] = "66 min"
```

```
# Use loc again, change ratings for 5542, 5795, 5814 to "TV-MA"

df.loc[df['show_id'] == 5542, 'rating'] = "TV-MA"
df.loc[df['show_id'] == 5795, 'rating'] = "TV-MA"
df.loc[df['show_id'] == 5814, 'rating'] = "TV-MA"
```

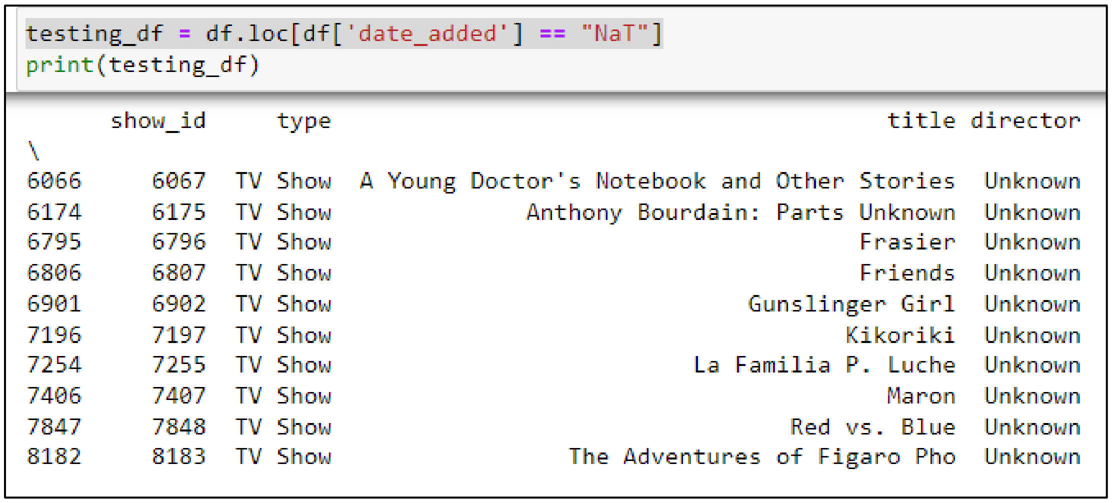
```
df[df['show_id'].isin([5542,5795,5814])]
```

	show_id	type	title	director	cast	country	date_added	release_year	rating	duratior
5541	5542	Movie	Louis C.K. 2017	[Louis C.K.]	[Louis C.K.]	[United States]	2017-04-04	2017	TV-MA	74 mir
5794	5795	Movie	Louis C.K.: Hilarious	[Louis C.K.]	[Louis C.K.]	[United States]	2016-09-16	2010	TV-MA	84 mir
5813	5814	Movie	Louis C.K.: Live at the Comedy Store	[Louis C.K.]	[Louis C.K.]	[United States]	2016-08-15	2015	TV-MA	66 mir

Since it is only 3 columns, we first swapped the duration back to its intended column 'duration'. And by fill up the actual rating for the three shows as "TV-MA".

Resolving NaT Values

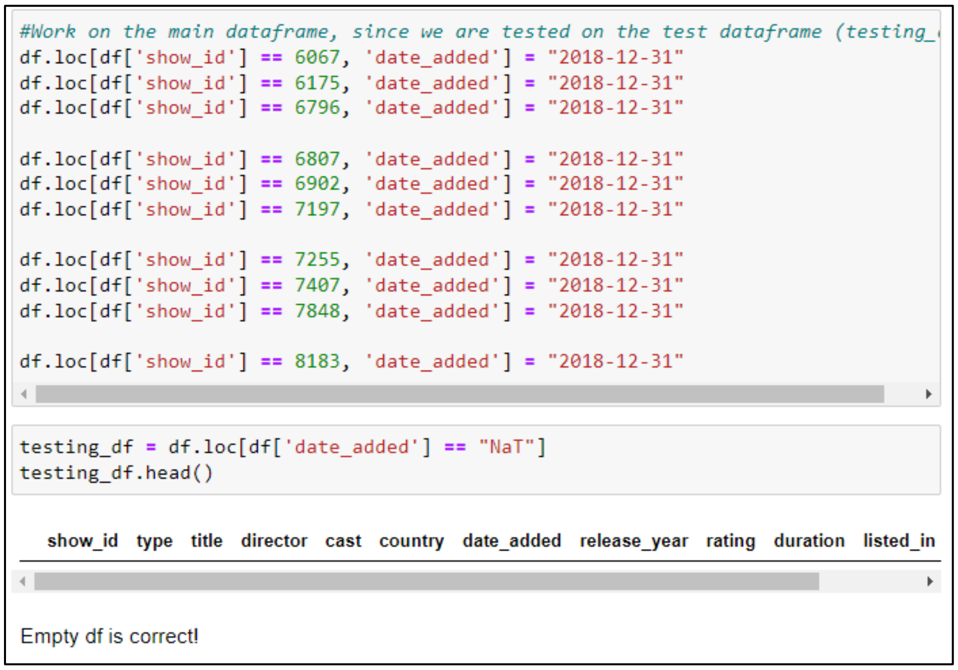
Like the scenario as before, we can go back to the transform phase and pinpoint which columns are showing up as “NaT” for the column ‘date_added’.



```
testing_df = df.loc[df['date_added'] == "NaT"]
print(testing_df)
```

```
      show_id     type                                      title director  \
6066     6067  TV Show  A Young Doctor's Notebook and Other Stories  Unknown
6174     6175  TV Show               Anthony Bourdain: Parts Unknown  Unknown
6795     6796  TV Show                                       Frasier  Unknown
6806     6807  TV Show                                       Friends  Unknown
6901     6902  TV Show                                Gunslinger Girl  Unknown
7196     7197  TV Show                                      Kikoriki  Unknown
7254     7255  TV Show                           La Familia P. Luche  Unknown
7406     7407  TV Show                                         Maron  Unknown
7847     7848  TV Show                                  Red vs. Blue  Unknown
8182     8183  TV Show                   The Adventures of Figaro Pho  Unknown
```

After locating the columns and their respective show_id, we assumed that these shows were mostly likely added around 2018, hence replacing the ”NAT” fields with ”2018-12-31” instead.



```
#Work on the main dataframe, since we are tested on the test dataframe (testing_
df.loc[df['show_id'] == 6067, 'date_added'] = "2018-12-31"
df.loc[df['show_id'] == 6175, 'date_added'] = "2018-12-31"
df.loc[df['show_id'] == 6796, 'date_added'] = "2018-12-31"

df.loc[df['show_id'] == 6807, 'date_added'] = "2018-12-31"
df.loc[df['show_id'] == 6902, 'date_added'] = "2018-12-31"
df.loc[df['show_id'] == 7197, 'date_added'] = "2018-12-31"

df.loc[df['show_id'] == 7255, 'date_added'] = "2018-12-31"
df.loc[df['show_id'] == 7407, 'date_added'] = "2018-12-31"
df.loc[df['show_id'] == 7848, 'date_added'] = "2018-12-31"

df.loc[df['show_id'] == 8183, 'date_added'] = "2018-12-31"
```

```
testing_df = df.loc[df['date_added'] == "NaT"]
testing_df.head()
```

show_id	type	title	director	cast	country	date_added	release_year	rating	duration	listed_in

Empty df is correct!

After replacing the values, we run the syntax again to ensure that there are no more ”NaT” values within the main DataFrame before we try to proceed with 3rd Normal Form.

Resolving Null Values Popping up During Data Visualisation Process

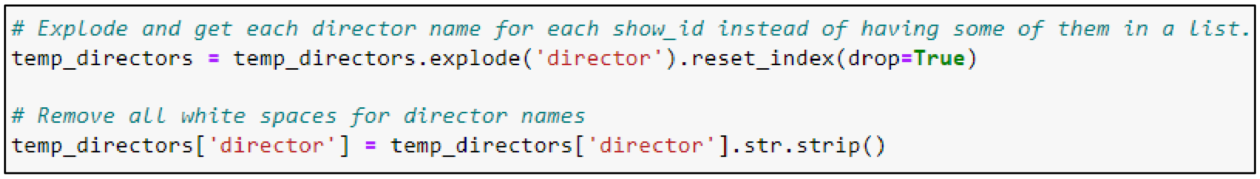
To fix this problem, we first amended the code to check for NULL values. So instead of:

```
missing_checker = countries['country_id'].isna().sum()
print(f"Number of missing or empty values in 'country_id': {missing_checker}")
```

We change included additional syntax to count in empty fields within columns like so:

```
missing_checker = countries['country_id'].isna().sum() + (countries['country_id'] == '').sum()
print(f"Number of missing or empty values in 'country_id': {missing_checker}")
```

We also went back to the transform process to include the syntax for stripping white spaces during the .explode step:



```
# Explode and get each director name for each show_id instead of having some of them in a list.
temp_directors = temp_directors.explode('director').reset_index(drop=True)

# Remove all white spaces for director names
temp_directors['director'] = temp_directors['director'].str.strip()
```

After which we ran the checker as form of validation and all NULL values and empty spaces were removed.

Resolving User Ratings

We will probably have to rely on additional data sources externally to supplement the Netflix dataset, one such example can be from the listicles generated by IMDB users. Those listicles are usually a list of movies and/or TV shows which are a user's favourites.

So, we can scrap those listicles, sorted by the actual IMDB ratings in descending order and then take the top 20 titles of each listicle.

But in this case, we have decided to pivot around this limitation by assuming that the Netflix dataset has already been built by the provider (Shivam Bansal/Kaggle) and consists of popular titles. And any titles that are already streaming on Netflix are mostly likely to be popular among its viewers globally.

12. Conclusion

Summary of Findings

The analysis of Netflix's genre distribution and trends reveals that "Drama," "Comedy," and "Documentaries" are the most common genres. Emerging genres like "International Movies" and "Documentaries" have grown in popularity. Regional differences highlight that the United States and India lead in content production, with varying genre preferences across countries. Additionally, the United States and United Kingdom produce the most diverse set of genres.

The insights gained from this analysis offer several strategic directions for Netflix:

- **Content Acquisition and Production**: Given the rise in popularity of certain genres, Netflix should consider investing more heavily in producing and acquiring content in these emerging areas, such as documentaries and international movies. This will help the platform stay relevant and meet the evolving tastes of its global audience.
- **Regional Content Strategies**: The variability in genre preferences across different countries suggests that Netflix should tailor its content offerings and marketing strategies to regional tastes. By focusing on the genres that resonate most in specific markets, Netflix can enhance viewer engagement and satisfaction.
- **Promoting Content Diversity**: Countries producing a diverse set of genres should be highlighted in Netflix’s global catalog, offering viewers a rich and varied content experience. This approach not only caters to diverse audience preferences but also promotes lesser-known genres and titles.
- **Expanding Global Reach**: With a strong demand for international movies, Netflix has an opportunity to further expand its global reach by increasing the availability and visibility of international content. This could involve more localized content production and strategic partnerships in key markets.

Web Scraping

Additionally, given a longer period of time frame to work with we can also include DataFrames from web scraping. A prototype web scraper for IMDB listicles was coded in Python/Jupyter Notebook.

The scraper make used of the BeautifulSoup and requests packages. It will first test the website to get its status code, and then parse the website's HTML code. And then it will proceed to find all 'h3' titles (which are movie titles), append them into an empty list and then converted into a DataFrame with the columns title and count, with count representing the number of times the title has been mentioned by users across different listicles on IMDB.

This is to compensate for the lack of user ratings as pointed out earlier in the challenges/limitations section of this report. It will also allow for more ways to plot our data visualisations such as finding out the most popular movie titles and then further group accordingly to their genres, etc.

For more information, refer to section 2 of Appendix.

Further Optimisations

We can also convert the data types for movie durations in string format into time format. And create functions for data validation instead of repeating the same set of codes for different columns within a DataFrame. This will improve the code readability within the Jupyter Notebook.

Loading into Cloud Database/Link with Microsoft BI

More data visualization can actually be done in Microsoft Power BI since it allows joining of tables, easy drag & drop interface and setting dimensions. It will also be useful for anyone who is looking at our project but with no technical background as it is an interactive dashboard. So with this concept in mind, perhaps we can load our tables from PostgreSQL into a cloud database such as Azure SQL Database and Power BI.

13. References

[1] J. Cherian, “Kaggle API – the missing Python documentation,” *Techno Whisp*, May 31, 2023. https://technowhisp.com/kaggle-api-python-documentation/

[2] “Getting started — pandas 2.2.2 documentation.” https://pandas.pydata.org/docs/getting_started/index.html

[3] “SQLAlchemy Documentation — SQLAlchemy 2.0 documentation.” https://docs.sqlalchemy.org/

[4] “PostgreSQL: documentation,” *The PostgreSQL Global Development Group*. https://www.PostgreSQL.org/docs/

[5] H. Team, “How to Delete a Character from a String in Python,” *Hostman*. https://hostman.com/tutorials/how-to-delete-a-character-from-a-string-in-python/

14. Appendix

Section 1. 3NF Tables

The original table can be found on page 7 [6. Source Of Data].

And after performing the normalisation up till to their respective 3rd Normal Form, we have the following tables:

- PK = Primary Key
- FK = Foreign Key

Table 1: netflix_titles

Column name	dtype	description
show_id [PK]	int64	The id of the Netflix movie or tv show.
show_type	string	The type of show (movie or tv show).
title	string	The title of the Netflix movie or tv show.
release_year	int64	The release year of the movie or tv show.
rating	string	The TV ratings (e.g. PG13)
description	string	Description of the movie or tv show.

Table 2: movies

Column name	dtype	description
show_id [PK/FK]	int64	The id of the Netflix movie or tv show.
duration	string	The total duration of the movie in minutes (e.g. 60 min)

Table 3: tv_shows

Column name	dtype	description
show_id [PK/FK]	int64	The id of the Netflix movie or tv show.
no_of_seasons	string	The number of seasons for the tv show. (e.g. 2 seasons)

Table 4: release_dates

Column name	dtype	description
show_id [PK/FK]	int64	The id of the Netflix movie or tv show.
dates_added	string	The date it was added on Netflix. (e.g. YYYY-MM-DD)

Table 5: directors

Column name	dtype	description
director_id [PK]	int64	The id of the director.
director_name	string	The name of the director.

Table 6: directs (Junction/Linking Table)

Column name	dtype	description
show_id [FK]	int64	The id of the Netflix movie or tv show.
director_id [FK]	int64	The id of the director.

Table 7: casts

Column name	dtype	description
cast_id [PK]	int64	The id of the cast.
cast_name	string	The name of the cast.

Table 8: acts (Junction/Linking Table)

Column name	dtype	description
show_id [FK]	int64	The id of the Netflix movie or tv show.
cast_id [FK]	int64	The id of the cast.

Table 9: countries

Column name	dtype	description
country_id [PK]	int64	The id of the country.
country_name	string	The name of the country.

Table 10: produces (Junction/Linking Table)

Column name	dtype	description
show_id [FK]	int64	The id of the Netflix movie or tv show.
country_id [FK]	int64	The id of the country.

Table 11: genre

Column name	dtype	description
genre_id [PK]	int64	The id of the genre.
genre_name	string	The name of the genre.

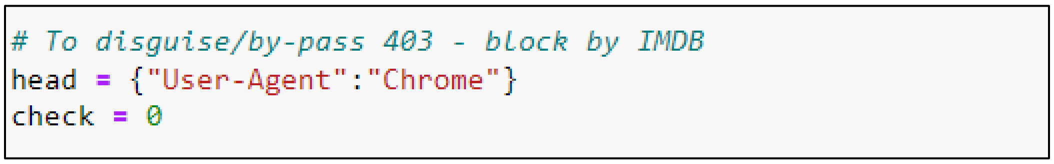
Table 12: belongs (Junction/Linking Table)

Column name	dtype	description
show_id [FK]	int64	The id of the Netflix movie or tv show.
genre _id [FK]	int64	The id of the genre.

Section 2. IMDB Listicles Scraper

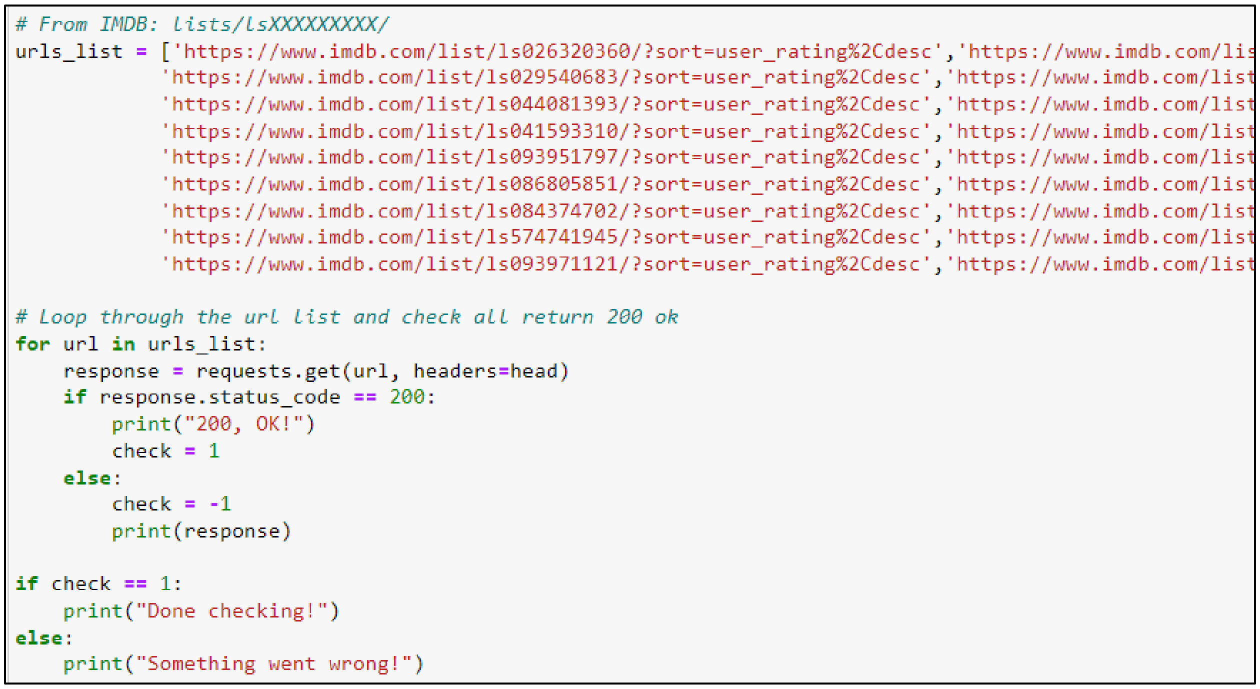
The Python Scraper uses requests, bs4 from BeautifulSoup and pandas to scrap popular movie/tv titles on Netflix across the time period of 2018 to 2021, sorted according to their popularity (descending order). These listicles are made by different actual users on IMDB, so the results scrapped will not be bias and has some randomness to it.

So it will first use a header to disguise the scraper as "Chrome" to bypass scraper restrictions from IMDB, as without it, we will fetch an error status code of 403:



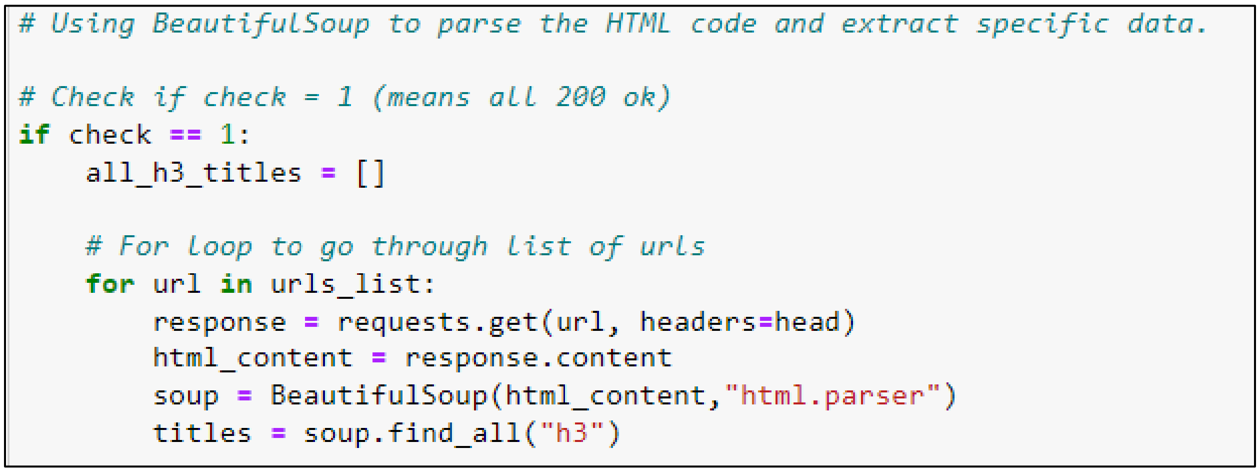
```
# To disguise/by-pass 403 - block by IMDB
head = {"User-Agent":"Chrome"}
check = 0
```

Afterwards, it will traverse through the lists of URL, using a for loop to get all the status code for each link and make sure it is all 200 (which means the request was successful). We set a variable check = 1 which will only turn check = -1 when a non 200 status code is fetch, in that scenario it will print an error message.



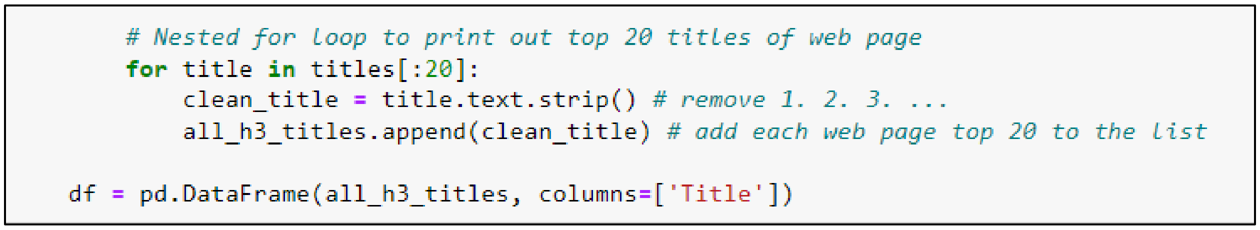
```
# From IMDB: lists/lsXXXXXXXXX/
urls_list = ['https://www.imdb.com/list/ls026320360/?sort=user_rating%2Cdesc','https://www.imdb.com/lis
             'https://www.imdb.com/list/ls029540683/?sort=user_rating%2Cdesc','https://www.imdb.com/list
             'https://www.imdb.com/list/ls044081393/?sort=user_rating%2Cdesc','https://www.imdb.com/list
             'https://www.imdb.com/list/ls041593310/?sort=user_rating%2Cdesc','https://www.imdb.com/list
             'https://www.imdb.com/list/ls093951797/?sort=user_rating%2Cdesc','https://www.imdb.com/list
             'https://www.imdb.com/list/ls086805851/?sort=user_rating%2Cdesc','https://www.imdb.com/list
             'https://www.imdb.com/list/ls084374702/?sort=user_rating%2Cdesc','https://www.imdb.com/list
             'https://www.imdb.com/list/ls574741945/?sort=user_rating%2Cdesc','https://www.imdb.com/list
             'https://www.imdb.com/list/ls093971121/?sort=user_rating%2Cdesc','https://www.imdb.com/list

# Loop through the url list and check all return 200 ok
for url in urls_list:
    response = requests.get(url, headers=head)
    if response.status_code == 200:
        print("200, OK!")
        check = 1
    else:
        check = -1
        print(response)

if check == 1:
    print("Done checking!")
else:
    print("Something went wrong!")
```

Once checking all URLs are working fine, the next step was to find use BeautifulSoup to parse the HTML code from those links and extract the specific data which we want. So in this case, the movie or tv show titles are actually h3 classes. So we can simply use another for loop to get find all the h3 classes to get the title names.



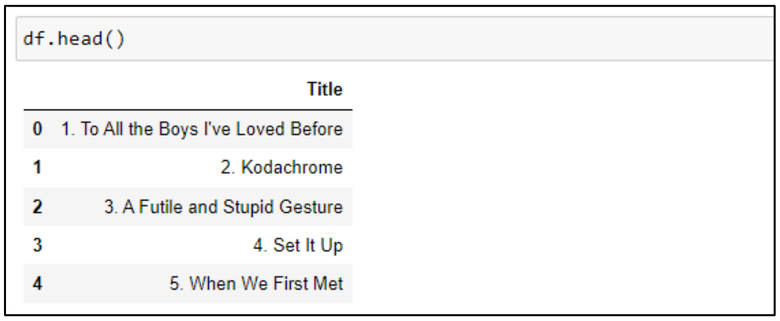
```
# Using BeautifulSoup to parse the HTML code and extract specific data.

# Check if check = 1 (means all 200 ok)
if check == 1:
    all_h3_titles = []

    # For loop to go through list of urls
    for url in urls_list:
        response = requests.get(url, headers=head)
        html_content = response.content
        soup = BeautifulSoup(html_content,"html.parser")
        titles = soup.find_all("h3")
```

However, we will still need to store all those information into a list, but we do not want everything from the listicles as it will take too long to process. Hence, we slice the collated titles for each URL using [:20] – which is to get the top 20, strip the title by removing the numbers of each title and finally append them into one long list.



```
        # Nested for loop to print out top 20 titles of web page
        for title in titles[:20]:
            clean_title = title.text.strip() # remove 1. 2. 3. ...
            all_h3_titles.append(clean_title) # add each web page top 20 to the list

    df = pd.DataFrame(all_h3_titles, columns=['Title'])
```

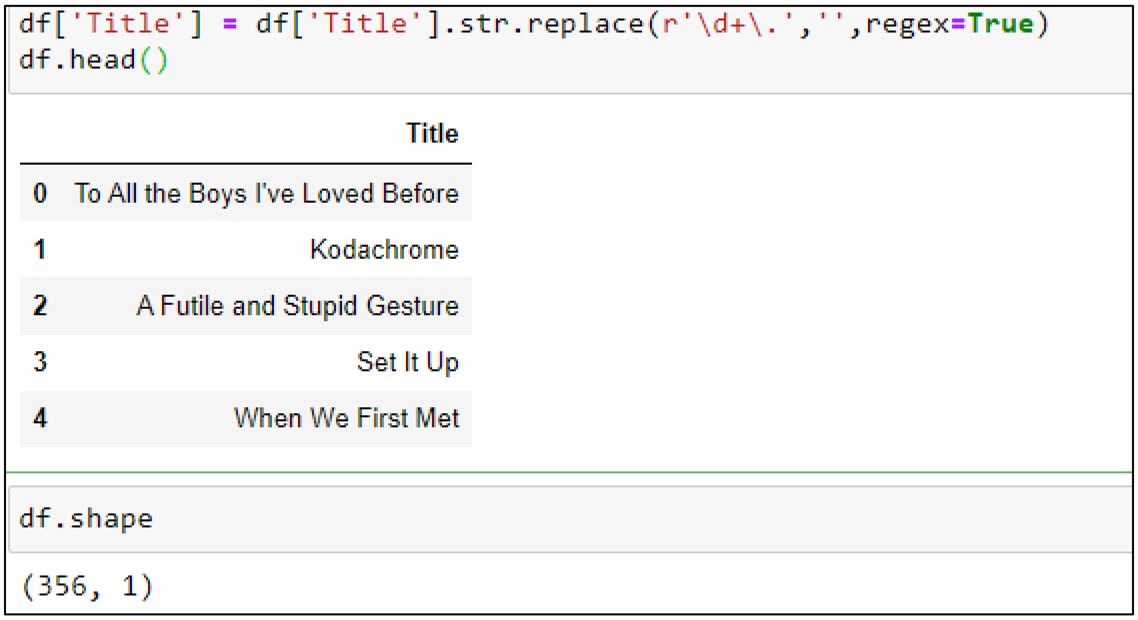
And we store them into a DataFrame with columns named 'Title'. The result will be something like this:



```
df.head()
```

	Title
0	1. To All the Boys I've Loved Before
1	2. Kodachrome
2	3. A Futile and Stupid Gesture
3	4. Set It Up
4	5. When We First Met

However, as shown above, the result is still not ideal as the labels (1., 2., …) are still within the DataFrame. The solution to this is actually using regex and delete characters from a string. More information can be found within this tutorial in the reference page [5].



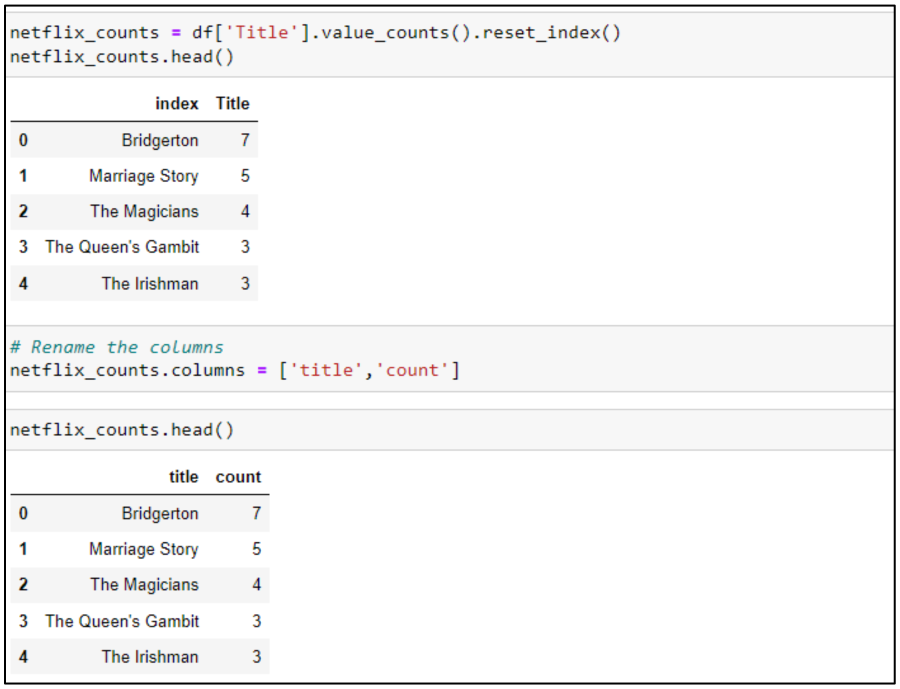
```
df['Title'] = df['Title'].str.replace(r'\d+\.','',regex=True)
df.head()
```

	Title
0	To All the Boys I've Loved Before
1	Kodachrome
2	A Futile and Stupid Gesture
3	Set It Up
4	When We First Met

```
df.shape
```

```
(356, 1)
```

The final steps are to count the number of times each title has appeared within the DataFrame, create a new column name ‘count’ to store it next to their respective title. This can be done using the following code and will result in something similar to this:



```
netflix_counts = df['Title'].value_counts().reset_index()
netflix_counts.head()
```

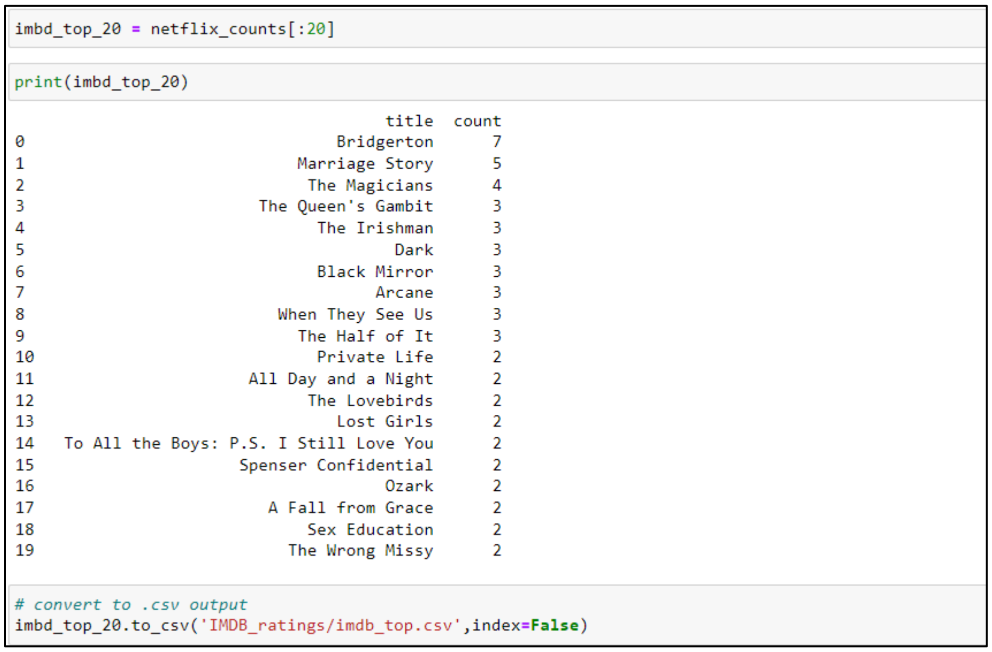
```
               index  Title
0         Bridgerton      7
1     Marriage Story      5
2       The Magicians      4
3  The Queen's Gambit      3
4        The Irishman      3
```

```
# Rename the columns
netflix_counts.columns = ['title','count']
```

```
netflix_counts.head()
```

```
               title  count
0         Bridgerton      7
1     Marriage Story      5
2       The Magicians      4
3  The Queen's Gambit      3
4        The Irishman      3
```

And we can use another slicing to get the final top 20 from this DataFrame and then export it as a flat file (.csv) like so:



```
imbd_top_20 = netflix_counts[:20]
```

```
print(imbd_top_20)
```

```
                                     title  count
0                               Bridgerton      7
1                           Marriage Story      5
2                            The Magicians      4
3                       The Queen's Gambit      3
4                             The Irishman      3
5                                     Dark      3
6                             Black Mirror      3
7                                   Arcane      3
8                         When They See Us      3
9                           The Half of It      3
10                            Private Life      2
11                      All Day and a Night     2
12                           The Lovebirds      2
13                              Lost Girls      2
14    To All the Boys: P.S. I Still Love You    2
15                     Spenser Confidential     2
16                                   Ozark      2
17                        A Fall from Grace     2
18                           Sex Education      2
19                         The Wrong Missy      2
```

```
# convert to .csv output
imbd_top_20.to_csv('IMDB_ratings/imdb_top.csv',index=False)
```