**Practical Assignment:**

**Coursework**

**Designing and Implementing a Database Structure**

**[Mohammad Faraj P418072]**

**Database Design and Implementation**

**16/4/2024**

I confirm that this assignment is my own work. Where I have referred to academic sources, I have provided in-text citations and included the sources in the final reference list.

**Introduction and Context**

The "Internet Movies Database" website, IMDb, was scraped to gather a list of movies that was then compiled into a CSV file named "movies.csv". The objective was to standardize the data in the file and add it to a database using Python code and SQL statements. This method ensures accurate access to the information by utilizing SQL statements. The report will encompass the normalization process, entity-relationship diagram, code implemented to generate tables and input values, SQL statements utilized to extract information and the corresponding outputs, and a reflective segment that will evaluate the design's strengths, weaknesses, and potential areas of improvement.

**Normalization Process**

To begin the process, it is important to normalize the CSV file housing an extensive 15-column movie database. These columns comprise of crucial details such as the movie's name, rating, genre, year, country-specific release information, score, votes, director, writer, star, shooting location, budget, gross income, production company, and runtime. Given the diverse nature of this information, it is essential to ensure proper normalization to avoid any issues such as update, deletion, or insertion anomalies.

**Normalizing to First Normal Form (1NF):**

Ensuring that each row is unique, all values are atomic, and there are no repeating groups in the table is crucial for bringing it to the first normal form. Although there are no repeating groups in the table, the "released" column contains two types of data, date and country, which can still be considered atomic, since it does not affect any other data in the table. However, there are non-unique rows in the table, such as two movies with the same name "The Captain," and there may be one production company that produced multiple movies. As a result, unique IDs must be created, and the table must be split into two.

The table will be split into two distinct tables, the first table, "movie\_production", will consist of crucial information such as the movie's name, director, stars, writer, and production company. This data can have an impact on other records, as a single writer may create scripts for multiple films, resulting in a one-to-many relationship with the movie's name. However, the name can be used to identify the production company and other pertinent information only if it is unique. Thus, to maintain uniqueness, unique IDs must be created for these fields, alongside a primary key named "production ID".

The second table however, “movie\_general”, will include all secondary information that does not depend on or rely on any other information, like the rating, genre, year, released, votes, score, country, budget, gross, and runtime. With a unique ID named “general ID” to serve as a primary key. It will be linked to the first table using the “production ID” however, it shall be a foreign key in the “movie\_general” table referencing the “movie\_production” table.

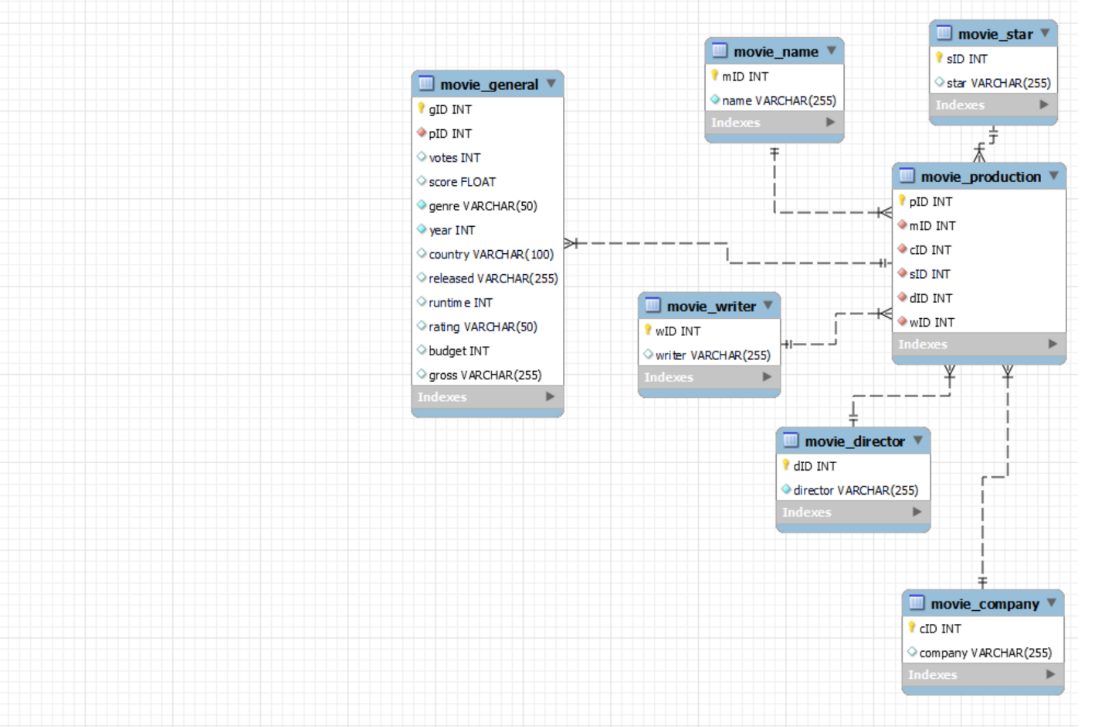
**Normalizing to Second Normal Form (2NF):**

To convert tables to the second normal form (2NF), all non-key attributes in the tables must rely on the primary key after applying normalization to the first normal form (1NF). The "movie\_general" table already meets this requirement, since all attributes depend on the primary key, "general ID". However, in the "movie\_production" table, each non-key attribute is dependent on its ID, not the primary key "production ID". As a result, the table should be split into six new tables, each with only one attribute and its corresponding ID, with the ID serving as the primary key. There will also be one "movie\_production" table that has the "production ID" as its primary key and includes all the other primary keys from the other five tables, to serve as foreign keys to link all tables to it.

Now, the tables are not just in 2NF, but they are also in 3NF since the 3NF points that there must not be any transitive dependencies in the tables, and there are not.

**Entity-relationship Diagram**

Here is an entity-relationship diagram, Figure 1, that presents and clarifies the relation between the tables mentioned before.

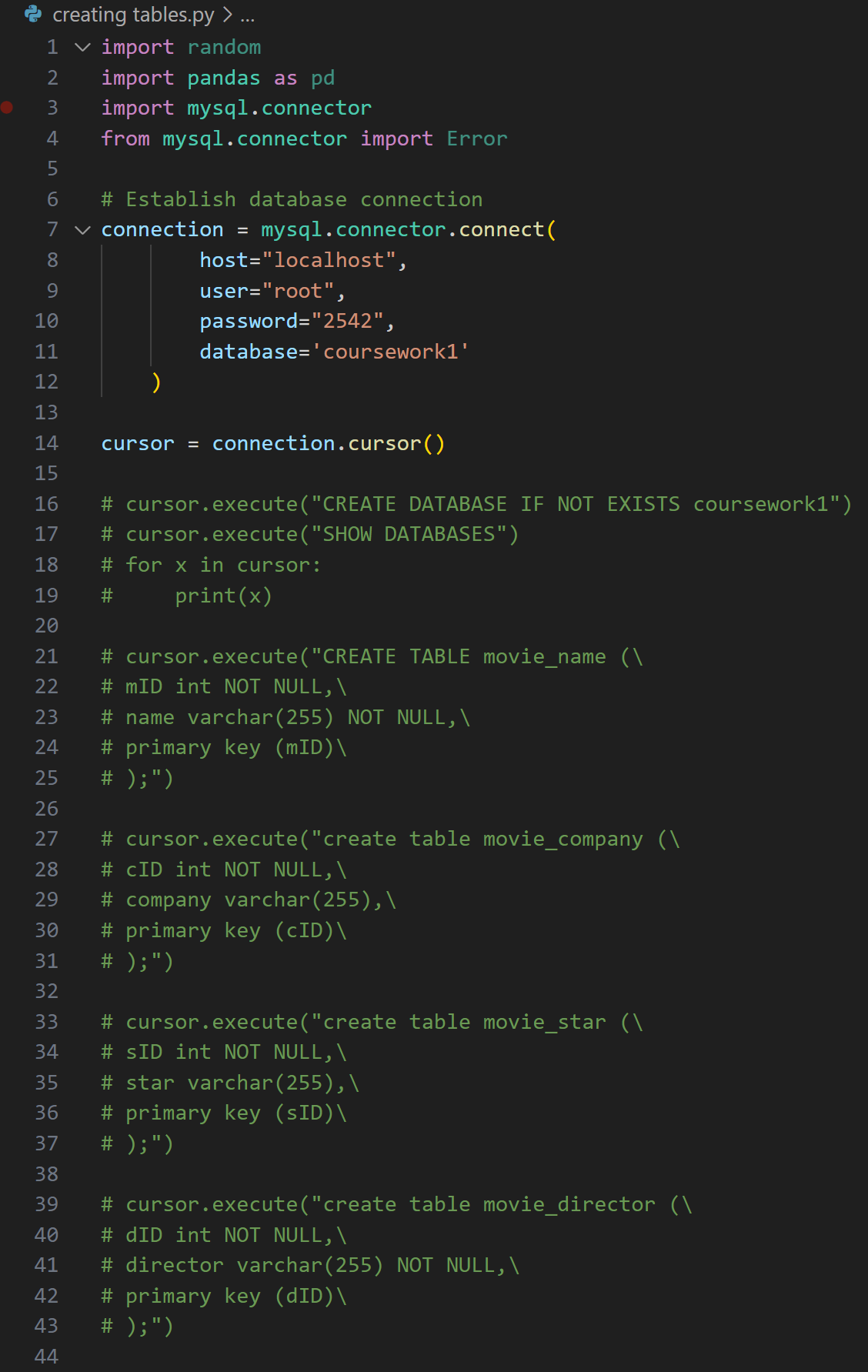


**Figure 1:** An Entity-relationship Diagram

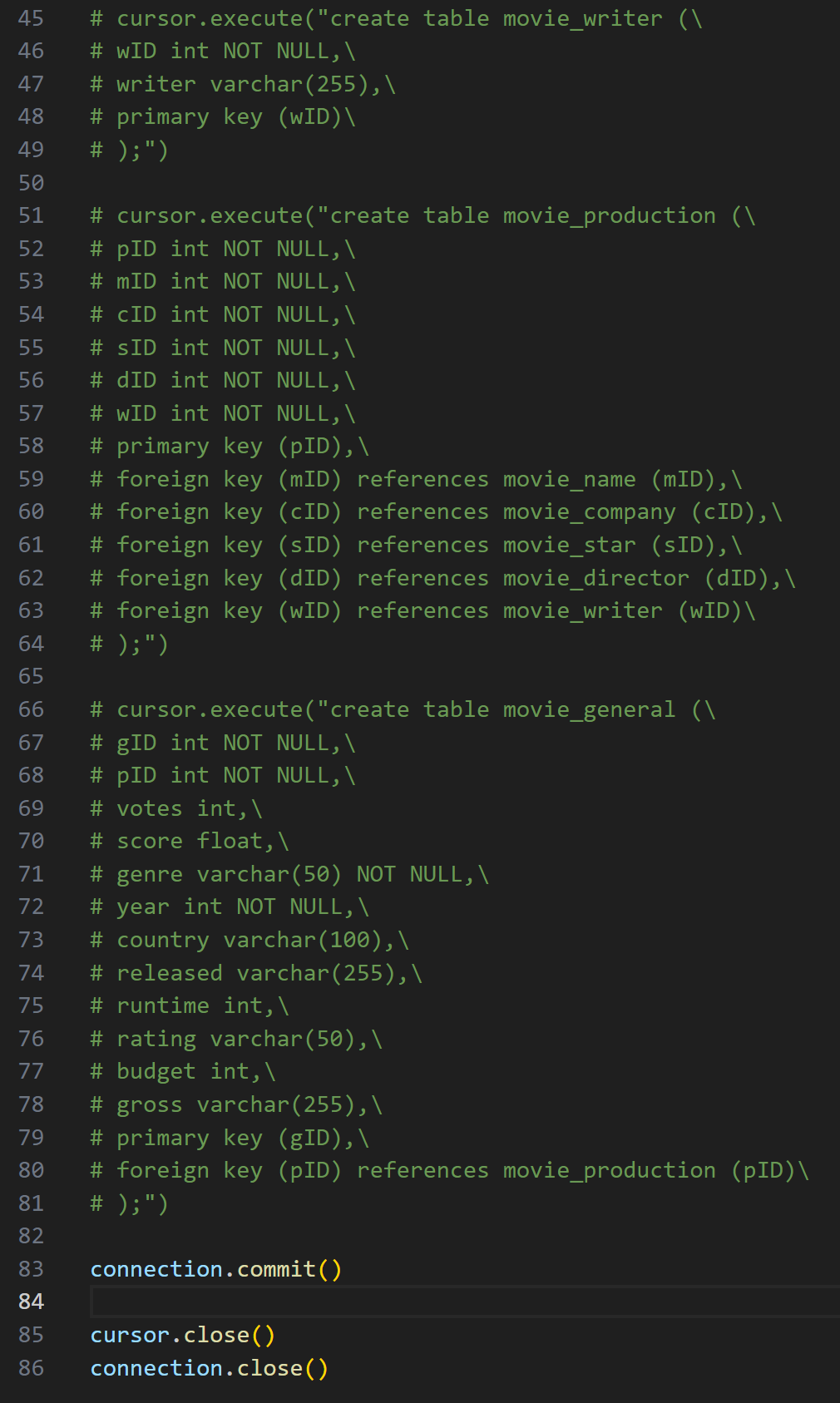
As how is clear in Figure 1, the relation between the five tables containing the non-key attributes and their IDs, and the “movie\_production” tables is a non-identifying one-to-many (1:n) relationship. While the relation between the “movie\_production” table and the “movie\_general” is also a non-identifying one-to-many relationship.

**SQL Database Implementation using Python**

In this section, the Python code, along with the SQL statements, used to create the database “coursework1” and the tables, will be presented in Figures 2 and 3, as in below.



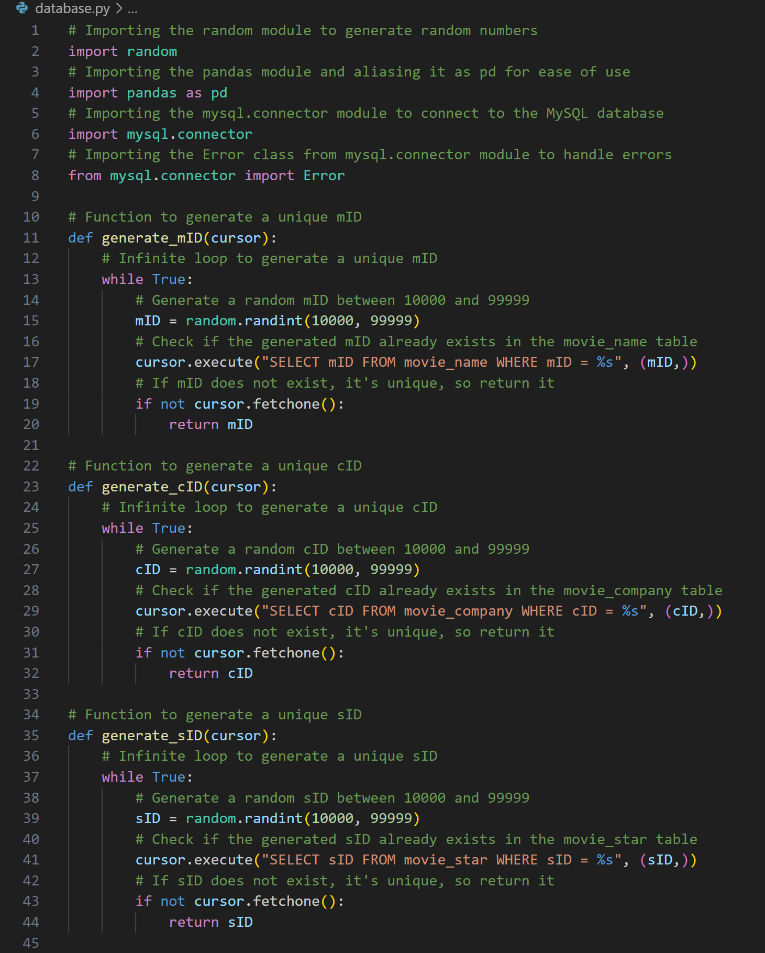
**Figure 2:** Code for creating the database and tables, displayed in Visual Studio



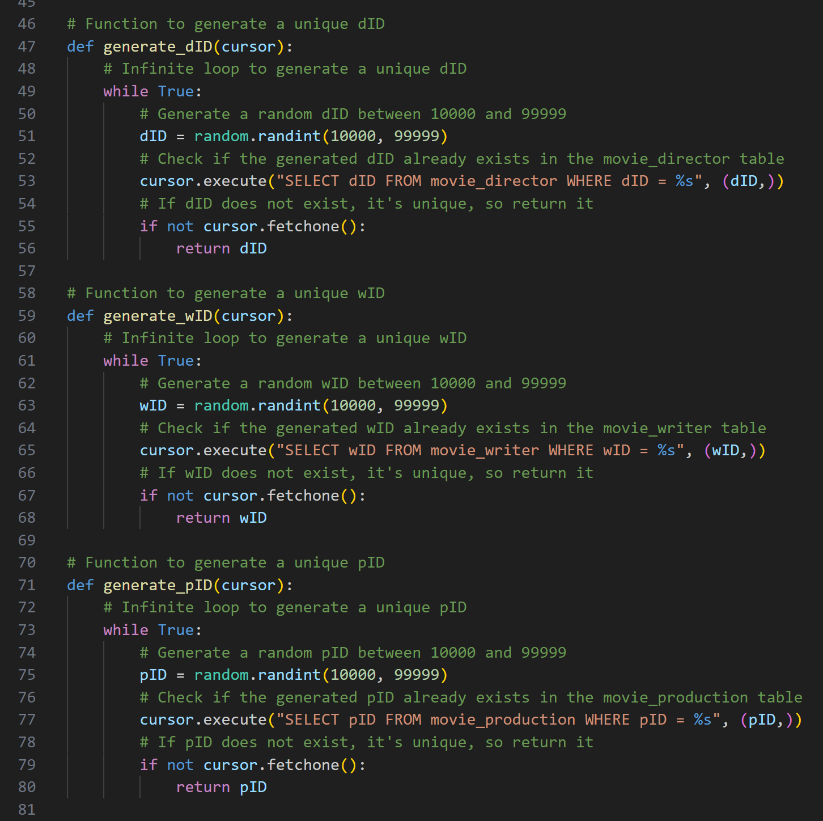
**Figure 3:** Completion of the Python code

**Data Importing using Python**

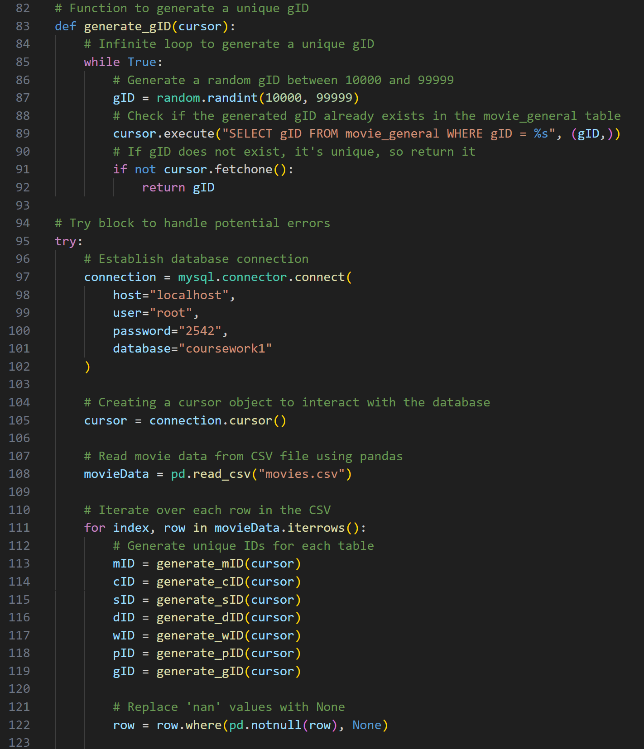
In Figures 4 through 8, there will be the code for importing information from the CSV file “movies” and inserting them into the tables created before.



**Figure 4:** Python code for importing information displayed in Visual Studio



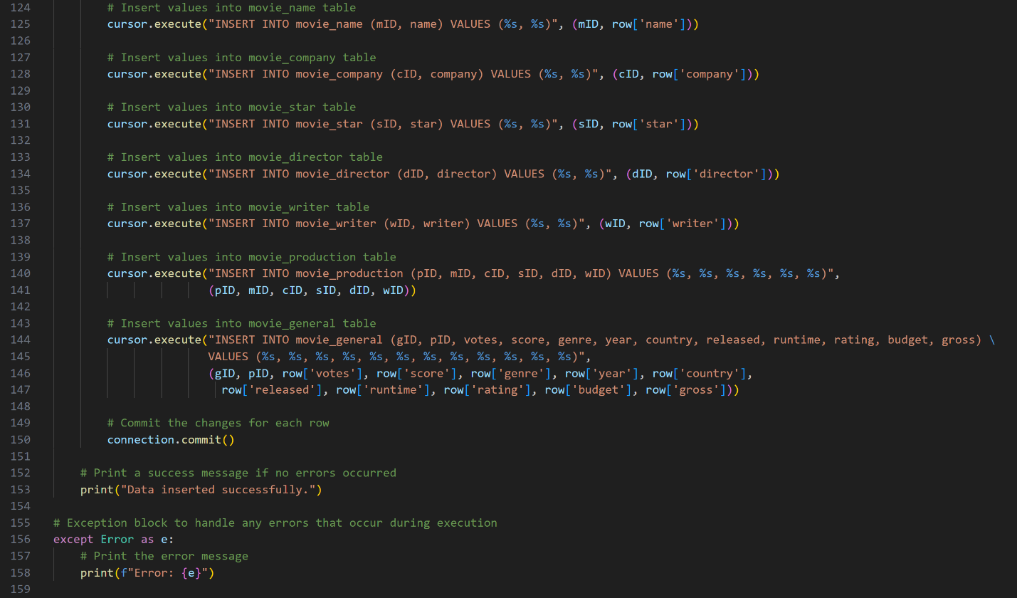
**Figure 5:** The rest of the code



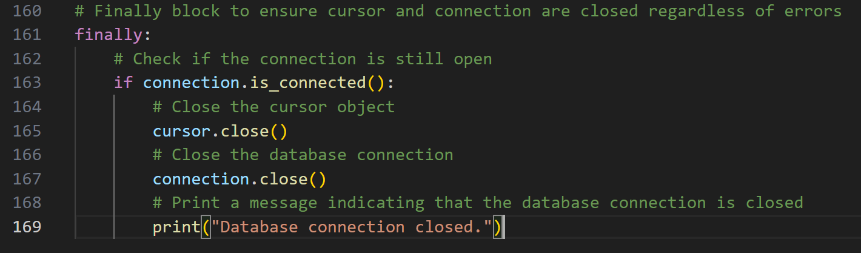
**Figure 6:** The rest of the code

As in Figures 4, 5, and 6, there are some functions to generate the unique IDs needed for the tables, where the “random” library was used to generate the random number for each ID, and it only generates a unique ID for each row, by checking if it is present already in the table or not.

Also to deal with ‘nan’ values, this line of code: “row = row.where(pd.notnull(row), None)”, replaces 'nan' values in a Pandas DataFrame row with 'None', using the “where()” function from the Pandas library. (Geeks for Geeks, 2023)



**Figure 7:** Rest of the code



**Figure 8:** The end of the code

**SQL Statements to Show Correct Implementation**

There are 5 types of information needed to be obtained about the data in this database, and they are:

1. The average, minimum and maximum budget of drama movies.

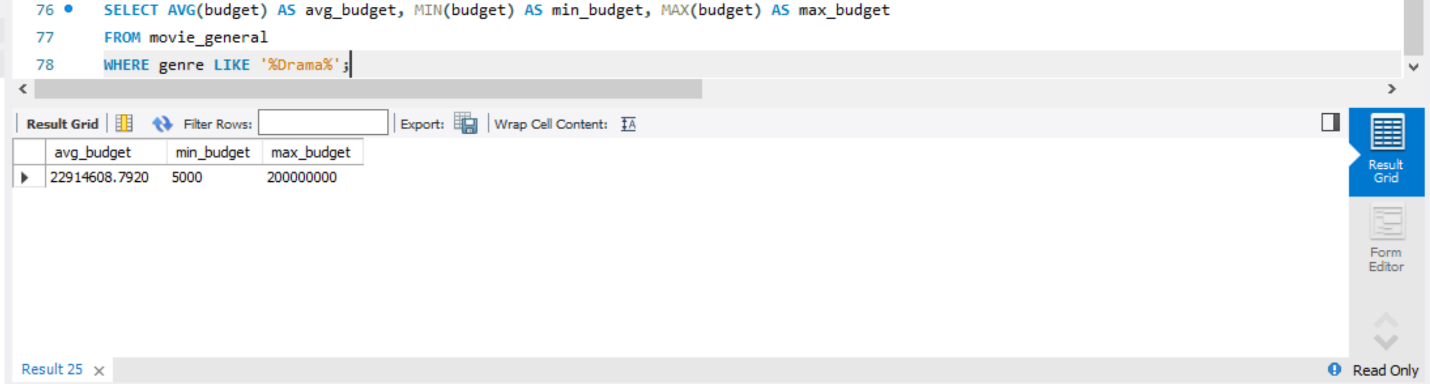
2. The minimum budget for each genre.

3. The average score of each rating (ignore not rated movies).

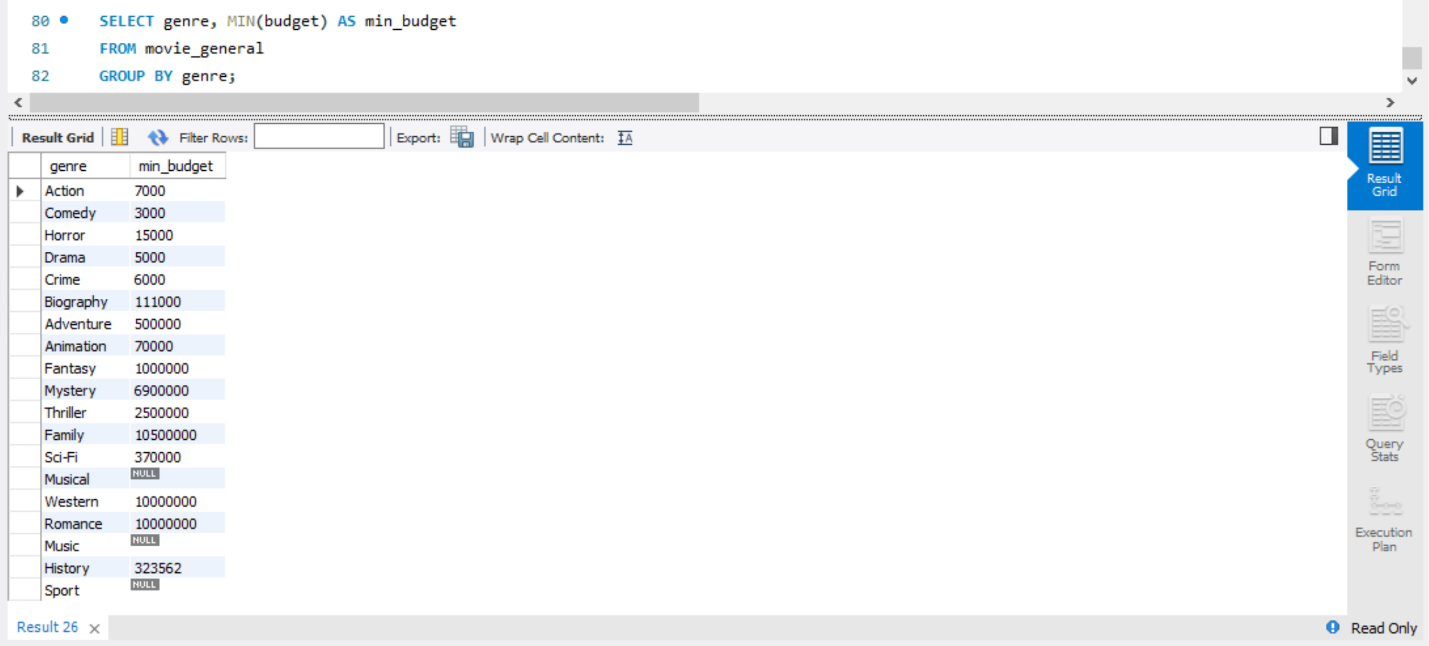
4. The average runtime of all movies between the years 2014 and 2016.

5. The number of movies released before 2000.

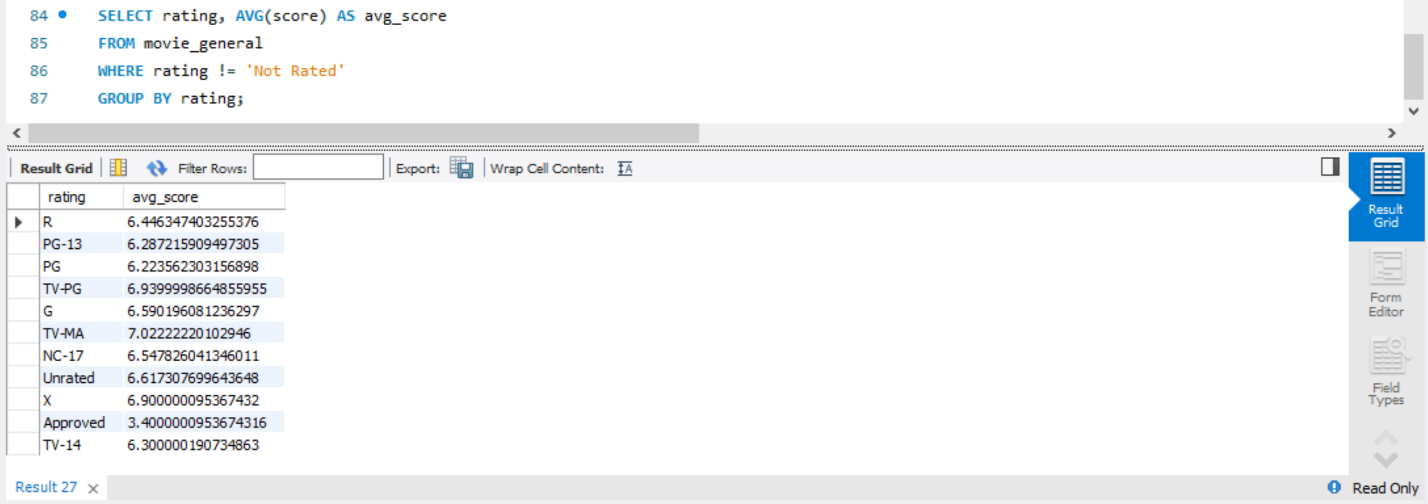
The results for these are obtained in MySQL Workbench by the following SQL statements in Figures 9 through 13.



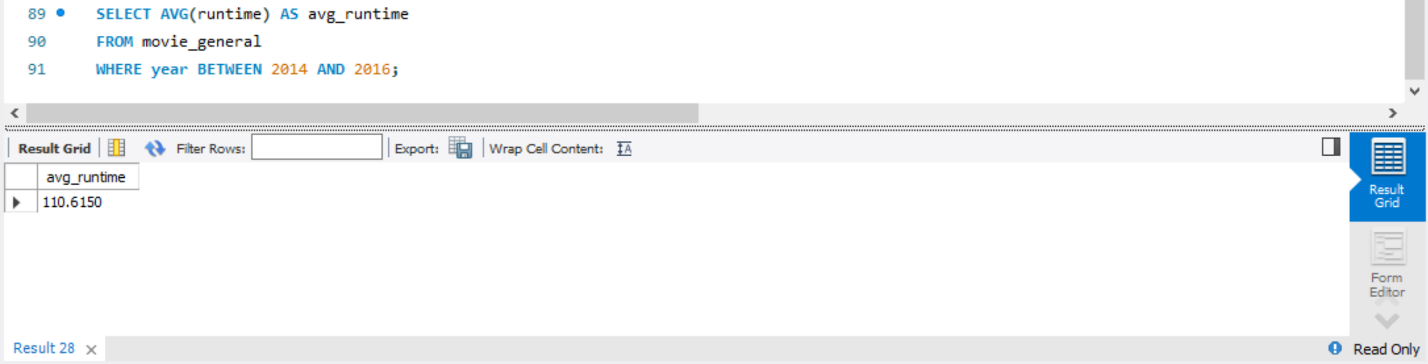
**Figure 9:** Average, minimum, and maximum budget



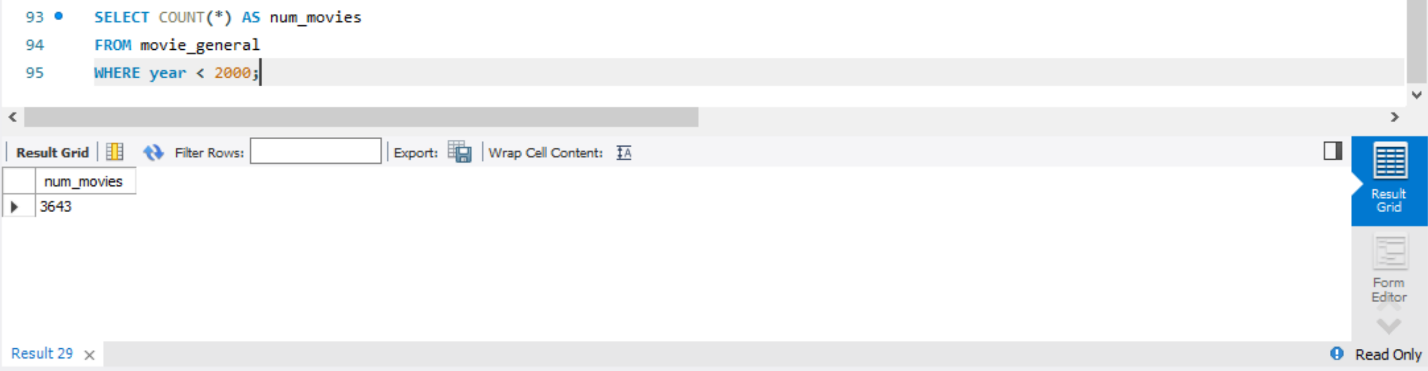
**Figure 10:** Minimum budget



**Figure 11:** Average score

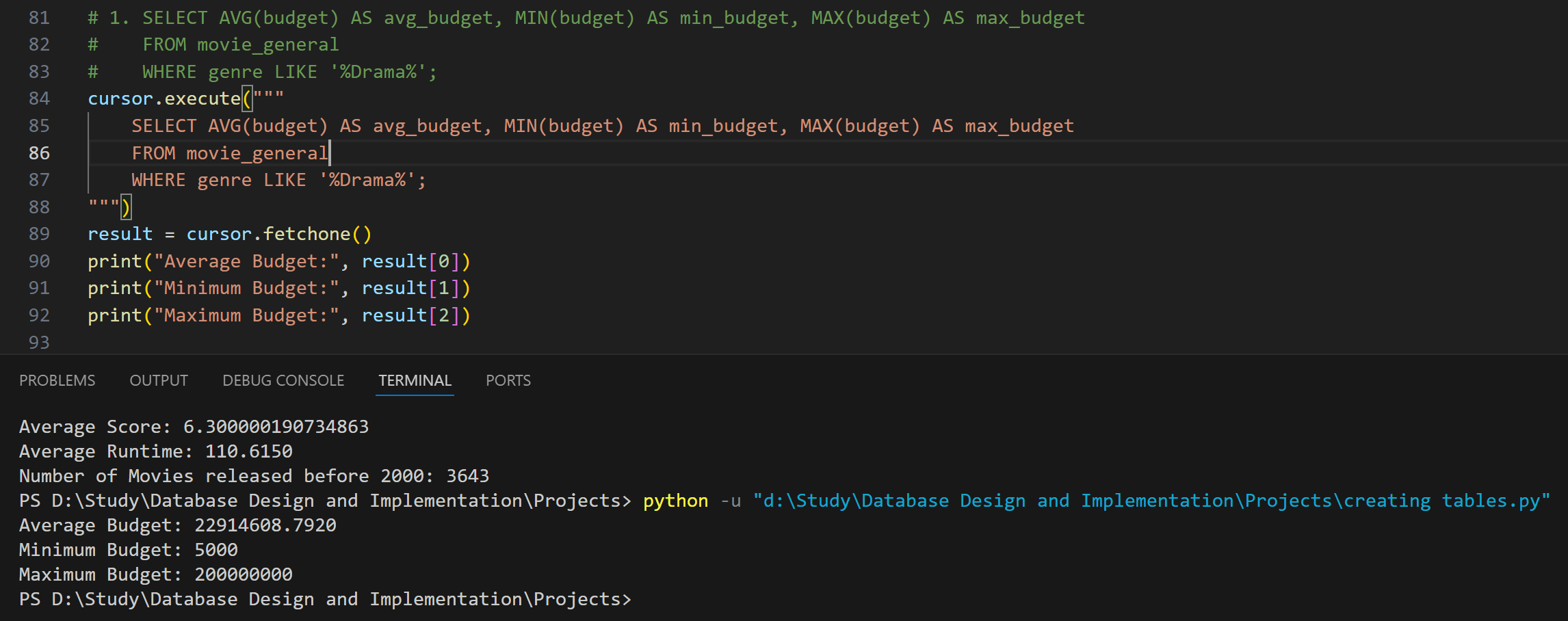


**Figure 12:** Average runtime

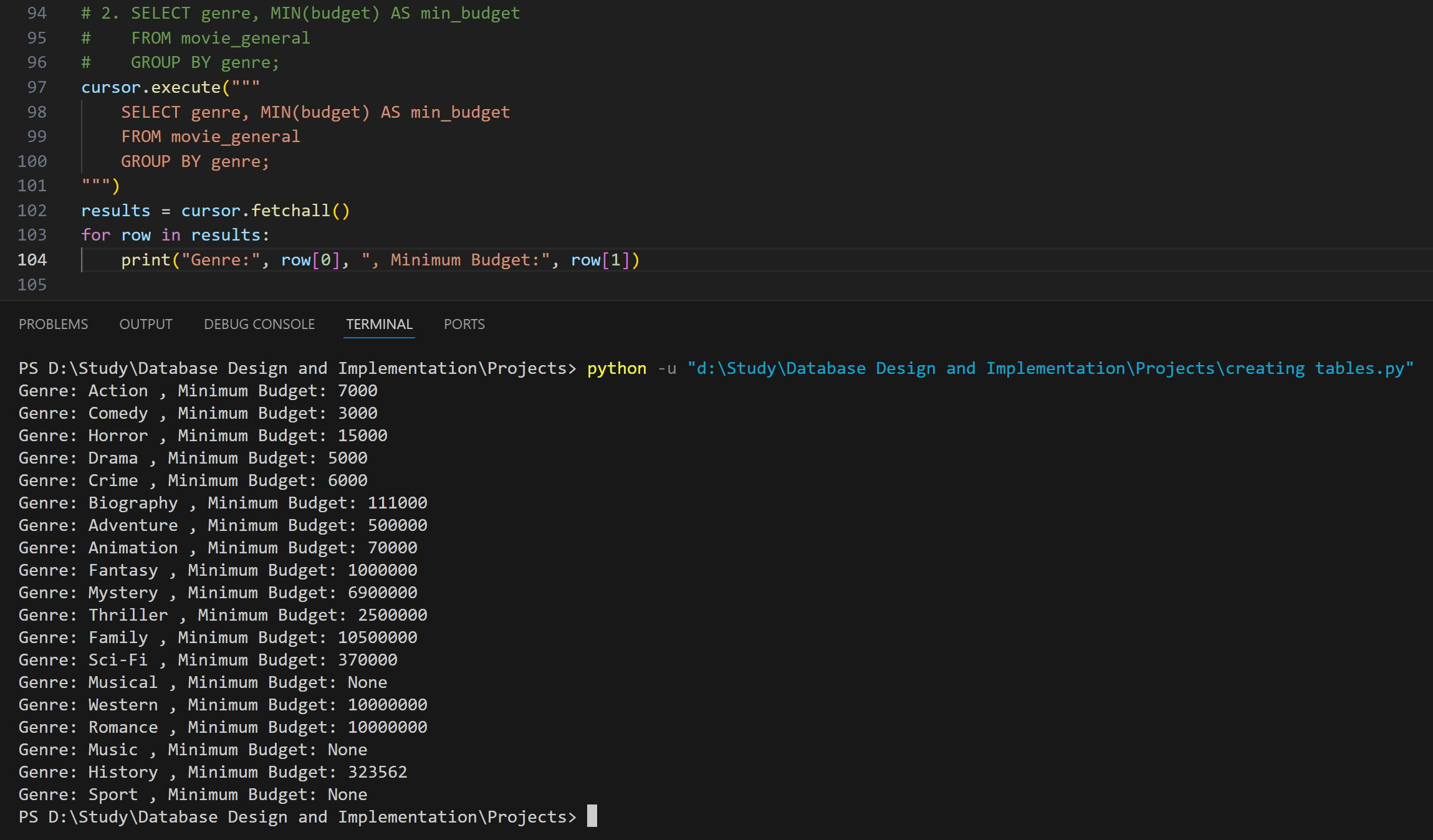


**Figure 13:** Number of movies before the year 2000

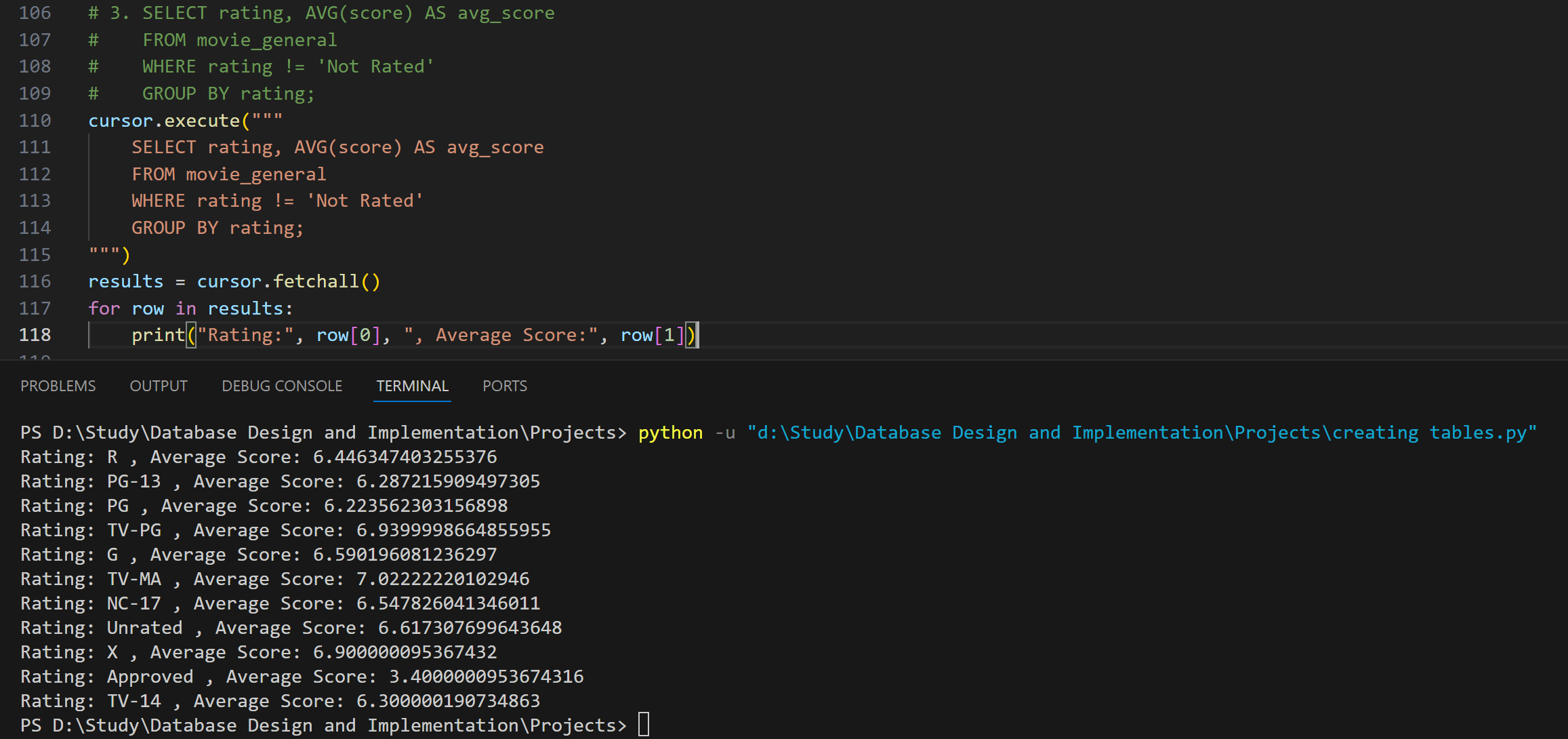
Also, the results obtained by these statements in Visual Studio, using Python, will be presented in Figures 14 through 18.



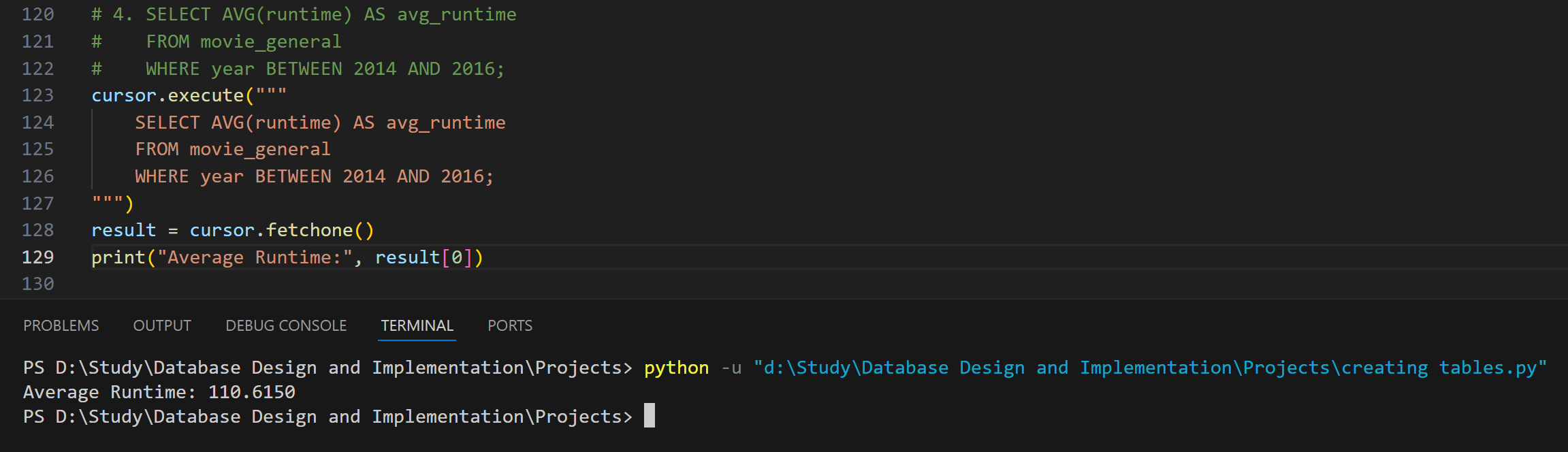
**Figure 14:** Average, minimum, and maximum budget



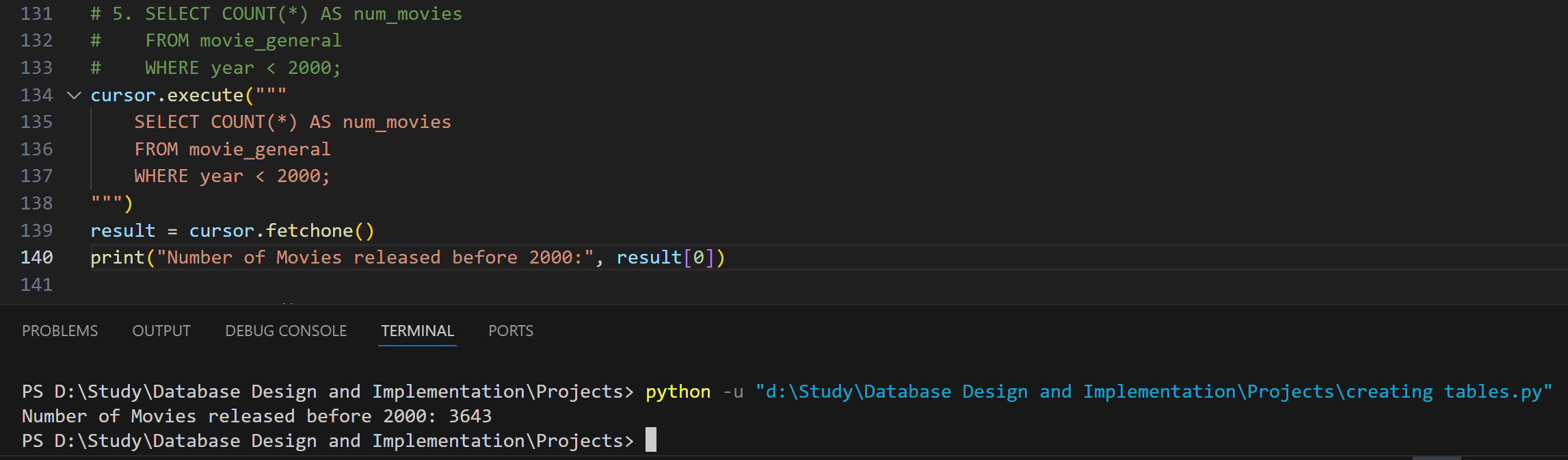
**Figure 15:** Minimum budget



**Figure 16:** Average score



**Figure 17:** Average runtime



**Figure 18:** Number of movies before the year 2000

**Reflective Segment**

This section aims to provide an in-depth analysis of the challenges encountered by the designer throughout the project. Additionally, any improvements that can be made to the design will be evaluated and their impact on the overall outcome.

**Challenges in the design:**

Throughout the design process, the initial obstacle was to standardize the database schema. At first, the "movie\_production" and "movie\_general" tables had a shared primary key, which was the "movie ID" attribute of the "movie\_name" table. This allowed users to effortlessly access all the information for each movie in other tables via the "movie ID" attribute as a foreign key. However, this approach had the potential for data anomalies and made maintaining referential integrity difficult. Additionally, it could create complications for database operations, such as updates and deletions. As a result, the design was modified to its current version, where each table has its own primary key that is not a foreign key. This guarantees the same output without any issues.

Another issue arose in the Python code for inserting values into tables. Initially, the logic was to extract all values from a particular column in the CSV file and insert them into the relevant table after assigning IDs if necessary. The code would then repeat the process with the next column. However, this approach was not efficient, as MySQL Workbench automatically rearranges rows in alphabetical order. Consequently, when moving to insert other columns, incorrect values could be assigned from one table to another, given that the ID was randomly generated rather than sequenced. Furthermore, the values would differ every time the code ran. Therefore, the logic was revised to navigate through the CSV file row by row, extract values and store them in variables before inserting them into the tables, and then move to the next row.

**Improvements to the design:**

An enhancement that could be made to the code involved in utilizing Python functions and the "random" library to generate distinct IDs. Although this method is effective and comprehensible, it may appear slightly complex when compared to an alternative technique that utilizes the "AUTO\_INCREMENT" SQL statement during table creation. The latter approach is more straightforward and eliminates the necessity of creating a specific function for each ID.

To handle 'nan' values in a CSV file and ensure their proper storage in a database, we utilized the "where()" function from the Pandas library. This function effectively transformed 'nan' values to "None". While this approach generally works well with most databases, there exists a simpler and more direct method. This method involves assigning a default value of "0" to all 'nan' values. However, this method has its drawbacks as it may lead to biased analysis in certain scenarios by concealing missing data. In our particular database, this method would have sufficed since no cell in any table holds the value "0".

**References**

Geeks for Geeks (2023) *Python | Pandas DataFrame.where().* Available from: <https://www.geeksforgeeks.org/python-pandas-dataframe-where/> [Accessed 16 April 2024].