Final Project Report - Debugging Dream Team

Alex Warren, Annie Liemohn, Jacob Harleton

Link to GitHub repository: https://github.com/alexrwarren/Final-Project-SI-206

Project Goals

For this project, we planned to gather information about paintings displayed in major art museums in the United States. We specifically aimed to gather painting objects from the Art Institute of Chicago's API, the Cleveland Museum of Art's API, and the Metropolitan Museum of Art's API. We aimed to collect the *title* of each painting and the *year* the painting was created for each API. Additionally, for the Art Institute of Chicago's API, we aimed to collect the *artist's nationality*; for the Cleveland Museum of Art's API, we aimed to collect the *artist's name;* and for the Metropolitan Museum of Art's API, we aimed to collect the *artist's gender*. We aimed to answer whether certain painter characteristics correlated to the length of painting titles. For example, do average painting title lengths differ depending on the artist's nationality, gender, or age?

Achieved Goals

We achieved gathering data from three art museum APIs: the Harvard Art Museums' API, the Cleveland Museum of Art's API, and the Metropolitan Museum of Art's API. Additionally, we gathered data from an online library API ('Open Library'). We did not achieve the goal of gathering data from the Art Institute of Chicago's API due to issues with pagination and filtering. Thus, we turned to Harvard Art Museums' API as a replacement. From Harvard's API, we gathered the *title*, *creation year*, and *height* of the art. From the Cleveland Museum of Art's API, we gathered the art's *title*, *creation year*, and *artist's name*, just as we set out to achieve. From the Metropolitan Museum of Art's API, we gathered the art's *title*, *creation year, and artist's gender*–as intended.

At one point during the project, the Cleveland API went offline, so we temporarily replaced this API with Open Library's API. Eventually, the Cleveland API began to work again. From the Open Library API, we gathered *titles*, *publication years*, and *authors' names* for fiction books in the collection.

Due to issues with the Art Institute of Chicago's API, our goals shifted. Rather than trying to answer whether certain painter characteristics correlated to painting title length, we instead achieved answers for various questions concerning the *paintings* themselves as opposed to the *painters*.

- Has the average painting height within Harvard Art Museums' galleries changed over time?
    - Answer: Painting height seems larger for paintings created during the 20th century than those of the 19th century.
- Does the average painting title length differ by museum?
    - Answer: Not significantly.
- Is the average title length of a painting different than that of a book?

    - Answer: Not significantly.
- How prevalent are female painters in the Metropolitan Museum of Art?
    - Answer: There are very few contemporary female painters compared to male painters in the Metropolitan Museum of Art's collections.
- Do painting title lengths differ across art departments in the Cleveland Museum of Art's collections?
    - Answer: Probably not. Running the code with different painting titles yields various results.

Problems We Faced

Alex: I had a smooth experience gathering painting information from the Cleveland Museum of Art API due to the detailed parameter options for filtering results. However, generating a set of unique entries every time I called the API was complicated. Eventually, I made a Piazza post about my issue and a student responded with an answer that helped me. I ended up using the 'skip' parameter as per the advice of the Piazza response, and it worked to gather a unique API response every time the function was called!

A second issue that came up during my work was trying to get enough paintings from the Art Institute of Chicago API. The main issue was that there was no ability to filter results by art type during the API request, which led to only a few of the returned responses being of the painting classification. Thus, to return enough paintings, the function had to be called upwards of 100 times. This inefficiency influenced our decision to change course and use the Harvard Art Museums' API instead. One issue with this database was gathering random/unique painting results each time, but because I had already troubleshooted this issue with the Cleveland API, we easily overcame this obstacle using pagination instructions detailed in the API documentation.

A third issue arose with the gender distribution calculation for the Metropolitan Museum of Art data. Because we're using date ranges on the x-axis, it would make the most sense to create a histogram as opposed to a bar chart. However, we struggled to generate a stacked histogram that showed the female counts 'stacked' on top of the male counts. Reading through advice on Stack Overflow did not fix our issues, so we eventually settled on the use of a bar chart so we could keep our stacked bars.

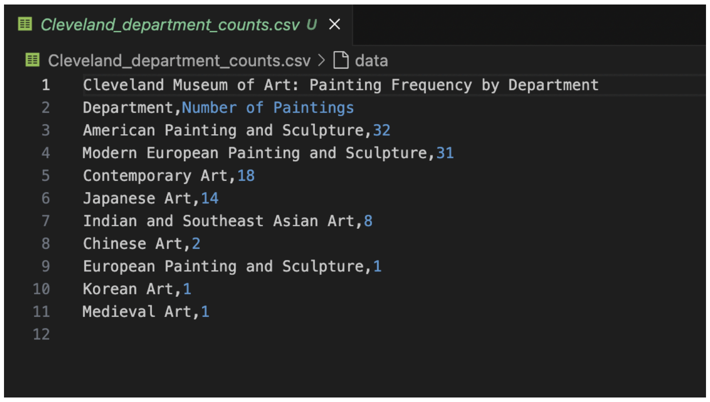
Annie: As Alex mentioned above, pulling paintings from the Chicago API presented challenges due to how few paintings were able to be pulled at a time, requiring more than ten iterations to gather the necessary 100 items from that API. The Harvard API, however, ran much easier and was able to gather enough paintings to reach the 100 item threshold in only a handful of iterations. Using parameters prior to retrieval, and filtration after retrieval, from the Harvard API, we were able to pull 25 paintings at a time and gather the *title*, *creation year*, and *height* of each painting.

The primary issue I encountered with using data from the Harvard API was when completing calculations in the calculation_average_painting_height.py file. I originally saved the total sum of all painting heights from a particular time period into values within one dictionary. From here, I looked to take the average of each sum and figure out the average painting height for each time period. This led to duplicates, however, so we edited the calculate_average_height() function within the file to instead save the sums of painting heights from a particular time period as values within nested dictionaries. This ensured each sum was stored under the correct time period. By the time the dictionary was returned at the end of the function, we also made sure to convert it back to a single (not nested) dictionary. Lastly, there was also a problem of the bars in the bar chart for Harvard_average_painting_height_in_cm.png not being in chronological order since each bar corresponds to each half-century since 1800. I solved this problem by using the .items() and sorted() functions to change how the data was saved into lists prior to creating the plot.

Jacob:

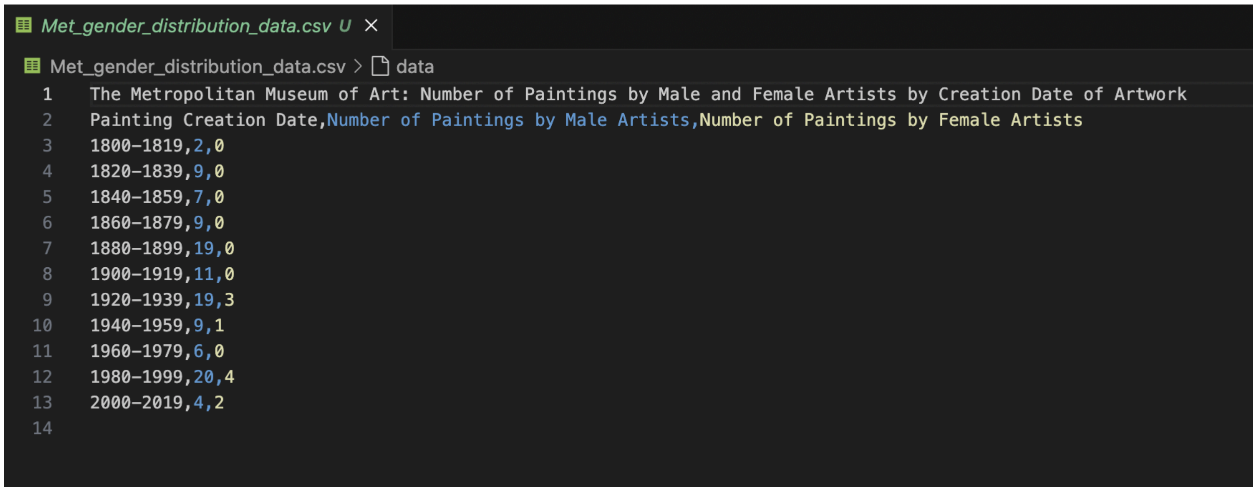
As our research topic centered around paintings, I had to make sure that within such an expansive art museum as The Met, my database included only paintings. This brought about challenges as the initial parameter in collecting information from the API was ‘medium’=paintings. However, this proved not specific enough as my research into the artwork within my insert_paintings_into_MET() function still sorted through artwork that was under the classification of 'Ephemera' or ‘Paper’. As a solution, I included a secondary line of filtration, noting down all of the classifications that originally fell through the first filtration method and identifying which types should be included in the research. Finally, I landed on acceptable_classifications = ['Paintings', 'Paintings-Decorative', 'Bark-Paintings'] | if painting.get('classification') not in acceptable_classifications: | continue, which helped me isolate the type of data that I was looking for and disregard the rest.

Calculations From Database Data


Cleveland_department_counts.csv U ×
Cleveland_department_counts.csv > data

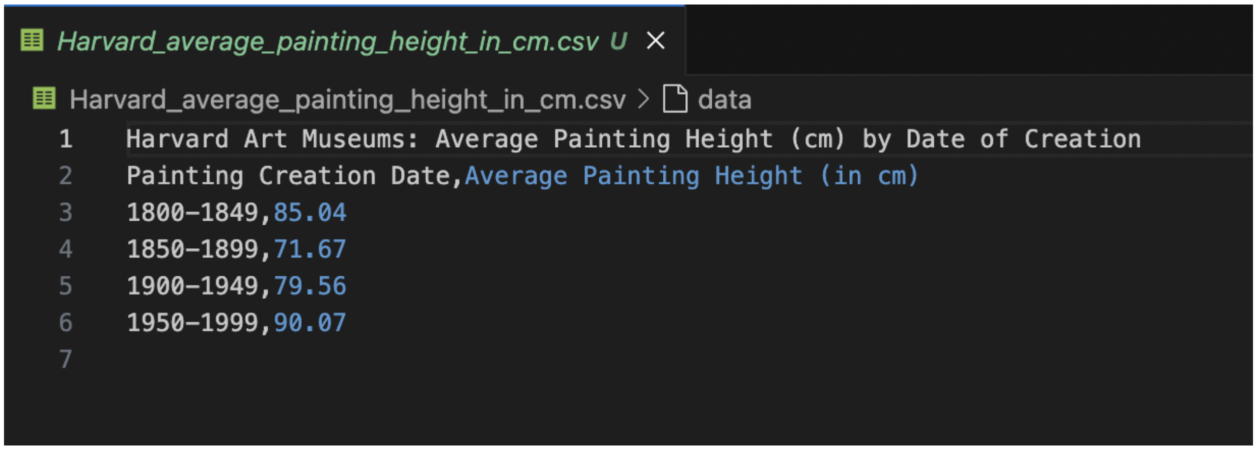
```
Cleveland Museum of Art: Painting Frequency by Department
Department,Number of Paintings
American Painting and Sculpture,32
Modern European Painting and Sculpture,31
Contemporary Art,18
Japanese Art,14
Indian and Southeast Asian Art,8
Chinese Art,2
European Painting and Sculpture,1
Korean Art,1
Medieval Art,1
```

Cleveland_department_counts.csv = CSV file with department frequency counts for paintings at the Cleveland Museum of Art. Used to make a pie chart.



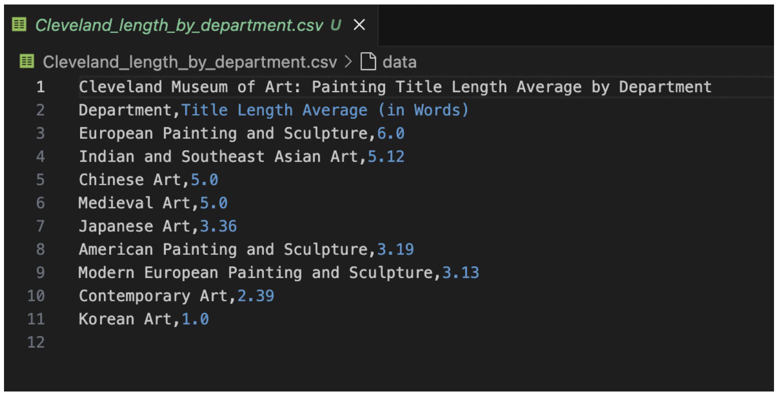
```
Met_gender_distribution_data.csv U ×
Met_gender_distribution_data.csv > data
The Metropolitan Museum of Art: Number of Paintings by Male and Female Artists by Creation Date of Artwork
Painting Creation Date,Number of Paintings by Male Artists,Number of Paintings by Female Artists
1800–1819,2,0
1820–1839,9,0
1840–1859,7,0
1860–1879,9,0
1880–1899,19,0
1900–1919,11,0
1920–1939,19,3
1940–1959,9,1
1960–1979,6,0
1980–1999,20,4
2000–2019,4,2
```

Met_gender_distribution_data.csv = CSV file with artist gender counts for paintings at The Met. Used to make a bar chart.



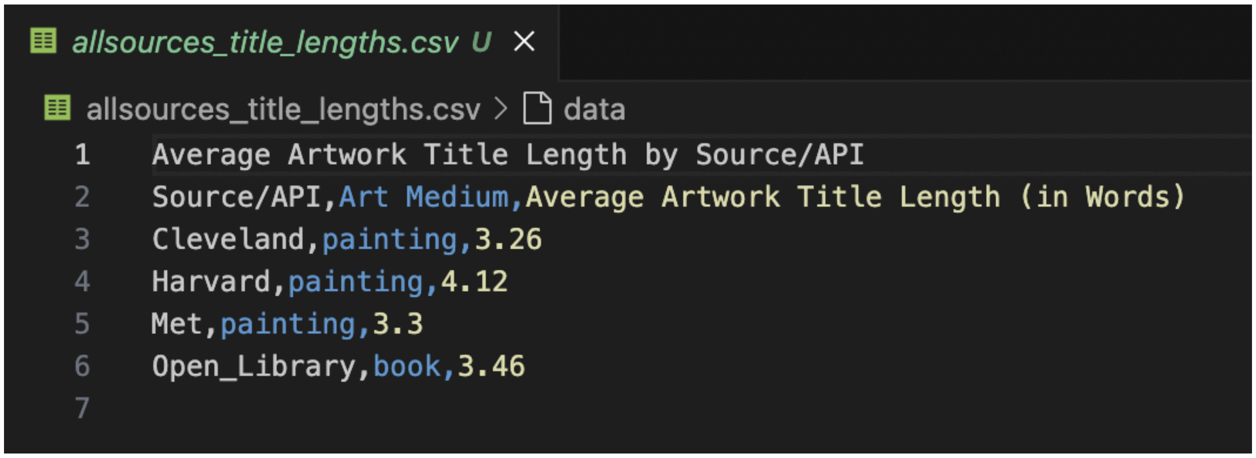
```
Harvard_average_painting_height_in_cm.csv U ×
Harvard_average_painting_height_in_cm.csv > data
Harvard Art Museums: Average Painting Height (cm) by Date of Creation
Painting Creation Date,Average Painting Height (in cm)
1800–1849,85.04
1850–1899,71.67
1900–1949,79.56
1950–1999,90.07
```

Harvard_average_painting_height_in_cm.csv = CSV file with average painting height by time period for artwork from Harvard Art Museums. Used to make a bar chart.



```
Cleveland_length_by_department.csv U ×
Cleveland_length_by_department.csv > data
Cleveland Museum of Art: Painting Title Length Average by Department
Department,Title Length Average (in Words)
European Painting and Sculpture,6.0
Indian and Southeast Asian Art,5.12
Chinese Art,5.0
Medieval Art,5.0
Japanese Art,3.36
American Painting and Sculpture,3.19
Modern European Painting and Sculpture,3.13
Contemporary Art,2.39
Korean Art,1.0
```

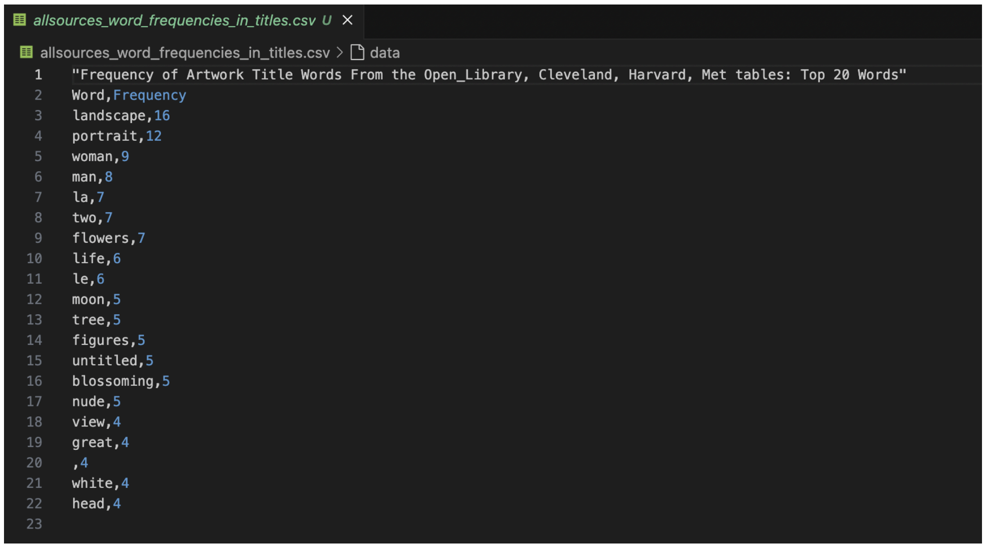
Cleveland_length_by_department.csv = CSV file with average painting title lengths by department for paintings at Cleveland Museum of Art. Used to make a bar chart.


allsources_title_lengths.csv U ×

allsources_title_lengths.csv > data

```
Average Artwork Title Length by Source/API
Source/API,Art Medium,Average Artwork Title Length (in Words)
Cleveland,painting,3.26
Harvard,painting,4.12
Met,painting,3.3
Open_Library,book,3.46

```

Allsources_title_lengths.csv = CSV file with average title lengths based on artwork from the source/API. Used to make a bar chart.

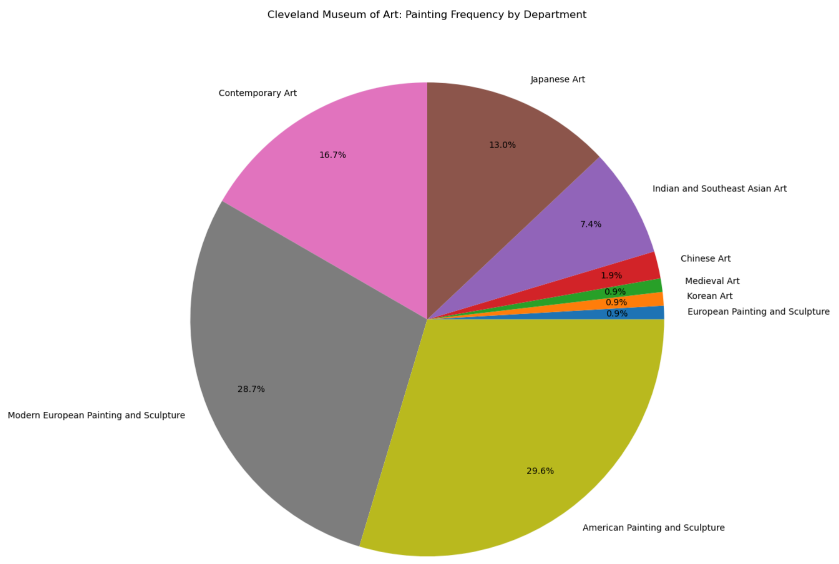

allsources_word_frequencies_in_titles.csv U ×

allsources_word_frequencies_in_titles.csv > data

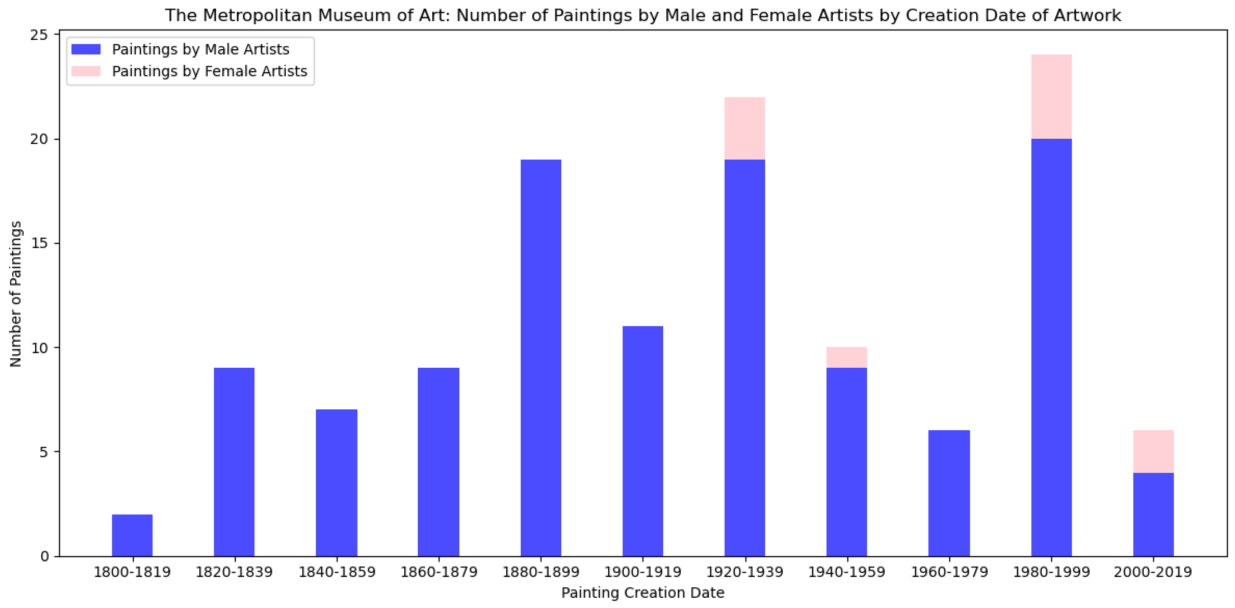
```
"Frequency of Artwork Title Words From the Open_Library, Cleveland, Harvard, Met tables: Top 20 Words"
Word,Frequency
landscape,16
portrait,12
woman,9
man,8
la,7
two,7
flowers,7
life,6
le,6
moon,5
tree,5
figures,5
untitled,5
blossoming,5
nude,5
view,4
great,4
,4
white,4
head,4

```

Allsources_word_frequencies_in_titles.csv = CSV file with word frequencies for artwork titles from the specified tables. Used to make a word cloud.

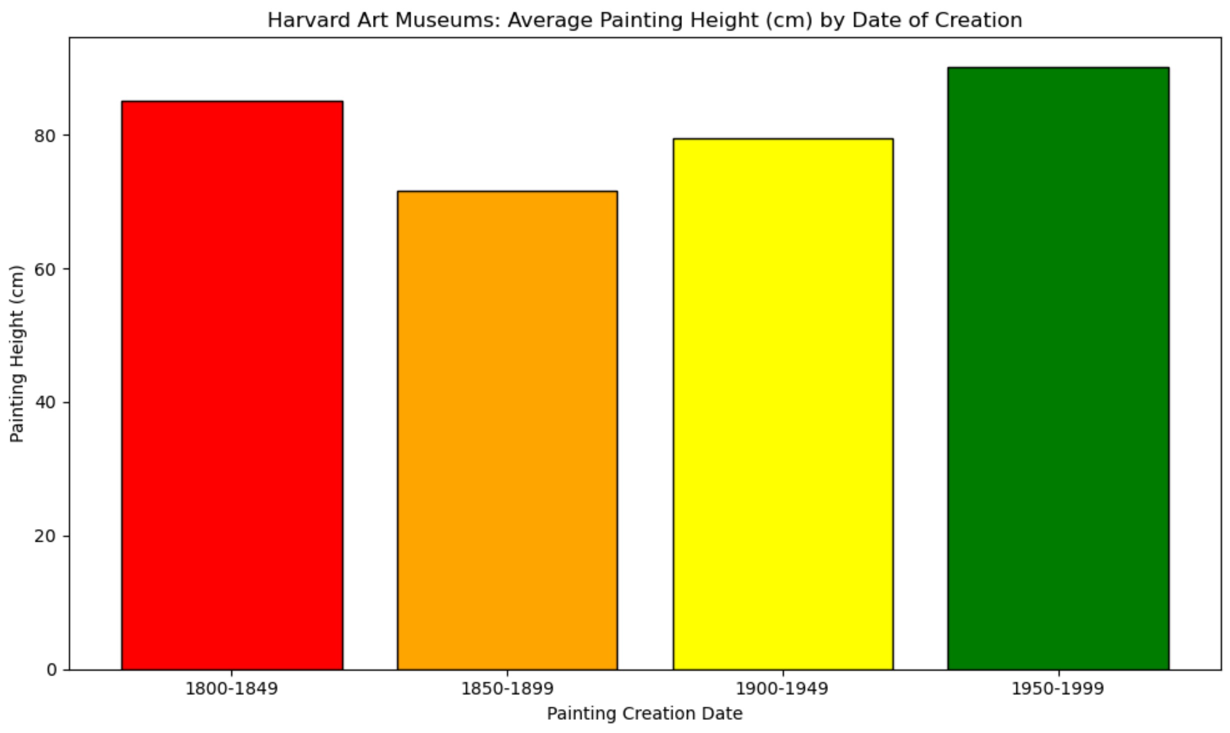
Visualizations For Calculations



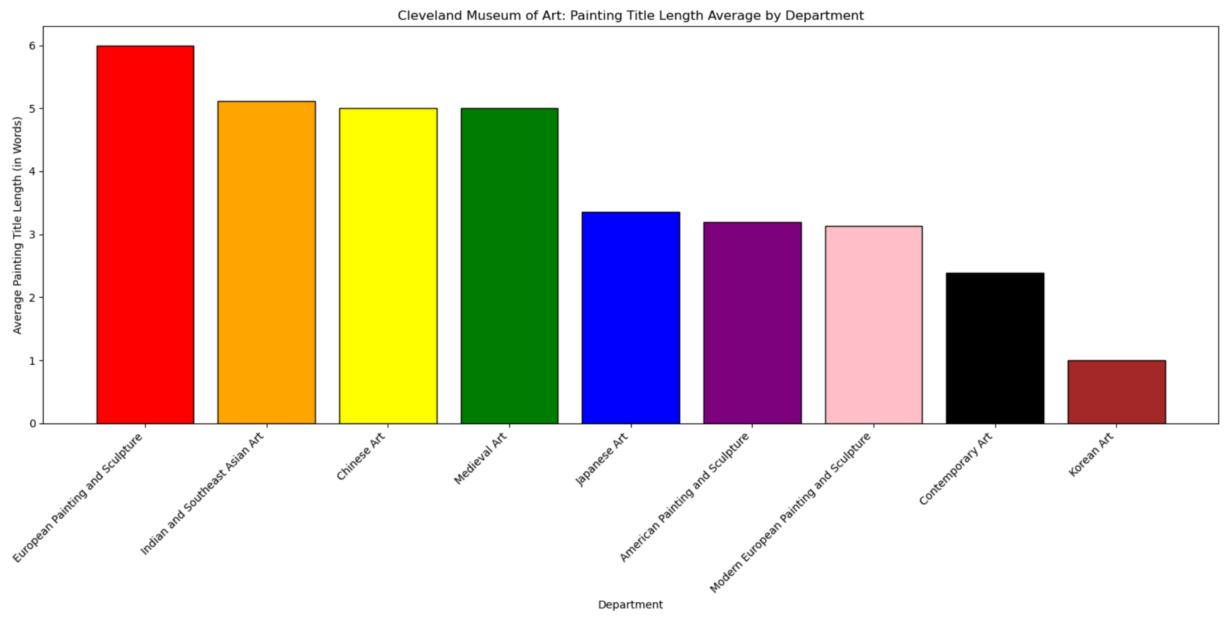
Paintings by department for Cleveland Museum of Art (Cleveland_department_frequency.png)



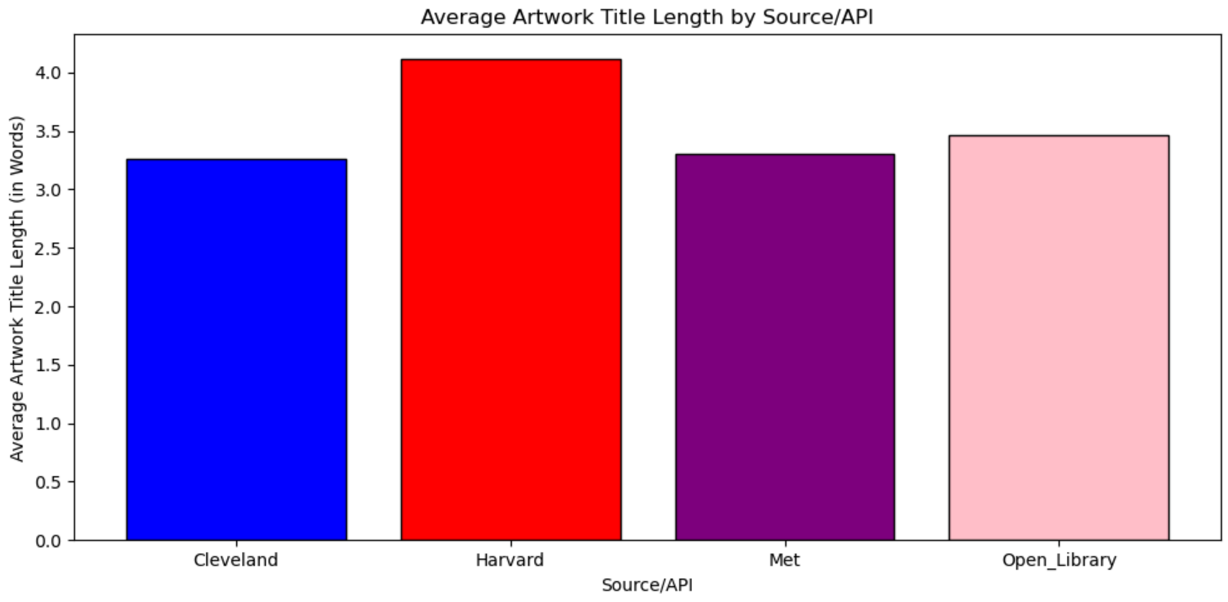
Distribution of Artist Gender for Paintings at The Metropolitan Museum of Art (Met_gender_distribution_over_time.png)



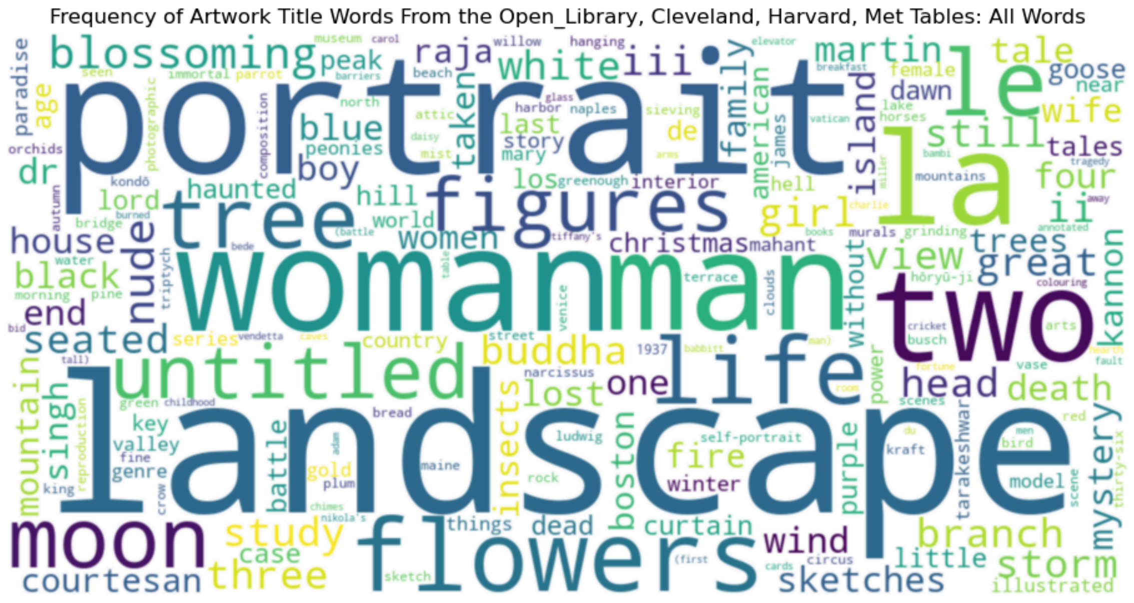
Painting height average (in cm) by time period for Harvard Art Museums (Harvard_average_painting_height_in_cm.png)



Painting title length average by department for Cleveland Museum of Art (Cleveland_length_by_dept.png)



Painting title length average (in words) by source/API for all APIs/sources (allsources_title_word_lengths_by_source.png)



Frequencies of artwork titles words from artworks in all APIs/sources (allsources_wordcloud_of_title_words.png)

Code Instructions

***If you want an empty database to start (meaning the database contains objects *only* added during the current session) first <u>delete the Museum.db file in VS Code</u>.**

1. Run the '**Cleveland.py**' file multiple times until the 'new painting count: #' printed statement reflects a number greater than or equal to 100. If the database was not deleted prior to the run, the new painting count will be 0, but 108 paintings will already be in the Cleveland table.
    a. The file has completed a single run when the "new painting count: #" is displayed in the terminal.
    b. This API is **API #1**.
2. Run the '**Met.py**' file multiple times until the 'new painting count: #' printed statement reflects a number greater than or equal to 100. If the database was not deleted prior to the run, the new painting count will be 0, but 125 paintings will already be in the Met table.
    a. The file has completed a single run when the "new painting count: #" is displayed in the terminal.
    b. This API is **API #2**.
3. Run the '**Harvard.py**' file multiple times until the 'new painting count: #' printed statement reflects a number greater than or equal to 100. If the database was not deleted prior to the run, the new painting count will be 0, but 112 paintings will already be in the Harvard table.
    a. The file has completed a single run when the "new painting count" is displayed in the terminal.
    b. This API is **API #3**.
4. Run the '**Books.py**' file multiple times until the 'new book count: #' printed statement reflects a number greater than or equal to 100. If the database was not deleted prior to the run, the new book count will be 0, but 125 books will already be in the Open_Library table.
    a. The file has completed a single run when the "new book count: #" is displayed in the terminal.
    b. Open Library ('Books') is our **bonus API** for extra credit.
5. Run the '**calculation_dept_freq.py**' file one time.
    a. This calculates the department frequencies for artwork in the Cleveland table and creates a bar chart visualization.
    b. This is **visualization #1** and uses data from the Cleveland API. This calculation uses a **JOIN** statement.
6. Run the '**calculation_gender_distribution.py**' file one time.
    a. This calculates the gender distribution for artwork in the Met table and creates a bar chart visualization.
    b. This is **visualization #2** and uses data from the Met API.
7. Run the '**calculation_average_painting_height.py**' file one time.

    a. This calculates the average painting height by time period in the Harvard table and creates a bar chart visualization.
    b. This is **visualization #3** and uses data from the Harvard API.
8. Run the ‘**calculation_length_by_dept.py**’ one time.
    a. This calculates the average painting title length by department for artwork from the Cleveland table
    b. This is a **bonus calculation/visualization** and uses data from the Cleveland API.
9. Run the ‘**calculation_title_length.py**’ file one time.
    a. This calculates the average artwork title lengths for objects within each database table (source/API).
    b. This is a **bonus calculation/visualization** and uses data from all APIs (Open Library, Cleveland, Met, and Harvard).
10. Run the ‘**calculation_wordcloud.py**’ file one time.
    a. This calculates the artwork title word frequencies across all tables (sources/APIs) in the database.
    b. This is a **bonus calculation/visualization** (we know we won’t get extra credit for more than 2 bonus visualizations) and uses data from all APIs (Open Library, Cleveland, Met, and Harvard).

Code Documentation

Functions that create the database and tables:

Cleveland.py

get_paintings(limit=25) – Fetches painting information from the API. The base URL is **https://openaccess-api.clevelandart.org/api/artworks**. There are four parameters used during the API request to limit the results returned. The ‘type’ of the artworks to be pulled is limited to only ‘Painting’ and the date of the artworks’ creations are limited to 1800 or after (‘created_after’). The ‘limit’ of the paintings to be returned is auto-set to 25, and there is an offset value that is randomly generated using the ‘skip’ parameter to ensure a unique set of paintings returned.

- Input:
    - The maximum number of paintings to pull (name = ‘limit’, default = 25, type = integer).
- Output:
    - The painting information (name = ‘paintings’, type = list of dictionaries, each dictionary corresponds to information for 1 painting).

set_up_database(database_name) – Sets up a connection to the SQLite database and initializes a cursor for database operations. Creates the database if it doesn’t already exist. The name of the database is set to ‘Museums.db’ in the main() function.

- Input:

        - The name of the SQLite database file (name = ‘database_name’, type = string).
    - Output:
        - The cursor object for executing SQL commands and the connection object to the SQLite database (names = ‘cur’, ‘conn’).

create_Cleveland_table(cur, conn) – Creates the Cleveland table in the SQLite database. The table stores painting data, including the *title* (must be unique strings), *creation year* (integer), *artist ID* (integer, shared integer key with the Cleveland Artists table), and *department ID* (integer, shared integer key with the Cleveland Departments table), along with an entry *ID key* that autoincrements from 1 (integer).

- Inputs:
    - The cursor and connection object for executing SQL commands and the connection object to commit the changes (names = ‘cur’, ‘conn’).
- There are no outputs.

create_Cleveland_Artist_table(cur, conn) – Creates the Cleveland_Artists table in the SQLite database. The table stores artist data, including the *artist's name* (string, unique values only) and an autoincremented *ID key* (integer, begins at 1).

- Inputs:
    - The cursor and connection object for executing SQL commands and the connection object to commit the changes (names = ‘cur’, ‘conn’).
- There are no outputs.

create_Cleveland_Departments_table(cur, conn) – Creates the Cleveland_Departments table in the SQLite database. The table stores departmental data, including the *department name* (string, unique values only) and an autoincremented *ID key* (integer, begins at 1).

- Inputs:
    - The cursor and connection object for executing SQL commands and the connection object to commit the changes (names = ‘cur’, ‘conn’).
- There are no outputs.

insert_paintings_into_Cleveland(paintings, cur, conn) – Gathers the specific information about each painting in the API response and inserts that information into the database tables at the specified columns. The function ensures that only unique painting titles are inserted, and duplicate titles are ignored. If any painting data is missing from an entry, it should be displayed as NULL. A print statement appears in the terminal after each new painting is added (or whether it already exists within the table), and a print statement appears after the loop of how many new paintings were added to the database. Once 25 paintings are inserted, the loop breaks and no more paintings are inserted.

- Inputs:
    - Painting information fetched from the get_paintings() function (name = ‘paintings’, type = list)

    - The cursor object for executing SQL commands and the connection object to commit the changes (names = ‘cur’, ‘conn’).
- Output:
    - The painting information for the paintings added to the database (name = ‘new_paintings’, type = list).

main() – The main function orchestrates the process of setting up the database and establishing a connection and cursor (running set_up_database(‘Museums.db’)); creating the Cleveland table (running create_Cleveland_table(cur, conn)); creating the Cleveland_Artists table (running create_Cleveland_Artist_table(cur, conn)), creating the Cleveland_Departments table (running create_Cleveland_Departments_table(cur, conn)); and fetching and inserting the painting data into the database (running insert_paintings_into_Cleveland(get_paintings(), cur, conn)). It also prints the total number of paintings in the database before and after the data insertion process.

- This function has no inputs or outputs.

Met.py

get_paintings(limit=100) **-** Fetches a list of painting IDs (as integers) from The Metropolitan Museum of Art's API, filtered only to include items classified as ‘paintings’ using a search parameter. The base URL is **https://collectionapi.metmuseum.org/public/collection/v1/search**. Returns a random sample of painting IDs up to the length of the limit.

- Input:
    - The maximum number of painting IDs to fetch (name = ‘limit,’ type = integer).
- Output:
    - A list of painting IDs (name = ‘object_ids’, type = a list of integers) if the request is successful and data exists, or None otherwise.

set_up_database(database_name) - Sets up a connection to the SQLite database and initializes a cursor for database operations. Creates the database if it doesn’t already exist. The name of the database is set to ‘Museums.db’ in the main() function.

- Input:
    - The name of the SQLite database file (name = ‘database_name’, type = string).
- Output:
    - The cursor object for executing SQL commands and the connection object to the SQLite database (names = ‘cur’, ‘conn’).

create_MET_table(cur, conn) - Creates the Met table in the SQLite database. The table stores painting data, including the *title* (string, unique values only), *creation year* (integer), and the *gender ID of the artist* (integer, **0 for men and 1 for women**), along with a general *ID key* that autoincrements (integer, begins at 1).

- Inputs:
    - The cursor and connection object for executing SQL commands and the connection object to commit the changes (names = ‘cur’, ‘conn’).
- There are no outputs.

insert_paintings_into_MET(paintings, cur, conn) - Fetches painting data for a given list of painting IDs and inserts the data into the Met table if the paintings meet specific criteria. It filters artwork based on its classification (must be a painting) and creation year (must be made between 1800–2024, inclusive). If any painting data is missing from an entry, it should be displayed as NULL. It also checks for duplicates and ensures that only unique painting titles are added. A print statement appears in the terminal after each new painting is added (or whether it already exists within the table), and a print statement appears after the loop of how many new paintings were added to the database. Once 25 paintings are inserted, the loop breaks and no more paintings are inserted.

- Inputs:
    - Painting IDs fetched from get_paintings() (name = ‘object_ids’, type = list)
    - The cursor object for executing SQL commands and the connection object to commit the changes (names = ‘cur’, ‘conn’).
- Output:
    - The object IDs for the paintings added to the database (name = ‘new_paintings’, type = list).

main() - The main function orchestrates the process of setting up the database (running set_up_database(‘Museums.db’)), creating the MET table (running create_MET_table(cur, conn)), and fetching and inserting the painting data into the database (running insert_paintings_into_MET(get_paintings(), cur, conn)). It also prints the total number of paintings in the database before and after the data insertion process.

- This function has no inputs or outputs.

Harvard.py

get_paintings() - Retrieves data from the Harvard Art Museums API. The base URL is **https://api.harvardartmuseums.org/object?apikey={key}**. It uses the API key specified within the function as the ‘key’ variable. It fetches only 25 items at a time from a randomly generated page (page number between 0-100, inclusive) within the API, and retrieves only items of the ‘Paintings’ classification.

- Input:
    - None.
- Output:
    - The painting information (name = ‘paintings’, type = list of dictionaries, each dictionary corresponds to information for 1 painting).

create_database(databasename) - Sets up a connection to the SQLite database and initializes a cursor for database operations. Creates the database if it doesn't already exist. The name of the database is set to 'Museums.db' in the main() function.

- Input:
  - The name of the SQLite database file to open (name = 'databasename', type = string).
- Output:
  - The cursor object for executing SQL commands and the connection object to the SQLite database (names = 'cur', 'conn').

create_harvard_table(cur, conn) - Creates the Harvard table in the SQLite database. The table stores painting data, including the *title* (string, unique entries only), *creation_year* (integer), and *height_cm* (float), along with an entry *id_key* that auto increments (integer, begins at 1).

- Inputs:
  - The cursor and connection object for executing SQL commands and the connection object to commit the changes (names = 'cur', 'conn').
- Output:
  - None.

insert_paintings_into_harvard(paintings, cur, conn) - Gathers the *title* (string), *creation_year* (integer), and *height_cm* (float) for each painting in the API response using regex or indexing with keys and inserts that information into the database. The function ensures that only unique painting titles are inserted, and duplicate titles are ignored. If any painting data is missing from an entry, it should be displayed as NULL. Once 25 paintings are inserted, the loop breaks and no more paintings are inserted. A print statement appears in the terminal after each new painting is added (or whether it already exists within the Harvard table), and a print statement appears after the loop of how many new paintings were added to the database.

- Inputs:
  - Painting information fetched from the get_paintings() function (name = 'paintings', type = list)
  - The cursor object for executing SQL commands and the connection object to commit the changes (names = 'cur', 'conn').
- Output:
  - The painting information for the paintings added to the database (name = 'new_paintings', type = list).

main() - The main function orchestrates the process of setting up the database: creating a connection and cursor for the database (create_database('Museums.db')), creating the Harvard table (create_Harvard_table(cur, conn)), printing the current number of paintings in the database, and fetching and inserting painting data into the database (insert_paintings_into_Harvard(get_paintings(), cur, conn)), and printing the total number of paintings in the database after calling the API.

- Input:

    - None.
- Output:
    - None.

Books.py

get_books(limit=100) - Fetches book information from the API (base URL: **https://openlibrary.org/subjects** – url altered in code to **https://openlibrary.org/subjects/fiction.json?published_in=1800-2024**). There are four parameters used during the API request to limit the results returned. The subject of the books to be pulled is limited to only 'fiction' and the publication date is limited to 1800 or after. These parameters are given within the base URL for the API request, as issues arose when including them as parameters within 'params'. Additionally, the 'limit' of the number of books to be returned is auto-set to 100, and there is an offset value that is randomly generated using the 'offset' parameter to ensure a unique set of books returned.

- Input:
    - The maximum number of books to pull (name = 'limit', default = 100, type = integer).
- Output:
    - The book information (name = 'books', type = list of dictionaries, each dictionary corresponds to information for 1 book).

set_up_database(database_name) - Sets up a connection to the SQLite database and initializes a cursor for database operations. Creates the database if it doesn't already exist. The name of the database is set to 'Museums.db' in the main() function.

- Input:
    - The name of the SQLite database file (name = 'database_name', type = string).
- Output:
    - The cursor object for executing SQL commands and the connection object to the SQLite database (names = 'cur', 'conn').

create_Open_Library_table(cur, conn) – Creates the Open_Library table in the SQLite database. The table stores book data, including the *title* (string, unique values only), *creation year* (integer), and *author ID* (integer, shared key with Open_Libary_Authors table), along with a general *ID key* that autoincrements (integer, begins at 1).

- Inputs:
    - The cursor and connection object for executing SQL commands and the connection object to commit the changes (names = cur', 'conn').
- Output:
    - None.

create_Open_Library_Authors_table(cur, conn) – Creates the Open_Library_Authors table in the SQLite database. The table stores author data, including the *author's name* (string, unique values only) and an autoincremented *ID key* (integer, begins at 1).

- Inputs:
    - The cursor and connection object for executing SQL commands and the connection object to commit the changes (names = ‘cur’, ‘conn’).
- There are no outputs.

insert_books_into_Open_Library(books, cur, conn) – Gathers the specific information about each book in the API response and inserts that information into the database. The function ensures that only unique book titles are inserted, and duplicate titles are ignored. If any book data is missing from an entry, it should be displayed as NULL. A print statement appears in the terminal after each new painting is added (or whether it already exists within the table), and a print statement appears after the loop of how many new books were added to the database. Once 25 books are inserted, the loop breaks and no more books are inserted.

- Inputs:
    - Book information fetched from the get_books() function (name = ‘books’, type = list)
    - The cursor object for executing SQL commands and the connection object to commit the changes (names = ‘cur’, ‘conn’).
- Output:
    - The book information for the books added to the database (name = ‘new_books’, type = list).

main() – The main function orchestrates the process of setting up the database (running set_up_database(‘Museums.db’)), creating the Open_Library table and Open_Library_Authors tables (running create_Open_Library_table(cur, conn) and create_Open_Library_Authors_table(cur, conn)), and fetching book information from the API and inserting it into the database (running insert_books_into_Open_Library(get_books(), cur, conn)). It also prints the total number of books in the database before and after the data insertion process.

- Input:
    - None.
- Output:
    - None.

<u>Functions that calculate the table data:</u>

calculation_dept_freq.py

get_data_from_database(database_name) – Retrieves data from the Cleveland table and Cleveland_Departments table in the specified SQLite database. It fetches the department name for every painting joined on the specified department ID.

- Input:
    - The name of the SQLite database file to connect to (name = ‘database_name’, type = string).

- Output:
    - The department name for each painting in the database (name = ‘department_data’, type = list).

get_frequency_counts(department_data) – Processes the department data to determine frequency counts for each department represented in the database.

- Input:
    - The department names for every painting in the database (name = ‘department_data’, returned from get_data_from_database(), type = list).
- Output:
    - The department names and their respective frequency counts from the data list (name = ‘data_dict’, type = dictionary, keys = department names, values = frequency count of department name in department_data list)

write_counts_to_file(data_dict, filename) – Creates a CSV file and writes the department frequencies to said file. In our main() function, the name of the CSV file is **Cleveland_department_counts.csv**. The title row is written as “Department Frequencies for Paintings From the Cleveland Museum of Art,” and the header row is written as “Department, Frequency.” Each subsequent line contains a department name and its respective frequency count.

- Inputs:
    - The department names and their respective frequency counts from the data list (name = ‘data_dict’, type = dictionary, keys = department names, values = frequency count of department name in department_data list, returned from get_frequency_counts()).
    - The name of the file to write to (name = ‘filename’, type = string)
- Output:
    - None.

visualize_counts(data_dict) – Creates a visualization of the department frequency counts as a pie chart. The chart is saved as a PNG file (Cleveland_department_frequency.png), and contains the title “Paintings by Department for Cleveland Museum of Art.” The chart contains labels set to the department names with the starting angle of the pie chart set to 0. The percentages within the pie chart are rounded to one decimal place. The completed plot is displayed to the user.

- Input:
    - The department names and their respective frequency counts from the data list (name = ‘data_dict’, type = dictionary, keys = department names, values = frequency count of department name in department_data list, returned from get_frequency_counts()).
- Output:
    - None.

main() – This creates the entire workflow, calling the functions in descending order (get_data_from_database(), get_frequency_counts(), write_counts_to_file(), and visualize_counts()) to generate a CSV file (Cleveland_department_counts.csv) and plot (Cleveland_department_frequency.png) based on the processed data from the database.

- There are no inputs or outputs.

calculation_gender_distribution.py

get_data_from_database(database_name) - Retrieves data from the Met table in the specified SQLite database. It fetches only rows where the creation_year is not NULL and returns the creation_year and gender_id for each painting.

- Input:
  - The name of the SQLite database file to connect to (name = 'database_name', type = string).
- Output:
  - The creation year and artist gender (0 for male, 1 for female) for each painting in the database (name = 'gender_distribution', type = list of tuples – each tuple is for a specific painting in the format [(creation year, gender), …].

process_data(data, interval=10) - Processes the painting data into a gender distribution based on specified year intervals. It calculates the count of paintings by male and female artists for each year interval.

- Inputs:
  - The size of each year interval (name = 'interval', type = integer, default = 10)
  - The year and gender data retrieved from the database - the output of the above function (name = 'gender_distribution', type = list of tuples).
- Output:
  - A dictionary containing gender distribution information (name = 'gender_distribution', type = dictionary, keys = year intervals, values = dictionary – keys = 'male' or 'female' (type = string) and values = counts of paintings for male or female artists (type = integer).
  - The size of each year interval (name = 'interval', type = integer, default = 10)

write_data_to_csv_file(gender_distribution, interval) – Writes the gender distribution data into a CSV file, '**gender_distribution_data.csv.**' There is a file title row written as "The Metropolitan Museum of Art: Number of Paintings by Male and Female Artists by Creation Date of Artwork" as well as a header row written as "Painting Creation Date, Number of Paintings by Male Artists, Number of Paintings by Female Artists". After that, each row contains the year range, the count of paintings by male artists, and the count of paintings by female artists.

- Inputs:

    - The processed data showing the gender distribution across year intervals (name = ‘gender_distribution’, output from process_data, type = dictionary)
    - The year interval (name = ‘interval’, input/output from process_data, type = integer).
- Output:
    - There are no outputs, the function just writes the data into a CSV file.

plot_gender_distribution(gender_distribution, interval) - Generates a bar chart visualizing the gender distribution of painters over time. Painting count by male vs. female painters are displayed bottom-top for each interval. The plot is saved as a PNG file, ‘gender_distribution_over_time.png’, and displayed to the user.

- Inputs:
    - The processed data showing the gender distribution across year intervals (name = ‘gender_distribution’, type - dictionary, output from process_data)
    - Year interval (name = ‘interval’, input and output from process_data, type = integer).
- There are no outputs, a plot is just created and saved as a PNG file.

main() - This creates the entire workflow, calling the functions in descending order (get_data_from_database(), process_data(), write_data_to_csv_file() and plot_gender_distribution()) to generate a CSV file (gender_distribution_data.csv) and plot (gender_distribution_over_time.png) based on the processed data from the database.

- There are no inputs or outputs.

calculation_average_painting_height.py

get_data(database) - Establishes a path to the specified database, creates a cursor and a connection to the database, and retrieves data from the Harvard table. It fetches the *creation_year*, and *height_cm* for every painting where the *creation_year* and *height_cm* is not NULL. It stores this retrieved painting data in a list of tuples called ‘data.’

- Input:
    - The name of the SQLite database file to connect to (name = ‘database’, type = string).
- Output:
    - The painting creation year and height for each pulled painting (name = ‘data’, type = list of tuples with each painting’s *creation_year* and *height_cm*).

calculate_average_height(data) - Creates a dictionary with time periods (half-centuries since 1800 – e.g. ‘1800-1849’, ‘1850-1899’) as keys and (ultimately) the average of all painting heights from a time period as its values. It iterates through the list of tuples called ‘data’ that is passed in which holds the *creation_year* and *height_cm* for each painting. A nested dictionary in the format {‘heights’: [painting 1 height, painting 2 height, …]} is stored as the value before being converted into a singular float value (the average painting height).

- Input:
  - The painting creation year and height for each pulled painting (name = ‘data’, type = list of tuples with each painting’s *creation_year* and *height_cm*).
- Output:
  - The average painting heights organized by time period (name = ‘height_dist’, type = dictionary with time periods as keys (strings) and average painting height for that time period as values (floats).

write_heights_to_csv_file(filename, dist) - Creates a CSV file and writes the average painting heights to said file. In the main() function, the name of the CSV file is set as ‘**Harvard_average_painting_height_in_cm.csv**’. The title row is written as “Harvard Art Museums: Average Painting Height (cm) by Date of Creation,” and the header row is written as “Painting Creation Date, Average Painting Height (in cm).” Each subsequent line contains a half-century time period since 1800 (e.g. ‘1800-1849’) in chronological order and its respective average painting height in decimal form.

- Inputs:
  - The name of the csv file being written to (name = ‘filename’, type = string)
  - A dictionary of the time periods and their respective average painting heights (name = ‘dist’, type = dictionary, keys = time periods as strings, values = average painting height for each time period as a float value rounded to two decimal points).
- Output:
  - None.

plot_average_height(height_dist) - Creates a visualization of the average painting heights by half-centuries since 1800 as a bar chart. The chart is saved as a PNG file named “Harvard_average_painting_height_in_cm.png,” and contains the title “Harvard Art Museums: Average Painting Height (cm) by Date of Creation.” Labels are set to the painting creation dates on the x-axis, and painting heights (in cm) on the y-axis. The completed plot is displayed to the user.

- Input:
  - The time periods and their respective average painting heights (name = ‘height_dist’, type = dictionary, keys = time periods as strings, values = average painting height for each time period as a float value rounded to two decimal points).
- Output:
  - None.

main() - This creates the entire workflow, calling get_data() with the name of the database, creating a dictionary of painting data with calculate_average_height(), writing the data from that dictionary into a csv file (Harvard_average_painting_height_in_cm.csv) using write_heights_to_csv_file(), and plotting the data from the dictionary into a bar plot (Harvard_average_painting_height_in_cm.png) with plot_average_height().

- Input:
  - None.
- Output:
  - None.

calculation_length_by_dept.py

get_data_from_database(database_name) – Retrieves data from the Cleveland table and Cleveland_Departments table in the specified SQLite database. It fetches the department name for every painting joined on the specified department ID.

- Input:
  - The name of the SQLite database file to connect to (name = ‘database_name’, type = string).
- Output:
  - The department name and painting title for each painting in the database (name = ‘titles_and_departments’, type = list of tuples in the format [(title, department), …]).

get_painting_title _length_averages(titles_and_departments) – Processes the department data to determine the average title length for paintings from each department represented in the database.

- Input:
  - The department name and painting title for each painting in the database (name = ‘titles_and_departments’, type = list of tuples in the format [(title, department), …], returned from get_data_from_database()).
- Output:
  - A dictionary of painting title length averages by department (name = ‘averages_dict’, key = department name as string, value = average length of painting title as float, type = dictionary).

write_calculations_to_csv(averages_dict, filename) – Creates a CSV file and writes the painting title length averages to said file. In the main() function, the name of the CSV file is set as ‘Cleveland_length_by_department.csv’. The title row is written as “Painting Title Length Average (in Words) by Department for Artwork From the Cleveland Museum of Art,” and the header row is written as “Department, Title Length Average (in Words).” Each subsequent line contains a department name and its respective title length average.

- Input:
  - The department names and their respective painting title length averages (name = ‘averages_dict’, type = dictionary, keys = department names, values = average painting title length for each department as a float value, returned from get_painting_title_length_averages()).
  - The name of the file to write to (name = ‘filename’, type = string)

- Output:
    - None.

visualize_averages_by_department(averages_dict) – Creates a visualization of the average painting title lengths by department as a bar chart. The chart is saved as a PNG file (Cleveland_length_by_dept.png), and contains the title “Average Painting Title Length (in Words) by Department for Cleveland Museum of Art.” Labels are set to the department names on the x-axis, and painting title length averages (in words) on the y-axis. The completed plot is displayed to the user.

- Input:
    - The department names and their respective painting title length averages (name = ‘averages_dict’, type = dictionary, keys = department names, values = average painting title length for each department as a float value, returned from get_painting_title_length_averages()).
- Output:
    - None.

main() – This creates the entire workflow, calling the functions in descending order (get_data_from_database(), get_painting_title _length_averages(), write_calculations_to_csv(), and visualize_averages_by_department()) to generate a CSV file (Cleveland_length_by_department.csv) and plot (Cleveland_length_by_dept.png) based on the processed data from the database.

- There are no inputs or outputs.

calculation_title_length.py

open_database(db_name) – Opens a database based on the name specified in the parameter ‘db_name’ and establishes a cursor and connection to the database.

- Input:
    - The database name to open (name = ‘db_name’, type = string).
- Output:
    - A cursor and connection to the database (names = ‘cur’, ‘conn’).

caluclate_title_length_averages_by_source(cur, conn, table_names) – Calculates the average title length for data based on each source specified. Opens each table that corresponds to a source, fetches all artwork title names for each source, calculates the average artwork title length for each source, and adds each average to a dictionary as a value under the corresponding source key.

- Inputs:
    - The cursor and connection to the database (names = ‘curr’, ‘conn’)
    - The names of the database tables to get painting titles from (name = ‘table_names’, type = list).
- Output:

    - A dictionary containing the artwork title length averages for each indicated source (name = ‘source_averages_dict, type = dictionary, keys = table ‘source’ names as strings, values = the average artwork title length for the table entries as floats).

write_lengths_to_csv_file(source_averages_dict, filename) – Creates a CSV file and writes the artwork title length averages to said file. In the main() function, the name of the CSV file is set as “allsources_title_lengths.csv”. The title row is written as “Average Artwork Title Length (in Words) by Source/API,” and the header row is written as “Source/API, Art Medium, Average Artwork Title Length (in Words).” Each subsequent line contains a source name and its respective art medium and title length average.

- Inputs:
    - The name of the file to create (name = ‘filename’, type = string)
    - A dictionary containing the artwork title length averages for each indicated source (name = ‘source_averages_dict’, type = dictionary, keys = table ‘source’ names as strings, values = the average artwork title length for the table entries as floats).
- Output:
    - None.

visualize_title_lengths(source_averages_dict) – Creates a visualization of the average painting title lengths by source as a bar chart. The chart is saved as a PNG file (allsources_title_word_lengths_by_source.png), and contains the title “Average Artwork Title Length (in Words) by Source/API.” Labels are set to the source/API names on the x-axis, and painting title length averages (in words) on the y-axis. The completed plot is displayed to the user.

- Input:
    - A dictionary containing the artwork title length averages for each indicated source (name = ‘source_averages_dict, type = dictionary, keys = table ‘source’ names as strings, values = the average artwork title length for the table entries as floats).
- Output:
    - None.

main() – This creates the entire workflow, calling the functions in descending order (open_database(), caluclate_title_length_averages_by_source(), write_lengths_to_csv_file(), visualize_title_lengths()) to generate a CSV file (allsources_title_lengths.csv) and plot (allsources_title_word_lengths_by_source.png) based on the processed data from the database.

- There are no inputs or outputs.

calculation_wordcloud.py

get_words_from_database(table_names_lst, database_name) – Gets all artwork title words from the specified tables in the database.

- Inputs:
    - Names of the database tables to pull artwork titles from (name = 'table_names_list', type = list of strings).
    - Name of the database that the tables are within (name = 'database_name', type = string)
- Output:
    - A list of words from all specified in the database (name = 'words', type = list).

calculate_word_frequency(words) – Calculates word frequency for each word in the artwork titles collected.

- Input:
    - List of words from artwork titles pulled from the database tables, duplicates are included (name = 'words', types = list).
- Output:
    - A dictionary of words and their respective frequencies from artwork titles in the specified database tables (name = 'word_dict', type = dictionary, keys = the artwork title words as strings, values = the word frequency counts as integers).

write_to_csv_file(filename, word_dict, table_names_lst) – Creates a CSV file and writes the artwork title word frequencies to said file. In the main() function, the name of the CSV file is set as "allsources_word_frequencies_in_titles.csv". The title row is written as "Word frequencies for artwork titles from the [table names] table(s)." Each subsequent line contains a word from an artwork title and its respective frequency. Only the 20 most frequent words are included in the file.

- Inputs:
    - The name of the file to create (name = 'filename', type = string)
    - A dictionary of words and their respective frequencies from artwork titles in the specified database tables (name = 'word_dict', type = dictionary, keys = the artwork title words as strings, values = the word frequency counts as integers)
    - The tables to collect artwork title names from (name = 'table_names_lst', type = list of stings).
- Output:
    - The table names to collect artwork title words from (name = 'table_names_string', type = string, desc: joined list of table names)
    - The word 'table' or 'tables' based on the plurality of the specified table name(s) (name = 'table', type = string).

visualize_word_cloud(word_dict, table_names_string, table) – Creates a visualization of the artwork title word frequencies as a word cloud. The chart is saved as a PNG file (allsources_wordcloud_of_title_words.png), and contains the title “Frequencies of Artwork Title Words in the [table names] table(s).” The completed plot is displayed to the user.

- Inputs:
    - A dictionary of words and their respective frequencies from artwork titles in the specified database tables (name = ‘word_dict’, type = dictionary, keys = the artwork title words as strings, values = the word frequency counts as integers)
    - The tables to collect artwork title names from (name = ‘table_names_lst’, type = list of stings)
    - The word ‘table’ or ‘tables’ based on the plurality of the specified table name(s) (name = ‘table’, type = string).
- Output:
    - None.

main() – This creates the entire workflow, calling the functions in descending order (get_words_from_database(), calculate_word_frequency(), write_to_csv_file(), visualize_word_cloud()) to generate a CSV file (allsources_word_frequencies_in_titles.csv) and plot (allsources_wordcloud_of_title_words.png) based on the processed data from the database.

- There are no inputs or outputs.

Resources Used

Date	**Issue Description**	**Location of Resource**	**Result**
11/11/24	Accessed Cleveland API documentation to begin work	Cleveland Museum of Art API Documentation	Successfully accessed data for paintings from Cleveland API
11/11/24	Accessed Met API documentation to begin work	Met Museum of Art API Documentation and GitHub - metmuseum/openaccess: The Metropolitan Museum of Art's Open Access Initiative	Successfully accessed data for paintings from Met API
11/18/24	Could not get unique	Piazza (Piazza Post	Successfully used the

	data from the Cleveland API when making requests.	250 - SI 206)	‘skip’ parameter to get unique data from the API on every call; used this knowledge to get unique data from the Harvard API as well
11/20/24	Could not get primary integer key to autoincrement correctly (had gaps when duplicate painting title was ignored when inserting data to database)	Office Hours with Vivian Le	Successfully used a primary integer key that worked and autoincremented without gaps for ignored paintings
11/21/24	Accessed Harvard API documentation to begin work	Harvard Art Museums - Github API Documentation	Successfully accessed data for paintings from Harvard API
11/25/24	Needed instruction on how to make a bar chart in matplotlib	matplotlib.pyplot.bar	Successfully created a bar chart to visualize data from Cleveland, Harvard, Met, and Open_Library titles (specifically ‘allsources_title_word_lengths_by_source.png’)
12/02/24	Wanted to explore if the distribution of artist gender could/would be better displayed as a histogram, since the x-axis uses years (numerical data) as opposed to categorical data	Matplotlib Histograms, how to plot two histograms with stacked bars, without stacking all the bars together? - Stack Overflow, chatGPT prompt “I’d like to make a histogram using matplotlib. I’m charting painting artist genders (either male or female) and the year of the painting’s creation. I’d	Determined that a bar chart is better used in this case, since we’re using discrete time blocks (1920-1939, e.g.).

		like the x-axis to have the years as date ranges, and the y-axis to have the counts of male and female artists. I'd like the female counts to be stacked on top of the male counts. My issue is that I'm not sure whether this should be a bar chart or a histogram. What do you think?"	
12/02/24	Wanted to find out how to make a pie chart with a legend and wedge	Week 13 discussion code, matplotlib.pyplot.pie	Successfully created a pie chart to display calculations for data from the Cleveland table (specifically, 'Cleveland_department_frequency.png')
12/02/24	Wanted to find out how to make a word cloud based on word frequencies in dictionary format as opposed to word frequencies in string format	Generating Word Cloud in Python - GeeksforGeeks, Using frequency — wordcloud 1.8.1 documentation	Successfully created a word cloud using a word frequency dictionary as opposed to a string
12/02/24	Needed to remove the most popular words from the word cloud (like 'he', 'i', 'it', etc.)	ChatGPT prompt "How do I exclude the most common words from a word cloud? Is there a set library for it?"	Successfully used 'stopwords' to exclude most common words from the word cloud
12/2/24	Was working on the Harvard.py file and got curious about using the pop() method to get rid of dictionary key-value pairs after creating empty value placeholders.	https://www.geeksforgeeks.org/python-list-pop-method/	Didn't end up using the pop() method and took a different approach using the creation of multiple nested dictionaries; achieved the desired result.

12/4/24	When writing the plot_average_height( ) function of the calculation_average_painting_height.py file, I looked up matplotlib documentation when creating my bar plot.	https://matplotlib.org/stable/users/index.	Successfully created a bar plot using data from a dictionary.
12/10/24	When figuring out how to order the x-axis variables chronologically, I looked up how to use sorted() and sort() functions again	https://www.w3schools.com/python/ref_func_sorted.asp	Successfully found a way to use sorted() to reach the desired result.
12/10/24	Could not get bars to fill to the width of the plot (they were super skinny!)	Auto bar width in matplotlib : r/pythontips	Successfully changed the type of the labels in the bar chart to ‘strings’ so that the bars would be set to auto-width.