

MET Museum of Art API, Countries Web Scrapping, and Visualizations

https://github.com/cmoy11/si206_finalProject

Note for grader: Since our in-class presentation, we modified our table by **only** returning the objectID, cityID, artworkYearID, imageURL, color1, color2, and color3. Additionally, to improve our program's efficiency, csv files and visualizations are only created once the database addition is complete.

Original Goals

We originally planned to use the MET API which had information about specific art pieces and also use a website called the "[Watson Library Digital Collections](#)", which provided a more expansive description of the art pieces. We wanted to find the number of paintings per dynasty, time period by using these two resources.

Goals Achieved

APIs and Websites Used: [MET API](#), [World Cities Population 2022](#)

During our project, we changed our scope to accommodate what was available through the MET API. The MET API did not have dynasty information for most of the artworks. However, we found that most of the art pieces instead had an image URL attached to them. Because of this, and our discovery of the K-Means image clustering module, we shifted our goal to find the most common colors used in the art based on city and time period.

We scraped the most common cities from the website, and searched MET artwork from corresponding cities from the API. From that we used the MET API to return artwork id, artwork location, artwork image. We then used image segmentation using K-Means clustering to recognize the most common colors in the image, and finally averaged artwork RGB values to find the average color of the city and time-period. We ended up creating two visualizations that serve as an interesting perspective on analyzing MET Museum artwork and how art differs per country and how it may shift over time.

Problems Faced

Problem 1: The first problem we faced was that the bulk of the MET API lacked the dynasty and images for the individual art pieces.

- Solution: We overcame this by changing our project goal to find specific cities' artworks (returned by the World Cities Population website) instead of dynasties. We also parsed through the data to only return artworks that had image links.

Problem 2: The color module only could return HEX codes after doing the K-Means clustering. This did not allow us to average the colors, since the HEX codes contain both letters and numbers, not being all a numeric value.

- Solution: We converted the 3 HEX colors that the module returned into RGB values. This enabled us to find the average color by dividing the R, G, B numbers.

Problem 3: We aimed to use a Color API to perform image color recognition, but we could not find an open API.

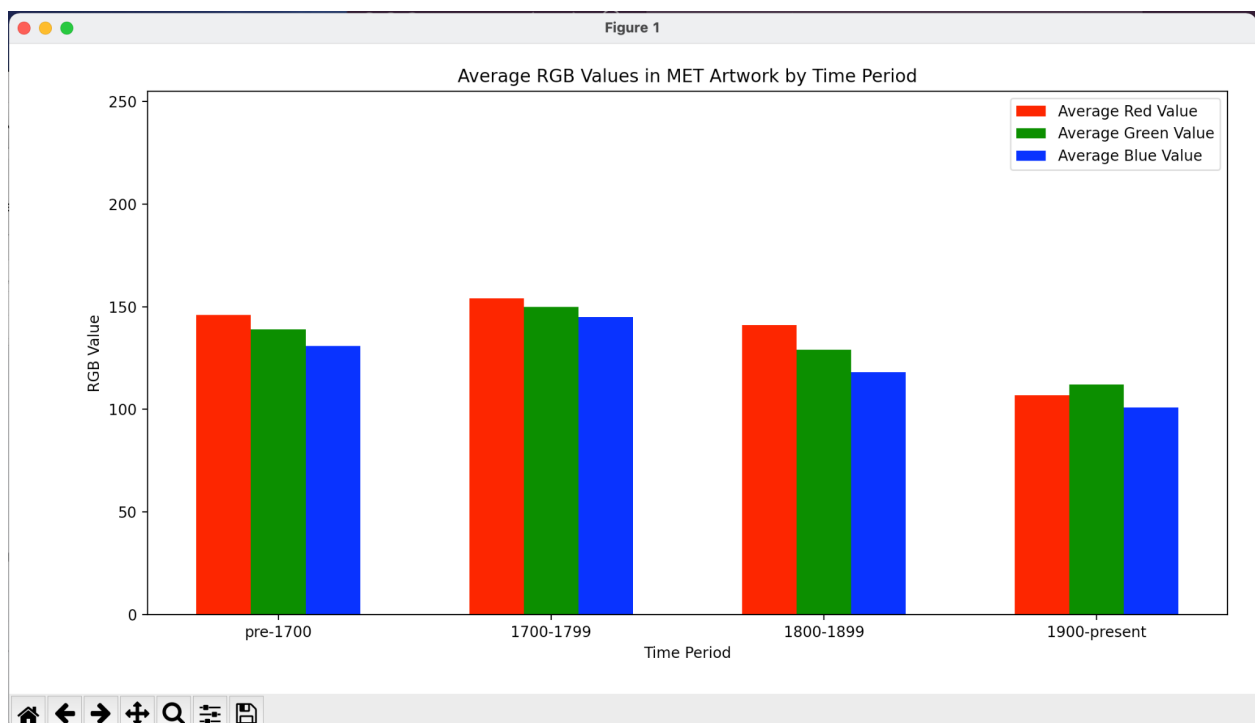
- Used a color module instead that did color clustering through K-Means segmentation. The module was customizable and we could specify the number of colors that were the most prominent per image.

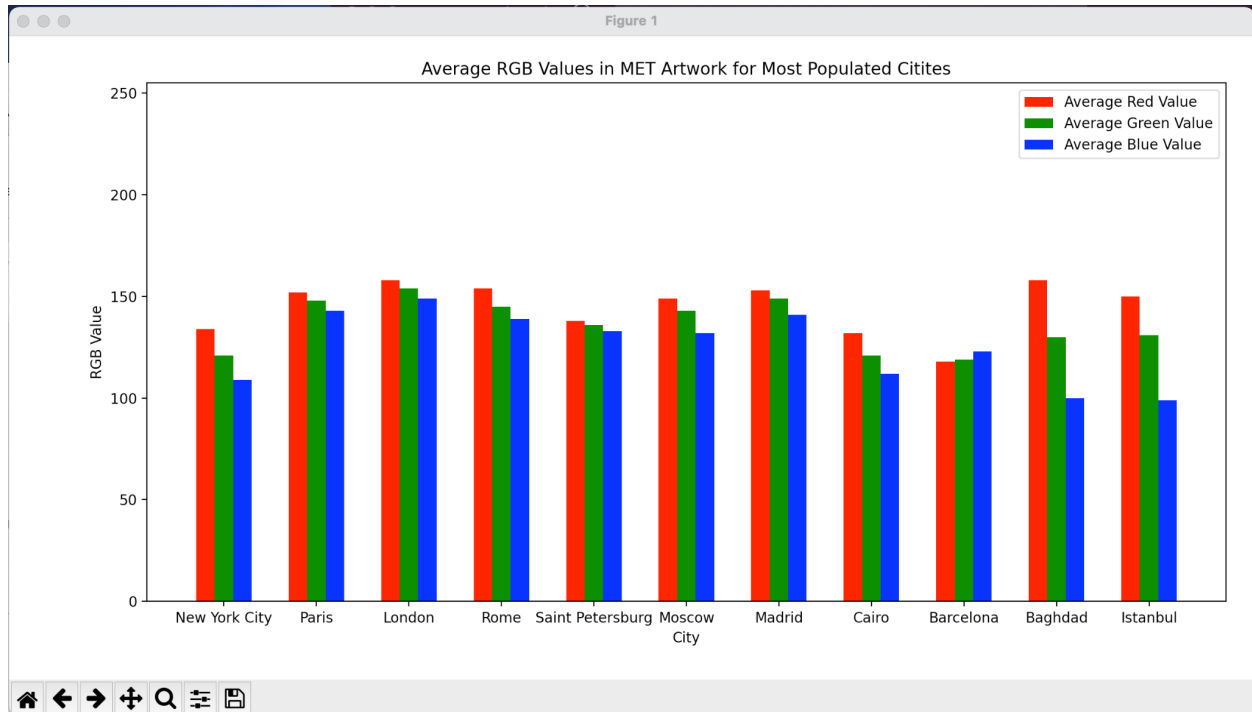
Calculations

1	city	number of pieces	average red value	average green value	average blue value
2	New York City	25	134	121	109
3	Paris	25	152	148	142
4	London	24	157	153	148
5	Rome	17	156	148	143
6	Saint Petersburg	9	137	136	132
7	Moscow	7	144	139	128
8	Madrid	4	153	148	141
9	Cairo	2	131	121	112
10	Barcelona	1	118	118	122
11	Baghdad	1	179	157	131
12	Istanbul	1	150	131	99

1	time period	number of pieces	average red value	average green value	average blue value
2	pre-1700	21	146	139	132
3	1700-1799	57	155	150	146
4	1800-1899	35	140	129	118
5	1900-present	3	106	111	100

Visualizations





Instructions for Running the Code

Our program is set up to be run without any additional work. The first six times the code is run, the program will populate the database in accordance with the table below. Never does the number of rows added to the Cities or Artwork tables exceed 25. The seventh (or any future) time a user executes our code, the program will print that the database addition is complete. The calculation csv files will then be created and populated and the matplotlib visualizations will appear.

Note: occasionally we ran into an error message: database locked. If this occurs run the following terminal commands (additional information available in resources used):

- 1) Fuser met.db
- 2) kill -9 [number returned from step 1]

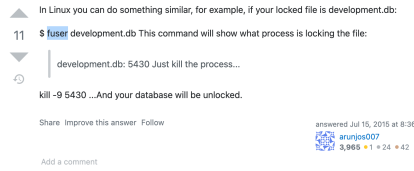

Number of City Rows	Number of Artwork Rows
0	0
25	10
34	35
44	59
51	84
76	99
100	116

Documentation

Function	Documentation
create_database()	Accepts a database name and first creates a cursor object and establishes a connection that will be used throughout the program. The function then creates our three tables, Artwork, Cities, and Years, if they do not already exist and adds the year ranges, which were predetermined, to Years. create_database() then returns the cur and conn to be used later in the program.
get_cities()	Uses BeautifulSoup on World Cities Population 2022 to create a dictionary, where the key is the ranking and the value is the city. This dictionary is then converted into a list of tuples.
get_API()	Accepts a list of cities (tuples with an index and city) returned from get_cities() as well as the cur and conn. First the function gets the number of existing rows in Artwork and assigns it to original_length, this will be used to ensure the maximum number of added rows/run does not exceed 25. From there, we pass each city into the MET API search endpoint and collect a list of object IDs (pieces of art) housed within the MET from the corresponding city. We then use the MET API again to obtain information on each of the object IDs, such as year created and image URL, limiting to at most 25. Once this is complete, the artwork is added or ignored into the Artwork table, converting the year to a corresponding yearID. Finally, the city is added or ignored to the Cities table. This is done last to ensure that if the 25 limit is reached before a city's artwork is exhausted the next run will scrape the remaining pieces of art. Nothing is returned from this function.
get_artwork_data()	Accepts the cur and conn objects and selects a majority of the columns from the Artwork, Cities, and Years, using a double JOIN statement. The function then returns this data as a list of tuples to be used in the add_colors() function.
download_image()	Accepts an image's object ID and its image URL. If the file does not already exist, the program downloads the image and adds it to the image folder.
hex_to_rgb()	Accepts a list of HEX values initial_hex_list that is returned by the color module. Iterates through the list and converts the HEX values into RGB values. Returns a new list called final_rgbs. Nothing is returned.
add_colors()	add_colors() accepts the cur and conn objects and executes a majority of the color based functions. The function begins by collecting the information from get_artwork_data and looping through the rows, downloading any images that do not currently exist. The function then uses the K-means module to find the three most prevalent colors, if they are not already in the database, and adds them to the Artwork table. The function returns a list of artworks with their corresponding

	RGB values.
make_dictionary()	Accepts object_id, city, timePeriod, colors from input_color. Takes the three rgb values within colors and averages the r, g, and b values for the respective city and timePeriod into avg_red, avg_green, and avg_blue for each art piece. Creates two new dictionaries: cities_dict and time_period_dict. Accumulates the cities and time periods and makes a nested dictionary, where the key is cities and time periods and the value is the avg_red, avg_green, and avg_blue. It does this by going through the dictionaries and finding the average color for the cities and time periods for each by averaging the respective average_rgb art piece values by the number of cities and time periods there are in the dictionary. The function returns cities_dict and time_period_dict where the average RGB color is added.
write_csv()	write_csv() accepts the city and time calculation dictionaries created in make_dictionary(). The function then writes two csv files with the corresponding data (found above) with predetermined file names. Nothing is returned.
visualize_data()	Selects the cities and average RGB values from met_city.csv and returns a multi bar chart (bars being the average RGB values for each city). Selects the time periods and average RGB values from met_time.csv and returns a multi bar chart (bars being the average RGB values for each time period).
main()	The main function first selects the length of the Cities table and assigns it to length, this will be used later to ensure no more than 25 rows are added to the Cities table at a time. The function then runs the get_cities(), get_API(), and add_colors() functions if the database is not complete. If the database is complete, the function ensures that all colors are added and then runs the make_dictionary(), write_csv() and visualize_data() functions.

Resources Used

Date	Issue Description	Location of Resource	Result
4/18	Database is locked error in code	https://stackoverflow.com/questions/2740806/python-sqlite-database-is-locked .	<p>The terminal commands found here did solve the issue, although we were never able to figure out why the database was locked in the first place.</p>  <p>answered Jul 15, 2015 at 8:36  arunp007 3,985 ●1 ●24 ●42 Add a comment</p>

4/17	Difficulties creating the side-by-side bar plots	https://pythonguides.com/matplotlib-multiple-bar-chart/ .	This resource was a good starting point for our visualizations. From there, we were able to build in the additional bar plot and add essential components such as titles and axis labels.
4/17	Needed to sort the date ranges in a custom order	https://www.i2tutorials.com/how-to-sort-a-list-in-custom-order-in-python/ .	This was a tricky issue that we needed to resolve to make logical visualizations, but this resource fixed them.
4/15	Needed to convert the hex values returned from the K-means module into decimal values	https://www.pythonpool.com/python-hexadecimal-to-decimal/ .	This resource was super helpful. We had no idea the built in int() function was capable of doing base conversions.
4/14	Needed to be able to download the images from image URLs to use the K-means module.	https://towardsdatascience.com/how-to-download-an-image-using-python-38a75cfa21c .	This code worked well and was useful in downloading images from our image URLs. We did need to use some problem solving, however, as we wanted all images in a separate image folder.
4/14	Could not find a color sampler API	https://github.com/curiousily/Machine-Learning-from-Scratch/blob/master/4_k_means.ipynb .	This module worked super well for our project. We were able to use the K-means to analyze our photos and hit minimal future issues.
4/13	Needed to check if the image files already existed so image files were not downloaded multiple times	https://www.pythontutorial.net/python-basics/python-check-if-file-exists/ .	This resource was a good starting point, but we were actually able to use a simpler method since our file names corresponded with object IDs.
4/13	Needed to use a double JOIN SQL statement	https://www.sqlshack.com/sql-multiple-joins-for-beginners-with-examples/ .	Worked super well. Easy process, just had never done before.