DATA SCRAPING FROM SOCIAL MEDIA PROFILES

Introduction:

This documentation provides an overview of the project on scraping Twitter data using python as a core language. The objective of this project is to extract and analyze data from Twitter to gain insights, understand trends. The requests and pandas are Python libraries that allow easy access to Twitter's public APIs and provide efficient methods for scraping tweets and user information.

Installation:

1. Install python in your operating system, it can be Windows/Linux.
2. Install Visual Studio Code IDE to create your project in it.
3. Install pandas and requests libraries by creating a virtual environment specifically for this project. As i have created a virtual environment, i will recommend you to use it too.
4. Get API key from the RapidAPI platform. (Steps to get the API key will be mentioned below)

How to get API from RapidAPI:

Sign Up on RapidAPI

- If you don't have an account on RapidAPI, you need to sign up. Visit the RapidAPI website (https://rapidapi.com/) and create an account.

Find Twitter API on RapidAPI

- Search for the Twitter API on the RapidAPI marketplace. You can do this by browsing through the available APIs or using the search functionality.

Select a Twitter API

- Choose a Twitter API that suits your requirements. There might be multiple options available based on the functionality you are looking for.

Subscribe to the API

- Once you've found the Twitter API you want to use, subscribe to it. This usually involves agreeing to the terms of use and getting an API key.

Get API Key

- After subscribing, you'll typically receive an API key. This key is essential for authenticating your requests to the Twitter API.

Understand API Documentation

- Read the API documentation provided by RapidAPI and the specific Twitter API you selected. Understand the available endpoints, parameters, and response formats.

Integrate API Key into Your Code

- In your code, include the API key in the request headers. This is often done using the X-RapidAPI-Key header.

Construct API Requests

- Use the information from the API documentation to construct the API requests in your code. Set the appropriate URL, headers, and parameters.

Make API Requests

- Use the requests library (or any other library suitable for your programming language) to make requests to the Twitter API endpoint. This is where you'll specify the action you want to perform (e.g., retrieve user data).

Handle API Responses

- Check the HTTP status code in the API response (response.status_code). If it's 200 (OK), the request was successful. Parse the response content (usually in JSON format) to extract the data you need.

Iterate and Process Data

- If you are making multiple requests or processing a list of users, iterate through the data and perform any necessary processing.

Handle Errors

- Implement error handling to deal with cases where the API request is unsuccessful. Print or log error details for debugging.

Comply with Rate Limits

- Check the API documentation for rate limits. Ensure that your code compiles with these limits to avoid being blocked.

Test Your Code

- Test your code with a small set of data to ensure that it works as expected before scaling up.

Monitor Usage

- Keep an eye on your API usage, especially if there are usage limits or quotas. Monitor for any notifications from RapidAPI regarding your API usage.

Overview of the Project:

Below is the line wise explanation of the code. Initially, we want to use two special sets of tools (libraries) in our program.

1. *requests*: This library helps us ask the internet for information. Imagine you're asking a friend for some data, and this library is like your messenger.

2. *pandas as pd*: This library is like a super organized notebook for handling and working with data. It helps us store and manipulate information neatly.

So, by saying import requests and import pandas as pd, we're basically telling the computer, "Hey, get ready! We're going to use these tools to bring in data from the internet and organize it really well."

```
import requests
import pandas as pd
```

You are setting up two pieces of information that act like your identification card when you request information from a service called RapidAPI. It's like telling the internet, "Hi, I'm [your identification key], and I want to get information from the Twitter service hosted at twitter135.p.rapidapi.com."

So, x_rapidapi_key is your special key that identifies you, and x_rapidapi_host is like the address of the place on the internet where you want to get information (in this case, the Twitter service). These values are usually provided by the service you're connecting to, and you should keep them secret, just like your password.

```
x_rapidapi_key = "ea1dace47fmsh2b2889e487740d7p187dc2jsnb7d164499b38"
x_rapidapi_host = "twitter135.p.rapidapi.com"
```

Here's what's happening:

1. *input("Enter Twitter usernames (comma-separated): "):* This line displays a message ("Enter Twitter usernames (comma-separated): ") on the screen and waits for the user to type something. Whatever the user types is stored in the variable usernames_input. It's like asking the user to give a list of Twitter usernames, and they type it in.

2. *usernames = [username.strip() for username in usernames_input.split(',')]*: This line takes the input received from the user (which is a string containing usernames separated by commas) and turns it into a list of usernames. The split(',') part splits the string into a list wherever there's a comma, and username.strip() removes any extra spaces around each username. So, if the user entered something like "user1, user2, user3", this line turns it into the list ['user1', 'user2', 'user3'].

```
usernames_input = input("Enter Twitter usernames (comma-separated): ")
usernames = [username.strip() for username in usernames_input.split(',')]
```

This line of code is constructing a URL for making a specific type of request to a web service. Here's what's happening:

1. *f"https://{x_rapidapi_host}/v2/UserByScreenName/"*: This is called an f-string in Python. It allows you to embed variables directly into a string. In this case, it's creating a URL by combining the protocol ("https://"), the RapidAPI host (x_rapidapi_host), and the specific endpoint or path ("/v2/UserByScreenName/"). This URL is where the program will send a request to get information about a Twitter user based on their screen name.

So, if x_rapidapi_host is "twitter135.p.rapidapi.com", the constructed URL becomes something like "https://twitter135.p.rapidapi.com/v2/UserByScreenName/". This URL is used to communicate with the Twitter API hosted at RapidAPI.

```
api_url = f"https://{x_rapidapi_host}/v2/UserByScreenName/"
```

This line of code is defining a set of headers that will be included in the request sent to the RapidAPI service.
Here's what each part does:

1. *"X-RapidAPI-Key"*: x_rapidapi_key: This part includes the RapidAPI key (x_rapidapi_key) in the headers. It's like saying, "Hey, RapidAPI, here's my special key. This is who I am."

2. *"X-RapidAPI-Host"*: x_rapidapi_host: This includes the RapidAPI host (x_rapidapi_host) in the headers. It tells RapidAPI which specific service you want to talk to.

3. *"Accept"*: "application/json": This part specifies that the program is willing to accept responses in JSON format. It's like saying, "I prefer to get the data in a format that I can easily work with in my code."

In simpler terms, these headers are like additional information the program sends along with its request to RapidAPI. They help RapidAPI identify the user (using the key), understand which service is being requested (using the host), and know the preferred format for the response (JSON).

```
headers = {
    "X-RapidAPI-Key": x_rapidapi_key,
    "X-RapidAPI-Host": x_rapidapi_host,
    "Accept": "application/json",
}
```

Here's what it does:

1. all_user_data = []: This initializes an empty list in Python. The variable all_user_data is set to an empty list, meaning it currently doesn't contain any data.

In the context of the provided code, it seems like all_user_data is intended to store user data. However, at this point in the code, it's just an empty container waiting to be filled. As the program runs and retrieves user data, it likely appends or adds that data to this list for further processing or analysis.

```
all_user_data = []
```

These lines are fetching data from the Twitter API for each username in the list, and it's making sure to handle any errors that might occur during this process.
Here's a breakdown:

1. *for username in usernames:* This starts a loop that goes through each username in the usernames list one by one.

2. *query_params = {"username": username}*: This line creates a dictionary called query_params with a single key-value pair. It's specifying that the API request should include a parameter called "username" with the current username from the loop.

3. *response = requests.get(api_url, headers=headers, params=query_params)*: This line makes a GET request to the specified api_url (the Twitter API endpoint). It includes the headers (containing the API key and host information) and the query parameters (containing the current username).

4. *response.raise_for_status()*: This checks if the HTTP response status code is not 200 (OK). If the status code is not 200, it raises an exception. This is a way to handle errors. If there's an issue with the request (like the server is down or the API key is incorrect), it will be caught and dealt with.

So, these lines are fetching data from the Twitter API for each username in the list, and it's making sure to handle any errors that might occur during this process.

```
for username in usernames:
    query_params = {"username": username}
    response = requests.get(api_url, headers=headers, params=query_params)
    response.raise_for_status()  # Raise an exception for non-200 status codes
```

This block of code is processing the data received from the Twitter API response when the request is successful.

Here's a breakdown:

1. *if response.ok*: This checks if the response from the Twitter API was successful, meaning the status code is 200 (OK). If it is, the code inside the block will be executed.

2. *user_data = response.json()*: This line converts the response content (which is in JSON format) into a Python dictionary and stores it in the user_data variable.

3. *print(user_data)*: This prints the user data to the console. It's useful for debugging or understanding the structure of the data.

4. *Creating a DataFrame (user_df)*: This part is creating a pandas DataFrame. Each column in the DataFrame corresponds to a specific piece of information about the user (like username, full name, followers count, etc.). The data for each column is extracted from the user_data dictionary.

For Example:
"Username": [user_data["data"]["user"]["result"]["legacy"]["screen_name"]] extracts the screen name from the user_data dictionary and creates a DataFrame column called "Username" with the extracted value.

The resulting DataFrame (user_df) will contain information about a Twitter user, and this DataFrame can be used for further analysis, display, or storage.

```
if response.ok:
    user_data = response.json()
    print(user_data)
    # Create a DataFrame with user data
    user_df = pd.DataFrame({
        "Username": [user_data["data"]["user"]["result"]["legacy"]["screen_name"]],
        "Full Name": [user_data["data"]["user"]["result"]["legacy"]["name"]],
        "Description": [user_data["data"]["user"]["result"]["legacy"]["description"]],
        "Location": [user_data["data"]["user"]["result"]["legacy"]["location"]],
        "Followers Count": [user_data["data"]["user"]["result"]["legacy"]["followers_count"]],
        "Following Count": [user_data["data"]["user"]["result"]["legacy"]["friends_count"]],
        "Tweet Count": [user_data["data"]["user"]["result"]["legacy"]["statuses_count"]],
        "Account Verified": [user_data["data"]["user"]["result"]["legacy"]["verified"]],
        "Media Count": [user_data["data"]["user"]["result"]["legacy"]["media_count"]],
        "Favourites Count": [user_data["data"]["user"]["result"]["legacy"]["favourites_count"]],
    })
```

This part of the code is responsible for printing the user data in a human-readable format:
Here's what each line does:

1. *print("User Data:")*: This line simply prints the text "User Data:" to the console. It's like a header indicating that the following output will be data related to the user.

2. *print(user_df)*: This line prints the entire DataFrame (user_df) to the console. The DataFrame is a structured table-like representation of the user data. When you print it, it shows the data in a tabular format, making it easy to read and understand.

```
print("User Data:")
print(user_df)
```

For Example: when this block of code is executed, you'll see output in the console like.

```
User Data:
   Username Full Name                                Description  Location  ...  Tweet Count  Account Verified  Media Count  Favourites Count
0  dawn_com  Dawn.com  This is http://t.co/vAYXvvKr's live Twitter fe...  Pakistan  ...       292116             False        34111                 0
```

This part of code is responsible for appending to list.

- all_user_data is a list that is initialized before the loop.
- user_df is a DataFrame containing data for a specific user.
- append(user_df) adds the DataFrame for the current user to the list.

This is done to accumulate DataFrames for each user in the list all_user_data. Each element in the list represents the data for one user.

```
    # Append the DataFrame to the list
    all_user_data.append(user_df)
```

This part of code is responsible for concatenating DataFrames.

- pd.concat(all_user_data, ignore_index=True) concatenates all DataFrames in the list all_user_data along the rows (axis=0) into a single DataFrame.
- ignore_index=True resets the index of the resulting DataFrame to a default integer index. This is useful when combining DataFrames with different indices.

The purpose is to combine individual DataFrames for each user into a single DataFrame (final_user_data) that contains data for all users.

```
  # Concatenate all DataFrames in the list
  final_user_data = pd.concat(all_user_data, ignore_index=True)
```

This part of code is responsible for saving to CSV.

- final_user_data.to_csv("multiple_user_data.csv", index=False) saves the combined DataFrame to a CSV file named "multiple_user_data.csv".
- index=False ensures that the index column is not saved in the CSV file.

```
  # Save the DataFrame to a CSV file (mode='a' is not used)
  final_user_data.to_csv("multiple_user_data.csv", index=False)
  print("Successfully saved the data in file.")
```

This block of code handles the case where the response from the Twitter API is not successful. If there is an error, it prints details about the error for debugging purposes:
Here's a breakdown:

1. else: This is the block of code that gets executed if the condition in the previous if statement (if response.ok:) is not satisfied. In other words, if the response status code is not 200 (OK), indicating an unsuccessful request.

2. print(f"Error: {response.status_code}"): This line prints an error message to the console, including the HTTP status code. For example, it might print something like "Error: 404" for a "Not Found" error.

3. print(f"Details: {response.text}"): This line prints additional details about the error. It outputs the response text, which typically contains more information about the error, such as an error message or additional details from the API.

This information is useful for debugging and understanding why a particular request to the Twitter API might have failed. It provides insights into the nature of the error, allowing developers to troubleshoot and fix issues in their code.

```
else:
    print(f"Error: {response.status_code}")
    print(f"Details: {response.text}")  # Print the response text for debugging
```

Conclusion:

- Summary of Key Points

In this guide, we explored a Python script that interacts with the Twitter API through RapidAPI to retrieve user data. The code demonstrated how to handle API requests, process responses, and store data in a CSV file.

- Usage Tips

- Always secure your API key and avoid sharing it publicly.
- Monitor and respect API rate limits to avoid disruptions.
- Review and comply with the terms of service of the APIs used.

- Important Reminders

- Ensure proper error handling for robust code.
- Check for updates or improvements to the code periodically.
- Be mindful of any changes in API behavior that may affect the code.

- Potential Enhancements

If you wish to extend the functionality:
- Consider adding additional data fields or user-related statistics.
- Explore options for visualizing the data, such as charts or graphs.

- Acknowledgments

We acknowledge the use of RapidAPI for simplifying API integration and handling.