

# Youtube Metadata Scraper

I want to build a web scraping python notebook, It should receive a list of Youtube channel page URLs then gather all the metadata for videos under the specified channels of length less than 98 seconds. The metadata of concern are video title, video description, video URL, video publish date, video tags, views, list of comments, comment count, likes, dislikes, channel name, channel URL, channel description (about section), channel subscriber count, channel joining date. The gathered data should be populated in a csv file with each row having a unique ID/index for later retrieval purposes.

Here's a complete Python Jupyter notebook solution to scrape the requested YouTube metadata and store it into a CSV file. The script uses **yt-dlp** (the modern version of youtube-dl) to fetch video metadata efficiently.

[View Notebook](#)

## Explanation of the Notebook:

- **yt-dlp** is employed to extract detailed metadata efficiently from YouTube.
- The script checks each video's duration and includes only videos shorter than or equal to 98 seconds.
- Extracted information includes:
  - **Video Data:** Title, Description, URL, Tags, Publish Date, Views, Comments, Likes, Dislikes.
  - **Channel Data:** Name, URL, Description, Subscriber Count, Join Date.
- Each row is uniquely identified by a UUID.
- Data is saved into a CSV file (`youtube_videos_metadata.csv`) for easy retrieval.

Adjust the `channel_urls` list to specify the channels you want to scrape.

Good! Now update the code so that it downloads the video for each row with 720p resolution (if 720p is unavailable then the next lower resolution and so on) in a specified folder. The downloaded video name should be the unique ID/Index from the corresponding metadata row. Also, add

resolution and downloaded columns containing the downloaded video resolution and whether video was downloaded or not. Keep in mind that I do have legal permission to obtain the video content.

## Youtube Metadata Scraper

```
# Install required libraries
!pip install yt-dlp pandas

import pandas as pd
import yt_dlp
import uuid
import os

# Function to scrape channel and video metadata and download videos
def scrape_channel_videos(channel_urls, duration_limit=98, download_folder='downloaded_videos'):
    os.makedirs(download_folder, exist_ok=True)

    data = []

    ydl_opts_metadata = {
        'quiet': True,
        'extract_flat': False,
        'skip_download': True,
        'ignoreerrors': True,
        'force_generic_extractor': False,
    }

    with yt_dlp.YoutubeDL(ydl_opts_metadata) as ydl:
        for channel_url in channel_urls:
            try:
                channel_info = ydl.extract_info(channel_url, download=False)
                channel_data = {
                    'channel_name': channel_info.get('channel'),
                    'channel_url': channel_info.get('channel_url'),
                    'channel_description': channel_info.get('description'),
                    'channel_subscriber_count': channel_info.get('channel_follower_count'),
                    'channel_join_date': channel_info.get('channel_created'),
                }

                for entry in channel_info.get('entries', []):
                    if entry and entry.get('duration', 0) <= duration_limit:
                        video_comments = [c.get('text') for c in entry.get('comments', [])]

                        video_id = str(uuid.uuid4())
                        downloaded = False
                        resolution = None

                        ydl_opts_download = {
```

```

        'quiet': True,
        'ignoreerrors': True,
        'format': 'bestvideo[height<=720]+bestaudio/best[height<=720]',
        'outtmpl': f'{download_folder}/{video_id}.%({ext}s}',
    }

    with yt_dlp.YoutubeDL(ydl_opts_download) as ydl_download:
        try:
            info_dict = ydl_download.extract_info(entry.get('webpage_url'))
            downloaded = True if info_dict else False
            resolution = info_dict.get('height') if info_dict else None
        except Exception as download_error:
            print(f"Error downloading video {entry.get('webpage_url')}")

            video_data = {
                'id': video_id,
                'video_title': entry.get('title'),
                'video_description': entry.get('description'),
                'video_url': entry.get('webpage_url'),
                'video_publish_date': entry.get('upload_date'),
                'video_tags': entry.get('tags'),
                'views': entry.get('view_count'),
                'comments': video_comments,
                'comment_count': entry.get('comment_count'),
                'likes': entry.get('like_count'),
                'dislikes': entry.get('dislike_count'),
                'resolution': resolution,
                'downloaded': downloaded,
                **channel_data
            }
        data.append(video_data)

    except Exception as e:
        print(f"Error processing {channel_url}: {e}")

return pd.DataFrame(data)

# List of YouTube channel URLs (replace with your URLs)
channel_urls = [
    'https://www.youtube.com/@OpenAI',
    'https://www.youtube.com/@Google'
]

# Scrape, download videos, and save to CSV
df = scrape_channel_videos(channel_urls)
df.to_csv('youtube_videos_metadata.csv', index=False)

# Display dataframe
df.head()

```

The updated notebook now includes functionality to download each video at 720p resolution or the next best available quality, saving videos to a specified folder (`downloaded_videos`). It also adds the resolution and download status to the metadata DataFrame and CSV output.

explain this please:

```
ydl_opts_metadata = {  
    'quiet': True,  
    'extract_flat': False,  
    'skip_download': True,  
    'ignoreerrors': True,  
    'force_generic_extractor': False,  
}
```

Here's a breakdown of what each option in `ydl_opts_metadata` does:

- `quiet: True`
  - Suppresses most of the standard output from **yt-dlp**, making it run in a “quiet” mode. Without this, **yt-dlp** would print download progress and other information to the console.
- `extract_flat: False`
  - Controls how **yt-dlp** parses playlists or channel pages. When `extract_flat` is `True`, **yt-dlp** only retrieves a flat list of video URLs (without detailed metadata for each video). When `False`, **yt-dlp** fully processes each video, retrieving deeper metadata (title, description, duration, etc.).
- `skip_download: True`
  - Tells **yt-dlp** not to download any video/audio content. Instead, it only retrieves metadata. This is helpful when you want to gather video/channel information without saving the actual files.
- `ignoreerrors: True`
  - If **yt-dlp** encounters an error (such as a missing video, region restriction, or other download issues), it will continue instead of stopping altogether. The failed item is simply skipped.
- `force_generic_extractor: False`

- `yt-dlp` supports many specialized extractors for different websites. If `force_generic_extractor` were `True`, `yt-dlp` would ignore all specialized site-specific extractors and try only the generic fallback. Since this is `False`, `yt-dlp` uses the most suitable extractor (in this case, the specialized YouTube extractor) for the best results.

The code didn't work! It kept printing "WARNING: ffmpeg not found. The downloaded format may not be the best available. Installing ffmpeg is strongly recommended: <https://github.com/yt-dlp/yt-dlp#dependencies>" iteratively

The warning message you're encountering—"WARNING: ffmpeg not found. The downloaded format may not be the best available."—indicates that `yt-dlp` requires **FFmpeg** to merge video and audio streams, especially when downloading videos in specific resolutions like 720p.

## Why FFmpeg Is Necessary

YouTube often stores video and audio separately, particularly for higher resolutions. `yt-dlp` downloads these streams individually and relies on **FFmpeg** to merge them into a single playable file. Without FFmpeg, `yt-dlp` can't perform this merging, leading to incomplete downloads or lower-quality videos.

## Installing FFmpeg on Windows

To resolve this issue, follow these steps to install FFmpeg and ensure it's accessible system-wide:

### 1. Download FFmpeg:

- Visit the official FFmpeg download page: <https://ffmpeg.org/download.html>.

Video Production Stack Exchange +1

- Under the Windows section, click on the link to a Windows build. For example, you can use <https://www.gyan.dev/ffmpeg/builds/>. Audacity Support | Audacity Support +1
- Download the latest "release full" zip file.

### 2. Extract the Files:

- Right-click the downloaded zip file and select "Extract All...".
- Choose a location to extract the files, such as `C:\ffmpeg`.

### 3. Add FFmpeg to System PATH:

- Press **Win + R**, type **sysdm.cpl**, and press Enter to open System Properties.  
Super User
- Navigate to the **Advanced** tab and click on **Environment Variables**.
- Under **System variables**, find and select the **Path** variable, then click **Edit**.
- Click **New** and add the path to FFmpeg's **bin** folder, e.g., **C:\ffmpeg\bin**.
- Click **OK** on all open dialogs to apply the changes.

#### 4. Verify the Installation:

- Open Command Prompt (**cmd**) and type **ffmpeg -version**.
- If installed correctly, you'll see FFmpeg's version information displayed.

## ✓ After Installation

Once FFmpeg is installed and added to your system's PATH, rerun your Python notebook. **ytdlp** should now be able to merge video and audio streams seamlessly, eliminating the warning messages.

If you need further assistance or encounter additional issues, feel free to ask!

   Sources

the code is stuck here without error (it is just running indefinitely) when processing the first url:

```
try:
    channel_info = ydl.extract_info(channel_url,
download=False)
    channel_data = {
        'channel_name': channel_info.get('channel'),
        'channel_url': channel_info.get('channel_url'),
        'channel_description':
channel_info.get('description')
        'channel_subscriber_count':
channel_info.get('channel_follower_count'),
        'channel_join_date':
channel_info.get('channel_created'),
    }
```

<https://www.youtube.com/user/Apple>

The issue you're encountering—where the code hangs indefinitely when processing the URL <https://www.youtube.com/user/Apple>—is likely due to how `yt-dlp` handles certain YouTube channel URLs, especially those using the `/user/` format.

## Understanding the Problem

YouTube has multiple URL formats for channels:

- `/channel/` (e.g., [https://www.youtube.com/channel/UC\\_x5XG10V2P6uZZ5FSM9Ttw](https://www.youtube.com/channel/UC_x5XG10V2P6uZZ5FSM9Ttw))
- `/user/` (e.g., <https://www.youtube.com/user/Apple>)
- `@handle` (e.g., <https://www.youtube.com/@Apple>)

The `/user/` format is older and can sometimes cause issues with tools like `yt-dlp`. In some cases, `yt-dlp` may hang or fail to retrieve information from these URLs.

## Recommended Solution

To avoid this issue, it's advisable to use the `/channel/` or `@handle` formats, which are more consistent and reliable with `yt-dlp`.

### Steps to Update the URL:

#### 1. Retrieve the Channel's Unique Identifier:

- Visit the `/user/` URL in a web browser. For example:  
<https://www.youtube.com/user/Apple>.
- Once on the page, right-click and select "View Page Source" or press `Ctrl+U`.
- Search for the string `"channelId"` in the source code.
- You'll find a line like:

```
html
"channelId": "UCYI5VqW2Zs1t1c2Q1uY5zNw"
```

- Copy the channel ID (e.g., `UCYI5VqW2Zs1t1c2Q1uY5zNw`).

#### 2. Construct the `/channel/` URL:

- Replace the `/user/` part with `/channel/` and append the channel ID.
- For example: <https://www.youtube.com/channel/UCYI5VqW2Zs1t1c2Q1uY5zNw>.

#### 3. Update Your Code:

- Replace the old URL in your list: [EndeavourOS +2](#)

```
python
```

```
channel_urls = [
    'https://www.youtube.com/channel/UCYI5VqW2zs1t1c2Q1uY5zNw',
    # other URLs
]
```

By using the `/channel/` format, `yt-dlp` should be able to process the channel without hanging.

## 🔗 Alternative: Use the `@handle` Format

YouTube has introduced `@handle` URLs, which are user-friendly and often more reliable.

Super User +1

- For the Apple channel, the handle is likely `@Apple`.
- The URL would be: `https://www.youtube.com/@Apple`.

You can verify the handle by visiting the channel's page and checking the URL or the channel's "About" section.

### Update Your Code:

```
python
```

```
channel_urls = [
    'https://www.youtube.com/@Apple',
    # other URLs
]
```

Using the `@handle` format can be more straightforward and is supported by `yt-dlp`.

## 🔍 Additional Tips

- Ensure `yt-dlp` is Up-to-Date:

- Run `yt-dlp --version` to check your current version. EndeavourOS
- Update if necessary: Stack Overflow

```
bash
```

```
pip install -U yt-dlp
```

- **Use Verbose Mode for Debugging:**

- If issues persist, enable verbose output to get more detailed logs:

```
python

ydl_opts = {
    'quiet': False,
    'verbose': True,
    # other options
}
```

- **Handle Exceptions Gracefully:**

- Wrap your `extract_info` calls in try-except blocks to catch and log errors without halting the entire process.

By updating the channel URLs to the `/channel/` or `@handle` formats and ensuring your tools are up-to-date, you should be able to avoid the hanging issue and successfully retrieve the desired metadata.

 Sources

the code still hangs even after updating the format of the URLs to all the above suggestions!

I'm sorry to hear that the issue persists even after updating the YouTube channel URLs. Let's explore some potential reasons and solutions for the hanging behavior you're experiencing with `yt-dlp`.

---

## Potential Causes and Solutions

### 1. IPv6 Connectivity Issues:

- **Cause:** Some users have reported that `yt-dlp` hangs due to IPv6-related network issues.
- **Solution:** Force `yt-dlp` to use IPv4 by adding the `--force-ipv4` option. Ask Ubuntu

```
python
```

```
ydl_opts = {
    'quiet': True,
    'extract_flat': False,
    'skip_download': True,
    'ignoreerrors': True,
    'force_generic_extractor': False,
    'force_ipv4': True, # Add this line
}
```

## 2. Outdated yt-dlp Version:

- **Cause:** An outdated version may lack fixes for recent YouTube changes.
- **Solution:** Update `yt-dlp` to the latest version.

```
bash
```

```
pip install -U yt-dlp
```

## 3. YouTube's Bot Detection Mechanisms:

- **Cause:** YouTube may block automated requests, causing `yt-dlp` to hang.
- **Solution:** Use cookies from your browser session to authenticate requests.
  - Export cookies using browser extensions like [Get cookies.txt](#).
  - Then, modify your `yt-dlp` options to include the cookies file:

```
python
```

```
ydl_opts = {
    'quiet': True,
    'extract_flat': False,
    'skip_download': True,
    'ignoreerrors': True,
    'force_generic_extractor': False,
    'cookiefile': 'path/to/cookies.txt', # Add this line
}
```

## 4. Verbose Logging for Debugging:

- **Cause:** Lack of detailed logs makes it hard to pinpoint issues.
- **Solution:** Enable verbose logging to get more insight.

```
python
```

```
ydl_opts = {
    'quiet': False,
    'verbose': True,
    'extract_flat': False,
    'skip_download': True,
    'ignoreerrors': True,
    'force_generic_extractor': False,
}
```

## Alternative Approach: Using YouTube Data API

If `yt-dlp` continues to hang, consider using the [YouTube Data API](#) to fetch channel and video metadata. This method is more reliable and less prone to issues caused by YouTube's frontend changes.

### Steps:

#### 1. Set Up API Access:

- Create a project in the [Google Developers Console](#).
- Enable the YouTube Data API v3.
- Generate an API key.

#### 2. Fetch Channel Videos:

- Use the `channels.list` method to get the `uploads` playlist ID.
- Use the `playlistItems.list` method to retrieve videos from the playlist.
- Use the `videos.list` method to get detailed metadata for each video.

#### 3. Filter Videos by Duration:

- The `videos.list` response includes the `contentDetails.duration` field in ISO 8601 format (e.g., `PT1M30S` for 1 minute 30 seconds).
- Parse this duration and filter out videos longer than 98 seconds.

#### 4. Download Videos:

- Once you have the video URLs, you can use `yt-dlp` to download them individually, reducing the risk of hanging.

**Note:** Using the YouTube Data API has quota limits. Ensure you handle quota usage appropriately.

## 📌 Summary

- Try adding `--force-ipv4` to your `yt-dlp` options to address potential IPv6 issues.  
Ask Ubuntu
- Ensure you're using the latest version of `yt-dlp`.
- Consider using browser cookies to authenticate your requests.
- Enable verbose logging to get more detailed error messages.
- If issues persist, use the YouTube Data API to fetch metadata and download videos individually.

If you need assistance implementing any of these solutions or have further questions, feel free to ask!

ask Sources

Still doesn't work! Please help!

```
import pandas as pd
import yt_dlp
import uuid
import os
import time

# Function to scrape channel and video metadata and
# download videos
def scrape_channel_videos(channel_urls, duration_limit=98,
                           download_folder='downloaded_videos'):
    os.makedirs(download_folder, exist_ok=True)

    data = []

    ydl_opts_metadata = {
        'quiet': True,
        'extract_flat': False,
```

```

'skip_download': True,
'ignoreerrors': True,
'force_generic_extractor': False,
'force_ipv4': True,
'cookiefile': 'www.youtube.com_cookies.txt',
}

with yt_dlp.YoutubeDL(ydl_opts_metadata) as ydl:
    print("Starting channel scraping...")
    for channel_url in channel_urls:
        print(f"Processing channel: {channel_url}")
        try:
            time.sleep(5)
            channel_info = ydl.extract_info(channel_url,
download=False)
            channel_data = {
                'channel_name': channel_info.get('channel'),
                'channel_url': channel_info.get('channel_url'),
                'channel_description':
channel_info.get('description'),
                'channel_subscriber_count':
channel_info.get('channel_follower_count'),
                'channel_join_date':
channel_info.get('channel_created'),
            }
            print(f"Processing channel:
{channel_info.get('channel')}")


for entry in channel_info.get('entries', []):
    time.sleep(5)
    print(f"Processing video: {entry.get('title')}")"
    if entry and entry.get('duration', 0) <=
duration_limit and entry.get('view_count') >=
channel_info.get('channel_follower_count'):
        video_comments = [c.get('text') for c in
entry.get('comments', [])] if entry.get('comments') else []

        video_id = str(uuid.uuid4())
        downloaded = False
        resolution = None


ydl_opts_download = {

```

```

        'quiet': True,
        'ignoreerrors': True,
        'format':
        'bestvideo[height<=720]+bestaudio/best[height<=720]',
        'outtmpl': f'{download_folder}/{video_id}.%{ext}s',
    }
    print(f"Downloading video
{entry.get('webpage_url')}")

    time.sleep(5)
    with yt_dlp.YoutubeDL(ydl_opts_download) as
ydl_download:
        try:
            info_dict =
ydl_download.extract_info(entry.get('webpage_url'),
download=True)
            downloaded = True if info_dict else False
            resolution = info_dict.get('height') if
info_dict else None
            print(f"Downloaded video
{entry.get('webpage_url')}")

        except Exception as download_error:
            print(f"Error downloading video
{entry.get('webpage_url')}: {download_error}")

    video_data = {
        'id': video_id,
        'video_title': entry.get('title'),
        'video_description': entry.get('description'),
        'video_url': entry.get('webpage_url'),
        'video_publish_date':
entry.get('upload_date'),
        'video_tags': entry.get('tags'),
        'views': entry.get('view_count'),
        'comments': video_comments,
        'comment_count':
entry.get('comment_count'),
        'likes': entry.get('like_count'),
        'dislikes': entry.get('dislike_count'),
        'resolution': resolution,
        'downloaded': downloaded,
    }

```

```

        **channel_data
    }

        data.append(video_data)
        print(f"Processed video
{entry.get('webpage_url')}")

    except Exception as e:
        print(f"Error processing {channel_url}: {e}")

    return pd.DataFrame(data)

# List of YouTube channel URLs (replace with your URLs)
channel_urls = [
    'https://www.youtube.com/channel/UCNhxq7He5p-
_FdBh0OaxcQg', #https://www.youtube.com/@Nike

'https://www.youtube.com/channel/UCuLUOxd7ezJ8c6NSLB
NRRfg', #https://www.youtube.com/@adidas

'https://www.youtube.com/channel/UCbpUSVxiBSjc0CHm0k
sujgw', #https://www.youtube.com/@McDonalds
'https://www.youtube.com/channel/UCYFQ33UIPERYx8-
ZHucZbDA', #https://www.youtube.com/@Apple

'https://www.youtube.com/channel/UCnEdfCdbxJJ9ouWKLS
RCRRw', #https://www.youtube.com/@Samsung

'https://www.youtube.com/channel/UC8VddvuHJzIj__Ud0rY
2_ww', #https://www.youtube.com/@redbull
'https://www.youtube.com/channel/UC-
WMwOzgFdvvGVLB1EZ-n-w',
#https://www.youtube.com/@GoPro

'https://www.youtube.com/channel/UC1xnncYc7586km_rIY
QLtLQ', #https://www.youtube.com/@Uber

'https://www.youtube.com/channel/UCGie8GMIUo3kBKIop
dvumVQ', #https://www.youtube.com/@Netflix

'https://www.youtube.com/channel/UCIrgJInjLS2BhlHOMD

```

```
W7v0g', #https://www.youtube.com/@disney
```

```
'https://www.youtube.com/channel/UC9G8DcGtPfHsVEfUT  
M_TjEw', #https://www.youtube.com/@intel
```

```
'https://www.youtube.com/channel/UCxGq825hl0AHP18I9-  
JGKgg', #https://www.youtube.com/@amazon
```

```
'https://www.youtube.com/channel/UCnba_sSOe_umiHCpY  
YvRCqQ', #https://www.youtube.com/@Microsoft
```

```
'https://www.youtube.com/channel/UCDCIVTeg-_V-  
xmFV8kAkblg', #https://www.youtube.com/@Sony
```

```
]
```

```
# Scrape, download videos, and save to CSV
```

```
df = scrape_channel_videos(channel_urls)
```

```
df.to_csv('youtube_videos_metadata.csv', index=False)
```

```
# Display dataframe
```

```
df.head()
```

Thanks for sharing your full code! To troubleshoot further and help you fix the hanging issue, could you please confirm a few things:

1. Are you running this in a local environment, Colab, or a cloud-based notebook?
2. Have you verified that your `www.youtube.com_cookies.txt` file exists and is valid?
3. Do any of the channels ever proceed past the `ydl.extract_info()` step, or does it hang consistently on the first one?
4. Do you want me to help you refactor this to use the YouTube Data API v3 instead for better reliability?

Once I have this info, I can guide you on the most effective fix.

Yes I'm running it in a local conda environment.

Yes, `www.youtube.com_cookies.txt` exists in the same directory as my notebook.

None of the channels made it past the `ydl.extract_info()` step, they always hang on that step.

If the YouTube Data API v3 is a reliable choice then yes.

As you can see, I've even introduced a 5 seconds wait before each API call to mimic the behavior of a human.

Keep doing that if it helps, I have no concerns for the code taking a long time to execute.

Thanks for confirming all that. I'll prepare a reliable and complete solution using the YouTube Data API v3 that mimics human-like delays and gathers metadata only for videos shorter than 98 seconds from the specified list of channels. It will also download the videos in the best resolution up to 720p, naming them by unique ID, and saving metadata in a CSV.

I'll get started and update you shortly with working code and API usage guidance.

# YouTube Channel Videos Data Collection Notebook

This notebook demonstrates how to use the **YouTube Data API v3** to gather video data from one or more YouTube channels, then download qualifying videos. We will:

- Accept a list of YouTube channel URLs in various formats (handle, channel ID, or legacy username).
- Determine each channel's unique **Channel ID** using the API (including support for YouTube handles via the `forHandle` parameter [developers.google.com](https://developers.google.com) ).
- Retrieve each channel's **uploads playlist ID**, which contains all uploaded videos [developers.google.com](https://developers.google.com) , and fetch *all videos* from that playlist.
- For each video, get detailed **metadata**: title, description, publish date, tags, view count, like count, dislike count (if available), comment count, duration, and video URL.
- Filter to include only videos shorter than **98 seconds** (i.e. 97 seconds or less).
- Retrieve **comments** (text only) for each video (top-level comments) using the CommentThreads API.
- Download each qualifying video in the best quality up to 720p (or nearest lower quality) using the **pytube** library, saving files with the video's unique ID as the filename. Record the downloaded resolution and status.
- Collect all channel-level and video-level data into a **pandas DataFrame** (one row per video) and save it to a CSV file.

**Note:** We include a 5-second delay before each API call or download to respect API rate limits and mimic human behavior. Ensure you have a valid YouTube Data API key before running this notebook (replace `YOUR_API_KEY` with your actual key).

## Setup and Imports

First, install and import the necessary libraries. We use `requests` for HTTP calls to the YouTube Data API, `pandas` for the DataFrame, and `pytube` for downloading videos. We also import `time` for delays and other helpers. If these libraries are not already installed in your environment, uncomment the `!pip install` lines below to install them.

```
python  
  
!pip install pandas pytube requests
```

Now import the libraries and define our API key and input channel list:

```
python  
  
import requests  
import pandas as pd  
import time  
import os  
from urllib.parse import urlparse, parse_qs, unquote  
  
# **Configuration:** Insert your YouTube Data API v3 key here  
API_KEY = "YOUR_API_KEY"  
  
# **Input:** List of YouTube channel URLs to process  
channel_urls = [  
    "https://www.youtube.com/@GoogleDevelopers",      # example handle  
    "https://www.youtube.com/channel/UC_x5XG10V2P6uZZ5FSM9Ttw",  # example channel  
    ID  
    "https://www.youtube.com/user/Google",            # example legacy username  
    # Add more channel URLs as needed...  
]
```

**Explanation:** In the `channel_urls` list, you can include channel page URLs in any of the following forms:

- **Handle URL:** e.g. `https://www.youtube.com/@YourChannelHandle` (introduced in 2022).
- **Channel ID URL:** e.g. `https://www.youtube.com/channel/UCabcdefgij...` (the `UC...` string is the unique Channel ID).
- **Legacy Username URL:** e.g. `https://www.youtube.com/user/YourUsername` (older channels with a legacy username).
- **Custom URL (legacy custom name):** e.g. `https://www.youtube.com/c/CustomName` (some channels have a custom URL; the script will attempt to resolve these via search).

Ensure the URLs are properly formatted strings. We will retrieve the unique Channel ID for each channel using the appropriate API method (handles via `forHandle`, usernames via `forUsername`, or direct ID) [developers.google.com](https://developers.google.com).

## Helper Functions: Channel ID Extraction and Duration Parsing

We define helper functions to (1) extract the Channel ID and basic channel info from a given channel URL, and (2) parse video duration strings (ISO 8601 format) into seconds for filtering.

### Channel ID Extraction Logic:

Depending on the URL format, we use different strategies to get the Channel ID:

- If the URL contains `/channel/`, we can parse the Channel ID directly from the URL.
- If it contains `/user/`, we call the `channels.list` endpoint with the `forUsername` parameter to get the Channel ID [developers.google.com](https://developers.google.com).
- If it contains an `@` (handle), we call `channels.list` with the new `forHandle` parameter (handles may be passed with or without the `@` prefix) [developers.google.com](https://developers.google.com).
- If it contains `/c/` (custom URL), which the API doesn't directly support, we use the Search API to find the channel by name as a fallback.

After obtaining a valid Channel ID, we retrieve the channel's snippet, statistics, and contentDetails in one API call. This provides us with channel metadata (name, description, published date, subscriber count) and the uploads playlist ID (under `contentDetails.relatedPlaylists/uploads`) [developers.google.com](#).

## Duration Parsing:

YouTube API returns video durations in ISO 8601 format (e.g. `PT1M37S` for 1 minute 37 seconds) [developers.google.com](#). We create a function to parse these strings into a total seconds count for easy comparison to our 98-second threshold.

Let's implement these helper functions:

python

```
def parse_iso8601_duration(duration: str) -> int:
    """
    Parse an ISO 8601 duration string (PT#H#M#S) to total seconds.
    """

    # Remove the 'PT' prefix and prepare to parse hours, minutes, seconds
    time_str = duration.replace('PT', '')
    hours = minutes = seconds = 0
    # Find hours, minutes, seconds in the string if they exist
    if 'H' in time_str:
        hours_part, time_str = time_str.split('H')
        hours = int(hours_part)
    if 'M' in time_str:
        minutes_part, time_str = time_str.split('M')
        if minutes_part != '':
            minutes = int(minutes_part)
        else:
            minutes = 0
    if 'S' in time_str:
        seconds_part = time_str.split('S')[0]
        if seconds_part != '':
            seconds = int(seconds_part)
        else:
            seconds = 0
    total_seconds = hours * 3600 + minutes * 60 + seconds
    return total_seconds

def get_channel_data(api_key: str, channel_url: str):
```

```

"""
Given a YouTube channel URL (handle, channel ID, or username),
retrieve the channel's unique ID, snippet, statistics, and uploads playlist ID.
Returns a dict with channel metadata and the uploads playlist ID.
"""

# Normalize URL and identify type
url = channel_url.strip()
# Prepend scheme if not present
if url.startswith("/"):
    url = "https://www.youtube.com" + url
parsed = urlparse(url)
path = parsed.path # e.g. "@/ChannelHandle", "/channel/UC123...", "/user/Name",
"/c/Custom"
query = parsed.query

channel_id = None
params = {} # parameters for channels.list
base_api_url = "https://www.googleapis.com/youtube/v3/channels"

if "/channel/" in path:
    # URL form: /channel/UCxxxxxxxxx
    channel_id = path.split("/channel/")[1].split("/")[0]
    params = {"id": channel_id}
elif "/user/" in path:
    # URL form: /user/Username
    username = path.split("/user/")[1].split("/")[0]
    params = {"forUsername": username}
elif "@" in path:
    # URL form: /@Handle
    # Extract handle (remove leading "@/" if present)
    handle = path.split("@")[1]
    # Remove any trailing slash or query if present
    handle = handle.split("/")[0]
    params = {"forHandle": handle}
elif "/c/" in path:
    # URL form: /c/CustomName (no direct API support; use search as fallback)
    custom_name = path.split("/c/")[1].split("/")[0]
    # Decode URL-encoded characters if any
    custom_name = unquote(custom_name)
    # Use YouTube Search API to find channel by this name
    search_url = "https://www.googleapis.com/youtube/v3/search"
    search_params = {

```

```

        "part": "snippet",
        "q": custom_name,
        "type": "channel",
        "maxResults": 1,
        "key": api_key
    }
    time.sleep(5) # delay before API call
    resp = requests.get(search_url, params=search_params)
    data = resp.json()
    if data.get("items"):
        channel_id = data["items"][0]["id"]["channelId"]
        params = {"id": channel_id}
    else:
        raise Exception(f"Channel not found for custom URL: {custom_name}")
else:
    # If none of the known patterns matched, perhaps the input is directly a
    channel ID or handle without prefix
    # For example, just "UC_xxx" or "ChannelHandle". We try to detect a channel
    ID by pattern (starts with UC).
    if path and path.startswith("/"):
        possible_id = path[1:]
    else:
        possible_id = path
    if possible_id.startswith("UC"):
        channel_id = possible_id
        params = {"id": channel_id}
    else:
        # Try as handle (without @) if it looks like a word
        if possible_id:
            params = {"forHandle": possible_id}
        else:
            raise Exception(f"Unrecognized channel URL format: {channel_url}")
# Now call the YouTube Data API to get channel info
params.update({
    "part": "snippet,contentDetails,statistics",
    "key": api_key
})
time.sleep(5) # delay before API call
resp = requests.get(base_api_url, params=params)
resp_data = resp.json()
if not resp_data.get("items"):
    raise Exception(f"No channel data found for URL: {channel_url}")

```

```

channel_info = resp_data["items"][0]
# Extract needed fields
snippet = channel_info.get("snippet", {})
statistics = channel_info.get("statistics", {})
content_details = channel_info.get("contentDetails", {})
channel_id = channel_info.get("id") # final channel ID
channel_name = snippet.get("title", "")
channel_description = snippet.get("description", "")
channel_join_date = snippet.get("publishedAt", "") # ISO date-time of channel
creation
subscriber_count = statistics.get("subscriberCount", None)
if subscriber_count is not None:
    # Convert subscriber count to int if present (it's a string in the API
response)
    subscriber_count = int(subscriber_count)
# Get uploads playlist ID
uploads_playlist_id = None
if content_details.get("relatedPlaylists"):
    uploads_playlist_id = content_details["relatedPlaylists"].get("uploads")
return {
    "channel_id": channel_id,
    "channel_name": channel_name,
    "channel_description": channel_description,
    "channel_join_date": channel_join_date,
    "subscriber_count": subscriber_count,
    "uploads_playlist_id": uploads_playlist_id,
    "input_url": channel_url # keep original URL (optional)
}

# Quick test of duration parser:
test_durations = ["PT1M37S", "PT2M", "PT45S", "PT1H2M3S", "PT98S"]
print({d: parse_iso8601_duration(d) for d in test_durations})

```

In the code above, `get_channel_data` will return a dictionary containing:

- `channel_id` : the unique Channel ID (starting with "UC...")
- Other channel metadata: name, description, join date (the channel's creation date), subscriber count (as an integer, if available), and the `uploads_playlist_id` which we will use to list videos.

The test for `parse_iso8601_duration` prints a quick check of a few examples (including "PT98S" which should parse to 98 seconds) to ensure our parser works correctly.

Let's run these helper functions:

# YouTube Channel Videos Data Collection Notebook

This notebook demonstrates how to use the **YouTube Data API v3** to gather video data from one or more YouTube channels, then download qualifying videos. We will:

- Accept a list of YouTube channel URLs in various formats (handle, channel ID, or legacy username).
- Determine each channel's unique **Channel ID** using the API (including support for YouTube handles via the `forHandle` parameter).
- Retrieve each channel's **uploads playlist ID**, which contains all uploaded video, and fetch *all videos* from that playlist (handling pagination if necessary).
- For each video, get detailed **metadata**: title, description, publish date, tags, view count, like count, dislike count (if possible), comment count, duration, and video URL.
- Filter to include only videos shorter than **98 seconds** (i.e. 97 seconds or less).
- Retrieve **comments** (text only) for each video (top-level comments using the CommentThreads API).
- Download each qualifying video in the best quality up to 720p (or nearest lower quality) using the **pytube** library, saving files with the video's unique ID as the filename. Record the downloaded resolution and status.
- Collect all channel-level and video-level data into a **pandas DataFrame** (one row per video) and save it to a CSV file.

**Note:** We include a 5-second delay before each API call or download to respect API rate limits and mimic human behavior. Ensure you have a valid YouTube Data API key before running this notebook (replace `YOUR_API_KEY` with your actual key). The YouTube Data API must be enabled for your project in the Google Developers Console.

## Setup and Imports

First, install and import the necessary libraries. We use `requests` for HTTP calls to the YouTube Data API, `pandas` for the DataFrame, and `pytube` for downloading videos. We also import `time` for delays and other helpers. If these libraries are not already installed in your environment, uncomment the `!pip install` lines below to install them.

```
python

# If running in a new environment, uncomment the installs:
# !pip install pandas pytube requests

import requests
import pandas as pd
import time
import os
from urllib.parse import urlparse, unquote

# **Configuration:** Insert your YouTube Data API v3 key here
API_KEY = "YOUR_API_KEY"

# **Input:** List of YouTube channel URLs to process
channel_urls = [
    "https://www.youtube.com/@GoogleDevelopers",      # example handle
    "https://www.youtube.com/channel/UC_x5XG10V2P6uZZ5FSM9Ttw",  # example channel
    "https://www.youtube.com/user/Google",            # example legacy username
    # Add more channel URLs as needed...
]
```

In the `channel_urls` list above, you can include channel page URLs in any of the following forms:

- **Handle URL:** e.g. `https://www.youtube.com/@YourChannelHandle` (new YouTube handle format).
- **Channel ID URL:** e.g. `https://www.youtube.com/channel/UCabcdefghijklmn...`  (the `UC...` string is the unique Channel ID).
- **Legacy Username URL:** e.g. `https://www.youtube.com/user/YourUsername` (older channels with a legacy username).
- **Legacy Custom URL:** e.g. `https://www.youtube.com/c/CustomName` (older custom URLs; the script will handle these via a search fallback).

Make sure the URLs are properly formatted as strings. We will retrieve each channel's Channel ID using the appropriate API query (handles via `forHandle`, usernames via `forUsername`, or direct channel ID) .

## Helper Functions

We define two helper functions:

1. `parse_iso8601_duration` : to convert YouTube's ISO 8601 duration format into total seconds (for filtering short videos). For example, a video duration of `"PT1M37S"` (1 minute 37 seconds) will be parsed to `97` second developers.google.com ] .
2. `get_channel_data` : to extract a channel's ID and metadata from a given channel URL. This function handles different URL patterns and uses the appropriate YouTube Data API call to get channel details and the uploads playlist ID (which lists all uploaded videos for the channel).

Let's implement these:

```
python

def parse_iso8601_duration(duration: str) -> int:
    """
    Parse an ISO 8601 duration string (e.g. 'PT1H2M3S', 'PT4M5S', 'PT30S') to total
    seconds.

    """
    time_str = duration.replace('PT', '') # remove the 'PT' prefix
    hours = minutes = seconds = 0
    if 'H' in time_str:
        hours_part, time_str = time_str.split('H')
        hours = int(hours_part) if hours_part else 0
    if 'M' in time_str:
        minutes_part, time_str = time_str.split('M')
        minutes = int(minutes_part) if minutes_part else 0
    if 'S' in time_str:
        seconds_part = time_str.split('S')[0]
        seconds = int(seconds_part) if seconds_part else 0
    return hours * 3600 + minutes * 60 + seconds

def get_channel_data(api_key: str, channel_url: str):
```

```

"""
Given a YouTube channel URL (handle, channel ID, legacy username, or custom
URL),
    retrieve the channel's unique ID, snippet, statistics, and uploads playlist ID.
    Returns a dictionary with channel metadata and the uploads playlist ID.
"""

url = channel_url.strip()
# Prepend scheme if the URL is provided in shorthand (e.g. "@/handle")
if url.startswith("/"):
    url = "https://www.youtube.com" + url
parsed = urlparse(url)
path = parsed.path # e.g. "@/ChannelHandle", "/channel/UC123...", "/user/Name",
"/c/Custom"

base_channels_url = "https://www.googleapis.com/youtube/v3/channels"
params = {"part": "snippet,contentDetails,statistics", "key": api_key}

# Determine query param based on URL pattern
if path.startswith("/channel/"):
    # URL form: /channel/UCxxxxxxxxx
    channel_id = path.split("/channel/")[1]
    params["id"] = channel_id
elif path.startswith("/user/"):
    # URL form: /user/Username (legacy username)
    username = path.split("/user/")[1]
    params["forUsername"] = username
elif path.startswith("/@"):
    # URL form: @Handle (new handle format)
    handle = path.split("/@")[1]
    # Remove trailing slash if any
    handle = handle.rstrip("/")
    # The API forHandle parameter accepts handle with or without ]
    params["forHandle"] = handle
elif path.startswith("/c/"):
    # URL form: /c/CustomName (legacy custom URL not directly supported by API)
    custom = path.split("/c/")[1]
    custom = custom.rstrip("/")
    custom = unquote(custom) # decode URL encoding if any
    # Use search API to resolve custom channel name to channelId
    search_url = "https://www.googleapis.com/youtube/v3/search"
    search_params = {
        "part": "snippet",

```

```

        "q": custom,
        "type": "channel",
        "maxResults": 1,
        "key": api_key
    }
time.sleep(5)
resp = requests.get(search_url, params=search_params)
data = resp.json()
if data.get("items"):
    channel_id = data["items"][0]["id"][" channelId"]
    params["id"] = channel_id
else:
    raise ValueError(f"Channel not found for custom URL: {channel_url}")
else:
    # Perhaps a direct channel ID (starting with UC) or just a handle without
prefix
    identifier = path.strip("/")
    if identifier.startswith("UC"):
        params["id"] = identifier
    elif identifier: # possibly a handle without the @ or some custom string
        params["forHandle"] = identifier
    else:
        raise ValueError(f"Unrecognized channel URL format: {channel_url}")

# Call the Channels API to get channel info
time.sleep(5)
resp = requests.get(base_channels_url, params=params)
data = resp.json()
if not data.get("items"):
    raise ValueError(f"No channel data found for URL: {channel_url}")
channel_info = data["items"][0]

# Extract channel metadata
channel_id = channel_info.get("id")
snippet = channel_info.get("snippet", {})
stats = channel_info.get("statistics", {})
content = channel_info.get("contentDetails", {})

channel_name = snippet.get("title", "")
channel_description = snippet.get("description", "")
channel_join_date = snippet.get("publishedAt", "") # Channel creation date
subscriber_count = stats.get("subscriberCount")

```

```

if subscriber_count is not None:
    subscriber_count = int(subscriber_count)
uploads_playlist_id = None
if content.get("relatedPlaylists"):
    uploads_playlist_id = content["relatedPlaylists"].get("uploads")

return {
    "channel_id": channel_id,
    "channel_name": channel_name,
    "channel_description": channel_description,
    "channel_join_date": channel_join_date,
    "subscriber_count": subscriber_count,
    "uploads_playlist_id": uploads_playlist_id,
    "channel_url": channel_url
}

# Quick test of duration parser:
test_durations = ["PT1M37S", "PT2M", "PT45S", "PT1H2M3S", "PT98S"]
print("Duration parsing test:", {d: parse_iso8601_duration(d) for d in
test_durations})

```

Running the test above will confirm our `parse_iso8601_duration` function. For example, `"PT1M37S"` should parse to 97 seconds, and `"PT98S"` should parse to 98 seconds.

The `get_channel_data` function handles different channel URL patterns and uses `channels.list` with either `id`, `forUsername`, or `forHandle` as needed to get channel information. If a custom URL (`/c/Name`) is provided, it falls back to using the Search API to find the channel's ID. The function returns a dictionary containing the channel's metadata (name, description, join date, subscriber count) and the uploads playlist ID (which we'll need to list videos).

## Retrieve Channel and Video List

Now, using the helper functions, we will iterate through each channel URL, get the channel data, and then retrieve all videos from the channel's uploads playlist. We will store metadata for each video and filter by duration (< 98 seconds) as required.

We'll use the `playlistItems.list` API endpoint to fetch video IDs from the uploads playlist. We need to handle pagination if the channel has many videos (the API returns up to 50 items per page).

After collecting all video IDs (and basic info like title and published date), we will call the `videos.list` endpoint for those IDs to get detailed info (snippet, contentDetails, statistics) in batches of up to 50 IDs at a time (to stay within query parameter limits). This is more efficient than calling the videos API for each video individually.

Let's implement this step:

```
python
```

```
video_records = [] # to collect video data dictionaries for DataFrame

for url in channel_urls:
    try:
        channel_data = get_channel_data(API_KEY, url)
    except Exception as e:
        print(f"Error retrieving channel data for {url}: {e}")
        continue

    channel_name = channel_data["channel_name"]
    print(f"\nProcessing channel: {channel_name}")
    uploads_playlist_id = channel_data.get("uploads_playlist_id")
    if not uploads_playlist_id:
        print(f"No uploads playlist found for channel {channel_name}. Skipping.")
        continue

    # Retrieve all videos from the uploads playlist
    playlist_items_url = "https://www.googleapis.com/youtube/v3/playlistItems"
    playlist_params = {
        "part": "snippet,contentDetails",
        "playlistId": uploads_playlist_id,
        "maxResults": 50,
        "key": API_KEY
    }
```

```

videos_in_playlist = []
while True:
    time.sleep(5)
    resp = requests.get(playlist_items_url, params=playlist_params)
    data = resp.json()
    if "error" in data:
        print(f"API error when fetching playlist items:
{data['error'].get('message')}")
        break
    items = data.get("items", [])
    for item in items:
        # snippet contains title, publishedAt; contentDetails contains videoId
        vid_id = item["contentDetails"]["videoId"]
        vid_title = item["snippet"].get("title", "")
        vid_published = item["snippet"].get("publishedAt", "")
        videos_in_playlist.append({
            "video_id": vid_id,
            "title": vid_title,
            "published_at": vid_published
        })
    # Pagination: check if there's another page
    if "nextPageToken" in data:
        playlist_params["pageToken"] = data["nextPageToken"]
    else:
        break

print(f"Found {len(videos_in_playlist)} videos in channel {channel_name}.")
```

if not videos\_in\_playlist:

continue

# Now retrieve detailed video info in batches of 50

video\_ids = [v["video\_id"] for v in videos\_in\_playlist]

for i in range(0, len(video\_ids), 50):

batch\_ids = video\_ids[i:i+50]

videos\_url = "https://www.googleapis.com/youtube/v3/videos"

videos\_params = {

"part": "snippet,contentDetails,statistics",

"id": ",".join(batch\_ids),

"key": API\_KEY

}

time.sleep(5)

```

resp = requests.get(videos_url, params=videos_params)
data = resp.json()
if "items" not in data:
    continue
for item in data["items"]:
    vid_id = item["id"]
    snippet = item.get("snippet", {})
    content_details = item.get("contentDetails", {})
    stats = item.get("statistics", {})

    # Video metadata fields
    title = snippet.get("title", "")
    description = snippet.get("description", "")
    published_at = snippet.get("publishedAt", "")
    tags = snippet.get("tags", [])
    duration_iso = content_details.get("duration", "")
    duration_seconds = parse_iso8601_duration(duration_iso) if duration_iso
else None
    view_count = int(stats.get("viewCount", 0))
    like_count = int(stats.get("likeCount", 0)) if "likeCount" in stats else
None
        # Dislike count may not be available (private as of Dec 2021]
        dislike_count = int(stats.get("dislikeCount", 0)) if "dislikeCount" in
stats else None
        comment_count = int(stats.get("commentCount", 0)) if "commentCount" in
stats else 0

    # Filter by video length < 98 seconds
    if duration_seconds is None or duration_seconds >= 98:
        continue # skip videos 98s or longer

video_url = f"https://www.youtube.com/watch?v={vid_id}"

# Prepare a record dict for this video
record = {
    "channel_name": channel_name,
    "channel_id": channel_data["channel_id"],
    "channel_url": channel_data["channel_url"],
    "channel_description": channel_data["channel_description"],
    "channel_join_date": channel_data["channel_join_date"],
    "subscriber_count": channel_data["subscriber_count"],
    "video_id": vid_id,
}

```

```

        "video_url": video_url,
        "title": title,
        "description": description,
        "published_at": published_at,
        "tags": ",".join(tags) if tags else "",
        "view_count": view_count,
        "like_count": like_count,
        "dislike_count": dislike_count,
        "comment_count": comment_count,
        "duration_seconds": duration_seconds
    }
video_records.append(record)

```

In the code above:

- We loop through each channel URL, get the channel data (including the `uploads_playlist_id`).
- We use the **PlaylistItems API** to get all videos in the uploads playlist. We include `part=snippet,contentDetails` to get both the `videoId` and basic snippet info (like title and published date for logging). We accumulate all video IDs in `videos_in_playlist`, handling pagination with `nextPageToken`.
- We then use the **Videos API** to get detailed info for up to 50 video IDs at a time. We request `snippet,contentDetails,statistics` parts for each video. This gives us title, description, tags, duration (ISO 8601 format), and statistics (view count, like count, dislike count, comment count) for each video.
- We convert the duration to seconds using our parser and filter out any video that is 98 seconds or longer. Only videos with `duration_seconds < 98` pass through.
- We assemble a `record` dictionary for each qualifying video with all requested fields. The `dislike_count` is included but note that this field will likely be `None` for public data because YouTube made the dislike count private in 2021 (it would only be populated if we had owner-authenticated access). We still include the field for completeness.
- Each `record` is added to the `video_records` list for later use in the DataFrame.

We also print out some progress information, such as how many videos were found in a channel and when we start processing a channel.

## Fetch Comments for Each Video

Next, for each video record, we will retrieve the top-level comments using the **CommentThreads API**. We will fetch comments in plain text (`textFormat="plainText"`) to avoid HTML formatting. If a video has many comments, the script will iterate through pages of comments up to a certain limit to avoid excessive usage (you can adjust or remove the limit as needed).

We will store all retrieved comments (concatenated) in the `record["comments"]` field as a single string (comments separated by newline). We also ensure to respect the delay between each API call.

**Note:** The YouTube API returns comment text in two formats: `textDisplay` (formatted) and `textOriginal` (raw). The `textOriginal` is only returned to the comment's author or the channel owner, so for our API-key-only access, we rely on `textDisplay` with `textFormat=plainText`, which gives a plaintext version of the comment (though it may still have some minor alterations, like replacing URLs with title]).

Let's implement the comment retrieval:

```
python
```

```
# Add a field for comments text in each record
for record in video_records:
    video_id = record["video_id"]
    comment_texts = []
    # Use commentThreads.list to get top-level comments
    comments_url = "https://www.googleapis.com/youtube/v3/commentThreads"
    comments_params = {
        "part": "snippet",
        "videoId": video_id,
        "textFormat": "plainText",
        "maxResults": 50,
        "key": API_KEY
    }
    fetched_count = 0
    max_to_fetch = 100 # limit to 100 comments per video for this example (adjust
    as needed, or remove limit to get all)
    while True:
        time.sleep(5)
        resp = requests.get(comments_url, params=comments_params)
        data = resp.json()
        if "error" in data:
```

```

# If comments are disabled or any error occurs, we break
# (YouTube disables comments on some videos, like those for kids)
# We store an indication if comments are disabled or unavailable.
error_msg = data["error"].get("message", "Unknown error")
print(f"Could not fetch comments for video {video_id}: {error_msg}")
break

items = data.get("items", [])
for item in items:
    top_comment = item["snippet"]["topLevelComment"]
    comment_snippet = top_comment["snippet"]
    comment_text = comment_snippet.get("textDisplay", "")
    comment_texts.append(comment_text.strip())
    fetched_count += 1
    if fetched_count >= max_to_fetch:
        break
if fetched_count >= max_to_fetch:
    # Reached our limit of comments to fetch
    break
if "nextPageToken" in data:
    comments_params["pageToken"] = data["nextPageToken"]
else:
    break

# Join all comments into one string (separated by newlines)
if comment_texts:
    record["comments"] = "\n\n".join(comment_texts)
else:
    # No comments or none fetched
    record["comments"] = ""

```

In the comment fetching loop:

- We set a `max_to_fetch` limit of 100 comments per video for demonstration. You can increase this or remove the limit to fetch all comments, but be mindful of rate limits and performance if videos have thousands of comments.
- We call `commentThreads.list` for each video with `part=snippet` (which includes the `topLevelComment` and its snippet) and `textFormat=plainText` to get plain text comments. We use `maxResults=50` to fetch 50 comments per page.

- We accumulate comments in `comment_texts`. If `comments` are disabled or an error occurs (for example, YouTube may disable comments on certain videos, such as those marked for kids), we catch the error and break out, leaving the comments list empty or noting the issue.
- We paginate through comments using `nextPageToken` until we reach the end or our `max_to_fetch` limit.
- Finally, we join the comments into a single string separated by blank lines and store it in the `record["comments"]`. If no comments were fetched, we store an empty string.

## Download Videos

Now we will download each qualifying video (those with duration < 98 seconds) using the `pytube` library. We will download at the best available resolution up to 720p. The `pytube.YouTube` object provides a `streams` filter; we can use `streams.get_highest_resolution()` which returns the highest resolution progressive stream (this typically maxes out at 720p for combined audio/video stream).

We will create a folder named `downloads` (if it doesn't exist) to store the video files. Each video file will be named with its unique video ID (to avoid duplicates and for easy reference). We also note the resolution of the downloaded stream and update the record with `downloaded_resolution` and `download_status`.

```
python
```

```
# Create a downloads directory if it doesn't exist
download_dir = "downloads"
os.makedirs(download_dir, exist_ok=True)

from pytube import YouTube
for record in video_records:
    video_id = record["video_id"]
    video_url = record["video_url"]
    try:
        time.sleep(5)
        yt = YouTube(video_url)
        # Get the highest resolution stream that includes audio (progressive stream)
        stream = yt.streams.get_highest_resolution()
        if stream is None:
            # If for some reason no progressive stream is found, pick the highest

```

```

itag progressive manually

    stream =
yt.streams.filter(progressive=True).order_by('resolution').desc().first()

    if stream:
        # Download the video to the download directory with video_id as filename
        out_file = stream.download(output_path=download_dir, filename=f'{video_id}.mp4')

        record['downloaded_resolution'] = stream.resolution
        record['download_status'] = "Downloaded"

    else:
        record['downloaded_resolution'] = None
        record['download_status'] = "No progressive stream available"
        print(f"No downloadable stream for video {video_id}.")

except Exception as e:
    record['downloaded_resolution'] = None
    record['download_status'] = f"Failed: {e}"
    print(f"Failed to download {video_url}: {e}")

```

In this code:

- We ensure the `downloads` directory exists.
- For each video, we create a `YouTube` object and attempt to get the highest resolution progressive stream (which will include audio). We prefer the highest up to 720p; `get_highest_resolution()` will typically choose 720p or the highest available under it, since higher qualities often require separate audio stream】 .
- We then call `stream.download`, specifying the `output_path` and a `filename` as `<video_id>.mp4`. This will save the video file in the `downloads` folder.
- We record the `stream.resolution` (e.g. "720p" or "480p") in the `downloaded_resolution` field and note the `download_status` as "Downloaded".
- If no appropriate stream is found or an error occurs, we catch the exception and mark the status accordingly. For example, age-restricted or private videos might fail to download without additional steps. We handle such cases by recording the failure and printing a message.

**Important:** Downloading videos may be subject to YouTube's Terms of Service. Ensure you have permission to download and use the video content. This script is intended for educational or archival purposes where such usage is allowed.

## Create DataFrame and Save to CSV

Finally, we will create a pandas DataFrame from the collected `video_records` and save it as a CSV file. Each row of the CSV corresponds to one video and includes both channel-level information (repeated for each video of that channel) and video-specific information.

We will name the CSV file `youtube_videos_data.csv` (you can change this as needed). We will also display the first few rows of the DataFrame as a quick preview.

```
python
```

```
# Create DataFrame from the records
df = pd.DataFrame(video_records)

# Save to CSV (without index)
df.to_csv("youtube_videos_data.csv", index=False)

print(f"\nSaved data for {len(df)} videos to youtube_videos_data.csv")
df.head(3) # Display first 3 rows as a sample
```

This will produce a CSV file with columns such as:

- `channel_name`, `channel_id`, `channel_url`, `channel_description`,  
`channel_join_date`, `subscriber_count` (channel metadata)
- `video_id`, `video_url`, `title`, `description`, `published_at`, `tags`, `view_count`,  
`like_count`, `dislike_count`, `comment_count`, `duration_seconds` (video metadata)
- `comments` (concatenated top-level comments text)
- `downloaded_resolution`, `download_status` (download information)

You can open the CSV file to analyze the data or import it into other tools. The DataFrame preview (`head`) shows a glimpse of the structured data.

## Notes and Next Steps

- **API Quota Considerations:** Each API call used (`channels.list`, `playlistItems.list`, `videos.list`, `commentThreads.list`) costs quota units (mostly 1 unit each for the types we used). Fetching many videos and comments can use a significant number of quota units if the channel or videos are large. Ensure your quota is sufficient or consider reducing the scope (e.g., limit the number of videos or comments).
- **Rate Limiting:** We added `time.sleep(5)` before each API call and download to throttle the requests. Adjust the delay as needed to comply with any usage guidelines and avoid hitting rate limits.
- **Extensibility:** This script can be extended to handle additional video or channel properties if needed (for example, adding category IDs, video thumbnails, etc., which are available from the API). For extremely large channels, consider using YouTube's pagination and quota strategically (or the YouTube Data API's client library for better efficiency and built-in retry handling).
- **Environment:** The script is meant to run in a standard Python environment (or Jupyter notebook). Make sure to install the required libraries. The `pytube` library can occasionally break due to changes in YouTube's site; ensure you have a relatively recent version of pytube, and consider updating it if you encounter download issues.

By following this notebook, you should be able to gather a comprehensive dataset of short videos from specified YouTube channels, complete with metadata and downloaded files, all structured in a convenient CSV output for further analysis or processing.

# YouTube Channel Videos Data Collection Notebook

This notebook demonstrates how to use the **YouTube Data API v3** to gather video data from one or more YouTube channels, then download qualifying videos. We will:

- Accept a list of YouTube channel URLs in various formats (handle, channel ID, or legacy username).
- Determine each channel's unique **Channel ID** using the API (including support for YouTube handles via the `forHandle` parameter [developers.google.com](https://developers.google.com) ).
- Retrieve each channel's **uploads playlist ID**, which contains all uploaded videos [developers.google.com](https://developers.google.com) , and fetch *all* videos from that playlist.

- For each video, get detailed **metadata**: title, description, publish date, tags, view count, like count, dislike count (if available), comment count, duration, and video URL.
- Filter to include only videos shorter than **98 seconds** (i.e. 97 seconds or less).
- Retrieve **comments** (text only) for each video (top-level comments) using the CommentThreads API.
- Download each qualifying video in the best quality up to 720p (or nearest lower quality) using the **pytube** library, saving files with the video's unique ID as the filename. Record the downloaded resolution and status.
- Collect all channel-level and video-level data into a **pandas DataFrame** (one row per video) and save it to a CSV file.

**Note:** We include a 5-second delay before each API call or download to respect API rate limits and mimic human behavior. Ensure you have a valid YouTube Data API key before running this notebook (replace `YOUR_API_KEY` with your actual key).

## Setup and Imports

First, install and import the necessary libraries. We use `requests` for HTTP calls to the YouTube Data API, `pandas` for the DataFrame, and `pytube` for downloading videos. We also import `time` for delays and other helpers. If these libraries are not already installed in your environment, uncomment the `!pip install` lines below to install them.

```
python
!pip install pandas pytube requests
```

Now import the libraries and define our API key and input channel list:

```
python
import requests
import pandas as pd
import time
import os
from urllib.parse import urlparse, parse_qs, unquote

# **Configuration:** Insert your YouTube Data API v3 key here
```

```

API_KEY = "YOUR_API_KEY"

# **Input:** List of YouTube channel URLs to process
channel_urls = [
    "https://www.youtube.com/@GoogleDevelopers",      # example handle
    "https://www.youtube.com/channel/UC_x5XG10V2P6uZZ5FSM9Ttw",  # example channel
    ID
    "https://www.youtube.com/user/Google",            # example legacy username
    # Add more channel URLs as needed...
]

```

**Explanation:** In the `channel_urls` list, you can include channel page URLs in any of the following forms:

- **Handle URL:** e.g. `https://www.youtube.com/@YourChannelHandle` (introduced in 2022).
- **Channel ID URL:** e.g. `https://www.youtube.com/channel/UCabcdefghij...` (the `UC...` string is the unique Channel ID).
- **Legacy Username URL:** e.g. `https://www.youtube.com/user/YourUsername` (older channels with a legacy username).
- **Custom URL (legacy custom name):** e.g. `https://www.youtube.com/c/CustomName` (some channels have a custom URL; the script will attempt to resolve these via search).

Ensure the URLs are properly formatted strings. We will retrieve the unique Channel ID for each channel using the appropriate API method (handles via `forHandle`, usernames via `forUsername`, or direct ID) [developers.google.com](https://developers.google.com).

## Helper Functions: Channel ID Extraction and Duration Parsing

We define helper functions to (1) extract the Channel ID and basic channel info from a given channel URL, and (2) parse video duration strings (ISO 8601 format) into seconds for filtering.

### Channel ID Extraction Logic:

Depending on the URL format, we use different strategies to get the Channel ID:

- If the URL contains `/channel/`, we can parse the Channel ID directly from the URL.

- If it contains `/user/`, we call the `channels.list` endpoint with the `forUsername` parameter to get the Channel ID [developers.google.com](#).
- If it contains an `@` (handle), we call `channels.list` with the new `forHandle` parameter (handles may be passed with or without the `@` prefix) [developers.google.com](#).
- If it contains `/c/` (custom URL), which the API doesn't directly support, we use the Search API to find the channel by name as a fallback.

After obtaining a valid Channel ID, we retrieve the channel's snippet, statistics, and `contentDetails` in one API call. This provides us with channel metadata (name, description, published date, subscriber count) and the `uploads` playlist ID (under `contentDetails.relatedPlaylists.uploads`) [developers.google.com](#).

### Duration Parsing:

YouTube API returns video durations in ISO 8601 format (e.g. `PT1M37S` for 1 minute 37 seconds) [developers.google.com](#). We create a function to parse these strings into a total seconds count for easy comparison to our 98-second threshold.

Let's implement these helper functions:

```
python

def parse_iso8601_duration(duration: str) -> int:
    """
    Parse an ISO 8601 duration string (PT#H#M#S) to total seconds.
    """

    # Remove the 'PT' prefix and prepare to parse hours, minutes, seconds
    time_str = duration.replace('PT', '')
    hours = minutes = seconds = 0
    # Find hours, minutes, seconds in the string if they exist
    if 'H' in time_str:
        hours_part, time_str = time_str.split('H')
        hours = int(hours_part)
    if 'M' in time_str:
        minutes_part, time_str = time_str.split('M')
        if minutes_part != '':
            minutes = int(minutes_part)
        else:
            minutes = 0
    if 'S' in time_str:
        seconds_part = time_str.split('S')[0]
```

```

    if seconds_part != '':
        seconds = int(seconds_part)
    else:
        seconds = 0
    total_seconds = hours * 3600 + minutes * 60 + seconds
    return total_seconds

def get_channel_data(api_key: str, channel_url: str):
    """
    Given a YouTube channel URL (handle, channel ID, or username),
    retrieve the channel's unique ID, snippet, statistics, and uploads playlist ID.
    Returns a dict with channel metadata and the uploads playlist ID.
    """
    # Normalize URL and identify type
    url = channel_url.strip()
    # Prepend scheme if not present
    if url.startswith("/"):
        url = "https://www.youtube.com" + url
    parsed = urlparse(url)
    path = parsed.path # e.g. "/@ChannelHandle", "/channel/UC123...", "/user/Name",
    "/c/Custom"
    query = parsed.query

    channel_id = None
    params = {} # parameters for channels.list
    base_api_url = "https://www.googleapis.com/youtube/v3/channels"

    if "/channel/" in path:
        # URL form: /channel/UCxxxxxxxxx
        channel_id = path.split("/channel/")[1].split("/")[0]
        params = {"id": channel_id}
    elif "/user/" in path:
        # URL form: /user/Username
        username = path.split("/user/")[1].split("/")[0]
        params = {"forUsername": username}
    elif "@" in path:
        # URL form: /@Handle
        # Extract handle (remove leading "/@" if present)
        handle = path.split("@")[1]
        # Remove any trailing slash or query if present
        handle = handle.split("/")[0]
        params = {"forHandle": handle}

```

```

elif "/c/" in path:
    # URL form: /c/CustomName (no direct API support; use search as fallback)
    custom_name = path.split("/c/")[1].split("/")[0]
    # Decode URL-encoded characters if any
    custom_name = unquote(custom_name)
    # Use YouTube Search API to find channel by this name
    search_url = "https://www.googleapis.com/youtube/v3/search"
    search_params = {
        "part": "snippet",
        "q": custom_name,
        "type": "channel",
        "maxResults": 1,
        "key": api_key
    }
    time.sleep(5) # delay before API call
    resp = requests.get(search_url, params=search_params)
    data = resp.json()
    if data.get("items"):
        channel_id = data["items"][0]["id"]["channelId"]
        params = {"id": channel_id}
    else:
        raise Exception(f"Channel not found for custom URL: {custom_name}")
else:
    # If none of the known patterns matched, perhaps the input is directly a
    # channel ID or handle without prefix
    # For example, just "UC_xxx" or "ChannelHandle". We try to detect a channel
    # ID by pattern (starts with UC).
    if path and path.startswith("/"):
        possible_id = path[1:]
    else:
        possible_id = path
    if possible_id.startswith("UC"):
        channel_id = possible_id
        params = {"id": channel_id}
    else:
        # Try as handle (without @) if it looks like a word
        if possible_id:
            params = {"forHandle": possible_id}
        else:
            raise Exception(f"Unrecognized channel URL format: {channel_url}")
# Now call the YouTube Data API to get channel info
params.update({

```

```

    "part": "snippet,contentDetails,statistics",
    "key": api_key
)
time.sleep(5) # delay before API call
resp = requests.get(base_api_url, params=params)
resp_data = resp.json()
if not resp_data.get("items"):
    raise Exception(f"No channel data found for URL: {channel_url}")
channel_info = resp_data["items"][0]
# Extract needed fields
snippet = channel_info.get("snippet", {})
statistics = channel_info.get("statistics", {})
content_details = channel_info.get("contentDetails", {})
channel_id = channel_info.get("id") # final channel ID
channel_name = snippet.get("title", "")
channel_description = snippet.get("description", "")
channel_join_date = snippet.get("publishedAt", "") # ISO date-time of channel creation
subscriber_count = statistics.get("subscriberCount", None)
if subscriber_count is not None:
    # Convert subscriber count to int if present (it's a string in the API response)
    subscriber_count = int(subscriber_count)
# Get uploads playlist ID
uploads_playlist_id = None
if content_details.get("relatedPlaylists"):
    uploads_playlist_id = content_details["relatedPlaylists"].get("uploads")
return {
    "channel_id": channel_id,
    "channel_name": channel_name,
    "channel_description": channel_description,
    "channel_join_date": channel_join_date,
    "subscriber_count": subscriber_count,
    "uploads_playlist_id": uploads_playlist_id,
    "input_url": channel_url # keep original URL (optional)
}

# Quick test of duration parser:
test_durations = ["PT1M37S", "PT2M", "PT45S", "PT1H2M3S", "PT98S"]
print({d: parse_iso8601_duration(d) for d in test_durations})

```

In the code above, `get_channel_data` will return a dictionary containing:

- `channel_id` : the unique Channel ID (starting with "UC...")
- Other channel metadata: name, description, join date (the channel's creation date), subscriber count (as an integer, if available), and the `uploads_playlist_id` which we will use to list videos.

The test for `parse_iso8601_duration` prints a quick check of a few examples (including "PT98S" which should parse to 98 seconds) to ensure our parser works correctly.

Let's run these helper functions:

# YouTube Channel Videos Data Collection Notebook

This notebook demonstrates how to use the **YouTube Data API v3** to gather video data from one or more YouTube channels, then download qualifying videos. We will:

- Accept a list of YouTube channel URLs in various formats (handle, channel ID, or legacy username).
- Determine each channel's unique **Channel ID** using the API (including support for YouTube handles via the `forHandle` parameter).
- Retrieve each channel's **uploads playlist ID**, which contains all uploaded video, and fetch *all* videos from that playlist (handling pagination if necessary).
- For each video, get detailed **metadata**: title, description, publish date, tags, view count, like count, dislike count (if possible), comment count, duration, and video URL.
- Filter to include only videos shorter than **98 seconds** (i.e. 97 seconds or less).
- Retrieve **comments** (text only) for each video (top-level comments using the CommentThreads API).
- Download each qualifying video in the best quality up to 720p (or nearest lower quality) using the `pytube` library, saving files with the video's unique ID as the filename. Record the downloaded resolution and status.
- Collect all channel-level and video-level data into a **pandas DataFrame** (one row per video) and save it to a CSV file.

**Note:** We include a 5-second delay before each API call or download to respect API rate limits and mimic human behavior. Ensure you have a valid YouTube Data API key before running this notebook (replace `YOUR_API_KEY` with your actual key). The YouTube Data API must be enabled for your project in the Google Developers Console.

## Setup and Imports

First, install and import the necessary libraries. We use `requests` for HTTP calls to the YouTube Data API, `pandas` for the DataFrame, and `pytube` for downloading videos. We also import `time` for delays and other helpers. If these libraries are not already installed in your environment, uncomment the `!pip install` lines below to install them.

```
python

# If running in a new environment, uncomment the installs:
# !pip install pandas pytube requests

import requests
import pandas as pd
import time
import os
from urllib.parse import urlparse, unquote

# **Configuration:** Insert your YouTube Data API v3 key here
API_KEY = "YOUR_API_KEY"

# **Input:** List of YouTube channel URLs to process
channel_urls = [
    "https://www.youtube.com/@GoogleDevelopers",      # example handle
    "https://www.youtube.com/channel/UC_x5XG10V2P6uZZ5FSM9Ttw",  # example channel
    ID
    "https://www.youtube.com/user/Google",            # example legacy username
    # Add more channel URLs as needed...
]
```

In the `channel_urls` list above, you can include channel page URLs in any of the following forms:

- **Handle URL:** e.g. `https://www.youtube.com/@YourChannelHandle` (new YouTube handle format).
- **Channel ID URL:** e.g. `https://www.youtube.com/channel/UCabcdefghijklmn...`  (the `UC...` string is the unique Channel ID).
- **Legacy Username URL:** e.g. `https://www.youtube.com/user/YourUsername` (older channels with a legacy username).
- **Legacy Custom URL:** e.g. `https://www.youtube.com/c/CustomName` (older custom URLs; the script will handle these via a search fallback).

Make sure the URLs are properly formatted as strings. We will retrieve each channel's Channel ID using the appropriate API query (handles via `forHandle`, usernames via `forUsername`, or direct channel ID) .

## Helper Functions

We define two helper functions:

1. `parse_iso8601_duration` : to convert YouTube's ISO 8601 duration format into total seconds (for filtering short videos). For example, a video duration of `"PT1M37S"` (1 minute 37 seconds) will be parsed to `97` second [developers.google.com](#) .
2. `get_channel_data` : to extract a channel's ID and metadata from a given channel URL. This function handles different URL patterns and uses the appropriate YouTube Data API call to get channel details and the uploads playlist ID (which lists all uploaded videos for the channel).

Let's implement these:

python

```
def parse_iso8601_duration(duration: str) -> int:
    """
    Parse an ISO 8601 duration string (e.g. 'PT1H2M3S', 'PT4M5S', 'PT30S') to total
    seconds.
    """
    time_str = duration.replace('PT', '') # remove the 'PT' prefix
    hours = minutes = seconds = 0
    if 'H' in time_str:
```

```

        hours_part, time_str = time_str.split('H')
        hours = int(hours_part) if hours_part else 0
    if 'M' in time_str:
        minutes_part, time_str = time_str.split('M')
        minutes = int(minutes_part) if minutes_part else 0
    if 'S' in time_str:
        seconds_part = time_str.split('S')[0]
        seconds = int(seconds_part) if seconds_part else 0
    return hours * 3600 + minutes * 60 + seconds

def get_channel_data(api_key: str, channel_url: str):
    """
    Given a YouTube channel URL (handle, channel ID, legacy username, or custom URL),
    retrieve the channel's unique ID, snippet, statistics, and uploads playlist ID.
    Returns a dictionary with channel metadata and the uploads playlist ID.
    """
    url = channel_url.strip()
    # Prepend scheme if the URL is provided in shorthand (e.g. "@/handle")
    if url.startswith("/"):
        url = "https://www.youtube.com" + url
    parsed = urlparse(url)
    path = parsed.path # e.g. "@/ChannelHandle", "/channel/UC123...", "/user/Name",
    "/c/Custom"

    base_channels_url = "https://www.googleapis.com/youtube/v3/channels"
    params = {"part": "snippet,contentDetails,statistics", "key": api_key}

    # Determine query param based on URL pattern
    if path.startswith("/channel/"):
        # URL form: /channel/UCxxxxxxxxx
        channel_id = path.split("/channel/")[1]
        params["id"] = channel_id
    elif path.startswith("/user/"):
        # URL form: /user/Username (legacy username)
        username = path.split("/user/")[1]
        params["forUsername"] = username
    elif path.startswith("@"):
        # URL form: @Handle (new handle format)
        handle = path.split("@")[1]
        # Remove trailing slash if any
        handle = handle.rstrip("/")

```

```

# The API forHandle parameter accepts handle with or without ]
params["forHandle"] = handle

elif path.startswith("/c/"):
    # URL form: /c/CustomName (legacy custom URL not directly supported by API)
    custom = path.split("/c/")[1]
    custom = custom.rstrip("/")
    custom = unquote(custom) # decode URL encoding if any
    # Use search API to resolve custom channel name to channelId
    search_url = "https://www.googleapis.com/youtube/v3/search"
    search_params = {
        "part": "snippet",
        "q": custom,
        "type": "channel",
        "maxResults": 1,
        "key": api_key
    }
    time.sleep(5)
    resp = requests.get(search_url, params=search_params)
    data = resp.json()
    if data.get("items"):
        channel_id = data["items"][0]["id"]["channelId"]
        params["id"] = channel_id
    else:
        raise ValueError(f"Channel not found for custom URL: {channel_url}")
else:
    # Perhaps a direct channel ID (starting with UC) or just a handle without
prefix
    identifier = path.strip("/")
    if identifier.startswith("UC"):
        params["id"] = identifier
    elif identifier: # possibly a handle without the @ or some custom string
        params["forHandle"] = identifier
    else:
        raise ValueError(f"Unrecognized channel URL format: {channel_url}")

# Call the Channels API to get channel info
time.sleep(5)
resp = requests.get(base_channels_url, params=params)
data = resp.json()
if not data.get("items"):
    raise ValueError(f"No channel data found for URL: {channel_url}")
channel_info = data["items"][0]

```

```

# Extract channel metadata
channel_id = channel_info.get("id")
snippet = channel_info.get("snippet", {})
stats = channel_info.get("statistics", {})
content = channel_info.get("contentDetails", {})

channel_name = snippet.get("title", "")
channel_description = snippet.get("description", "")
channel_join_date = snippet.get("publishedAt", "") # Channel creation date
subscriber_count = stats.get("subscriberCount")
if subscriber_count is not None:
    subscriber_count = int(subscriber_count)
uploads_playlist_id = None
if content.get("relatedPlaylists"):
    uploads_playlist_id = content["relatedPlaylists"].get("uploads")

return {
    "channel_id": channel_id,
    "channel_name": channel_name,
    "channel_description": channel_description,
    "channel_join_date": channel_join_date,
    "subscriber_count": subscriber_count,
    "uploads_playlist_id": uploads_playlist_id,
    "channel_url": channel_url
}

# Quick test of duration parser:
test_durations = ["PT1M37S", "PT2M", "PT45S", "PT1H2M3S", "PT98S"]
print("Duration parsing test:", {d: parse_iso8601_duration(d) for d in
test_durations})

```

Running the test above will confirm our `parse_iso8601_duration` function. For example, `"PT1M37S"` should parse to 97 seconds, and `"PT98S"` should parse to 98 seconds.

The `get_channel_data` function handles different channel URL patterns and uses `channels.list` with either `id`, `forUsername`, or `forHandle` as needed to get channel information. If a custom URL (`/c/Name`) is provided, it falls back to using the Search API to find the channel's ID. The function returns a dictionary containing the channel's metadata (name, description, join date, subscriber count) and the uploads playlist ID (which we'll need to list videos).

# Retrieve Channel and Video List

Now, using the helper functions, we will iterate through each channel URL, get the channel data, and then retrieve all videos from the channel's uploads playlist. We will store metadata for each video and filter by duration (< 98 seconds) as required.

We'll use the `playlistItems.list` API endpoint to fetch video IDs from the uploads playlist. We need to handle pagination if the channel has many videos (the API returns up to 50 items per page).

After collecting all video IDs (and basic info like title and published date), we will call the `videos.list` endpoint for those IDs to get detailed info (snippet, contentDetails, statistics) in batches of up to 50 IDs at a time (to stay within query parameter limits). This is more efficient than calling the videos API for each video individually.

Let's implement this step:

```
python

video_records = [] # to collect video data dictionaries for DataFrame

for url in channel_urls:
    try:
        channel_data = get_channel_data(API_KEY, url)
    except Exception as e:
        print(f"Error retrieving channel data for {url}: {e}")
        continue

    channel_name = channel_data["channel_name"]
    print(f"\nProcessing channel: {channel_name}")
    uploads_playlist_id = channel_data.get("uploads_playlist_id")
    if not uploads_playlist_id:
        print(f"No uploads playlist found for channel {channel_name}. Skipping.")
        continue

    # Retrieve all videos from the uploads playlist
    playlist_items_url = "https://www.googleapis.com/youtube/v3/playlistItems"
    playlist_params = {
        "part": "snippet,contentDetails",
        "playlistId": uploads_playlist_id,
        "maxResults": 50,
```

```

    "key": API_KEY
}

videos_in_playlist = []
while True:
    time.sleep(5)
    resp = requests.get(playlist_items_url, params=playlist_params)
    data = resp.json()
    if "error" in data:
        print(f"API error when fetching playlist items:
{data['error'].get('message')}")
        break
    items = data.get("items", [])
    for item in items:
        # snippet contains title, publishedAt; contentDetails contains videoId
        vid_id = item["contentDetails"]["videoId"]
        vid_title = item["snippet"].get("title", "")
        vid_published = item["snippet"].get("publishedAt", "")
        videos_in_playlist.append({
            "video_id": vid_id,
            "title": vid_title,
            "published_at": vid_published
        })
    # Pagination: check if there's another page
    if "nextPageToken" in data:
        playlist_params["pageToken"] = data["nextPageToken"]
    else:
        break

print(f"Found {len(videos_in_playlist)} videos in channel {channel_name}.")
```

if not videos\_in\_playlist:

continue

# Now retrieve detailed video info in batches of 50

video\_ids = [v["video\_id"] for v in videos\_in\_playlist]

for i in range(0, len(video\_ids), 50):

batch\_ids = video\_ids[i:i+50]

videos\_url = "https://www.googleapis.com/youtube/v3/videos"

videos\_params = {

"part": "snippet,contentDetails,statistics",

"id": ",".join(batch\_ids),

"key": API\_KEY

```

}

time.sleep(5)
resp = requests.get(videos_url, params=videos_params)
data = resp.json()
if "items" not in data:
    continue
for item in data["items"]:
    vid_id = item["id"]
    snippet = item.get("snippet", {})
    content_details = item.get("contentDetails", {})
    stats = item.get("statistics", {})

    # Video metadata fields
    title = snippet.get("title", "")
    description = snippet.get("description", "")
    published_at = snippet.get("publishedAt", "")
    tags = snippet.get("tags", [])
    duration_iso = content_details.get("duration", "")
    duration_seconds = parse_iso8601_duration(duration_iso) if duration_iso
else None
    view_count = int(stats.get("viewCount", 0))
    like_count = int(stats.get("likeCount", 0)) if "likeCount" in stats else
None
    # Dislike count may not be available (private as of Dec 2021]
    dislike_count = int(stats.get("dislikeCount", 0)) if "dislikeCount" in
stats else None
    comment_count = int(stats.get("commentCount", 0)) if "commentCount" in
stats else 0

    # Filter by video length < 98 seconds
    if duration_seconds is None or duration_seconds >= 98:
        continue # skip videos 98s or longer

video_url = f"https://www.youtube.com/watch?v={vid_id}"

# Prepare a record dict for this video
record = {
    "channel_name": channel_name,
    "channel_id": channel_data["channel_id"],
    "channel_url": channel_data["channel_url"],
    "channel_description": channel_data["channel_description"],
    "channel_join_date": channel_data["channel_join_date"],
}

```

```

    "subscriber_count": channel_data["subscriber_count"],
    "video_id": vid_id,
    "video_url": video_url,
    "title": title,
    "description": description,
    "published_at": published_at,
    "tags": ",".join(tags) if tags else "",
    "view_count": view_count,
    "like_count": like_count,
    "dislike_count": dislike_count,
    "comment_count": comment_count,
    "duration_seconds": duration_seconds
}
video_records.append(record)

```

In the code above:

- We loop through each channel URL, get the channel data (including the `uploads_playlist_id`).
- We use the **PlaylistItems API** to get all videos in the uploads playlist. We include `part=snippet,contentDetails` to get both the `videoId` and basic snippet info (like title and published date for logging). We accumulate all video IDs in `videos_in_playlist`, handling pagination with `nextPageToken`.
- We then use the **Videos API** to get detailed info for up to 50 video IDs at a time. We request `snippet,contentDetails,statistics` parts for each video. This gives us title, description, tags, duration (ISO 8601 format), and statistics (view count, like count, dislike count, comment count) for each video.
- We convert the duration to seconds using our parser and filter out any video that is 98 seconds or longer. Only videos with `duration_seconds < 98` pass through.
- We assemble a `record` dictionary for each qualifying video with all requested fields. The `dislike_count` is included but note that this field will likely be `None` for public data because YouTube made the dislike count private in 2021 (it would only be populated if we had owner-authenticated access). We still include the field for completeness.
- Each `record` is added to the `video_records` list for later use in the DataFrame.

We also print out some progress information, such as how many videos were found in a channel and when we start processing a channel.

## Fetch Comments for Each Video

Next, for each video record, we will retrieve the top-level comments using the **CommentThreads API**. We will fetch comments in plain text (`textFormat="plainText"`) to avoid HTML formatting. If a video has many comments, the script will iterate through pages of comments up to a certain limit to avoid excessive usage (you can adjust or remove the limit as needed).

We will store all retrieved comments (concatenated) in the `record["comments"]` field as a single string (comments separated by newline). We also ensure to respect the delay between each API call.

**Note:** The YouTube API returns comment text in two formats: `textDisplay` (formatted) and `textOriginal` (raw). The `textOriginal` is only returned to the comment's author or the channel owner, so for our API-key-only access, we rely on `textDisplay` with `textFormat=plainText`, which gives a plaintext version of the comment (though it may still have some minor alterations, like replacing URLs with titles).

Let's implement the comment retrieval:

```
python

# Add a field for comments text in each record
for record in video_records:
    video_id = record["video_id"]
    comment_texts = []
    # Use commentThreads.list to get top-level comments
    comments_url = "https://www.googleapis.com/youtube/v3/commentThreads"
    comments_params = {
        "part": "snippet",
        "videoId": video_id,
        "textFormat": "plainText",
        "maxResults": 50,
        "key": API_KEY
    }
    fetched_count = 0
    max_to_fetch = 100 # limit to 100 comments per video for this example (adjust
    as needed, or remove limit to get all)
    while True:
        time.sleep(5)
        resp = requests.get(comments_url, params=comments_params)
```

```

data = resp.json()
if "error" in data:
    # If comments are disabled or any error occurs, we break
    # (YouTube disables comments on some videos, like those for kids)
    # We store an indication if comments are disabled or unavailable.
    error_msg = data["error"].get("message", "Unknown error")
    print(f"Could not fetch comments for video {video_id}: {error_msg}")
    break

items = data.get("items", [])
for item in items:
    top_comment = item["snippet"]["topLevelComment"]
    comment_snippet = top_comment["snippet"]
    comment_text = comment_snippet.get("textDisplay", "")
    comment_texts.append(comment_text.strip())
    fetched_count += 1
    if fetched_count >= max_to_fetch:
        break
if fetched_count >= max_to_fetch:
    # Reached our limit of comments to fetch
    break
if "nextPageToken" in data:
    comments_params["pageToken"] = data["nextPageToken"]
else:
    break

# Join all comments into one string (separated by newlines)
if comment_texts:
    record["comments"] = "\n\n".join(comment_texts)
else:
    # No comments or none fetched
    record["comments"] = ""

```

In the comment fetching loop:

- We set a `max_to_fetch` limit of 100 comments per video for demonstration. You can increase this or remove the limit to fetch all comments, but be mindful of rate limits and performance if videos have thousands of comments.
- We call `commentThreads.list` for each video with `part=snippet` (which includes the `topLevelComment` and its snippet) and `textFormat=plainText` to get plain text comments. We use `maxResults=50` to fetch 50 comments per page.

- We accumulate comments in `comment_texts`. If `comments` are disabled or an error occurs (for example, YouTube may disable comments on certain videos, such as those marked for kids), we catch the error and break out, leaving the comments list empty or noting the issue.
- We paginate through comments using `nextPageToken` until we reach the end or our `max_to_fetch` limit.
- Finally, we join the comments into a single string separated by blank lines and store it in the `record["comments"]`. If no comments were fetched, we store an empty string.

## Download Videos

Now we will download each qualifying video (those with duration < 98 seconds) using the `pytube` library. We will download at the best available resolution up to 720p. The `pytube.YouTube` object provides a `streams` filter; we can use `streams.get_highest_resolution()` which returns the highest resolution progressive stream (this typically maxes out at 720p for combined audio/video stream).

We will create a folder named `downloads` (if it doesn't exist) to store the video files. Each video file will be named with its unique video ID (to avoid duplicates and for easy reference). We also note the resolution of the downloaded stream and update the record with `downloaded_resolution` and `download_status`.

```
python
```

```
# Create a downloads directory if it doesn't exist
download_dir = "downloads"
os.makedirs(download_dir, exist_ok=True)

from pytube import YouTube
for record in video_records:
    video_id = record["video_id"]
    video_url = record["video_url"]
    try:
        time.sleep(5)
        yt = YouTube(video_url)
        # Get the highest resolution stream that includes audio (progressive stream)
        stream = yt.streams.get_highest_resolution()
        if stream is None:
            # If for some reason no progressive stream is found, pick the highest

```

```

itag progressive manually

    stream =
yt.streams.filter(progressive=True).order_by('resolution').desc().first()

    if stream:
        # Download the video to the download directory with video_id as filename
        out_file = stream.download(output_path=download_dir, filename=f'{video_id}.mp4')

        record['downloaded_resolution'] = stream.resolution
        record['download_status'] = "Downloaded"

    else:
        record['downloaded_resolution'] = None
        record['download_status'] = "No progressive stream available"
        print(f"No downloadable stream for video {video_id}.")

except Exception as e:
    record['downloaded_resolution'] = None
    record['download_status'] = f"Failed: {e}"
    print(f"Failed to download {video_url}: {e}")

```

In this code:

- We ensure the `downloads` directory exists.
- For each video, we create a `YouTube` object and attempt to get the highest resolution progressive stream (which will include audio). We prefer the highest up to 720p; `get_highest_resolution()` will typically choose 720p or the highest available under it, since higher qualities often require separate audio stream】 .
- We then call `stream.download`, specifying the `output_path` and a `filename` as `<video_id>.mp4`. This will save the video file in the `downloads` folder.
- We record the `stream.resolution` (e.g. "720p" or "480p") in the `downloaded_resolution` field and note the `download_status` as "Downloaded".
- If no appropriate stream is found or an error occurs, we catch the exception and mark the status accordingly. For example, age-restricted or private videos might fail to download without additional steps. We handle such cases by recording the failure and printing a message.

**Important:** Downloading videos may be subject to YouTube's Terms of Service. Ensure you have permission to download and use the video content. This script is intended for educational or archival purposes where such usage is allowed.

## Create DataFrame and Save to CSV

Finally, we will create a pandas DataFrame from the collected `video_records` and save it as a CSV file. Each row of the CSV corresponds to one video and includes both channel-level information (repeated for each video of that channel) and video-specific information.

We will name the CSV file `youtube_videos_data.csv` (you can change this as needed). We will also display the first few rows of the DataFrame as a quick preview.

```
python
```

```
# Create DataFrame from the records
df = pd.DataFrame(video_records)

# Save to CSV (without index)
df.to_csv("youtube_videos_data.csv", index=False)

print(f"\nSaved data for {len(df)} videos to youtube_videos_data.csv")
df.head(3) # Display first 3 rows as a sample
```

This will produce a CSV file with columns such as:

- `channel_name`, `channel_id`, `channel_url`, `channel_description`,  
`channel_join_date`, `subscriber_count` (channel metadata)
- `video_id`, `video_url`, `title`, `description`, `published_at`, `tags`, `view_count`,  
`like_count`, `dislike_count`, `comment_count`, `duration_seconds` (video metadata)
- `comments` (concatenated top-level comments text)
- `downloaded_resolution`, `download_status` (download information)

You can open the CSV file to analyze the data or import it into other tools. The DataFrame preview (`head`) shows a glimpse of the structured data.

## Notes and Next Steps

- **API Quota Considerations:** Each API call used (`channels.list`, `playlistItems.list`, `videos.list`, `commentThreads.list`) costs quota units (mostly 1 unit each for the types we used). Fetching many videos and comments can use a significant number of quota units if the channel or videos are large. Ensure your quota is sufficient or consider reducing the scope (e.g., limit the number of videos or comments).
- **Rate Limiting:** We added `time.sleep(5)` before each API call and download to throttle the requests. Adjust the delay as needed to comply with any usage guidelines and avoid hitting rate limits.
- **Extensibility:** This script can be extended to handle additional video or channel properties if needed (for example, adding category IDs, video thumbnails, etc., which are available from the API). For extremely large channels, consider using YouTube's pagination and quota strategically (or the YouTube Data API's client library for better efficiency and built-in retry handling).
- **Environment:** The script is meant to run in a standard Python environment (or Jupyter notebook). Make sure to install the required libraries. The `pytube` library can occasionally break due to changes in YouTube's site; ensure you have a relatively recent version of pytube, and consider updating it if you encounter download issues.

By following this notebook, you should be able to gather a comprehensive dataset of short videos from specified YouTube channels, complete with metadata and downloaded files, all structured in a convenient CSV output for further analysis or processing.

 Sources