
pytube Documentation

Release 12.1.3

Nick Ficano

Mar 27, 2023

Contents

1	Features	3
2	The User Guide	5
2.1	Installation of pytube	5
2.2	Quickstart	6
2.3	Working with Streams and StreamQuery	7
2.4	Filtering Streams	7
2.5	Downloading Streams	8
2.6	Subtitle/Caption Tracks	9
2.7	Using Playlists	10
2.8	Using Channels	10
2.9	Using the search feature	11
2.10	Additional functionality	12
2.11	Command-line interface (CLI)	12
2.12	Exception handling	13
3	The API Documentation	15
3.1	API	15
4	Indices and tables	39
	Python Module Index	41
	Index	43

Release v12.1.3. (*Installation*)

pytube is a lightweight, Pythonic, dependency-free, library (and command-line utility) for downloading YouTube Videos.

Behold, a perfect balance of simplicity versus flexibility:

```
>>> from pytube import YouTube
>>> YouTube('https://youtu.be/9bZkp7q19f0').streams.first().download()
>>> yt = YouTube('http://youtube.com/watch?v=9bZkp7q19f0')
>>> yt.streams
... .filter(progressive=True, file_extension='mp4')
... .order_by('resolution')
... .desc()
... .first()
... .download()
```


CHAPTER 1

Features

- Support for Both Progressive & DASH Streams
- Easily Register `on_download_progress` & `on_download_complete` callbacks
- Command-line Interfaced Included
- Caption Track Support
- Outputs Caption Tracks to .srt format (SubRip Subtitle)
- Ability to Capture Thumbnail URL.
- Extensively Documented Source Code
- No Third-Party Dependencies

This part of the documentation begins with some background information about the project, then focuses on step-by-step instructions for getting the most out of pytube.

2.1 Installation of pytube

This guide assumes you already have python and pip installed.

To install pytube, run the following command in your terminal:

```
$ pip install pytube
```

2.1.1 Get the Source Code

pytube is actively developed on GitHub, where the source is [available](#).

You can either clone the public repository:

```
$ git clone git://github.com/pytube/pytube.git
```

Or, download the [tarball](#):

```
$ curl -OL https://github.com/pytube/pytube/tarball/master  
# optionally, zipball is also available (for Windows users).
```

Once you have a copy of the source, you can embed it in your Python package, or install it into your site-packages by running:

```
$ cd pytube  
$ python -m pip install .
```

2.2 Quickstart

This guide will walk you through the basic usage of pytube.

Let's get started with some examples.

2.2.1 Downloading a Video

Downloading a video from YouTube with pytube is incredibly easy.

Begin by importing the YouTube class:

```
>>> from pytube import YouTube
```

Now, let's try to download a video. For this example, let's take something like the YouTube Rewind video for 2019:

```
>>> yt = YouTube('http://youtube.com/watch?v=2lAe1cqCOXo')
```

Now, we have a *YouTube* object called *yt*.

The pytube API makes all information intuitive to access. For example, this is how you would get the video's title:

```
>>> yt.title
YouTube Rewind 2019: For the Record | #YouTubeRewind
```

And this would be how you would get the thumbnail url:

```
>>> yt.thumbnail_url
'https://i.ytimg.com/vi/2lAe1cqCOXo/maxresdefault.jpg'
```

Neat, right? For advanced use cases, you can provide some additional arguments when you create a YouTube object:

```
>>> yt = YouTube(
    'http://youtube.com/watch?v=2lAe1cqCOXo',
    on_progress_callback=progress_func,
    on_complete_callback=complete_func,
    proxies=my_proxies,
    use_oauth=False,
    allow_oauth_cache=True
)
```

When instantiating a YouTube object, these named arguments can be passed in to improve functionality.

The `on_progress_callback` function will run whenever a chunk is downloaded from a video, and is called with three arguments: the stream, the data chunk, and the bytes remaining in the video. This could be used, for example, to display a progress bar.

The `on_complete_callback` function will run after a video has been fully downloaded, and is called with two arguments: the stream and the file path. This could be used, for example, to perform post-download processing on a video like trimming the length of it.

The `use_oauth` and `allow_oauth_cache` flags allow you to authorize pytube to interact with YouTube using your account, and can be used to bypass age restrictions or access private videos and playlists. If `allow_oauth_cache` is set to `True`, you should only be prompted to do so once, after which point pytube will cache the tokens it needs to act on your behalf. Otherwise, you will be prompted again for each action that requires you to be authenticated.

Once you have a YouTube object set up, you're ready to start looking at different media streams for the video, which is discussed in the next section.

2.3 Working with Streams and StreamQuery

The next section will explore the various options available for working with media streams, but before we can dive in, we need to review a new-ish streaming technique adopted by YouTube. It assumes that you have already created a YouTube object in your code called “yt”.

2.3.1 DASH vs Progressive Streams

Begin by running the following to list all streams:

```
>>> yt.streams
[<Stream: itag="18" mime_type="video/mp4" res="360p" fps="30fps" vcodec="avc1.42001E"
↳acodec="mp4a.40.2" progressive="True" type="video">,
<Stream: itag="22" mime_type="video/mp4" res="720p" fps="30fps" vcodec="avc1.64001F"
↳acodec="mp4a.40.2" progressive="True" type="video">,
<Stream: itag="137" mime_type="video/mp4" res="1080p" fps="30fps" vcodec="avc1.640028
↳" progressive="False" type="video">,
...
<Stream: itag="250" mime_type="audio/webm" abr="70kbps" acodec="opus" progressive=
↳"False" type="audio">,
<Stream: itag="251" mime_type="audio/webm" abr="160kbps" acodec="opus" progressive=
↳"False" type="audio">]
```

You may notice that some streams listed have both a video codec and audio codec, while others have just video or just audio, this is a result of YouTube supporting a streaming technique called Dynamic Adaptive Streaming over HTTP (DASH).

In the context of pytube, the implications are for the highest quality streams; you now need to download both the audio and video tracks and then post-process them with software like FFmpeg to merge them.

The legacy streams that contain the audio and video in a single file (referred to as “progressive download”) are still available, but only for resolutions 720p and below.

2.4 Filtering Streams

Pytube has built-in functionality to filter the streams available in a YouTube object with the `.filter()` method. You can pass it a number of different keyword arguments, so let’s review some of the different options you’re most likely to use. For a complete list of available properties to filter on, you can view the API documentation here: `pytube.StreamQuery.filter()`.

2.4.1 Filtering by streaming method

As mentioned before, progressive streams have the video and audio in a single file, but typically do not provide the highest quality media; meanwhile, adaptive streams split the video and audio tracks but can provide much higher quality. Pytube makes it easy to filter based on the type of stream that you’re interested.

For example, you can filter to only progressive streams with the following:

```
>>> yt.streams.filter(progressive=True)
[<Stream: itag="18" mime_type="video/mp4" res="360p" fps="30fps" vcodec="avc1.42001E"
↳acodec="mp4a.40.2" progressive="True" type="video">,
<Stream: itag="22" mime_type="video/mp4" res="720p" fps="30fps" vcodec="avc1.64001F"
↳acodec="mp4a.40.2" progressive="True" type="video">]
```

Conversely, if you only want to see the DASH streams (also referred to as “adaptive”) you can do:

```
>>> yt.streams.filter(adaptive=True)
[<Stream: itag="137" mime_type="video/mp4" res="1080p" fps="30fps" vcodec="avc1.640028"
↳ progressive="False" type="video">,
<Stream: itag="248" mime_type="video/webm" res="1080p" fps="30fps" vcodec="vp9"
↳ progressive="False" type="video">,
<Stream: itag="399" mime_type="video/mp4" res="None" fps="30fps" vcodec="av01.0.08M.08"
↳ progressive="False" type="video">,
...
<Stream: itag="250" mime_type="audio/webm" abr="70kbps" acodec="opus" progressive=
↳ "False" type="audio">,
<Stream: itag="251" mime_type="audio/webm" abr="160kbps" acodec="opus" progressive=
↳ "False" type="audio">]
```

2.4.2 Filtering for audio-only streams

To query the streams that contain only the audio track:

```
>>> yt.streams.filter(only_audio=True)
[<Stream: itag="140" mime_type="audio/mp4" abr="128kbps" acodec="mp4a.40.2"
↳ progressive="False" type="audio">,
<Stream: itag="249" mime_type="audio/webm" abr="50kbps" acodec="opus" progressive=
↳ "False" type="audio">,
<Stream: itag="250" mime_type="audio/webm" abr="70kbps" acodec="opus" progressive=
↳ "False" type="audio">,
<Stream: itag="251" mime_type="audio/webm" abr="160kbps" acodec="opus" progressive=
↳ "False" type="audio">]
```

2.4.3 Filtering for MP4 streams

To query only streams in the MP4 format:

```
>>> yt.streams.filter(file_extension='mp4')
[<Stream: itag="18" mime_type="video/mp4" res="360p" fps="30fps" vcodec="avc1.42001E"
↳ acodec="mp4a.40.2" progressive="True" type="video">,
<Stream: itag="22" mime_type="video/mp4" res="720p" fps="30fps" vcodec="avc1.64001F"
↳ acodec="mp4a.40.2" progressive="True" type="video">,
<Stream: itag="137" mime_type="video/mp4" res="1080p" fps="30fps" vcodec="avc1.640028"
↳ progressive="False" type="video">,
...
<Stream: itag="394" mime_type="video/mp4" res="None" fps="30fps" vcodec="av01.0.00M.08"
↳ progressive="False" type="video">,
<Stream: itag="140" mime_type="audio/mp4" abr="128kbps" acodec="mp4a.40.2"
↳ progressive="False" type="audio">]
```

2.5 Downloading Streams

After you’ve selected the *Stream* you’re interested, you’re ready to interact with it. At this point, you can query information about the stream, such as its filesize, whether the stream is adaptive, and more. You can also use the download method to save the file:

```
>>> stream = yt.streams.get_by_itag(22)
>>> stream.download()
```

The download method has a number of different useful arguments, which are documented in the API reference here: [pytube.Stream.download\(\)](#).

2.6 Subtitle/Caption Tracks

Pytube exposes the caption tracks in much the same way as querying the media streams. Let's begin by switching to a video that contains them:

```
>>> yt = YouTube('http://youtube.com/watch?v=2lAelcqCOXo')
>>> yt.captions
{'ar': <Caption lang="Arabic" code="ar">, 'zh-HK': <Caption lang="Chinese (Hong Kong)"
code="zh-HK">, 'zh-TW': <Caption lang="Chinese (Taiwan)" code="zh-TW">, 'hr':
<Caption lang="Croatian" code="hr">, 'cs': <Caption lang="Czech" code="cs">, 'da':
<Caption lang="Danish" code="da">, 'nl': <Caption lang="Dutch" code="nl">, 'en':
<Caption lang="English" code="en">, 'en-GB': <Caption lang="English (United Kingdom)"
code="en-GB">, 'et': <Caption lang="Estonian" code="et">, 'fil': <Caption lang="Filipino"
code="fil">, 'fi': <Caption lang="Finnish" code="fi">, 'fr-CA': <Caption
lang="French (Canada)" code="fr-CA">, 'fr-FR': <Caption lang="French (France)" code="fr-FR">,
'de': <Caption lang="German" code="de">, 'el': <Caption lang="Greek" code="el">,
'iw': <Caption lang="Hebrew" code="iw">, 'hu': <Caption lang="Hungarian"
code="hu">, 'id': <Caption lang="Indonesian" code="id">, 'it': <Caption lang="Italian"
code="it">, 'ja': <Caption lang="Japanese" code="ja">, 'ko': <Caption
lang="Korean" code="ko">, 'lv': <Caption lang="Latvian" code="lv">, 'lt': <Caption
lang="Lithuanian" code="lt">, 'ms': <Caption lang="Malay" code="ms">, 'no':
<Caption lang="Norwegian" code="no">, 'pl': <Caption lang="Polish" code="pl">, 'pt-BR':
<Caption lang="Portuguese (Brazil)" code="pt-BR">, 'pt-PT': <Caption lang="Portuguese (Portugal)"
code="pt-PT">, 'ro': <Caption lang="Romanian" code="ro">, 'ru': <Caption lang="Russian"
code="ru">, 'sk': <Caption lang="Slovak" code="sk">, 'es-419': <Caption lang="Spanish (Latin America)"
code="es-419">, 'es-ES': <Caption lang="Spanish (Spain)" code="es-ES">, 'sv': <Caption lang="Swedish"
code="sv">, 'th': <Caption lang="Thai" code="th">, 'tr': <Caption lang="Turkish" code="tr">, 'uk':
<Caption lang="Ukrainian" code="uk">, 'ur': <Caption lang="Urdu" code="ur">, 'vi':
<Caption lang="Vietnamese" code="vi">}
```

Now let's checkout the english captions:

```
>>> caption = yt.captions.get_by_language_code('en')
```

Great, now let's see how YouTube formats them:

```
>>> caption.xml_captions
'<?xml version="1.0" encoding="utf-8" ?><transcript><text start="10.2" dur="0.94">K-
pop!</text>...'
```

Oh, this isn't very easy to work with, let's convert them to the srt format:

```
>>> print(caption.generate_srt_captions())
1
00:00:10,200 --> 00:00:11,140
K-pop!

2
```

(continues on next page)

(continued from previous page)

```
00:00:13,400 --> 00:00:16,200
That is so awkward to watch.
...
```

2.7 Using Playlists

This guide will walk you through the basics of working with pytube Playlists.

2.7.1 Creating a Playlist

Using pytube to interact with playlists is very simple. Begin by importing the Playlist class:

```
>>> from pytube import Playlist
```

Now let's create a playlist object. You can do this by initializing the object with a playlist URL:

```
>>> p = Playlist('https://www.youtube.com/playlist?list=PLS1QulWo1RIaJECMeUT4LFwJ-
↳ ghgoSH6n')
```

Or you can create one from a video link in a playlist:

```
>>> p = Playlist('https://www.youtube.com/watch?v=41qgdwd3zAg&
↳ list=PLS1QulWo1RIaJECMeUT4LFwJ-ghgoSH6n')
```

Now, we have a Playlist object called `p` that we can do some work with.

2.7.2 Interacting with a playlist

Fundamentally, a Playlist object is just a container for YouTube objects.

If, for example, we wanted to download all of the videos in a playlist, we would do the following:

```
>>> print(f'Downloading: {p.title}')
Downloading: Python Tutorial for Beginners (For Absolute Beginners)
>>> for video in p.videos:
>>>     video.streams.first().download()
```

Or, if we're only interested in the URLs for the videos, we can look at those as well:

```
>>> for url in p.video_urls[:3]:
>>>     print(url)
['https://www.youtube.com/watch?v=41qgdwd3zAg',
'https://www.youtube.com/watch?v=Lbs7vmx3YwU',
'https://www.youtube.com/watch?v=YtX-Rmoea0M']
```

And that's basically all there is to it!

2.8 Using Channels

This guide will walk you through the basics of working with pytube Channels.

2.8.1 Creating a Channel

Using pytube to interact with channels is similar to interacting with playlists. Begin by importing the Channel class:

```
>>> from pytube import Channel
```

Now let's create a channel object. You can do this by initializing the object with a channel URL:

```
>>> c = Channel('https://www.youtube.com/c/ProgrammingKnowledge')
```

Or you can create one from a link to the channel's video page:

```
>>> c = Channel('https://www.youtube.com/c/ProgrammingKnowledge/videos')
```

Now, we have a Channel object called `c` that we can do some work with.

2.8.2 Interacting with a channel

Fundamentally, a Channel object is just a container for YouTube objects.

If, for example, we wanted to download all of the videos created by a channel, we would do the following:

```
>>> print(f'Downloading videos by: {c.channel_name}')
Downloading videos by: ProgrammingKnowledge
>>> for video in c.videos:
>>>     video.streams.first().download()
```

Or, if we're only interested in the URLs for the videos, we can look at those as well:

```
>>> for url in c.video_urls[:3]:
>>>     print(url)
['https://www.youtube.com/watch?v=tMqMU1U2MCU',
 'https://www.youtube.com/watch?v=YBfInrtWq8Y',
 'https://www.youtube.com/watch?v=EP9WrMw6Gzg']
```

And that's basically all there is to it!

2.9 Using the search feature

Pytube includes functionality to search YouTube and return results almost identical to those you would find using the search bar on YouTube's website. The integration into pytube means that we can directly provide you with YouTube objects that can be inspected and downloaded, instead of needing to do additional processing.

Using the Search object is really easy:

```
>>> from pytube import Search
>>> s = Search('YouTube Rewind')
>>> len(s.results)
17
>>> s.results
[ \
  <pytube.__main__.YouTube object: videoId=YbJOTdZBX1g>, \
  <pytube.__main__.YouTube object: videoId=PKtnafFtfEo>, \
  ... \
```

(continues on next page)

(continued from previous page)

```
]
>>>
```

Due to the potential for an endless stream of results, and in order to prevent a user from accidentally entering an infinite loop of requesting additional results, the `.results` attribute will only ever request the first set of search results. Additional results can be explicitly requested by using the `.get_next_results()` method, which will append any additional results to the `.results` attribute:

```
>>> s.get_next_results()
>>> len(s.results)
34
>>>
```

2.10 Additional functionality

In addition to the basic search functionality which returns YouTube objects, searches also have associated autocomplete suggestions. These can be accessed as follows:

```
>>> s.completion_suggestions
[\\
  'can this video get 1 million dislikes', \\
  'youtube rewind 2020 musical', \\
  ...\\
]
```

2.11 Command-line interface (CLI)

Pytube also ships with a tiny CLI for interacting with videos and playlists.

To download the highest resolution progressive stream:

```
$ pytube https://www.youtube.com/watch?v=2lAe1cqCOXo
```

To view available streams:

```
$ pytube https://www.youtube.com/watch?v=2lAe1cqCOXo --list
```

To download a specific stream, use the itag

```
$ pytube https://www.youtube.com/watch?v=2lAe1cqCOXo --itag=22
```

To get a list of all subtitles (caption codes)

```
$ pytube https://www.youtube.com/watch?v=2lAe1cqCOXo --list-captions
```

To download a specific subtitle (caption code) - in this case the English subtitles (in srt format) - use:

```
$ pytube https://www.youtube.com/watch?v=2lAe1cqCOXo -c en
```

It is also possible to just download the audio stream (default AAC/mp4):


```
$ pytube https://www.youtube.com/watch?v=2lAe1cqCOXo -a
```

To list all command line options, simply type

```
$ pytube --help
```

Finally, if you're filing a bug report, the cli contains a switch called `--build-playback-report`, which bundles up the state, allowing others to easily replay your issue.

2.12 Exception handling

Pytube implements a number of useful exceptions for handling program flow. There are a number of cases where pytube simply cannot access videos on YouTube and relies on the user to handle these exceptions. Generally speaking, if a video is unaccessible for any reason, this can be caught with the generic `VideoUnavailable` exception. This could be used, for example, to skip private videos in a playlist, videos that are region-restricted, and more.

Let's see what your code might look like if you need to do exception handling:

```
>>> from pytube import Playlist, YouTube
>>> playlist_url = 'https://youtube.com/playlist?list=special_playlist_id'
>>> p = Playlist(playlist_url)
>>> for url in p.video_urls:
...     try:
...         yt = YouTube(url)
...     except VideoUnavailable:
...         print(f'Video {url} is unavaialable, skipping.')
...     else:
...         print(f'Downloading video: {url}')
...         yt.streams.first().download()
```

This will automatically skip over videos that could not be downloaded due to a limitation with the pytube library. You can find more details about what specific exceptions can be handled here: [`pytube.exceptions`](#).

The API Documentation

If you are looking for information on a specific function, class, or method, this part of the documentation is for you.

3.1 API

3.1.1 YouTube Object

```
class pytube.YouTube(url: str, on_progress_callback: Optional[Callable[[Any, bytes, int], None]] =  
                    None, on_complete_callback: Optional[Callable[[Any, Optional[str]], None]]  
                    = None, proxies: Dict[str, str] = None, use_oauth: bool = False, al-  
                    low_oauth_cache: bool = True)
```

Core developer interface for pytube.

author

Get the video author. :rtype: str

bypass_age_gate()

Attempt to update the vid_info by bypassing the age gate.

caption_tracks

Get a list of *Caption*.

Return type List[*Caption*]

captions

Interface to query caption tracks.

Return type CaptionQuery.

channel_id

Get the video poster's channel id.

Return type str

channel_url

Construct the channel url for the video's poster from the channel id.

Return type `str`

check_availability()

Check whether the video is available.

Raises different exceptions based on why the video is unavailable, otherwise does nothing.

description

Get the video description.

Return type `str`

fmt_streams

Returns a list of streams if they have been initialized.

If the streams have not been initialized, finds all relevant streams and initializes them.

static from_id(*video_id: str*) → `pytube.__main__.YouTube`

Construct a `YouTube` object from a video id.

Parameters **video_id** (*str*) – The video id of the YouTube video.

Return type `YouTube`

keywords

Get the video keywords.

Return type `List[str]`

length

Get the video length in seconds.

Return type `int`

metadata

Get the metadata for the video.

Return type `YouTubeMetadata`

publish_date

Get the publish date.

Return type `datetime`

rating

Get the video average rating.

Return type `float`

register_on_complete_callback (*func: Callable[[Any, Optional[str]], None]*)

Register a download complete callback function post initialization.

Parameters **func** (*callable*) – A callback function that takes `stream` and `file_path`.

Return type `None`

register_on_progress_callback (*func: Callable[[Any, bytes, int], None]*)

Register a download progress callback function post initialization.

Parameters **func** (*callable*) –

A callback function that takes **stream**, **chunk**, and **bytes_remaining** as parameters.

Return type `None`

streaming_data
Return streamingData from video info.

streams
Interface to query both adaptive (DASH) and progressive streams.
Return type `StreamQuery`.

thumbnail_url
Get the thumbnail url image.
Return type `str`

title
Get the video title.
Return type `str`

vid_info
Parse the raw vid info and return the parsed result.
Return type `Dict[Any, Any]`

views
Get the number of the times the video has been viewed.
Return type `int`

3.1.2 Playlist Object

class `pytube.contrib.playlist.Playlist` (*url: str, proxies: Optional[Dict[str, str]] = None*)
Load a YouTube playlist with URL

count (*value*) → integer – return number of occurrences of value

html
Get the playlist page html.
Return type `str`

index (*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises `ValueError` if the value is not present.
Supporting start and stop arguments is optional, but recommended.

initial_data
Extract the initial data from the playlist page html.
Return type `dict`

last_updated
Extract the date that the playlist was last updated.

For some playlists, this will be a specific date, which is returned as a datetime object. For other playlists, this is an estimate such as “1 week ago”. Due to the fact that this value is returned as a string, pytube does a best-effort parsing where possible, and returns the raw string where it is not possible.
Returns Date of last playlist update where possible, else the string provided
Return type `datetime.date`

length
Extract the number of videos in the playlist.
Returns Playlist video count

Return type `int`

owner

Extract the owner of the playlist.

Returns Playlist owner name.

Return type `str`

owner_id

Extract the channel_id of the owner of the playlist.

Returns Playlist owner's channel ID.

Return type `str`

owner_url

Create the channel url of the owner of the playlist.

Returns Playlist owner's channel url.

Return type `str`

playlist_id

Get the playlist id.

Return type `str`

playlist_url

Get the base playlist url.

Return type `str`

sidebar_info

Extract the sidebar info from the playlist page html.

Return type `dict`

title

Extract playlist title

Returns playlist title (name)

Return type `Optional[str]`

trimmed (*video_id: str*) → `Iterable[str]`

Retrieve a list of YouTube video URLs trimmed at the given video ID

i.e. if the playlist has video IDs 1,2,3,4 calling `trimmed(3)` returns `[1,2]` :type video_id: str

video ID to trim the returned list of playlist URLs at

Return type `List[str]`

Returns List of video URLs from the playlist trimmed at the given ID

url_generator ()

Generator that yields video URLs.

Yields Video URLs

video_urls

Complete links of all the videos in playlist

Return type `List[str]`

Returns List of video URLs

videos

Yields YouTube objects of videos in this playlist

Return type List[*YouTube*]

Returns List of YouTube

views

Extract view count for playlist.

Returns Playlist view count

Return type int

yt_api_key

Extract the INNERTUBE_API_KEY from the playlist ytcfg.

Return type str

ytcfg

Extract the ytcfg from the playlist page html.

Return type dict

3.1.3 Channel Object

class pytube.contrib.channel.**Channel** (*url: str, proxies: Optional[Dict[str, str]] = None*)

about_html

Get the html for the /about page.

Currently unused for any functionality.

Return type str

channel_id

Get the ID of the YouTube channel.

This will return the underlying ID, not the vanity URL.

Return type str

channel_name

Get the name of the YouTube channel.

Return type str

community_html

Get the html for the /community page.

Currently unused for any functionality.

Return type str

count (*value*) → integer – return number of occurrences of value

featured_channels_html

Get the html for the /channels page.

Currently unused for any functionality.

Return type str

html

Get the html for the /videos page.

Return type `str`

index (*value*`[, start[, stop]]`) \rightarrow integer – return first index of value.
Raises `ValueError` if the value is not present.

Supporting start and stop arguments is optional, but recommended.

initial_data

Extract the initial data from the playlist page html.

Return type `dict`

last_updated

Extract the date that the playlist was last updated.

For some playlists, this will be a specific date, which is returned as a datetime object. For other playlists, this is an estimate such as “1 week ago”. Due to the fact that this value is returned as a string, pytube does a best-effort parsing where possible, and returns the raw string where it is not possible.

Returns Date of last playlist update where possible, else the string provided

Return type `datetime.date`

length

Extract the number of videos in the playlist.

Returns Playlist video count

Return type `int`

owner

Extract the owner of the playlist.

Returns Playlist owner name.

Return type `str`

owner_id

Extract the `channel_id` of the owner of the playlist.

Returns Playlist owner’s channel ID.

Return type `str`

owner_url

Create the channel url of the owner of the playlist.

Returns Playlist owner’s channel url.

Return type `str`

playlist_id

Get the playlist id.

Return type `str`

playlist_url

Get the base playlist url.

Return type `str`

playlists_html

Get the html for the `/playlists` page.

Currently unused for any functionality.

Return type `str`

sidebar_info

Extract the sidebar info from the playlist page html.

Return type `dict`

title

Extract playlist title

Returns playlist title (name)

Return type `Optional[str]`

trimmed (*video_id: str*) → `Iterable[str]`

Retrieve a list of YouTube video URLs trimmed at the given video ID

i.e. if the playlist has video IDs 1,2,3,4 calling `trimmed(3)` returns `[1,2]` :type video_id: str

video ID to trim the returned list of playlist URLs at

Return type `List[str]`

Returns List of video URLs from the playlist trimmed at the given ID

url_generator ()

Generator that yields video URLs.

Yields Video URLs

vanity_url

Get the vanity URL of the YouTube channel.

Returns None if it doesn't exist.

Return type `str`

video_urls

Complete links of all the videos in playlist

Return type `List[str]`

Returns List of video URLs

videos

Yields YouTube objects of videos in this playlist

Return type `List[YouTube]`

Returns List of YouTube

views

Extract view count for playlist.

Returns Playlist view count

Return type `int`

yt_api_key

Extract the INNERTUBE_API_KEY from the playlist ytcfg.

Return type `str`

ytcfg

Extract the ytcfg from the playlist page html.

Return type `dict`

3.1.4 Stream Object

class `pytube.Stream` (*stream: Dict[KT, VT], monostate: pytube.monostate.Monostate*)

Container for stream manifest data.

default_filename

Generate filename based on the video title.

Return type `str`

Returns An os file system compatible filename.

download (*output_path: Optional[str] = None, filename: Optional[str] = None, filename_prefix: Optional[str] = None, skip_existing: bool = True, timeout: Optional[int] = None, max_retries: Optional[int] = 0*) \rightarrow `str`

Write the media stream to disk.

Parameters

- **output_path** (*str or None*) – (optional) Output path for writing media file. If one is not specified, defaults to the current working directory.
- **filename** (*str or None*) – (optional) Output filename (stem only) for writing media file. If one is not specified, the default filename is used.
- **filename_prefix** (*str or None*) – (optional) A string that will be prepended to the filename. For example a number in a playlist or the name of a series. If one is not specified, nothing will be prepended This is separate from filename so you can use the default filename but still add a prefix.
- **skip_existing** (*bool*) – (optional) Skip existing files, defaults to True
- **timeout** (*int*) – (optional) Request timeout length in seconds. Uses system default.
- **max_retries** (*int*) – (optional) Number of retries to attempt after socket timeout. Defaults to 0.

Returns Path to the saved video

Return type `str`

filesize

File size of the media stream in bytes.

Return type `int`

Returns Filesize (in bytes) of the stream.

filesize_approx

Get approximate filesize of the video

Falls back to HTTP call if there is not sufficient information to approximate

Return type `int`

Returns size of video in bytes

filesize_gb

File size of the media stream in gigabytes.

Return type `float`

Returns Rounded filesize (in gigabytes) of the stream.

filesize_kb

File size of the media stream in kilobytes.

Return type `float`

Returns Rounded filesize (in kilobytes) of the stream.

filesize_mb

File size of the media stream in megabytes.

Return type `float`

Returns Rounded filesize (in megabytes) of the stream.

includes_audio_track

Whether the stream only contains audio.

Return type `bool`

includes_video_track

Whether the stream only contains video.

Return type `bool`

is_adaptive

Whether the stream is DASH.

Return type `bool`

is_progressive

Whether the stream is progressive.

Return type `bool`

on_complete (*file_path: Optional[str]*)

On download complete handler function.

Parameters **file_path** (*str*) – The file handle where the media is being written to.

Return type `None`

on_progress (*chunk: bytes, file_handler: BinaryIO, bytes_remaining: int*)

On progress callback function.

This function writes the binary data to the file, then checks if an additional callback is defined in the monostate. This is exposed to allow things like displaying a progress bar.

Parameters

- **chunk** (*bytes*) – Segment of media file binary data, not yet written to disk.
- **file_handler** (*io.BufferedReader*) – The file handle where the media is being written to.
- **bytes_remaining** (*int*) – The delta between the total file size in bytes and amount already downloaded.

Return type `None`

parse_codecs () → `Tuple[Optional[str], Optional[str]]`

Get the video/audio codecs from list of codecs.

Parse a variable length sized list of codecs and returns a constant two element tuple, with the video codec as the first element and audio as the second. Returns None if one is not available (adaptive only).

Return type `tuple`

Returns A two element tuple with audio and video codecs.

stream_to_buffer (*buffer: BinaryIO*) → None

Write the media stream to buffer

Return type io.BytesIO buffer

title

Get title of video

Return type str

Returns Youtube video title

3.1.5 StreamQuery Object

class pytube.query.StreamQuery (*fmt_streams*)

Interface for querying the available media streams.

all () → List[pytube.streams.Stream]

Get all the results represented by this query as a list.

Return type list

asc () → pytube.query.StreamQuery

Sort streams in ascending order.

Return type StreamQuery

count (*value: Optional[str] = None*) → int

Get the count of items in the list.

Return type int

desc () → pytube.query.StreamQuery

Sort streams in descending order.

Return type StreamQuery

filter (*fps=None, res=None, resolution=None, mime_type=None, type=None, subtype=None, file_extension=None, abr=None, bitrate=None, video_codec=None, audio_codec=None, only_audio=None, only_video=None, progressive=None, adaptive=None, is_dash=None, custom_filter_functions=None*)

Apply the given filtering criterion.

Parameters

- **fps** (*int or None*) – (optional) The frames per second.
- **resolution** (*str or None*) – (optional) Alias to *res*.
- **res** (*str or None*) – (optional) The video resolution.
- **mime_type** (*str or None*) – (optional) Two-part identifier for file formats and format contents composed of a “type”, a “subtype”.
- **type** (*str or None*) – (optional) Type part of the *mime_type* (e.g.: audio, video).
- **subtype** (*str or None*) – (optional) Sub-type part of the *mime_type* (e.g.: mp4, mov).
- **file_extension** (*str or None*) – (optional) Alias to *sub_type*.
- **abr** (*str or None*) – (optional) Average bitrate (ABR) refers to the average amount of data transferred per unit of time (e.g.: 64kbps, 192kbps).
- **bitrate** (*str or None*) – (optional) Alias to *abr*.

- **video_codec** (*str or None*) – (optional) Video compression format.
- **audio_codec** (*str or None*) – (optional) Audio compression format.
- **progressive** (*bool*) – Excludes adaptive streams (one file contains both audio and video tracks).
- **adaptive** (*bool*) – Excludes progressive streams (audio and video are on separate tracks).
- **is_dash** (*bool*) – Include/exclude dash streams.
- **only_audio** (*bool*) – Excludes streams with video tracks.
- **only_video** (*bool*) – Excludes streams with audio tracks.
- **custom_filter_functions** (*list or None*) – (optional) Interface for defining complex filters without subclassing.

first () → Optional[pytube.streams.Stream]

Get the first Stream in the results.

Return type Stream or None

Returns the first result of this query or None if the result doesn't contain any streams.

get_audio_only (*subtype: str = 'mp4'*) → Optional[pytube.streams.Stream]

Get highest bitrate audio stream for given codec (defaults to mp4)

Parameters **subtype** (*str*) – Audio subtype, defaults to mp4

Return type Stream or None

Returns The Stream matching the given itag or None if not found.

get_by_itag (*itag: int*) → Optional[pytube.streams.Stream]

Get the corresponding Stream for a given itag.

Parameters **itag** (*int*) – YouTube format identifier code.

Return type Stream or None

Returns The Stream matching the given itag or None if not found.

get_by_resolution (*resolution: str*) → Optional[pytube.streams.Stream]

Get the corresponding Stream for a given resolution.

Stream must be a progressive mp4.

Parameters **resolution** (*str*) – Video resolution i.e. “720p”, “480p”, “360p”, “240p”, “144p”

Return type Stream or None

Returns The Stream matching the given itag or None if not found.

get_highest_resolution () → Optional[pytube.streams.Stream]

Get highest resolution stream that is a progressive video.

Return type Stream or None

Returns The Stream matching the given itag or None if not found.

get_lowest_resolution () → Optional[pytube.streams.Stream]

Get lowest resolution stream that is a progressive mp4.

Return type Stream or None

Returns The `Stream` matching the given itag or `None` if not found.

index (*value*`[, start[, stop]]`) → integer – return first index of value.
Raises `ValueError` if the value is not present.

Supporting start and stop arguments is optional, but recommended.

last ()

Get the last `Stream` in the results.

Return type `Stream` or `None`

Returns Return the last result of this query or `None` if the result doesn't contain any streams.

order_by (*attribute_name*: `str`) → `pytube.query.StreamQuery`
Apply a sort order. Filters out stream the do not have the attribute.

Parameters **attribute_name** (`str`) – The name of the attribute to sort by.

otf (*is_otf*: `bool = False`) → `pytube.query.StreamQuery`
Filter stream by OTF, useful if some streams have 404 URLs

Parameters **is_otf** (`bool`) – Set to `False` to retrieve only non-OTF streams

Return type `StreamQuery`

Returns A `StreamQuery` object with otf filtered streams

3.1.6 Caption Object

class `pytube.Caption` (*caption_track*: `Dict[KT, VT]`)

Container for caption tracks.

download (*title*: `str`, *srt*: `bool = True`, *output_path*: `Optional[str] = None`, *filename_prefix*: `Optional[str] = None`) → `str`
Write the media stream to disk.

Parameters

- **title** (`str`) – Output filename (stem only) for writing media file. If one is not specified, the default filename is used.
- **srt** – Set to `True` to download srt, false to download xml. Defaults to `True`.

:type srt `bool` :param *output_path*:

(optional) Output path for writing media file. If one is not specified, defaults to the current working directory.

Parameters **filename_prefix** (`str` or `None`) – (optional) A string that will be prepended to the filename. For example a number in a playlist or the name of a series. If one is not specified, nothing will be prepended This is separate from filename so you can use the default filename but still add a prefix.

Return type `str`

static **float_to_srt_time_format** (*d*: `float`) → `str`
Convert decimal durations into proper srt format.

Return type `str`

Returns SubRip Subtitle (srt) formatted time duration.

`float_to_srt_time_format(3.89)` -> `'00:00:03,890'`

generate_srt_captions () → str
Generate “SubRip Subtitle” captions.

Takes the xml captions from `xml_captions()` and recompiles them into the “SubRip Subtitle” format.

json_captions
Download and parse the json caption tracks.

xml_caption_to_srt (*xml_captions: str*) → str
Convert xml caption tracks to “SubRip Subtitle (srt)”.

Parameters **xml_captions** (*str*) – XML formatted caption tracks.

xml_captions
Download the xml caption tracks.

3.1.7 CaptionQuery Object

class `pytube.query.CaptionQuery` (*captions: List[pytube.captions.Caption]*)
Interface for querying the available captions.

all () → List[pytube.captions.Caption]
Get all the results represented by this query as a list.

Return type list

get (*k*, *d*) → D[k] if k in D, else d. d defaults to None.

get_by_language_code (*lang_code: str*) → Optional[pytube.captions.Caption]
Get the Caption for a given lang_code.

Parameters **lang_code** (*str*) – The code that identifies the caption language.

Return type Caption or None

Returns The Caption matching the given lang_code or None if it does not exist.

items () → a set-like object providing a view on D’s items

keys () → a set-like object providing a view on D’s keys

values () → an object providing a view on D’s values

3.1.8 Search Object

class `pytube.contrib.search.Search` (*query*)

completion_suggestions
Return query autocompletion suggestions for the query.

Return type list

Returns A list of autocomplete suggestions provided by YouTube for the query.

fetch_and_parse (*continuation=None*)
Fetch from the innertube API and parse the results.

Parameters **continuation** (*str*) – Continuation string for fetching results.

Return type tuple

Returns A tuple of a list of YouTube objects and a continuation string.

fetch_query (*continuation=None*)

Fetch raw results from the innertube API.

Parameters **continuation** (*str*) – Continuation string for fetching results.

Return type *dict*

Returns The raw json object returned by the innertube API.

get_next_results ()

Use the stored continuation string to fetch the next set of results.

This method does not return the results, but instead updates the results property.

results

Return search results.

On first call, will generate and return the first set of results. Additional results can be generated using `.get_next_results()`.

Return type *list*

Returns A list of YouTube objects.

3.1.9 Extract

This module contains all non-cipher related data extraction logic.

`pytube.extract.apply_descrambler (stream_data: Dict[KT, VT]) → None`

Apply various in-place transforms to YouTube's media stream data.

Creates a list of dictionaries by string splitting on commas, then taking each list item, parsing it as a query string, converting it to a dict and unquoting the value.

Parameters **stream_data** (*dict*) – Dictionary containing query string encoded values.

Example:

```
>>> d = {'foo': 'bar=1&var=test,em=5&t=url%20encoded'}
>>> apply_descrambler(d, 'foo')
>>> print(d)
{'foo': [{'bar': '1', 'var': 'test'}, {'em': '5', 't': 'url encoded'}]}
```

`pytube.extract.apply_signature (stream_manifest: Dict[KT, VT], vid_info: Dict[KT, VT], js: str)`

→ None

Apply the decrypted signature to the stream manifest.

Parameters

- **stream_manifest** (*dict*) – Details of the media streams available.
- **js** (*str*) – The contents of the base.js asset file.

`pytube.extract.channel_name (url: str) → str`

Extract the channel_name or channel_id from a YouTube url.

This function supports the following patterns:

- `https://youtube.com/c/channel_name/ *`
- `:samp:https://youtube.com/channel/{channel_id}/*`
- `https://youtube.com/u/channel_name/ *`
- `:samp:https://youtube.com/user/{channel_id}/*`

Parameters `url` (*str*) – A YouTube url containing a channel name.

Return type `str`

Returns YouTube channel name.

`pytube.extract.get_ytcfg(html: str) → str`

Get the entirety of the ytcfg object.

This is built over multiple pieces, so we have to find all matches and combine the dicts together.

Parameters `html` (*str*) – The html contents of the watch page.

Return type `str`

Returns Substring of the html containing the encoded manifest data.

`pytube.extract.get_ytplayer_config(html: str) → Any`

Get the YouTube player configuration data from the watch html.

Extract the `ytplayer_config`, which is json data embedded within the watch html and serves as the primary source of obtaining the stream manifest data.

Parameters `html` (*str*) – The html contents of the watch page.

Return type `str`

Returns Substring of the html containing the encoded manifest data.

`pytube.extract.get_ytplayer_js(html: str) → Any`

Get the YouTube player base JavaScript path.

:param str html The html contents of the watch page.

Return type `str`

Returns Path to YouTube's base.js file.

`pytube.extract.initial_data(watch_html: str) → str`

Extract the `ytInitialData` json from the `watch_html` page.

This mostly contains metadata necessary for rendering the page on-load, such as video information, copyright notices, etc.

@param watch_html: Html of the watch page @return:

`pytube.extract.initial_player_response(watch_html: str) → str`

Extract the `ytInitialPlayerResponse` json from the `watch_html` page.

This mostly contains metadata necessary for rendering the page on-load, such as video information, copyright notices, etc.

@param watch_html: Html of the watch page @return:

`pytube.extract.is_age_restricted(watch_html: str) → bool`

Check if content is age restricted.

Parameters `watch_html` (*str*) – The html contents of the watch page.

Return type `bool`

Returns Whether or not the content is age restricted.

`pytube.extract.is_private(watch_html)`

Check if content is private.

Parameters `watch_html` (*str*) – The html contents of the watch page.

Return type `bool`

Returns Whether or not the content is private.

`pytube.extract.js_url` (*html: str*) → `str`

Get the base JavaScript url.

Construct the base JavaScript url, which contains the decipher “transforms”.

Parameters `html` (*str*) – The html contents of the watch page.

`pytube.extract.metadata` (*initial_data*) → `Optional[pytube.metadata.YouTubeMetadata]`

Get the informational metadata for the video.

e.g.: [

```
{ 'Song': '(Gangnam Style)', 'Artist': 'PSY', 'Album': 'PSY SIX RULES Pt.1', 'Licensed to
  YouTube by': 'YG Entertainment Inc. [...]'
}
```

]

Return type `YouTubeMetadata`

`pytube.extract.mime_type_codec` (*mime_type_codec: str*) → `Tuple[str, List[str]]`

Parse the type data.

Breaks up the data in the `type` key of the manifest, which contains the mime type and codecs serialized together, and splits them into separate elements.

Example:

`mime_type_codec('audio/webm; codecs="opus"') -> ('audio/webm', ['opus'])`

Parameters `mime_type_codec` (*str*) – String containing mime type and codecs.

Return type `tuple`

Returns The mime type and a list of codecs.

`pytube.extract.playability_status` (*watch_html: str*) -> (`<class 'str'>`, `<class 'str'>`)

Return the playability status and status explanation of a video.

For example, a video may have a status of `LOGIN_REQUIRED`, and an explanation of “This is a private video. Please sign in to verify that you may see it.”

This explanation is what gets incorporated into the media player overlay.

Parameters `watch_html` (*str*) – The html contents of the watch page.

Return type `bool`

Returns Playability status and reason of the video.

`pytube.extract.playlist_id` (*url: str*) → `str`

Extract the `playlist_id` from a YouTube url.

This function supports the following patterns:

- `https://youtube.com/playlist?list=playlist_id`
- `https://youtube.com/watch?v=video_id&list=playlist_id`

Parameters `url` (*str*) – A YouTube url containing a playlist id.

Return type `str`

Returns YouTube playlist id.

`pytube.extract.publish_date(watch_html: str)`

Extract publish date :param str watch_html:

The html contents of the watch page.

Return type `str`

Returns Publish date of the video.

`pytube.extract.recording_available(watch_html)`

Check if live stream recording is available.

Parameters `watch_html (str)` – The html contents of the watch page.

Return type `bool`

Returns Whether or not the content is private.

`pytube.extract.video_id(url: str) → str`

Extract the video_id from a YouTube url.

This function supports the following patterns:

- `https://youtube.com/watch?v=video_id`
- `https://youtube.com/embed/video_id`
- `https://youtu.be/video_id`

Parameters `url (str)` – A YouTube url containing a video id.

Return type `str`

Returns YouTube video id.

`pytube.extract.video_info_url(video_id: str, watch_url: str) → str`

Construct the video_info url.

Parameters

- `video_id (str)` – A YouTube video identifier.
- `watch_url (str)` – A YouTube watch url.

Return type `str`

Returns `https://youtube.com/get_video_info` with necessary GET parameters.

`pytube.extract.video_info_url_age_restricted(video_id: str, embed_html: str) → str`

Construct the video_info url.

Parameters

- `video_id (str)` – A YouTube video identifier.
- `embed_html (str)` – The html contents of the embed page (for age restricted videos).

Return type `str`

Returns `https://youtube.com/get_video_info` with necessary GET parameters.

3.1.10 Cipher

This module contains all logic necessary to decipher the signature.

YouTube’s strategy to restrict downloading videos is to send a ciphered version of the signature to the client, along with the decryption algorithm obfuscated in JavaScript. For the clients to play the videos, JavaScript must take the ciphered version, cycle it through a series of “transform functions,” and then signs the media URL with the output.

This module is responsible for (1) finding and extracting those “transform functions” (2) maps them to Python equivalents and (3) taking the ciphered signature and decoding it.

`pytube.cipher.get_initial_function_name(js: str) → str`
Extract the name of the function responsible for computing the signature. :param str js:
The contents of the base.js asset file.

Return type `str`

Returns Function name from regex match

`pytube.cipher.get_throttling_function_array(js: str) → List[Any]`
Extract the “c” array.

Parameters `js (str)` – The contents of the base.js asset file.

Returns The array of various integers, arrays, and functions.

`pytube.cipher.get_throttling_function_code(js: str) → str`
Extract the raw code for the throttling function.

Parameters `js (str)` – The contents of the base.js asset file.

Return type `str`

Returns The name of the function used to compute the throttling parameter.

`pytube.cipher.get_throttling_function_name(js: str) → str`
Extract the name of the function that computes the throttling parameter.

Parameters `js (str)` – The contents of the base.js asset file.

Return type `str`

Returns The name of the function used to compute the throttling parameter.

`pytube.cipher.get_throttling_plan(js: str)`
Extract the “throttling plan”.

The “throttling plan” is a list of tuples used for calling functions in the c array. The first element of the tuple is the index of the function to call, and any remaining elements of the tuple are arguments to pass to that function.

Parameters `js (str)` – The contents of the base.js asset file.

Returns The full function code for computing the throttling parameter.

`pytube.cipher.get_transform_map(js: str, var: str) → Dict[KT, VT]`
Build a transform function lookup.

Build a lookup table of obfuscated JavaScript function names to the Python equivalents.

Parameters

- `js (str)` – The contents of the base.js asset file.
- `var (str)` – The obfuscated variable name that stores an object with all functions that descrambles the signature.

`pytube.cipher.get_transform_object` (*js: str, var: str*) → List[str]
Extract the “transform object”.

The “transform object” contains the function definitions referenced in the “transform plan”. The `var` argument is the obfuscated variable name which contains these functions, for example, given the function call `DE.AJ(a, 15)` returned by the transform plan, “DE” would be the var.

Parameters

- **js** (*str*) – The contents of the base.js asset file.
- **var** (*str*) – The obfuscated variable name that stores an object with all functions that descrambles the signature.

Example:

```
>>> get_transform_object(js, 'DE')
['AJ:function(a){a.reverse()} ',
'VR:function(a,b){a.splice(0,b)} ',
'kT:function(a,b){var c=a[0];a[0]=a[b%a.length];a[b]=c} ']
```

`pytube.cipher.get_transform_plan` (*js: str*) → List[str]
Extract the “transform plan”.

The “transform plan” is the functions that the ciphered signature is cycled through to obtain the actual signature.

Parameters **js** (*str*) – The contents of the base.js asset file.

Example:

```
['DE.AJ(a,15)', 'DE.VR(a,3)', 'DE.AJ(a,51)', 'DE.VR(a,3)', 'DE.kT(a,51)', 'DE.kT(a,8)', 'DE.VR(a,3)', 'DE.kT(a,21)']
```

`pytube.cipher.js_splice` (*arr: list, start: int, delete_count=None, *items*)
Implementation of javascript’s splice function.

Parameters

- **arr** (*list*) – Array to splice
- **start** (*int*) – Index at which to start changing the array
- **delete_count** (*int*) – Number of elements to delete from the array
- ***items** – Items to add to the array

Reference: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/splice #noqa:E501

`pytube.cipher.map_functions` (*js_func: str*) → Callable
For a given JavaScript transform function, return the Python equivalent.

Parameters **js_func** (*str*) – The JavaScript version of the transform function.

`pytube.cipher.reverse` (*arr: List[T], _: Optional[Any]*)
Reverse elements in a list.

This function is equivalent to:

```
function(a, b) { a.reverse() }
```

This method takes an unused `b` variable as their transform functions universally sent two arguments.

Example:

```
>>> reverse([1, 2, 3, 4])
[4, 3, 2, 1]
```

`pytube.cipher.splice (arr: List[T], b: int)`
Add/remove items to/from a list.

This function is equivalent to:

```
function(a, b) { a.splice(0, b) }
```

Example:

```
>>> splice([1, 2, 3, 4], 2)
[1, 2]
```

`pytube.cipher.swap (arr: List[T], b: int)`
Swap positions at b modulus the list length.

This function is equivalent to:

```
function(a, b) { var c=a[0];a[0]=a[b%a.length];a[b]=c }
```

Example:

```
>>> swap([1, 2, 3, 4], 2)
[3, 2, 1, 4]
```

`pytube.cipher.throttling_cipher_function (d: list, e: str)`

This ciphers d with e to generate a new list.

In the javascript, the operation is as follows: `var h = [A-Za-z0-9-_], f = 96; // simplified from switch-case loop`
`d.forEach(`

```
function(l,m,n){
  this.push(
    n[m]=h[ (h.indexOf(l)-h.indexOf(this[m])+m-32+f-)%h.length
  ]
)
}, e.split(""))
)
```

`pytube.cipher.throttling_mod_func (d: list, e: int)`

Perform the modular function from the throttling array functions.

In the javascript, the modular operation is as follows: `e = (e % d.length + d.length) % d.length`

We simply translate this to python here.

`pytube.cipher.throttling_nested_splice (d: list, e: int)`

Nested splice function in throttling js.

In the javascript, the operation is as follows: `function(d,e){`

```
e=(e%d.length+d.length)%d.length; d.splice(
  0, 1, d.splice(
    e, 1, d[0]
```

```

        )[0]
    )
}

```

While testing, all this seemed to do is swap element 0 and e, but the actual process is preserved in case there was an edge case that was not considered.

```

pytube.cipher.throttling_prepend(d: list, e: int)
In the javascript, the operation is as follows: function(d,e){
    e=(e%d.length+d.length)%d.length; d.splice(-e).reverse().forEach(
        function(f){ d.unshift(f)
        }
    )
}

```

Effectively, this moves the last e elements of d to the beginning.

```

pytube.cipher.throttling_push(d: list, e: Any)
Pushes an element onto a list.

```

```

pytube.cipher.throttling_reverse(arr: list)
Reverses the input list.

```

Needs to do an in-place reversal so that the passed list gets changed. To accomplish this, we create a reversed copy, and then change each individual element.

```

pytube.cipher.throttling_swap(d: list, e: int)
Swap positions of the 0'th and e'th elements in-place.

```

```

pytube.cipher.throttling_unshift(d: list, e: int)
Rotates the elements of the list to the right.

```

In the javascript, the operation is as follows: for(e=(e%d.length+d.length)%d.length;e--;)d.unshift(d.pop())

3.1.11 Exceptions

Library specific exception definitions.

```

exception pytube.exceptions.AgeRestrictedError(video_id: str)
Video is age restricted, and cannot be accessed without OAuth.

```

```

exception pytube.exceptions.ExtractError
Data extraction based exception.

```

```

exception pytube.exceptions.HTMLParseError
HTML could not be parsed

```

```

exception pytube.exceptions.LiveStreamError(video_id: str)
Video is a live stream.

```

```

exception pytube.exceptions.MaxRetriesExceeded
Maximum number of retries exceeded.

```

```

exception pytube.exceptions.MembersOnly(video_id: str)
Video is members-only.

```

YouTube has special videos that are only viewable to users who have subscribed to a content creator. ref: <https://support.google.com/youtube/answer/7544492?hl=en>

exception `pytube.exceptions.PytubeError`

Base pytube exception that all others inherit.

This is done to not pollute the built-in exceptions, which *could* result in unintended errors being unexpectedly and incorrectly handled within implementers code.

exception `pytube.exceptions.RecordingUnavailable` (*video_id: str*)

exception `pytube.exceptions.RegexMatchError` (*caller: str, pattern: Union[str, Pattern[AnyStr]]*)

Regex pattern did not return any matches.

exception `pytube.exceptions.VideoPrivate` (*video_id: str*)

exception `pytube.exceptions.VideoRegionBlocked` (*video_id: str*)

exception `pytube.exceptions.VideoUnavailable` (*video_id: str*)

Base video unavailable error.

3.1.12 Helpers

Various helper functions implemented by pytube.

class `pytube.helpers.DeferredGeneratorList` (*generator*)

A wrapper class for deferring list generation.

Pytube has some continuation generators that create web calls, which means that any time a full list is requested, all of those web calls must be made at once, which could lead to slowdowns. This will allow individual elements to be queried, so that slowdowns only happen as necessary. For example, you can iterate over elements in the list without accessing them all simultaneously. This should allow for speed improvements for playlist and channel interactions.

generate_all ()

Generate all items.

`pytube.helpers.cache` (*func: Callable[[...], GenericType]*) → *GenericType*

mypy compatible annotation wrapper for `lru_cache`

`pytube.helpers.create_mock_html_json` (*vid_id*) → *Dict[str, Any]*

Generate a `json.gz` file with sample html responses.

:param str vid_id YouTube video id

:return dict data Dict used to generate the `json.gz` file

`pytube.helpers.deprecated` (*reason: str*) → *Callable*

This is a decorator which can be used to mark functions as deprecated. It will result in a warning being emitted when the function is used.

`pytube.helpers.generate_all_html_json_mocks` ()

Regenerate the video mock json files for all current test videos.

This should automatically output to the `test/mocks` directory.

`pytube.helpers.regex_search` (*pattern: str, string: str, group: int*) → *str*

Shortcut method to search a string for a given pattern.

Parameters

- **pattern** (*str*) – A regular expression pattern.
- **string** (*str*) – A target string to search.
- **group** (*int*) – Index of group to return.

Return type `str` or `tuple`

Returns Substring pattern matches.

`pytube.helpers.safe_filename (s: str, max_length: int = 255) → str`
Sanitize a string making it safe to use as a filename.

This function was based off the limitations outlined here: <https://en.wikipedia.org/wiki/Filename>.

Parameters

- **s** (`str`) – A string to make safe for use as a file name.
- **max_length** (`int`) – The maximum filename character length.

Return type `str`

Returns A sanitized string.

`pytube.helpers.setup_logger (level: int = 40, log_filename: Optional[str] = None) → None`
Create a configured instance of logger.

Parameters **level** (`int`) – Describe the severity level of the logs to handle.

`pytube.helpers.target_directory (output_path: Optional[str] = None) → str`
Function for determining target directory of a download. Returns an absolute path (if relative one given) or the current path (if none given). Makes directory if it does not exist.

Returns An absolute directory path as a string.

`pytube.helpers.uniqueify (duped_list: List[T]) → List[T]`
Remove duplicate items from a list, while maintaining list order.

:param List duped_list List to remove duplicates from

:return List result De-duplicated list

3.1.13 Request

Implements a simple wrapper around `urlopen`.

`pytube.request.filesize`
Fetch size in bytes of file at given URL

Parameters **url** (`str`) – The URL to get the size of

Returns `int`: size in bytes of remote file

`pytube.request.get (url, extra_headers=None, timeout=<object object>)`
Send an http GET request.

Parameters

- **url** (`str`) – The URL to perform the GET request for.
- **extra_headers** (`dict`) – Extra headers to add to the request

Return type `str`

Returns UTF-8 encoded string of response

`pytube.request.head (url)`
Fetch headers returned http GET request.

Parameters **url** (`str`) – The URL to perform the GET request for.

Return type `dict`

Returns dictionary of lowercase headers

`pytube.request.post(url, extra_headers=None, data=None, timeout=<object object>)`
Send an http POST request.

Parameters

- **url** (*str*) – The URL to perform the POST request for.
- **extra_headers** (*dict*) – Extra headers to add to the request
- **data** (*dict*) – The data to send on the POST request

Return type *str*

Returns UTF-8 encoded string of response

`pytube.request.seq_filesize`
Fetch size in bytes of file at given URL from sequential requests

Parameters **url** (*str*) – The URL to get the size of

Returns int: size in bytes of remote file

`pytube.request.seq_stream(url, timeout=<object object>, max_retries=0)`
Read the response in sequence. :param str url: The URL to perform the GET request for. :rtype: Iterable[bytes]

`pytube.request.stream(url, timeout=<object object>, max_retries=0)`
Read the response in chunks. :param str url: The URL to perform the GET request for. :rtype: Iterable[bytes]

CHAPTER 4

Indices and tables

- `genindex`
- `modindex`
- *Using the search feature*

p

- `pytube`, [15](#)
- `pytube.cipher`, [32](#)
- `pytube.exceptions`, [35](#)
- `pytube.extract`, [28](#)
- `pytube.helpers`, [36](#)
- `pytube.request`, [37](#)

A

about_html (*pytube.contrib.channel.Channel* attribute), 19
 AgeRestrictedError, 35
 all() (*pytube.query.CaptionQuery* method), 27
 all() (*pytube.query.StreamQuery* method), 24
 apply_descrambler() (in module *pytube.extract*), 28
 apply_signature() (in module *pytube.extract*), 28
 asc() (*pytube.query.StreamQuery* method), 24
 author (*pytube.YouTube* attribute), 15

B

bypass_age_gate() (*pytube.YouTube* method), 15

C

cache() (in module *pytube.helpers*), 36
 Caption (class in *pytube*), 26
 caption_tracks (*pytube.YouTube* attribute), 15
 CaptionQuery (class in *pytube.query*), 27
 captions (*pytube.YouTube* attribute), 15
 Channel (class in *pytube.contrib.channel*), 19
 channel_id (*pytube.contrib.channel.Channel* attribute), 19
 channel_id (*pytube.YouTube* attribute), 15
 channel_name (*pytube.contrib.channel.Channel* attribute), 19
 channel_name() (in module *pytube.extract*), 28
 channel_url (*pytube.YouTube* attribute), 15
 check_availability() (*pytube.YouTube* method), 16
 community_html (*pytube.contrib.channel.Channel* attribute), 19
 completion_suggestions (*pytube.contrib.search.Search* attribute), 27
 count() (*pytube.contrib.channel.Channel* method), 19
 count() (*pytube.contrib.playlist.Playlist* method), 17
 count() (*pytube.query.StreamQuery* method), 24

create_mock_html_json() (in module *pytube.helpers*), 36

D

default_filename (*pytube.Stream* attribute), 22
 DeferredGeneratorList (class in *pytube.helpers*), 36
 deprecated() (in module *pytube.helpers*), 36
 desc() (*pytube.query.StreamQuery* method), 24
 description (*pytube.YouTube* attribute), 16
 download() (*pytube.Caption* method), 26
 download() (*pytube.Stream* method), 22

E

ExtractError, 35

F

featured_channels_html (*pytube.contrib.channel.Channel* attribute), 19
 fetch_and_parse() (*pytube.contrib.search.Search* method), 27
 fetch_query() (*pytube.contrib.search.Search* method), 27
 filesize (in module *pytube.request*), 37
 filesize (*pytube.Stream* attribute), 22
 filesize_approx (*pytube.Stream* attribute), 22
 filesize_gb (*pytube.Stream* attribute), 22
 filesize_kb (*pytube.Stream* attribute), 22
 filesize_mb (*pytube.Stream* attribute), 23
 filter() (*pytube.query.StreamQuery* method), 24
 first() (*pytube.query.StreamQuery* method), 25
 float_to_srt_time_format() (*pytube.Caption* static method), 26
 fmt_streams (*pytube.YouTube* attribute), 16
 from_id() (*pytube.YouTube* static method), 16

G

generate_all() (*pytube.helpers.DeferredGeneratorList* method),

36
generate_all_html_jsonmocks() (in module *pytube.helpers*), 36
generate_srt_captions() (*pytube.Caption* method), 26
get() (in module *pytube.request*), 37
get() (*pytube.query.CaptionQuery* method), 27
get_audio_only() (*pytube.query.StreamQuery* method), 25
get_by_itag() (*pytube.query.StreamQuery* method), 25
get_by_language_code() (*pytube.query.CaptionQuery* method), 27
get_by_resolution() (*pytube.query.StreamQuery* method), 25
get_highest_resolution() (*pytube.query.StreamQuery* method), 25
get_initial_function_name() (in module *pytube.cipher*), 32
get_lowest_resolution() (*pytube.query.StreamQuery* method), 25
get_next_results() (*pytube.contrib.search.Search* method), 28
get_throttling_function_array() (in module *pytube.cipher*), 32
get_throttling_function_code() (in module *pytube.cipher*), 32
get_throttling_function_name() (in module *pytube.cipher*), 32
get_throttling_plan() (in module *pytube.cipher*), 32
get_transform_map() (in module *pytube.cipher*), 32
get_transform_object() (in module *pytube.cipher*), 32
get_transform_plan() (in module *pytube.cipher*), 33
get_ytcfg() (in module *pytube.extract*), 29
get_ytplayer_config() (in module *pytube.extract*), 29
get_ytplayer_js() (in module *pytube.extract*), 29

H

head() (in module *pytube.request*), 37
html (*pytube.contrib.channel.Channel* attribute), 19
html (*pytube.contrib.playlist.Playlist* attribute), 17
HTMLParseError, 35

I

includes_audio_track (*pytube.Stream* attribute), 23
includes_video_track (*pytube.Stream* attribute), 23
index() (*pytube.contrib.channel.Channel* method), 20

index() (*pytube.contrib.playlist.Playlist* method), 17
index() (*pytube.query.StreamQuery* method), 26
initial_data (*pytube.contrib.channel.Channel* attribute), 20
initial_data (*pytube.contrib.playlist.Playlist* attribute), 17
initial_data() (in module *pytube.extract*), 29
initial_player_response() (in module *pytube.extract*), 29
is_adaptive (*pytube.Stream* attribute), 23
is_age_restricted() (in module *pytube.extract*), 29
is_private() (in module *pytube.extract*), 29
is_progressive (*pytube.Stream* attribute), 23
items() (*pytube.query.CaptionQuery* method), 27

J

js_splice() (in module *pytube.cipher*), 33
js_url() (in module *pytube.extract*), 30
json_captions (*pytube.Caption* attribute), 27

K

keys() (*pytube.query.CaptionQuery* method), 27
keywords (*pytube.YouTube* attribute), 16

L

last() (*pytube.query.StreamQuery* method), 26
last_updated (*pytube.contrib.channel.Channel* attribute), 20
last_updated (*pytube.contrib.playlist.Playlist* attribute), 17
length (*pytube.contrib.channel.Channel* attribute), 20
length (*pytube.contrib.playlist.Playlist* attribute), 17
length (*pytube.YouTube* attribute), 16
LiveStreamError, 35

M

map_functions() (in module *pytube.cipher*), 33
MaxRetriesExceeded, 35
MembersOnly, 35
metadata (*pytube.YouTube* attribute), 16
metadata() (in module *pytube.extract*), 30
mime_type_codec() (in module *pytube.extract*), 30

O

on_complete() (*pytube.Stream* method), 23
on_progress() (*pytube.Stream* method), 23
order_by() (*pytube.query.StreamQuery* method), 26
otf() (*pytube.query.StreamQuery* method), 26
owner (*pytube.contrib.channel.Channel* attribute), 20
owner (*pytube.contrib.playlist.Playlist* attribute), 18
owner_id (*pytube.contrib.channel.Channel* attribute), 20

owner_id (*pytube.contrib.playlist.Playlist attribute*), 18
 owner_url (*pytube.contrib.channel.Channel attribute*), 20
 owner_url (*pytube.contrib.playlist.Playlist attribute*), 18

P

parse_codecs() (*pytube.Stream method*), 23
 playability_status() (*in module pytube.extract*), 30
 Playlist (*class in pytube.contrib.playlist*), 17
 playlist_id (*pytube.contrib.channel.Channel attribute*), 20
 playlist_id (*pytube.contrib.playlist.Playlist attribute*), 18
 playlist_id() (*in module pytube.extract*), 30
 playlist_url (*pytube.contrib.channel.Channel attribute*), 20
 playlist_url (*pytube.contrib.playlist.Playlist attribute*), 18
 playlists_html (*pytube.contrib.channel.Channel attribute*), 20
 post() (*in module pytube.request*), 38
 publish_date (*pytube.YouTube attribute*), 16
 publish_date() (*in module pytube.extract*), 31
 pytube (*module*), 15
 pytube.cipher (*module*), 32
 pytube.exceptions (*module*), 35
 pytube.extract (*module*), 28
 pytube.helpers (*module*), 36
 pytube.request (*module*), 37
 PytubeError, 36

R

rating (*pytube.YouTube attribute*), 16
 recording_available() (*in module pytube.extract*), 31
 RecordingUnavailable, 36
 regex_search() (*in module pytube.helpers*), 36
 RegexMatchError, 36
 register_on_complete_callback() (*pytube.YouTube method*), 16
 register_on_progress_callback() (*pytube.YouTube method*), 16
 results (*pytube.contrib.search.Search attribute*), 28
 reverse() (*in module pytube.cipher*), 33

S

safe_filename() (*in module pytube.helpers*), 37
 Search (*class in pytube.contrib.search*), 27
 seq_filesize (*in module pytube.request*), 38
 seq_stream() (*in module pytube.request*), 38
 setup_logger() (*in module pytube.helpers*), 37

sidebar_info (*pytube.contrib.channel.Channel attribute*), 20
 sidebar_info (*pytube.contrib.playlist.Playlist attribute*), 18
 splice() (*in module pytube.cipher*), 34
 Stream (*class in pytube*), 22
 stream() (*in module pytube.request*), 38
 stream_to_buffer() (*pytube.Stream method*), 23
 streaming_data (*pytube.YouTube attribute*), 16
 StreamQuery (*class in pytube.query*), 24
 streams (*pytube.YouTube attribute*), 17
 swap() (*in module pytube.cipher*), 34

T

target_directory() (*in module pytube.helpers*), 37
 throttling_cipher_function() (*in module pytube.cipher*), 34
 throttling_mod_func() (*in module pytube.cipher*), 34
 throttling_nested_splice() (*in module pytube.cipher*), 34
 throttling_prepend() (*in module pytube.cipher*), 35
 throttling_push() (*in module pytube.cipher*), 35
 throttling_reverse() (*in module pytube.cipher*), 35
 throttling_swap() (*in module pytube.cipher*), 35
 throttling_unshift() (*in module pytube.cipher*), 35
 thumbnail_url (*pytube.YouTube attribute*), 17
 title (*pytube.contrib.channel.Channel attribute*), 21
 title (*pytube.contrib.playlist.Playlist attribute*), 18
 title (*pytube.Stream attribute*), 24
 title (*pytube.YouTube attribute*), 17
 trimmed() (*pytube.contrib.channel.Channel method*), 21
 trimmed() (*pytube.contrib.playlist.Playlist method*), 18

U

uniqueify() (*in module pytube.helpers*), 37
 url_generator() (*pytube.contrib.channel.Channel method*), 21
 url_generator() (*pytube.contrib.playlist.Playlist method*), 18

V

values() (*pytube.query.CaptionQuery method*), 27
 vanity_url (*pytube.contrib.channel.Channel attribute*), 21
 vid_info (*pytube.YouTube attribute*), 17
 video_id() (*in module pytube.extract*), 31
 video_info_url() (*in module pytube.extract*), 31

`video_info_url_age_restricted()` (in module `pytube.extract`), 31
`video_urls` (`pytube.contrib.channel.Channel` attribute), 21
`video_urls` (`pytube.contrib.playlist.Playlist` attribute), 18
`VideoPrivate`, 36
`VideoRegionBlocked`, 36
`videos` (`pytube.contrib.channel.Channel` attribute), 21
`videos` (`pytube.contrib.playlist.Playlist` attribute), 18
`VideoUnavailable`, 36
`views` (`pytube.contrib.channel.Channel` attribute), 21
`views` (`pytube.contrib.playlist.Playlist` attribute), 19
`views` (`pytube.YouTube` attribute), 17

X

`xml_caption_to_srt()` (`pytube.Caption` method), 27
`xml_captions` (`pytube.Caption` attribute), 27

Y

`YouTube` (class in `pytube`), 15
`yt_api_key` (`pytube.contrib.channel.Channel` attribute), 21
`yt_api_key` (`pytube.contrib.playlist.Playlist` attribute), 19
`ytcfg` (`pytube.contrib.channel.Channel` attribute), 21
`ytcfg` (`pytube.contrib.playlist.Playlist` attribute), 19