# pytube Documentation

***Release 10.5.1***

**Nick Ficano**

**Mar 02, 2021**

# Contents

Release v10.5.1. (*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

CHAPTER 2

# The User Guide

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
    )
```

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.

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 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=2lAe1cqCOXo')
>>> 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
00:00:13,400 --> 00:00:16,200
That is so awkward to watch.
...
```

## 2.6 Using Playlists

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

### 2.6.1 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.

### 2.6.2 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 Beginers (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)
Python Tutorial for Beginners 1 - Getting Started and Installing Python (For Absolute␣
↪Beginners)
Python Tutorial for Beginners 2 - Numbers and Math in Python
Python Tutorial for Beginners 3 - Variables and Inputs
```

And that's basically all there is to it! The Playlist class is relatively straightforward.

## 2.7 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.

## 2.8 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*.

CHAPTER 3

# 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, defer_prefetch_init: bool = False, 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*)

Core developer interface for pytube.

**author**
Get the video author. :rtype: str

**caption_tracks**
Get a list of *Caption*.

**Return type** List[*Caption*]

**captions**
Interface to query caption tracks.

**Return type** CaptionQuery.

**check_availability**()
Check whether the video is available. Raises different exceptions based on why the video is unavailable, otherwise does nothing.

**descramble**() → None
Descramble the stream data and build Stream instances.

The initialization process takes advantage of Python's "call-by-reference evaluation," which allows dictionary transforms to be applied in-place, instead of holding references to mutations at each interstitial step.

**Return type** None

**description**
Get the video description.

**Return type** str

**initialize_stream_objects** (*fmt: str*) → None
Convert manifest data to instances of *Stream*.

Take the unscrambled stream data and uses it to initialize instances of *Stream* for each media stream.

**Parameters** **fmt** (*str*) – Key in stream manifest (`ytplayer_config`) containing progressive download or adaptive streams (e.g.: `url_encoded_fmt_stream_map` or `adaptive_fmts`).

**Return type** None

**keywords**
Get the video keywords. :rtype: List[str]

**length**
Get the video length in seconds.

**Return type** int

**metadata**
Get the metadata for the video.

**Return type** YouTubeMetadata

**prefetch** () → None
Eagerly download all necessary data.

Eagerly executes all necessary network requests so all other operations don't does need to make calls outside of the interpreter which blocks for long periods of time.

**Return type** None

**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

**`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

**`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

**`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.

**`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

**`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

**Yields** YouTube

### 3.1.3 Stream Object

**class** pytube.**Stream**(*stream: Dict[KT, VT]*, *player_config_args: 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*) → 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

**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

**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.BufferedWriter) – 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

### 3.1.4 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

### 3.1.5 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 (str) 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.

**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.

### 3.1.6 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

### 3.1.7 Extract

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

pytube.extract.**apply_descrambler**(*stream_data: Dict[KT, VT], key: str*) → 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.
- **key** (*str*) – Name of the key in dictionary.

**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**(*config_args: Dict[KT, VT], fmt: str, js: str*) → None
Apply the decrypted signature to the stream manifest.

**Parameters**

- **config_args** (*dict*) – Details of the media streams available.
- **fmt** (*str*) – Key in stream manifests (`ytplayer_config`) containing progressive download or adaptive streams (e.g.: `url_encoded_fmt_stream_map` or `adaptive_fmts`).
- **js** (*str*) – The contents of the base.js asset file.

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.**is_region_blocked**(*watch_html: str*) → bool
Determine if a video is not available in the user's region.

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

**Return type** bool

**Returns** True if the video is blocked in the users region. False if not, or if unknown.

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.

### 3.1.8 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_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.**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]
```

### 3.1.9 Exceptions

Library specific exception definitions.

**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.**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.

### 3.1.10 Helpers

Various helper functions implemented by pytube.

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.**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.

### 3.1.11 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*)
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.**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*, *chunk_size=4096*, *range_size=9437184*)
Read the response in sequence. :param str url: The URL to perform the GET request for. :param int chunk_size: The size in bytes of each chunk. Defaults to 4KB :param int range_size: The size in bytes of each range request. Defaults to 9MB :rtype: Iterable[bytes]

pytube.request.**stream**(*url*, *chunk_size=4096*, *range_size=9437184*)
Read the response in chunks. :param str url: The URL to perform the GET request for. :param int chunk_size: The size in bytes of each chunk. Defaults to 4KB :param int range_size: The size in bytes of each range request. Defaults to 9MB :rtype: Iterable[bytes]

CHAPTER 4

# Indices and tables

- genindex
- modindex
- search

# Python Module Index

## p

# Index

## J



## K



## L



## M



## O



## P



## R



## S



## T



## V

## X



## Y