

---

# **pytube3 Documentation**

***Release 9.6.4***

**Nick Ficano**

**Feb 25, 2020**



---

## Contents

---

<b>1</b>	<b>Features</b>	<b>3</b>
<b>2</b>	<b>Roadmap</b>	<b>5</b>
<b>3</b>	<b>The User Guide</b>	<b>7</b>
3.1	Installation of pytube . . . . .	7
3.2	Quickstart . . . . .	8
3.3	Working with Streams . . . . .	9
3.4	Subtitle/Caption Tracks . . . . .	11
<b>4</b>	<b>The API Documentation / Guide</b>	<b>13</b>
4.1	API . . . . .	13
<b>5</b>	<b>Indices and tables</b>	<b>27</b>
	<b>Python Module Index</b>	<b>29</b>
	<b>Index</b>	<b>31</b>



Release v9.6.4. (*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

---

### Roadmap

---

- Allow downloading age restricted content
- Complete ffmpeg integrationn



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.

### 3.1 Installation of pytube

This part of the documentation covers the installation of pytube.

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

```
$ pip install pytube3
```

#### 3.1.1 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/nficano/pytube.git
```

Or, download the [tarball](#):

```
$ curl -OL https://github.com/hbmartin/pytube3/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  
$ pip install .
```

## 3.2 Quickstart

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

Let's get started with some examples.

### 3.2.1 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 popular like PSY - Gangnam Style:

```
>>> yt = YouTube('https://www.youtube.com/watch?v=9bZkp7q19f0')
```

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
PSY - GANGNAM STYLE() M/V
```

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

```
>>> yt.thumbnail_url
'https://i.ytimg.com/vi/mTOYClXhJD0/default.jpg'
```

Neat, right? Next let's see the available media formats:

```
>>> yt.streams.all()
[<Stream: itag="22" mime_type="video/mp4" res="720p" fps="30fps" vcodec="avc1.64001F"
↳acodec="mp4a.40.2">,
<Stream: itag="43" mime_type="video/webm" res="360p" fps="30fps" vcodec="vp8.0"
↳acodec="vorbis">,
<Stream: itag="18" mime_type="video/mp4" res="360p" fps="30fps" vcodec="avc1.42001E"
↳acodec="mp4a.40.2">,
<Stream: itag="36" mime_type="video/3gpp" res="240p" fps="30fps" vcodec="mp4v.20.3"
↳acodec="mp4a.40.2">,
<Stream: itag="17" mime_type="video/3gpp" res="144p" fps="30fps" vcodec="mp4v.20.3"
↳acodec="mp4a.40.2">,
<Stream: itag="137" mime_type="video/mp4" res="1080p" fps="30fps" vcodec="avc1.640028"
↳">,
<Stream: itag="136" mime_type="video/mp4" res="720p" fps="30fps" vcodec="avc1.4d401f">
↳,
<Stream: itag="135" mime_type="video/mp4" res="480p" fps="30fps" vcodec="avc1.4d401f">
↳,
<Stream: itag="134" mime_type="video/mp4" res="360p" fps="30fps" vcodec="avc1.4d401e">
↳,
<Stream: itag="133" mime_type="video/mp4" res="240p" fps="30fps" vcodec="avc1.4d4015">
↳,
<Stream: itag="160" mime_type="video/mp4" res="144p" fps="30fps" vcodec="avc1.4d400c">
↳,
<Stream: itag="140" mime_type="audio/mp4" abr="128kbps" acodec="mp4a.40.2">,
<Stream: itag="171" mime_type="audio/webm" abr="128kbps" acodec="vorbis">]
```

Let's say we want to get the first stream:

```
>>> stream = yt.streams.first()
>>> stream
<Stream: itag="22" mime_type="video/mp4" res="720p" fps="30fps" vcodec="avc1.64001F"
↳ acodec="mp4a.40.2">
```

And to download it to the current working directory:

```
>>> stream.download()
```

You can also specify a destination path:

```
>>> stream.download('/tmp')
```

## 3.3 Working with Streams

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.

### 3.3.1 DASH vs Progressive Streams

Begin by running the following:

```
>>> yt.streams.all()
[<Stream: itag="22" mime_type="video/mp4" res="720p" fps="30fps" vcodec="avc1.64001F"
↳ acodec="mp4a.40.2">,
<Stream: itag="43" mime_type="video/webm" res="360p" fps="30fps" vcodec="vp8.0"
↳ acodec="vorbis">,
<Stream: itag="18" mime_type="video/mp4" res="360p" fps="30fps" vcodec="avc1.42001E"
↳ acodec="mp4a.40.2">,
<Stream: itag="36" mime_type="video/3gpp" res="240p" fps="30fps" vcodec="mp4v.20.3"
↳ acodec="mp4a.40.2">,
<Stream: itag="17" mime_type="video/3gpp" res="144p" fps="30fps" vcodec="mp4v.20.3"
↳ acodec="mp4a.40.2">,
<Stream: itag="137" mime_type="video/mp4" res="1080p" fps="30fps" vcodec="avc1.640028"
↳ ">,
<Stream: itag="136" mime_type="video/mp4" res="720p" fps="30fps" vcodec="avc1.4d401f">
↳ ,
<Stream: itag="135" mime_type="video/mp4" res="480p" fps="30fps" vcodec="avc1.4d401f">
↳ ,
<Stream: itag="134" mime_type="video/mp4" res="360p" fps="30fps" vcodec="avc1.4d401e">
↳ ,
<Stream: itag="133" mime_type="video/mp4" res="240p" fps="30fps" vcodec="avc1.4d4015">
↳ ,
<Stream: itag="160" mime_type="video/mp4" res="144p" fps="30fps" vcodec="avc1.4d400c">
↳ ,
<Stream: itag="140" mime_type="audio/mp4" abr="128kbps" acodec="mp4a.40.2">,
<Stream: itag="171" mime_type="audio/webm" abr="128kbps" acodec="vorbis">]
```

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.

To only view these progressive download streams:

```
>>> yt.streams.filter(progressive=True).all()
[<Stream: itag="22" mime_type="video/mp4" res="720p" fps="30fps" vcodec="avc1.64001F"
  ↳acodec="mp4a.40.2">,
<Stream: itag="43" mime_type="video/webm" res="360p" fps="30fps" vcodec="vp8.0"
  ↳acodec="vorbis">,
<Stream: itag="18" mime_type="video/mp4" res="360p" fps="30fps" vcodec="avc1.42001E"
  ↳acodec="mp4a.40.2">,
<Stream: itag="36" mime_type="video/3gpp" res="240p" fps="30fps" vcodec="mp4v.20.3"
  ↳acodec="mp4a.40.2">,
<Stream: itag="17" mime_type="video/3gpp" res="144p" fps="30fps" vcodec="mp4v.20.3"
  ↳acodec="mp4a.40.2">]
```

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

```
>>> yt.streams.filter(adaptive=True).all()
[<Stream: itag="137" mime_type="video/mp4" res="1080p" fps="30fps" vcodec="avc1.640028"
↳>,
<Stream: itag="136" mime_type="video/mp4" res="720p" fps="30fps" vcodec="avc1.4d401f">
↳,
<Stream: itag="135" mime_type="video/mp4" res="480p" fps="30fps" vcodec="avc1.4d401f">
↳,
<Stream: itag="134" mime_type="video/mp4" res="360p" fps="30fps" vcodec="avc1.4d401e">
↳,
<Stream: itag="133" mime_type="video/mp4" res="240p" fps="30fps" vcodec="avc1.4d4015">
↳,
<Stream: itag="160" mime_type="video/mp4" res="144p" fps="30fps" vcodec="avc1.4d400c">
↳,
<Stream: itag="140" mime_type="audio/mp4" abr="128kbps" acodec="mp4a.40.2">,
<Stream: itag="171" mime_type="audio/webm" abr="128kbps" acodec="vorbis">]
```

Pytube allows you to filter on every property available (see `pytube.StreamQuery.filter()` for a complete list of filter options), let's take a look at some common examples:

### 3.3.2 Query audio only Streams

To query the streams that contain only the audio track:

```
>>> yt.streams.filter(only_audio=True).all()
[<Stream: itag="140" mime_type="audio/mp4" abr="128kbps" acodec="mp4a.40.2">,
 <Stream: itag="171" mime_type="audio/webm" abr="128kbps" acodec="vorbis">]
```

### 3.3.3 Query MPEG-4 Streams

To query only streams in the MPEG-4 format:

```
>>> yt.streams.filter(file_extension='mp4').all()
[<Stream: itag="22" mime_type="video/mp4" res="720p" fps="30fps" vcodec="avc1.64001F"
  acodec="mp4a.40.2">,
```

(continues on next page)

(continued from previous page)

```

<Stream: itag="18" mime_type="video/mp4" res="360p" fps="30fps" vcodec="avc1.42001E"
↪acodec="mp4a.40.2">,
<Stream: itag="137" mime_type="video/mp4" res="1080p" fps="30fps" vcodec="avc1.640028
↪">,
<Stream: itag="136" mime_type="video/mp4" res="720p" fps="30fps" vcodec="avc1.4d401f">
↪,
<Stream: itag="135" mime_type="video/mp4" res="480p" fps="30fps" vcodec="avc1.4d401f">
↪,
<Stream: itag="134" mime_type="video/mp4" res="360p" fps="30fps" vcodec="avc1.4d401e">
↪,
<Stream: itag="133" mime_type="video/mp4" res="240p" fps="30fps" vcodec="avc1.4d4015">
↪,
<Stream: itag="160" mime_type="video/mp4" res="144p" fps="30fps" vcodec="avc1.4d400c">
↪,
<Stream: itag="140" mime_type="audio/mp4" abr="128kbps" acodec="mp4a.40.2">]

```

### 3.3.4 Get Streams by itag

To get a stream by a specific itag:

```

>>> yt.streams.get_by_itag('22')
<Stream: itag="22" mime_type="video/mp4" res="720p" fps="30fps" vcodec="avc1.64001F"
↪acodec="mp4a.40.2">

```

## 3.4 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('https://youtube.com/watch?v=XJGiS83eQLk')
>>> yt.captions.all()
[<Caption lang="Arabic" code="ar">,
<Caption lang="English (auto-generated)" code="en">,
<Caption lang="English" code="en">,
<Caption lang="English (United Kingdom)" code="en-GB">,
<Caption lang="German" code="de">,
<Caption lang="Greek" code="el">,
<Caption lang="Indonesian" code="id">,
<Caption lang="Sinhala" code="si">,
<Caption lang="Spanish" code="es">,
<Caption lang="Turkish" code="tr">]

```

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="0" dur="5.541">well
↪i&amp;#39;...

```

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

```
>>> print(caption.generate_srt_captions())
1
000:000:00,000 --> 000:000:05,541
well i'm just an editor and i dont know what to type

2
000:000:05,541 --> 000:000:12,321
not new to video. In fact, most films before 1930 were silent and used captions with
↪video

...
```



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

## 4.1 API

### 4.1.1 YouTube Object

```
class pytube.YouTube(url: str, defer_prefetch_init: bool = False, on_progress_callback: Optional[pytube.monostate.OnProgress] = None, on_complete_callback: Optional[pytube.monostate.OnComplete] = 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.

**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`

**length**

Get the video length in seconds.

**Return type** `str`

**prefetch** () → None

Eagerly download all necessary data.

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

**Return type** `None`

**rating**

Get the video average rating.

**Return type** `float`

**register\_on\_complete\_callback** (*func: pytube.monostate.OnComplete*)

Register a download complete callback function post initialization.

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

**Return type** `None`

**register\_on\_progress\_callback** (*func: pytube.monostate.OnProgress*)

Register a download progress callback function post initialization.

**Parameters** **func** (*callable*) – A callback function that takes `stream`, `chunk`, `file_handle`, `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** `str`

## 4.1.2 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

### 4.1.3 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

#### 4.1.4 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*)  $\rightarrow$  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*)  $\rightarrow$  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** ()  $\rightarrow$  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*)  $\rightarrow$  str

Convert xml caption tracks to “SubRip Subtitle (srt)”.

**Parameters xml\_captions** (*str*) – XML formatted caption tracks.

**xml\_captions**

Download the xml caption tracks.

#### 4.1.5 CaptionQuery Object

**class** `pytube.query.CaptionQuery` (*captions: List[pytube.captions.Caption]*)

Interface for querying the available captions.

**all** ()  $\rightarrow$  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

## 4.1.6 Extract

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

**class** `pytube.extract.PytubeHTMLParser` (\*, *convert\_charrefs=True*)

`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.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.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.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.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.

### 4.1.7 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]
```

## 4.1.8 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.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.RegexMatchError` (*caller: str, pattern: Union[str, Pattern[AnyStr]]*)  
Regex pattern did not return any matches.

**exception** `pytube.exceptions.VideoUnavailable` (*video\_id: str*)  
Video is unavailable.

## 4.1.9 Mixins

### 4.1.10 Helpers

Various helper functions implemented by pytube.

`pytube.helpers.cache` (*func: Callable[[...], GenericType]*) → `GenericType`  
mypy compatible annotation wrapper for `lru_cache`

`pytube.helpers.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*)  
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.

### 4.1.11 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*) → str

Send an http GET request.

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

**Return type** str

**Returns** UTF-8 encoded string of response

`pytube.request.head` (*url: str*) → Dict[KT, VT]

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.stream` (*url: str, chunk\_size: int = 4096, range\_size: int = 9437184*) → Iterable[bytes]

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 5

---

### Indices and tables

---

- `genindex`
- `modindex`
- `search`





### p

- `pytube`, [13](#)
- `pytube.cipher`, [22](#)
- `pytube.exceptions`, [23](#)
- `pytube.extract`, [20](#)
- `pytube.helpers`, [24](#)
- `pytube.request`, [25](#)



## A

all() (*pytube.query.CaptionQuery* method), 19  
all() (*pytube.query.StreamQuery* method), 16  
apply\_descrambler() (in module *pytube.extract*), 20  
apply\_signature() (in module *pytube.extract*), 20  
asc() (*pytube.query.StreamQuery* method), 16  
author (*pytube.YouTube* attribute), 13

## C

cache() (in module *pytube.helpers*), 24  
Caption (class in *pytube*), 19  
caption\_tracks (*pytube.YouTube* attribute), 13  
CaptionQuery (class in *pytube.query*), 19  
captions (*pytube.YouTube* attribute), 13  
count() (*pytube.query.StreamQuery* method), 17

## D

default\_filename (*pytube.Stream* attribute), 15  
deprecated() (in module *pytube.helpers*), 24  
desc() (*pytube.query.StreamQuery* method), 17  
descramble() (*pytube.YouTube* method), 13  
description (*pytube.YouTube* attribute), 13  
download() (*pytube.Caption* method), 19  
download() (*pytube.Stream* method), 15

## E

ExtractError, 23

## F

filesize (in module *pytube.request*), 25  
filesize (*pytube.Stream* attribute), 15  
filesize\_approx (*pytube.Stream* attribute), 15  
filter() (*pytube.query.StreamQuery* method), 17  
first() (*pytube.query.StreamQuery* method), 17  
float\_to\_srt\_time\_format() (*pytube.Caption* static method), 19

## G

generate\_srt\_captions() (*pytube.Caption* method), 19  
get() (in module *pytube.request*), 25  
get() (*pytube.query.CaptionQuery* method), 19  
get\_audio\_only() (*pytube.query.StreamQuery* method), 17  
get\_by\_itag() (*pytube.query.StreamQuery* method), 18  
get\_by\_language\_code() (*pytube.query.CaptionQuery* method), 20  
get\_by\_resolution() (*pytube.query.StreamQuery* method), 18  
get\_highest\_resolution() (*pytube.query.StreamQuery* method), 18  
get\_initial\_function\_name() (in module *pytube.cipher*), 22  
get\_lowest\_resolution() (*pytube.query.StreamQuery* method), 18  
get\_transform\_map() (in module *pytube.cipher*), 22  
get\_transform\_object() (in module *pytube.cipher*), 22  
get\_transform\_plan() (in module *pytube.cipher*), 22  
get\_ytplayer\_config() (in module *pytube.extract*), 20

## H

head() (in module *pytube.request*), 25  
HTMLParseError, 24

## I

includes\_audio\_track (*pytube.Stream* attribute), 15  
includes\_video\_track (*pytube.Stream* attribute), 15  
index() (*pytube.query.StreamQuery* method), 18  
initialize\_stream\_objects() (*pytube.YouTube* method), 14

`is_adaptive` (*pytube.Stream attribute*), 15  
`is_age_restricted()` (*in module pytube.extract*), 21  
`is_progressive` (*pytube.Stream attribute*), 15  
`items()` (*pytube.query.CaptionQuery method*), 20

## J

`js_url()` (*in module pytube.extract*), 21

## K

`keys()` (*pytube.query.CaptionQuery method*), 20

## L

`last()` (*pytube.query.StreamQuery method*), 18  
`length` (*pytube.YouTube attribute*), 14  
`LiveStreamError`, 24

## M

`map_functions()` (*in module pytube.cipher*), 23  
`mime_type_codec()` (*in module pytube.extract*), 21

## O

`on_complete()` (*pytube.Stream method*), 16  
`on_progress()` (*pytube.Stream method*), 16  
`order_by()` (*pytube.query.StreamQuery method*), 18  
`otf()` (*pytube.query.StreamQuery method*), 18

## P

`parse_codecs()` (*pytube.Stream method*), 16  
`prefetch()` (*pytube.YouTube method*), 14  
`pytube` (*module*), 13  
`pytube.cipher` (*module*), 22  
`pytube.exceptions` (*module*), 23  
`pytube.extract` (*module*), 20  
`pytube.helpers` (*module*), 24  
`pytube.request` (*module*), 25  
`PytubeError`, 24  
`PytubeHTMLParser` (*class in pytube.extract*), 20

## R

`rating` (*pytube.YouTube attribute*), 14  
`regex_search()` (*in module pytube.helpers*), 24  
`RegexMatchError`, 24  
`register_on_complete_callback()` (*pytube.YouTube method*), 14  
`register_on_progress_callback()` (*pytube.YouTube method*), 14  
`reverse()` (*in module pytube.cipher*), 23

## S

`safe_filename()` (*in module pytube.helpers*), 24  
`setup_logger()` (*in module pytube.helpers*), 24  
`splice()` (*in module pytube.cipher*), 23

`Stream` (*class in pytube*), 15  
`stream()` (*in module pytube.request*), 25  
`stream_to_buffer()` (*pytube.Stream method*), 16  
`StreamQuery` (*class in pytube.query*), 16  
`streams` (*pytube.YouTube attribute*), 14  
`swap()` (*in module pytube.cipher*), 23

## T

`target_directory()` (*in module pytube.helpers*), 25  
`thumbnail_url` (*pytube.YouTube attribute*), 14  
`title` (*pytube.Stream attribute*), 16  
`title` (*pytube.YouTube attribute*), 14

## V

`values()` (*pytube.query.CaptionQuery method*), 20  
`video_id()` (*in module pytube.extract*), 21  
`video_info_url()` (*in module pytube.extract*), 21  
`video_info_url_age_restricted()` (*in module pytube.extract*), 21  
`VideoUnavailable`, 24  
`views` (*pytube.YouTube attribute*), 14

## X

`xml_caption_to_srt()` (*pytube.Caption method*), 19  
`xml_captions` (*pytube.Caption attribute*), 19

## Y

`YouTube` (*class in pytube*), 13