pytube Documentation

Release 9.5.2

Nick Ficano

Sep 10, 2019

Contents

Release v9.5.2. (*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:

```
>>> 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

CHAPTER 3

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.

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 pytube
```

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/nficano/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
$ 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

...
```

CHAPTER 4

The API Documentation / Guide

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=None*, *defer_prefetch_init=False*, *on_progress_callback=None*, *on_complete_callback=None*, *proxies=None*)

Core developer interface for pytube.

captions

Interface to query caption tracks.

Return type CaptionQuery.

description

Get the video description.

Return type str

init()

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

initialize_caption_objects()

Populate instances of *Caption*.

Take the unscrambled player response data, and use it to initialize instances of *Caption*.

Return type None

initialize_stream_objects(*fmt*)
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()
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

prefetch_init()
Download data, descramble it, and build Stream instances.

Return type None

rating
Get the video average rating.

Return type str

register_on_complete_callback(*func*)
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*)
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*, *player_config_args*, *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=None*, *filename=None*, *filename_prefix=None*)
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 seperate from filename so you can use the default filename but still add a prefix.

Return type str

filesize
File size of the media stream in bytes.

Return type int

Returns Filesize (in bytes) 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_handle*)
On download complete handler function.

Parameters **file_handle** (io.BufferedWriter) – The file handle where the media is being written to.

Return type None

on_progress(*chunk*, *file_handler*, *bytes_remaining*)
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** (*str*) – 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()
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.

set_attributes_from_dict(*dct*)
Set class attributes from dictionary items.

Return type None

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

Return type list

asc()
Sort streams in ascending order.

Return type *StreamQuery*

count()
Get the count the query would return.

Return type int

desc()
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*, *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).
- **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()
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_by_itag(*itag*)
Get the corresponding `Stream` for a given itag.

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

Return type `Stream` or None

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

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*)
Apply a sort order to a resultset.

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

4.1.4 Caption Object

class pytube.**Caption**(*caption_track*)
Container for caption tracks.

float_to_srt_time_format(*d*)
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()
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*)
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*)
Interface for querying the available captions.

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

Return type list

get_by_language_code(*lang_code*)
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.

4.1.6 Extract

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

pytube.extract.**get_ytplayer_config**(*html*, *age_restricted=False*)
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

- **watch_html** (*str*) – The html contents of the watch page.
- **age_restricted** (*bool*) – Is video age restricted.

Return type str

Returns Substring of the html containing the encoded manifest data.

pytube.extract.**is_age_restricted**(*watch_html*)
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*, *age_restricted=False*)
Get the base JavaScript url.

Construct the base JavaScript url, which contains the decipher "transforms".

Parameters

- **watch_html** (*str*) – The html contents of the watch page.
- **age_restricted** (*bool*) – Is video age restricted.

pytube.extract.**mime_type_codec**(*mime_type_codec*)
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*)
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*, *watch_url*, *watch_html*, *embed_html*, *age_restricted*)
Construct the video_info url.

Parameters

- **video_id** (*str*) – A YouTube video identifier.
- **watch_url** (*str*) – A YouTube watch url.
- **watch_html** (*str*) – The html contents of the watch page.
- **embed_html** (*str*) – The html contents of the embed page (for age restricted videos).
- **age_restricted** (*bool*) – Is video age restricted.

Return type str

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

`pytube.extract.watch_url`(*video_id*)
Construct a sanitized YouTube watch url, given a video id.

Parameters **video_id** (*str*) – A YouTube video identifier.

Return type str

Returns Sanitized YouTube watch url.

4.1.7 Cipher

This module countains 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*)
Extract the name of the function responsible for computing the signature.

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

`pytube.cipher.get_signature`(*js*, *ciphered_signature*)
Decipher the signature.

Taking the ciphered signature, applies the transform functions.

Parameters

- **js** (*str*) – The contents of the base.js asset file.
- **ciphered_signature** (*str*) – The ciphered signature sent in the `player_config`.

Return type str

Returns Decrypted signature required to download the media content.

pytube.cipher.**get_transform_map**(*js*, *var*)
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*, *var*)
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*)
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:

```
>>> get_transform_plan(js)
['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*)
For a given JavaScript transform function, return the Python equivalent.

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

pytube.cipher.**parse_function**(*js_func*)
Parse the Javascript transform function.

Break a JavaScript transform function down into a two element `tuple` containing the function name and some integer-based argument.

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

Return type tuple

Returns two element tuple containing the function name and an argument.

Example:

```
>>> parse_function('DE.AJ(a,15)')
('AJ', 15)
```

pytube.cipher.**reverse**(*arr*, *b*)

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*, *b*)

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*, *b*)

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**(*msg*, *video_id=None*)

Data extraction based exception.

exception pytube.exceptions.**LiveStreamError**(*msg*, *video_id=None*)

Video is a live stream.

exception pytube.exceptions.**PytubeError**

Base pytube exception that all others inherent.

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**(*msg*, *video_id=None*)
Regex pattern did not return any matches.

exception pytube.exceptions.**VideoUnavailable**
Video is unavailable.

4.1.9 Mixins

Applies in-place data mutations.

pytube.mixins.**apply_descrambler**(*stream_data*, *key*)
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

- **dct** (*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.mixins.**apply_signature**(*config_args*, *fmt*, *js*)
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.

4.1.10 Compat

Python 2/3 compatibility support.

pytube.compat.**install_proxy**(*proxy_handler*)
install global proxy. :param proxy_handler:

"http":"http://my.proxy.com:1234", "https":"https://my.proxy.com:1234"

Returns

pytube.compat.**unescape**(*s*)
Strip HTML entries from a string.

pytube.compat.**unicode**(*s*)
Encode a string to utf-8.

4.1.11 Helpers

Various helper functions implemented by pytube.

pytube.helpers.**apply_mixin**(*dct*, *key*, *func*, **args*, ***kwargs*)
Apply in-place data mutation to a dictionary.

Parameters

- **dct** (*dict*) – Dictionary to apply mixin function to.
- **key** (*str*) – Key within dictionary to apply mixin function to.
- **func** (*callable*) – Transform function to apply to `dct[key]`.
- ***args** – (optional) positional arguments that `func` takes.
- ****kwargs** – (optional) keyword arguments that `func` takes.

Return type None

pytube.helpers.**regex_search**(*pattern*, *string*, *groups=False*, *group=None*, *flags=0*)
Shortcut method to search a string for a given pattern.

Parameters

- **pattern** (*str*) – A regular expression pattern.
- **string** (*str*) – A target string to search.
- **groups** (*bool*) – Should the return value be `.groups()`.
- **group** (*int*) – Index of group to return.
- **flags** (*int*) – Expression behavior modifiers.

Return type str or tuple

Returns Substring pattern matches.

pytube.helpers.**safe_filename**(*s*, *max_length=255*)
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.

4.1.12 Request

Implements a simple wrapper around urlopen.

pytube.request.**get**(*url=None*, *headers=False*, *streaming=False*, *chunk_size=8192*)
Send an http GET request.

Parameters

- **url** (*str*) – The URL to perform the GET request for.
- **headers** (*bool*) – Only return the http headers.
- **streaming** (*bool*) – Returns the response body in chunks via a generator.
- **chunk_size** (*int*) – The size in bytes of each chunk.

pytube.request.**stream_response**(*response*, *chunk_size=8192*)

Read the response in chunks.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index