Please include the full output of youtube-dl when run with `-v`, i.e. **add `-v` flag to your command line**, copy the **whole** output and post it in the issue body wrapped in "‘ for better formatting. It should look similar to this:

```
$ youtube-dl -v <your command line>
[debug] System config: []
[debug] User config: []
[debug] Command-line args: [u'-v', u'https://www.youtube.com/watch?v=BaW_jenozKcj']
[debug] Encodings: locale cp1251, fs mbcs, out cp866, pref cp1251
[debug] youtube-dl version 2015.12.06
[debug] Git HEAD: 135392e
[debug] Python version 2.6.6 - Windows-2003Server-5.2.3790-SP2
[debug] exe versions: ffmpeg N-75573-g1d0487f, ffprobe N-75573-g1d0487f, rtmpdump 2.4
[debug] Proxy map: {}
...
```

Do not post screenshots of verbose logs; only plain text is acceptable.

The output (including the first lines) contains important debugging information. Issues without the full output are often not reproducible and therefore do not get solved in short order, if ever.

Please re-read your issue once again to avoid a couple of common mistakes (you can and should use this as a checklist):

Is the description of the issue itself sufficient?

We often get issue reports that we cannot really decipher. While in most cases we eventually get the required information after asking back multiple times, this poses an unnecessary drain on our resources. Many contributors, including myself, are also not native speakers, so we may misread some parts.

So please elaborate on what feature you are requesting, or what bug you want to be fixed. Make sure that it's obvious

- What the problem is
- How it could be fixed
- How your proposed solution would look like

If your report is shorter than two lines, it is almost certainly missing some of these, which makes it hard for us to respond to it. We're often too polite to close the issue outright, but the missing info makes misinterpretation likely. As a committer myself, I often get frustrated by these issues, since the only possible way for me to move forward on them is to ask for clarification over and over.

For bug reports, this means that your report should contain the *complete* output of youtube-dl when called with the `-v` flag. The error message you get for (most) bugs even says so, but you would not believe how many of our bug reports do not contain this information.

If your server has multiple IPs or you suspect censorship, adding `--call-home` may be a good idea to get more diagnostics. If the error is `ERROR: Unable to extract ...` and you cannot reproduce it from multiple countries, add `--dump-pages` (warning: this will yield a rather large output, redirect it to the file `log.txt` by adding `>log.txt 2>&1` to your command-line) or upload the `.dump` files you get when you add `--write-pages` somewhere.

Site support requests must contain an example URL. An example URL is a URL you might want to download, like `https://www.youtube.com/watch?v=BaW_jenozKc`. There should be an obvious video present. Except under very special circumstances, the main page of a video service (e.g. `https://www.youtube.com/`) is *not* an example URL.

Are you using the latest version?

Before reporting any issue, type `youtube-dl -U`. This should report that you're up-to-date. About 20% of the reports we receive are already fixed, but people are using outdated versions. This goes for feature requests as well.

Is the issue already documented?

Make sure that someone has not already opened the issue you're trying to open. Search at the top of the window or browse the GitHub Issues of this repository. If there is an issue, feel free to write something along the lines of "This affects me as well, with version 2015.01.01. Here is some more information on the issue: ...". While some issues may be old, a new post into them often spurs rapid activity.

Why are existing options not enough?

Before requesting a new feature, please have a quick peek at the list of supported options. Many feature requests are for features that actually exist already! Please, absolutely do show off your work in the issue report and detail how the existing similar options do *not* solve your problem.

Is there enough context in your bug report?

People want to solve problems, and often think they do us a favor by breaking down their larger problems (e.g. wanting to skip already downloaded files) to a specific request (e.g. requesting us to look whether the file exists before downloading the info page). However, what often happens is that they break down the problem into two steps: One simple, and one impossible (or extremely complicated one).

We are then presented with a very complicated request when the original problem could be solved far easier, e.g. by recording the downloaded video IDs in a separate file. To avoid this, you must include the greater context where it is non-obvious. In particular, every feature request that does not consist of adding support for a

new site should contain a use case scenario that explains in what situation the missing feature would be useful.

Does the issue involve one problem, and one problem only?

Some of our users seem to think there is a limit of issues they can or should open. There is no limit of issues they can or should open. While it may seem appealing to be able to dump all your issues into one ticket, that means that someone who solves one of your issues cannot mark the issue as closed. Typically, reporting a bunch of issues leads to the ticket lingering since nobody wants to attack that behemoth, until someone mercifully splits the issue into multiple ones.

In particular, every site support request issue should only pertain to services at one site (generally under a common domain, but always using the same backend technology). Do not request support for vimeo user videos, White house podcasts, and Google Plus pages in the same issue. Also, make sure that you don't post bug reports alongside feature requests. As a rule of thumb, a feature request does not include outputs of youtube-dl that are not immediately related to the feature at hand. Do not post reports of a network error alongside the request for a new video service.

Is anyone going to need the feature?

Only post features that you (or an incapacitated friend you can personally talk to) require. Do not post features because they seem like a good idea. If they are really useful, they will be requested by someone who requires them.

Is your question about youtube-dl?

It may sound strange, but some bug reports we receive are completely unrelated to youtube-dl and relate to a different, or even the reporter's own, application. Please make sure that you are actually using youtube-dl. If you are using a UI for youtube-dl, report the bug to the maintainer of the actual application providing the UI. On the other hand, if your UI for youtube-dl fails in some way you believe is related to youtube-dl, by all means, go ahead and report the bug.

DEVELOPER INSTRUCTIONS

Most users do not need to build youtube-dl and can download the builds or get them from their distribution.

To run youtube-dl as a developer, you don't need to build anything either. Simply execute

```
python -m youtube_dl
```

To run the test, simply invoke your favorite test runner, or execute a test file directly; any of the following work:

```
python -m unittest discover
python test/test_download.py
nosetests
```

See item 6 of new extractor tutorial for how to run extractor specific test cases.

If you want to create a build of youtube-dl yourself, you'll need

- python
- make (only GNU make is supported)
- pandoc
- zip
- nosetests

Adding support for a new site

If you want to add support for a new site, first of all **make sure** this site is **not dedicated to copyright infringement**. youtube-dl does **not support** such sites thus pull requests adding support for them **will be rejected**.

After you have ensured this site is distributing its content legally, you can follow this quick list (assuming your service is called `yourextractor`):

1. Fork this repository
2. Check out the source code with:

   ```
   git clone git@github.com:YOUR_GITHUB_USERNAME/youtube-dl.git
   ```

3. Start a new git branch with

   ```
   cd youtube-dl
   git checkout -b yourextractor
   ```

4. Start with this simple template and save it to `youtube_dl/extractor/yourextractor.py`:

   ```
   # coding: utf-8
   from __future__ import unicode_literals

   from .common import InfoExtractor


   class YourExtractorIE(InfoExtractor):
       _VALID_URL = r'https?://(?:www\.)?yourextractor\.com/watch/(?P<id>[0-9]+)'
       _TEST = {
           'url': 'https://yourextractor.com/watch/42',
           'md5': 'TODO: md5 sum of the first 10241 bytes of the video file (use --test)',
           'info_dict': {
               'id': '42',
               'ext': 'mp4',
               'title': 'Video title goes here',
   ```

```
            'thumbnail': r're:^https?://.*\.jpg$',
            # TODO more properties, either as:
            # * A value
            # * MD5 checksum; start the string with md5:
            # * A regular expression; start the string with re:
            # * Any Python type (for example int or float)
        }
    }

    def _real_extract(self, url):
        video_id = self._match_id(url)
        webpage = self._download_webpage(url, video_id)

        # TODO more code goes here, for example ...
        title = self._html_search_regex(r'<h1>(.+?)</h1>', webpage, 'title')

        return {
            'id': video_id,
            'title': title,
            'description': self._og_search_description(webpage),
            'uploader': self._search_regex(r'<div[^>]+id="uploader"[^>]*>([^<]+)<', web
            # TODO more properties (see youtube_dl/extractor/common.py)
        }
```

5. Add an import in `youtube_dl/extractor/extractors.py`.
6. Run `python test/test_download.py TestDownload.test_YourExtractor`. This *should fail* at first, but you can continually re-run it until you're done. If you decide to add more than one test, then rename `_TEST` to `_TESTS` and make it into a list of dictionaries. The tests will then be named `TestDownload.test_YourExtractor`, `TestDownload.test_YourExtractor_1`, `TestDownload.test_YourExtractor_2`, etc. Note that tests with `only_matching` key in test's dict are not counted in.
7. Have a look at `youtube_dl/extractor/common.py` for possible helper methods and a detailed description of what your extractor should and may return. Add tests and code for as many as you want.
8. Make sure your code follows youtube-dl coding conventions and check the code with flake8:

   ```
   $ flake8 youtube_dl/extractor/yourextractor.py
   ```

9. Make sure your code works under all Python versions claimed supported by youtube-dl, namely 2.6, 2.7, and 3.2+.
10. When the tests pass, add the new files and commit them and push the result, like this:

```
$ git add youtube_dl/extractor/extractors.py
$ git add youtube_dl/extractor/yourextractor.py
$ git commit -m '[yourextractor] Add new extractor'
$ git push origin yourextractor
```

11. Finally, create a pull request. We'll then review and merge it.

In any case, thank you very much for your contributions!

youtube-dl coding conventions

This section introduces a guide lines for writing idiomatic, robust and future-proof extractor code.

Extractors are very fragile by nature since they depend on the layout of the source data provided by 3rd party media hosters out of your control and this layout tends to change. As an extractor implementer your task is not only to write code that will extract media links and metadata correctly but also to minimize dependency on the source's layout and even to make the code foresee potential future changes and be ready for that. This is important because it will allow the extractor not to break on minor layout changes thus keeping old youtube-dl versions working. Even though this breakage issue is easily fixed by emitting a new version of youtube-dl with a fix incorporated, all the previous versions become broken in all repositories and distros' packages that may not be so prompt in fetching the update from us. Needless to say, some non rolling release distros may never receive an update at all.

Mandatory and optional metafields

For extraction to work youtube-dl relies on metadata your extractor extracts and provides to youtube-dl expressed by an information dictionary or simply *info dict*. Only the following meta fields in the *info dict* are considered mandatory for a successful extraction process by youtube-dl:

- `id` (media identifier)
- `title` (media title)
- `url` (media download URL) or `formats`

In fact only the last option is technically mandatory (i.e. if you can't figure out the download location of the media the extraction does not make any sense). But by convention youtube-dl also treats `id` and `title` as mandatory. Thus the aforementioned metafields are the critical data that the extraction does not make any sense without and if any of them fail to be extracted then the extractor is considered completely broken.

Any field apart from the aforementioned ones are considered **optional**. That means that extraction should be **tolerant** to situations when sources for these fields can potentially be unavailable (even if they are always available at the

moment) and **future-proof** in order not to break the extraction of general purpose mandatory fields.

Example Say you have some source dictionary `meta` that you've fetched as JSON with HTTP request and it has a key `summary`:

```
meta = self._download_json(url, video_id)
```

Assume at this point `meta`'s layout is:

```
{
    ...
    "summary": "some fancy summary text",
    ...
}
```

Assume you want to extract `summary` and put it into the resulting info dict as `description`. Since `description` is an optional meta field you should be ready that this key may be missing from the `meta` dict, so that you should extract it like:

```
description = meta.get('summary')  # correct
```

and not like:

```
description = meta['summary']  # incorrect
```

The latter will break extraction process with `KeyError` if `summary` disappears from `meta` at some later time but with the former approach extraction will just go ahead with `description` set to `None` which is perfectly fine (remember `None` is equivalent to the absence of data).

Similarly, you should pass `fatal=False` when extracting optional data from a webpage with `_search_regex`, `_html_search_regex` or similar methods, for instance:

```
description = self._search_regex(
    r'<span[^>]+id="title"[^>]*>([^<]+)<',
    webpage, 'description', fatal=False)
```

With `fatal` set to `False` if `_search_regex` fails to extract `description` it will emit a warning and continue extraction.

You can also pass `default=<some fallback value>`, for example:

```
description = self._search_regex(
    r'<span[^>]+id="title"[^>]*>([^<]+)<',
    webpage, 'description', default=None)
```

On failure this code will silently continue the extraction with `description` set to `None`. That is useful for metafields that may or may not be present.

Provide fallbacks

When extracting metadata try to do so from multiple sources. For example if `title` is present in several places, try extracting from at least some of them. This makes it more future-proof in case some of the sources become unavailable.

Example Say `meta` from the previous example has a `title` and you are about to extract it. Since `title` is a mandatory meta field you should end up with something like:

```
title = meta['title']
```

If `title` disappears from `meta` in future due to some changes on the hoster's side the extraction would fail since `title` is mandatory. That's expected.

Assume that you have some another source you can extract `title` from, for example `og:title` HTML meta of a `webpage`. In this case you can provide a fallback scenario:

```
title = meta.get('title') or self._og_search_title(webpage)
```

This code will try to extract from `meta` first and if it fails it will try extracting `og:title` from a `webpage`.

Regular expressions

Don't capture groups you don't use Capturing group must be an indication that it's used somewhere in the code. Any group that is not used must be non capturing.

Example Don't capture id attribute name here since you can't use it for anything anyway.

Correct:

```
r'(?:id|ID)=(?P<id>\d+)'
```

Incorrect:

```
r'(id|ID)=(?P<id>\d+)'
```

Make regular expressions relaxed and flexible When using regular expressions try to write them fuzzy, relaxed and flexible, skipping insignificant parts that are more likely to change, allowing both single and double quotes for quoted values and so on.

Example Say you need to extract `title` from the following HTML code:

```
<span style="position: absolute; left: 910px; width: 90px; float: right; z-index: 9999;" cla
```

The code for that task should look similar to:

```
title = self._search_regex(
    r'<span[^>]+class="title"[^>]*>([^<]+)', webpage, 'title')
```

Or even better:

```
title = self._search_regex(
    r'<span[^>]+class=(["\'])title\1[^>]*>(?P<title>[^<]+)',
    webpage, 'title', group='title')
```

Note how you tolerate potential changes in the `style` attribute's value or switch from using double quotes to single for `class` attribute:

The code definitely should not look like:

```
title = self._search_regex(
    r'<span style="position: absolute; left: 910px; width: 90px; float: right; z-index: 999
    webpage, 'title', group='title')
```

Long lines policy

There is a soft limit to keep lines of code under 80 characters long. This means it should be respected if possible and if it does not make readability and code maintenance worse.

For example, you should **never** split long string literals like URLs or some other often copied entities over multiple lines to fit this limit:

Correct:

```
'https://www.youtube.com/watch?v=FqZTN594JQw&list=PLMYEtVRpaqY00V9W81Cwmzp6N6vZqfUKD4'
```

Incorrect:

```
'https://www.youtube.com/watch?v=FqZTN594JQw&list='
'PLMYEtVRpaqY00V9W81Cwmzp6N6vZqfUKD4'
```

Inline values

Extracting variables is acceptable for reducing code duplication and improving readability of complex expressions. However, you should avoid extracting variables used only once and moving them to opposite parts of the extractor file, which makes reading the linear flow difficult.

Example Correct:

```
title = self._html_search_regex(r'<title>([^<]+)</title>', webpage, 'title')
```

Incorrect:

```
TITLE_RE = r'<title>([^<]+)</title>'
# ...some lines of code...
title = self._html_search_regex(TITLE_RE, webpage, 'title')
```

Collapse fallbacks

Multiple fallback values can quickly become unwieldy. Collapse multiple fallback values into a single expression via a list of patterns.

Example Good:

```
description = self._html_search_meta(
    ['og:description', 'description', 'twitter:description'],
    webpage, 'description', default=None)
```

Unwieldy:

```
description = (
    self._og_search_description(webpage, default=None)
    or self._html_search_meta('description', webpage, default=None)
    or self._html_search_meta('twitter:description', webpage, default=None))
```

Methods supporting list of patterns are: `_search_regex`, `_html_search_regex`, `_og_search_property`, `_html_search_meta`.

Trailing parentheses

Always move trailing parentheses after the last argument.

Example Correct:

```
    lambda x: x['ResultSet']['Result'][0]['VideoUrlSet']['VideoUrl'],
    list)
```

Incorrect:

```
    lambda x: x['ResultSet']['Result'][0]['VideoUrlSet']['VideoUrl'],
    list,
)
```

Use convenience conversion and parsing functions

Wrap all extracted numeric data into safe functions from `youtube_dl/utils.py`: `int_or_none`, `float_or_none`. Use them for string to number conversions as well.

Use `url_or_none` for safe URL processing.

Use `try_get` for safe metadata extraction from parsed JSON.

Use `unified_strdate` for uniform `upload_date` or any `YYYYMMDD` meta field extraction, `unified_timestamp` for uniform `timestamp` extraction, `parse_filesize` for `filesize` extraction, `parse_count` for count meta fields extraction, `parse_resolution`, `parse_duration` for `duration` extraction, `parse_age_limit` for `age_limit` extraction.

Explore `youtube_dl/utils.py` for more useful convenience functions.

More examples

Safely extract optional description from parsed JSON

```
description = try_get(response, lambda x: x['result']['video'][0]['summary'], compat_str)
```

Safely extract more optional metadata

```
video = try_get(response, lambda x: x['result']['video'][0], dict) or {}
description = video.get('summary')
duration = float_or_none(video.get('durationMs'), scale=1000)
view_count = int_or_none(video.get('views'))
```