Grab Documentation

Release 0.6

Gregory Petukhov

Jan 29, 2019
1 Grab Web Resources 1
2 What is Grab? 3
3 Table of Contents 5
   3.1 Grab User Manual ................................................. 5
   3.2 Grab::Spider User Manual ....................................... 36
   3.3 API Reference .................................................... 46
4 Indices and tables 57
Python Module Index 59
CHAPTER 1

Grab Web Resources

- **Source code:** https://github.com/lorien/grab
- **Documentation:** http://grablib.org
- **Telegram chat:** https://t.me/grablab
Grab is a python framework for building web scrapers. With Grab you can build web scrapers of various complexity, from simple 5-line scripts to complex asynchronous website crawlers processing millions of web pages. Grab provides an API for performing network requests and for handling the received content e.g. interacting with DOM tree of the HTML document.

There are two main parts in the Grab library:

1) The single request/response API that allows you to build network request, perform it and work with the received content. The API is a wrapper of the pycurl and lxml libraries.

2) The Spider API to build asynchronous web crawlers. You write classes that define handlers for each type of network request. Each handler is able to spawn new network requests. Network requests are processed concurrently with a pool of asynchronous web sockets.
Chapter 2. What is Grab?
3.1 Grab User Manual

3.1.1 Grab Installation

Common problems

1) If you got the error `requirement specifiers; Expected version spec ...` while installing grab via the pip then you have outdated setuptools or pip packages.

Fix the error with following command:

```
pip install -U setuptools pip
```

2) If you got out of memory error while installing lxml on linux machine with 512Mb RAM or less then check that you swap file is enabled.

Installation on Linux

Update installation tools:

```
pip install -U setuptools pip
```

Run the command:

```
pip install -U grab
```

This command will install Grab and all dependencies. Be aware that you need to have some libraries installed in your system to successfully build lxml and pycurl dependencies.

To build pycurl successfully you need to install:
apt-get install libcurl4-openssl-dev

To build lxml successfully you need to install:

apt-get install libxml2-dev libxslt-dev

If your system has 512Mb RAM or less you might experience issues during installation of Grab dependencies. Installation of lxml requires quite a few RAM. In such case enable swap file if it is disabled.

**Installation on Windows**

Step 1. Install lxml. You can get lxml here [http://www.lfd.uci.edu/~gohlke/pythonlibs/#lxml](http://www.lfd.uci.edu/~gohlke/pythonlibs/#lxml)

Step 2. Install pycurl.

**Warning**: Do not use the recent version of pycurl (7.43.0 at the moment). This version fails randomly on windows platform. Use 7.19.5.3 version. You can get it here [https://bintray.com/pycurl/pycurl/pycurl/view#files](https://bintray.com/pycurl/pycurl/pycurl/view#files)

Step 3. Update installation tools

```
python -m pip install -U pip setuptools
```

If you don’t have pip installed, install pip first. Download the file get-pip.py from [https://bootstrap.pypa.io/get-pip.py](https://bootstrap.pypa.io/get-pip.py) and then run the command

```
python get-pip.py
```

Step 4. Install Grab

Now you can install Grab via pip with this command:

```
python -m pip install grab
```

**Installation on FreeBSD**

Update installation tools:

```
pip install -U setuptools pip
```

Run the command:

```
pip install -U grab
```

You can also install Grab from FreeBSD ports (thanks to Ruslan Makhmatkhanov):

- To install the port: cd /usr/ports/devel/py-grab/ && make install clean
- To add the package: pkg_add -r py27-grab

**Installation on MacOS**

Update installation tools:
pip install -U setuptools pip

Run the command:

pip install -U grab

**Dependencies**

All required dependencies should be installed automatically if you install Grab with pip. Here is list of Grab dependencies:

- lxml
- pycurl
- selection
- weblib
- six
- user_agent

**Upgrade Grab from 0.5.x version to 0.6.x**

In Grab 0.6.x some features were moved out into separate packages. If you install/upgrade Grab with pip, all dependencies should be installed automatically. Anyway, if you have some ImportError issues then try to install dependencies manually with the command:

```bash
pip install -U lxml pycurl selection weblib six user_agent
```

### 3.1.2 Testing Grab Framework

To run the full set of Grab tests you should install the additional dependencies listed in `requirements_dev.txt` and `requirements_dev_backend.txt`.

To run all tests run the command:

```bash
./runtest.py --test-all --backend-mongo --backend-mysql --backend-redis --backend-postgres
```

**Controlling what parts of Grab to test**

You can run tests for specific parts of the Grab framework. Here are the available options for `runtest.py`:

- `--test-grab` - Grab API tests
- `--test-spider` - Grab::Spider API tests
- `--test-all` - shortcut to run both Grab and Grab::Spider tests
- `--backend-redis` - enable tests of things that work with redis
- `--backend-mysql` - enable tests of things that work with mysql
- `--backend-postgres` - enable tests of things that work with postgres
- `--backend-mongo` - enable tests of things that work with mongo

If you want to run specific test cases then use the `-t` option. For example:
./runtest.py -t test.grab_api

Testing with Tox

To run Grab tests in different python environments you can use tox command:

```
 tox
```

By default it will run the full set of tests in two environments: python3.4 and python2.7

You can specify a specific environment with `-e` option:

```
 tox -e py34
```

To run all tests except backend tests, use `-nobackend` suffix:

```
 tox -e py34-nobackend,py27-nobackend
```

Database configuration

To run tests on a specific machine you may need to change the default database connection settings. Default settings are stored in the `test_settings.py` file. You can override any default setting in the `test_settings_local.py` file.

Travis Testing

The Grab project is configured to run the full set of tests for each new commit placed into the project repository. You can see the status of a recent commit and the status of all previous commits here: https://travis-ci.org/lorien/grab

Test Coverage

To see test coverage run the commands:

```
 coverage erase
 coverage run --source=grab ./runtest.py --test-all --backend-mongo --backend-mysql --backend-redis --backend-postgres
 coverage report -m
```

Also you can use shortcut:

```
 make coverage
```

The Grab project is configured to submit coverage statistics to the coveralls.io service after each test session is completed by travis-ci. You can see the coverage history at this URL: https://coveralls.io/r/lorien/grab

3.1.3 Grab Quickstart

Before working with Grab ensure that you have the latest version. The recommended way of installing Grab is by using pip:

```
 pip install -U Grab
```
You should also manually install the lxml and pycurl libraries.

Let’s get started with some simple examples.

**Make a request**

First, you need to import the Grab class:

```python
>>> from grab import Grab
```

Then you can build Grab instances and make simple network requests:

```python
>>> from grab import Grab
>>> g = Grab()
>>> resp = g.go('http://livejournal.com/)
```

Now, we have a `Response` object which provides an interface to the response's content, cookies, headers and other things.

We’ve just made a GET request. To make other request types, you need to configure the Grab instance via the `setup` method with the `method` argument:

```python
>>> g.setup(method='put')
>>> g.setup(method='delete')
>>> g.setup(method='options')
>>> g.setup(method='head')
```

Let’s see a small example of HEAD request:

```python
>>> g = Grab()
>>> g.setup(method='head')
>>> resp = g.go('http://google.com/robots.txt')
>>> print len(resp.body)
0
>>> print resp.headers['Content-Length']
1776
```

**Creating POST requests**

When you build site scrapers or work with network APIs it is a common task to create POST requests. You can build POST request using the `post` option:

```python
>>> g = Grab()
>>> g.setup(post={'username': 'Root', 'password': 'asd7DD&*ssd'})
>>> g.go('http://example.com/log-in-form')
```

Another common task is to get a web form, fill it in and submit it. Grab provides an easy way to work with forms:

```python
>>> g = Grab()
>>> g.go('http://example.com/log-in')
>>> g.set_input('username', 'Foo')
>>> g.set_input('password', 'Bar')
>>> g.submit()
```

When you call `submit`, Grab will build a POST request using the values passed in via `set_input`. If you did not specify values for some form elements then Grab will use their default values.
Grab also provides an interface to upload files:

```python
>>> from grab import Grab, UploadFile
>>> g = Grab()
>>> g.setup(post={'name': 'Flower', 'file': UploadFile('/path/to/image.png')})
>>> g.submit()
```

Also you can upload files via the form API:

```python
>>> from grab import Grab, UploadFile
>>> g = Grab()
>>> g.go('http://example.com/some-form')
>>> g.set_input('name', 'A flower')
>>> g.set_input('file', UploadFile('/path/to/image.png'))
>>> g.submit()
```

**Response Content**

Consider a simple page retrieving example:

```python
>>> g = Grab()
>>> resp = g.go('http://google.com/')
```

To get the response content as unicode use:

```python
>>> resp.unicode_body()
```

Note that grab will automatically detect the character set (charset for short) of the response content and convert it to unicode. You can detected the charset with:

```python
>>> resp.charset
```

If you need the original response body then use:

```python
>>> resp.body
```

Original content is useful if you need to save a binary file (e.g. an image):

```python
>>> resp = g.go('http://example.com/some-log.png')
>>> open('logo.png', 'w').write(resp.body)
```

The gzip and deflate encodings are automatically decoded.

**Response Status Code**

TO BE CONTINUED

### 3.1.4 Request Methods

You can make any type of HTTP request you need. By default Grab will make a GET request.
GET Request

GET is the default request method.

```
g = Grab()
g.go('http://example.com/')
```

If you need to pass arguments in through the query string, then you have to build the URL manually:

```
from urllib import urlencode

g = Grab()
qs = urlencode({'foo': 'bar', 'arg': 'val'})
g.go('http://dumpz.org/?%s % qs)
```

If your URL contains unsafe characters then you must escape them manually.

```
from urllib import quote

g = Grab()
url = u'https://ru.wikipedia.org/wiki/
g.go(quote(url.encode('utf-8')))```

POST Request

To make a POST request you have to specify POST data with the `post` option. Usually, you will want to use a dictionary:

```
g = Grab()
g.go('http://example.com/', post={'foo': 'bar'})
```

You can pass unicode strings and numbers in as values for the `post` dict, they will be converted to byte strings automatically. If you want to specify a charset that will be used to convert unicode to byte string, then use `request_charset` option.

```
g = Grab()
g.go('http://example.com/', post={'who': u' '}, charset='cp1251')
```

If the `post` option is a string then it is submitted as-is:

```
g = Grab()
g.go('http://example.com/', post='raw data')
```

If you want to pass multiple values with the same key use the list of tuples version:

```
g = Grab()
g.go('http://example.com/', post=[('key', 'val1'), ('key', 'val2')])
```

By default, Grab will compose a POST request with 'application/x-www-form-urlencoded' encoding method. To enable `multipart/form-data` use the `post_multipart` argument instead of `post`:

```
g = Grab()
g.go('http://example.com/', multipart_post=[('key', 'val1'), ('key', 'val2')])
```
To upload a file, use the `grab.upload.UploadFile` class:

```python
g = Grab()
g.go('http://example.com/',
    multipart_post={'foo': 'bar', 'file': UploadFile('/path/to/file')})
```

**PUT Request**

To make a PUT request use both the `post` and `method` arguments:

```python
g = Grab()
g.go('http://example.com/', post='raw data', method='put')
```

**Other Methods**

To make DELETE, OPTIONS and other HTTP requests, use the `method` option.

```python
g = Grab()
g.go('http://example.com/', method='options')
```

### 3.1.5 Setting up the Grab Request

To set up specific parameters of a network request you need to build the Grab object and configure it. You can do both things at the same time:

```python
g = Grab(url='http://example.com/', method='head', timeout=10,
          user_agent='grab')
g.request()
```

Or you can build the Grab object with some initial settings and configure it later:

```python
g = Grab(timeout=10, user_agent='grab')
g.setup(url='http://example.com', method='head')
g.request()
```

Also you can pass settings as parameters to `request` or `go`:

```python
g = Grab(timeout=10)
g.setup(method='head')
g.request(url='http://example.com', user_agent='grab')
# OR
g.go('http://example.com', user_agent='grab')
```

`request` and `go` are almost same except for one small thing. You do not need to specify the explicit name of the first argument with `go`. The first argument of the `go` method is always `url`. Except for this, all other named arguments of `go` and `request` are just passed to the `setup` function.

For a full list of available settings you can check *Grab Settings*
Grab Config Object

Every time you configure a Grab instance, all options are saved in the special object, `grab.config`, that holds all Grab instance settings. You can receive a copy of the config object and also you can setup a Grab instance with the config object:

```python
>>> g = Grab(url='http://google.com/
>>> g.config['url']
'http://google.com/
>>> config = g.dump_config()
>>> g2 = Grab()
>>> g2.load_config(config)
>>> g2.config['url']
'http://google.com/
```

The Grab config object is simply a `dict` object. Some of the values may also be a `dict`.

Grab Instance Cloning

If you need to copy a Grab object there is a more elegant way than using the `dump_config` and `load_config` methods:

```python
g2 = g1.clone()
```

`g2` gets the same state as `g1`. In particular, `g2` will have the same cookies.

There is also `adopt`, which does the opposite of the `clone` method:

```python
g2.adopt(g1)
```

The `g2` instance receives the state of the `g1` instance.

Setting Up the Pycurl Object

Sometimes you need more detailed control of network requests than Grab allows. In such cases you can configure pycurl directly. All Grab’s network features are only a wrapper to the pycurl library. Any available Grab option just sets some option of the underlying pycurl object. Here is a simple example of how to change the type of the HTTP authentication:

```python
import pycurl
from grab import Grab
g = Grab()
g.setup(userpwd='root:123')
g.transport.curl.setopt(pycurl.HTTPAUTH, pycurl.HTTPAUTH_NTLM)
```

3.1.6 Grab Settings

Network options

url

- Type: string
- Default: None
The URL of the requested web page. You can use relative URLs, in which case Grab will build the absolute url by joining the relative URL with the URL or previously requested document. Be aware that Grab does not automatically escape unsafe characters in the URL. This is a design feature. You can use `urllib.quote` and `urllib.quote_plus` functions to make your URLs safe.

More info about valid URLs is in RFC 2396.

**timeout**

    Type  int
    Default  15

Maximum time for a network operation. If it is exceeded, GrabNetworkTimeout is raised.

**connect_timeout**

    Type  int
    Default  3

Maximum time for connection to the remote server and receipt of an initial response. If it is exceeded, GrabNetworkTimeout is raised.

**follow_refresh**

    Type  bool
    Default  False

Automatically follow the URL specified in `<meta http-equiv="refresh">` tag.

**follow_location**

    Type  bool
    Default  True

Automatically follow the location in 301/302 response.

**interface**

    Type  string
    Default  None

The network interface through which the request should be submitted.

To specify the interface by its OS name, use “if!***” format, e.g. “if/eth0”. To specify the interface by its name or ip address, use “host!***” format, e.g. “host!127.0.0.1” or “host!localhost”.

See also the pycurl manual: [http://curl.haxx.se/libcurl/c/curl_easy_setopt.html#CURLOPTINTERFACE](http://curl.haxx.se/libcurl/c/curl_easy_setopt.html#CURLOPTINTERFACE)
redirect_limit

Type  int
Default  10

Set the maximum number of redirects that Grab will do for one request. Redirects follow the “Location” header in 301/302 network responses, and also follow the URL specified in meta refresh tags.

userpwd

Type  string
Default  None

The username and the password to send during HTTP authorization. The value of that options is the string of format “username:password”.

HTTP Options

user_agent

Type  string
Default  see below

Sets the content of the “User-Agent” HTTP-header. By default, Grab randomly chooses a user agent from the list of real user agents that is built into Grab itself.

user_agent_file

Type  string
Default  None

Path to the text file with User-Agent strings. If this option is specified, then Grab randomly chooses one line from that file.

method

Type  string
Default  “GET”

Possible values  “GET”, “POST”, “PUT”, “DELETE”

The HTTP request method to use. By default, GET is used. If you specify post or multipart_post options, then Grab automatically changes the method to POST.

post

Type  sequence of pairs or dict or string
Default  None
Data to be sent with the POST request. Depending on the type of data, the corresponding method of handling that data is selected. The default type for POST requests is “application/x-www-form-urlencoded”.

In case of `dict` or sequence of pairs, the following algorithm is applied to each value:

- objects of `grab.upload.UploadFile` class are converted into pycurl structures
- unicode strings are converted into byte strings
- None values are converted into empty strings

If `post` value is just a string, then it is placed into the network request without any modification.

**multipart_post**

Type: sequence of pairs or `dict`  
Default: None

Data to be sent with the POST request. This option forces the POST request to be in “multipart/form-data” form.

**headers**

Type: `dict`  
Default: None

Additional HTTP-headers. The value of this option will be added to headers that Grab generates by default. See details in *Work with HTTP Headers*.

**common_headers**

Type: `dict`  
Default: None

By default, Grab generates some common HTTP headers to mimic the behaviour of a real web browser. If you have trouble with these default headers, you can specify your own headers with this option. Please note that the usual way to specify a header is to use the `headers` option. See details in *Work with HTTP Headers*.

**reuse_cookies**

Type: `bool`  
Default: True

If this option is enabled, then all cookies in each network response are stored locally and sent back with further requests to the same server.

**cookies**

Type: `dict`  
Default: None

Cookies to send to the server. If the option `reuse_cookies` is also enabled, then cookies from the `cookies` option will be joined with stored cookies.
**cookiefile**

*Type* string

*Default* None

Before each request, Grab will read cookies from this file and join them with stored cookies. After each response, Grab will save all cookies to that file. The data stored in the file is a dict serialized as JSON.

**referer**

*Type* string

*Default* see below

The content of the “Referer” HTTP-header. By default, Grab builds this header with the URL of the previously requested document.

**reuse_referer**

*Type* bool

*Default* True

If this option is enabled, then Grab uses the URL of the previously requested document to build the content of the “Referer” HTTP header.

**Proxy Options**

**proxy**

*Type* string

*Default* None

The address of the proxy server, in either “domain:port” or “ip:port” format.

**proxy_userpwd**

*Type* string

*Default* None

Security data to submit to the proxy if it requires authentication. Form of data is “username:password”

**proxy_type**

*Type* string

*Default* None

Type of proxy server. Available values are “http”, “socks4” and “socks5”.
proxy_auto_change

Type bool
Default True

If Grab should change the proxy before every network request.

Response Processing Options

encoding

Type string
Default “gzip”

List of methods that the remote server could use to compress the content of its response. The default value of this option is “gzip”. To disable all compression, pass the empty string to this option.

document_charset

Type string
Default None

The character set of the document’s content. By default Grab detects the charset of the document automatically. If it detects the charset incorrectly you can specify exact charset with this option. The charset is used to get unicode representation of the document content and also to build DOM tree.

charset

Type string
Default ‘utf-8’

To send a request to the network Grab should convert all unicode data into bytes. It uses the charset for encoding. For example:

```python
g.setup(post=b"abc")
```

no conversion required. But if

```python
g.setup(post='", !')
```

then the unicode data has to be converted to charset encoding. By default that would be utf-8.

nobody

Type bool
Default False

Ignore the body of the network response. When this option is enabled, the connection is abandoned at the moment when remote server transfers all response headers and begins to transfer the body of the response. You can use this option with any HTTP method.
**body_maxsize**

_TYPE_ int

_Default_ None

A limit on the maximum size of data that should be received from the remote server. If the limit is reached, the connection is abandoned and you can work with the data received so far.

**lowercased_tree**

_TYPE_ bool

_Default_ False

Convert the content of the document to lowercase before passing it to the lxml library to build the DOM tree. This option does not affect the content of `response.body`, which always stores the original data.

**strip_null_bytes**

_TYPE_ bool

_Default_ True

Control the removal of null bytes from the body of HTML documents before they are passed to lxml to build a DOM tree. lxml stops processing HTML documents at the first place where it finds a null byte. To avoid such issues Grab, removes null bytes from the document body by default. This option does not affect the content of `response.body` that always stores the original data.

**body_inmemory**

_TYPE_ bool

_Default_ True

Control the way the network response is received. By default, Grab downloads data into memory. To handle large files, you can set `body_inmemory=False` to download the network response directly to the disk.

**body_storage_dir**

_TYPE_ bool

_Default_ None

If you use `body_inmemory=False`, then you have to specify the directory where Grab will save network requests.

**body_storage_filename**

_TYPE_ string

_Default_ None
If you use `body_inmemory=False`, you can let Grab automatically choose names for the files where it saves network responses. By default, Grab randomly builds unique names for files. With the `body_storage_filename` option, you can choose the exact file name to save response to. Note that Grab will save every response to that file, so you need to change the `body_storage_filename` option before each new request, or set it to None to enable default randomly generated file names.

**content_type**

*Type* string

*Default* “html”

*Available values* “html” and “xml”

This option controls which lxml parser is used to process the body of the response. By default, the html parser is used. If you want to parse XML, then you may need to change this option to “xml” to force the use of an XML parser which does not strip the content of CDATA nodes.

**fix_special_entities**

*Type* bool

*Default* True

Fix `&#X;` entities, where X between 128 and 160. Such entities are parsed by modern browsers as windows-1251 entities, independently of the real charset of the document. If this option is True, then such entities will be replaced with appropriate unicode entities, e.g.: `&#151;` -> `—` and `&#8212;`.

**Debugging**

**log_file**

*Type* string

*Default* None

Path to the file where the body of the recent network response will be saved. See details at Saving the content of requests and responses.

**log_dir**

*Type* string

*Default* None

Directory to save the content of each response in. Each response will be saved to a unique file. See details at Saving the content of requests and responses.

**verbose_logging**

*Type* bool

*Default* False
This option enables printing to console of all detailed debug info about each pycurl action. Sometimes this can be useful.

**debug_post**

*Type*  bool

*Default*  False

Enable logging of POST request content.

### 3.1.7 Debugging

#### Using the logging module

The easiest way to see what is going on is to enable DEBUG logging messages. Write the following code at every entry point to your program:

```python
>>> import logging

>>> logging.basicConfig(level=logging.DEBUG)
```

That logging configuration will output all logging messages to console, not just from Grab but from other modules too. If you are interested only in Grab’s messages:

```python
>>> import logging

>>> logger = logging.getLogger('grab')

>>> logger.addHandler(logging.StreamHandler())

>>> logger.setLevel(logging.DEBUG)
```

You can also use a `default_logging` function that configures logging as follows:

- all messages of any level except from Grab modules are printed to console
- all “grab*” messages with level INFO or higher are printed to console
- all “grab*” messages of any level are saved to /tmp/grab.log
- all “grab.network*” messages (usually these are URLs being requested) of any level are saved to /tmp/grab.network.log

Usage of `default_logging` function is simple:

```python
>>> from weblib.logs import default_logging

>>> default_logging()
```

#### Logging messages about network request

For each network request, Grab generates the “grab.network” logging message with level DEBUG. Let’s look at an example:

```
[5864] GET http://www.kino-govno.com/movies/rusichi via 188.120.244.68:8080 proxy of →
˓
→type http with authorization
```

We can see the requested URL and also that request has ID 5864, that the HTTP method is GET, and that the request goes through a proxy with authorization. For each network request Grab uses the next ID value from the sequence
that is shared by all Grab instances. That does mean that even different Grab instances will generates network logging messages with unique ID.

You can also turn on logging of POST request content. Use the `debug_post` option:

```python
>>> g.setup(debug_post=True)
```

The output will be like this:

```
[01] POST http://yandex.ru
POST request:
  foo          : bar
  name         : Ivan
```

### Saving the content of requests and responses

You can ask Grab to save the content of each network response to the file located at the path passed as the `log_file` option:

```python
>>> g.setup(log_file='log.html')
```

Of course, each new response will overwrite the content of the previous response.

If you want to log all traffic, then consider using the `log_dir` option, which tells Grab to save the contents of all responses to files inside the specified directory. Note that each such file will contain a request ID in its filename. For each response, there will be two files: XXX.log and XXX.html. The file XXX.html contains the raw response. Even if you requested an image or large movie, you’ll get its raw content in that file. The file XXX.log contains headers of network response. If you configure Grab with `debug=True`, the file XXX.log will also contain request headers.

#### 3.1.8 Work with HTTP Headers

### Custom headers

If you need to submit custom HTTP headers, you can specify any number of them via `headers` option. A common case is to emulate an AJAX request:

```python
>>> g = Grab()
>>> g.setup(headers={'X-Requested-With': 'XMLHttpRequest'})
```

Bear in mind, that except headers in `headers` option (that is empty by default) Grab also generates a bunch of headers to emulate a typical web browser. At the moment of writing these docs these headers are:

- `Accept`
- `Accept-Language`
- `Accept-Charset`
- `Keep-Alive`
- `Except`

If you need to change one of these headers, you can override its value with the `headers` option. You can also subclass the Grab class and define your own `common_headers` method to completely override the logic of generating these extra headers.
User-Agent header

By default, for each request Grab randomly chooses one user agent from a builtin list of real user agents. You can specify the exact User-Agent value with the user_agent option. If you need to randomly choose user agents from your own list of user agents, then you can put your list into a text file and pass its location as user_agent_file.

Referer header

To specify the content of the Referer header, use the referer option. By default, Grab use the URL of previously request document as value of Referer header. If you do not like this behaviour, you can turn it off with reuse_referer option.

HTTP Authentication

To send HTTP authentication headers, use the userpwd option with a value of the form “username:password”.

3.1.9 Processing the Response Body

In this document, options related to processing the body of network response are discussed.

Partial Response Body Download

If you do not need the response body at all, use nobody option. When it is enabled, Grab closes the network connection to the server right after it receives all response headers from the server. This is not the same as sending a GET request. You can submit a request of any type, e.g. POST, and not download the response body if you do not need it.

Another option to limit body processing is body_maxsize. It allows you to download as many bytes of the response body as you need, and then closes the connection.

Note that neither of these options break the processing of the response into a Python object. In both cases you get a response object with a body attribute that contains only part of the response body data - whatever was received before connection interrupted.

Response Compression Method

You can control the compression of the server response body with encoding. The default value is “gzip”. That means that Grab sends “Accept-Encoding: gzip” to the server, and if the server answers with a response body packed with gzip then Grab automatically unpacks the gzipped body, and you have unpacked data in the response.body. If you do not want the server to send you gziped data, use an empty string as the value of encoding.

3.1.10 File Uploading

To upload file you should use UploadFile or UploadContent classes.

UploadFile example:

```python
from grab import Grab, UploadFile

g = Grab()
g.setup(post={'image': UploadFile('/path/to/image.jpg')})
g.go('http://example.com/form.php')
```
UploadContent example:

```python
from grab import Grab, UploadContent

g = Grab()
g.setup(post={'image': UploadContent('......', filename='image.jpg')})
g.go('http://example.com/form.php')
```

Form Processing

You can use `UploadFile` and `UploadContent` in all methods that set values in form fields:

```python
from grab import Grab, UploadFile

g = Grab()
g.go('http://example.com/form.php')
g.doc.set_input('image', UploadFile('/path/to/image.jpg'))
g.doc.submit()
```

Custom File Name

With both `UploadFile` and `UploadContent` you can use custom filename.

If you do not specify filename then:

- `UploadFile` will use the filename extracted from the path to the file passed in first argument.
- `UploadContent` will generate random file name

```python
>>> from grab import UploadFile, UploadContent
>>> UploadFile('/path/to/image.jpg').filename
'image.jpg'
>>> UploadFile('/path/to/image.jpg', filename='avatar.jpg').filename
'avatar.jpg'
>>> UploadContent('......').filename
'528e418951'
>>> UploadContent('......', filename='avatar.jpg').filename
'avatar.jpg'
```

Custom Content Type

With both `UploadFile` and `UploadContent` you can use custom content type.

If you do not specify content type then filename will be used to guess the content type e.g. “image.jpg” will have “image/jpeg” content type and “asdfsdf” will be just a “application/octet-stream”

```python
>>> from grab import UploadFile, UploadContent
>>> UploadFile('/path/to/image.jpg').content_type
'image/jpeg'
>>> UploadFile('/path/to/image.jpg', content_type='text/plain').content_type
'text/plain'
>>> UploadContent('/path/to/image.jpg').content_type
'image/jpeg'
>>> UploadContent('....', content_type='text/plain').content_type
'text/plain'
```
3.1.11 Redirect Handling

Grab supports two types of redirects:

- HTTP redirects with HTTP 301 and 302 status codes
- HTML redirects with the `<meta>` HTML tag

**HTTP 301/302 Redirect**

By default, Grab follows any 301 or 302 redirect. You can control the maximum number of redirects per network query with the `redirect_limit` option. To completely disable handling of HTTP redirects, set `follow_location` to False.

Let’s see how it works:

```python
>>> g = Grab()
>>> g.setup(follow_location=False)
>>> g.go('http://google.com')
<grab.response.Response object at 0x1246ae0>
>>> g.response.code
301
>>> g.response.headers['Location']
'http://www.google.com/'
>>> g.setup(follow_location=True)
>>> g.go('http://google.com')
<grab.response.Response object at 0x1246ae0>
>>> g.response.code
200
>>> g.response.url
'http://www.google.ru/?gws_rd=cr&ei=8spFUtS8EOWq4ATAooDwAg'
```

**Meta Refresh Redirect**

An HTML Page could contain special tags that instructs the browser to go to a specified URL:

```html
<meta http-equiv="Refresh" content="0; url=http://some/url" />
```

By default, Grab ignores such instructions. If you want automatically follow meta refresh tags, then set `follow_refresh` to True.

**Original and Destination URLs**

You can always get information about what URL you’ve requested initially and what URL you ended up with:

```python
>>> g = Grab()
>>> g.go('http://google.com')
<grab.response.Response object at 0x20fcae0>
>>> g.config['url']
'http://google.com'
>>> g.response.url
'http://www.google.ru/?gws_rd=cr&ei=8spFUtS8EOWq4ATAooDwAg'
```

The initial URL is stored on the config object. The destination URL is written into `response` object.

You can even track redirect history with `response.head`:
3.1.12 Form Processing

Grab can help you process web forms. It can automatically fill all input fields that have default values, letting you fill only fields you need. The typical workflow is:

- request a page
- fill input fields with `set_input` method
- submit the form with `submit` method

When you are using `set_input` you just specify the name of an input and the value, and Grab automatically finds the form field containing the input with that name. When you call `submit`, the automatically-chosen form is submitted (the form that has the largest number of fields). You can also explicitly choose the form with the `choose_form` method.

Let's look at a simple example of how to use these form features:

```python
>>> g = Grab()
>>> g.go('http://ya.ru/')
>>> g.set_input('text', 'grab lib')
>>> g.submit()
>>> g.doc.select('//a[@class="b-serp-item__title-link"]/@href').text()
'http://grablib.org/

The form that has been chosen automatically is available in the `grab.form` attribute.'
To specify input values you can use `set_input`, `set_input_by_id` and `set_input_by_xpath` methods.

### 3.1.13 Network Errors Handling

**Network Errors**

If a network request fails, Grab raises `grab.error.GrabNetworkError`. There are two situations when a network error exception will raise:

- the server broke connection or the connection timed out
- the response had any HTTP status code that is not 2XX or 404

Note particularly that 404 is a valid status code, and does not cause an exception to be raised.

**Network Timeout**

You can configure timeouts with the following options:

- connect to server timeout with `connect_timeout` option
- whole request/response operation timeout with `timeout` option

In case of a timeout, Grab raises `grab.error.GrabTimeoutError`.

### 3.1.14 HTML Document Charset

**Why does charset matter?**

By default, Grab automatically detects the charset of the body of the HTML document. It uses this detected charset to

- build a DOM tree
- convert the bytes from the body of the document into a unicode stream
- search for some unicode string in the body of the document
- convert unicode into bytes data, then some unicode data needs to be sent to the server from which the response was received.

The original content of the network response is always accessible at `response.body` attribute. A unicode representation of the document body can be obtained by calling `response.unicode_body()`:

```python
>>> g.go('http://mail.ru/')
<grab.response.Response object at 0x7f7d38af8940>
>>> type(g.response.body)
<type 'str'>
>>> type(g.response.unicode_body())
<type 'unicode'>
>>> g.response.charset
'utf-8'
```

**Charset Detection Algorithm**

Grab checks multiple sources to find out the real charset of the document’s body. The order of sources (from most important to less):
• HTML meta tag:

```html
<meta name="http-equiv" content="text/html; charset=cp1251" >
```

• XML declaration (in case of XML document):

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
```

• Content-Type HTTP header:

```text
Content-Type: text/html; charset=koi8-r
```

If no source indicates the charset, or if the found charset has an invalid value, then `grab` falls back to a default of UTF-8.

### Setting the charset manually

You can bypass automatic charset detection and specify it manually with `charset` option.

### 3.1.15 Cookie Support

By default, `grab` automatically handles all cookies it receives from the remote server. `grab` remembers all cookies and sends them back in future requests. That allows you to easily implement scripts that log in to some site and then do some activity in a member-only area. If you do not want `grab` to automatically process cookies, use `reuse_cookies` option to disable it.

#### Custom Cookies

To send some custom cookies, use the `cookies` option. The value of that option should be a dict. When you specify some cookies with `cookies` option and then fire network request, all specified cookies are bound to the hostname of the request. If you want more granular control on custom cookies, you can use the `grab.cookies` cookie manager to specify a cookie with any attributes you want:

```python
>>> g = Grab()
>>> g.cookies.set(name='foo', value='bar', domain='yandex.ru', path='/host')
```

#### Loading/dumping cookies

To dump current cookies to a file, use `grab.cookie.CookieManager.save_to_file()`. To load cookies from a file, use `grab.cookie.CookieManager.load_from_file()`.

#### Permanent file to load/store cookies

With the `cookiefile` option, you can specify the path to the file that `grab` will use to store/load cookies for each request. `grab` will load any cookies from that file before each network request, and after a response is received `grab` will save all cookies to that file.

More details about `grab.cookies` you can get in `API grab.cookie`
3.1.16 Proxy Server Support

Basic Usage

To make Grab send requests through a proxy server, use the `proxy` option:

```python
g.setup(proxy='example.com:8080')
```

If the proxy server requires authentication, use the `proxy_userpwd` option to specify the username and password:

```python
g.setup(proxy='example.com:8080', proxy_userpwd='root:777')
```

You can also specify the type of proxy server: “http”, “socks4” or “socks5”. By default, Grab assumes that proxy is of type “http”:

```python
g.setup(proxy='example.com:8080', proxy_userpwd='root:777', proxy_type='socks5')
```

You can always see which proxy is used at the moment in `g.config['proxy']`:

```python
>>> g = Grab()
>>> g.setup(proxy='example.com:8080')
>>> g.config['proxy']
'example.com:8080'
```

Proxy List Support

Grab supports working with a list of multiple proxies. Use the `g.proxylist` attribute to get access to the proxy manager. By default, the proxy manager is created and initialized with an empty proxy list:

```python
>>> g = Grab()
>>> g.proxylist
<grab.proxy.ProxyList object at 0x2e15b10>
>>> g.proxylist.proxy_list
[]
```

Proxy List Source

You need to setup the proxy list manager with details of the source that manager will load proxies from. Using the `g.proxylist.set_source` method, the first positional argument defines the type of source. Currently, two types are supported: “file” and “remote”.

Example of loading proxies from local file:

```python
>>> g = Grab()
>>> g.proxylist.set_source('file', location='/web/proxy.txt')
<grab.proxy.ProxyList object at 0x2e15b10>
>>> g.proxylist.set_source('file', location='/web/proxy.txt')
>>> g.proxylist.get_next()
<grab.proxy.Proxy object at 0x2d7c610>
>>> g.proxylist.get_next_proxy().server
'example.com'
>>> g.proxylist.get_next_proxy().address
```
(continues on next page)
'example.com:8080'

```python
>>> len(g.proxylist.proxy_list)
1000
```

And here is how to load proxies from the web:

```python
>>> g = Grab()
>>> g.proxylist.set_source('remote', url='http://example.com/proxy.txt')
```

### Automatic Proxy Rotation

By default, if you set up any non-empty proxy source, Grab starts rotating through proxies from the proxy list for each request. You can disable proxy rotation with `proxy_auto_change` option set to False:

```python
>>> from grab import Grab
>>> import logging

>>> logging.basicConfig(level=logging.DEBUG)

>>> g = Grab()

```python
>>> g.proxylist.set_source('file', location='/web/proxy.txt')
```python
>>> g.go('http://yandex.ru/')
```

<grab.response.Response object at 0x109d9f0>

```python
>>> g.go('http://rambler.ru/)
```

DEBUG:grab.network:[03] GET http://rambler.ru/ via 194.29.185.38:8080 proxy of type http with authorization
<grab.response.Response object at 0x109d9f0>

Now let's see how Grab works when `proxy_auto_change` is False:

```python
>>> from grab import Grab
>>> import logging

>>> g = Grab()

```python
>>> g.proxylist.set_source('file', location='/web/proxy.txt')
```python
>>> g.setup(proxy_auto_change=False)
```python
>>> g.go('http://ya.ru')
```

DEBUG:grab.network:[04] GET http://ya.ru
<grab.response.Response object at 0x109de50>

```python
>>> g.change_proxy()
```

DEBUG:grab.network:[05] GET http://ya.ru via 62.122.73.30:8080 proxy of type http with authorization
<grab.response.Response object at 0x109d9f0>

```python
>>> g.go('http://ya.ru')
```

DEBUG:grab.network:[06] GET http://ya.ru via 62.122.73.30:8080 proxy of type http with authorization
<grab.response.Response object at 0x109d9f0>

### Getting Proxy From Proxy List

Each time you call `g.proxylist.get_next_proxy`, you get the next proxy from the proxy list. When you receive the last proxy in the list, you'll continue receiving proxies from the beginning of the list. You can also use `g.proxylist.get_random_proxy` to pick a random proxy from the proxy list.
Automatic Proxy List Reloading

Grab automatically re-reads the proxy source each \texttt{g.proxylist.reload\_time} seconds. You can set the value of this option as follows:

\begin{verbatim}
>>> g = Grab()
>>> g.proxylist.setup(reload_time=3600)  # reload proxy list one time per hour
\end{verbatim}

Proxy Accumulating

By default, Grab overwrites the proxy list each time it reloads the proxy source. You can change that behaviour:

\begin{verbatim}
>>> g.proxylist.setup(accumulate\_updates=True)
\end{verbatim}

That will setup Grab to append new proxies to existing ones.

3.1.17 Searching the response body

String search

With the \texttt{doc.text\_search} method, you can find out if the response body contains a certain string or not:

\begin{verbatim}
>>> g = Grab('<h1>test</h1>')
>>> g.doc.text\_search('tes')
True
\end{verbatim}

If you prefer to raise an exception if string was not found, then use the \texttt{doc.text\_assert} method:

\begin{verbatim}
>>> g = Grab('<h1>test</h1>')
>>> g.doc.text\_assert('tes')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/lorien/web/grab/grab/document.py", line 109, in text\_assert
    raise DataNotFound(u'Substring not found: %s' % anchor)
grab.error.DataNotFound: Substring not found: tez
\end{verbatim}

By default, all text search methods operate with unicode; i.e., you should pass unicode arguments to these methods and these methods will search inside document’s body converted to unicode. There is an option to work with raw bytes, just pass \texttt{byte=True} to any method:

\begin{verbatim}
>>> g = Grab('<h1>test</h1>')
>>> g.doc.text\_search(b'tez', byte=True)
\end{verbatim}

Regexp Search

You can search for a regular expression with \texttt{doc.rex\_search} method that accepts compiled regexp object or just a text of regular expression:

\begin{verbatim}
>>> g = Grab('<h1>test</h1>')
>>> g.doc.rex\_search('<.+?>').group(0)
'u<h1>'
\end{verbatim}

Method \texttt{doc.rex\_text} returns you text contents of \texttt{.group(1)} of the found match object:
Method `doc.rex_assert` raises `DataNotFound` exception if no match is found:

```
>>> g = Grab('<h1>test</h1>')
>>> g.doc.rex_assert('\w{10}')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/lorien/web/grab/grab/grab/document.py", line 189, in rex_assert
    self.rex_search(rex, byte=byte)
  File "/home/lorien/web/grab/grab/grab/document.py", line 180, in rex_search
    raise DataNotFound('Could not find regexp: %s' % regexp)
grab.error.DataNotFound: Could not find regexp: <_sre.SRE_Pattern object at 0x7fa40e97d1f8>
```

### 3.1.18 Pycurl Tips

#### Asynchronous DNS Resolving

Pycurl allows you to drive network requests asynchronously with the multicurl interface. Unfortunately, by default multicurl do not handle DNS requests asynchronously. That means that every DNS request blocks other network activity. You can manage it by building curl from source and configuring it to use the ares library, which knows how to do asynchronous DNS requests.

First, you need to download curl sources from http://curl.haxx.se/download.html. Then unpack source code and run the command:

```
$ ./configure --prefix=/opt/curl --enable-ares
```

We use a custom prefix because we do not want to mix up our custom curl with the curl lib that could be already installed in your system. Do not forget to install cares packages before configuring curl with ares:

```
$ apt-get install libc-ares-dev
```

If `./configure` command has finished successfully, then run:

```
$ make
$ make install
```

Now you have customized the curl library files at `/opt/curl/lib`.

To let your python script know that you want to use this custom curl lib, use the following feature:

```
$ LD_LIBRARY_PATH="/opt/curl/lib" python your_script.py
```

You can manually check that you used a curl compiled with ares support:

```
>>> import pycurl
>>> pycurl.version
'libcurl/7.32.0 OpenSSL/1.0.1e zlib/1.2.7 c-ares/1.10.0 libidn/1.25'
```

You should see something like “c-ares/1.10.0” if everything was correct.
Supported Protocols

By default, pycurl supports a ton of protocols including SMTP, POP3, SSH, media streams, and FTP. If you do not need all this crap, you can disable it at the configure stage. Here is example of what you can do:

```
./configure --without-libssh2 --disable-ipv6 --disable-ldap --disable-ldaps
   --without-librtmp --disable-rtsp --disable-ftp --disable-dict
   --disable-telnet --disable-tftp --disable-pop3 --disable-imap
   --disable-smtp --disable-gopher --without-winssl --without-darwinssl
   --without-winidn
```

To see all available options, just run the command:

```
./configure --help
```

Multicurl and SOCKS proxy

This combination just does not work. Use HTTP proxies with multicurl.

3.1.19 Work With Network Response

Response Object

The result of doing a network request via Grab is a `Response` object.

You get a Response object as a result of calling to `g.go`, `g.request` and `g.submit` methods. You can also access the response object of a recent network query via the `g.response` attribute:

```
>>> from grab import Grab
>>> g = Grab()
>>> g.go('http://google.com')
<grab.response.Response object at 0x2cff9f0>
>>> g.response
<grab.response.Response object at 0x2cff9f0>
```

You can find a full list of response attributes in the Response API document. Here are the most important things you should know:

- **body** original body contents of HTTP response
- **code** HTTP status of response
- **headers** HTTP headers of response
- **charset** charset of the response
- **cookies** cookies in the response
- **url** the URL of the response document. In case of some automatically processed redirect, the `url` attribute contains the final URL.
- **name_lookup_time** time spent to resolve host name
- **connect_time** time spent to connect to remote server
- **total_time** total time spent to complete the request
- **download_size** size of received data
**upload_size**  size of uploaded data except the HTTP headers

Now, a real example:

```python
>>> from grab import Grab
>>> g = Grab()
>>> g.go('http://wikipedia.org')
<grab.response.Response object at 0x1ff99f0>
>>> g.response.body[:100]
'<!DOCTYPE html>
<html lang="mul" dir="ltr">
<head>
    <!-- Sysops: Please do not edit the main template -->
</head>
<body>
    <!-- The main content goes here -->
</body>
</html>
>>> g.response.code
200
>>> g.response.headers['Content-Type']
'text/html; charset=utf-8'
>>> g.response.charset
'utf-8'
>>> g.response.cookies
<grab.cookie.CookieManager object at 0x1f6b248>
>>> g.response.url
'http://www.wikipedia.org/
>>> g.response.name_lookup_time
0.103931
>>> g.response.connect_time
0.221996
>>> g.response.total_time
0.7791399999999999
>>> g.response.download_size
11100.0
>>> g.response.upload_size
0.0
```

Now let's see some useful methods available in the response object:

- **unicode_body()**  this method returns the response body converted to unicode
- **copy()**  returns a clone of the response object
- **save(path)**  saves the response object to the given location
- **json**  treats the response content as json-serialized data and de-serializes it into a python object. Actually, this is not a method, it is a property.
- **url_details()**  return the result of calling urlparse.urlsplit with response.url as an argument.
- **query_param(name)**  extracts the value of the key argument from the query string of response.url.

### 3.1.20 Network Transport

Grab can use two libraries to submit network requests: pycurl and urllib3. You may access transport object with `Grab.transport` attribute. In most cases you do not need direct access to transport object.

#### Pycurl transport

The pycurl transport is the default network transport. You can control low-level options of pycurl object by accessing `Grab.transport.pycurl` object. For example:
from grab import Grab
import pycurl

g = Grab()
g.transport.pycurl.setopt(pycurl.LOW_SPEED_LIMIT, 100)
g.go('http://example.com/download/porn.mpeg')

### urllib3 transport

If you want to use Grab in gevent environment then consider to use urllib3 transport. The urllib3 uses native python sockets that could be patched by `gevent.monkey.patch_all`.

```python
import gevent
import gevent.monkey
from grab import Grab
import time

def worker():
    g = Grab(user_agent='Medved', transport='urllib3')
    # Request the document that is served with 1 second delay
    g.go('http://httpbin.org/delay/1')
    return g.doc.json['headers']['User-Agent']

started = time.time()
gevent.monkey.patch_all()
pool = []
for _ in range(10):
    pool.append(gevent.spawn(worker))
for th in pool:
    th.join()
    assert th.value == 'Medved'
# The total time would be less than 2 seconds
# unless you have VERY bad internet connection
assert (time.time() - started) < 2
```

### Use your own transport

You can implement you own transport class and use it. Just pass your transport class to `transport` option.

Here is the crazy example of wget-powered transport. Note that this is VERY simple transport that understands only one option: the URL.

```python
from grab import Grab
from grab.document import Document
from subprocess import check_output

class WgetTransport(object):
    def __init__(self):
        self.request_head = b''
        self.request_body = b''
```

(continues on next page)
def reset(self):  
    pass

def process_config(self, grab):
    self._request_url = grab.config['url']

def request(self):
    out = check_output(['/usr/bin/wget', '-O', '-',
                        self._request_url])
    self._response_body = out

def prepare_response(self, grab):
    doc = Document()
    doc.body = self._response_body
    return doc

g = Grab(transport=WgetTransport)
g.go('http://protonmail.com')
assert 'Secure email' in g.doc('//title').text()

3.2 Grab::Spider User Manual

Grab::Spider is a framework to build well-structured asynchronous web-site crawlers.

3.2.1 What is Grab::Spider?

The Spider is a framework that allow to describe web-site crawler as set of handlers. Each handler handles only one specific type of web pages crawled on web-site e.g. home page, user profile page, search results page. Each handler could spawn new requests which will be processed in turn by other handlers.

The Spider process network requests asynchronously. There is only one process that handles all network, business logic and HTML-processing tasks. Network requests are performed by multicurl library. In short, when you create new network request it is processed by multicurl and when the response is ready, then the corresponding handler from your spider class is called with result of network request.

Each handler receives two arguments. First argument is a Grab object, that contains all data bout network request and response. The second argument is Task object. Whenever you need to send network request you create Task object. The second argument is Task object. Whenever you need to send network request you create Task object.

Let's check out simple example. Let's say we want to go to habrahabr.ru web-site, read titles of recent news, then for each title find the image on images.yandex.ru and save found data to the file.

```python
# coding: utf-8
import urllib
import csv
import logging

from grab.spider import Spider, Task

class ExampleSpider(Spider):
    # List of initial tasks
    # For each URL in this list the Task object will be created
    initial_urls = ['http://habrahabr.ru/']
```
def prepare(self):
    # Prepare the file handler to save results.
    # The method 'prepare' is called one time before the
    # spider has started working
    self.result_file = csv.writer(open('result.txt', 'w'))

    # This counter will be used to enumerate found images
    # to simplify image file naming
    self.result_counter = 0

def task_initial(self, grab, task):
    print 'Habrahabr home page'

    # This handler for the task named 'initial i.e.
    # for tasks that have been created from the
    # 'self.initial_urls' list

    # As you see, inside handler you can work with Grab
    # in usual way i.e. just if you have done network request
    # manually
    for elem in grab.doc.select('//h1[@class="title"]' /'
        a[@class="post_title"]'):
        # For each title link create new Task
        # with name "habrapost"
        # Pay attention, that we create new tasks
        # with yield call. Also you can use 'add_task' method:
        # self.add_task(Task('habrapost', url=...))
        yield Task('habrapost', url=elem.attr('href'))

def task_habrapost(self, grab, task):
    print 'Habrahabr topic: %s' % task.url

    # This handler receives results of tasks we
    # created for each topic title found on home page

    # First, save URL and title into dictionary
    post = {
        'url': task.url,
        'title': grab.xpath_text('//h1/span[@class="post_title"]'),
    }

    # Next, create new network request to search engine to find
    # the image related to the title.
    # We pass info about the found publication in the arguments to
    # the Task object. That allows us to pass information to next
    # handler that will be called for found image.
    query = urllib.quote_plus(post['title'].encode('utf-8'))
    search_url = 'http://images.yandex.ru/yandsearch' + '
       ?text=%s&rpt=image' % query
    yield Task('image_search', url=search_url, post=post)

def task_image_search(self, grab, task):
    print 'Images search result for %s' % task.post['title']

    # In this handler we have received result of image search.
    # That is not image! This is just a list of found images.
    # Now, we take URL of first image and spawn new network
In this example, we have considered the simple spider. I hope you have got idea about how it works. See other parts of Grab::Spider User Manual to get detailed description of spider features.

### 3.2.2 Task Object

Any Grab::Spider crawler is a set of handlers that process network responses. Each handler can spawn new network requests or just process/save data. The spider add each new request to task queue and process this task when there is free network stream. Each task is assigned a name that defines its type. Each type of task are handles by specific handler. To find the handler the Spider takes name of the task and then looks for task_<name> method.

For example, to handle result of task named “contact_page” we need to define “task_contact_page” method:

```python
... 
    self.add_task(Task('contact_page', url='http://domain.com/contact.html'))
... 

def task_contact_page(self, grab, task):
    ...
```

**Constructor of Task Class**

Constructor of Task Class accepts multiple arguments. At least you have to specify URL of requested document OR the configured Grab object. Next, you see examples of different task creation. All three examples do the same:
Also you can specify these arguments:

- **priority** task priority, it’s unsigned natural number, the less number mean the higher priority.
- **disable_cache** don’t use spider’s cache for this request, network response will not stored into cache as well.
- **refresh_cache** do not use spider’s cache, in case of success response it will refresh cache.
- **valid_status** procces the following response codes in task handler. By default only 2xx and 404 statuses will be processed in task handlers.
- **use_proxylist** use spider’s global proxy list, by default this oprion is True

**Task Object as Data Storage**

If you pass the argument that is unknown then it will be saved in the Task object. That allows you to pass data between network request/response.

There is `get` method that return value of task attribute or `None` if that attribute have not been defined.

```python
t = Task('bing', url='http://bing.com/', disable_cache=True, foo='bar')
t.foo # == "bar"
t.get('foo') # == "bar"
t.get('asdf') # == None
t.get('asdf', 'qwerty') # == "qwerty"
```

**Cloning Task Object**

Sometimes it is useful to create copy of Task object. For example:

```python
# task.clone()
# TODO: example
```

**Setting Up Initial Tasks**

When you call `run` method of your spider it starts working from initial tasks. There are few ways to setup initial tasks.
initial_urls

You can specify list of URLs in `self.initial_urls`. For each URI in this list the spider will create Task object with name "initial":

```python
class ExampleSpider(Spider):
    initial_urls = ['http://google.com/', 'http://yahoo.com/']
```

task_generator

More flexible way to define initial tasks is to use `task_generator` method. Its interface is simple, you just have to yield new Task objects.

There is common use case when you need to process big number of URLs from the file. With `task_generator` you can iterate over lines of the file and yield new tasks. That will save memory used by the script because you will not read whole file into the memory. Spider consumes only portion of tasks from `task_generator`. When there are free networks resources the spiders consumes next portion of task. And so on.

Example:

```python
class ExampleSpider(Spider):
    def task_generator(self):
        for line in open('var/urls.txt'):
            yield Task('download', url=line.strip())
```

Explicit Ways to Add New Task

Adding Tasks With add_task method

You can use `add_task` method anywhere, even before the spider have started working:

```python
bot = ExampleSpider()
bot.setup_queue()
bot.add_task('google', url='http://google.com')
bot.run()
```

Yield New Tasks

You can use yield statement to add new tasks in two places. First, in `task_generator`. Second, in any handler. Using yield is completely equal to using `add_task` method. The yielding is just a bit more beautiful:

```python
class ExampleSpider(Spider):
    initial_urls = ['http://google.com']

    def task_initial(self, grab, task):
        # Google page was fetched
        # Now let's download yahoo page
        yield Task('yahoo', url='yahoo.com')

    def task_yahoo(self, grab, task):
        pass
```
**Default Grab Instance**

You can control the default config of Grab instances used in spider tasks. Define the `create_grab_instance` method in your spider class:

```python
class TestSpider(Spider):
    def create_grab_instance(self, **kwargs):
        g = super(TestSpider, self).create_grab_instance(**kwargs)
        g.setup(timeout=20)
        return g
```

Be aware, that this method allows you to control only those Grab instances that were created automatically. If you create task with explicit grab instance it will not be affected by `create_grab_instance` method:

```python
class TestSpider(Spider):
    def create_grab_instance(self, **kwargs):
        g = Grab(**kwargs)
        g.setup(timeout=20)
        return g

    def task_generator(self):
        g = Grab(url='http://example.com', timeout=5)
        yield Task('page', grab=g)
        # The grab instance in the yielded task
        # will not be affected by `create_grab_instance` method.
```

**Updating Any Grab Instance**

With method `update_grab_instance` you can update any Grab instance, even those instances that you have passed explicitly to the Task object. Be aware, that any option configured in this method overwrites the previously configured option:

```python
class TestSpider(Spider):
    def update_grab_instance(self, grab):
        grab.setup(timeout=20)

    def task_generator(self):
        g = Grab(url='http://example.com', timeout=5)
        yield Task('page', grab=g)
        # The effective timeout setting will be equal to 20!
```

### 3.2.3 Task Queue

**Task Priorities**

All new tasks are places into task queue. The Spider get tasks from task queue when there are free network streams. Each task has priority. Lower number means higher priority. Task are processed in the order of their priorities: from highest to lowest. If you do not specify the priority for the new task then it is assigned automatically. There are two algorithms of assigning default task priorities:

- `random` random priorities
- `const` same priority for all tasks

By default random priorities are used. You can control the algorithm of default priorities with `priority_mode` argument:
bot = SomeSpider(priority_mode='const')

**Tasks Queue Backends**

You can choose the storage for the task queue. By default, Spider uses Python `PriorityQueue` as storage. In other words, the storage is memory. You can also use redis and mongo backends.

**In-memory backend:**

```python
bot = SomeSpider()
bot.setup_queue()  #
# OR (that is the same)
bot.setup_queue(backend='memory')
```

**MongoDB backend:**

```python
bot = SomeSpider()
bot.setup_queue(backend='mongo', database='database-name')
```

All arguments except `backend` go to MongoDB connection constructor. You can setup database name, host name, port, authorization arguments and other things.

**Redis backend:**

```python
bot = SomeSpider()
bot.setup_queue(backend='redis', db=1, port=7777)
```

### 3.2.4 Spider Cache

There is a cache built in the spider. It could be helpful on development stage. When you need to scrape the same documents for many times to check the results and to fix bugs. Also you can crawl whole web-site, put it into cache and then work only with cache.

Keep in mind that if the web-site is large, millions of web pages then working with cache could be slower than working with live web-site. This is because of limited disk I/O where the cache storage is hosted.

Also keep in mind that the spider cache is very simple:

- it allows to cache only GET requests
- it does not allow to differentiate documents with same URL but different cookies/headers
- it does not support max-age and other cache headers

**Spider Cache Backends**

You can choose what storage to use for the cache. You can use mongodb, mysql and postgresql.

**MongoDB example:**

```python
bot = ExampleSpider()
bot.setup_cache(backend='mongo', database='some-database')
bot.run()
```
In this example the spider is configured to use mongodb as cache storage. The name of database is “some-database”. The name of collection would be “cache”.

All arguments except backend, database and use_compression go to database connection constructor. You can setup database name, host name, port, authorization arguments and other things.

Example of custom host name and port for mongodb connection:

```python
bot = SomeSpider() bot.setup_cache(backend='mongo', port=7777, host='mongo.localhost')
```

### Cache Compression

By default cache compression is enabled. That means that all documents placed in the cache are compressed with gzip library. Compression decreases the disk space required to store the cache and increases the CPU load (a bit).

#### 3.2.5 Spider Error Handling

### Rules of Network Request Handling

- If request is completed successfully then the corresponding handler is called
- If request is failed due the network error, then the task is submitted back to the task queue
- If the request is completed and the handler is called and failed due to any error inside the handler then the task processing is aborted. This type of errors is not fatal. The handler error is logged and other requests and handlers are processed in usual way.

### Network Errors

Network error is:

- error occurred in process of data transmission to or back from the server e.g. connection aborted, connection timeout, server does not accept connection and so on
- data transmission has been completed but the HTTP status of received document differs from 2XX or from 404

Yes, by default documents with 404 status code counts as valid! That makes sense to me :) If that is not you want then you can configure custom rule to mark status as valid or failed. You have two ways.

First way is to use valid_status argument in Task constructor. With this argument you can only extend the default valid status. This arguments accepts list of additional valid status codes:

```python
t = Task('example', url='http://example.com', valid_status=(500, 501, 502))
```

Second way is to redefine valid_response_code method. In this way you can implement any logic you want. Method accepts two arguments: status code and task object. Method returns boolean value, True means that the status code is valid:

```python
class SomeSpider(Spider):
    def valid_response_code(self, code, task):
        return code in (200, 301, 302)
```
Handling of Failed Tasks

The task failed due to the network error is put back to tas queue. The number of tries is limited to the Spider.network_try_limit and is 10 by default. The try's number is stored in the Task.network_try_count. If network_try_count reaches the network_try_limit the task is aborted.

When the task is aborted and there is method with name task_<task-name>_fallback then it is called and receives the failed task as first argument.

Also, it happens that you need to put task back to task queue even if it was not failed due to the network error. For example, the response contains captcha challenge or other invalid data reasoned by the anti-scraping protection. You can control number of such tries. Max tries number is configured by Spider.task_try_count. The try's number is stored in Task.task_try_count. Keep in mind, that you have to increase task_try_count explicitly when you put task back to task queue.

```python
def task_google(self, grab, task):
    if captcha_found(grab):
        yield Task('google', url=grab.config['url'],
                    task_try_count=task.task_try_count + 1)

def task_google_fallback(self, task):
    print 'Google is not happy with you IP address'
```

Manual Processing of Failed Tasks

You can disable default mechanism of processing failed tasks and process failures manually. Use raw=True parameter in Task constructor. If the network request would fail then the grab object passed to the handler would contain information about failure in two attributes: grab.response.error_code and grab.response.error_msg

See example:

```python
class TestSpider(Spider):
    def task_generator(self):
        yield Task('page', url='http://example.com/', raw=True)

    def task_page(self, grab, task):
        if grab.response.error_code:
            print('Request failed. Reason: %s' % grab.response.error_msg)
        else:
            print('Request completed. HTTP code: %d' % grab.response.code)
```

Error Statistics

After spider has completed the work or even in the process of working you can receive the information about number of completed requests, failed requests, number of specific network errors with method Spider.render_stats.

3.2.6 Spider Transport

Spider transport is a component of Spider that controls network connections i.e. makes possible multiple network requests to run in parallel.
Multicurl transport

This is default spider transport. It operates with multiple pycurl instances. You can use only pycurl Grab transport with multicurl Spider transport.

```python
from grab.spider import Spider, Task
from grab import Grab
import logging

class SimpleSpider(Spider):
    def task_generator(self):
        yield Task('reddit', 'http://reddit.com')

    def task_reddit(self, grab, task):
        url = grab.doc('//p[contains(@class, "title")]/a/@href').text()
        # DO NOT DO THAT:
        # > g = Grab()
        # > g.go(url)
        # Do not use Grab directly
        # that will blocks all other parallel network requests
        # Only use `yield Task(...)`
        url = grab.make_url_absolute(url)
        yield Task('link', url=url)

    def task_link(self, grab, task):
        print('Title: %s' % grab.doc('//title').text())

logging.basicConfig(level=logging.DEBUG)
bot = SimpleSpider();
bot.run()
```

Threaded transport

The threaded transport operates with a pool of threads. Network requests are spread by these threads. You can use pycurl or urllib3 Grab transport with threaded transport.

Grab can use two libraries to submit network requests: pycurl and urllib3. You may acess transport object with `Grab.transport` attribute. In most cases you do not need direct access to transport object.

```python
from grab.spider import Spider, Task
from grab import Grab
import logging

class SimpleSpider(Spider):
    def task_generator(self):
        yield Task('reddit', 'http://reddit.com')

    def task_reddit(self, grab, task):
        url = grab.doc('//p[contains(@class, "title")]/a/@href').text()
        # DO NOT DO THAT:
        # > g = Grab()
        # > g.go(url)
        # Do not use Grab directly
        # that will blocks all other parallel network requests
        # Only use `yield Task(...)`
```

(continues on next page)
url = grab.make_url_absolute(url)
yield Task('link', url=url)

def task_link(self, grab, task):
    print('Title: %s' % grab.doc('//title').text())

logging.basicConfig(level=logging.DEBUG)
bot = SimpleSpider(transport='threaded', grab_transport='urllib3')
# Also you can use pycurl Grab transport with threaded transport
# bot = SimpleSpider(transport='threaded', grab_transport='pycurl')
bot.run()

3.3 API Reference

Using the API Reference you can get an overview of what modules, classes, and methods exist, what they do, what they return, and what parameters they accept.

3.3.1 Module grab.base

Here is the heart of the library, the Grab class.

class grab.base.Grab(document_body=None, transport=None, **kwargs)

    __init__ (document_body=None, transport=None, **kwargs)
        Create Grab instance

    setup_transport (transport_param, reset=False)

    reset ()
        Reset all attributes which could be modified during previous request or which is not initialized yet if this
        is the new Grab instance.
        This methods is automatically called before each network request.

    clone (**kwargs)
        Create clone of Grab instance.
        Cloned instance will have the same state: cookies, referrer, response document data

        Parameters  **kwargs – overrides settings of cloned grab instance

    adopt (grab)
        Copy the state of another Grab instance.
        Use case: create backup of current state to the cloned instance and then restore the state from it.

    dump_config ()
        Make clone of current config.

    load_config (config)
        Configure grab instance with external config object.

    setup (**kwargs)
        Setting up Grab instance configuration.
go (url, **kwargs)
    Go to url

    Args:
        url  could be absolute or relative. If relative then t will be
             appended to the absolute URL of previous request.

download (url, location, **kwargs)
    Fetch document located at url and save to to location.

request (**kwargs)
    Perform network request.

    You can specify grab settings in **kwargs. Any keyword argument will be passed to self.config.

    Returns: Document objects.

fake_response (*args, **kwargs)

load_proxylist (source, source_type, proxy_type='http', auto_init=True, auto_change=True)

change_proxy (random=True)
    Set random proxy from proxylist.

make_url_absolute (url, resolve_base=False)
    Make url absolute using previous request url as base url.

clear_cookies ()
    Clear all remembered cookies.

load_cookies (path, file_required=None)

dump_cookies (path)

setup_with_proxyline (line, proxy_type='http')

3.3.2 Module grab.error

Custom exception classes for Grab.

class grab.error.GrabError
    All custom Grab exception should be children of that class.

class grab.error.GrabNetworkError (*args, **kwargs)
    Raises in case of network error.

class grab.error.GrabTimeoutError (*args, **kwargs)
    Raises when configured time is outed for the request.

    In curl transport it is CURLE_OPERATION_TIMEDOUT (28)

class grab.error.DataNotFound

class grab.error.GrabMisuseError
    Indicates incorrect usage of grab API.

class grab.error.GrabConnectionError (*args, **kwargs)
    Raised when it is not possible to establish network connection.

    In curl transport it is CURLE_COULDNT_CONNECT (7)
class grab.error.GrabAuthError
    Raised when remote server denies authentication credentials.
    In curl transport it is CURLE_COULDNT_CONNECT (67)

class grab.error.GrabTooManyRedirectsError
    Raised when Grab reached max. allowed number of redirects for one request.

class grab.error.GrabInvalidUrl
    Raised when Grab have no idea how to handle the URL or when some error occurred while normalizing URL e.g. IDN processing.

class grab.util.warning.GrabDeprecationWarning
    Warning category used in Grab to generate warning messages.

3.3.3 Module `grab.cookie`

This module contains some classes to work with cookies.

```
grab.cookie.create_cookie(name, value, domain, httponly=None, **kwargs)
```
Creates `cookielib.Cookie` instance

```
class grab.cookie.CookieManager(cookiejar=None)
```
Each Grab instance has cookies attribute that is instance of CookieManager class.

That class contains helpful methods to create, load, save cookies from/to different places.

```
__init__(cookiejar=None)
    Initialize self. See help(type(self)) for accurate signature.

set(name, value, domain, **kwargs)
    Add new cookie or replace existing cookie with same parameters.

update(cookies)

clear()

__getitem__(key)
    Implements dict interface, allows to get cookie value by its name.

items()

get_dict()

load_from_file(path)
    Load cookies from the file.
    Content of file should be a JSON-serialized list of dicts.

save_to_file(path)
    Dump all cookies to file.
    Cookies are dumped as JSON-serialized dict of keys and values.
3.3.4 Module grab.spider

class grab.spider.base.Spider(thread_number=None, network_try_limit=None, task_try_limit=None, request_pause=<object object>, priority_mode='random', meta=None, only_cache=False, config=None, args=None, parser_requests_per_process=10000, parser_pool_size=1, http_api_port=None, network_service='threaded', grab_transport='pycurl', transport=None)

Asynchronous scraping framework.

add_task (task, queue=None, raise_error=False)
Add task to the task queue.

check_task_limits (task)
Check that task’s network & try counters do not exceed limits.
Returns: * if success: (True, None) * if error: (False, reason)

is_valid_network_response_code (code, task)
Answer the question: if the response could be handled via usual task handler or the task failed and should be processed as error.

load_proxylist (source, source_type=None, proxy_type='http', auto_init=True, auto_change=True)
Load proxy list.

Parameters

• source – Proxy source. Accepts string (file path, url) or BaseProxySource instance.
• source_type – The type of the specified source. Should be one of the following: ‘text_file’ or ‘url’.
• proxy_type – Should be one of the following: ‘socks4’, ‘socks5’ or ‘http’.
• auto_change – If set to True then automatical random proxy rotation will be used.

Proxy source format should be one of the following (for each line):

• ip:port
• ip:port:login:password

prepare ()
You can do additional spider customization here before it has started working. Simply redefine this method in your Spider class.

process_grab_proxy (task, grab)
Assign new proxy from proxylist to the task

process_next_page (grab, task, xpath, resolve_base=False, **kwargs)
Generate task for next page.

Parameters

• grab – Grab instance
• task – Task object which should be assigned to next page url
• xpath – xpath expression which calculates list of URLs
• **kwargs – extra settings for new task object
Example:
```
self.follow_links(grab, 'topic', '//div[@class="topic"]//a/@href')
```

**setup_cache** (*backend*='mongodb', *database=None*, **kwargs)
Setup cache.

Parameters
- **backend** – Backend name. Should be one of the following: ‘mongo’, ‘mysql’ or ‘postgresql’.
- **database** – Database name.
- **kwargs** – Additional credentials for backend.

**setup_queue** (*backend*='memory', **kwargs)
Setup queue.

Parameters
- **backend** – Backend name. Should be one of the following: ‘memory’, ‘redis’ or ‘mongo’.
- **kwargs** – Additional credentials for backend.

**shutdown** ()
You can override this method to do some final actions after parsing has been done.

**stop** ()
This method sets internal flag which signals spider to stop processing new tasks and shuts down.

**task_generator** ()
You can override this method to load new tasks smoothly.

It will be used each time as number of tasks in task queue is less than number of threads multiplied on 2. This allows you to not overload all free memory if total number of tasks is big.

**update_grab_instance** (*grab*)
Use this method to automatically update config of any *Grab* instance created by the spider.

### 3.3.5 Module grab.document

The Document class is the result of network request made with *Grab* instance.

**class** *grab.document*.*Document* (*grab=None*)

Document (in most cases it is a network response) i.e. result of network request.

**browse** ()
Save response in temporary file and open it in GUI browser.

**choose_form** (*number=None*, *xpath=None*, *name=None*, **kwargs)
Set the default form.

Parameters
- **number** – number of form (starting from zero)
- **id** – value of “id” attribute
- **name** – value of “name” attribute
- **xpath** – XPath query

Raises **DataNotFound** if form not found
Raises `GrabMisuseError` if method is called without parameters

Selected form will be available via `form` attribute of `Grab` instance. All form methods will work with default form.

Examples:

```python
# Select second form
g.choose_form(1)

# Select by id
g.choose_form(id="register")

# Select by name
g.choose_form(name="signup")

# Select by xpath
g.choose_form(xpath='//form[contains(@action, "/submit")}')
```

copy (`new_grab=None`)

Clone the Response object.

detect_charset ()

Detect charset of the response.

Try following methods: * meta[name="Http-Equiv"] * XML declaration * HTTP Content-Type header

Ignore unknown charsets.

Use utf-8 as fallback charset.

form

This attribute points to default form.

If form was not selected manually then select the form which has the biggest number of input elements.

The form value is just an `lxml.html` form element.

Example:

```python
g.go('some URL')
# Choose form automatically
print g.form

# And now choose form manually
g.choose_form(1)
print g.form
```

form_fields ()

Return fields of default form.

Fill some fields with reasonable values.

get_form_request (`submit_name=None, url=None, extra_post=None, remove_from_post=None`)

Submit default form.

Parameters

- `submit_name` – name of button which should be “clicked” to submit form
- `url` – explicitly specify form action url
- `extra_post` – (dict or list of pairs) additional form data which will override data automatically extracted from the form.
• **remove_from_post** – list of keys to remove from the submitted data

Following input elements are automatically processed:

• input[type="hidden"] - default value
• select: value of last option
• radio - ???
• checkbox - ???

Multipart forms are correctly recognized by grab library.

**json**
Return response body deserialized into JSON object.

**parse**(charset=None, headers=None)
Parse headers.

This method is called after Grab instance performs network request.

**pyquery**
Returns pyquery handler.

**query_param**(key)
Return value of parameter in query string.

**rex_assert**(rex, byte=False)
If rex expression is not found then raise `DataNotFound` exception.

**rex_search**(regexp, flags=0, byte=False, default=<object object>)
Search the regular expression in response body.

**Parameters**
- `byte` – if False then search is performed in `response.unicode_body()` else the rex is searched in `response.body`.

Note: if you use default non-byte mode than do not forget to build your regular expression with re.U flag.

Return found match object or None

**rex_text**(regexp, flags=0, byte=False, default=<object object>)
Search regular expression in response body and return content of first matching group.

**Parameters**
- `byte` – if False then search is performed in `response.unicode_body()` else the rex is searched in `response.body`.

**save**(path)
Save response body to file.

**save_hash**(location, basedir, ext=None)
Save response body into file with special path builded from hash. That allows to lower number of files per directory.

**Parameters**

- `location` – URL of file or something else. It is used to build the SHA1 hash.

- `basedir` – base directory to save the file. Note that file will not be saved directly to this directory but to some sub-directory of `basedir`

- `ext` – extension which should be appended to file name. The dot is inserted automatically between filename and extension.

**Returns** path to saved file relative to `basedir`
Example:

```python
>>> url = 'http://yandex.ru/logo.png'
>>> g.go(url)
>>> g.response.save_hash(url, 'some_dir', ext='png')
'e8/dc/f2918108788296df1facadc975d32b361a6a.png'
# the file was saved to $PWD/some_dir/e8/dc/...
```

TODO: replace basedir with two options: root and save_to. And returns save_to + path

**set_input** *(name, value)*

Set the value of form element by its *name* attribute.

**Parameters**

- *name* – name of element
- *value* – value which should be set to element

To check/uncheck the checkbox pass boolean value.

Example:

```python
g.set_input('sex', 'male')
# Check the checkbox
```

```python
g.set_input('accept', True)
```

**set_input_by_id** *(id, value)*

Set the value of form element by its *id* attribute.

**Parameters**

- *id* – id of element
- *value* – value which should be set to element

**set_input_by_number** *(number, value)*

Set the value of form element by its number in the form

**Parameters**

- *number* – number of element
- *value* – value which should be set to element

**set_input_by_xpath** *(xpath, value)*

Set the value of form element by xpath

**Parameters**

- *xpath* – xpath path
- *value* – value which should be set to element

**text_assert** *(anchor, byte=False)*

If *anchor* is not found then raise *DataNotFound* exception.

**text_assert_any** *(anchors, byte=False)*

If no *anchors* were found then raise *DataNotFound* exception.

**text_search** *(anchor, byte=False)*

Search the substring in response body.

**Parameters**
• anchor – string to search
• byte – if False then anchor should be the unicode string, and search will be performed in response.unicode_body() else anchor should be the byte-string and search will be performed in response.body

If substring is found return True else False.

tree
Return DOM tree of the document built with HTML DOM builder.

unicode_body (ignore_errors=True, fix_special_entities=True)
Return response body as unicode string.

url_details ()
Return result of urlsplit function applied to response url.

xml_tree
Return DOM-tree of the document built with XML DOM builder.

grab.document.read_bom (data)
Read the byte order mark in the text, if present, and return the encoding represented by the BOM and the BOM.

If no BOM can be detected, (None, None) is returned.

3.3.6 Module grab.spider.task

class grab.spider.task.Task (name=None, url=None, grab=None, grab_config=None, priority=None, priority_set_explicitly=True, network_try_count=0, task_try_count=1, disable_cache=False, refresh_cache=False, valid_status=None, use_proxylist=True, cache_timeout=None, delay=None, raw=False, callback=None, fallback_name=None, **kwargs)
Task for spider.

__init__ (name=None, url=None, grab=None, grab_config=None, priority=None, priority_set_explicitly=True, network_try_count=0, task_try_count=1, disable_cache=False, refresh_cache=False, valid_status=None, use_proxylist=True, cache_timeout=None, delay=None, raw=False, callback=None, fallback_name=None, **kwargs)
Create Task object.

If more than one of url, grab and grab_config options are non-empty then they processed in following order: * grab overwrite grab_config * grab_config overwrite url

Args:

param name name of the task. After successful network operation task’s result will be passed to task_<name> method.

param url URL of network document. Any task requires url or grab option to be specified.

param grab configured Grab instance. You can use that option in case when url option is not enough. Do not forget to configure url option of Grab instance because in this case the url option of Task constructor will be overwritten with grab.config['url'].

param priority
• priority of the Task. Tasks with lower priority will be processed earlier. By default each new task is assigned with random priority from (80, 100) range.
param priority_set_explicitly

- internal flag which tells if that
task priority was assigned manually or generated by spider according to priority generation
rules.

param network_try_count you’ll probably will not need to use it. It is used internally to
control how many times this task was restarted due to network errors. The Spider instance
has network_try_limit option. When network_try_count attribute of the task exceeds the
network_try_limit attribute then processing of the task is abandoned.

param task_try_count the as network_try_count but it increased only then you use clone
method. Also you can set it manually. It is useful if you want to restart the task after it was
cancelled due to multiple network errors. As you might guessed there is task_try_limit op-
tion in Spider instance. Both options network_try_count and network_try_limit guarantee
you that you’ll not get infinite loop of restarting some task.

param disable_cache if True disable cache subsystem. The document will be fetched from
the Network and it will not be saved to cache.

param refresh_cache if True the document will be fetched from the Network and saved to
cache.

param valid_status extra status codes which counts as valid

param use_proxystar it means to use proxylist which was configured via setup_proxystar
method of spider

param delay if specified tells the spider to schedule the task and execute it after delay sec-
onds

param raw if raw is True then the network response is forwarding to the corresponding
handler without any check of HTTP status code of network error, if raw is False (by de-
fault) then failed response is putting back to task queue or if tries limit is reached then the
processing of this request is finished.

param callback if you pass some function in callback option then the network response will
be passed to this callback and the usual ‘task_*’ handler will be ignored and no error will
be raised if such ‘task_*’ handler does not exist.

param fallback_name the name of method that is called when spider gives up to do the task
(due to multiple network errors)

Any non-standard named arguments passed to Task constructor will be saved as attributes of
the object. You can get their values later as attributes or with get method which allows to use default
value if attribute does not exist.

clone(**kwargs)
Clone Task instance.

Reset network_try_count, increase task_try_count. Reset priority attribute if it was not set explicitly.

get(key, default=None)
Return value of attribute or None if such attribute does not exist.
CHAPTER 4

Indices and tables

• genindex
• modindex
• search
g

grab.base, 46
grab.cookie, 48
grab.document, 50
grab.error, 47
grab.spider.base, 49
grab.spider.task, 54
grab.util.warning, 48
Symbols

<table>
<thead>
<tr>
<th>Method/Function</th>
<th>File/Module</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>getitem</strong>() (grab.cookie.CookieManager method)</td>
<td>48</td>
</tr>
<tr>
<td><strong>init</strong>() (grab.base.Grab method)</td>
<td>46</td>
</tr>
<tr>
<td><strong>init</strong>() (grab.cookie.CookieManager method)</td>
<td>48</td>
</tr>
<tr>
<td><strong>init</strong>() (grab.spider.task.Task method)</td>
<td>54</td>
</tr>
<tr>
<td>add_task() (grab.spider.base.Spider method)</td>
<td>49</td>
</tr>
<tr>
<td>adopt() (grab.base.Grab method)</td>
<td>46</td>
</tr>
<tr>
<td>BaseGrab (in module grab.base)</td>
<td>46</td>
</tr>
<tr>
<td>browse() (grab.document.Document method)</td>
<td>50</td>
</tr>
<tr>
<td>change_proxy() (grab.base.Grab method)</td>
<td>47</td>
</tr>
<tr>
<td>check_task_limits() (grab.spider.base.Spider method)</td>
<td>49</td>
</tr>
<tr>
<td>choose_form() (grab.document.Document method)</td>
<td>50</td>
</tr>
<tr>
<td>clear() (grab.cookie.CookieManager method)</td>
<td>48</td>
</tr>
<tr>
<td>clear_cookies() (grab.base.Grab method)</td>
<td>47</td>
</tr>
<tr>
<td>clone() (grab.base.Grab method)</td>
<td>46</td>
</tr>
<tr>
<td>clone() (grab.spider.task.Task method)</td>
<td>55</td>
</tr>
<tr>
<td>CookieManager (class in grab.cookie)</td>
<td>48</td>
</tr>
<tr>
<td>copy() (grab.document.Document method)</td>
<td>51</td>
</tr>
<tr>
<td>create_cookie() (in module grab.cookie)</td>
<td>48</td>
</tr>
<tr>
<td>dataNotFound (class in grab.error)</td>
<td>47</td>
</tr>
<tr>
<td>detect_charset() (grab.document.Document method)</td>
<td>51</td>
</tr>
<tr>
<td>Document (class in grab.document)</td>
<td>50</td>
</tr>
<tr>
<td>download() (grab.base.Grab method)</td>
<td>47</td>
</tr>
<tr>
<td>dump_config() (grab.base.Grab method)</td>
<td>46</td>
</tr>
<tr>
<td>dump_cookies() (grab.base.Grab method)</td>
<td>47</td>
</tr>
<tr>
<td>fake_response() (grab.base.Grab method)</td>
<td>47</td>
</tr>
<tr>
<td>form (grab.document.Document attribute)</td>
<td>51</td>
</tr>
<tr>
<td>form_fields() (grab.document.Document method)</td>
<td>51</td>
</tr>
<tr>
<td>get() (grab.spider.task.Task method)</td>
<td>55</td>
</tr>
<tr>
<td>get_dict() (grab.cookie.CookieManager method)</td>
<td>48</td>
</tr>
<tr>
<td>get_form_request() (grab.document.Document method)</td>
<td>51</td>
</tr>
<tr>
<td>go() (grab.base.Grab method)</td>
<td>46</td>
</tr>
<tr>
<td>Grab (class in grab.base)</td>
<td>46</td>
</tr>
<tr>
<td>grab.base (module)</td>
<td>46</td>
</tr>
<tr>
<td>grab.cookie (module)</td>
<td>48</td>
</tr>
<tr>
<td>grab.document (module)</td>
<td>50</td>
</tr>
<tr>
<td>grab.error (module)</td>
<td>47</td>
</tr>
<tr>
<td>grab.spider.base (module)</td>
<td>48, 49</td>
</tr>
<tr>
<td>grab.spider.task (module)</td>
<td>54</td>
</tr>
<tr>
<td>grab.util.warning (module)</td>
<td>48</td>
</tr>
<tr>
<td>GrabAuthError (class in grab.error)</td>
<td>47</td>
</tr>
<tr>
<td>GrabConnectionError (class in grab.error)</td>
<td>47</td>
</tr>
<tr>
<td>GrabDeprecationWarning (class in grab.util.warning)</td>
<td>48</td>
</tr>
<tr>
<td>GrabError (class in grab.error)</td>
<td>47</td>
</tr>
<tr>
<td>GrabInvalidUrl (class in grab.error)</td>
<td>48</td>
</tr>
<tr>
<td>GrabMisuseError (class in grab.error)</td>
<td>47</td>
</tr>
<tr>
<td>GrabNetworkError (class in grab.error)</td>
<td>47</td>
</tr>
<tr>
<td>GrabTimeoutError (class in grab.error)</td>
<td>47</td>
</tr>
<tr>
<td>GrabTooManyRedirectsError (class in grab.error)</td>
<td>48</td>
</tr>
<tr>
<td>is_valid_network_response_code()</td>
<td>(grab.spider.base.Spider method)</td>
</tr>
<tr>
<td>items() (grab.cookie.CookieManager method)</td>
<td>48</td>
</tr>
<tr>
<td>json (grab.document.Document attribute)</td>
<td>52</td>
</tr>
<tr>
<td>load_config() (grab.base.Grab method)</td>
<td>46</td>
</tr>
<tr>
<td>load_cookies() (grab.base.Grab method)</td>
<td>47</td>
</tr>
<tr>
<td>load_from_file() (grab.cookie.CookieManager method)</td>
<td>48</td>
</tr>
<tr>
<td>load_proxylist() (grab.base.Grab method)</td>
<td>47</td>
</tr>
<tr>
<td>load_proxylist() (grab.spider.base.Spider method)</td>
<td>49</td>
</tr>
</tbody>
</table>
M
make_url_absolute() (grab.base.Grab method), 47

P
parse() (grab.document.Document method), 52
prepare() (grab.spider.base.Spider method), 49
process_grab_proxy() (grab.spider.base.Spider method),
49
process_next_page() (grab.spider.base.Spider method),
49
pyquery (grab.document.Document attribute), 52

Q
query_param() (grab.document.Document method), 52

R
read_bom() (in module grab.document), 54
request() (grab.base.Grab method), 47
reset() (grab.base.Grab method), 46
rex_assert() (grab.document.Document method), 52
rex_search() (grab.document.Document method), 52
rex_text() (grab.document.Document method), 52

S
save() (grab.document.Document method), 52
save_hash() (grab.document.Document method), 52
save_to_file() (grab.cookie.CookieManager method), 48
set() (grab.cookie.CookieManager method), 48
set_input() (grab.document.Document method), 53
set_input_by_id() (grab.document.Document method), 53
set_input_by_number() (grab.document.Document method), 53
set_input_by_xpath() (grab.document.Document method), 53
setup() (grab.base.Grab method), 46
setup_cache() (grab.spider.base.Spider method), 50
setup_queue() (grab.spider.base.Spider method), 50
setup_transport() (grab.base.Grab method), 46
setup_with_proxyline() (grab.base.Grab method), 47
shutdown() (grab.spider.base.Spider method), 50
Spider (class in grab.spider.base), 49
stop() (grab.spider.base.Spider method), 50

T
Task (class in grab.spider.task), 54
task_generator() (grab.spider.base.Spider method), 50
text_assert() (grab.document.Document method), 53
text_assert_any() (grab.document.Document method), 53
text_search() (grab.document.Document method), 53
tree (grab.document.Document attribute), 54

U
unicode_body() (grab.document.Document method), 54