Grab Documentation

Release 0.6.41

Gregory Petukhov

Jul 01, 2023

CONTENTS

CHAPTER

ONE

USEFUL LINKS

- Source code: https://github.com/lorien/grab
- Documentation: https://grab.readthedocs.io/en/latest/
- Russian Web Scraping Chat Group: https://t.me/grablab_ru
- English Web Scraping Chat Group: https://t.me/grablab

CHAPTER

TWO

WHAT IS GRAB?

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 built on top of urllib3 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

THREE

TABLE OF CONTENTS

3.1 Grab User Manual

3.1.1 Grab Installation

Installation on Linux

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 dependency. To build lxml successfully you need to install:

```
apt-get install libxml2-dev libxslt-dev
```

Installation on Windows

If you have problems with installing lxml with pip try to manually install lxml with packages available here http://www.lfd.uci.edu/~gohlke/pythonlibs/#lxml I

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 and then run the command

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

Run the command:

```
.. code:: shell
```

    python -m pip install grab

Installation on MacOS

Run the command:

```
pip install -U grab
```

Dependencies

All required dependencies should be installed automatically if you install Grab with pip. Actual list of grab dependencies is stored in https://github.com/lorien/grab/blob/master/pyproject.toml in "dependencies" key.

3.1.2 Testing Grab Framework

Building test environment

Run command:

```
make bootstrap
```

Run tests

Run all tests in 30 parallel threads:

```
pytest -n30
```

Run all tests in 30 parallel threads, fail on first error:

```
pytest -n30 -x
```

Same as previous but via make command:

```
make pytest
```

Run specific test:

```
pytest tests/test_util_types.py
```

Run all test cases which has "redis" in their name:

```
pytest -k redis
```

Github 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://github.com/lorien/grab/actions

Test Coverage

To see test coverage just run full set of tests through pytest

```
pytest -n30 -x --cov grab --cov-report term-missing
```

You can use shortcut:

```
make pytest
```

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/github/lorien/grab

Linters

Run with:

```
make check
```

That will run mypy, pylint and flake8 linters.

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
```

Let's get started with some simple examples.

Make a request

First, you need to import the Grab class:

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

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

```
>>> from grab import Grab
>>> g = Grab()
>>> resp = g.request('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:

```
>>> 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:

```
>>> g = Grab()
>>> g.setup(method='head')
>>> resp = g.request('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:

```
>>> g = Grab()
>>> g.setup(post={'username': 'Root', 'password': 'asd7DD&*ssd'})
>>> g.request('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:

```
>>> g = Grab()
>>> g.request('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.

Response Content

Consider a simple page retrieving example:

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

To get the response content as unicode use:

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

Note that grab will automatically detect the encoding (character set) of the response's content and convert it to unicode. Detected encoding is available through "encoding" attribute:

```
>>> resp.encoding
```

If you need the original response body then use:

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

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

```
>>> resp = g.request('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.request('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.request('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.request(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.request('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 encoding that will be used to convert unicode to byte string, then use *request_encoding* option.

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

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

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

PUT Request

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

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

Other Methods

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

```
g = Grab()
g.request('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:

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

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

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

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

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

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:

```
>>> 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:

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

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

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, GrabNetwork-Timeout is raised.

process_redirect

Type
bool

Default
True

Automatically process HTTP 30* redirects.

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.

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.

fields

Type
sequence of pairs or dict

Default
None

Data to be sent in serialized form. Serialization depends on the type of request. For GET on mulitpart post requests the "urlencode" method is used. By default POST/PUT requests are multipart.

multipart

Type
boolean

Default
True

Control if multipart encoding must be used for PUT/POST requests.

body

Type
bytes

Default
None

Raw bytes content to send

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.

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
None

The encoding (character set) is used to store document's content as bytes. By default Grab detects encoding of document automatically. If it detects the encoding incorrectly you can specify exact encoding with this option. The encoding option is used to convert document's bytes content into Unicode text also for biilding DOM tree of the document.

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.

Debugging

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:

```
>>> 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:

```
>>> import logging
>>> logger = logging.getLogger('grab')
>>> logger.addHandler(logging.StreamHandler())
>>> logger.setLevel(logging.DEBUG)
```

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.

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:

```
>>> 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.

3.1.9 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 *process_redirect* to False.

Let's see how it works:

```
>>> g = Grab()
>>> g.setup(process_redirect=False)
>>> g.request('http://google.com')
<grab.response.Response object at 0x1246ae0>
>>> g.response.code
301
>>> g.response.headers['Location']
'http://www.google.com/'
>>> g.setup(process_redirect=True)
>>> g.request('http://google.com')
<grab.response.Response object at 0x1246ae0>
>>> g.response.code
200
>>> g.response.url
'http://www.google.ru/?gws_rd=cr&ei=BspFUtS8EOWq4ATAooGADA'
```

Original and Destination URLs

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

```
>>> g = Grab()
>>> g.request('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=8spFUo32Huem4gT6ooDwAg'
```

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*:

```
>>> print g.response.head
HTTP/1.1 301 Moved Permanently
Location: http://www.google.com/
Content-Type: text/html; charset=UTF-8
Date: Fri, 27 Sep 2013 18:19:13 GMT
Expires: Sun, 27 Oct 2013 18:19:13 GMT
Cache-Control: public, max-age=2592000
Server: gws
Content-Length: 219
X-XSS-Protection: 1; mode=block
X-Frame-Options: SAMEORIGIN

HTTP/1.1 302 Found
Location: http://www.google.ru/?gws_rd=cr&ei=IsxFUp-8CsT64QTZooDwBA
Cache-Control: private
Content-Type: text/html; charset=UTF-8
Date: Fri, 27 Sep 2013 18:19:14 GMT
Server: gws
Content-Length: 258
X-XSS-Protection: 1; mode=block
X-Frame-Options: SAMEORIGIN

HTTP/1.1 200 OK
Date: Fri, 27 Sep 2013 18:19:14 GMT
Expires: -1
Cache-Control: private, max-age=0
Content-Type: text/html; charset=UTF-8
Content-Encoding: gzip
Server: gws
X-XSS-Protection: 1; mode=block
X-Frame-Options: SAMEORIGIN
Transfer-Encoding: chunked
```

3.1.10 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:

```
>>> g = Grab()
>>> g.request("http://ya.ru/")
>>> g.doc.set_input("text", "grab documenation")
>>> g.doc.submit()
>>> g.doc.select('//a[@class="b-serp-item__title-link"]/@href').text()
'https://grab.readthedocs.io/'
```

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.

3.1.11 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`.

3.1.12 HTML Document Encoding

Why does encoding matter?

By default, Grab automatically detects the encoding of the body of the HTML document. It uses this detected encoding 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 with *response.body* attribute. A unicode representation of the document body can be obtained by calling *response.unicode_body()*:

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

Encoding Detection Algorithm

Grab users https://github.com/lorien/unicodec library to detect encoding of the document. The unicodec library searches for encoding in following sources (from most priority to less):

- BOM (byte order mark)
- Content-Type HTTP header:

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

- HTML meta tag or XML declaration:

```
<meta charset="utf-8">
<meta name="http-equiv" content="text/html; encoding=cp1251" >
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
```

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

Setting the encoding manually

You can bypass automatic encoding detection and specify it manually with *encoding* option.

3.1.13 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.

Internally Grab instance stores cookies in *http.cookiejar:CookieJar* instance.

It is important to understand that Response object contains only cookies has been provided by the server in recent HTTP response. If you need all session cookies use "Grab:cookies" attribute.

If you want more granular control on custom cookies, you can use the *grab.util.cookies.create_cookie* function to create Cookie object and add its to Grab instance:

```
>>> from grab.util.cookies import create_cookie
>>> g = Grab()
>>> g.cookies.set_cookie(create_cookie(name='foo', value='bar', domain='yandex.ru', path=
↪'/host'))
```

3.1.14 Proxy Server Support

Basic Usage

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

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

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

```
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":

```
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']*:

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

Proxy List Support

Grab supports a work with a list of proxy servers. You need to manually create instance of ProxyList class from "proxylist" package (it is installed along Grab) and assign it to *g.proxylist* attribute. When you create ProxyList instance do not forget to specify the default proxy type ("socks5" or "http"), if proxy list does not provide information about proxy the Grab raises runtime error.

Loading proxy list from local file:

```
>>> from proxylist import Proxylist
>>> g = Grab()
>>> g.proxylist = ProxyList.from_local_file("var/proxy.txt", proxy_type="socks5")
```

Loading proxy list from network:

```
>>> from proxylist import Proxylist
>>> g = Grab()
>>> g.proxylist = ProxyList.from_network_file("https://example.com/proxy.txt", proxy_
↪type="socks5")
```

For more information about ProxyList class please refer to https://proxylist.readthedocs.io

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:

```
>>> from grab import Grab
>>> import logging
>>> logging.basicConfig(level=logging.DEBUG)
>>> g = Grab()
>>> g.proxylist = ProxyList.from_local_file("/web/proxy.txt", proxy_type="http")
>>> g.request('http://yandex.ru/')
DEBUG:grab.network:[02] GET http://yandex.ru/ via 91.210.101.31:8080 proxy of type http␣
↪with authorization
<grab.response.Response object at 0x109d9f0>
>>> g.request('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:

```
>>> from grab import Grab
>>> import logging
>>> g = Grab()
>>> g.proxylist = ProxyList.from_local_file("/web/proxy.txt", proxy_type="http")
>>> g.setup(proxy_auto_change=False)
>>> g.request('http://ya.ru')
DEBUG:grab.network:[04] GET http://ya.ru
<grab.response.Response object at 0x109de50>
>>> g.change_proxy()
>>> g.request('http://ya.ru')
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>
>>> g.request('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_server()*, 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_server()* to pick a random proxy from the proxy list.

3.1.15 Searching the response body

String search

With the *doc.text_search* method, you can find out if the response body contains a certain string or not:

```
>>> g = Grab('<h1>test</h1>')
>>> g.doc.text_search(u'tes')
True
```

If you prefer to raise an exception if string was not found, then use the *doc.text_assert* method:

```
>>> g = Grab('<h1>test</h1>')
>>> g.doc.text_assert(u'tez')
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
```

All text search methods understands both str and bytes queries. If you search for str the document body converted to str is used. If you search for bytes the original bytes document body is used.

Regexp Search

You can search for a regular expression with *doc.rex_search* method that accepts compiled regexp object or just a text of regular expression:

```
>>> g = Grab('<h1>test</h1>')
>>> g.doc.rex_search('<.+?>').group(0)
u'<h1>'
```

Method *doc.rex_text* returns you text contents of *.group(1)* of the found match object:

```
>>> g = Grab('<h1>test</h1>')
>>> g.doc.rex_text('<.+?>(.+)<')
u'test'
```

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/document.py", line 189, in rex_assert
    self.rex_search(rex)
  File "/home/lorien/web/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>
```

3.1.16 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.request('http://google.com')
<grab.doc.Document object at 0x2cff9f0>
>>> g.doc
<grab.doc.Document 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

encoding
: encoding 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.

download_size
: size of received data

upload_size
: size of uploaded data except the HTTP headers

Now, a real example:

```
>>> from grab import Grab
>>> g = Grab()
>>> g.request('http://wikipedia.org')
<grab.doc.Document object at 0x1ff99f0>
>>> g.doc.body[:100]
'<!DOCTYPE html>\n<html lang="mul" dir="ltr">\n<head>\n<!-- Sysops: Please do not edit␣
↪the main template'
>>> g.doc.code
200
>>> g.doc.headers['Content-Type']
'text/html; charset=utf-8'
>>> g.doc.encoding
```

(continues on next page)

(continued from previous page)

```
'utf-8'
>>> g.doc.cookies
<CookieJar[Cookie(...), Cookie(..)]>
>>> g.doc.url
'http://www.wikipedia.org/'
>>> g.doc.download_size
11100.0
>>> g.doc.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*.

3.1.17 Network Transport

Network transport is a component which utilize one of well known 3rd-party network packages to do network requests and retrieve network response. At the moment Grab supports only one network library: urllib3. You may access transport object with *Grab.transport* attribute. In most cases you do not need direct access to transport object.

Urllib3 transport

This transport also could be used in gevent environment. The urllib3 uses native python sockets that could be patched by *gevent.monkey.patch_all*.

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


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

(continues on next page)

(continued from previous page)

```
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 minimal example to build Grab transport powered by wget.

```
import email.message
from contextlib import contextmanager
from subprocess import check_output

from grab import Grab
from grab.base_transport import BaseTransport
from grab.document import Document


class WgetTransport(BaseTransport):
    def reset(self):
        pass

    def process_config(self, grab_config, cookies):
        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_config, *, document_class=Document):
        return document_class(
            grab_config=grab_config,
            body=self._response_body,
            headers=email.message.Message(),
        )

    @contextmanager
    def wrap_transport_error(self):
        yield
```

(continues on next page)

(continued from previous page)

```
g = Grab(transport=WgetTransport)
g.request("https://github.com")
print(g.doc("//title").text())
assert "github" in g.doc("//title").text().lower()
```

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.

Spider uses multiple python threads to process network reqeusts in parallel. In short, when you create new network request it is processed one of free network thread, when the response is ready 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.

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.

```
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
```

(continues on next page)

(continued from previous page)

```
        # `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
        # request to download the image.
        # Also we pass the info about pulication, we need it be
        # available in next handler.
        image_url = grab.xpath_text('//div[@class="b-image"]/a/img/@src')
        yield Task('image', url=image_url, post=task.post)

    def task_image(self, grab, task):
        print 'Image downloaded for %s' % task.post['title']

        # OK, this is last handler in our spider.
```

(continues on next page)

(continued from previous page)

```
        # We have received the content of image,
        # we need to save it.
        path = 'images/%s.jpg' % self.result_counter
        grab.response.save(path)
        self.result_file.writerow([
            task.post['url'].encode('utf-8'),
            task.post['title'].encode('utf-8'),
            path
        ])
        # Increment image counter
        self.result_counter += 1


if __name__ == '__main__':
    logging.basicConfig(level=logging.DEBUG)
    # Let's start spider with two network concurrent streams
    bot = ExampleSpider(thread_number=2)
    bot.run()
```

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:

```
    ...
    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 name of task and either URL or Request instance. Next, you see examples of different task creation. All three examples do the same:

```
# Using `url` argument
t = Task('wikipedia', url 'http://wikipedia.org/')

# Using Request instancec
t = Task('wikipedia', Request(url="http://wikipedia.org/"))
```

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.

```
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:

```
# 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 URl in this list the spider will create Task object with name "initial":

```
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:

```
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:

```
bot = ExampleSpider()
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:

```
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:

```
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*:

```
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')
        yield Task('page', grab_config=g.dump_config())
        # The grab instance in the yielded task
        # will not be affected by `create_grab_instance` method.
```

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 used redis and mongo backends.

In-memory backend:

```
bot = SomeSpider() # task queue is stored in memory
# OR (that is the same)
from grab.spider.queue_backend.memory import MemoryTaskQueue

bot = SomeSpider(task_queue=MemoryTaskQueue())
```

MongoDB backend:

```
from grab.spider.queue_backend.mongodb import MongodbTaskQueue

bot = SomeSpider(task_queue=MongodbTaskQueue())
```

Redis backend:

```
from grab.spider.queue_backend.redis import RedisTaskQueue

bot = SomeSpider(task_queue=RedisTaskQueue())
```

3.2.4 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:

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

Second way is to redefine *is_valid_network_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:

```
class SomeSpider(Spider):
    def is_valid_network_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 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.

```
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:

```
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*.

3.2.5 Explanation

Previously Grab library has been supporint multiple options for spider and grab transports. At current moment there are no options. Spider use only threaded transport. Grab uses only urllib3 transport.

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

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

At the moment Grab supports only one network library to send network requests: urllib3. You may access transport object with *Grab.transport* attribute. In most cases you do not need direct access to transport object.

```
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()
        url = grab.make_url_absolute(url)
        yield Task('link', url=url)

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

(continues on next page)

(continued from previous page)

```
logging.basicConfig(level=logging.DEBUG)
bot = SimpleSpider(transport='threaded', grab_transport='urllib3')
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 API Reference

This page contains auto-generated API reference documentation[1].

grab

Subpackages

grab.spider

Subpackages

grab.spider.queue_backend

Submodules

grab.spider.queue_backend.base

Module Contents

Classes

BaseTaskQueue

class grab.spider.queue_backend.base.**BaseTaskQueue**(***kwargs: Any*)

random_queue_name() → str

abstract put(*task:* grab.spider.task.Task, *priority: int*, *schedule_time: None | datetime.datetime = None*) → None

[1] Created with sphinx-autoapi

abstract get() → *grab.spider.task.Task*

Return *Task* object or raise *Queue.Empty* exception.

@returns: *grab.spider.task.Task* object @raises: *Queue.Empty* exception

abstract size() → int

abstract clear() → None

Remove all tasks from the queue.

abstract close() → None

grab.spider.queue_backend.memory

Module Contents

Classes

MemoryTaskQueue

class grab.spider.queue_backend.memory.**MemoryTaskQueue**

Bases: *grab.spider.queue_backend.base.BaseTaskQueue*

put(*task:* grab.spider.task.Task, *priority: int*, *schedule_time: None | datetime.datetime = None*) → None

get() → *grab.spider.task.Task*

Return *Task* object or raise *Queue.Empty* exception.

@returns: *grab.spider.task.Task* object @raises: *Queue.Empty* exception

size() → int

clear() → None

Remove all tasks from the queue.

close() → None

grab.spider.queue_backend.mongodb

Module Contents

Classes

MongodbTaskQueue

Attributes

LOG

grab.spider.queue_backend.mongodb.**LOG**

class grab.spider.queue_backend.mongodb.**MongodbTaskQueue**(*connection_args: None | dict[str, Any] = None*, *collection_name: None | str = None*, *database_name: str = 'grab_spider'*)

Bases: *grab.spider.queue_backend.base.BaseTaskQueue*

size() → int

put(*task:* grab.spider.task.Task, *priority: int*, *schedule_time: None | datetime.datetime = None*) → None

get() → *grab.spider.task.Task*

Return *Task* object or raise *Queue.Empty* exception.

@returns: *grab.spider.task.Task* object @raises: *Queue.Empty* exception

clear() → None

Remove all tasks from the queue.

close() → None

grab.spider.queue_backend.redis

Spider task queue backend powered by redis.

Module Contents

Classes

CustomPriorityQueue

RedisTaskQueue

Attributes

system_random

grab.spider.queue_backend.redis.**system_random**

class grab.spider.queue_backend.redis.**CustomPriorityQueue**(*key: str*, ***kwargs: Any*)

Bases: fastrq.priorityqueue.PriorityQueue

connect() → redis.Redis[Any]

clear() → None

class grab.spider.queue_backend.redis.**RedisTaskQueue**(*queue_name: None | str = None*, *connection_args: None | dict[str, Any] = None*)

Bases: *grab.spider.queue_backend.base.BaseTaskQueue*

put(*task:* grab.spider.task.Task, *priority: int*, *schedule_time: None | datetime.datetime = None*) → None

get() → *grab.spider.task.Task*

Return *Task* object or raise *Queue.Empty* exception.

@returns: *grab.spider.task.Task* object @raises: *Queue.Empty* exception

size() → int

clear() → None

Remove all tasks from the queue.

close() → None

grab.spider.service

Submodules

grab.spider.service.base

Module Contents

Classes

ServiceWorker
BaseService

Attributes

logger

grab.spider.service.base.**logger**

class grab.spider.service.base.**ServiceWorker**(*fatal_error_queue: queue.Queue[grab.spider.interface.FatalErrorQueueItem]*, *worker_callback: collections.abc.Callable[Ellipsis, Any]*)

build_thread_name(*worker_callback: collections.abc.Callable[Ellipsis, Any]*) → str

worker_callback_wrapper(*callback: collections.abc.Callable[Ellipsis, Any]*) → collections.abc.Callable[Ellipsis, None]

start() → None

stop() → None

process_pause_signal() → None

pause() → None

resume() → None

is_alive() → bool

class grab.spider.service.base.**BaseService**(*fatal_error_queue: queue.Queue[grab.spider.interface.FatalErrorQueueItem]*)

create_worker(*worker_action: collections.abc.Callable[Ellipsis, None]*) → *ServiceWorker*

iterate_workers(*objects: list[*ServiceWorker*]*) → collections.abc.Iterable[*ServiceWorker*]

start() → None

stop() → None

pause() → None

resume() → None

register_workers(**args: Any*) → None

is_busy() → bool

is_alive() → bool

grab.spider.service.network

Module Contents

Classes

BaseNetworkService
NetworkServiceThreaded

Attributes

NetworkResult

grab.spider.service.network.**NetworkResult**

class grab.spider.service.network.**BaseNetworkService**(*fatal_error_queue: queue.Queue[grab.spider.interface.FatalErrorQueueItem]*)

Bases: *grab.spider.service.base.BaseService*

abstract get_active_threads_number() → int

class grab.spider.service.network.**NetworkServiceThreaded**(*fatal_error_queue: queue.Queue[grab.spider.interface.FatalErrorQueueItem], thread_number: int, process_task: collections.abc.Callable[[*grab.spider.task.Task*], None], get_task_from_queue: collections.abc.Callable[[], None | Literal[True] |* grab.spider.task.Task*]*)

Bases: *BaseNetworkService*

get_active_threads_number() → int

worker_callback(*worker:* grab.spider.service.base.ServiceWorker) → None

grab.spider.service.parser

Module Contents

Classes

ParserService

class grab.spider.service.parser.**ParserService**(*fatal_error_queue: queue.Queue[grab.spider.interface.FatalErrorQueueItem], pool_size: int, task_dispatcher:* grab.spider.service.task_dispatcher.TaskDispatcherService*, stat: procstat.Stat, parser_requests_per_process: int, find_task_handler: collections.abc.Callable[[*grab.spider.task.Task*], collections.abc.Callable[Ellipsis, None]]*)

Bases: *grab.spider.service.base.BaseService*

check_pool_health() → None

supervisor_callback(*worker:* grab.spider.service.base.ServiceWorker) → None

worker_callback(*worker:* grab.spider.service.base.ServiceWorker) → None

execute_task_handler(*handler: collections.abc.Callable[[grab.Grab, grab.spider.task.Task], None]*, *result: grab.spider.service.network.NetworkResult*, *task:* grab.spider.task.Task) → None

grab.spider.service.task_dispatcher

Module Contents

Classes

TaskDispatcherService

class grab.spider.service.task_dispatcher.**TaskDispatcherService**(*fatal_error_queue: queue.Queue[grab.spider.interface.FatalErrorQueu*, *process_service_result: collections.abc.Callable[[Any, grab.spider.task.Task, None | dict[str, Any]], Any]*)

Bases: *grab.spider.service.base.BaseService*

start() → None

worker_callback(*worker:* grab.spider.service.base.ServiceWorker) → None

grab.spider.service.task_generator

Module Contents

Classes

TaskGeneratorService

class grab.spider.service.task_generator.**TaskGeneratorService**(*fatal_error_queue: queue.Queue[grab.spider.interface.FatalErrorQueueI*, *real_generator: collections.abc.Iterator[grab.spider.task.Task]*, *thread_number: int*, *get_task_queue: collections.abc.Callable[[], grab.spider.queue_backend.base.BaseTaskQueue]*, *parser_service: grab.spider.service.parser.ParserService*, *task_dispatcher: grab.spider.service.task_dispatcher.TaskDispatcherSer*

Bases: *grab.spider.service.base.BaseService*

worker_callback(*worker:* grab.spider.service.base.ServiceWorker) → None

Submodules

grab.spider.base

Module Contents

Classes

Spider	Asynchronous scraping framework.

Attributes

DEFAULT_TASK_PRIORITY
DEFAULT_NETWORK_STREAM_NUMBER
DEFAULT_TASK_TRY_LIMIT
DEFAULT_NETWORK_TRY_LIMIT
RANDOM_TASK_PRIORITY_RANGE
logger
system_random
HTTP_STATUS_ERROR
HTTP_STATUS_NOT_FOUND
WAIT_SERVICE_SHUTDOWN_SEC

grab.spider.base.**DEFAULT_TASK_PRIORITY = 100**

grab.spider.base.**DEFAULT_NETWORK_STREAM_NUMBER = 3**

grab.spider.base.**DEFAULT_TASK_TRY_LIMIT = 5**

grab.spider.base.**DEFAULT_NETWORK_TRY_LIMIT = 5**

grab.spider.base.**RANDOM_TASK_PRIORITY_RANGE = (50, 100)**

grab.spider.base.**logger**

grab.spider.base.**system_random**

grab.spider.base.**HTTP_STATUS_ERROR = 400**

grab.spider.base.**HTTP_STATUS_NOT_FOUND = 404**

grab.spider.base.**WAIT_SERVICE_SHUTDOWN_SEC = 10**

class grab.spider.base.**Spider**(*task_queue: None | grab.spider.queue_backend.base.BaseTaskQueue = None, thread_number: None | int = None, network_try_limit: None | int = None, task_try_limit: None | int = None, priority_mode: str = 'random', meta: None | dict[str, Any] = None, config: None | dict[str, Any] = None, parser_requests_per_process: int = 10000, parser_pool_size: int = 1, network_service: None | grab.spider.service.network.BaseNetworkService = None, grab_transport: None | grab.base.BaseTransport[grab.request.HttpRequest, grab.document.Document] | type[grab.base.BaseTransport[grab.request.HttpRequest, grab.document.Document]] = None*)

Asynchronous scraping framework.

spider_name

initial_urls: list[str] = []

collect_runtime_event(*name: str*, *value: None | str*) → None

setup_queue(**_args: Any*, ***_kwargs: Any*) → None

Set up queue.

add_task(*task:* grab.spider.task.Task, *queue: None* | grab.spider.queue_backend.base.BaseTaskQueue = *None*, *raise_error: bool = False*) → bool

Add task to the task queue.

stop() → None

Instruct spider to stop processing new tasks and start shutting down.

load_proxylist(*source: str | proxylist.base.BaseProxySource*, *source_type: None | str = None*, *proxy_type: str = 'http'*, *auto_init: bool = True*, *auto_change: bool = True*) → None

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 automatically random proxy rotation will be used.

Proxy source format should be one of the following (for each line): - ip:port - ip:port:login:password

render_stats() → str

prepare() → None

Do additional spider customization here.

This method runs before spider has started working.

shutdown() → None

Override this method to do some final actions after parsing has been done.

create_grab_instance(***kwargs: Any*) → *grab.Grab*

task_generator() → collections.abc.Iterator[*grab.spider.task.Task*]

You can override this method to load new tasks.

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

check_task_limits(*task:* grab.spider.task.Task) → tuple[bool, str]

Check that task's network & try counters do not exceed limits.

Returns: * if success: (True, None) * if error: (False, reason)

generate_task_priority() → int

process_initial_urls() → None

get_task_from_queue() → None | Literal[True] | *grab.spider.task.Task*

is_valid_network_response_code(*code: int*, *task:* grab.spider.task.Task) → bool

Test if response is valid.

Valid response is handled with associated task handler. Failed respoosne is processed with error handler.

process_parser_error(*func_name: str*, *task:* grab.spider.task.Task, *exc_info: tuple[type[Exception], Exception, types.TracebackType]*) → None

find_task_handler(*task:* grab.spider.task.Task) → collections.abc.Callable[Ellipsis, Any]

log_network_result_stats(*res: grab.spider.service.network.NetworkResult*, *task:* grab.spider.task.Task) → None

process_grab_proxy(*task:* grab.spider.task.Task, *grab:* grab.Grab) → None

Assign new proxy from proxylist to the task.

change_active_proxy(*task:* grab.spider.task.Task, *grab:* grab.Grab) → None

get_task_queue() → *grab.spider.queue_backend.base.BaseTaskQueue*

is_idle_estimated() → bool

is_idle_confirmed(*services: list[*grab.spider.service.base.BaseService*]*) → bool

Test if spider is fully idle.

WARNING: As side effect it stops all services to get state of queues anaffected by sercies.

Spider is full idle when all conditions are met: * all services are paused i.e. the do not change queues * all queues are empty * task generator is completed

run() → None

shutdown_services(*services: list[*grab.spider.service.base.BaseService*]*) → None

log_failed_network_result(*res: grab.spider.service.network.NetworkResult*) → None

log_rejected_task(*task:* grab.spider.task.Task, *reason: str*) → None

get_fallback_handler(*task:* grab.spider.task.Task) → None | collections.abc.Callable[Ellipsis, Any]

srv_process_service_result(*result:* grab.spider.task.Task | *None* | *Exception* | *dict[str, Any]*, *task:* grab.spider.task.Task, *meta: None* | *dict[str, Any]* = *None*) → None

Process result submitted from any service to task dispatcher service.

Result could be: * Task * None * Task instance * ResponseNotValidError-based exception * Arbitrary exception * Network response:

{ok, ecode, emsg, exc, grab, grab_config_backup}

Exception can come only from parser_service and it always has meta {"from": "parser", "exc_info": <...>}

srv_process_network_result(*result: grab.spider.service.network.NetworkResult*, *task:* grab.spider.task.Task) → None

srv_process_task(*task:* grab.spider.task.Task) → None

grab.spider.errors

Module Contents

exception grab.spider.errors.**SpiderError**

Bases: *grab.errors.GrabError*

Base class for Spider exceptions.

exception grab.spider.errors.**SpiderMisuseError**

Bases: *SpiderError*

Improper usage of Spider framework.

exception grab.spider.errors.**FatalError**

Bases: *SpiderError*

Fatal error which should stop parsing process.

exception grab.spider.errors.**SpiderInternalError**

Bases: *SpiderError*

Raises when error throwned by internal spider logic.

Like spider class discovering, CLI error.

exception grab.spider.errors.**NoTaskHandlerError**

Bases: *SpiderError*

Raise when no handler found to process network response.

exception grab.spider.errors.**NoDataHandlerError**

Bases: *SpiderError*

Raise when no handler found to process Data object.

grab.spider.interface

Module Contents

grab.spider.interface.**FatalErrorQueueItem**

grab.spider.task

Module Contents

Classes

BaseTask	
Task	Task for spider.

class grab.spider.task.**BaseTask**

class grab.spider.task.**Task**(*name: None | str = None*, *url: None | str | grab.request.HttpRequest = None*, *request: None | grab.request.HttpRequest = None*, *priority: None | int = None*, *priority_set_explicitly: bool = True*, *network_try_count: int = 0*, *task_try_count: int = 1*, *valid_status: None | list[int] = None*, *use_proxylist: bool = True*, *delay: None | float = None*, *raw: bool = False*, *callback: None | collections.abc.Callable[Ellipsis, None] = None*, *fallback_name: None | str = None*, *store: None | dict[str, Any] = None*, ***kwargs: Any*)

Bases: *BaseTask*

Task for spider.

check_init_kwargs(*kwargs: collections.abc.Mapping[str, Any]*) → None

get(*key: str*, *default: Any = None*) → Any

Return value of attribute or None if such attribute does not exist.

process_delay_option(*delay: None | float*) → None

clone(*url: None | str = None*, *request: None | grab.request.HttpRequest = None*, ***kwargs: Any*) → *Task*

Clone Task instance.

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

__repr__() → str

Return repr(self).

__lt__(*other:* Task) → bool

Return self<value.

__eq__(*other: object*) → bool

Return self==value.

Package Contents

Classes

Spider	Asynchronous scraping framework.
Task	Task for spider.

class grab.spider.**Spider**(*task_queue: None | grab.spider.queue_backend.base.BaseTaskQueue = None*, *thread_number: None | int = None*, *network_try_limit: None | int = None*, *task_try_limit: None | int = None*, *priority_mode: str = 'random'*, *meta: None | dict[str, Any] = None*, *config: None | dict[str, Any] = None*, *parser_requests_per_process: int = 10000*, *parser_pool_size: int = 1*, *network_service: None | grab.spider.service.network.BaseNetworkService = None*, *grab_transport: None | grab.base.BaseTransport[grab.request.HttpRequest, grab.document.Document] | type[grab.base.BaseTransport[grab.request.HttpRequest, grab.document.Document]] = None*)

Asynchronous scraping framework.

spider_name

initial_urls: list[str] = []

collect_runtime_event(*name: str*, *value: None | str*) → None

setup_queue(**_args: Any*, ***_kwargs: Any*) → None

Set up queue.

add_task(*task: grab.spider.task.Task*, *queue: None | grab.spider.queue_backend.base.BaseTaskQueue = None*, *raise_error: bool = False*) → bool

Add task to the task queue.

stop() → None

Instruct spider to stop processing new tasks and start shutting down.

load_proxylist(*source: str | proxylist.base.BaseProxySource*, *source_type: None | str = None*, *proxy_type: str = 'http'*, *auto_init: bool = True*, *auto_change: bool = True*) → None

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 automatically random proxy rotation will be used.

Proxy source format should be one of the following (for each line): - ip:port - ip:port:login:password

render_stats() → str

prepare() → None

Do additional spider customization here.

This method runs before spider has started working.

shutdown() → None

Override this method to do some final actions after parsing has been done.

create_grab_instance(***kwargs: Any*) → *grab.Grab*

task_generator() → collections.abc.Iterator[*grab.spider.task.Task*]

You can override this method to load new tasks.

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

check_task_limits(*task:* grab.spider.task.Task) → tuple[bool, str]

Check that task's network & try counters do not exceed limits.

Returns: * if success: (True, None) * if error: (False, reason)

generate_task_priority() → int

process_initial_urls() → None

get_task_from_queue() → None | Literal[True] | *grab.spider.task.Task*

is_valid_network_response_code(*code: int*, *task:* grab.spider.task.Task) → bool

Test if response is valid.

Valid response is handled with associated task handler. Failed respoosne is processed with error handler.

process_parser_error(*func_name: str*, *task:* grab.spider.task.Task, *exc_info: tuple[type[Exception], Exception, types.TracebackType]*) → None

find_task_handler(*task:* grab.spider.task.Task) → collections.abc.Callable[Ellipsis, Any]

log_network_result_stats(*res: grab.spider.service.network.NetworkResult*, *task:* grab.spider.task.Task) → None

process_grab_proxy(*task:* grab.spider.task.Task, *grab:* grab.Grab) → None

Assign new proxy from proxylist to the task.

change_active_proxy(*task:* grab.spider.task.Task, *grab:* grab.Grab) → None

get_task_queue() → *grab.spider.queue_backend.base.BaseTaskQueue*

is_idle_estimated() → bool

is_idle_confirmed(*services: list[*grab.spider.service.base.BaseService*]*) → bool

Test if spider is fully idle.

WARNING: As side effect it stops all services to get state of queues anaffected by sercies.

Spider is full idle when all conditions are met: * all services are paused i.e. the do not change queues * all queues are empty * task generator is completed

run() → None

shutdown_services(*services: list[*grab.spider.service.base.BaseService*]*) → None

log_failed_network_result(*res: grab.spider.service.network.NetworkResult*) → None

log_rejected_task(*task:* grab.spider.task.Task, *reason: str*) → None

get_fallback_handler(*task:* grab.spider.task.Task) → None | collections.abc.Callable[Ellipsis, Any]

srv_process_service_result(*result:* grab.spider.task.Task | *None* | *Exception* | *dict[str, Any]*, *task:* grab.spider.task.Task, *meta: None* | *dict[str, Any]* = *None*) → None

Process result submitted from any service to task dispatcher service.

Result could be: * Task * None * Task instance * ResponseNotValidError-based exception * Arbitrary exception * Network response:

{ok, ecode, emsg, exc, grab, grab_config_backup}

Exception can come only from parser_service and it always has meta {"from": "parser", "exc_info": <...>}

srv_process_network_result(*result: grab.spider.service.network.NetworkResult*, *task:* grab.spider.task.Task) → None

srv_process_task(*task:* grab.spider.task.Task) → None

class grab.spider.**Task**(*name: None* | *str* = *None*, *url: None* | *str* | grab.request.HttpRequest = *None*, *request: None* | grab.request.HttpRequest = *None*, *priority: None* | *int* = *None*, *priority_set_explicitly: bool* = *True*, *network_try_count: int* = *0*, *task_try_count: int* = *1*, *valid_status: None* | *list[int]* = *None*, *use_proxylist: bool* = *True*, *delay: None* | *float* = *None*, *raw: bool* = *False*, *callback: None* | *collections.abc.Callable[Ellipsis, None]* = *None*, *fallback_name: None* | *str* = *None*, *store: None* | *dict[str, Any]* = *None*, ***kwargs: Any*)

Bases: BaseTask

Task for spider.

check_init_kwargs(*kwargs: collections.abc.Mapping[str, Any]*) → None

get(*key: str*, *default: Any* = *None*) → Any

Return value of attribute or None if such attribute does not exist.

process_delay_option(*delay: None* | *float*) → None

clone(*url: None* | *str* = *None*, *request: None* | grab.request.HttpRequest = *None*, ***kwargs: Any*) → *Task*

Clone Task instance.

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

__repr__() → str

Return repr(self).

__lt__(*other:* Task) → bool

Return self<value.

__eq__(*other: object*) → bool

Return self==value.

grab.util

Submodules

grab.util.cookies

THe module provides things to operate with cookies.

Manuals:

- http://docs.python.org/2/library/cookielib.html#cookie-objects

Some code got from
https://github.com/kennethreitz/requests/blob/master/requests/cookies.py

Module Contents

Classes

MockRequest	Wraps a *requests.Request* to mimic a *urllib2.Request*.
MockResponse	Wraps a *httplib.HTTPMessage* to mimic a *urllib.addinfourl*.

Functions

create_cookie(→ http.cookiejar.Cookie)	Create cookielib.Cookie instance.
build_cookie_header(→ None | str)	Build HTTP Cookie header value for given cookies.
build_jar(→ http.cookiejar.CookieJar)	
extract_response_cookies(...)	

class grab.util.cookies.**MockRequest**(*url: str*, *headers: dict[str, str]*)

Wraps a *requests.Request* to mimic a *urllib2.Request*.

The code in *cookielib.CookieJar* expects this interface in order to correctly manage cookie policies, i.e., determine whether a cookie can be set, given the domains of the request and the cookie. The original request object is read-only. The client is responsible for collecting the new headers via *get_new_headers()* and interpreting them appropriately. You probably want *get_cookie_header*, defined below.

property unverifiable: bool

property origin_req_host: str

property host: str

get_type() → str

get_host() → str

get_origin_req_host() → str

get_full_url() → str

is_unverifiable() → bool

has_header(*name: str*) → bool

get_header(*name: str*, *default: Any = None*) → str

abstract add_header(*key: str*, *val: str*) → None

Cookielib has no legitimate use for this method.

Add it back if you find one.

add_unredirected_header(*name: str*, *value: str*) → None

get_new_headers() → dict[str, str]

class grab.util.cookies.**MockResponse**(*headers: http.client.HTTPMessage | urllib3._collections.HTTPHeaderDict*)

Wraps a *httplib.HTTPMessage* to mimic a *urllib.addinfourl*.

...what? Basically, expose the parsed HTTP headers from the server response the way *cookielib* expects to see them.

info() → http.client.HTTPMessage | urllib3._collections.HTTPHeaderDict

grab.util.cookies.**create_cookie**(**, name: str, value: str, domain: str, comment: None | str = None, comment_url: None | str = None, discard: bool = True, domain_initial_dot: None | bool = None, domain_specified: None | bool = None, expires: None | int = None, path: str = '/', path_specified: None | bool = None, port: None | int = None, port_specified: None | bool = None, rest: None | dict[str, Any] = None, rfc2109: bool = False, secure: bool = False, version: int = 0, httponly: None | bool = None*) → http.cookiejar.Cookie

Create cookielib.Cookie instance.

grab.util.cookies.**build_cookie_header**(*cookiejar: http.cookiejar.CookieJar, url: str, headers: collections.abc.Mapping[str, str]*) → None | str

Build HTTP Cookie header value for given cookies.

grab.util.cookies.**build_jar**(*cookies: collections.abc.Sequence[http.cookiejar.Cookie]*) → http.cookiejar.CookieJar

grab.util.cookies.**extract_response_cookies**(*req_url: str, req_headers: collections.abc.Mapping[str, Any] | http.client.HTTPMessage | urllib3._collections.HTTPHeaderDict, response_headers: http.client.HTTPMessage | urllib3._collections.HTTPHeaderDict*) → collections.abc.Sequence[http.cookiejar.Cookie]

grab.util.html

Module Contents

Functions

find_base_url(→ None | str)	Find value of attribute "url" in "<base>" tag.

Attributes

RE_BASE_URL

grab.util.html.**RE_BASE_URL**

grab.util.html.**find_base_url**(*content: str*) → None | str

Find value of attribute "url" in "<base>" tag.

grab.util.metrics

Module Contents

Functions

in_unit(→ int | float)

format_traffic_value(→ str)

Attributes

KB

MB

GB

grab.util.metrics.**KB = 1000**

grab.util.metrics.**MB**

grab.util.metrics.**GB**

grab.util.metrics.**in_unit**(*num: int*, *unit: str*) → int | float

grab.util.metrics.**format_traffic_value**(*num: int*) → str

grab.util.structures

Module Contents

Functions

merge_with_dict(→ collections.abc.MutableMapping[str, Any])	

grab.util.structures.**merge_with_dict**(*hdr1: collections.abc.MutableMapping[str, Any]*, *hdr2: collections.abc.Mapping[str, Any]*, *replace: bool*) → collections.abc.MutableMapping[str, Any]

grab.util.timeout

Module Contents

Classes

UndefinedParam	Generic enumeration.
Timeout	

Attributes

UNDEFINED_PARAM
DEFAULT_TOTAL_TIMEOUT

class grab.util.timeout.**UndefinedParam**

Bases: enum.Enum

Generic enumeration.

Derive from this class to define new enumerations.

value

grab.util.timeout.**UNDEFINED_PARAM: *UndefinedParam***

grab.util.timeout.**DEFAULT_TOTAL_TIMEOUT**

class grab.util.timeout.**Timeout**(*total: None | float | UndefinedParam = UNDEFINED_PARAM*, *connect: None | float | UndefinedParam = UNDEFINED_PARAM*, *read: None | float | UndefinedParam = UNDEFINED_PARAM*)

__slots__ = ['total', 'connect', 'read']

__repr__() → str
Return repr(self).

grab.util.types

Types used in Grab projects and utility functions to deal with these types.

I can not build generic function for combined logic of resolve_transport_entity and resolve_grab_entity because mypy does not allow to parametrise Generic with base class.

Module Contents

Functions

resolve_entity(→ T)	

Attributes

T	

grab.util.types.**T**

grab.util.types.**resolve_entity**(*base_type: type[T]*, *entity: None | T | type[T]*, *default: type[T]*) → T

Submodules

grab.base

Module Contents

Classes

BaseRequest	
BaseExtension	Abstract base class for generic types.
BaseClient	Abstract base class for generic types.
BaseTransport	Abstract base class for generic types.

class grab.base.**BaseRequest**

init_keys: set[str]

__repr__() → str
Return repr(self).

classmethod create_from_mapping(*mapping: collections.abc.Mapping[str, Any]*) → RequestT

class grab.base.**BaseExtension**

Bases: Generic[RequestT, ResponseT]

Abstract base class for generic types.

A generic type is typically declared by inheriting from this class parameterized with one or more type variables. For example, a generic mapping type might be defined as:

```
class Mapping(Generic[KT, VT]):
    def __getitem__(self, key: KT) -> VT:
        ...
    # Etc.
```

This class can then be used as follows:

```
def lookup_name(mapping: Mapping[KT, VT], key: KT, default: VT) -> VT:
    try:
        return mapping[key]
    except KeyError:
        return default
```

ext_handlers: collections.abc.Mapping[str, collections.abc.Callable[Ellipsis, Any]]

registry: collections.abc.MutableMapping[str, tuple[type[*BaseClient*[RequestT, ResponseT]], *BaseExtension*[RequestT, ResponseT]]]

__slots__ = ('owners',)

__set_name__(*owner: type[*BaseClient*[RequestT, ResponseT]]*, *attr: str*) → None

abstract reset() → None

classmethod get_extensions(*obj:* BaseClient*[RequestT, ResponseT]*) → collections.abc.Sequence[tuple[str, *BaseExtension*[RequestT, ResponseT]]]

class grab.base.**BaseClient**(*transport: None* | BaseTransport*[RequestT, ResponseT]* | *type[*BaseTransport*[RequestT, ResponseT]] = None*)

Bases: Generic[RequestT, ResponseT]

Abstract base class for generic types.

A generic type is typically declared by inheriting from this class parameterized with one or more type variables. For example, a generic mapping type might be defined as:

```
class Mapping(Generic[KT, VT]):
    def __getitem__(self, key: KT) -> VT:
        ...
    # Etc.
```

This class can then be used as follows:

```
def lookup_name(mapping: Mapping[KT, VT], key: KT, default: VT) -> VT:
    try:
        return mapping[key]
    except KeyError:
        return default
```

abstract property request_class: type[RequestT]

abstract property default_transport_class: type[*BaseTransport*[RequestT, ResponseT]]

__slots__ = ('transport', 'ext_handlers')

transport: *BaseTransport*[RequestT, ResponseT]

ext_handlers: collections.abc.Mapping[str, list[collections.abc.Callable[Ellipsis, Any]]]

abstract process_request_result(*req: RequestT*) → ResponseT

request(*req: None | RequestT = None*, ***request_kwargs: Any*) → ResponseT

clone() → T

class grab.base.**BaseTransport**

Bases: Generic[RequestT, ResponseT]

Abstract base class for generic types.

A generic type is typically declared by inheriting from this class parameterized with one or more type variables. For example, a generic mapping type might be defined as:

```
class Mapping(Generic[KT, VT]):
    def __getitem__(self, key: KT) -> VT:
        ...
    # Etc.
```

This class can then be used as follows:

```
def lookup_name(mapping: Mapping[KT, VT], key: KT, default: VT) -> VT:
    try:
        return mapping[key]
    except KeyError:
        return default
```

__slots__ = ()

abstract reset() → None

abstract prepare_response(*req: RequestT*, ***, *document_class: type[ResponseT]*) → ResponseT

abstract wrap_transport_error() → collections.abc.Generator[None, None, None]

abstract request(*req: RequestT*) → None

classmethod resolve_entity(*entity: None | BaseTransport[RequestT, ResponseT] | type[BaseTransport[RequestT, ResponseT]]*, *default: type[BaseTransport[RequestT, ResponseT]]*) → *BaseTransport*[RequestT, ResponseT]

grab.client

Module Contents

Classes

HttpClient	Abstract base class for generic types.

Functions

request(→ grab.document.Document)

class grab.client.**HttpClient**(*transport: None | BaseTransport[RequestT, ResponseT] | type[BaseTransport[RequestT, ResponseT]] = None*)

Bases: *grab.base.BaseClient*[*grab.request.HttpRequest*, *grab.document.Document*]

Abstract base class for generic types.

A generic type is typically declared by inheriting from this class parameterized with one or more type variables. For example, a generic mapping type might be defined as:

```
class Mapping(Generic[KT, VT]):
    def __getitem__(self, key: KT) -> VT:
        ...
    # Etc.
```

This class can then be used as follows:

```
def lookup_name(mapping: Mapping[KT, VT], key: KT, default: VT) -> VT:
    try:
        return mapping[key]
    except KeyError:
        return default
```

document_class: type[*grab.document.Document*]

extension

request_class

default_transport_class

request(*req: None | str | grab.request.HttpRequest = None*, ***request_kwargs: Any*) → *grab.document.Document*

process_request_result(*req:* grab.request.HttpRequest) → *grab.document.Document*

Process result of real request performed via transport extension.

grab.client.**request**(*url: None | str | grab.request.HttpRequest = None*, *client: None | HttpClient | type[HttpClient] = None*, ***request_kwargs: Any*) → *grab.document.Document*

grab.document

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

Module Contents

Classes

FormRequestParams	dict() -> new empty dictionary
Document	Network response.

Functions

normalize_pairs(→ tions.abc.Sequence[tuple[str, ...]	collec-

Attributes

THREAD_STORAGE
logger
UNDEFINED

grab.document.**THREAD_STORAGE**

grab.document.**logger**

grab.document.**UNDEFINED**

class grab.document.**FormRequestParams**

Bases: TypedDict

dict() -> new empty dictionary dict(mapping) -> new dictionary initialized from a mapping object's

(key, value) pairs

dict(iterable) -> new dictionary initialized as if via:
d = {} for k, v in iterable:

d[k] = v

dict(kwargs) -> new dictionary initialized with the name=value pairs**
in the keyword argument list. For example: dict(one=1, two=2)

url: str

method: str

multipart: bool

fields: collections.abc.Sequence[tuple[str, Any]]

grab.document.**normalize_pairs**(*inp: collections.abc.Sequence[tuple[str, Any]] | collections.abc.Mapping[str, Any]*) → collections.abc.Sequence[tuple[str, Any]]

class grab.document.**Document**(*body: bytes*, *, *document_type: None | str = 'html'*, *head: None | bytes = None*, *headers: None | email.message.Message = None*, *encoding: None | str = None*, *code: None | int = None*, *url: None | str = None*, *cookies: None | collections.abc.Sequence[http.cookiejar.Cookie] = None*)

Bases: grab.base.BaseResponse

Network response.

property status: None | int

property json: Any

Return response body deserialized into JSON object.

property pyquery: Any

Return pyquery handler.

property body: bytes

property tree: lxml.etree._Element

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

property form: lxml.html.FormElement

Return default document's 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:

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

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

__slots__ = ('document_type', 'code', 'head', 'headers', 'url', 'cookies', 'encoding', '_bytes_body',...

__call__(*query: str*) → selection.SelectorList[lxml.etree._Element]

select(**args: Any*, ***kwargs: Any*) → selection.SelectorList[lxml.etree._Element]

process_encoding(*encoding: None | str = None*) → str

Process explicitly defined encoding or auto-detect it.

If encoding is explicitly defined, ensure it is a valid encoding the python can deal with. If encoding is not specified, auto-detect it.

Raises unicodec.InvalidEncodingName if explicitly set encoding is invalid.

copy() → *Document*

save(*path: str*) → None

Save response body to file.

url_details() → urllib.parse.SplitResult

Return result of urlsplit function applied to response url.

query_param(*key: str*) → str

Return value of parameter in query string.

browse() → None

Save response in temporary file and open it in GUI browser.

__getstate__() → collections.abc.Mapping[str, Any]

Reset cached lxml objects which could not be pickled.

__setstate__(*state: collections.abc.Mapping[str, Any]*) → None

text_search(*anchor: str | bytes*) → bool

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.

text_assert(*anchor: str | bytes*) → None

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

text_assert_any(*anchors: collections.abc.Sequence[str | bytes]*) → None

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

rex_text(*regexp: str | bytes | re.Pattern[str] | re.Pattern[bytes], flags: int = 0, default: Any = UNDEFINED*) → Any

Return content of first matching group of regexp found in response body.

rex_search(*regexp: str | bytes | re.Pattern[str] | re.Pattern[bytes], flags: int = 0, default: Any = UNDEFINED*) → Any

Search the regular expression in response body.

Return found match object or None

rex_assert(*rex: str | bytes | re.Pattern[str] | re.Pattern[bytes]*) → None

Raise *DataNotFound* exception if *rex* expression is not found.

get_body_chunk() → bytes

unicode_body() → str

Return response body as unicode string.

classmethod wrap_io(*inp: bytes | str*) → io.StringIO | io.BytesIO

classmethod _build_dom(*content: bytes | str, mode: str, encoding: str*) → lxml.etree._Element

build_html_tree() → lxml.etree._Element

build_xml_tree() → lxml.etree._Element

choose_form(*number: None | int = None*, *xpath: None | str = None*, *name: None | str = None*, ***kwargs: Any*) → None

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:

```
# 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")]')
```

get_cached_form() → lxml.html.FormElement

Get form which has been already selected.

Returns None if form has not been selected yet.

It is for testing mainly. To not trigger pylint warnings about accessing protected element.

set_input(*name: str*, *value: Any*) → None

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:

```
g.set_input('sex', 'male')

# Check the checkbox
g.set_input('accept', True)
```

set_input_by_id(*_id: str*, *value: Any*) → None

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: int*, *value: Any*) → None

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: str*, *value: Any*) → None

Set the value of form element by xpath.

Parameters

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

process_extra_post(*post_items: list[tuple[str, Any]]*, *extra_post_items: collections.abc.Sequence[tuple[str, Any]]*) → list[tuple[str, Any]]

clean_submit_controls(*post: collections.abc.MutableMapping[str, Any]*, *submit_name: None | str*) → None

get_form_request(*submit_name: None | str = None*, *url: None | str = None*, *extra_post: None | collections.abc.Mapping[str, Any] | collections.abc.Sequence[tuple[str, Any]] = None*, *remove_from_post: None | collections.abc.Sequence[str] = None*) → *FormRequestParams*

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.

build_fields_to_remove(*fields: collections.abc.Mapping[str, Any]*, *form_inputs: collections.abc.Sequence[lxml.html.HtmlElement]*) → set[str]

process_form_fields(*fields: collections.abc.MutableMapping[str, Any]*) → None

form_fields() → collections.abc.MutableMapping[str, lxml.html.HtmlElement]

Return fields of default form.

Fill some fields with reasonable values.

choose_form_by_element(*xpath: str*) → None

grab.errors

Custom exception which Grab instance could generate.

Exceptions

- **GrabError**
  - **GrabNetworkError**
    - GrabTimeoutError
    - GrabConnectionError
    - GrabCouldNotResolveHostError
  - GrabAuthError
  - GrabMisuseError
  - GrabTooManyRedirectsError
  - GrabInvalidUrl
  - GrabInternalError
  - GrabFeatureIsDeprecated
  - ResponseNotValid
- DataNotFound == IndexError

Warnings

- GrabDeprecationWarning

Module Contents

Functions

raise_feature_is_deprecated(→ None)

Attributes

DataNotFound

exception grab.errors.**GrabError**

Bases: Exception

All custom Grab exception should be children of that class.

exception grab.errors.**OriginalExceptionGrabError**(**args: Any*, ***kwargs: Any*)

Bases: *GrabError*

Sub-class which constructor accepts original exception as second argument.

exception grab.errors.**GrabNetworkError**(**args: Any*, ***kwargs: Any*)

Bases: *OriginalExceptionGrabError*

Raises in case of network error.

exception grab.errors.**GrabTimeoutError**(**args: Any*, ***kwargs: Any*)

Bases: *GrabNetworkError*

Raises when configured time is outed for the request.

exception grab.errors.**GrabConnectionError**(**args: Any*, ***kwargs: Any*)

Bases: *GrabNetworkError*

Raised when it is not possible to establish network connection.

exception grab.errors.**GrabCouldNotResolveHostError**(**args: Any*, ***kwargs: Any*)

Bases: *GrabNetworkError*

Raised when couldn't resolve host. The given remote host was not resolved.

exception grab.errors.**GrabAuthError**

Bases: *GrabError*

Raised when remote server denies authentication credentials.

exception grab.errors.**GrabMisuseError**

Bases: *GrabError*

Indicates incorrect usage of grab API.

exception grab.errors.**GrabTooManyRedirectsError**

Bases: *GrabError*

Raised when max. allowed number of redirects is reaced.

exception grab.errors.**GrabInvalidUrlError**

Bases: *GrabError*

Raised when error occurred while normalizing URL e.g. IDN processing.

exception grab.errors.**GrabInvalidResponseError**(**args: Any*, ***kwargs: Any*)

Bases: *OriginalExceptionGrabError*

Raised when network response's data could not be processed.

exception grab.errors.**GrabInternalError**(**args: Any*, ***kwargs: Any*)

Bases: *OriginalExceptionGrabError*

Sub-class which constructor accepts original exception as second argument.

exception grab.errors.**GrabFeatureIsDeprecatedError**

Bases: *GrabError*

Raised when user tries to use feature that is deprecated and has been dropped.

grab.errors.**raise_feature_is_deprecated**(*feature_name: str*) → None

grab.errors.**DataNotFound**

exception grab.errors.**ResponseNotValidError**

Bases: *GrabError*

All custom Grab exception should be children of that class.

exception grab.errors.**GrabDeprecationWarning**

Bases: UserWarning

Base class for warnings generated by user code.

grab.extensions

Module Contents

Classes

RedirectExtension	Abstract base class for generic types.
CookiesStore	
CookiesExtension	Abstract base class for generic types.

class grab.extensions.**RedirectExtension**

Bases: *grab.base.BaseExtension*[*grab.request.HttpRequest*, *grab.document.Document*]

Abstract base class for generic types.

A generic type is typically declared by inheriting from this class parameterized with one or more type variables. For example, a generic mapping type might be defined as:

```
class Mapping(Generic[KT, VT]):
    def __getitem__(self, key: KT) -> VT:
        ...
    # Etc.
```

This class can then be used as follows:

```
def lookup_name(mapping: Mapping[KT, VT], key: KT, default: VT) -> VT:
    try:
        return mapping[key]
    except KeyError:
        return default
```

__get__(*obj:* grab.base.BaseClient[grab.request.HttpRequest, grab.document.Document], *objtype: None* | *type*[grab.base.BaseClient[grab.request.HttpRequest, grab.document.Document]] = *None*) → *RedirectExtension*

find_redirect_url(*doc:* grab.document.Document) → None | str

process_init_retry(*retry: Any*) → None

reset() → None

process_retry(*retry: Any*, *req:* grab.request.HttpRequest, *resp:* grab.document.Document) → tuple[None, None] | tuple[Any, *grab.request.HttpRequest*]

class grab.extensions.**CookiesStore**(*cookiejar: None* | *http.cookiejar.CookieJar* = *None*)

__slots__ = ('cookiejar', 'ext_handlers')

process_request_pre(*req:* grab.request.HttpRequest) → None

process_response_post(*req:* grab.request.HttpRequest, *doc:* grab.document.Document) → None

reset() → None

set_cookie(*cookie: http.cookiejar.Cookie*) → None

clear() → None

Clear all remembered cookies.

clone() → *CookiesStore*

update(*cookies: collections.abc.Mapping[str, Any]*, *request_url: str*) → None

__getstate__() → collections.abc.MutableMapping[str, Any]

__setstate__(*state: collections.abc.Mapping[str, Any]*) → None

class grab.extensions.**CookiesExtension**

Bases: *grab.base.BaseExtension*[*grab.request.HttpRequest*, *grab.document.Document*]

Abstract base class for generic types.

A generic type is typically declared by inheriting from this class parameterized with one or more type variables. For example, a generic mapping type might be defined as:

```
class Mapping(Generic[KT, VT]):
    def __getitem__(self, key: KT) -> VT:
        ...
    # Etc.
```

This class can then be used as follows:

```
def lookup_name(mapping: Mapping[KT, VT], key: KT, default: VT) -> VT:
    try:
        return mapping[key]
    except KeyError:
        return default
```

__slots__ = ()

owner_store_reg:
collections.abc.MutableMapping[*grab.base.BaseClient*[*grab.request.HttpRequest*, *grab.document.Document*], *CookiesStore*]

__get__(*obj:* grab.base.BaseClient[grab.request.HttpRequest, grab.document.Document], *objtype: None* | *type*[grab.base.BaseClient[grab.request.HttpRequest, grab.document.Document]] = *None*) → *CookiesStore*

reset() → None

grab.grab

Module Contents

Classes

Grab	Abstract base class for generic types.

class grab.grab.**Grab**(*transport: None* | BaseTransport[*RequestT, ResponseT*] | *type*[BaseTransport[*RequestT, ResponseT*]] = *None*)

Bases: *grab.client.HttpClient*

Abstract base class for generic types.

A generic type is typically declared by inheriting from this class parameterized with one or more type variables. For example, a generic mapping type might be defined as:

```
class Mapping(Generic[KT, VT]):
    def __getitem__(self, key: KT) -> VT:
        ...
    # Etc.
```

This class can then be used as follows:

```
def lookup_name(mapping: Mapping[KT, VT], key: KT, default: VT) -> VT:
    try:
        return mapping[key]
    except KeyError:
        return default
```

cookies

grab.request

Module Contents

Classes

HttpRequest	

class grab.request.**HttpRequest**(*url: str*, *, *method: None | str = None*, *headers: None | collections.abc.MutableMapping[str, Any] = None*, *timeout: None | int | grab.util.timeout.Timeout = None*, *cookies: None | dict[str, Any] = None*, *encoding: None | str = None*, *proxy_type: None | str = None*, *proxy: None | str = None*, *proxy_userpwd: None | str = None*, *fields: Any = None*, *body: None | bytes = None*, *multipart: None | bool = None*, *document_type: None | str = None*, *redirect_limit: None | int = None*, *process_redirect: None | bool = None*, *meta: None | collections.abc.Mapping[str, Any] = None*)

Bases: *grab.base.BaseRequest*

init_keys

get_full_url() → str

_process_timeout_param(*value: None | float | grab.util.timeout.Timeout*) → *grab.util.timeout.Timeout*

compile_request_data() → CompiledRequestData

grab.response

grab.transport

Module Contents

Classes

Urllib3Transport	Grab network transport based on urllib3 library.

Attributes

LOG

grab.transport.**LOG**

class grab.transport.**Urllib3Transport**

Bases: *grab.base.BaseTransport*[*grab.request.HttpRequest*, *grab.document.Document*]

Grab network transport based on urllib3 library.

__getstate__() → dict[str, Any]

__setstate__(*state: collections.abc.Mapping[str, Any]*) → None

build_pool() → urllib3.PoolManager

reset() → None

wrap_transport_error() → collections.abc.Generator[None, None, None]

select_pool_for_request(*req:* grab.request.HttpRequest) → urllib3.PoolManager | urllib3.ProxyManager | urllib3.contrib.socks.SOCKSProxyManager

log_request(*req:* grab.request.HttpRequest) → None

Log request details via logging system.

request(*req:* grab.request.HttpRequest) → None

read_with_timeout(*req:* grab.request.HttpRequest) → bytes

get_response_header_items() → list[tuple[str, Any]]

Return current response headers as items.

This funciton is required to isolated smalles part of untyped code and hide it from mypy

prepare_response(*req:* grab.request.HttpRequest, *, *document_class: type[*grab.document.Document*] = Document*) → *grab.document.Document*

Prepare response, duh.

This methed is called after network request is completed hence the "self._request" is not None.

Good to know: on python3 urllib3 headers are converted to str type using latin encoding.

Package Contents

Classes

HttpClient	Abstract base class for generic types.
Document	Network response.
Grab	Abstract base class for generic types.
HttpRequest	

Functions

request(→ grab.document.Document)	

Attributes

DataNotFound

class grab.**HttpClient**(*transport: None* | BaseTransport*[RequestT, ResponseT]* | *type[*BaseTransport*[RequestT, ResponseT]] = None*)

Bases: *grab.base.BaseClient*[*grab.request.HttpRequest*, *grab.document.Document*]

Abstract base class for generic types.

A generic type is typically declared by inheriting from this class parameterized with one or more type variables. For example, a generic mapping type might be defined as:

```
class Mapping(Generic[KT, VT]):
    def __getitem__(self, key: KT) -> VT:
        ...
    # Etc.
```

This class can then be used as follows:

```
def lookup_name(mapping: Mapping[KT, VT], key: KT, default: VT) -> VT:
    try:
        return mapping[key]
    except KeyError:
        return default
```

document_class: type[*grab.document.Document*]

extension

request_class

default_transport_class

request(*req: None | str | grab.request.HttpRequest = None*, ***request_kwargs: Any*) → *grab.document.Document*

process_request_result(*req:* grab.request.HttpRequest) → *grab.document.Document*

Process result of real request performed via transport extension.

grab.**request**(*url: None | str | grab.request.HttpRequest = None*, *client: None | HttpClient | type[HttpClient] = None*, ***request_kwargs: Any*) → *grab.document.Document*

class grab.**Document**(*body: bytes*, ***, *document_type: None | str = 'html'*, *head: None | bytes = None*, *headers: None | email.message.Message = None*, *encoding: None | str = None*, *code: None | int = None*, *url: None | str = None*, *cookies: None | collections.abc.Sequence[http.cookiejar.Cookie] = None*)

Bases: grab.base.BaseResponse

Network response.

property status: None | int

property json: Any

Return response body deserialized into JSON object.

property pyquery: Any

Return pyquery handler.

property body: bytes

property tree: lxml.etree._Element

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

property form: lxml.html.FormElement

Return default document's 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:

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

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

__slots__ = ('document_type', 'code', 'head', 'headers', 'url', 'cookies', 'encoding', '_bytes_body',...

__call__(*query: str*) → selection.SelectorList[lxml.etree._Element]

select(**args: Any*, ***kwargs: Any*) → selection.SelectorList[lxml.etree._Element]

process_encoding(*encoding: None | str = None*) → str

Process explicitly defined encoding or auto-detect it.

If encoding is explicitly defined, ensure it is a valid encoding the python can deal with. If encoding is not specified, auto-detect it.

Raises unicodec.InvalidEncodingName if explicitly set encoding is invalid.

copy() → *Document*

save(*path: str*) → None

Save response body to file.

url_details() → urllib.parse.SplitResult

Return result of urlsplit function applied to response url.

query_param(*key: str*) → str

Return value of parameter in query string.

browse() → None

Save response in temporary file and open it in GUI browser.

__getstate__() → collections.abc.Mapping[str, Any]

Reset cached lxml objects which could not be pickled.

__setstate__(*state: collections.abc.Mapping[str, Any]*) → None

text_search(*anchor: str | bytes*) → bool

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.

text_assert(*anchor: str | bytes*) → None

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

text_assert_any(*anchors: collections.abc.Sequence[str | bytes]*) → None

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

rex_text(*regexp: str | bytes | re.Pattern[str] | re.Pattern[bytes]*, *flags: int = 0*, *default: Any = UNDEFINED*) → Any

Return content of first matching group of regexp found in response body.

rex_search(*regexp: str | bytes | re.Pattern[str] | re.Pattern[bytes]*, *flags: int = 0*, *default: Any = UNDEFINED*) → Any

Search the regular expression in response body.

Return found match object or None

rex_assert(*rex: str | bytes | re.Pattern[str] | re.Pattern[bytes]*) → None

Raise *DataNotFound* exception if *rex* expression is not found.

get_body_chunk() → bytes

unicode_body() → str

Return response body as unicode string.

classmethod wrap_io(*inp: bytes | str*) → io.StringIO | io.BytesIO

classmethod _build_dom(*content: bytes | str*, *mode: str*, *encoding: str*) → lxml.etree._Element

build_html_tree() → lxml.etree._Element

build_xml_tree() → lxml.etree._Element

choose_form(*number: None | int = None*, *xpath: None | str = None*, *name: None | str = None*, ***kwargs: Any*) → None

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:

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

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

(continues on next page)

(continued from previous page)

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

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

get_cached_form() → lxml.html.FormElement

Get form which has been already selected.

Returns None if form has not been selected yet.

It is for testing mainly. To not trigger pylint warnings about accessing protected element.

set_input(*name: str*, *value: Any*) → None

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:

```
g.set_input('sex', 'male')

# Check the checkbox
g.set_input('accept', True)
```

set_input_by_id(*_id: str*, *value: Any*) → None

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: int*, *value: Any*) → None

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: str*, *value: Any*) → None

Set the value of form element by xpath.

Parameters

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

process_extra_post(*post_items: list[tuple[str, Any]]*, *extra_post_items: collections.abc.Sequence[tuple[str, Any]]*) → list[tuple[str, Any]]

clean_submit_controls(*post: collections.abc.MutableMapping[str, Any]*, *submit_name: None | str*) → None

get_form_request(*submit_name: None | str = None*, *url: None | str = None*, *extra_post: None | collections.abc.Mapping[str, Any] | collections.abc.Sequence[tuple[str, Any]] = None*, *remove_from_post: None | collections.abc.Sequence[str] = None*) → *FormRequestParams*

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.

build_fields_to_remove(*fields: collections.abc.Mapping[str, Any]*, *form_inputs: collections.abc.Sequence[lxml.html.HtmlElement]*) → set[str]

process_form_fields(*fields: collections.abc.MutableMapping[str, Any]*) → None

form_fields() → collections.abc.MutableMapping[str, lxml.html.HtmlElement]

Return fields of default form.

Fill some fields with reasonable values.

choose_form_by_element(*xpath: str*) → None

grab.**DataNotFound**

exception grab.**GrabError**

Bases: Exception

All custom Grab exception should be children of that class.

exception grab.**GrabMisuseError**

Bases: *GrabError*

Indicates incorrect usage of grab API.

exception grab.**GrabNetworkError**(**args: Any*, ***kwargs: Any*)

Bases: OriginalExceptionGrabError

Raises in case of network error.

exception grab.**GrabTimeoutError**(**args: Any*, ***kwargs: Any*)

Bases: *GrabNetworkError*

Raises when configured time is outed for the request.

class grab.**Grab**(*transport: None | BaseTransport[RequestT, ResponseT] | type[BaseTransport[RequestT, ResponseT]] = None*)

Bases: *grab.client.HttpClient*

Abstract base class for generic types.

A generic type is typically declared by inheriting from this class parameterized with one or more type variables. For example, a generic mapping type might be defined as:

```
class Mapping(Generic[KT, VT]):
    def __getitem__(self, key: KT) -> VT:
        ...
    # Etc.
```

This class can then be used as follows:

```
def lookup_name(mapping: Mapping[KT, VT], key: KT, default: VT) -> VT:
    try:
        return mapping[key]
    except KeyError:
        return default
```

cookies

class grab.**HttpRequest**(*url: str*, ***, *method: None | str = None*, *headers: None | collections.abc.MutableMapping[str, Any] = None*, *timeout: None | int | grab.util.timeout.Timeout = None*, *cookies: None | dict[str, Any] = None*, *encoding: None | str = None*, *proxy_type: None | str = None*, *proxy: None | str = None*, *proxy_userpwd: None | str = None*, *fields: Any = None*, *body: None | bytes = None*, *multipart: None | bool = None*, *document_type: None | str = None*, *redirect_limit: None | int = None*, *process_redirect: None | bool = None*, *meta: None | collections.abc.Mapping[str, Any] = None*)

Bases: *grab.base.BaseRequest*

init_keys

get_full_url() → str

_process_timeout_param(*value: None | float | grab.util.timeout.Timeout*) → *grab.util.timeout.Timeout*

compile_request_data() → CompiledRequestData

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

g

INDEX

Symbols

A

B

C

D

E

F

G

H

I

J

Q

R

S

T

U

V

W